Average Case Analysis of Dynamic Graph Algorithms

David Alberts*
Monika Rauch Henzinger**

B 95–03
March 1995

Abstract

We present a model for edge updates with restricted randomness in dynamic graph algorithms and a general technique for analyzing the expected running time of an update operation. This model is able to capture the average case in many applications, since (1) it allows restrictions on the set of edges which can be used for insertions and (2) the type (insertion or deletion) of each update operation is arbitrary, i.e., *not* random. We use our technique to analyze existing and new dynamic algorithms for the following problems: maximum cardinality matching, minimum spanning forest, connectivity, 2-edge connectivity, k-edge connectivity, k-vertex connectivity, and bipartiteness. Given a random graph G with m_0 edges and n vertices and a sequence of l update operations such that the graph contains m_i edges after operation i, the expected time for performing the updates for any l is $O(l \log n + n \sum_{i=1}^{l} 1/\sqrt{m_i})$ in the case of minimum spanning forests, connectivity, 2-edge connectivity, and bipartiteness. The expected time per update operation is $O(n)$ in the case of maximum matching. We also give improved bounds for k-edge and k-vertex connectivity. Additionally we give an insertions-only algorithm for maximum cardinality matching with worst-case $O(n)$ amortized time per insertion.

Key words. Average case analysis, Dynamic algorithms, Matching, Connectivity, Bipartiteness.

*Freie Universität Berlin, Institut für Informatik, Takustr. 9, D-14195 Berlin, Germany. Email: alberts@inf.fu-berlin.de. Phone/Fax: +49 30 838 75 164/75 109. Graduiertenkolleg "Algorithmische Diskrete Mathematik", supported by the Deutsche Forschungsgemeinschaft, grant We 1265/2-1. This research was done in part while visiting the Max-Planck Institute for Computer Science, Im Stadtwald, 66123 Saarbrücken, Germany. This work was (partly) supported by the ESPRIT Basic Research Action No. 7141 (ALCOM II).
**Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: mhr@cs.cornell.edu. This research was done in part while visiting the International Computer Science Institute, 1947 Center St., Suite 600, Berkeley, CA 94704 and at the Max-Planck Institute for Computer Science, Im Stadtwald, 66123 Saarbrücken, Germany.

1 Introduction

In many applications a solution to a problem has to be maintained while the problem instance changes incrementally. *Dynamic* algorithms incrementally update the solution by maintaining an additional data structure. Their goal is to be more efficient than recomputing the solution with a static algorithm after every change.

Given an undirected graph $G = (V, E)$, a (fully) dynamic data structure allows the following three operations:

- $Insert(u, v)$: Insert an edge between the node u and the node v.
- $Delete(e)$: Delete the edge e.
- $Query$: Output the current solution. (Depending on the the particular problem a query might be parametrized.)

Two nodes u and v are *k-edge (k-vertex) connected* for fixed k if there are k edge-disjoint (k vertex-disjoint) paths between u and v. A query in the case of connectivity (2-edge connectivity) has 2 parameters u and v and returns "yes" if u and v are connected (2-edge connected). In the case of k-edge (k-vertex) connectivity a query returns "yes" if the graph is k-edge (k-vertex) connected. A *matching* is a subset of the edge set such that no two edges are incident to the same vertex. A *maximum matching* is a matching of maximum possible cardinality. In the case of maximum matching a query outputs a current maximum matching. Alternatively, a query could also be: "Is the edge e in the current graph in the current maximum matching?"

Recently, a lot of work has been done on dynamic algorithms for various connectivity properties [10–13, 16, 24–26]. The current best deterministic bound for maintaining connected or 2-edge connected components of a graph is $O(\sqrt{n})$ [10]. The best randomized algorithm achieves $O(\log^3 n)$, resp. $O(\log^4 n)$ per update [17]. It is an open problem if the connected or 2-edge connected components of a graph can be maintained deterministically faster than $O(\sqrt{n})$. A second interesting question is if a maximum matching can be maintained in time $o(m)$ per update. Note that a dynamic algorithm which executes one phase of the static algorithm by Tarjan [30] for each update operation achieves an update time $O(m)$. This was used for example in [2]. It is already an improvement over recomputation from scratch which takes time $O(\sqrt{n}m)$ [21,32].

We achieve better (average case) bounds for both problems in the following *model of restricted randomness (rr-model)*: Given a random graph G with n vertices and m edges, an adversary can determine whether the type of the next operation is an insertion or a deletion. If the type is an insertion, an edge chosen uniformly from all "allowed" edges not in G is inserted. If the type is a deletion, an edge chosen uniformly from all edges in G is deleted. Thus, only the *parameter* of the next operation is chosen at random, but not the *type* of the next operation.

The rr-model is especially suited to capture the average case in many applications, since (1) it allows restrictions on the set of edges which can be used for insertions and (2) the type (insertion or deletion) of each update operation is arbitrary, i.e., *not* random.

Related Work Karp [18] gave a deletions-only connectivity algorithm. If the initial graph is random and random edges are deleted, the total expected time for a sequence of deletions is $O(n^2 \log n)$.

In [26] a different random input model for dynamic graph algorithms is presented, called *fair stochastic graph process (fsgp)*. It assumes that the type of the next operation as well as its parameter are chosen uniformly at random. Since the rr-model does not make any assumptions about the distribution of the types of update operations, it is more general than a fsgp, which assumes that insertions (deletions) occur with probability 1/2. The algorithm, presented in [26] takes expected time $O(lk \log^3 n)$ maintaining the k-vertex connected components (k constant) for a sequence of $l \geq n^2 \log n$ update operations. This bound is better than our bound in the case of connectivity if the sequence of update operations is long enough and the graphs are not dense, but since the model is weaker, the results are incomparable.

The rr-model is a variation of a model for random update sequences used before in computational geometry (see, e.g., [6,8,22,27]). Eppstein [8] considers the dynamic (geometric) maximum spanning tree problem and related problems for points in the plane. Exploiting their geometry, he gives data structures with polylogarithmic expected update times for these problems.

New Results

- We show that a conceptually simple dynamic algorithm for maximum cardinality matching has an average update time of $O(n)$ per update with respect to the rr-model. The space needed is linear and the preprocessing time is $O(\sqrt{n}m)$. Additionally we give an insertions-only algorithm for maximum cardinality matching with $O(n)$ amortized time per insertion.

- Assuming that the weight of an edge is arbitrary, but fixed, we show that a modified version of Frederickson's topology tree data structure [12] for dynamic minimum spanning forests has an average case update time of $O(\log n + n/\sqrt{m})$ plus amortized constant time. The data structure needs linear space and linear expected preprocessing time. The best worst case update time for this problem is $O(\sqrt{n})$ [10].

- Dynamic connectivity, 2-edge connectivity, and bipartiteness ("Is the current graph bipartite?") are closely related to the dynamic minimum spanning forest problem. They can be updated within the same bounds for space and time. In the worst case the best deterministic bound is $O(\sqrt{n})$ [10] and the best randomized algorithms take polylogarithmic time per update [17].

In the case of k-edge and k-vertex connectivity we slightly improve the known bounds:

- Eppstein et al. [11] describe an algorithm for dynamic k-edge connectivity with worst case update time $O(k^2 n \log(n/k))$ using a minimum edge cut algorithm by Gabow [14]. We show that (with a slight modification) its average case update time is $O(\min(1, kn/m)k^2 n \log(n/k))$ plus $O(k)$ amortized time. This gives time $O(\min(1, n/m)n \log n)$ plus amortized constant time for constant k. The data structure is able to answer a query whether the current graph is k-edge connected in constant time. The data structure needs $O(m + kn)$ space and preprocessing time.

- We create a dynamic k-vertex connectivity algorithm, using the algorithm by Nagamochi and Ibaraki for finding sparse k-vertex certificates [23] and the $O(k^3 n^{1.5} + k^2 n^2)$ minimum vertex cut algorithm by Galil [15]. A query takes constant time. The average update time is $O(\min(1, kn/m)(k^3 n^{1.5} + k^2 n^2))$, which is $O(\min(n^2, n^3/m))$ for constant k. The preprocessing time and the space requirement is linear.

Note that our algorithms are deterministic and *not* randomized. The average case performance of all algorithms matches the best known worst case bounds in the case of sparse graphs, but it is significantly better if there are more edges. In the case of dense graphs these improvements are exponential for some of the problems.

After presenting the rr-model in Section 2 we give a general technique for analyzing the expected running time of an update operation using backwards analysis [28] in Section 3. As far as we know, this is the first application of backwards analysis to dynamic graph problems. In Section 4, 5, 6, 7, 8, 9 we apply this technique to analyze the expected running time of dynamic algorithms for maximum matching, minimum spanning forest, connectivity, bipartiteness, 2-edge connectivity, and k-edge and k-vertex connectivity, respectively. A preliminary version of this paper appeared in [1].

2 A Model for Random Update Sequences

To model the average case it is common practice to consider the expected performance with respect to a "random" input. So we have to define a probability distribution on possible updates. An update consists of two parts, its *type*, i.e., either insert or delete, and its *parameter*, i.e., the specific edge to be inserted or deleted. If the type and the parameter of an operation are given by an adversary, we are in a worst case setting. For the average case analysis at least the edge to be inserted or deleted should be given with some probability distribution. Now two cases are possible: either the type of the update operation is random or not. Reif, Spirakis, and Yung [26] studied a model in which the probability of an insertion (deletion) is 1/2. In contrast, we do not make any assumptions on the distribution of types of update operations. Thus, our analysis also applies if an adversary provides the (worst case) types of update operations.

We adopt a generic model for random update sequences from computational geometry (see, e.g., [6,8,22,27]). The dynamically changing object is a set E which is a random subset of a fixed set $\bar{E}$, the universe. An update is arbitrarily either a deletion of an element of E which has to be chosen uniformly at random from the elements which are currently in the set E, or an insertion of an element chosen uniformly at random from the set $\bar{E} \setminus E$. Since the type of an update operation is not random, the cardinality of E is also not random. Applied to the dynamic graph algorithms setting we get the following model which we call the *model of restricted randomness* or *rr-model*. We have a fixed set of vertices V of cardinality n. $\bar{E}$ is a subset of $\binom{V}{2}$ called *the set of allowed edges* and we call $G = (V, E)$ the *current graph*. If we start with a random subset of $\bar{E}$ of cardinality m_0 (for any m_0) and apply a sequence of updates as described above we get a current graph with a certain number m of edges depending on the type of updates. This graph is with equal probability any of the possible m-edge subgraphs of $\bar{G} = (V, \bar{E})$. If $\bar{E}$ is equal to $\binom{V}{2}$, then G is a random graph in the well-known $G_{n,m}$ model [3].

Note that there are two ways to control the graphs in the rr-model to suit the needs of a particular application. First, we can prescribe $\bar{E}$. In consequence we can impose some structure, e.g., bipartiteness on the graphs which occur. Moreover, if certain edges never occur, we can model this by excluding these edges from $\bar{E}$. Second, we may assume that an adversary gives us a worst case sequence of update types. Thus, we can also handle highly regular update patterns, e.g., l insertions, l deletions, l insertions, and so on.

3 Average Case Analysis

In this section we present an abstract setting for the average case analysis of dynamic data structures with respect to the rr-model. We use a technique called backwards analysis, which already lead to a variety of elegant proofs for randomized incremental geometric algorithms, see [28] and its references. If all updates are performed in approximately the same time bound, there is no need for an average case analysis. We are interested in dynamic data structures where we employ two update algorithms: a slow algorithm that works in any case and a fast algorithm that works only when certain conditions are met. We assume that these conditions depend on some structural property S of the current graph G and the update, namely we can apply the fast algorithm if and only if the update does not change S. In our case the structural property S will be a subgraph of the current graph.

(In the following we could also handle asymmetric update times for insertions and deletions, e.g., the slow insertion time is not the same as the slow deletion time. Since we do not need this for our applications, we stick to the symmetric case.)

In this paper a graph property $\mathcal{P}$ is either a predicate on the graph (e.g. bipartiteness), or a predicate on pairs of nodes (e.g. connectivity), or a set of subgraphs (e.g. maximum cardinality matching). To efficiently maintain a graph property under updates we maintain a "stronger" graph property $\mathcal{S}$ that can be represented by a set of subgraphs. In the following we will map every current graph G to such a set of its subgraphs $\mathcal{S}(G)$. The property $\mathcal{S}$ is stronger than the property $\mathcal{P}$ in the following sense: if $\mathcal{P}$ is different for two graphs G and $G \cup \{e\}$ then no element of $\mathcal{S}(G)$ is contained in $\mathcal{S}(G \cup \{e\})$. This guarantees that to test if $\mathcal{P}$ changes after an update it suffices for our algorithms to maintain one element of $\mathcal{S}(G)$ instead of the whole set. In order to use backwards analysis to analyze the average case behavior we pose two additional constraints on $\mathcal{S}$.

To be precise: for a given graph property $\mathcal{P}$ and a given graph $\bar{G}$ a mapping $\mathcal{S}_\mathcal{P}$ from every subgraph G of $\bar{G}$ into the set of subgraphs of G is a *suitable property for $\mathcal{P}$ and $\bar{G}$* if it has the following properties:

1. Let G and $G \cup \{e\}$ be subgraphs of $\bar{G}$ with $e \notin G$. If $\mathcal{P}(G)$ is not equal to $\mathcal{P}(G \cup \{e\})$ then every $S \in \mathcal{S}_\mathcal{P}(G \cup \{e\})$ contains e.

2. Let G and $G \cup \{e\}$ be subgraphs of $\bar{G}$ with $e \notin G$. If $\mathcal{P}(G)$ is equal to $\mathcal{P}(G \cup \{e\})$ then $\mathcal{S}_\mathcal{P}(G)$ is equal to the set of all $S \in \mathcal{S}_\mathcal{P}(G \cup \{e\})$ which do not contain e.

3. There exists a function s such that every element of $\mathcal{S}_\mathcal{P}(G)$ has size at most $s(n)$.

We define a *suitable subgraph S for $\mathcal{P}$ and G* to be an arbitrary element of $\mathcal{S}_\mathcal{P}(G)$. If $\mathcal{P}$ is understood we say S is a *suitable subgraph for G*. For example in the case of connectivity we can choose $\mathcal{S}(G)$ to be the set of all possible spanning forests of G.

We want to analyze a dynamic algorithm which maintains a suitable subgraph along with other information. For a current graph and a current suitable subgraph S we define an update to be a *good case* if S is also suitable for the new current graph. If S is no longer suitable we define the update to be a *bad case*. The dynamic algorithm performs an update by testing whether it is a good or a bad case and then performing the fast update algorithm in the good case and the slow update algorithm otherwise. Properties 1 and 2 guarantee that a current suitable subgraph S has the following properties:

- The deletion of an edge which is not in S is always a good case.
- If the insertion of an edge e is a bad case, then e has to be in any possible suitable subgraph S' for the new graph.

We now want to derive a bound on the expected running time of one update according to the rr-model. We do not consider the time for testing here. Let D be the dynamic data structure. Let $g(n,m)$ ($b(n,m)$) be the running time of the fast (slow) update algorithm. We assume that $m \geq s(n)$. Otherwise we get a bound of $b(n,m)$. First we analyze a deletion. Let $T_{del}(n,m)$ be the expected running time for deleting an edge in a random m-element subset of $\bar{E}$. Let E be an arbitrary m-element subset of $\bar{E}$ and let $\bar{m} = |\bar{E}|$. Fix one suitable subgraph S for E. Let $T_{del}(E,e)$ be the worst case running time for updating D when $e \in E$ is deleted. Since the bad case occurs only if $e \in S$, we get

$$T_{del}(n,m) = \frac{1}{\binom{\bar{m}}{m} m} \sum_{\substack{E \subset \bar{E} \\ |E|=m}} \sum_{e \in E} T_{del}(E,e) \leq \frac{1}{\binom{\bar{m}}{m} m} \sum_{\substack{E \subset \bar{E} \\ |E|=m}} s(n)b(n,m) + (m - s(n))g(n,m)$$

$$= O(\frac{s(n)}{m} b(n,m) + g(n,m)).$$

Next, we consider the insertion of an edge. Let $T_{ins}(n,m)$ be the expected time needed to insert a random edge if the current random graph has n vertices and m edges. In analogy to $T_{del}(E,e)$ let $T_{ins}(E,e)$ be the time needed to update D if $e \in \bar{E} \setminus E$ is inserted into E. Then we have

$$T_{ins}(n,m) = \frac{1}{\binom{\bar{m}}{m}(\bar{m} - m)} \sum_{\substack{E \subset \bar{E} \\ |E|=m}} \sum_{e \in \bar{E} \setminus E} T_{ins}(E,e),$$

since every pair (E,e) is equally likely according to the rr-model. Now backwards analysis appears on the scene. We formulate the cost in terms of the edge set E' which results by inserting e into E. Choosing m elements from $\bar{E}$ and afterwards an additional one from the remaining set is the same as choosing $m+1$ elements from $\bar{E}$ first and then selecting one of the chosen elements. Thus, we get

$$T_{ins}(n,m) = \frac{1}{\binom{\bar{m}}{m+1}(m+1)} \sum_{\substack{E' \subseteq \bar{E} \\ |E'|=m+1}} \sum_{e \in E'} T_{ins}(E' - e, e).$$

Now, we look at the inner sum. Let $G' = (V, E')$ and let S' be a suitable subgraph for G'. If the insertion of e was a bad case, then e has to be contained in S'. Since $|S'| \leq s(n)$, this happens at most $s(n)$ times. So, we get

$$T_{ins}(n,m) \leq \frac{1}{\binom{\bar{m}}{m+1}(m+1)} \sum_{\substack{E' \subseteq \bar{E} \\ |E'|=m+1}} s(n)b(n,m) + (m+1-s(n))g(n,m)$$

$$= O(\frac{s(n)}{m} b(n,m) + g(n,m)).$$

This implies the following theorem.

Theorem 3.1 *Let $\bar{G}$ be a graph on n vertices; let $\mathcal{P}$ be a graph property; let there be a suitable property $\mathcal{S}_{\mathcal{P}}$ for $\mathcal{P}$ and $\bar{G}$; let D be a dynamic data structure for $\mathcal{P}$ with*

- *a query time of $q(n,m)$,*
- *a bad case update time of $b(n,m)$,*
- *a good case update time of $g(n,m)$, and*
- *a bound of $t(n,m)$ for testing whether an update is a good case.*

Then there is a dynamic graph algorithm for $\mathcal{P}$ with an expected update time with respect to the rr-model of $O(t(n,m)+g(n,m)+\min(1,s(n)/m)b(n,m))$. Its worst case query time is $q(n,m)$.

Note that the gap between average case and worst case performance is the largest if the graph is dense.

4 Maximum Cardinality Matching

4.1 Terminology

The cardinality of a maximum matching is the *matching number* of the graph. In general a maximum matching is not unique. All of the following definitions are with respect to a fixed matching M. A path P in G is an *alternating path with respect to M* iff the edges in P are alternately in the matching M and not in M as we walk along P. We will drop the phrase "with respect to M" whenever there are no ambiguities. A *free vertex* is a vertex which is not incident with any matching edge. An *alternating forest* is a forest in G with the free vertices as roots whose paths are alternating.

An *augmenting path* is an alternating path which starts and ends with a free vertex. A matching can be *augmented* along an augmenting path P by removing the matching edges on P from the matching and inserting the non-matching edges on P into the matching. This yields a matching M' which contains one more edge than M.

A graph H is *factor-critical* if $H-v$ has a perfect matching for every vertex $v \in V(H)$. This implies that $|V(H)|$ is odd and H itself has no perfect matching. Let $G=(V,E)$ be a graph with some matching M. A *blossom* B in G with respect to M is a factor-critical subgraph of G which contains k matching edges where $|V(B)| = 2k+1$. One vertex is a trivial blossom. The easiest nontrivial case is just an odd cycle where all vertices but one are matched. Note that the definition of a blossom is not unique in the literature, we define it similar to [20]. A blossom which is not properly contained in another one is a *maximal blossom*. A *blossom forest with respect to M* is a subgraph F of G containing vertex-disjoint blossoms such that contracting each blossom in F to a single vertex — which is called *shrinking* the blossom — leads to an alternating forest. A *maximum blossom forest* is a blossom forest with maximal cardinality of its vertex set. In the following we only deal with maximum blossom forests and drop the word "maximum". Since one can add an arbitrary number of edges to a blossom and it remains a blossom, blossom forests are not necessarily sparse. But it is easy to see that there always exist sparse blossom forests.

Now let M be a maximum matching again. If there exists an alternating path with respect to M from some free vertex to a certain vertex v, then v is *reachable*. If one of the

alternating paths from a reachable vertex v to some free vertex is of even length then v is an *even vertex*. The length of a path is the number of edges it contains. If v is reachable, but only using odd alternating paths, then it is an *odd vertex*. Free vertices are also even. The sets of even and odd vertices are unique, i.e., they are independent of the particular choice of a maximum matching. Edmonds also proved this in [7]. A non-reachable vertex is called *out-of-forest vertex*.

4.2 The algorithm

The data structure we maintain consists of a sparse blossom forest, parity informations (even, odd, or out-of-forest) for the vertices, and a list consisting of the edges in a current maximum matching. The matching and forest edges are marked for quick recognition. Thus, it is trivial to answer a query. Additionally, we store at each node in the blossom forest a pointer to the tree that it belongs to. A blossom forest is a well-known data structure used in static maximum cardinality matching algorithms, see, e.g., [7, 20, 30]. We will show below that the union of a maximum matching of the current graph and a blossom forest with respect to it is a suitable subgraph for the current graph. It follows that $s(n) = O(n)$. Conceptually, the data structure is a sparse subgraph of the current graph G, which has the same matching number and the same parities as G.

Tarjan [30] describes a static algorithm for computing a maximum matching in general graphs. It proceeds in phases. In each phase it either constructs a sparse blossom forest or it finds an augmenting path with respect to an intermediate matching computed so far and augments this matching in $O(n+m)$ time. The algorithm computes the reachable vertices, their parities, the blossoms and informations to retrieve augmenting paths. It grows an alternating forest and shrinks nontrivial blossoms reachable via an even alternating path when they are detected.

To analyze when S has to be changed we look first at deletions. A change occurs only if we delete an edge in the forest or the matching. Since they are marked, we can easily test this condition. Now suppose we insert an edge e into the current edge set E. Let $E' = E \cup \{e\}$. We have to update the blossom forest or the matching only if one of the following three conditions applies. By using the information stored at the nodes we can test in constant time if one of these conditions applies [30].

(1) The insertion of e increases the matching number. In this case we find an augmenting path when e is inserted, we augment the matching and have to rebuild the blossom forest. If there is a maximum matching in E' not containing e, then the deletion of e from E' does not decrease the matching number, a contradiction, since the matching number is unique. So e has to be in every maximum matching in E'.

(2) The insertion of e increases the number of reachable vertices. In this case the blossom forest grows. Since the reachable vertices are unique and they form the vertex set of every blossom forest, we can argue in the same way as in the previous case that e is in every possible blossom forest for the new graph.

(3) The insertion of e does not change the number of reachable vertices but changes the parity of some odd vertices to even. In this case there is a new blossom in the forest. Since the parities of the reachable vertices within the blossom forest are the same as in the whole graph and they are unique, we can again deduce that e has to be in every possible blossom forest for the new graph.

Assume that in a bad case we simply recompute the data structure by using one phase

of Tarjan's algorithm. If the change also affects the current maximum matching, we have to apply the algorithm twice, once for augmenting and once for computing a new blossom forest with respect to the new maximum matching. These bad cases take $O(n+m)$ time. All good cases can be handled in constant time, since we just update the adjacency structure of the graph. For preprocessing we use the static $O(\sqrt{n}m)$ algorithm of Micali and Vazirani [21, 32] to construct a maximum matching in the initial random graph and one phase of Tarjan's algorithm to construct a sparse blossom forest with respect to the initial maximum matching. Using Theorem 3.1 we get the following result.

Theorem 4.1 *There exists a data structure for dynamic maximum matching which can be updated in $O(n)$ expected time with respect to the rr-model. It returns a current maximum matching or answers the question whether a particular edge is in the current maximum matching in optimal time.*

Even, odd, and out-of-forest vertices correspond to the Gallai-Edmonds-Decomposition of a graph. For a definition and properties of this decomposition see [20]. Since our algorithm maintains the partition of the vertices into even, odd, and out-of-forest vertices, it also maintains the Gallai-Edmonds-Decomposition of the graph.

4.3 Insertions only

We sketch below an insertions-only maximum cardinality matching algorithm with $O(n)$ amortized time per insertion of an arbitrary (*not* random) edge if the initial edge set is empty.

A close look at Tarjan's algorithm [30] shows that each phase is essentially a semi-dynamic algorithm. It is guided by the edges reachable via even alternating paths in the growing blossom forest. It does not depend on a particular order on these edges. Thus, as long as an inserted edge changes only the blossom forest, but creates no augmenting path there is no need to recompute the data structure from scratch. A sequence of insertions of this type plus an edge which increases the matching number corresponds to one phase of Tarjan's algorithm with a special order of scanning edges. All the work which has to be done in one such phase, i.e., growing the forest, shrinking blossoms, augmenting the matching at the end of the phase, and rebuilding the blossom forest with respect to the new maximum matching afterwards can be charged to the edges which are involved. By doing so, each edge is charged only for a constant amount of work per phase. Since there are at most $n/2$ phases, an insertion can be done in $O(n)$ amortized time.

5 Minimum Spanning Forests

Frederickson [12] introduced the topology tree data structure to maintain a minimum spanning forest dynamically. In this section we modify the topology tree data structure to give a dynamic minimum spanning forest algorithm with good average and the same worst-case performance as the algorithm in [12]. The new variant of topology trees is the key data structure for a number of other dynamic graph algorithms, like different kinds of connectivity and bipartiteness, presented in the following sections.

The worst case update time of the topology tree data structure is $O(\sqrt{m})$. By using improved sparsification [10] this can be reduced to $O(\sqrt{n})$. To apply our technique of

Section 3 we modify the topology trees such that updates involving tree edges take time $O(\sqrt{m})$ (bad case) and updates involving non-tree edges take time $O(\log n)$ plus amortized constant time for rebuilding parts of the data structure (good case). The minimum spanning forest is its own suitable subgraph. It is sparse so by Theorem 3.1 we get an average case update time with respect to the rr-model of $O(n/\sqrt{m} + \log n)$ expected time plus $O(1)$ amortized time if we consider an arbitrary but fixed weight for every edge in $\bar{G}$. We cannot use the sparsification technique of [10, 11] since in their data structure a constant fraction of the edges causes expensive updates.

5.1 Basic Definitions

We first review the data structure by Frederickson and make some changes needed to speed up the good case. We always keep the graph connected by dummy edges of weight ∞. To build a topology tree we map G to a graph G' of maximum degree 3 by replacing a vertex x of G of degree $d \geq 3$ by a cycle of d new vertices $x_1, \ldots, x_d$ in G'. The edges connecting x_i and x_{i+1} and the edge connecting x_d and x_1 get a weight of $-\infty$, so they always stay in the minimum spanning forest of G'. They are called *dashed* edges. Every edge (x, y) is replaced by an edge (x_i, y_j), where i and j are the appropriate indices of the edge in the adjacency lists for x and y. Note, that the edges of a minimum spanning forest of G are a "subset" of those for G'. The topology tree data structure decomposes G' based on a minimum spanning tree T of G'. In the following whenever we use the term *tree edge* we refer to an edge of the current minimum spanning tree in case of a deletion or the new minimum spanning tree in case of an insertion.

To achieve fast updates for non-tree edges we modify the definitions of clusters given in [13]. A *basic cluster* is a set of vertices that induces a subgraph of T that is connected. An edge is *incident* to a cluster if exactly one of its endpoints is in the cluster and it is *internal* to a cluster if both endpoints are in the cluster. The *tree degree* of a cluster is the number of tree edges incident to the cluster. We call a cluster blue if it has tree degree 1 or if it has tree degree 2 and is not incident to a tree degree 3 cluster. A *dynamic restricted partition of order k* with respect to T is a partition of the vertices so that

(1) each cluster with tree degree 3 has cardinality 1,

(2) each set in the partition is a basic cluster with tree degree ≤ 3 and cardinality $\leq 2k$,

(3) each blue cluster has cardinality at least $k/3$.

These conditions guarantee that there are $O(m/k)$ basic clusters in a dynamic restricted partition of order k. A *dynamic restricted multilevel partition of order k* is a hierarchy of dynamic partitions, one partition per *level*. A *level-0* cluster is a basic cluster created by a dynamic restricted partition of order $\sqrt{k}$. A *level-i* cluster is

(1) the union of two level-(i-1) clusters that are connected by a tree edge such that one of them has tree degree 1 or both have tree degree 2, or

(2) one level-(i-1) cluster if the previous rule does not apply.

The topmost partition consists of just one set.

5.2 Data Structure

A *topology tree* TT is a tree which represents a dynamic restricted multilevel partition such that each node C at level i of the tree corresponds to a level-i cluster in the multilevel partition. If C is the union of two clusters C_1 and C_2 at level i-1, then C_1 and C_2 are the children of C and the tree edge (C_1, C_2) is stored at C. If C consists of one level-(i-1) cluster C_1 at level i, then C_1 is the only child of C in TT.

A *2-dimensional topology tree* $2TT$ is a tree that contains a node $C \times D$ at level i for every pair (C, D) of level-i clusters. A level-(i-1) node $C_1 \times D_1$ is a child of a level-i node $C \times D$ iff C_1 is a child of C and D_1 is a child of D.

We maintain a minimum spanning tree T of G' in a dynamic tree data structure. We implicitly maintain a dynamic restricted multilevel partition of order $\sqrt{m}$ by means of a topology tree TT and a two-dimensional topology tree $2TT$.

At every node $C \times D$ of $2TT$ with $C \neq D$ we keep a bit that is set to 1 iff there is an edge between cluster C and cluster D. Let $min(C, D)$ be the edge of minimum cost between C and D. At each leaf $C \times D$ of $2TT$ we keep a priority queue of all non-tree edges with one endpoint in C and one endpoint in D and $min(C, D)$. (This priority queue is not part of the original topology tree data structure, but added to speed up updates.) At an internal node $C \times D$ of $2TT$ we only keep $min(C, D)$ which can be computed in constant time from the *min* value of the (at most four) children of $C \times D$.

5.3 Updating the Mapping from G to G'

Whenever an edge (x, y) is inserted into G and the degree d of x (or y) is already greater than two, we have to update the mapping from G to G'. In general we just add a new node x_* at a suitable location in the cycle representing x in G' and a new edge (x_*, y_j) for an appropriate index j to G'. A special case occurs when the degree of u in G is exactly 2: u is represented by just one vertex u_1 in G' before the insertion and by 3 vertices after the insertion. In order to stay within the claimed space bound we also have to remove a node from the cycle in G' representing a particular vertex in G when an edge is deleted.

5.4 Updating the Dynamic Partition

The insertion or deletion of a node in G' might invalidate the partition by violating some of the conditions. We restore them by the following *rebuild* procedure.

1. If Condition (1) is violated then

   - If this *rebuild* is caused by a tree-edge then we just restore Condition (1) by splitting all affected clusters into subclusters that fulfill Condition (1).
   - If this *rebuild* is caused by a non-tree edge (x, y) in G then we do the following:
     - If x_1 or x_2 lies in a cluster with tree degree less than three, we add the new node x_* to this cluster.
     - Otherwise we create a new cluster containing x_* between the (tree degree 3) clusters containing x_1 and x_2.

   The vertex y is handled symmetrically.

2. If the cardinality of a cluster is larger than $2\sqrt{m}$ (thus violating Condition (2)), the cluster is split using the splitting procedure of [12] which splits a cluster into two clusters of size at most $4/3\sqrt{m}$ and at least $2/3\sqrt{m}$.

3. If a cluster violates Condition (3) we join it with one of its neighbors resulting in either a non-blue cluster or a cluster with cardinality at least $2/3\sqrt{m} - 1$. If in the latter case this cardinality is larger than $5/3\sqrt{m}$, this cluster split again resulting in two clusters of size at most $2/3(2\sqrt{m} + \sqrt{m}/3 - 1) < 5/3\sqrt{m}$ and at least $5/9\sqrt{m}$.

In each step of the *rebuild* procedure previously restored conditions are preserved. We say that a cluster C is *touched* by an update if one of the vertices of the updated edge is an element of C. Note that C grows or shrinks by at most 4 vertices when it is touched.

It is easy to see that only a constant number of clusters are involved in a *rebuild*. Thus, its cost is $O(\sqrt{m})$. In the good case (a non-tree update) we can even show an amortized constant *rebuild* cost per operation. For this purpose we show the following lemmas.

Lemma 5.1 *Each cluster of tree degree 1 or 2 created by a non-tree operation has cardinality less than* $5/3\sqrt{m}$.

Proof. Let C be such a cluster. Consider the *rebuild* which created C. If C was created in Step 1 it has cardinality 1. If C was created either in Step 2 or Step 3 it has cardinality less than $5/3\sqrt{m}$. □

Lemma 5.2 *Each blue cluster created by a non-tree operation has cardinality at least* $1/2\sqrt{m}$.

Proof. Let C be such a cluster. Consider the *rebuild* which created C. Since C is a blue cluster resulting from a non-tree update C was not created in Step 1. If C was created either in Step 2 or Step 3 it has cardinality at least than $1/2\sqrt{m}$. □

Lemma 5.3 *A non-tree update does not convert a non-blue cluster into a blue cluster.*

Proof. Tree degree 3 clusters can only be destroyed, but cannot become blue during a non-tree update. The only possibility to convert a non-blue cluster C into a blue one during a non-tree update is to insert a new cluster between C and its neighboring tree degree 3 cluster. We explicitly excluded it in Step 1. It also cannot happen in Step 2 or 3. □

Lemma 5.4 *The amortized* rebuild *cost of a non-tree update is constant.*

Proof. Step 1 clearly takes $O(1)$ worst case time for a non-tree update. Thus a *rebuild* can incur non-constant worst case cost only if Step 2 or Step 3 are executed. Step 2 is executed if the affected cluster C becomes larger than $2\sqrt{m}$. If less than $1/12\sqrt{m}$ non-tree updates touched C since its creation we know that its size at creation was at least $5/3\sqrt{m}$. Thus by Lemma 5.1 it was created by a tree update. Since each *rebuild* creates only a constant number of new clusters we can charge the cost of destroying C to the tree update that created C. If more than $1/12\sqrt{m}$ non-tree updates touched C we can charge a constant amount of the cost for destroying C to each of these operations.

Step 3 is executed if a blue cluster becomes too small. If this cluster was non-blue when created by Lemma 5.3 there must have been a tree update which converted this cluster from non-blue to blue. Since each tree-edge operation converts only a constant number of clusters we can charge the cost of destroying C to the tree update which converted it. If C was created by a tree update we charge the cost of destroying it to the tree update which created it. Otherwise, C was created as a blue cluster by a non-tree operation. Thus by Lemma 5.2 it had size at least $1/2\sqrt{m}$ when it was created. Thus, at least $1/24\sqrt{m}$ updates must have touched C since its creation. We can charge a constant amount of the cost for destroying C to each of them. □

5.5 Updating the Data Structure

To decide whether we are in a good or a bad case, we have check if the updated edge is a tree edge. In the case of a deletion, this is easy to decide, since we know for each edge in G if it is in T. In the case of an insertion we use an additional dynamic tree data structure [29] (described below) to decide in time $O(\log n)$ which case occurs.

The algorithm in [12] updates TT and $2TT$ in time $O(\sqrt{m})$ after an arbitrary edge insertion or deletion. Building and maintaining the priority queues at the leaves of $2TT$ increases this time only by a constant factor. A *rebuild* in the dynamic partition causes additional costs of $O(\sqrt{m})$ to update TT and $2TT$. In the good case these costs can be amortized as described above. Thus, we omit these additional costs in the following discussion.

We are left with showing that TT and $2TT$ can be updated in time $O(\log n)$ if a non-tree edge (u, v) is inserted or deleted. Note first that TT has not to be modified (when there are no *rebuilds*). Furthermore the insertion or deletion of (u, v) only affects the *min* value of (1) the node $C \times D$, where C is the cluster that contains u and D is the cluster that contains v, and (2) ancestors of $C \times D$. Adding or deleting (u, v) from the priority queue of $C \times D$ takes time $O(\log n)$. Afterwards we compute bottom-up the new *min* value of every ancestor of $C \times D$. Using the *min* value of its children this takes constant time per ancestor. Since $2TT$ has depth $O(\log n)$ [12], updating all ancestors takes time $O(\log n)$ as well. Thus, updating $2TT$ takes time $O(\log n)$ if we know that (u, v) is not a tree edge. To determine if an edge (u, v) that has to be inserted becomes a tree edge, we use the dynamic tree data structure for T. We compare the cost of (u, v) with the maximum cost on the tree path between u and v. Testing this and updating the dynamic tree takes time $O(\log n)$ per operation.

The space requirement for the data structure is linear. Using the linear expected time algorithm for minimum spanning trees [19] during preprocessing gives the following lemma.

Lemma 5.5 *There exists a data structure that maintains a minimum spanning forest of a graph with any real-valued cost-function on the edges. The data structure can be updated in time $O(\sqrt{m})$ if a tree edge is inserted or deleted and in time $O(\log n)$ plus $O(1)$ amortized time if a non-tree edge is inserted or deleted. The data structure needs linear space and linear expected preprocessing time.*

Now we want to analyze the average case update time of this data structure according to the rr-model. We consider an arbitrary but fixed weight for every edge in $\bar{G}$ (i.e. whenever an edge is in G it has the same weight). Every minimum spanning forest is a suitable

subgraph. Therefore we can apply Theorem 3.1 to analyze the expected time per operation, ignoring the cost of rebuilds. Since we showed before that the total time spent for rebuilds during l updates is $O(l)$, this implies the following result.

Theorem 5.6 *There exists a data structure for maintaining a minimum spanning forest such that for any l the expected time for a sequence of l updates starting with a random subgraph of $\bar{G}$ of size m_0 for any m_0 is $O(l \log n + \sum_{i=1}^{l} n/\sqrt{m_i})$, where m_i is the number of edges in G after operation i.*

6 Connectivity

For dynamic connectivity we achieve an average update time according to the rr-model which consists of two parts: an expected time of $O(n/\sqrt{m} + \log n)$ and an amortized cost of $O(1)$, where m is the number of edges in G after the update operation.

To maintain connectivity dynamically Frederickson [12] assigns cost 1 to edges in the current graphs and connects different connected component by cost 2 (dummy) edges. Then he augments the topology tree data structure in order to answer connectivity queries in constant time. However his algorithm modifies the additional parts of the data structure only if the minimum spanning tree of G changes, i.e., in the bad case. Using the minimum spanning forest data structure presented in the previous section gives the following result.

Theorem 6.1 *There exists a data structure that answers connectivity queries in constant time and that can be updated in total expected time $O(l \log n + \sum_{i=1}^{l} n/\sqrt{m_i})$ during a sequence of l update operations starting with a random subgraph of $\bar{G}$ of size m_0 for any m_0, where m_i is the number of edges in G after operation i.*

7 Bipartiteness

In this section we analyze the average case performance of an algorithm for dynamic bipartiteness due to Eppstein et al. [10, 11]. As before we give each edge cost 1 and connect different connected components by dummy edges of cost 2. The algorithm uses again the topology tree data structure (see Section 5). The basic idea is to maintain a minimum spanning tree T of the graph G and to use TT and $2TT$ additionally to maintain the parities of the cycles which are induced by the non-tree edges. The graph is bipartite if and only if no non-tree edge induces an odd cycle.

For a non-tree edge e let λ_e denote its induced cycle. Let $d(u, v)$ be the distance of the vertices u and v in T, i.e., edges introduced to satisfy the degree constraints are not counted. A *boundary vertex* of a cluster is an endpoint of a tree edge connecting the cluster with a different cluster at the same level of the topology tree. The following lemma is shown in [11].

Lemma 7.1 *Let V_j and V_r be any two clusters at the same level of the topology tree, and let f_1 and f_2 be any two non-tree edges between V_j and V_r. Let w_j be a boundary vertex of V_j, and let w_r be a boundary vertex of V_r. Let j_1 and j_2 be respectively the endpoints of f_1 and f_2 in V_j and let r_1 and r_2 be respectively the endpoints of f_1 and f_2 in V_r. The two cycles λ_{f_1} and λ_{f_2} have the same parity if and only if the quantity $d(j_1, w_j) + d(j_2, w_j) + d(r_1, w_r) + d(r_2, w_r)$ is even.*

We can partition the edges joining two clusters V_j and V_r at the same level of TT by means of this lemma in two parity classes using only local information, i.e. information available in V_j and V_r. This is useful, since updates which are not local to V_j and V_r may nevertheless change the parities of the cycles induced by edges joining V_j and V_r but the two classes remain the same. This leads to a data structure which is a slight modification of the data structure given in [11] consisting of

1. a MST T,
2. a dynamic tree data structure [29] of T (for determining distances between nodes in T) giving dashed edges length 0 and non-dashed edges length 1,
3. a topology tree TT for T where we store at each node V_j the distances between every pair of boundary vertices of V_j,
4. the corresponding 2-dimensional topology tree $2TT$. The nodes of $2TT$ are augmented with the following labels:

   (a) At each leaf $V_j \times V_r$ of $2TT$ we keep two lists, each one containing the non-tree edges of G between V_j and V_r of the same parity. We added this data structure to speed up updates in the good case.

   (b) Associated with each node $V_j \times V_r$ of $2TT$ are up to two edges which represent the two parity classes. These are called the *selected edges*. For each selected edge we maintain the distances of its endpoints to the boundary vertices of the corresponding clusters.

Note that at each node of TT and $2TT$ we store only a constant amount of distance information because every cluster has tree degree at most three. In contrast to [11] we do not need to maintain selected edges *of minimum weight*. However, we could do this by replacing the lists in 4.(a) by priority queues.

We answer a query as follows: If the root node $V \times V$ of $2TT$ does not have a selected edge, then G is a forest and hence bipartite. If $V \times V$ has two selected edges, one of them introduces an odd cycle. Thus, G is not bipartite. If exactly one selected edge is stored at $V \times V$, we determine after each update the length of the cycle induced by this selected edge. This takes logarithmic time using the dynamic tree data structure of T.

As shown in [11] the worst-case update time for this data structure is $O(\sqrt{m})$. We show next that the insertion or deletion of a non-tree edge $e = (u, v)$ that does not modify the dynamic restricted partition takes time $O(\log n)$. If the dynamic restricted partition does not change, Part 1, 2, and 3 of the data structure do not change. We describe below how to update Part 4.

First assume that e is inserted. Let $u \in V_j$ and $v \in V_r$. We have to compute the parity class of e in order to insert it into the right list at the leaf node $V_j \times V_r$ in $2TT$. If $j = r$ we use the dynamic tree data structure to determine the parity of e and of the selected edges of $V_j \times V_j$. If $j \neq r$ we determine the distance of u (v) to a boundary vertex of V_j (V_r) by determining the number of non-dashed edges on the path in T between them. This can be computed in time $O(\log n)$ using the dynamic tree data structure for T. Then we compare the parity of e with the parities of the selected edges stored at $V_j \times V_r$ (if they exist) in constant time using the distance information in the data structure and Lemma 7.1.

After determining the parity class of e we insert e in the appropriate list. This takes constant time. If the selected edges of $V_j \times V_r$ change, we percolate this change up in $2TT$. Since we can update each level in constant time using Lemma 7.1 the whole procedure takes time $O(\log n)$.

If e is to be deleted, we delete it from the list L at $V_j \times V_r$ in which it is contained. If e was a selected edge we replace it by the next edge in L if there exists one. This takes constant time. Updating the ancestors of $V_j \times V_r$ takes time $O(\log n)$ as in the case of insertions.

As before we take the minimum spanning tree to be the suitable subgraph. Rebuilds caused by non-tree edges add constant amortized time to each update as in Section 5. The analysis for minimum spanning trees carries over, so we get the following theorem.

Theorem 7.2 *There exists a data structure that answers bipartiteness queries in constant time and that can be updated in total expected time $O(l \log n + \sum_{i=1}^{l} n/\sqrt{m_i})$ during a sequence of l update operations starting with a random subgraph of G of size m_0 for any m_0, where m_i is the number of edges in G after operation i.*

8 2-Edge Connectivity

Frederickson gives a data structure, called *ambivalent data structure*, that answers 2-edge connectivity queries in time $O(\log n)$ [13]. It can be updated in time $O(\sqrt{m})$. This data structure is also based on the topology tree data structure. We can modify the ambivalent data structure and the update algorithm in order to speed up the good case. We call the modified ambivalent data structure *good-average case ambivalent data structure (gaca data structure)* and describe it below.

As before we give each edge G cost 1 and then connect G by dummy edges of cost 2 and maintain a minimum spanning tree T of G. The basic idea for testing 2-edge connectivity is the concept of *covering*. A tree edge e is *covered* if there exists a non-tree edge (x, y) such that e lies on the tree path between x and y. As shown in [13], two nodes u and v are 2-edge connected iff all edges in the tree path between u and v are covered. Thus, to answer 2-edge connectivity queries the ambivalent data structure maintains for every tree edge a bit indicating if it is covered. To update the data structure fast if non-tree edges are modified, the gaca data structure maintains instead a *cover* value for each tree edge which is a lower bound on the number of non-tree edges covering the tree edge.

To do this efficiently we partition T into chains, called *complete paths*, as in [13]. Subpaths of complete paths are called *partial paths* and are used to compute the *cover* values of edges on complete paths efficiently while traversing the topology tree bottom-up.

8.1 Partial and Complete Paths

The *partial path* of a basic cluster C is

1. the unique node v of C that is incident to the tree edge incident to C, if the tree degree of C is 1,

2. the tree path between the nodes of C that are incident to the tree edges incident to C, if the tree degree of C is 2, and

3. empty, if the tree degree of C is 3.

The partial path of a level-i cluster C with $i > 0$ is

1. the concatenation of the partial paths of the two children of C in TT, if C consists of the union of two clusters and one of them has tree-degree 2,
2. the unique vertex in the child with tree degree 3, if C is the union of a tree degree 1 and a tree degree 3 cluster,
3. empty, if C is the union of two clusters of tree degree 1, and
4. the partial path of the child of C in TT, if C consists of one cluster of level $i-1$.

The *complete path* of each basic cluster C is empty. The complete path of a level-i cluster C with $i > 0$ is

1. the partial path of the child with tree degree 1, the tree edge connecting it to its sibling with tree degree 3, and the (only) node of its sibling, if C is the union of a cluster with tree degree 1 and a cluster with tree degree 3 (which is called the *head* of the complete path)
2. the concatenation of the partial paths of the children of C in TT, if C is the union of two clusters with tree degree 1, and
3. empty in all other cases.

The data structure $PP(C)$ ($CP(C)$) for the partial (complete) path of a cluster C is a binary search tree of depth $O(\log n)$ whose leaves in left-to-right order correspond to the vertices on the partial path. This data structure is described in detail in Section 8.4. We share the data structures among several paths in the following way:

1. If C is a non-basic cluster with one child, its $PP(C)$ and $CP(C)$ are identical to the data structures of the child of C.
2. If C is a non-basic cluster with children C_1 and C_2 and one of them has tree-degree 2, then $CP(C)$ is empty and the data structure $PP(C)$ representing the partial path of C consists of a new root whose children are $PP(C_1)$ and $PP(C_2)$.
3. If C is a non-basic cluster with a child C_1 with tree degree 1 and a child C_2 with tree degree 3, then $PP(C)$ is empty and $CP(C)$ consists of a new root whose children are a newly created node X and the root of $PP(C_1)$. We store at X the tree edge between C_1 and C_2.
4. If C is a non-basic cluster with a child C_1 with tree degree 1 and a child C_2 with tree degree 1, then $PP(C)$ is empty and $CP(C)$ consists of a new root whose children are the roots of $PP(C_1)$ and $PP(C_2)$.

Every node of TT except for the root or a leaf with tree-degree 3 has a non-empty partial path. Thus, every cluster that is incident to a non-tree edge has a non-empty partial path. We extend the dynamic path data structure of [29] to the following *extended dynamic path data structure*. It represents a set of paths such that two paths are either

vertex-disjoint or one path is contained in the other one. Note that each edge on one of the paths is represented just once because of the sharing of data structures described above. There is a cover value associated to each edge e' in one of the paths. It counts the number of edges which cover e'.

- $Initialize(P, E')$ Build a data structure for a partial path P with a set of covering edges E'.
- $Cover(P, e)$ Increase the cover value of each edge e' in P which is covered by e.
- $Uncover(P, e)$ Decrease the cover value of each edge e' in P which was covered by e.
- $Link(P_1, P_2, e)$ Link the data structures for P_1 and P_2 by the edge e. This is allowed if neither P_1 nor P_2 are subpaths of another path in the data structure.
- $Unlink(P)$ Undo the $Link$ operation that created P. This is allowed if P is currently not linked with another path.
- $RightUncovered(P)$ Return the rightmost uncovered edge on P if it exists.
- $LeftUncovered(P)$ Return the leftmost uncovered edge on P if it exists.

A sequence of $Link$ and $Unlink$ operations results in a "linkage tree". Let d be the depth of this tree. Below we describe an implementation of this data structure that takes constant time for $Link$ and $Unlink$; $O(d+\log n)$ time for $RightUncovered$, $LeftUncovered$, $Cover$, and $Uncover$; and $O(|P| + |E'|)$ time for $Initialize(P, E')$.

We use this data structure to maintain the complete and partial paths together with their coverage information. Note that the head of a complete path can be contained in another complete path. To make them vertex-disjoint we simply create a second copy of these shared nodes in the extended path data structure.

Since d is $O(\log n)$ in our application $RightUncovered$, $LeftUncovered$, $Cover$, and $Uncover$ take time $O(\log n)$. While inserting and deleting edges in G we $Cover$ and $Uncover$ partial and complete paths such that we maintain the following invariant:
(I) *An edge on a complete path is covered by no edges iff it is a bridge in G.*

8.2 The Complete Data Structure

The gaca data structure maintains a minimum spanning tree T, a dynamic restricted partition of order $O(\sqrt{m})$, a topology tree TT, and a 2-dimensional topology tree $2TT$ with additional information stored at the nodes of the topology trees.

To describe the labels we need some further definitions. The distance between two nodes in a basic cluster is defined to be the number of edges on the unique tree path connecting them. If the non-tree edge (u, v) is incident to C and u lies in C, let $proj(u)$ be the node on the partial path of C that is closest to u and let $dist(u, e)$ be the number of edges on the partial path of C between $proj(u)$ and the tree edge e incident to C. For each tree edge e incident to C we denote by $maxcover(C, D, e)$ the maximum of of $dist(u, e)$ over all nodes $u \in C$ that are connected by a non-tree edge to a node in D. We often denote by $maxcover(C, D, e)$ also the node on the partial path that has distance $maxcover(C, D, e)$ from e, since it simplifies the description.

At each node $C \times D$ of $2TT$ with $C \neq D$ we keep the integer value $maxcover(C, D, e)$ for every tree edge e incident to C. For two basic clusters C and D and each tree edge e incident to C we keep a heap $max(C, D, e)$ that contains $dist(u, e)$ for all nodes $u \in C$ that are connected to a node in D by a non-tree edge. The maximum of $max(C, D, e)$ is $maxcover(C, D, e)$. For two level-i clusters C and D with $i > 0$ and every tree edge e incident to C the value $maxcover(C, D, e)$ can be computed in constant time from the $maxcover$ values of the children of $C \times D$ in $2TT$ [13].

For each level-i cluster C we keep at its node $C \times C$ of $2TT$ the following data structure:

1. If C is a basic cluster, we keep a dynamic tree data structure of the spanning tree of C. For each tree edge e not on the partial path of C we maintain the number of non-tree edges covering e, its *cover-counter*. The *cover-counter* of e is used as its cost in the dynamic tree for C. Edges on the partial path have cost zero.

2. If C has a non-empty complete path, we maintain an additional heap $max(C)$. Let C_1 be the degree-1 child of C, and let e be the tree edge incident to C_1. Note that C_1 is a level-(i-1) cluster. The heap $max(C)$ maintains the maximum of all $maxcover(C_1, D, e)$ values for all level-(i-1) clusters $D \neq C_1$.

3. We maintain partial and complete paths in a data structure as described in the previous section.

4. We keep an integer value $length(C)$ which denotes the length of the partial path of C.

5. If C has a complete path, we also keep an integer value $toptobr(C)$, which denotes the distance between the head of the complete path and the uncovered edge on the complete path closest to the head.

To speed up the good case we added the heaps $max(C, D, e)$, the dynamic tree data structures, the heaps that maintain the $max(C)$ to the data structure of [13], and we replaced the data structure for partial and complete paths.

The algorithm of [13] uses the $maxcover$ values and the maximum elements stored in the max heaps to maintain the invariant (I) as follows. Let C_1 and C_2 be children of a cluster C and let e be the tree edge connecting them. If the neither C_1 nor C_2 has tree degree 3 then the partial path of C is covered by $maxcover(C_1, C_2, e)$ and $maxcover(C_2, C_1, e)$. If the tree degree of C_2 is 3, the maximum element of $max(C)$ is used to cover the complete path of C.

We employ the query algorithm [13] which uses the $length$ and $toptobr$ values of the partial and complete paths and some information at basic clusters. (The remaining data structures are actually only needed to maintain these values efficiently.)

8.3 Updates

The ambivalent data structure can be updated in time $O(\sqrt{m})$ after every edge insertion or deletion [13]. Our modifications increase the worst case running time only by a constant factor. As shown in Section 5 *rebuilding* the dynamic partition creates only $O(1)$ amortized cost per non-tree update, so we do not consider *rebuilds* in the following. To show the good average case behavior of the gaca data structure we show below that an insertion or deletion of a non-tree edge requires time $O(\log n)$.

The insertion or deletion of a non-tree edge (u, v) does not affect the structure of TT or $2TT$ or the *length* values. Let $C(u, v)$ be the set of clusters at all levels containing u or v, and let $C_i(x)$ be the level-i cluster containing x. We show first that the update affects only the data structures at clusters in $C(u, v)$ and discuss afterwards how to update these data structures.

(1) The definition of *maxcover* implies that only $maxcover(C_i(u), C_i(v), \cdot)$ and $maxcover(C_i(v), C_i(u), \cdot)$ values with $C_i(u) \neq C_i(v)$ are affected.

(2) The only dynamic trees affected are the ones of $C(u)$ and of $C(v)$.

(3) The only *max* heaps affected are the ones of the at most two clusters C_u and C_v whose complete path contain the partial paths of $C_0(u)$ or $C_0(v)$.

(4) This implies that only for $C \in C(u, v)$ the $PP(C)$ and $CP(C)$ data structures are affected.

(5) This in turn shows that only the *toptobr* values of $O(1)$ clusters are affected, namely, those of C_u and of C_v.

Note that in $C(u, v)$ contains $O(\log n)$ clusters, at most two at each level. We update the data structures at all of these clusters bottom-up starting with the level-0 clusters.

Now we discuss how to update each of the above mentioned data structures.

(1) First we discuss updating the *maxcover* values. Using the dynamic tree data structure of the spanning tree of $C_0(u)$ we can find $dist(u, e)$ to each tree edge e incident to $C_0(u)$ in time $O(\log n)$. Inserting or deleting $dist(u, e)$ from the heap $max(C_0(u), C_0(v), e)$ determines the new value of $maxcover(C_0(u), C_0(v), e)$ in time $O(\log n)$. Since at most four heaps are affected, updating all *maxcover* values at level-0 clusters takes time $O(\log n)$. Each *maxcover* value of an internal node $C_i(u) \times C_i(v)$ can be computed in constant time from the *maxcover* value of its children and, thus, all *maxcover* values can be updated in time $O(\log n)$.

(2) Next we update the dynamic tree data structures of $C_0(u)$ and $C_0(v)$. If $proj(u) = proj(v)$ (and thus $C_0(u) = C_0(v)$) then we increase the *cover-counter* of all tree edges between u and v. Otherwise we increment the *cover-counters* of all edges on the tree path between u and $proj(u)$, and between v and $proj(v)$. In any case this can be done in time $O(\log n)$ in the corresponding dynamic tree data structures.

(3) To update $max(C_u)$ and $max(C_v)$ we delete the old *maxcover* value of the corresponding tree degree 1 child and insert the new one if the value has actually changed. This takes time $O(\log n)$.

(4) Now we discuss updating the partial and complete paths. We distinguish three cases:

(4a) If u and v are contained in the same basic cluster C and the update is an insertion, then we execute $Cover(PP(C), (proj(u), proj(v))$. If they are in the same cluster and the update is a deletion we execute $Uncover(PP(C), (proj(u), proj(v))$.

(4b) If u and v are not contained in the same basic cluster, but both belong to the same complete path, let i be the highest level such that $C_i(u) \neq C_i(v)$. We can determine i in time $O(\log n)$. The only *maxcover* values that have changed

and are used to cover a partial or complete path are $maxcover(C_i(u), C_i(v), e)$ and $maxcover(C_i(v), C_i(u), e)$, where e is the tree edge connecting $C_i(u)$ and $C_i(v)$. Let $m(u)$ and $m(v)$ denote the former values of $maxcover(C_i(u), C_i(v), e)$ and $maxcover(C_i(v), C_i(u), e)$ respectively, and let $m'(u)$ and $m'(v)$ be the current values. We execute $Uncover(PP(C_{i+1}(u)), m(u))$ and $Uncover(PP(C_{i+1}(u)), m(v))$, and then $Cover(PP(C_{i+1}(u)), m'(u))$ and $Cover(PP(C_{i+1}(u)), m'(v))$.

(4c) If v does not belong to the complete path of C_u, then the maximum *maxcover* value in $max(C_u)$ is the only *maxcover* value that changes the coverage of some partial or complete path containing u (namely the complete path of C_u). Thus, we uncover $CP(C_u)$ from the old maximum element of $max(C_u)$ and cover it with the new maximum element of $max(C_u)$. We do the same for same for v.

(5) Finally we discuss updating the *toptobr* values. Since at most 2 complete paths are affected by the update and for a cluster C the $toptobr(C)$ value can be computed in time $O(\log n)$ using $CP(C)$, the data structure for the complete path of C, updating all *toptobr* values takes time $O(\log n)$.

This shows in a that the data structure can be updated in time $O(\log n)$ in the case of a non-tree update. Using the analysis of Section 3 gives the following theorem.

Theorem 8.1 *There exists a dynamic data structure that answers 2-edge connectivity queries in time $O(\log n)$ and that can be updated in $O(l \log n + \sum_{i=1}^{l} n/\sqrt{m_i})$ total expected time during a sequence of l update operations starting with a random subgraph of $\bar{G}$ of size m_0, where m_i is the number of edges in G after operation i.*

8.4 An Extended Dynamic Path Data Structure

In this section we present the extended dynamic path data structure for the maintenance of the cover values of the edges of paths. It is based on the dynamic paths data structure which Sleator and Tarjan used for their dynamic trees [29].

We consider the following problem. We are given a set of paths such that two paths are either vertex-disjoint or one path is contained in the other one. Each path has a leftmost degree one vertex (also called the *head*) and a rightmost degree one vertex (also called the *tail*). There is a cover value associated to each edge e' in one of the paths. It counts the number of edges which cover e'. The data structure allows the following operations:

- $Initialize(P, E')$ Build a data structure for a partial path P with a set of covering edges E'.
- $Cover(P, e)$ Increase the cover value of each edge e' in P which is covered by e.
- $Uncover(P, e)$ Decrease the cover value of each edge e' in P which was covered by e.
- $Link(P_1, P_2, e)$ Link the data structures for P_1 and P_2 by the edge e. This is allowed if neither P_1 nor P_2 are subpaths of another path in the data structure.
- $Unlink(P)$ Undo the *Link* operation that created P. This is allowed if P is currently not linked with another path.
- $RightUncovered(P)$ Return the rightmost uncovered edge on P if it exists.

- $LeftUncovered(P)$ Return the leftmost uncovered edge on P if it exists.

Multiple edges are allowed, but not self-loops. A sequence of $Link$ and $Unlink$ operations results in a "linkage tree". Let d be the depth of this tree. In this section we describe an implementation of the data structure that takes constant time for $Link$ and $Unlink$; $O(d+\log|P|)$ time for $RightUncovered$, $LeftUncovered$, $Cover$, and $Uncover$; and $O(|P|+|E'|)$ time for $Initialize(P, E')$.

In their paper on dynamic trees [29] Sleator and Tarjan introduce a data structure for the dynamic maintenance of a collection of vertex-disjoint edge weighted paths. Each path p has a head and a tail. The data structure supports 11 kinds of operations. A subset of them is quoted below from [29]. The operations $path$, $head$, $tail$, $before$, and $after$ have the obvious meaning.

> $pmincost($**path** $p)$: Return the vertex v closest to $tail(p)$ such that $(v, after(v))$ has minimum cost among edges on p.
>
> $pupdate($**path** p, **real** $x)$: Add x to the cost of every edge on p.
>
> $reverse($**path** $p)$: Reverse the direction of p, making the head the tail and vice versa.
>
> $concatenate($**path** p, q, **real** $x)$: Combine p and q by adding the edge $(tail(p), head(q))$ of cost x. Return the combined path.
>
> $split($**vertex** $v)$: Divide $path(v)$ into (up to) three parts by deleting the edges incident to v. Return a list $[p, q, x, y]$, where p is the subpath consisting of all the vertices from $head(path(v))$ to $before(v)$, q is the subpath consisting of all vertices from $after(v)$ to $tail(path(v))$, x is the cost of the deleted edge $(before(v), v)$, and y is the cost of the deleted edge $(v, after(v))$. If v is originally the head of $path(v)$, p is null and x is undefined; if v is originally the tail of $path(v)$, q is null and y is undefined.

Every path in the dynamic path data structure is represented by a balanced binary tree whose leaves represent the vertices of the path, and whose internal nodes represent the edges of the path. At each internal node of such a tree a constant amount of local (weight) information is stored.

Every path in the extended dynamic path data structure is stored as a path or a subpath of a dynamic path data structure. The edge weights are the cover values. Whenever an operation (except $Link$ and $Unlink$) involves a path P that is a subpath of another path, we reconstruct P by a suitable sequence of $Unlink$ operations. After performing the operation we execute the corresponding $Link$ sequence.

- To execute $Initialize(P, E')$ we compute first the cover value for the edges of P by a left-to-right scan of P with each edge of E' stored at its endpoints in P. Then we build a dynamic tree data structure for P using the cover values as edge weights.

- We realize $Cover(P, (u, v))$ by using $split$, $pupdate$, and $concatenate$ as follows. W.l.o.g. assume that u is closer to $head(P)$ than v. If u is not the head of P then we split P at $before(u)$. If v is not the tail of P then we split the subpath containing u at $after(v)$. We add 1 to all edge weights in the subpath starting at u by using $pupdate$ and merge P together again using $concatenate$. Obviously, $Uncover(P, (u, v))$ can be realized in the same way, except that we subtract 1 instead of adding 1.

- To implement the $Link(P_1, P_2, e)$ operation we do not use the *concatenate* operation because we want to execute this operation in constant time. Instead we create a new node for e whose children are the roots of the data structures for P_1 and P_2. Afterwards we update the local information. An $Unlink(P)$ is the reversal of the $Link$ operation.
- A $LeftUncovered(P)$ query can be answered by using *pmincost*. If we want to answer a $RightUncovered(P)$ query we first execute $reverse(P)$, use $pmincost(P)$, and execute $reverse(P)$ again.

The running time of $Initialize(P, E')$ is $O(|P| + |E'|)$ since the scan can be executed in linear time and the dynamic tree for a path P with given edge weights can be built in time $O(|P|)$. A $Link$ or $Unlink$ operation takes constant time since, as shown in [29], the local information can be updated in constant time. Any of the other operations is enclosed in a sequence at most $2d$ $Unlink$ and $Link$ operations. The operation itself consists of a constant number of dynamic path operations which take time $O(\log |P|)$ giving a total of time $O(d + \log |P|)$. This shows the claimed bounds on the running times.

9 k-Edge Connectivity and k-Vertex Connectivity

Eppstein et al. [11] give a dynamic algorithm for k-edge connectivity with worst case update time $O(k^2 n \log(n/k))$, which we slightly modify in order to speed up the good case. It uses an algorithm by Gabow [14] for the static problem and the following lemma.

Lemma 9.1 [23,31] *Let G be a graph and $T_1 = U_1$ a spanning forest of G. Let T_i be a spanning forest of $G \setminus U_{i-1}$ and let U_i be $U_{i-1} \cup T_i$. Then G is k-edge connected if and only if U_k is k-edge connected.*

For notational convenience let U_0 be the empty graph. For each i we store $G \setminus U_{i-1}$ in the above minimum spanning tree data structure to maintain T_i. We choose U_k to be the suitable subgraph. If an update operation does not change U_k (good case) we incur amortized cost $O(k \log n)$. In the bad case we incur $O(k\sqrt{m} + k^2 n \log(n/k)) = O(k^2 n \log(n/k))$.

The size of the suitable subgraph in this case is $O(kn)$, so by Theorem 3.1 we get the following result.

Theorem 9.2 *There exists a data structure that answers the question whether the current graph is k-edge connected in constant time and that can be updated in amortized expected time $O(\min(1, kn/m)(k^2 n \log(n/k)))$ with respect to the rr-model.*

We discuss next how to test dynamically if the graph is k-vertex connected. Lemma 9.1 also holds for k-vertex connectivity provided that T_i is chosen to be a scan-first search forest of $G \setminus U_{i-1}$ [4, 23]. To quickly test for the good case we define a suitable subgraph S as follows: we number all vertices during a preprocessing phase with a unique label between 1 and n in an arbitrary, but fixed way. Then, we use the linear-time algorithm of [23] to find U_k. This algorithm sometimes makes arbitrary choices which vertex to select next. We make S unique by requiring that if more than one vertices can be selected, the algorithm has to use the one with the minimum label. Even with this additional requirement the

algorithm runs in time $O(m + n \log n)$. Thus, we can test if the insertion of an edge e forces S to change by running this algorithm on $S \cup e$ in time $O(kn + n \log n)$. If this is the case we can construct a new suitable subgraph S' by running this algorithm on $G \cup e$ in time $O(m + n \log n)$. Testing if a deletion changes S is obvious: If an edge of S is deleted, S has to be recomputed, otherwise nothing has to be done.

In the good case we are done. In the bad case we additionally might have to check whether the new suitable subgraph S' is k-vertex connected. For this purpose we use the (static) $O(k^3 n^{1.5} + k^2 n^2)$ time k-vertex algorithm by Galil [15]. This provides the following result.

Theorem 9.3 *There exists a data structure that answers the question whether the current graph is k-vertex connected in constant time and that can be updated in $O(\min(1, kn/m)(k^3 n^{1.5} + k^2 n^2))$ expected update time with respect to the rr-model.*

Conclusion

We present a general technique for analyzing dynamic graph algorithms in the average case setting. Note that this technique can also be used for analyzing the expected time of randomized incremental algorithms for static graph problems. There we have a worst case input graph and the algorithm works by maintaining a current solution while inserting the edges one by one in random order. In fact, backwards analysis first was used in computational geometry for exactly this purpose by Chew [5].

Note that our technique can also be used to analyze the average case performance of randomized dynamic graph algorithms. (A randomized algorithm is an algorithm that makes use of random choices for computing the solution to a worst case input.)

For the connectivity problems considered in this paper the running time of an update consists of two parts: an expected running time of $O(n/\sqrt{m} + \log n)$ (where m is the number of edges after the update) plus an amortized constant time for rebuilds. It is an interesting open question whether the data structure can be improved by distributing the costs of rebuilds over previous updates in a way that gives an expected time bound of $O(n/\sqrt{m} + \log n)$ per update.

Eppstein [9] suggested that a good average case behavior for some of the above problems can also be shown for node insertions and deletions.

Acknowledgments

The authors would like to thank Emo Welzl for helpful discussions.

References

[1] D. Alberts and M. Rauch Henzinger. Average case analysis of dynamic graph algorithms. In *Proc. 6th Symp. on Discrete Algorithms*, pages 312 – 321, 1995.

[2] H. Alt, K. Mehlhorn, H. Wagener, and E. Welzl. Congruence, similarity and symmetries of geometric objects. *Discrete Comp. Geom.*, 3:237 – 256, 1988.

[3] B. Bollobás. *Random Graphs.* Academic Press, London, 1985.

[4] J. Cheriyan, M. Y. Kao, and R. Thurimella. Algorithms for parallel k-vertex connectivity and sparse certificates. *SIAM J. Comput.*, 22:157 – 174, 1993.

[5] L. P. Chew. Building voronoi diagrams for convex polygons in linear expected time. CS Tech Report TR90-147, Dartmouth College, 1986.

[6] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comp. Geom.: Theory and Appl.*, 3:185 – 212, 1993.

[7] J. Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17:449 – 467, 1965.

[8] D. Eppstein. Average case analysis of dynamic geometric optimization. In *Proc. 5th Symp. on Discrete Algorithms*, pages 77 – 86, 1994.

[9] D. Eppstein. Personal communication, 1995.

[10] D. Eppstein, Z. Galil, and G. F. Italiano. Improved sparsification. Technical Report 93-20, Dept. of Inf. and Comp. Sc., Univ. of Calif., Irvine, CA 92717, 1993.

[11] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. In *Proc. 33rd Symp. on Foundations of Computer Science*, pages 60 – 69, 1992.

[12] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14:781 – 798, 1985.

[13] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *Proc. 32nd Symp. on Foundations of Computer Science*, pages 632 – 641, 1991.

[14] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. In *Proc. 23rd Symp. on Theory of Computing*, pages 112 – 122, 1991.

[15] Z. Galil. Finding the vertex connectivity of graphs. *SIAM J. Comput.*, 9:197 – 199, 1980.

[16] M. Rauch Henzinger. Fully dynamic cycle equivalence in graphs. In *Proc. 35th Symp. on Foundations of Computer Science*, pages 744 – 755, 1994.

[17] M. Rauch Henzinger and V. King. Randomized dynamic algorithms with polylogarithmic time per operation. To appear in *Proc. 27th Symp. on Theory of Computing*, 1995.

[18] R. M. Karp. personal communications.

[19] P. N. Klein and R. E. Tarjan. A linear-time algorithm for minimum spanning tree. In *Proc. 26th Symp. on Theory of Computing*, pages 9 – 15, 1994.

[20] L. Lovász and M. D. Plummer. *Matching Theory*, volume 29 of *Annals of Discrete Mathematics*. North-Holland, Amsterdam, 1986.

[21] S. Micali and V. Vazirani. An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st Symp. on Foundations of Computer Science*, pages 17 – 27, 1980.

[22] K. Mulmuley. Randomized, multidimensional search trees: dynamic sampling. In *Proc. 7th Symp. on Computational Geometry*, pages 121 – 131, 1991.

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

[24] M. H. Rauch. Fully dynamic biconnectivity in graphs. In *Proc. 33rd Symp. on Foundations of Computer Science*, pages 50 – 59, 1992.

[25] M. H. Rauch. Improved data structures for fully dynamic biconnectivity. In *Proc. 26th Symp. on Theory of Computing*, pages 686 – 695, 1994.

[26] J. H. Reif, P. G. Spirakis, and M. Yung. Re-randomization and average case analysis of fully dynamic graph algorithms. Alcom Technical Report TR 93.01.3.

[27] O. Schwarzkopf. *Dynamic Maintenance of Convex Polytopes and Related Structures.* PhD thesis, Freie Universität Berlin, 1992.

[28] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, pages 37 – 67. Springer Verlag, Berlin, 1993.

[29] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Sys. Sci.*, 26:362 – 391, 1983.

[30] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.

[31] R. Thurimella. *Techniques for the Design of Parallel Graph Algorithms.* PhD thesis, University of Texas, Austin, 1989.

[32] V. V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm. *Combinatorica*, 14(1):71 – 109, 1994.