A Static 2-Approximation Algorithm for Vertex Connectivity and Imcremental Approximation Algorithms for Edge and Vertex Connectivity

Monika Rauch Henzinger*

Digital Systems Research Center,130 Lytton Avenue Palo Alto, CA 94301

Abstract. This paper presents insertions-only algorithms for maintaining the exact and/or approximate size of the minimum edge cut and the minimum vertex cut of a graph. The algorithms output the approximate or exact size k in time $O(1)$ and a cut of size k in time linear in its size. For the minimum edge cut problem and for any $0 < \epsilon \leq 1$, the amortized time per insertion is $O(1/\epsilon^2)$ for a $(2+\epsilon)$-approximation, $O((\log \lambda)((\log n)/\epsilon)^2)$ for a $(1+\epsilon)$-approximation, and $O(\lambda \log n)$ for the exact size, where n is the number of nodes in the graph and λ is the size of the minimum cut. The $(2+\epsilon)$-approximation algorithm and the exact algorithm are deterministic, the $(1+\epsilon)$-approximation algorithm is randomized.

We also present a static 2-approximation algorithm for the size κ of the minimum vertex cut in a graph, which takes time $O(n^2 \min(\sqrt{n}, \kappa))$. This is a factor of κ faster than the best algorithm for computing the exact size, which takes time $O((\kappa^3 n + \kappa n^2) \min(\sqrt{n}, \kappa))$. We give an insertions-only algorithm for maintaining a $(2+\epsilon)$-approximation of the minimum vertex cut with amortized insertion time $O(n/\epsilon)$.

1 Introduction

Computing the connectivity of a graph is a fundamental problem that has achieved a lot of attention (see for example [1, 2, 6, 8, 9, 10, 11, 15, 16, 18, 20]). In this paper we study the problem of maintaining the connectivity of the graph during modifications of the graph.

Let $G = (V, E)$ be an undirected, unweighted multigraph. Two vertices x and y of G are *k-edge connected* if there exist k pairwise edge-disjoint paths connecting x and y. A graph G is *k-edge connected* if every pair of vertices is k-edge connected.

Let $\lambda(G, x, y)$ be the maximum k such that x and y are k-edge connected in G and let $\lambda(G)$ be the maximum k such that G is k-edge connected. An *edge cut* of G is a set of edges in G whose removal disconnects G. An edge cut C *separates* x and y if x and y belong to different connected components of $G \setminus C$. A *minimum edge cut* is a cut of minimum size. By Menger's theorem [19], (a)

* Maiden Name: Monika H. Rauch. This research was supported by an NSF CAREER Award.

the size of the minimum edge cut is $\lambda(G)$, and (b) the size of the minimum edge cut separating x and y is $\lambda(G, x, y)$.

Let $G = (V, E)$ be an undirected, unweighted graph. We say two paths in G are *openly disjoint* if they are vertex-disjoint except for their endpoints. Two vertices of G are *k-vertex connected* if there exist k pairwise openly disjoint paths connecting x and y. A graph G is *k-vertex connected* if $|V| > k$ and every pair of vertices is k-vertex connected.

Let $\kappa(G, x, y)$ be the maximum k such that x and y are k-vertex connected in G and let $\kappa(G)$ to be the maximum number k such that G is k-vertex connected. [2] A *vertex cut* of G is a set C of vertices in G such that $G \setminus C$ is disconnected and non-trivial [3]. A vertex cut C *separates* x and y if x and y belong to different connected components of $G \setminus C$. A *minimum vertex cut* is a vertex cut of minimum size. By Menger's theorem [19], (a) the size of the minimum vertex cut is $\kappa(G)$ if $\kappa(G) < |V| - 1$, and (b) the size of the minimum vertex cut separating two non-adjacent vertices x and y is $\kappa(G, x, y)$. Note that k-vertex connectivity implies k-edge connectivity, but not vice versa.

Given an initial graph G with n nodes and m_0 edges a *fully dynamic* algorithm maintains G during an arbitrary sequence of the following operations.

- *Insert*(u, v): Insert the edge (u, v) into G.
- *Delete*(u, v): Delete the edge (u, v) from G.
- *Query-Size*: Return the exact (approximate) size of a minimum cut in G.
- *Query-Cut*: Return an exact (approximate) minimum cut of G.

An *incremental* algorithm maintains the graph under an arbitary sequence of *Insert*, *Query-Size*, and *Query-Cut* operations.

We say that 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 incremental algorithms for maintaining the exact and approximate size of the minimum edge cut and the approximate size of the minimum vertex cut. 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, otherwise, a new solution is computed using the static algorithm. The difficulty lies in finding an appropriate data structure and to amortize the cost for the static algorithm over previous insertions. The algorithms can output the exact or approximate size of the minimum cut in time $O(1)$ and a minimum or approximate minimum cut in time linear in its size.

We denote by m the total number of edges in the graph, consisting of m_0 initial edges and m_1 inserted edges, by δ the minimum degree in the final graph, by λ the size of the minimum edge cut in the final graph, and by κ the size of

[2] Our definition of $\kappa(G, x, y)$ for two *adjacent* nodes x and y differs from the literature: Usually $\kappa(G, x, y)$ for two adjacent nodes x and y is defined to be $n - 1$, while we allow $\kappa(G, x, y)$ to be less than $n - 1$. Our definition is introduced for notational convenience and does not affect the definition of $\kappa(G)$.

[3] A graph is *trivial* if is consists of 1 vertex.

the minimum vertex cut in the final graph. Note that $\kappa \leq \lambda \leq \delta$ and $m_0 + m_1 = \Omega(\delta n)$, i.e. $\delta n = O(m_0 + m_1)$.

The incremental algorithms presented in this paper are almost optimal, (up to a factor of $\log n / \log(n/\lambda)$, resp. $1/\epsilon$, resp. $\log(\lambda/\epsilon)$) in the following sense: A faster incremental algorithm would lead to an improvement in the running time of the best static algorithm.

For example, we give an exact minimum edge cut algorithm with amortized time $O(\lambda \log n)$ per operation. Any incremental algorithm with amortized time $t(n, m) = o(\lambda \log(n/\lambda))$ per insertion leads to a static minimum cut algorithm with time $O(m + \lambda n t(n, m))$, which would improve the current best deterministic bound of $O(m + \lambda^2 n \log(n/\lambda))$ [9]: The static algorithm is created from the incremental algorithm by (1) computing a 3-approximation of the minimum edge cut, (2) computing a subgraph of the graph with the same minimum cut but at most $3\lambda n$ edge, and (3) adding all edges of this subgraph to an initially empty graph. Step (1) takes time $O(m)$ using Matula's 3-approximation [18], Step (2) takes time $O(m + n)$ using the algorithm of Nagamochi and Ibaraki [20], and Step (3) takes time $O(\lambda n t(n, m))$ using the incremental algorithm.

Minimum Edge Cuts. We give incremental algorithms that maintain the exact size, $(1 + \epsilon)$-, or $(2 + \epsilon)$-approximate size of the minimum edge cut. The algorithms for minimum edge cuts apply to any multigraph $G = (V, E)$.

The first algorithm maintains a $(2 + \epsilon)$-approximation of λ in total time $O((m_0 + \lambda n)/\epsilon + m_1/\epsilon^2)$ for any $1 \geq \epsilon > 0$. It is an incremental version of Matula's static $(2 + \epsilon)$-approximation algorithm [18], which takes time $O(m/\epsilon)$. If the initial graph is empty, our amortized time per operation is $O(1/\epsilon^2)$.

The second algorithm is an incremental version of Gabow's (exact) minimum edge cut algorithm [9], which computes the size of the minimum edge cut in time $O(m + \lambda^2 n \log(n^2/(m + 1))) = O(m + \lambda^2 n \log(n/(\lambda + 1)))$. Our incremental algorithm takes time $O(m_0 + m_1 + \lambda^2 n \log(n/(\lambda_0 + 1)))$, where λ_0 is the size of the minimum edge cut in the initial graph. If the initial graph is empty, the amortized time per insertion is $O(\lambda^2 (\log n) n / m_1) = O(\lambda \log n)$, since $m_1 = \Omega(\lambda n)$. Apart from answering *Query-Size* or *Query-Cut* operations, our algorithm can answer queries that ask if two given nodes are separated by a cut of size λ in amortized time $O(\alpha(n, n))$. If λ is a constant, our running time is close to the running time of the best "special purpose" algorithms: Determining if two nodes are connected, 2-edge-connected, or 3-edge-connected takes amortized time $O(\alpha(m, n))$ per insertion or query[22, 12, 17].

Gabow's algorithm implies a fully dynamic algorithm which allows insertions and deletions of edges in worst-case time $O(m \log(n/\lambda))$. Our algorithm can be modified to improve this bound to $O(\lambda_{max} \log n)$ amortized time per insertion and $O(\lambda n \log(n/\lambda))$ amortized time per deletion, where λ is the size of the minimum cut during the operation and λ_{max} is the maximum size of the minimum cut during the whole sequence of operations.

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 $O((m_0 + m_1)\log^2 n(\log\lambda)/\epsilon^2)$. Thus, if the initial graph is empty, the amortized expected time per insertion is $O(\log^2 n(\log\lambda)/\epsilon^2)$. This technique was introduced by Karger [16] in his static algorithm, which needs time $O(m + n((\log n)/\epsilon)^3)$ to compute a $(1+\epsilon)$-approximation of λ.

Related results: (1) For any fixed k, Dinitz and Westbrook [4] give an algorithm that supports *Same-k-Component?* queries and edge insertions in a (k-1)-edge-connected graph. It takes time $O(q+n)$ with a preprocessing time of $O(m_0 + k^2 n\log(n/k))$, where q is the total number of operations. (2) For any fixed k, Eppstein et. al [5] give a fully dynamic algorithm that tests if the whole graph is k-edge-connected in $O(1)$ time. The times per edge insertion or deletion is $O(k^2 n\log(n/k))$. (3) Karger [15] gives a randomized algorithm, which maintains a $\sqrt{1+2/\epsilon}$-approximation of the minimum edge cut in expected time $\tilde{O}(n^\epsilon)$ per insertion. Thus, for a $(2+\epsilon)$-approximation it takes time $\tilde{O}(n^{2/(3+4\epsilon+\epsilon^2)})$ per insertion and for a $(1+\epsilon)$-approximation it takes time $\tilde{O}(n^{1/\epsilon})$. Our algorithms achieve exponential improvements. He also gives a fully dynamic algorithm that maintains a $\sqrt{1+2/\epsilon}$-approximation algorithm in $\tilde{O}(n^{1/2+\epsilon})$ time per edge insertions and deletions. (4) Recently, Dinitz and Nutov [3] gave an algorithm that maintains all cuts of size λ or $\lambda+1$. The total time for m_1 edge insertions and q *Same-*$(\lambda+2)$*-Component?* queries is $O((n+m_1)\alpha(m_1,n) + q\alpha(q,n))$ for odd λ and $O(m_1 + n\log n + q\alpha(q,n))$ for even λ.

Minimum Vertex Cuts. We present a static algorithm that computes a 2-approximation of the size κ of the minimum vertex cut, i.e. it computes the exact size of κ if $\kappa \le \lfloor\delta/2\rfloor$ and it returns $\lfloor\delta/2\rfloor$ if $\kappa > \lfloor\delta/2\rfloor$. It takes time $O(n^2\min(\sqrt{n},\kappa))$. This is a speed-up of a factor of at least κ over the fastest exact algorithm for computing κ, which takes time $O((\kappa^3 n + \kappa n^2)\min(\sqrt{n},\kappa))$ [1, 11, 20]. Using this 2-approximation algorithm as subroutine gives an incremental algorithm with total time $O(m_0 + m_1 n + \kappa n^2/\epsilon)$. If the initial graph is empty, the amortized time per insertion is $O(n/\epsilon)$.

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 we use the following generic algorithm.

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 add the new edge to the graph.
   endwhile
3. Compute a new solution using previous solutions.
   Goto 2.

The algorithm decomposes the sequence of insertions into subsequences, between which a new solution is computed in Step 3. The difficulty lies in deciding (1) how to quickly test if the current solution is still correct and (2) how to efficiently compute a new solution using previous solutions. To analyze the running time we amortize the cost of computing a new solution over the sequence of insertions since the last computation of a solution. We restrict our description to answering *Query-Size* operations. However, it is straightforward to extend the algorithms to answer *Query-Cut* operations.

3 Basic Definitions

Let $G = (V, E)$ be an undirected graph. 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 $G \setminus (F_1 \cup F_2 \cup \ldots \cup F_{i-1})$. If two nodes are connected in F_i, they are i-edge connected. Nagamochi and Ibaraki [20] give a linear time algorithm (referred to as *decomposition algorithm (DA)*) that computes a special msfd, called *DA-msfd*, of order m in time $O(m+n)$. A DA-msfd fulfills the following additional conditions [20]: If G is a graph then, (1) G is k-vertex connected iff $F_1 \cup \ldots \cup F_k$ is k-vertex connected. If G is a multigraph then, (2) for all $1 \leq i \leq k$ if x and y are connected in F_i, then x and y are i-edge connected in G, (3) G is k-edge connected iff $F_1 \cup \ldots \cup F_k$ is k-edge connected, and (4) for any i and $x, y \in V$, $\lambda(\cup_{j \leq i} F_j, x, y) \geq \min(\lambda(G, x, y), i)$.

The decomposition algorithm also determines a 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 (but not parallel edges) are discarded. A contraction reduces the number of nodes in G, but does not reduce the size of the minimum edge cut. We *contract* a forest F if we contract all edges of G that are in F.

We use the following fact repeatedly: If a graph is k-edge connected, it contains $\Omega(kn)$ edges:

Lemma 1. [14] *If a n-node (multi)graph is k-edge connected, then it contains at least $kn/2$ edges.*

4 Incremental Algorithms for the Minimum Edge Cut

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

Using the generic algorithm of Section 2 we create an incremental algorithm that maintains a $(2+\epsilon)$-approximation of the minimum cut. The results in this section hold for any multigraph $G = (V, E)$. We describe below (1) how to test the correctness of the current solution after an insertion and (2) how to efficiently compute a new solution. Let ϵ' be $\epsilon/2$.

(1) Let k be the current solution, i.e. $k/(2+\epsilon) \leq \lambda \leq k$. At the start of each subsequence of insertions we contract G in Procedure *Contract* such that in the

resulting n'-node graph the number m' of edges in the contracted graph is at most $kn'/(2+\epsilon'/2)$. Lemma 1 shows that $\lambda \leq k$ as long as $m' < (k+1)n'/2$. Thus to test the correctness of the current solution k after an insertion we simply check if $m' < (k+1)n'/2$.

(2) To efficiently compute a new solution, we first test if we can find a cut of size $< k$ in the graph. If not, we repeatedly increase k until we find a cut of size $< k$ in the graph. This proves that k is an upper bound on λ. To find a cut of size $< k$ we compute a DA-msfd, contracting the forest F_p of the DA-msfd with $p = \lceil k/(2+\epsilon) \rceil$, and checking if in the resulting n'-node graph (a) $n' > 1$ and (b) the number of edges $m' \leq kn'/(2+\epsilon'/2)$. This implies that the contracted graph contains a node with degree $< k$. The set of vertices of G contracted to this node defines a cut of size $< k$ in G.

To improve the efficiency of the algorithm, every DA-msfd is computed on a "sparse" graph that has a minimum cut of size $\leq k$ iff the original graph has a minimum cut of size $\leq k$: we use the graph consisting of the union of the forests $F_1, \ldots, F_k$ of the most recent DA-msfd and all newly inserted edges. This idea was developped independently also by Dinitz and Westbrook [4]. The algorithm is 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. $\epsilon' = \epsilon/4$, $N = \emptyset$.
   Compute a $(2+\epsilon')$-approximation k of λ using the static algorithm.
   $p = \lceil k/(2+\epsilon') \rceil$
   Call *Contract*(G', k, p).
2. **while** $m' < (k+1)n'/2$ **do**
   if the new operation is a query **then return** k
   else add the inserted edge to N and to G'.
   endwhile
3. *Contract*(G', k, p).
   while $n' = 1$ **do**
   $k = k(1+\epsilon')$, $p = \lceil k/(2+\epsilon') \rceil$, $V' = V$, $E' = N \cup \cup_{q \leq k} F_q$, $N = \emptyset$.
   Call *Contract*(G', k, p).
   endwhile
   Goto 2.

Contract(G', k, p)
repeat
Compute a DA-msfd $\tilde{F}_1, \ldots, \tilde{F}_{m'}$ of G' of order m'.
Contract all edges in the forest $\tilde{F}_p$ and discard all self-loops.
until $m' \leq kn'/(2+\epsilon'/2)$

Correctness

Lemma 2. *Let $G = (V, E)$ be a multigraph with $N \subseteq E$, let $F_1, \ldots, F_m$ be a*

DA-msfd of $G \setminus N$, and let $G' = (V, E')$ be a multigraph with $E' = N \cup \cup_{q \le k} F_q$ for an integer k. If a cut in G' has size $s \le k$, it has size s in G as well.

Proof: By property (1) of a DA-msfd, the multigraph G' can be constructed from G by removing edges whose endpoints are k-edge connected in $\cup_{q \le k} F_q$, and, thus, in G'. Thus, for a cut of size at most k, its size is not decreased during the construction of G'. This implies that if a cut has size $s \le k$ in G', it has size s in G as well. ∎

Lemma 3. *The algorithm returns a $(2+\epsilon)$-approximation of λ for $0 < \epsilon \le 4$.*

Proof: We show by induction that $k/(2+\epsilon) \le \lambda \le k$. Initially k is a $(2+\epsilon)$-approximation and, thus, $k/(2+\epsilon) \le \lambda \le k$. Assume inductively that the claim holds and consider the next insertion. Note that insertions only increase λ. We distinguish two cases:

(1) If $m' < (k+1)n'/2$ after the insertion, then Lemma 1 shows that $\lambda \le k$ in G' also after the insertion. Thus, there exists a cut in G' of size at most k, which implies by Lemma 2 that there exists a cut in G with size at most k. By induction $k/(2+\epsilon) \le \lambda$ holds.

(2) If $m' = (k+1)n'/2$ after the insertion, then the algorithm repeatedly contracts G' and multiplies k by $(1+\epsilon)$ until it finds the smallest k such that the contractions stop with $n' > 1$ and $m' \le kn'/(2+\epsilon/2)$. This implies that there exists a node in G' with degree at most $2(k/(2+\epsilon/2)) < k$. By Lemma 2 it follows that $\lambda < k$ in G. If k is unchanged, it follows by induction that $k/(2+\epsilon) \le \lambda$. Otherwise, k is the smallest value for which the contractions terminate with $n' > 1$. Thus, contracting $F_{p'}$ for $p' = \lceil k/((2+\epsilon')(1+\epsilon')) \rceil$ contracted G' to a single node. This implies that all nodes in G' are connected in $F_{p'}$ and are p'-edge connected in G', by property (1) of a DA-msfd. Since $G' \subseteq G$, all nodes in G are p'-edge connected. Thus $\lambda \ge p' = \lceil k/((2+\epsilon')(1+\epsilon')) \rceil \ge k/(2+4\epsilon') = k/(2+\epsilon)$ for $\epsilon' \le 1$. ∎

Running Time Analysis

Lemma 4. *A call to Contract(G', k, p) takes time $O(m' + kn'/\epsilon)$, where n' is the number of nodes and m' is the number of edges in G'.*

Proof: The first iteration takes time $O(m')$. Afterwards G' has $O(kn')$ edges, since all edges in F_q for $q \ge p$ are discarded during the removal of self-loops.. Every further iteration reduces the number of edges from at least $kn''/(2+\epsilon'/2)$ to at most $\lceil k/(2+\epsilon') \rceil n''$, where n'' is the number of nodes at the beginning of the iteration. This shows that the iteration reduces the number of edges by a factor of at least $(2+\epsilon'/2)/(2+\epsilon') = 1 - \Omega(\epsilon)$. Each iteration takes time linear in the number of edges. Thus, the cost of all further iterations gives a geometric series which sums to $O(kn'/\epsilon)$. ∎

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 u_i be the number of insertions during Phase i.

Lemma 5. *Phase i takes time $O(k_i n + u_i/\epsilon^2 + u_{i-1})$.*

Proof: Let $p = \lceil k_i/(2+\epsilon') \rceil$. There are at most $k_i(n-1) + u_{i-1}$ edges in $N \cup \cup_{q \leq k_i} F_q$. Computing a DA-msfd, contracting all edges of F_p, and the first call to *Contract* takes time $O(k_i n + u_{i-1})$. Let n' be the number of nodes after the first call to *Contract.* The resulting graph has $O(k_i n')$ edges, since all edges in F_q for $q \geq p$, are discarded during the removal of self-loops.
Now Step 2 and procedure *Contract* are executed in turns (possibly 0 times) until a contraction reduces the number of nodes to 1. We analyze the time of an execution of Step 2 followed by a call to *Contract.*

Let n'' be the number of nodes in the graph at the beginning of Step 2. Then the graph has $O(k_i n'')$ edges at the beginning of Step 2. Let d be the number of insertions during the execution of Step 2. The time for processing them is $O(d)$. The resulting graph has $O(k_i n'' + d)$ edges. Thus, the following call to *Contract* takes time $O(k_i n''/\epsilon + d)$. This shows that the time spent for the execution of Step 2 and the following call to *Contract* is $O(k_i n''/\epsilon + d)$.

Next we give a lower bound on d. Before Step 2 the graph contained at most $k_i n''/(2+\epsilon'/2)$ edges, afterwards at least $(k_i+1)n''/2$. Thus, $d \geq (k_i+1)n''/2 - k_i n''/(2+\epsilon'/2) = \Omega(k_i n'' \epsilon)$. Hence, the time spent for the d insertions and following call to *Contract* is $O(d/\epsilon^2)$. Since the sum of the number of insertions over all executions of Step 2 in Phase i is u_i, the total time for Phase i is $O(k_i n + u_{i-1} + u_i/\epsilon^2)$. ∎

Theorem 6. *Given a multigraph with n nodes and m_0 edges the total time for inserting m_1 edges and maintaining a $(2+\epsilon)$-approximation of the minimum cut is*

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

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 $O(\sum_i k_i n + u_i/\epsilon^2 + u_{i-1})$. Since $\sum_{i \leq q} k_i = O(\lambda/\epsilon)$ and since all u_i sum to m_1, the total time is $O(m_0/\epsilon + \sum_i (k_i n/\epsilon + u_i/\epsilon^2 + u_{i-1})) = O((m_0 + \lambda n)/\epsilon + m_1 \epsilon^2)$. ∎

4.2 An Incremental Exact Algorithm

In this section we present a deterministic incremental algorithm that maintains λ. The algorithm in this section applies to any multigraph $G = (V, E)$. The basic idea for testing efficiently if the current solution is still correct is to compute and store *all* minimum edge cuts when λ assumes a new value. 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 still exists 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 multigraph $G = (V, E)$ is a graph $G_c = (V_c, E_c)$ with a weight function w defined as follows: There is a mapping $\phi : V \rightarrow V_c$ such that

(1) every node in V maps to exactly one node in V_c and thus every node in 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 $\lambda/2$.

As observed by Dinitz and Westbrook [4], given a cactus-tree the data structure of [12, 17] can maintain the cactus tree for a fixed value of λ such that the total time for u insertions is $O(u + n)$. Determining if there exists a minimum cut whose size has not been increased takes constant time.

To quickly compute the cactus tree representation of a multigraph we use an algorithm by Gabow [10]. The algorithm computes first a subgraph of G, called *a complete λ-intersection* or $I(\lambda)$, with at most λn edges, and uses the complete λ-intersection to compute the cactus tree. The algorithm needs time $O(m_0 + \lambda^2 n \log(n^2/(m_0 + 1)))$ to compute $I(\lambda)$ and the cactus tree in the initial graph. Additionally given $I(\lambda)$ and a sequence of insertions that increase the minimum cut size by 1, the new $I(\lambda)$ and the new cactus tree can be computed in time $O(m' \log(n^2/m'))$, where m' is the current number of edges.

This leads to the following algorithm.

An Incremental Minimum Edge Cut Algorithm

1. Compute the size λ of the minimum cut, a DA-msfd $F_1, \ldots, F_m$ of order m, $I(\lambda)$, and a cactus-tree of $\cup_{i \le \lambda} F_i$.
2. $N = \emptyset$.

   while there is ≥ 1 minimum cut of size λ **do**

       if the next operation is a query **then return** λ

       else update the cactus tree according to the insertion of the new edge and add the edge to N.

   endwhile
3. $\lambda = \lambda + 1$.

   Compute a DA-msfd $F_1, \ldots, F_m$ of order m of $\cup_{i \le \lambda+1} F_i \cup N$ and call the forests $F_1, \ldots, F_m$.

   Let $G' = (V, E')$ be a graph with $E' = I(\lambda - 1) \cup \cup_{i \le \lambda+1} F_i$.

   Compute $I(\lambda)$ and a cactus tree of G'.

   Goto 2.

Correctness

Lemma 7. *Let $G = (V, E)$ be a multigraph with minimum cut λ and $N \subseteq E$, let $F_1, \ldots, F_m$ be a DA-msfd of G, and let $G' = (V, E')$ be a graph with $E' = N \cup \cup_{i \leq \lambda+1} F_i$. Then, a cut is a minimum cut in G' iff it is a minimum cut in G.*

Proof: Follows from Lemma 2. ∎

Running Time Analysis

Theorem 8. *Let G be a multigraph with n nodes, m_0 edges, and minimum cut size λ_0. The total time for inserting m_1 edges and maintaining a minimum edge cut of G is*

$$O(m_0 + m_1 + \lambda^2 n \log(n/(\lambda_0 + 1))),$$

where λ_0, resp. λ is the size of the minimum cut in the initial, resp. final graph. The size of the minimum cut can be output in constant time, a query if two given nodes are separated by a minimum cut can be answered in amortized constant time.

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

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 all step executed while $\lambda = \lambda_i$. Let u_i be the number of insertions in Phase i.

In Phase i, we compute a new msfd and build a new cactus tree in Step 3, and maintain the cactus tree in Step 2. The total time for Step 2 is $O((n + u_i))$. The total time for DA in Step 3 is $O(u_{i-1} + \lambda_i n)$. The graph G' has $O(\lambda_i n)$ edges. Thus, it takes time $O(\lambda_i n \log(n/(\lambda_0 + 1)))$ to compute $I(\lambda_i)$ and the new cactus tree.

The total time spent in Phase i is $O((n + u_i) + \lambda_i n \log(n/\lambda_i + 1)))$. Thus, the total work in all phases is $O(m_0 + m_1 + \lambda^2 n \log(n/(\lambda_0 + 1)))$. ∎

A Fully Dynamic Algorithm

Updating $I(\lambda)$ after an edge deletion requires the recomputation of one subgraph (to be precise, one spanning tree) of the complete λ intersection and takes time $O(m \log(n/\lambda))$ [9]. To obtain an insertions and deletions algorithm, we execute Step 3 (except for incrementing λ) after each deletion.

Theorem 9. *Let G be a multigraph with n nodes that is initially without edges. The exact size λ of a minimum edge cut can be maintained in amortized time $O(\lambda n \log(n/(\lambda + 1)))$ per deletion, where λ is the size of the minimum cut after the deletion, and $O(\lambda_{max} \log n)$ per insertion, where λ_{max} is the maximum value that λ assumes during the sequence of operations.*

Proof: Consider a sequence of u edge insertions and d edge deletions. We define Phase 0 to consist of Step 1 and the first execution of Step 2. We define *Phase* i for $i > 0$ to consist of the ith execution of Step 3 and the following execution of Step 2 (if it exists). A phase can end because of two reasons: (a) The last insertion increases the minimum cut by 1. (b) The last operation was a deletion. Let u_i be the number of insertions during phase i, and let λ be the current size of the minimum cut. As was shown in the proof of Theorem 8, the time spent in a phase is $O(u_i + u_{i-1} + \lambda n \log(n/(\lambda + 1))))$.

Each insertion is charged $O(1)$ to pay for its cost in the current and in the next phase. Thus, the amortized cost per phase is $O(\lambda n \log(n/(\lambda + 1)))$.

Each deletion decreases λ by at most 1. Consider an arbitrary deletion. Let λ' be the size of the minimum cut after the deletion. We charge two phases to the deletion: (1) the current phase, (2) the phase closest to the current phase, where the minimum cut size was λ'. The amortized cost of the two phases and, thus, of the deletion, is $O(\lambda' n \log(n/(\lambda')))$.

Now consider all remaining phases, i.e. all phases whose costs have not been charged to deletions. Let us call them $P_1, \ldots P_g$. Consider the phase P_i. Assume the minimum cut size of P_i is λ'. The minimum cut in P_j with $j > i$ is larger than λ', since otherwise there would have been a deletion decreasing the minimum cut size to λ' and paying for P_i. Thus, the amortized cost of all phases P_i is $O(\sum_{1 \leq \lambda' \leq \lambda_{max}} \lambda' n \log(n/(\lambda' + 1))) = O(\lambda_{max}^2 n \log(n/(\lambda_{max} + 1)))$. We amortize these costs over all insertions as in the insertions-only case. Thus, the amortized cost of an insertion is $O(\lambda_{max}^2 n \log(n/(\lambda_{max} + 1))/m_1)$. If the initial graph is empty, then $\lambda_{max} n = O(m_1)$, which gives an amortized insertion time of $O(\lambda_{max} \log(n/(\lambda_{max} + 1)))$. ∎

4.3 A Randomized $(1 + \epsilon)$-Approximation for the Minimum Cut

In this section we present an incremental Monte Carlo algorithm that maintains a $(1 + \epsilon)$-approximation of the minimum cut.

Karger [15] pointed out 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: Let $G(p)$ be a subgraph of G that is constructed by sampling each edge of G with probability p and adding it to $G(p)$ if sampling was successful. We build $G(p)$ incrementally by sampling each edge with probability p when it is inserted. The following lemma shows that the resulting incremental algorithm maintains a $(1 + \epsilon)$-approximation of the minimum cut with high probability.

Lemma 10. [15] *Let G be any graph with minimum cut λ and let $p = 2(d + 2)(\ln n)/(\epsilon^2 \lambda)$ for any $\epsilon \leq 1$.*

1. *With probability* $1-O(1/n^d)$ *the size of the minimum cut in* $G(p)$ *is* $\Theta((\log n)/\epsilon^2)$.
2. *With probability* $1-O(1/n^d)$, $(1-\epsilon)\lambda(G(p))/p \le \lambda \le (1+\epsilon)\lambda(G(p))/p$.

Set $\epsilon' = \epsilon/4$ and set $p = 2(d+2)(\ln n)/(\epsilon'^2\lambda)$. Lemma 10 shows that $(1-\epsilon')\lambda(G(p))/p$ is a $(1+\epsilon')/(1-\epsilon') \le 1+4\epsilon' = 1+\epsilon$ approximation for $\epsilon \le 2$.

If p is chosen to have 3 times the value in Lemma 10, then the probabilities in the lemma are increased and $G(p)$ correctly approximates $\lambda(G)$ until $\lambda(G)$ has increased by a factor of 3. Then the incremental exact algorithm has to be restarted. We test incrementally if $\lambda(G)$ has increased by a factor of 3 by maintaining a 3-approximation of $\lambda(G)$. 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.
   while $k \le 8(d+4)\ln n$ **do**
      Maintain the exact minimum cut of G incrementally.
   Initialize the incremental 3-approximation algorithm.
   $\epsilon' = \epsilon/4$, $p = 8(d+4)(\ln n)/(k\epsilon'^2)$.
   Construct $G(p)$ by sampling every edge with probability p. Start an incremental exact data structure for $G(p)$.
2. $N = \emptyset$.
   while $\lambda(G(p))$ is not increased **do**
      if the next operation is a query **then**
         return $\lambda(G(p))/p$
      else add the new edge e to N and sample it. If sampling is successful,
         insert e into the incremental exact data structure of $G(p)$.
   endwhile
3. Add all edges of N to the incremental 3-approximation algorithm to determine a new 3-approximation k'.
   if $k' > 3k$ **then** $k = k'$, $p = 8(d+4)(\ln n)/(k\epsilon'^2)$.
      Construct $G(p)$ by sampling every edge with probability p.
   Start an incremental exact data structure for $G(p)$.
   Goto 2.

The probability that the algorithm does not maintain a $(1+\epsilon)$-approximation is bounded by the sum of the probabilities that at any step the algorithm does not compute a $(1+\epsilon)$-approximation, which is $O(1/n^{d+2})$. Since there are $O(n^2)$ insertions, the probability of failure is $O(1/n^d)$. The same holds for the size of the minimum cut in $G(p)$.

Theorem 11. *Let G be a graph with n nodes and m_0 initial edges. For any $\epsilon \le 1$ and any $1 \le d \le \log\lambda/\epsilon^2$, the given algorithm maintains a $(1+\epsilon)$-approximation of the minimum edge cut of G with probability $1-O(1/n^d)$. The expected time for inserting m_1 edges is*

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

where λ is the size of the minimum cut in the final graph. Each query can be answered in time $O(\log n)$.

Proof: Since $k \leq \lambda$ it follows that $p = 8(d+4)\ln n/(\epsilon^2 k) \geq 8(d+4)\ln n/(\epsilon^2\lambda)$. Since increasing p increases the probability that Lemma 10 holds, it follows that with probability $1 - O(1/n^d)$ the incremental algorithm returns a $(1+\epsilon)$-approximation of the minimum cut in G.
Next we analyze the running time. The algorithm executes a 3-approximation algorithm on G whose total time is $O(m_0 + m_1)$. To analyze the remaining steps note that the total time for the incremental exact algorithm is $O(m_0 + m_1\alpha(n,n) + \lambda^2 n \log(n/(\lambda_0+1))) = O(\lambda(m_0+m_1)\log(n/(\lambda_0+1))$.

We first analyze Step 1. As long as $\lambda = O(d \log n)$ the cost of the incremental exact algorithm is $O((m_0+m_1)d\log^2 n)$. Since the expected size of the minimum cut in $G(p)$ is $O(\log n/\epsilon^2)$, the expected time for initializing the 3-approximation and the incremental exact algorithm for $G(p)$ is $O((m_0+m_1)\log^2 n(d+1/\epsilon^2))$. Thus, the total expected time for Step 1 is $O((m_0+m_1)\log^2 n(d+1/\epsilon^2))$.

Next we analyze Step 2 and Step 3. 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 $k = k_i$ (except for the 3-approximation algorithm). In each phase the algorithm executes an incremental exact minimum cut algorithm on $G(p)$. The total expected time for the incremental exact algorithm is $O(\lambda(G(p))(m_0+m_1)\log(n/(\lambda_0+1)) = O((m_0+m_1)\log^2 n/\epsilon^2)$, since $\lambda(G(p)) = O(\log n/\epsilon^2)$ with high probability.

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 $k_f \leq 3\lambda$. Thus, the total expected time for all phases is $O((m_0+m_1)\log^2 n(\log\lambda)/\epsilon^2)$. Adding the cost for Step 1 and the 3-approximation, the total time is $O((m_0+m_1)\log^2 n(\log\lambda)/\epsilon^2)$ ∎

5 A 2-Approximation of the Minimum Vertex Cut Algorithm

5.1 A Static Algorithm

We give an algorithm that in time $O(n^2 \min(\sqrt{n}, \kappa))$ approximates the vertex connectivity κ of a graph (*not* a multigraph) as follows: If $\kappa \leq \lfloor \delta/2 \rfloor$, then the algorithm outputs κ, otherwise, it outputs $\lfloor \delta/2 \rfloor$. Our algorithm uses ideas from [18].

Even and Tarjan [8] give an algorithm to compute the minimum vertex cut $\kappa(a,b)$ between the nodes a and b in time $c(\min(m\sqrt{n}, m\kappa))$ for some constant c, which we call the *pair algorithm (PA(a,b))*. The exact minimum vertex cut algorithm [11] makes $O(\kappa^2+n)$ calls to PA. Let δ_2 be $\lfloor \delta/2 \rfloor$. The basic approach of our algorithm is to reduce the number of calls to PA to $\lceil n/\delta_2 \rceil$ using the following two observations:

1. The last δ_2 nodes in the mcs-order computed by DA are pairwise δ_2-connected.

2. Let $L = \{v_1, \ldots, v_{\delta_2}\}$ be a set of δ_2 pairwise δ_2-connected nodes and let a be a node not in L. Let G' be the graph constructed from G by adding a node b and connecting it to every node in L. Then a is δ_2-connected in G to each v_j iff a is δ_2-connected in G' to b.

We use these two observations as follows. We add a node a to G, repeatedly *select* a set L of δ_2 pairwise δ_2-connected nodes (using DA), and *test* if all of them are δ_2-connected to a (using one call to PA). We guarantee that (1) all but the last call to DA selects δ_2 not-yet selected nodes, and (2) the last call to DA returns a set of δ_2 nodes containing all not-yet selected nodes. Thus, the algorithms calls PA only $\lceil n/\delta_2 \rceil$ times.

To guarantee (1) and (2) we actually fulfill the stronger condition that all already-selected nodes are before all unselected nodes in mcs-order in all calls to DA. To achieve this we execute the ith call of DA on a suitably defined graph G_i, for $i = 1, 2, \ldots, \lceil n/\delta_2 \rceil$. Let L_i be the δ_2 nodes selected by the ith call to DA (i.e., the last δ_2 nodes visited by the ith call to DA) and let S_i be the set of selected nodes after the ith call to DA. The graph G_i is constructed from G by adding two nodes a and b and adding edges so that

1. all nodes of S_{i-1} are connected by an edge to a, and
2. all nodes of S_{i-1} and edges between them form a complete graph.

We show below that in the mcs-order created by PA on G_i started on a, all nodes in S_{i-1} appear before all other nodes.

Here are the details of the algorithms.

A 2-Approximation Algorithm for the Minimum Vertex Cut

Compute a DA-msfd $F_1, \ldots F_m$ of G and set $G_0 = \cup_{i \le \delta} F_i$.
Determine L_0 using DA on G_0, set $S_0 = L_0$.
$k = \lfloor \delta/2 \rfloor$.
for $i = 1$ **to** $\lceil n/\lfloor \delta/2 \rfloor \rceil$
    Create graph $G_i'' = (V \cup \{a, b_i\}, E_i'')$, where $E_i'' = \{F_j, 1 \le j \le \delta\} \cup$
    $\cup \{(x, a), x \in S_{i-1}\} \cup \{(y, b_i), y \in L_i\}$.
    if $i < \lceil n/\lfloor \delta/2 \rfloor \rceil$ **then**
        determine L_i simulating DA on G_i. Let $S_i = \bigcup_{j \le i} L_j$.
endfor
steps $= 0$;
while *steps* $< c(\delta + 1) n \min(k, \sqrt{n})$ **do**
    for $i = 1$ **to** $\lceil n/\lfloor \delta/2 \rfloor \rceil$ **in lockstep**
        Execute one step of $PA(a, b_i)$ on G_i''.
        if $PA(a, b_i)$ terminates and $\kappa(a, b_i) < k$ **then** $k = \kappa(a, b_i)$.
        steps $=$ *steps* $+ 1$.
    endfor
endwhile

Correctness

We first show that the δ_2 nodes selected by the execution of DA on G_i fulfill (1) and (2) for $i = 0, 1, \ldots \lceil n/\delta_2 \rceil - 1$.

Lemma 12. *If DA is executed on G_i and its startnode is a, then in the mcs-order created by DA, all nodes of S_{i-1} lie before all nodes not belonging to S_{i-1}.*

Proof: The mcs-order produced by DA is the order in which DA visits the nodes. Thus, if DA is started on a, the first node in mcs-order is a. To repeatedly determine the next node, DA fulfills the following invariant: The next visited node is an unvisited node with maximum *num*-value, where $num(x)$ is the number of visited neighbors of x.
We show the lemma by induction on j, the number of already visited nodes. Since a is only connected to nodes of S_{i-1}, the second visited node is a node of S_{i-1} and, thus, the claim holds for $j = 2$. Assume next DA has to determine the j-th node to visit with $2 < j \leq |S_{i-1}| + 1$. By induction, all visited nodes belong to $S_{i-1} \cup \{a\}$. Thus, for each unvisited node x of S_{i-1}, $num(x) = j - 1$, and for each unvisited node $y \notin S_{i-1}$, $num(y) < j - 1$, since these nodes are not incident to a. It follows that DA chooses a node of S_{i-1}, and, thus, all nodes in S_{i-1} are visited before all nodes not in S_{i-1}. ∎

Next we proof that observation 1 holds.

Lemma 13. *The last $\lfloor \delta/2 \rfloor$ nodes in mcs-order are connected in $F_{\lfloor \delta/2 \rfloor}$.*

Proof: See Appendix A. ∎

Lemma 14. *All nodes that are connected in F_i are i-connected.*

Proof: See Appendix A. ∎

For efficiency, we do not actually execute DA on the graph $(V \cup \{a\}, E \cup \{(a, x), x \in S_{i-1}\}$, but instead "simulate" this execution: For $i > 0$, instead of executing DA on G_i, we call DA on G_i'' starting at a, force DA to first visit all nodes in $S_i \setminus \{a\}$, and afterwards allow DA to select nodes as usual. This takes time $O(\delta n)$ and it can be easily shown that it generates the same msfd-order as DA on G_i.

So far, we have shown an efficient way of repeatedly finding nodes that are δ_2-connected, but we only can guarantee that the nodes in L_i are δ_2-connected in G_i, *not* in G. Thus, instead of running PA on G, we are forced to execute PA on G_i, i.e., we compute $\kappa(a, b)$ in G_i instead of G. However, the next lemma shows that this is not a problem: the minimum over all the $\kappa(G_i, a, v)$-values equals $\kappa(G)$ if $\kappa(G) \leq \delta_2$, and it equals δ_2 otherwise.

We use $\Gamma(G, a)$ to denote the set of neighbors of a in G.

Lemma 15. *If $\kappa(G) \leq \delta_2$, then*

$$\min_{i>0} \min_{v \in L_i} \kappa(G_i, a, v) = \kappa(G).$$

If $\kappa(G) > \delta_2$, then

$$\min_{i>0} \min_{v \in L_i} \kappa(G_i, a, v) = \delta_2.$$

Proof: By property (1) of a DA-msfd, $\kappa(G) = \kappa(G_0)$. To prove the lemma we use the following claim whose proof is given in the Appendix B.

Claim 16 *For any integers $i > 1$ and p, let $L_i = \{v_1, \ldots, v_p\}$ be a set of pairwise p-connected nodes in G_{i-1}.*

1. *Let G_1 be the graph created from G_0 by creating a new node a and connecting the nodes of $L_0 \cup \{a\}$ by a complete graph.*
   (a) If $\kappa(G_0) \leq p$, then $\min_{u \notin \Gamma(G_1,a)} \kappa(G_1, a, u) = \kappa(G_0)$.
   (b) If $\kappa(G_0) > p$, then $\kappa(G_1, a, u) = p$ for every node $u \notin \Gamma(G_1, a)$.
2. *For any $i > 1$, let G_i be the graph constructed from G_{i-1} by connecting all nodes in $S_{i-1} \cup \{a\}$ by a complete graph, and let $\kappa_{min} = \min_{v \in L_i} \kappa(G_{i-1}, a, v)$. If L_i contains a node v such that $v \notin \Gamma(G_{i-1}, a)$, then*

$$\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u)) = \min(\kappa_{min}, \min_{u \notin \Gamma(G_i,a)} \kappa(G_i, a, u)).$$

By induction on part 2 of the claim it follows that

$$\begin{aligned}\min_{u \notin \Gamma(G_1,a)} \kappa(G_1, a, u)) &= \min(\min_{v \in L_1} \kappa(G_1, a, v), \min_{u \notin \Gamma(G_2,a)} \kappa(G_2, a, u)) \\ &= \min_{i>0} \min_{v \in L_i} \kappa(G_i, a, v).\end{aligned}$$

From part 1 of the claim it follows that

$$\min_{u \notin \Gamma(G_1,a)} \kappa(G_1, a, u) = \delta_2 \text{ if } \kappa(G) > \delta_2,$$

and

$$\min_{u \notin \Gamma(G_1,a)} = \kappa(G_0), \quad \text{otherwise.}$$

∎

Of course, we want to compute $\kappa(G_i, a, v)$ using observation 2. Let G_i' be created from G_i by adding a new node b and adding edges from each node in L_i to b. We show next a generalization of observation 2, using repeatedly the following fact.

Fact 17 *Let X, C, and Y be a partition of V such that $E \cap X \times Y = \emptyset$. Let $a \in X$, $u \in Y$ such that $|C| = \kappa(a, u)$ and let $L = X \cup C$. Then adding edges between nodes in L does not modify $\kappa(a, u)$.*

Note that the fact also holds if L is a subset of $X \cup C$.

Lemma 18. *For any integer p, let $L = \{v_1, \ldots, v_p\}$ be a set of nodes that are pairwise p-connected in G and let a be a node of G. Let G' be the graph created from G by adding a vertex b and edges (b, v_j) for every j. Let κ_{min} be $\min_{v \in L} \kappa(G, a, v)$.*

(A) *If $\kappa_{min} \leq p$ then $\kappa(G', a, b) = \kappa_{min}$.*
(B) *If $\kappa_{min} > p$, then $\kappa(G', a, b) = p$.*

Proof: We show the following two claims:
(1) $\kappa(G', a, b) \leq \kappa_{min}$
(2) $\kappa(G', a, b) \geq \min(p, \kappa_{min})$.

(A) If $\kappa_{min} \leq p$, it follows from (1) and (2) that $\kappa(G', a, b) = \kappa_{min}$. (B) If $\kappa_{min} > p$, it follows from (2) that $\kappa(G', a, b) \geq p$. Since the degree of b is p, $\kappa(G', a, b) = p$.

(1) Let $c := \kappa(G', a, b)$. Assume there exists an $v \in L$ such that $\kappa(G, a, v) < c$. Then there exists a partition X, C, and Y of V in G such that there is no edge between a node of X and a node of Y, $|C| < c$, and $a \in X$ and $v \in Y$. Since the degree of b is p, $p \geq c$. Since all nodes in L are p-connected, they are all contained in $C \cup Y$. Now consider this partition in G'. Since b is only incident to nodes in L, adding b to Y shows that C is also a vertex cut in G' separating a and b. Thus, $\kappa(G', a, b) < c$, which leads to a contradiction.

(2) Let $c := \min(p, \kappa_{min})$. Assume that $\kappa(G', a, b) < c$. Then there exists a partition X, C, and Y of V in G' such that there is no edge between a node of X and a node of Y, $|C| < c$, and $a \in X$ and $b \in Y$. Since all nodes in L are p-connected, they are all contained in $C \cup Y$ and since $|C| < c \leq |L|$, there exists a node $v_j \in Y$. Thus, $c > \kappa(G', a, v_j) \geq \kappa(G, a, v_j)$, which leads to a contradiction. ∎

The graphs G'_i might have many edges that do not belong to G. Since the running time of PA depends on the edges of the graph on which it is executed, we execute PA instead on a family of graphs G''_i such that

- G''_i has $O(m)$ edges, and
- $\kappa(G'_i, a, b) = \kappa(G''_i, a, b)$.

Let G''_i be the graph created from G'_i by removing all edges between nodes of S_{i-1} that do not belong to G_0.

Lemma 19. *For all i, $\kappa(G'_i, a, b) = \kappa(G''_i, a, b)$.*

Proof: Consider the partition X, C, and Y of $V \cup \{a, b\}$ in G''_i such that $a \in X$, $b \in Y$, and $|C| = \kappa(G''_i, a, b)$. Since all nodes incident to a lie in $X \cup C$ and all edges in $G'_i \setminus G''_i$ are incident to two of them, it follows by Fact 17 that $\kappa(G'_i, a, b) = \kappa(G''_i, a, b)$. ∎

Thus, executing PA on G''_i computes $\kappa(G'_i, a, b)$ and takes time $O(m \min(\sqrt{n}, \kappa))$.

Lemma 20. *At termination of the algorithm if $\kappa(G) \leq \delta_2$ then $k = \kappa(G)$, if $\kappa(G) > \delta_2$, then $k = \delta_2$.*

Proof: For $\kappa(G_0) \leq \delta_2$, combining Lemmata 15, 18, and 19 gives

$$\kappa(G) = \min_{i>0} \min_{v \in L_i} \kappa(G_i, a, v) = \min_{i>0} \kappa(G_i', a, b) = \min_{i>0} \kappa(G_i'', a, b).$$

Using Lemmata 18 and 19 for $\kappa(G_0) > \delta_2$, shows that for all i,

$$\delta_2 = \kappa(G_i', a, b) = \kappa(G_i'', a, b).$$

Thus, $\min_{i>0} \kappa(G_i'', a, b) = \delta_2$.
We are left with showing that at termination $k = \min_{i>0} \kappa(G_i'', a, b)$. Obviously, $k \geq \min_{i>0} \kappa(G_i'', a, b)$. Assume by contradiction that $k > \min_{i>0} \kappa(G_i'', a, b)$ and let j be an index such that $\min_{i>0} \kappa(G_i'', a, b) = \kappa(G_j'', a, b)$. Each PA algorithm is executed for at least $c(\delta+2)n \min(k, \sqrt{n})$ steps. Thus, PA on G_j'', which takes only $c(\delta + 2)n \min(\kappa(G_j'', a, b), \sqrt{n})$ steps must have terminated when the *while*-loop terminates. Hence, $k \leq \kappa(G_j'', a, b) = \min_{i>0} \kappa(G_i'', a, b)$, a contradiction. ∎

Running Time Analysis

Theorem 21. *A 2-approximation of the minimum vertex cut can be computed in time $O(n^2 \min(\kappa, \sqrt{n}))$, where κ is the size of the minimum vertex cut.*

Proof: The time spent before the *while*-loop is $O(m + (n/\delta)\delta n) = O(n^2)$. The while loop takes at most $(\delta + 2)n \min(\kappa, \sqrt{n})$ steps, each takes n/δ_2 time. time $O((n/\delta)(\delta + 2)n \min(\kappa, \sqrt{n})) = O(n^2 \min(\kappa, \sqrt{n}))$. ∎

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

Using the generic incremental algorithm, we construct an incremental $(2 + \epsilon)$-approximation algorithm. We compute a 2-approximation k of κ in the initial graph. Let $b_1, \ldots b_{\lceil n/\delta_2 \rceil}$ be all the nodes b created by the static 2-approximation algorithm. Note that PA computes a maximum (a, b_j) flow in a directed graph. To test if the current solution is still correct, we maintain (1) the minimum degree δ in G, (2) $\kappa(a, b_j)$, (3) the residual graphs of all maximum (a, b_j)-flows, and (4) a breadth-first search tree in each residual graph rooted at a. For each b_j we incremently maintain the maximum flow from a to b_j by augmenting the breadth-first search tree. Thus, for any j the total time spent for incrementing the maximum flow from a to b_j by 1 is $O(m + n)$ (see also [13]).

The algorithm recomputes every new solution from scratch. We denote by δ the minimum degree in the current graph, and by δ' the minimum degree in the graph during the last recomputation. To test efficiently the correctness of the current solution, we check if $k < \delta'/2$ or $\delta \leq \delta_2(1 + \epsilon)$, where $k = \min_j \kappa(a, b_j)$. If $k < \delta'/2$, then by Lemma 20, $k = \kappa$. Since the degree of every node b_j is fixed to be $\delta'/2$, k is always $\leq \delta'/2$ and $\leq \kappa$. However, if $\delta \leq d(1 + \epsilon/2)$, then $k \leq \kappa \leq \delta \leq \delta'(1 + \epsilon/2) = (2 + \epsilon)k$. Thus, in either case, k is a $(2 + \epsilon)$-approximation of κ. If these conditions are no longer fulfilled, we recompute. As we show below, this happens $O((\log \kappa)/\epsilon)$ times.

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

1. Set δ' and δ to the minimum degree in G.
   Compute a DA-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, storing all the maximum flows from a to b_j and their residual graphs.
   For each such residual graph keep a breadth-first search tree rooted at a.
2. **while** $\delta \leq \delta'(1+\epsilon/2)$ or $k < \delta'/2$ **do**
   if the next operation is a query **then return** k
   else for each j with $\kappa(a, b_j) = k$ **do**
   try to augment every maximum flow $\kappa(a, b_j)$ by adding the new edge to its residual graph.
   Maintain k as the minimum of all $\kappa(a, b_j)$.
   endwhile
   Goto 1.

Theorem 22. *Let G be a graph with n nodes and m_0 initial edges. The total time for maintaining a $(2+\epsilon)$-approximation of the minimum vertex cut κ during m_1 insertions is*

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

Each query can be answered in constant time.

Proof: Let δ'_i be the value that δ' assumes before the $i+1$st execution of Step 1. We define *Phase i* to consist of all the steps executed while $\delta' = \delta'_i$ and let u_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 any j is $O(\delta'_i(\delta'_i n + u_i))$. It takes time $O(\delta'_i n + u_i)$ to increment $\kappa(a, b_j)$ by one. Since $\kappa(a, b_j) \leq \delta'_i/2$, we spend $O(\delta'_i(\delta'_i n + u_i))$ time for every $1 \leq j \leq \lceil n/\delta'_i \rceil$. Thus, the total time spent for all j in Phase i is $O(n(\delta'_i n + u_i n))$. We denote the number of phases by f. We first bound the cost of Phases 1 to $f-1$. Since δ' increases by a factor of at least $(1+\epsilon/2)$ per phase, $\sum_{i<f} \delta'_i = O(\delta'_{f-1}/\epsilon)$. Since $k = \delta'_{f-1}/2$ and $k \leq \kappa$, it follows that $\delta'_{f-1}/2 \leq \kappa$, i.e. $\delta'_{f-1} = O(\kappa)$. Thus, the total cost of Phases 1 to $f-1$ is $O(\kappa n^2/\epsilon + m_1 n)$.

Since at the end of Phase f, $\kappa(a, b_j) = \kappa$, it follows that, for each j, $\kappa(a, b_j)$ is incremented at most κ times. Thus, the work for each j is $O(\kappa(\delta'_f n + u_f))$. The total work for all j is, thus, $O(\kappa n^2 + m_1 n)$.

This implies that the total time for the algorithm is $O(m_0 + m_1 n + \kappa n^2/\epsilon)$. ∎

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 if the algorithm starts with an empty graph.

6 Appendix A

Let G be a graph without multiple edges. We want to prove Lemma 13 and 14. We first present DA and show two properties of DA. Using a further lemma of [20], we then prove Lemma 13. The proof of Lemma 14 follows.

Lemma 14 and Lemma 23 has been shown independently in [7].

In DA, whenever a node is added to a forest F_i, the edge is also given an direction.

The Decomposition Algorithm [20]

1. $E_1 = E_2 = \ldots = E_m = \emptyset$, every node and every edge in G is unscanned.
2. **for** all $v \in V$ **do** $r(v) = 0$.
3. **while** there exists an unscanned node **do**
4. Let x be an unscanned node with maximum $r(x)$.
5. **for** each unscanned edge $\{x, y\}$ incident to x **do**
6. $E_{r(y)+1} = E_{r(y)+1} \cup \{x \to y\}$.
7. **if** $r(x) = r(y)$ **then** $r(x) = r(x) + 1$.
8. $r(y) = r(y) + 1$.

We say that a node that is chosen in Line 4 is *selected.* We need to show the following property:

Lemma 23. *Let u be a vertex of V, let T be a rooted tree of F_i (for any i) that contains u, and let x be the root of T. Then all nodes selected after x and before u lie in T.*

Proof: Let $x_1, \ldots, x_j, \ldots, x_l = u$ be the nodes that have been selected after x in this order. When x is selected, it has maximum r-value, and $r(x) < i$. Thus, $r(z) < i$ for all nodes z when x is selected.
We show the claim by induction on j. We assume inductively that $\{x_1, \ldots x_{j-1}\}$ belongs to T and want to show that x_j belongs to T. Consider the time when x_j is selected for $1 \leq j \leq l$. Let v the node closest to u on the path P from x to u in T that belongs already to T when x_j is selected. Since $x \in P$ belongs to T before x_j is selected, v is well-defined. Obviously v has not yet been selected, since otherwise it would not be the node closest to u on P that belongs already to T. Also $r(v) \geq i$, since v belong to T. Thus, $r(x_j) \geq r(v) \geq i$ at the time when x_j is selected.

Since $r(x_j) < i$ when x was selected, $r(x_j)$ must have been incremented to i after x was selected and before x_j was selected. This can happen only if an edge (z, x_j) was put into E_i. This implies that z was selected after x and before x_j, i.e. $z \in \{x_1, \ldots x_{j-1}\}$. By induction z belongs to T, and, thus, x_j belongs to T. ∎

Lemma 24. *Each node is a root in at most one of the forests $F_1, \ldots, F_m$.*

Proof: Consider the node x. Let $r(x)$ have value i, when x is selected. Then $r(z) \leq i$ for all nodes z. The node x becomes a root of a tree whenever $r(x)$ is increased. This happens whenever there exists an unscanned edge $\{x, z\}$ and $r(x) = r(z)$. After scanning the first such edge, $r(x)$ is incremented to $i+1$, but for each unscanned edge $\{x, y\}$, $r(y) \leq i$. Thus, $r(x)$ cannot be incremented again, i.e. x cannot become the root of another tree. ∎

In [20] the following properties of DA are proven:

Lemma 25. *1. For a graph $G = (V, E)$, let $F_1, \dots, F_m$ be a DA-msfd of G and let $G' = (V, E')$ with $E' = \cup_{i \leq \delta} F_i$. Then the last node x in mcs-order has exactly one incoming edge from each forest $F_1, \dots, F_\delta$.*
2. If two nodes are connected in F_i, they are connected in F_j with $j < i$.

Lemma 13. *When DA is started on a graph G, the last $\lfloor \delta/2 \rfloor$ nodes in mcs-order are connected in $F_{\lfloor \delta/2 \rfloor}$.*

Proof: The last node x in mcs-order belongs by Lemma 25 to a tree of every forest F_i with $1 \leq i \leq \delta$ and is not a root in any of these trees. Let r_i be the root in the forest F_i. By Lemma 24 all these roots are different nodes. Let r_i be the root out of $\{r_{\lfloor \delta/2 \rfloor}, \dots r_\delta\}$ that is selected first. By Lemma 23 all (at least $\lfloor \delta/2 \rfloor$) nodes scanned after r_i belong to the same tree of F_i. Since $i \geq \lfloor \delta/2 \rfloor$, they belong to the same tree of $F_{\lfloor \delta/2 \rfloor}$ by Lemma 25. Thus, the last $\lfloor \delta/2 \rfloor$ nodes in mcs-order are connected in $F_{\lfloor \delta/2 \rfloor}$. ∎

Lemma 14. *All nodes that are connected in F_i are i-connected.*

Proof: Let u and v be two nodes that are connected in F_i. Let $S = \{s_1, s_2, \dots, s_j\}$ be a minimum vertex cut separating u and v in G, let X be the node set of the connected component of u in $G \setminus S$, and let Y be $V \setminus (S \cup X)$.

Assume by contradiction that $j < i$ and let s_j be the last node of S that is scanned by DA. We show below that at termination there is no path between u and v in F_l with $l > j$. Since $i > j$, this contradicts the fact that u and v are connected in F_i.

We are left with showing that at termination there is no path between u and v in F_l with $l > j$. It suffices to show that at termination no vertex in S has both an incoming edge from X in F_l and an outgoing edge to Y in F_l.

All edges leaving nodes of S belong to a forest immediatelly after s_j was scanned. Thus, if a node of S has no outgoing edge into Y in F_l immediatelly after s_j was scanned, this will also hold at termination. If a node s of S has an outgoing edge into Y in F_l immediatelly after s_j was scanned, then

1. by Lemma 3.2 of [20], s does not have an incoming edge from X in F_l immediatelly after s_j was scanned, and

2. by Lemma 2.2 of [20], then $r(s)$ was at least l when s was scanned, and thus is at least l immediatelly after s_j was scanned.

It follows that any edge from a node of X to s that is added to a forest after s_j was scanned is added to a forest F_k with $k > l$. Thus, at termination s does not have an incoming edge from X in F_l. ∎

7 Appendix B

Claim 16. *For any integers $i > 1$ and p, let $L_i = \{v_1, \dots, v_p\}$ be a set of pairwise p-connected nodes in G_{i-1}.*

1. *Let G_1 be the graph created from G_0 by creating a new node a and connecting the nodes of $L_0 \cup \{a\}$ by a complete graph.*
   (a) If $\kappa(G_0) \le p$, then $\min_{u \notin \Gamma(G_1,a)} \kappa(G_1, a, u) = \kappa(G_0)$.
   (b) If $\kappa(G_0) > p$, then $\kappa(G_1, a, u) = p$ for every node $u \notin \Gamma(G_1, a)$.
2. *For any $i > 1$, let G_i be the graph constructed from G_{i-1} by connecting all nodes in $S_{i-1} \cup \{a\}$ by a complete graph, and let $\kappa_{min} = \min_{v \in L_i} \kappa(G_{i-1}, a, v)$. If L_i contains a node v such that $v \notin \Gamma(G_{i-1}, a)$, then*

$$\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u)) = \min(\kappa_{min}, \min_{u \notin \Gamma(G_i,a)} \kappa(G_i, a, u)).$$

Proof: We first show Part 1 in three steps:

(1) If $\kappa(G_0) \le p$, then $\min_{u \notin \Gamma(G_1,a)} \kappa(G_1, a, u) \ge \kappa(G_0)$.
(2) If $\kappa(G_0) \le p$, then $\min_{u \notin \Gamma(G_1,a)} \kappa(G_1, a, u) \le \kappa(G_0)$.
(3) If $\kappa(G_0) > p$, then $\kappa(G_1, a, u) = p$ for every node $u \notin \Gamma(G_1, a)$.

Then we show Part 2 in three steps:

(4) For any $i > 1$, $\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u) \ge \min(\kappa_{min}, \min_{u \notin \Gamma(G_i,a)} \kappa(G_i, a, u))$.
(5) For any $i > 1$, $\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u) \le \min_{u \notin \Gamma(G_i,a)} \kappa(G_i, a, u)$.
(6) For any $i > 1$, $\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u) \le \kappa_{min}$.

Since $x \ge \min(\kappa_{min}, y)$, $x \le y$, and $x \le \kappa_{min}$ implies $x = \min(\kappa_{min}, y)$, the claim follows.

(1) Assume there exists a $u \notin \Gamma(G_1, a)$ such that $\kappa(G_1, a, u) < \kappa(G_0) \le p$. Since there are less than p nodes in the cut C separating a and u in G_1, there exists a $v_j \in L_0$ such that v_j and u are separated by C. Since $\kappa(G_0, v_j, u) \le \kappa(G_1, v_j, u)$, it follows that $\kappa(G_0, v_j, u) < \kappa(G_0)$ which is a contradiction.

(2) Assume $\kappa(G_0) < \min_{u \notin \Gamma(G_1,a)} \kappa(G_1, a, u) =: c$. Then there exists a partition X, C, and Y in G_0 such that there is no edge between a node of X and a node of Y and $|C| < c$. Since the degree of a is p, $p \ge c$. Since the nodes in L_0 are p-connected, the nodes of L_0 are either contained in $X \cup C$ or in $Y \cup C$. Consider this partition in G_1. Note that no edges between X and Y have been added to created G_1. Since a is only connected to nodes in L_0, a is either contained in X or in Y and, thus, there exists a node $u \notin \Gamma(G_1, a)$ such that $\kappa(G_1, a, u) \le |C| < c$, a contradiction.

(3) Note that $\kappa(G_1, a, u) \leq p$, since the degree of a is p. Assume there exists a $u \notin \Gamma(G_1, a)$ such that $\kappa(G_1, a, u) < p$. The same argument as in (1) shows that there exists a v_j such that $\kappa(G_0, v_j, u) < p < \kappa(G_0)$, which is a contradiction. Thus, $\kappa(G_1, a, u) = p$.

(4) Assume there exists a $u \notin \Gamma(G_{i-1}, a)$ such that $\kappa(G_{i-1}, a, u) < \min(\kappa_{min}, \min_{u \notin \Gamma(G_i,a)} \kappa(G_i, a, u)) =: c$. Then there exists a partition X, C, and Y in G_{i-1} such that $a \in X$, $u \in Y$, there is no edge between a node of X and a node of Y, and $|C| = \kappa(G_{i-1}, a, u) < c \leq \kappa_{min}$. The latter shows that $u \notin L_i$, which in turn implies that $u \notin \Gamma(G_i, a)$. Since all nodes of L_i are κ_{min}-connected to a in G_{i-1}, they lie in $X \cup C$. Thus, no edge connecting X and Y is added to G_{i-1} to generate G_i. Hence, by Fact 17, $\kappa(G_i, a, u) = \kappa(G_{i-1}, a, u) < c$ which is a contradiction.

(5) Since adding edges only increases κ it follows that $\kappa(G_i, a, u) \geq \kappa(G_{i-1}, a, u)$. Additionally $\Gamma(G_{i-1}, a) \subseteq \Gamma(G_i, a)$. Thus, $\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u) \leq \min_{u \notin \Gamma(G_i,a)} \kappa(G_i, a, u)$.

(6) For every node $x \in \Gamma(G_{i-1}, a)$ and every node $y \notin \Gamma(G_{i-1}, a)$, $\kappa(G_{i-1}, a, x) = deg(a) \geq \kappa(G_{i-1}, a, y)$. Thus, $\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u) \leq \min_{u \in L_i \cap \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u)$. Since $L_i \cap (V \setminus \Gamma(G_{i-1}, a, u)) \subseteq V \setminus \Gamma(G_{i-1}, a, u)$, $\min_{u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u) \leq \min_{u \in L_i, u \notin \Gamma(G_{i-1},a)} \kappa(G_{i-1}, a, u)$. Thus, the lemma follows. ∎

Acknowledgements

We want to thank David Karger for useful discussions. We are also thankful to Srinivasa Arikati, Yefim Dinitz, Gene Itkis, and Robert Kleinberg for helpful comments on the paper.

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. Dinic, A. V. Karzanov, and M.V. Lomonosov, "A structure of the system of all minimal cuts of a graph", in: *Studies in Discrete Optimization*, A. A. Fridman ed., "Nauka", Moscow, 1976, 290–306 (in Russian).
3. Ye. Dinitz, Z. Nutov, "A 2-level cactus model for the system of minimum and minimum+1 edge-cuts in a graph and its incremental maintenance", to appear in *Proc. 27nd Symp. on Theory of Computing*, 1995.
4. Ye. Dinitz and J. Westbrook, "Maintaining the Classes 4-Edge-Connectivity of a Graph On-Line", Technical Report # 871, Dep. of Comp. Sci., Techmion, 1995, 47p.
5. D. Eppstein, Z. Galil, G. F. Italiano, A. Nissenzweig, "Sparsification - A Technique for Speeding up Dynamic Graph Algorithms" *Proc. 33rd Symp. on Foundations of Computer Science*, 1992, 60–69.
6. S. Even, "An algorithm for determining whether the connectivity of a graph is at least k" *SIAM Journal on Computing*, 4, 1975, 393–396.

7. S. Even, G. Itkis, S. Rajsbaum. "On mixed connectivity certificates", *Algorithms - ESA'95, Proc. 3rd European Symp.*, Springer-Verlag, LNCS 979, 1995, 1–16.
8. S. Even and R. E. Tarjan, "Network flow and testing graph connectivity", *SIAM Journal on Computing*, 4, 1975, 507–518.
9. H. N. Gabow, "A matroid approach to finding edge connectivity and packing arborescences" *Proc. 23rd Symp. on Theory of Computing*, 1991, 112–122.
10. H. N. Gabow, "Applications of a poset representation to edge connectivity and graph rigidity" *Proc. 32nd Symp. on Foundations of Computer Science*, 1991, 812–821.
11. Z. Galil, "Finding the vertex connectivity of graphs", *SIAM Journal on Computing*, 1980, 197–199.
12. Z. Galil and G. P. Italiano, "Maintaining the 3-edge-connected components of a graph on-line", *SIAM Journal on Computing*, 1993, 11–28.
13. A. Ya. Gordon, "One algorithm for the solution of the minimax assignment problem", *Studies in Discrete Optimization*, A. A. Fridman (Ed.), Nauka, Moscow, 1976, 327–333 (in Russian).
14. F. Harary, "Graph Theory", *Addison-Wesley, Reading, MA*, 1969.
15. D. Karger, "Using randomized sparsification to approximate minimum cuts" *Proc. 5th Symp. on Discrete Algorithms*, 1994, 424–432.
16. D. Karger, "Random sampling in cut, flow, and network design problems", *Proc. 26rd Symp. on Theory of Computing*, 1994, 648–657.
17. 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.
18. D. W. Matula, "A linear time $2+\epsilon$ approximation algorithm for edge connectivity" *Proc. 4th Symp. on Discrete Algorithms*, 1993, 500–504.
19. K. Menger, "Zur allgemeinen Kurventheorie", *Fund. Math.* 10, 1927, 96–115.
20. 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.
21. D. D. Sleator, R. E. Tarjan, "A data structure for dynamic trees" *J. Comput. System Sci.* 24, 1983, 362–381.
22. J. Westbrook and R. E. Tarjan, "Maintaining bridge-connected and biconnected components on-line", *Algorithmica* (7) 5/6, 1992, 433–464.
23. H. Whitney, "Non-separable and planar graphs", *Trans. Amer. Math. Soc.* 34, 1932, 339–362.