Algorithmica (1998) 20: 31–60

Algorithmica

Average-Case Analysis of Dynamic Graph Algorithms

D. Alberts[1] and M. R. Henzinger[2]

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 $O(l \log n + \sum_{i=1}^{l} n/\sqrt{m_i})$ 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. Dynamic graph algorithm, Average-case analysis, Minimum spanning forest, Connectivity, Bipartiteness, Maximum matching.

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.)

[1] Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik, Kurt-Mothes-Strasse 1, 06099 Halle, Germany. alberts@informatik.uni-halle.de. Research supported by the Deutsche Forschungsgemeinschaft, Grant We 1265/2-1 (Graduiertenkolleg "Algorithmische Diskrete Mathematik"), and Grant We 1265/5-1 (Leibniz-Preis). This research was done in part while visiting the Max-Planck Institute for Computer Science, Im Stadtwald, 66123 Saarbrücken, Germany, and Cornell University, Ithaca, NY 14853, USA.
[2] Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301, USA. monika@pa.dec.com. This research was done in part while visiting the International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley, CA 94704, USA, and at the Max-Planck Institute for Computer Science, Im Stadtwald, 66123 Saarbrücken, Germany.

Received June 11, 1995; revised March 8, 1996. Communicated by K. Mehlhorn.

Two nodes u and v are *k-edge* (resp. *k-vertex*) *connected* for fixed k if there are k edge-disjoint (resp. k vertex-disjoint) paths between u and v. A query in the case of connectivity (resp. 2-edge connectivity) has two parameters u and v and returns "yes" if u and v are connected (resp. 2-edge connected). In the case of k-edge (resp. k-vertex) connectivity a query returns "yes" if the graph is k-edge (resp. 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, much work has been done on dynamic algorithms for various connectivity properties [10]–[13], [17], [27]–[29]. 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 $O(\log^2 n)$ (resp. $O(\log^3 n)$) per update [19], [18]. It is an open problem whether the connected or 2-edge connected components of a graph can be maintained deterministically faster than $O(\sqrt{n})$. A second interesting question is whether 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 described by Tarjan in [33] for each update operation achieves an update time $O(m)$. This was used, for example, in [2]. This is the only known improvement over recomputation from scratch which takes time $O(\sqrt{n}m)$ [24], [35].

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.

1.1. *Related Work.* Karp [20] 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 [29] a different random input model for dynamic graph algorithms is presented, called the *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 an fsgp, which assumes that insertions (deletions) occur with probability $1/2$. The algorithm, presented in [29], 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], [25], and [30]). 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.

1.2. *New Results*

- 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 using [21]. 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 [18].
- We show that a conceptually simple dynamic algorithm for maximum cardinality matching has an average update time of $O(n)$ with respect to the rr-model. The algorithm is based on the static maximum matching algorithm described in [33]. The space needed is linear and the preprocessing time is $O(\sqrt{n}m)$ using [24]. Additionally we give an insertions-only algorithm for maximum cardinality matching with $O(n)$ amortized time per insertion.

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 [15]. 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 [26] and the $O(k^3 n^{1.5} + k^2 n^2)$ minimum vertex cut algorithm by Galil [16]. 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 (except for preprocessing in the case of minimum spanning trees, but by increasing the running time by a factor of $\log(\log^* n)$ the algorithm can be made deterministic). 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 [31] in Section 3. As far as we know, this is the first application of backwards analysis to dynamic graph problems. In Section 4–9 we apply this technique to analyze the expected running time of dynamic algorithms for minimum spanning forest, connectivity, bipartiteness, 2-edge connectivity, maximum matching, 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 *et al*. [29] 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], [25], and [30]). 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 $\bar{E}\backslash E$. 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: (1) We can prescribe $\bar{E}$ and thus, e.g., force the graph to be bipartite, and (2) the adversary can give us an arbitrary sequence of updates, e.g., highly regular update patterns, like 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 [31] 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 the update operation fulfills certain conditions (that depend on the current graph). Of course, the update algorithm applies the fast algorithm whenever possible. To achieve a good expected time performance we show that the conditions for the fast algorithm are met by an update operation with a relatively high probability, i.e., that the probability for the slow algorithm is relatively low.

We explain the ideas for bounding the probability for the slow algorithm using the dynamic minimum spanning tree problem as an example:

Deletions: If a deletion does not remove an edge of the minimum spanning tree, the minimum spanning tree does not change and the update can be handled quickly (as we show in Section 4.2). Thus, if a deletion does not remove a minimum spanning tree edge, it fulfills the conditions for the fast algorithm. The probability that a randomly chosen edge of G is an edge of the minimum spanning tree is $(n-1)/m$. Thus, the probability that we have to use the slow algorithm is $(n-1)/m$.

Insertions: If the minimum spanning tree is still correct after the insertion of an edge e, then the conditions of the fast algorithm are fulfilled. If an insertion modifies the minimum spanning tree, the newly inserted edge e either (i) connects two disconnected pieces of G or (ii) the cost of e is less than the cost of an edge on the tree path connecting the endpoints of e. In both cases e belongs to each minimum spanning tree of $G \cup e$. Thus, the probability that we have to use the slow algorithm is the probability that a randomly chosen edge not in G fulfills (i) or (ii). Using the fact that E is a random subgraph of $\bar{E}$ and that e belongs to the minimum spanning tree of $G \cup e$, we argue below that the probability of this case is identical to the probability that a randomly chosen edge of $G \cup e$ belongs to the minimum spanning tree of $G \cup e$. The latter probability is $(n-1)/(m+1)$. Thus, the probability that we have to use the slow algorithm is $(n-1)/(m+1)$.

Let S denote a minimum spanning tree of the current graph. Note that we use only two facts to bound the probability of the slow algorithm:

- If a deletion does not delete an edge of S, then S is a valid minimum spanning tree in the new graph.
- If, after an insertion, S is no longer a valid minimum spanning tree for the new current graph, then every minimum spanning tree of the new current graph contains the new edge.

Thus, our strategy for bounding the probability of the slow algorithm is as follows: We choose for each graph G a set of subgraphs that we call *suitable* (defined below). The suitable subgraphs of G correspond to the minimum spanning trees of G in the above example. The algorithm maintains a suitable subgraph S of the current graph such that the following two conditions are fulfilled:

A. If a deletion does not delete an edge of S, then S is suitable in the new graph.
B. If, after an insertion, S is no longer suitable for the new current graph, then every suitable subgraph of the new current graph contains the new edge.

The fast algorithm is used when the update does not lead to a change in S. If Conditions A and B are fulfilled and the size of all suitable subgraphs is limited by some integer function $s(n)$, then we bound the probability of the slow algorithm by $s(n)/m$ (resp. $s(n)/(m+1)$) using the same arguments as for minimum spanning trees.

Let *Suit* be a function that maps every graph G on n vertices to a subset of the set of subgraphs of G. A set S is *suitable* for G if $S \in Suit(G)$. Conditions A and B put the following conditions on *Suit*, where e is an edge not in G:

A′. All sets in $Suit(G \cup \{e\})$ that do not contain e belong to $Suit(G)$.
B′. If there exists a set $S \in Suit(G)$ and $S \notin Suit(G \cup \{e\})$, then every set in $Suit(G \cup \{e\})$ contains e.
The latter is equivalent to saying:
If $Suit(G \cup \{e\})$ contains a set without e, then $Suit(G) \subseteq Suit(G \cup \{e\})$.

Combining the two conditions finally leads to the following condition on *Suit*:

C. Let e be an edge with $e \notin G$. If $Suit(G \cup \{e\})$ contains a set without e, then $\{S;\, S \in Suit(G \cup \{e\}) \text{ and } e \notin S\} \subseteq Suit(G) \subseteq Suit(G \cup \{e\})$.

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. Instead of repeating the average-case analysis for each dynamic graph problem in this paper, we give one average-case analysis that applies to any dynamic graph problem for which we can find a function *Suit* fulfilling Condition C.

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 $g(n, m)$ (resp. $b(n, m)$) be the running time of the fast (resp. 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 $T_{\text{del}}(n, m)$ 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 $T_{\text{del}}(E, e)$ 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

$$\begin{aligned} T_{\text{del}}(n, m) &= \frac{1}{\binom{\bar{m}}{m} m} \sum_{\substack{E \subset \bar{E} \\ |E| = m}} \sum_{e \in E} T_{\text{del}}(E, e) \\ &\leq \frac{1}{\binom{\bar{m}}{m} m} \sum_{\substack{E \subset \bar{E} \\ |E| = m}} s(n) b(n, m) + (m - s(n)) g(n, m) \\ &= O\left(\frac{s(n)}{m} b(n, m) + g(n, m)\right). \end{aligned}$$

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

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

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 afterward 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

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

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

$$\begin{aligned} T_{\text{ins}}(n, m) &\leq \frac{1}{\binom{\bar{m}}{m+1}(m+1)} \sum_{\substack{E' \subseteq \bar{E} \\ |E'|=m+1}} s(n)b(n, m) + (m+1-s(n))g(n, m) \\ &= O\left(\frac{s(n)}{m} b(n, m) + g(n, m)\right). \end{aligned}$$

This implies the following theorem.

THEOREM 3.1. *Let $\bar{G}$ be a graph on n vertices; let $\mathcal{P}$ be a dynamic graph problem such that a function Suit fulfilling Condition C exists; 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 largest if the graph is dense.

Using the same line of proof, 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. We do not include this for the sake of clarity, and since it is not needed for our applications.

4. Minimum Spanning Forests. Frederickson [12] introduced the topology tree data structure to maintain a minimum spanning forest dynamically. In this section we slightly 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]. This data structure is also the key data structure for the dynamic graph algorithms described in Sections 5–7.

To apply our technique of Section 3 we choose *Suit*(G) to consist of all minimum spanning trees of G. Additionally, we modify the topology trees such that updates involving nontree edges take time $O(\log n)$ plus amortized constant time for rebuilding parts of the data structure (good case), while the time for updates involving tree edges stays $O(\sqrt{m})$ (bad case), which is the bound of [12]. By Theorem 3.1 this results in 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 $\bar{G}$.

To guarantee that nontree edge updates are fast we make three modifications in the topology tree data structure: (1) We add a condition to the definition of a *restricted partition of order* k. This is necessary to guarantee that $\Omega(\sqrt{m})$ updates are executed before part of the data structure is rebuilt. (2) We add priority queues to the data structure to avoid that the minimum of $O(\sqrt{m})$ edge costs is recomputed from scratch after each update. (3) We remove some parts of the data structure at which no new information is stored. While the second modification leads immediately to an improvement, we show in Section 4.4 that the first modification leads to the desired amortized $O(1)$ rebuild time per update. The third modification is necessary to speed up updates in the good case.

Note that the running time of [12] can be reduced to $O(\sqrt{n})$ using improved sparsification [10], [11]. Sparsification is a technique which was designed to reduce the number of edges that a dynamic graph algorithm has to deal with from m to $O(n)$. This is accomplished by splitting the edge set into groups of size at most $2n$ and maintaining a spanning tree for each group. It follows that about half of the edges belong to the spanning tree of a group and, thus, are expensive to update. This implies that the probablity for a bad-case update is about $1/2$. Hence, combining sparsification with our approach does not improve the running time.

4.1. *Data Structure.* We first review parts of the data structure in [12], [13], 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 $d > 3$ by a cycle of d new vertices $x_1, \ldots, x_d$ in G'. The edges connecting x_i and x_{i+1} get a weight of $-\infty$, which implies that they always stay in the minimum spanning forest of G'. The edge connecting x_d and x_1 gets a weight of 0. Edges between the x_i nodes 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 there are $O(m)$ nodes in G' and that the edges of a minimum spanning forest of G are a "subset" of those for G'. We denote by T' the minimum spanning tree of G'. We describe next how the topology tree data structure achieve an $O(\sqrt{m})$ time per update operation. The topology tree data structure decomposes the vertex set of G' into sets, called *clusters*. The update algorithm spends time proportional to the size of $O(1)$ clusters plus the number of clusters. Initially the nodes are decomposed in a roughly balanced way such that each cluster contains at most $2k$ nodes and there are $O(m/k)$ clusters, for some parameter k. Choosing $k = \sqrt{m}$ gives an $O(k + m/k) = O(\sqrt{m})$ time update algorithm.

Adding edges can increase the number of nodes in a cluster (since the cycle representing a node of G can increase), deleting nodes can decrease the number of nodes in a cluster. By splitting and merging clusters the above roughly balanced decomposition is maintained and, thus, every update operation takes time $O(\sqrt{m})$.

We explain next the basic idea to reduce the time for updates in the good case to $O(\log n)$ plus $O(1)$ amortized time. Clusters are *created* and *deleted* in three ways: (A) If all the nodes in a cluster have been deleted, the cluster is deleted. (B) If a cluster is merged with another cluster, the two old clusters are deleted and a new cluster is created. (C) If a cluster is split, the cluster is deleted and two new clusters are created. A cluster is *created* (resp. *deleted*) by an update operation if it is created (resp. deleted) while processing the update.

Creating and deleting a cluster in Cases (B) and (C) takes time $O(k)$. Each update creates and deletes at most a constant number of clusters and incurs, thus, an $O(k)$ *rebuilding cost*. To achieve $O(k + m/k)$ update time in the bad case and $O(\log n)$ update time plus $O(1)$ amortized rebalancing time in the good case, we charge each bad-case update $O(k)$ rebuilding costs and we charge each good-case update $O(1)$ amortized rebuilding costs as follows: If the current update is a bad case, it is charged its $O(k)$ rebuilding cost. If the current update is a good case, but one of the cluster that it deletes was created by a bad-case update, the rebuilding cost of the current update is charged to this bad-case update. If the current update is a good case and none of the clusters that it deletes was created by a bad-case update, we guarantee that $\Omega(k)$ rebuilds have "contributed" to the cluster(s) deleted by the current update and amortize the rebuilding costs of the current update over them. This adds an amortized $O(1)$ rebuilding cost to every update. (All initial clusters are considered to be created by a bad-case update, since the cost of deleting them can be charged to the linear preprocessing time.)

For this amortization scheme to work we call some clusters *essential* and we maintain the following invariant:

(I) *Every cluster created by a good-case update consists of at most* $5k/3$ *nodes and, if it is essential, by at least* $k/2$ *nodes*.

As shown below, a cluster is deleted by a good-case update only if its size is either less than $k/3$ or more than $2k$. Since each update increases or decreases the size of a cluster by at most six nodes, it follows that in either case at least $k/18$ (namely, $k/2 - k/3$ or $2k - 5k/3$) updates have modified the size of the deleted cluster since the creation of the cluster. Amortizing the rebuilding costs of the good-case update over these updates adds an amortized $O(1)$ rebuilding cost to every update, since each update affects the size of only a constant number of clusters.

We give next the exact definitions. A *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. The *tree degree* of a cluster is the number of tree edges incident to the cluster. We call a cluster *essential* 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* (l, u)*-partition* 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 cluster with tree degree ≤ 3 and cardinality $\leq u$, and
(3) each essential cluster has cardinality at least l.

Our definition is a modification of the definition of a *restricted partition of order k* in [13]: Condition (3) is modified, since our amortized plan outline above would not apply: in the definition of [13] it is possible that two clusters C_1 and C_2 are merged and only $O(1)$ updates have occurred since the creation of C_1 and C_2. Thus, the $\Theta(k)$ rebuilding costs to delete C_1 and C_2 cannot be amortized over $\Omega(k)$ updates that occurred after the creation of C_1 and C_2.

Our algorithm maintains a dynamic $(k/3, 2k)$ partition subject to invariant I.

We say cluster C_2 is a *tree neighbor* of cluster C_1 if there exists a tree edge with one endpoint in C_1 and one endpoint in C_2. To initialize the partition we first use the procedure given in [13], which finds in linear time 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 cluster with tree degree ≤ 3 and cardinality $\leq k$, and
(3′) each essential cluster has a tree neighbor such that the combined cardinality of the two clusters is larger than k.

To fulfill (3), we join every essential cluster of size less than $k/3$ with its tree neighbor of Condition (3′) to create a cluster of size at least k and at most $4k/3$.

Given a dynamic $(k/3, 2k)$ partition, a *topology tree* is a binary tree of depth $O(\log n)$ whose leaves correspond to the clusters in the partition. An internal node C of a topology tree TT corresponds to a cluster of larger size that is formed by unifying the clusters corresponding to the leaves in the subtree of C in TT. The *level* of a leaf is 0, the level of an internal node is 1 plus the level of its children, which are all at the same level.

A *two-dimensional topology tree* is a tree of depth $O(\log n)$ whose leaves are pairs of clusters $C \times D$. Each leaf $C \times D$ is labeled with the minimum edge cost of an edge between C and D or $-\infty$ if no such edge exists. Each internal node has degree at most 4 and is labeled with the minimum label of its children. See [12] for a detailed definition.

The dynamic connectivity data structure of [12] consists of

- a *topology tree* TT,
- a *two-dimensional topology tree* $2TT$, and
- a dynamic tree data structure storing the minimum spanning tree T' of G'.

We modify the data structure as follows: (A) We omit some of the nodes of $2TT$ with label $-\infty$ together with their whole subtree. (This does not create problems in the query or update algorithm of [12] since these subtrees do not store the cost of an edge, i.e., do not contain any useful information for the algorithms.) (B) At each leaf $C \times D$ of $2TT$ we keep a priority queue of all nontree edges with one endpoint in C and one endpoint in D.

4.2. *Updates*. To update the data structure we make use of the following well-known lemma to split a cluster of size x into two clusters of size at most $2/3x$:

LEMMA 4.1 [22]. *Every n-vertex tree with degree at most* 3 *can be split into two subtrees, each with at most* $2/3n$ *vertices, by removing one edge.*

An update operation (a) tests if the good case or the bad case occurs and (b) executes the corresponding algorithm.

(a) The dynamic tree data structure that maintains T' is used (as in [12]) to decide which case occurs.
(b) The algorithm consists of three steps:
  (b1) *Updating the mapping from G to G', i.e., maintaining G' as a degree-*3 *graph*. This includes adding or removing the inserted or deleted edge and additional nodes and edges. Since it is not explicitely stated in [12], we give the details in Section 4.3. It takes constant time per update.
  (b2) *Updating the dynamic restricted partition and the structure of TT and $2TT$*. In the bad case we restore Conditions (1) and (2) as in [13], which modifies $O(1)$ clusters. Each of the resulting (at most constant) essential clusters of size less than $k/3$ is merged with neighboring clusters until its size is at least $k/3$ or it is no longer essential. If a resulting cluster contains more than $2k$ nodes, it is split into two clusters of size at least $2k/3$ and at most $4k/3$ using Lemma 4.1.
  Each step takes $O(k)$ time, which gives a total time of $O(k)$ for the bad case. Updating TT and the structure of $2TT$ whenever the dynamic restricted partition changes is identical to [12] and takes time $O(k)$ per update.
  The procedure for the good case is described in Section 4.4 and takes time $O(\log n)$ plus constant amortized time.
  (b3) *Updating the labels of $2TT$ and the dynamic tree*. For a bad-case update the algorithm consists of the algorithm in [12] plus the obvious updates of the priority queues.
  In the good case, let (x, y) be the edge that is updated, let C be the cluster containing x, and let D be the cluster containing y. The cost of (x, y) is added or removed from the heap of $C \times D$. If $min(C, D)$ changes, this change is propagated up the tree $2TT$, updating the labels of the ancestors of $2TT$. Since $2TT$ has depth $O(\log n)$, this takes time $O(\log n)$.

Summing the time for steps (a)–(b3) gives a total time of $O(k)$ for the bad case and $O(\log n)$ plus $O(1)$ amortized time for the good case.

4.3. *Updating the Mapping from G to G' in the Good Case*. We describe only the insertion of an edge (x, y)—a deletion is the inverse operation. We first update the node(s) representing x and the node(s) representing y and then we add the appropriate edge.

Let d be the degree of x before the insertion. We call x' the node representing x that will be incident to the new edge. To update the node(s) representing x the algorithm considers three cases:

$d < 3$: Set $x' = x$, since the node representing x is unchanged.
$d = 3$: Replace the node representing x by four nodes that are put into the same cluster as x and set $x' = x_3$. (In the case of a deletion x_1 and x_2 belong to the same cluster C in the good case. We replace x_1 to x_4 by a new node x which is put into C. This does not change the tree degree of any cluster.)
$d > 3$: Add a new node x_{d+1} between x_d and x_1 in the cycle representing x in G' and

add the node x_{d+1} to the cluster of x_d. Set $x' = x_{d+1}$. (In the case of a deletion the node x_i that is incident to (x, y) is removed and x_{i-1} and x_{i+1} are connected. This does not change the tree degree of the cluster containing x_i, since either x_{i-1} or x_{i+1} must belong to the cluster of x_i in the good case.)

The node y is processed in the same way. Finally a new edge (x', y') is added to G'.

Note that updating the mapping takes constant time and in the good case leaves the tree degree of all clusters unchanged.

4.4. *Updating the Dynamic Partition and the Structure of TT and $2TT$ in the Good Case.* The insertion or deletion of a node in G' might invalidate the partition by violating some of the conditions of the dynamic restricted partition. We restore Conditions (1)–(3) in this order such that fixing Condition (i) for $i = 2$ or 3 does not disturb the previously restored conditions.

Condition (1). Every tree-degree-3 cluster C with more than one node consists of at most four nodes, one with tree degree 3 and three with tree degree 2. To restore Condition (1), C is split: The tree-degree-3 node forms a new tree-degree-3 cluster. The remaining nodes are added to tree neighbors of C with tree degree 1 or 2, if this is possible. Otherwise, they are grouped into up to two clusters of constant size. See Figure 1.

Condition (2). If the cardinality $|C|$ of a cluster C is larger than $2k$, the cluster is split using Lemma 4.1.

Condition (3). An essential cluster of size less than $k/3$ is called a *violated* cluster. A good update can create at most two violated clusters, namely during an edge deletion. Restoring Conditions (1) and (2) does not create a violated cluster. Thus, in the good case the violated clusters have size at least $k/3 - 6$ and merging each violated cluster with a tree neighbor will result either in a cluster of size at least $k/2$ (if merged with another essential or violated cluster) or in a nonessential cluster (if merged with a nonessential

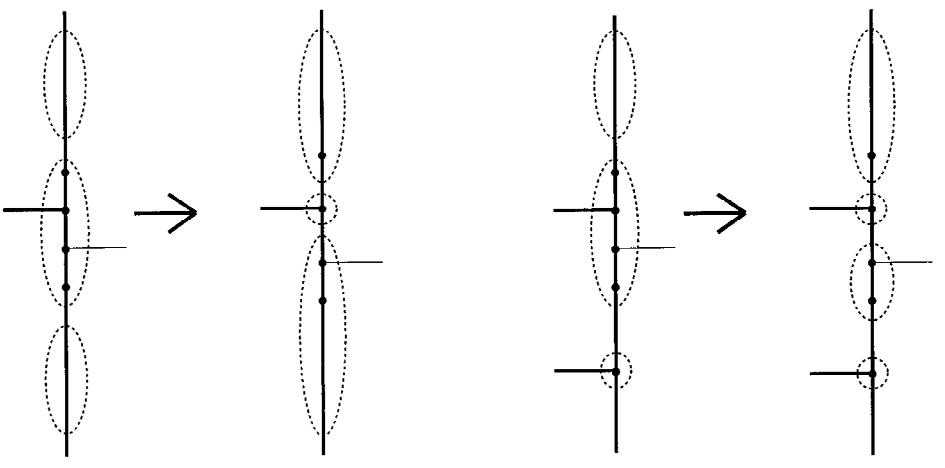

Fig. 1. Restoring Condition (1). Bold edges represent tree edges, dotted ellipses represent clusters.

cluster). If the cardinality of a new cluster is larger than $5k/3$, this cluster is split using Lemma 4.1.

The Structure of TT *and* $2TT$. We update TT as in [12]. If Conditions (2) or (3) had to be restored (i.e., the update already spent $O(k)$ time), we update $2TT$ as in [12]. If only Condition (1) had to be restored (i.e., the update spent only $O(1)$ time so far), the update added at most six constant-size clusters. For each such cluster C and each neighbor C' of C we add a leaf $C \times C'$ and its appropriate $O(\log n)$ ancestors to $2TT$. Note that we add a total of $O(\log n)$ nodes instead of a leaf $C \times D$ for *every* other cluster D and all the ancestors of these leaves. However, the omitted nodes of $2TT$ would be labeled with $-\infty$ and form subtrees of $2TT$. Thus, the resulting tree $2TT$ agrees with our modified definition of $2TT$.

LEMMA 4.2. *The updating algorithm maintains invariant I.*

PROOF. We have to show that every cluster created in the good case has size at most $5k/3$ and, if it is essential, at least $k/2$. The good-case update algorithm creates clusters when restoring Conditions (1)–(3). We check below that the invariant is maintained in every step.

When restoring Condition (1), no essential clusters are created and each created cluster has constant size. When Condition (2) is restored, the cardinality $|C|$ of the deleted cluster C is larger than $2k$ and at most $2k + 6$. Thus, the resulting clusters have size at most $4/3k + 4$ and at least $2k/3 + 1$. When Condition (3) is restored, each new essential cluster has size at least $k/2$ and at most $2k + k/3 - 1$. If it is larger than $5k/3$ it is split, resulting in two clusters of size at most $2/3(2k + k/3 - 1) < 5k/3$ and at least $1/3(5k/3) = 5k/9 > k/2$. Thus, in each of the three cases invariant I is maintained. $\square$

Next we analyse the running time of updating the dynamic partition and the structure of TT and $2TT$ in the good case. If only Condition (1) is restored, it takes $O(1)$ time to restore Condition (1), and time $O(\log n)$ to update the structure of TT and $2TT$.

If either Conditions (2) or (3) are restored, it takes time $O(k)$ to restore the conditions and update TT and $2TT$. In both cases if (one of) the deleted cluster(s) were created by a bad-case update, the $O(k)$ rebuilding cost are charged to this bad-case update. Only if (all) the deleted cluster(s) were created by a good-case update, the $O(k)$ rebuilding cost are amortized over previous updates: If Condition (2) is restored, the deleted cluster(s) consisted of at most $5k/3$ nodes at creation (by Invariant I) and now contains more than $2k$ nodes. If Condition (3) is restored, the deleted essential cluster(s) consisted of at least $k/2$ nodes at creation (by Invariant I) and now contains less than $k/3$ nodes. As described before, each update operation increases or decreases the size of a constant number of cluster by at most six nodes. Thus, in either case at least $k/18$ update operations must have increased (resp. decreased) the size of the deleted cluster(s). Amortizing the $O(k)$ rebuilding cost over these updates gives an amortized constant rebuilding cost per update.

4.5. *Final Result.* Choosing $k = O(\sqrt{m})$ gives a data structure that fulfills the following lemma, using the linear expected time algorithm for minimum spanning trees [21] during preprocessing.

LEMMA 4.3. *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 nontree edge is inserted or deleted. The data structure needs linear space and linear expected preprocessing time.*

If the weight for every edge in $\bar{G}$ is arbitrary but fixed 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 4.4. *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.*

5. Connectivity. To maintain connectivity dynamically the algorithm by Frederickson in [12] assigns cost 1 to edges in the current graphs and connects different connected components by cost 2 (dummy) edges. Queries can be answered in worst-case logarithmic time using the dynamic tree data structure representing T'. However, Frederickson describes an additional data structure which allows constant-time connectivity queries. Its update time is dominated by the update time of the dynamic minimum spanning forest data structure. Using the same approach with the minimum spanning forest data structure presented in the previous section gives the following result.

THEOREM 5.1. *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.*

6. Bipartiteness. In this section we analyze the average-case performance of an algorithm for dynamic bipartiteness due to Eppstein *et al*. [10], [11]. As in Section 5, we give each edge cost 1 and connect different connected components by dummy edges of cost 2. The basic idea is to maintain a spanning tree T of the graph G and additionally to maintain the parities of the cycles which are induced by the nontree edges. The graph is bipartite if and only if no nontree edge induces an odd cycle.

As in Section 4, we choose *Suit*(G) to consist of all minimum spanning trees of G. The minimum spanning tree T of G is maintained by creating a degree-3 graph G' and maintaining the minimum spanning tree T' of G' using a topology tree TT and a two-dimensional topology tree $2TT$.

6.1. *Data Structure.* For a nontree 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., dashed 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 data structure in [11] consists of

1. the MST T',
2. a topology tree TT where we store at each node C the distances between every pair of boundary vertices of C, and
3. the corresponding two-dimensional topology tree $2TT$. The nodes of $2TT$ are augmented with the following labels:

   Associated with each node 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.

We extend this data structure as follows to speed up updates in the good case.

1. We keep a dynamic tree data structure [32] of T' (for determining distances between nodes in T) giving dashed edges length 0 and nondashed edges length 1.
2. At each leaf $C \times D$ of $2TT$ we keep two lists, each one containing the nontree edges of G between C and D of the same parity.

6.2. *Updates*. An update operation (a) tests if the good case or the bad case occurs and (b) executes the corresponding algorithm.

(a) The dynamic tree data structure that maintains the minimum spanning tree T' is used (as in [12]) to decide which case occurs.

(b) The algorithm consists of three steps:

  (b1) *Updating the mapping from G to G', i.e., maintaining G' as a degree*-3 *graph*. See Section 4.3. It takes constant time per update.

  (b2) *Updating the dynamic restricted partition and the structure of TT and $2TT$*. The procedure for the bad case is described in Section 4.2 and takes time $O(\sqrt{m})$, the procedure for the good case is described in Section 4.4 and takes time $O(\log n)$ plus constant amortized time.

  (b3) *Updating the labels of $2TT$ and the dynamic tree*. In [11] it is shown that the worst-case update time for this data structure is $O(\sqrt{m})$. Our extensions only increase the running time by a constant factor. Thus, the update time in the bad case is $O(\sqrt{m})$.

  We show in Section 6.3 that updates in labels of $2TT$ and the dynamic tree takes time $O(\log n)$ plus constant amortized time in the good case.

Summing the time for steps (a)–(b3) gives a total time of $O(k)$ for the bad case and $O(\log n)$ plus $O(1)$ amortized time for the good case.

6.3. *Updating the Labels of $2TT$ and the Dynamic Tree in the Good Case*. We describe how to update in the good case the labels of $2TT$ and the dynamic tree data structure.

The Labels of $2TT$. If Conditions (2) or (3) are restored when updating the dynamic partition (see Section 4.4), then the labels are updated in time $O(\sqrt{m})$, as in the bad case. Otherwise at most $O(1)$ clusters of size $O(1)$ are created. The data structure for them and their ancestors in TT and $2TT$ can be built in time $O(\log n)$. We next show how to update the data structure of the remaining clusters in time $O(\log n)$. Amortizing the cost if Conditions (2) or (3) are restored as in Section 4.4 gives a running time of $O(\log n)$ plus $O(1)$ amortized time.

First assume that e is inserted. Let $u \in C$ and $v \in D$. We have to compute the parity class of e in order to insert it into the right list at the leaf node $C \times D$ in $2TT$. If $C = D$ we use the dynamic tree data structure to determine the parity of e and of the selected edges of $C \times C$. If $C \neq D$ we determine the distance of u (resp. v) to a boundary vertex of C (resp. D) by determining the number of nondashed 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 $C \times D$ (if they exist) in constant time using the distance information in the data structure and the following lemma shown in [11].

LEMMA 6.1. *Let C and D be any two clusters at the same level of the topology tree, and let f_1 and f_2 be any two nontree edges between C and D. Let w_C be a boundary vertex of C, and let w_D be a boundary vertex of D. Let j_1 and j_2 be respectively the endpoints of f_1 and f_2 in C and let r_1 and r_2 be respectively the endpoints of f_1 and f_2 in D. The two cycles λ_{f_1} and λ_{f_2} have the same parity if and only if the quantity $d(j_1, w_C) + d(j_2, w_C) + d(r_1, w_D) + d(r_2, w_D)$ is even.*

After determining the parity class of e we insert e in the appropriate list. This takes constant time. If the selected edges of $C \times D$ change, we percolate this change up in $2TT$. Since we can update each level in constant time using Lemma 6.1 the whole procedure takes time $O(\log n)$.

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

The Dynamic Tree. In the good case updating the mapping from G to G' changes a constant number of edges of T'. Each modification takes time $O(\log n)$.

6.4. *The Final Result.* The analysis for minimum spanning trees carries over, so we get the following theorem.

THEOREM 6.2. *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 $\underline{G}$ of size m_0 for any m_0, where m_i is the number of edges in G after operation i.*

7. 2-Edge Connectivity. Frederickson gives a data structure, called an *ambivalent data structure*, that answers 2-edge connectivity queries in time $O(\log n)$ [13]. It can be updated in time $O(\sqrt{m})$.

The basic idea is to maintain a spanning tree T of the graph G and *coverage information* for each tree edge. A tree edge e is *covered* if there exists a nontree 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 coverage information in various forms such that it can quickly find uncovered edges on any path in T.

We modify the ambivalent data structure and its update algorithm in order to speed up the good case.

7.1. *Data Structure.* We first describe the data structure of [13] and then give our modifications. As in Section 4, the algorithm gives each edge G cost 1 and connects G by dummy edges of cost 2. We choose $Suit(G)$ to consist of all minimum spanning trees of G. The algorithm creates a degree-3 graph G' and maintains a minimum spanning tree T' of G' in a topology TT and a two-dimensional topology $2TT$.

The algorithm partitions the edges of T' into chains, called *complete paths*, for which it keeps coverage information. Subpaths of complete paths are called *partial paths*. They are used to compute coverage information for edges on complete paths efficiently and to answer coverage queries about parts of complete paths.

Each cluster in the partition, i.e., each leaf of TT has an associated partial path, but no complete path, and each internal node of TT has either an associated partial or an associated complete path. The path associated with a cluster C is a subpath of the spanning tree T', formed by edges of C. See [13] for the definition of complete and partial paths.

For a node $u \in C$, let $proj(u)$ be the node on the partial path of C that is closest to u in T' 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 $dist(u, e)$ over all nodes $u \in C$ that are connected by a nontree edge to a node in D,
- by $maxcover\_node(C, D, e)$ a node u such that $dist(u, e) = maxcover(C, D, e)$, and
- by $maxcover\_edge(C, D, e)$ a nontree edge between C and D that is incident to $maxcover\_node(C, D, e)$.

The ambivalent data structure consists of:

1. An MST T'.
2. The partial and complete paths represented in binary trees.
3. A topology tree TT for T', extended with the following labels:
   (A) At each leaf C of TT the algorithm stores the following labels:
      (a) It stores a value *disttobr* for each node $u \in C$: In a graph that only contains T and the nontree edges incident to C, *disttobr* contains the number of edges (in T) from u to the closest bridge on the path from u to (but excluding) the partial path of C if such a bridge exists and ∞ otherwise.
      (b) It also keeps a least common ancestor data structure in T' for nodes of C rooted at an arbitrary boundary vertex of C.
   (B) For each node C of TT the data structure keeps
      (a) a pointer to the partial or complete path of C,
      (b) the *length* of the partial path of C (if it exists),

(c) a value *toptobr*, which is the number of edges (in T') from a fixed endpoint of the complete path associated with C to the closest bridge on the complete path (if it exists), and
(d) additional values that do not change in the good case and that can be created in time linear in the size of C.

4. The corresponding two-dimensional topology tree $2TT$. The nodes of $2TT$ are labeled with the following values:
   (A) At each leaf $C \times D$ with $C \neq D$ it keeps for each tree edge e incident to C the value $maxcover(C, D, e)$.
   (B) At each internal node of $2TT$ it keeps a constant number of *maxcover* values. These values are computed in constant time from the *maxcover* values of its children. In this way, for each pair (c, D) of nodes on the same level of TT and for each tree edge incident to e, a $maxcover(C, D, e)$ value is computed.

We modify the data structure as follows:

1. *Extended Dynamic Path Data Structure.* Inserting or deleting nontree edges can change the coverage information at $\Omega(\sqrt{m})$ leaves of the binary trees representing partial and complete paths. To avoid this cost, we maintain all partial and complete paths in a new data structure, called the *extended dynamic path data structure*. We present the interface of the data structure next and give its implementation in Section 7.5.

The extended dynamic path data structure extends the dynamic path data structure of [32]. It represents a set of paths such that two paths are either vertex-disjoint or one path is contained in the other one.[3] Note that each edge on one of the paths is represented just once, since a path P_1 contained in a path P_2 shares parts of the data structure of P_2. There is a unique *cover value* associated to each edge e', counting the number of edges which cover e'.

The data structure supports 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.
- $Add(P, x, y)$: Replace the edge (x, y) of P by the edges (x, z) and (z, y), where z is a new node that does not appear on any path. The cost of both new edges is equal to the cost of (x, y).

---

[3] The definition of complete paths in [13] does not make them vertex-disjoint: 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.

- *Remove*(P, z): Remove the two edges (x, z) and (z, y) of P and add the new edge (x, y). The operation demands that the cost of the two removed edges be identical. The cost of the new edge is the cost of a removed edge.

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*, *Uncover*, *Add*, and *Remove*; and $O(|P| + |E'|)$ time for *Initialize*(P, E'). Since d is $O(\log n)$ in our application *RightUncovered*, *LeftUncovered*, *Cover*, *Uncover*, *Add*, and *Remove* take time $O(\log n)$.

We use this data structure to maintain the complete and partial paths together with their coverage information. An edge e on a partial or complete path P is covered in the extended dynamic path data structure iff it is covered in the binary tree representation of [13]. Expressed more formally, the cover value of e is larger than 0 in the extended dynamic path data structure iff the *somecov* value of an ancestor of e is set to 1 in the binary tree representation of P.

2. *Labeled Dynamic Tree.* This data structure is used for three different reasons: (i) It replaces the *disttobr* of 3(A)(a). (ii) It replaces the least common ancest data structure of 3(A)(b). (iii) It computes a *dist*(u, e) value in time $O(\log n)$ instead of $O(\sqrt{m})$.

(i) We do not store the *disttobr* values, since one good-case update might change $\Omega(\sqrt{m})$ *disttobr* values. Instead we store the spanning tree of T' in a dynamic tree data structure [32] and keep for each tree edge e in C a *cover-counter*: If e is not on the partial path of C, its cover-counter counts the number of nontree edges incident to C that cover e. If e is on the partial path of C, its cover-counter is always 1. Determining the *disttobr* value of a node u corresponds to a *findmin*-query in the dynamic tree data structure to determine the bridge nearest to u and to determine the length of the path from u to this bridge.[4]

A constant number of *disttobr* values change during an update operation. The new values can be computed in time $O(\log n)$ in the modified data structure, as opposed to $O(\sqrt{m})$ in the original data structure. The *disttobr* values are used during a 2-edge connectivity query. However, each query only needs to know the value of a constant number of *disttobr* values, which takes time $O(\log n)$ using our data structure. Thus, our data structure does not increase the query time of $O(\log n)$.

(ii) We do not store the least common ancestor data structure in T', since even good-case updates might change a constant number of edges of T' (see Section 4.3). Instead we use the above dynamic tree data structure to answer least common ancestor queries in time $O(\log n)$. As described for *disttobr* values, this does not increase the query time, since only a constant number of least common ancestor queries are asked during a 2-edge connectivity query. It also reduces the time to update the least common ancestor information to $O(\log n)$.

(iii) Given a new nontree edge (u, v) with u in the level-0 cluster C and $v \notin C$, a slight variant of this data structure can also be used to compute *proj*(u) and to compute

[4] The latter can be done with a straightforward extension of the dynamic tree data structure in time $O(\log n)$.

dist(u, e) for each tree edge e incident to C. It takes time $O(\log n)$. We leave the details to the reader.

3. *Max-heaps*. We do not keep *maxcover* values, but instead the corresponding *maxcover_edge* at each node of $2TT$. While the data structure in [13] used *maxcover* values to cover paths, our algorithm uses *maxcover_edges* instead.

Storing the edge instead of the value has the following advantage: Even during good-case updates, edges can be added to or removed from a partial or complete path when updating the mapping from G to G' and the dynamic partition. Thus, the *maxcover* value becomes outdated, while the *maxcover_edge* and the relative order of the nontree edges incident to a cluster in the "*maxcover*-order" does not becomes outdated.

Note that for an internal node with a partial path of $2TT$ its *maxcover_edges* can be computed in time $O(1)$ from the *maxcover_edges* of its children. (i) To determine quickly the *maxcover_edge* at a leaf of $2TT$ we keep *max*-heaps at leaves of $2TT$. (ii) We also keep them at internal nodes of TT with complete paths to speed up updating their coverage information.

(i) At each leaf $C \times D$ with $C \neq D$ of $2TT$ we keep for each tree edge e incident to C a heap $max(C, D, e)$ that contains all nontree edges (u, v) with $u \in C$ and $v \in D$ in the order of the *dist*(u, e) values. The maximum element of the heap is the *maxcover_edge*(C, D, e).

(ii) If a node C of TT has a complete path, it has a degree-1 child C_1 in TT (see [13]). Let e be the tree edge incident to C_1. For all cluster $D \neq C_1$ on the same level as C_1, the heap $max(C)$ contains all nontree edges (u, v) with $u \in C_1$ and $v \in D$ in the order of the *dist*(u, e) values. The algorithm of [13] recomputes this value, which is a maximum of $O(\sqrt{m})$ numbers, from scratch after each update. We avoid this by adding the heap.

7.2. *Updates*. We now describe the modified update algorithm. As in Section 4.2 an update executes steps (a)–(b3). Steps (a)–(b2) are identical to Section 4.2. Step (b3) updates the partial and complete paths, the labels of TT, the labels of $2TT$, and the dynamic tree of T. In the bad case it updates the labels in the original data structure as in [13] and it updates the new labels of the modified data structure in time $O(\sqrt{m})$ in the straightforward way. The partial and complete paths are updated using the same operations as in [13], but using our new data structure instead of the binary tree data structure. For each operation, its running time matches the running time of the binary tree representation.

The algorithm for step (b3) in the good case is given in Section 7.3. Step (b3) takes time $O(\sqrt{m})$ for the bad case and $O(\log n)$ plus $O(1)$ amortized time for the good case.

Summing the time for steps (a)–(b3) gives a total time of $O(\sqrt{m})$ for the bad case and $O(\log n)$ plus $O(1)$ amortized time for the good case.

7.3. *Updating the Partial and Complete Paths, the Labels of TT and $2TT$, and the Dynamic Tree of T' in the Good Case.* If Conditions (2) or (3) are restored when updating the dynamic partition, (see Section 4.4), then the partial and complete paths, the labels of TT and $2TT$, and the dynamic tree of T are updated in time $O(\sqrt{m})$, as in the bad case. The costs are amortized as discussed before and contribute an $O(1)$

amortized cost to each update. Otherwise, there are at most six new clusters, each of constant size. The data structures for them and their ancestors can be created in time $O(\log n)$. We show that each part of the data structure for the old clusters can be updated in time $O(\log n)$.

Let $C_i(x)$ be the level-i cluster containing x. We consider the insertion or deletion of an edge (u, v). The only labels that have to be updated are the labels of clusters (at various levels) containing u or v. We achieve $O(\log n)$ update time, since there are $O(\log n)$ such clusters at which we spend $O(1)$ time each, and there are $O(1)$ clusters at which we spend $O(\log n)$ time.

The Partial and Complete Paths. We denote by $PP(C)$ the partial path of cluster C and by $CP(C)$ the complete path of cluster C. Let C_u be the least ancestor of $C_0(u)$ in TT with associated complete path. If edges are added to or removed from the partial or complete path of a cluster C', then there exists a level-0 cluster C such that the edges are also added to or removed from the partial path of C. The algorithm that updates the partial path data structure of C also updates the partial path of C', by data structure sharing.

When edges are added to the partial path, the algorithm first executes *Unlink* operations until the resulting partial path corresponds to the partial path of the level-0 cluster C. Then it executes an *Add* operation. Finally it executes the steps below to add the nontree edge. When edges are removed from the partial path, the algorithm first executes the steps below to remove the nontree edge. Then it executes *Unlink* operations until the resulting partial path is the partial path of a level-0 cluster. Since each pair of edges to be removed from the partial path by a *Remove* operation has the same cost, they are finally removed by a *Remove* operation.

The coverage information of at most two partial or complete paths needs to be updated when a nontree edge (u, v) is inserted or deleted. Which paths have to be updated depends on u and v. We distinguish three cases:

(i) If u and v are contained in the same level-0 cluster C and the update is an insertion, then we execute $\mathit{Cover}(PP(C), (\mathit{proj}(u), \mathit{proj}(v)))$. If they are in the same level-0 cluster and the update is a deletion we execute $\mathit{Uncover}(PP(C), (\mathit{proj}(u), \mathit{proj}(v)))$.

(ii) If u and v are not contained in the same level-0 cluster, but $C_u = C_w$, 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_edges* that have changed and are used to cover a partial or complete path are $\mathit{maxcover\_edge}(C_i(u), C_i(v), e)$ and $\mathit{maxcover\_edge}(C_i(v), C_i(u), e)$, where e is the tree edge connecting $C_i(u)$ and $C_i(v)$. Let $m(u)$ (resp. $m(v)$) denote the former value of $\mathit{maxcover\_edge}(C_i(u), C_i(v), e)$ (resp. $\mathit{maxcover\_edge}(C_i(v), C_i(u), e)$), and let $m'(u)$ and $m'(v)$ be the current edges. We execute first $\mathit{Uncover}(PP(C_{i+1}(u)), m(u))$ and $\mathit{Uncover}(PP(C_{i+1}(u)), m(v))$, and then $\mathit{Cover}(PP(C_{i+1}(u)), m'(u))$ and $\mathit{Cover}(PP(C_{i+1}(u)), m'(v))$.

(iii) If $C_u \neq C_v$, then the maximum *maxcover_edge* in $\mathit{max}(C_u)$ is the only *maxcover_edge* that has changed and is used to cover a partial or complete path (namely, the complete path of C_u). Thus, we uncover $CP(C_u)$ from the old maximum element of $\mathit{max}(C_u)$ and cover it with the new maximum element of $\mathit{max}(C_u)$. We do the same for same for v.

The Labels of TT. (A) We update the labeled dynamic trees of $C_0(u)$ and of $C_0(v)$ by adding a constant number of edges with the appropriate cover counter. If $proj(u) = proj(v)$ (and thus $C_0(u) = C_0(v)$) 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 $proj(v)$. Either case takes time $O(\log n)$.

(B) We discuss the items in the order of Section 7.1.

(b) If tree edges are added to the partial path of $C_0(u)$ or $C_0(v)$, then their *length* values are updated. To update their ancestors, the changes are percolated up the tree.
(c) For a cluster C the $toptobr(C)$ value can be computed in time $O(\log n)$ using the data structure for the complete path of C. Since at most two complete paths are affected by the update, updating all *toptobr* values takes time $O(\log n)$.
(d) Instead of the least common ancestor data structure, we update the dynamic trees of $C_0(u)$ and $C_0(v)$ as described in (A).

The Labels of $2TT$. (A) 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 (u, v) from the heap $max(C_0(u), C_0(v), e)$ determines the new value of $maxcover\_edge(C_0(u), C_0(v), e)$ in time $O(\log n)$. Since at most four heaps are affected, updating all *maxcover_edge* values at level-0 clusters takes time $O(\log n)$.

(B) Each *maxcover_edge* of an internal node of $2TT$ can be computed in constant time from the *maxcover_edges* of its children. Since $2TT$ has depth $O(\log n)$, all *maxcover_edges* can be updated in time $O(\log n)$.

Additionally. The only *max*-heaps of internal nodes that change are the heaps of C_u and C_v. To update $max(C_u)$ and $max(C_v)$ we delete the old *maxcover_edge* of the corresponding tree-degree-1 child and insert the new one if the value has actually changed. This takes time $O(\log n)$.

The Dynamic Tree. In the good case updating the mapping from G to G' changes a constant number of edges of T'. Each modification takes time $O(\log n)$.

This shows that the data structure can be updated in time $O(\log n)$ plus $O(1)$ amortized time in the good case.

7.4. *Final Result.* Using the analysis of Section 3 gives the following theorem.

THEOREM 7.1. *There exists a dynamic data structure that answers* 2*-edge connectivity queries in time* $O(\log n)$ *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, where m_i is the number of edges in G after operation i.*

7.5. *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 [32].

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. Each path has a leftmost degree-1 vertex (also called the *head*) and a rightmost degree-1 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.
- *Add*(P, x, y): Replace the edge (x, y) of P by the edges (x, z) and (z, y), where z is a new node that does not appear on any path. The cost of both new edges is equal to the cost of (x, y).
- *Remove*(P, z): Remove the two edges (x, z) and (z, y) of P and add the new edge (x, y). The operation demands that the cost of the two removed edges is identical. The cost of the new edge is the cost of a removed edge.

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*, *Uncover*, *Add*, and *Remove*; and $O(|P| + |E'|)$ time for *Initialize*(P, E').

In their paper on dynamic trees [32] 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 [32]. 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 $(v, \textit{after}(v))$ 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 $(\textit{tail}(p), \textit{head}(q))$ 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 $(\textit{before}(v), v)$, and y is the cost of the deleted edge $(v, \textit{after}(v))$. 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 first compute 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. Without loss of generality 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. Afterward 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.
- We realize *Add*(P, x, y) by the following sequence of operations. First, we *split* P at x. This returns (up to) two paths and weights as described above. Then we *concatenate* x to the path ending at its former predecessor again (if it existed) using the corresponding weight which was returned by *split*. We create a new path consisting only of z, and *concatenate* it with the paths ending at x and starting at y with the weight of the edge (x, y) which was returned by *split* as well.
- The operation *Remove*(P, z) is realized by a *split* at z followed by a *concatenate* operation for the two paths returned by *split* with one of the two (identical) weights returned by *split*.

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 [32], the local information can be updated in constant time. Any of the other operations is enclosed in a sequence of 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.

8. Maximum Cardinality Matching. Unlike minimum spanning tree and connectivity, the dynamic maximum matching problem is not solvable using sparsification [10],

[11], because there are no nontrivial certificates. However, there are sparse suitable subgraphs, so this problem reveals an interesting difference between the otherwise similar concepts of certificates and suitable subgraphs.

Using just one phase of a static maximum cardinality matching algorithm per update leads to a dynamic algorithm with $O(n+m)$ worst-case update time (see, e.g., [2]). This is still the best known algorithm. In the following we show that a variant of this simple approach yields a bound of $O(n)$ expected time for inputs which are random according to the rr-model.

8.1. *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 alternate between being in the matching M and not being in M as we walk along P. We drop the phrase "with respect to M" whenever there are no ambiguities. A *free vertex* is a vertex which is not incident to 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 nonmatching 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 similarly to [23]. 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 an arbitrary number of edges can be added 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,[5] then v is an *even vertex*. 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 [7]. A nonreachable vertex is called an *out-of-forest vertex.*

[5] The length of a path is the number of edges it contains.

8.2. *Data Structure and Suit*(G). 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. 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], [23], and [33].

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. 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 [23]. 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.

We define the set *Suit*(G) as follows. An element of *Suit*(G) is a maximum matching of the current graph unioned with a blossom forest with respect to this matching. It follows that $s(n) = O(n)$. We show next that the mapping *Suit* meets the requirements for Theorem 3.1.

LEMMA 8.1. *The mapping Suit as defined above fulfills Condition C* (*see p*. 36).

PROOF. It is equivalent to show that Conditions A and B hold (see p. 35). Let G be the current graph. Let S be the current suitable subgraph, consisting of the union of the current maximum matching M, and a blossom forest B with respect to M.

We begin with Condition A. Assume that we delete an edge e which does not belong to S. Since e is not in M, its deletion does not decrease the matching number. Thus M is maximum in $G - \{e\}$. Since e is not in B, its deletion has no influence on the parities of the vertices. Thus B is a blossom forest with respect to M for $G - \{e\}$, too. Hence, the union $M \cup B$ is a member of *Suit*$(G - \{e\})$.

In order to show Condition B, suppose that 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:

(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. This is 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, but it does not change the matching number.* 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 neither changes the matching number nor the number of reachable vertices, but it 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.

In all three cases where S is no longer suitable after the insertion of the new edge e, e has to be part of any new suitable subgraph. Thus, Condition B holds. ☐

8.3. *Updates*. It is easy to detect whether an update implies a change in the suitable subgraph (the bad case) or not. In case of a deletion, this is done using the labels of the edges. In case of an insertion, we can check whether one of the three conditions mentioned in Lemma 8.1 applies by using the parity information and the tree pointers at the vertices. In both cases this can be done in constant time.

Tarjan [33] describes a static algorithm for computing a maximum matching in general graphs. This algorithm is a variant of Gabow's earlier implementation [14] of Edmond's algorithm [7]. 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.

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 [24], [35] 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 8.2. *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.*

8.4. *Insertions Only.* We give 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.

Each phase of Tarjan's algorithm scans the edges in G in arbitrary order until an augmenting path is found. Scanning them in the order of insertion leads immediately to a semidynamic algorithm. Whenever an insertion creates an augmenting path, the data structure is rebuilt. A sequence of insertions between two rebuilds corresponds to one phase of Tarjan's algorithm. 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 afterward, takes time $O(n + m)$. Since there are at most $n/2$ phases, the total time is $O((n + m)n)$, i.e., the amortized time per insertion is $O(n)$, provided the algorithm is started with an empty edge set.

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 [15] for the static problem and the following lemma.

Let G be a graph and let $T_1 = U_1$ be a spanning forest of G. Let T_i be a spanning forest of $G \backslash U_{i-1}$ and let U_i be $U_{i-1} \cup T_i$. Then U_k is called a *sparse k-edge connectivity certificate* for G.

LEMMA 9.1 [26], [34]. *Let G be a graph and let U_k be a sparse k-edge connectivity certificate for G. 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 $G \backslash U_{i-1}$ in the above minimum spanning tree data structure to maintain T_i. We choose $Suit(G)$ to be the set of all sparse k-edge connectivity certificates of G. 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* $O(\min(1, kn/m)(k^2 n \log(n/k)))$ *amortized expected time 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 $G \backslash U_{i-1}$ [4], [26]. To test quickly for the good case we define the *smallest sparse k-edge connectivity certificate* 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 [26] to find U_k. This algorithm sometimes makes arbitrary choices of which vertex to select next. We require that if more than one vertex can be selected, the algorithm has to use the one with the minimum label. The resulting sparse k-edge connectivity certificate S_k is called the smallest sparse k-edge connectivity certificate. We choose $Suit(G)$ to be the unique smallest sparse k-edge connectivity certificate S_k of G.

Note that even with this additional requirement the algorithm of [26] runs in time $O(m + n \log n)$. Thus, we can test if the insertion of an edge e is a good case or a bad case by running this algorithm on $S_k \cup e$ in time $O(kn + n \log n)$. If this is the case we can construct a new suitable subgraph S'_k by running this algorithm on $G \cup e$ in time $O(m + n \log n)$. Testing if a deletion changes S_k is obvious: if an edge of S_k is deleted, S_k 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'_k is k-vertex connected. For this purpose we use the (static) $O(k^3n^{1.5} + k^2n^2)$ time k-vertex algorithm by Galil [16]. 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* $O(\min(1, kn/m)(k^3n^{1.5} + k^2n^2))$ *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 was first 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. R. Henzinger. Average case analysis of dynamic graph algorithms. In *Proc*. 6*th 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 Comput*. *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, Hanover, NH, 1986.

[6] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput*. *Geom*. *Theory 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*. 5*th 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 Information and Computer Science, University of California, Irvine, CA, 1993.

[11] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. In *Proc*. 33*rd 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*. 32*nd Symp*. *on Foundations of Computer Science*, pages 632–641, 1991.

[14] H. N. Gabow. Implementation of algorithms for maximum matching on nonbipartite graphs. Ph.D. thesis, Dept. of Computer Science, Stanford University, Stanford, CA, 1973.

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

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

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

[18] M. R. Henzinger and V. King. Randomized dynamic algorithms with polylogarithmic time per operation. In *Proc*. 27*th Symp*. *on Theory of Computing*, pages 519–527, 1995.

[19] M. R. Henzinger and M. Thorup. Improved sampling with applications to dynamic graph algorithms. In *Proc*. *ICALP* '96, pages 290–299, 1996.

[20] R. M. Karp. Personal communications.

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

[22] P. M. Lewis, R. E. Stearns, and J. Hartmanis. Memory bounds for recognition of context-free and context-sensitve languages. In *Proc*. *IEEE Conf. on Switching Theory and Logical Design*, pages 191–202, 1965.

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

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

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

[26] 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.

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

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

[29] 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.

[30] O. Schwarzkopf. Dynamic maintenance of convex polytopes and related structures. Ph.D. thesis, Freie Universität Berlin, Berlin, 1992.

[31] 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.

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

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

[34] R. Thurimella. Techniques for the design of parallel graph algorithms. Ph.D. thesis, University of Texas, Austin, TX, 1989.

[35] 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.