Dynamic Graph Algorithms

David Eppstein[*] Zvi Galil[†] Giuseppe F. Italiano[‡]

1 Introduction

In many applications of graph algorithms, including communication networks, graphics, assembly planning, and VLSI design, graphs are subject to discrete changes, such as additions or deletions of edges or vertices. In the last decade there has been a growing interest in such dynamically changing graphs, and a whole body of algorithms and data structures for dynamic graphs has been discovered. This chapter is intended as an overview of this field.

In a typical dynamic graph problem one would like to answer queries on graphs that are undergoing a sequence of updates, for instance, insertions and deletions of edges and vertices. The goal of a dynamic graph algorithm is to update efficiently the solution of a problem after dynamic changes, rather than having to recompute it from scratch each time. Given their powerful versatility, it is not surprising that dynamic algorithms and dynamic data structures are often more difficult to design and analyze than their static counterparts.

We can classify dynamic graph problems according to the types of updates allowed. A problem is said to be *fully dynamic* if the update operations include unrestricted insertions and deletions of edges. A problem is called *partially dynamic* if only one type of update, either insertions or deletions, is allowed. If only insertions are allowed, the problem is called *incremental*; if only deletions are allowed it is called *decremental.*

In this chapter, we focus our attention to dynamic algorithms for undirected graphs, which were studied more extensively than dynamic algorithms for directed graphs. Indeed, designing efficient fully dynamic data structures for directed graphs has turned out to be an extremely difficult task.

[*]University of California, Irvine, CA 92717. Supported in part by NSF Grant CCR-9258355.

[†]Columbia University, New York, NY 10027. Supported in part by NSF Grant CCR-9316209 and the CISE Institutional Infrastructure Grant CDA-9024735.

[‡]University of Venice "Ca' Foscari", Venice, Italy. Supported in part by the ESPRIT LTR Project no. 20244 (ALCOM-IT) and by a Research Grant from University of Venice "Ca' Foscari".

Most of the efficient data structures available for directed graphs are partially dynamic [2, 13, 29, 30, 31, 37, 39, 43, 53], and only preliminary results are available for fully dynamic problems [25]. For this reason, an alternative viewpoint that has been proposed is to measure the complexity of a dynamic algorithm as a function of the *output change* [17, 40]. The main dynamic problems considered on directed graphs include shortest paths and transitive closure. For lack of space, we do not include in this chapter dynamic algorithms for planar graphs, which have received considerable attention in recent years [6, 7, 11, 12, 18, 21, 22, 28, 32, 34, 42, 46, 47, 48, 51], and focus our attention to general undirected graphs only.

The remainder of the chapter is organized as follows. In Section 2 we give some preliminary definitions and a little terminology. Dynamic tree problems are considered in Section 3, while in Section 4 we describe partially dynamic algorithms for undirected graphs. Fully dynamic algorithms for undirected graphs are described in Section 5. Finally, in Section 6 we describe some open problems.

2 Preliminary Definitions

Given an undirected graph G with non–negative edge weights, the *minimum spanning forest of G* is the subgraph of minimum total weight that has the same connected components as the original graph. Whenever G is connected, this forest consists of a unique tree, and we refer to this tree as a minimum spanning tree of G. Note that a minimum spanning forest, or a mininum spanning tree is not necessarily unique. It is well known that a minimum spanning forest can be computed by either of two dual greedy algorithms, based on the following properties:

Cut Property: Add edges one at a time to the spanning forest until it spans the graph. At each step, find a cut in the graph that contains no edges of the current forest, and add the edge with lowest weight crossing the cut.

Cycle Property: Remove edges one at a time from the graph until only a forest is left. At each step, find a cycle in the remaining graph and remove the edge with the highest weight in the cycle.

Given an undirected graph $G = (V, E)$, and an integer $k \geq 2$, a pair of vertices $\langle u, v \rangle$ is said to be *k–edge–connected* if the removal of any $(k-1)$ edges in G leaves u and v connected. This is an equivalence relationship, and we denote it by $\equiv_k$, i.e., if a pair of vertices $\langle x, y \rangle$ is k–edge–connected we write $x \equiv_k y$. The vertices of a graph G are partitioned by this relationship into equivalence classes called *k–edge–connected components*. G is said to be *k–edge–connected* if the removal of any $(k-1)$ edges leaves G connected. As a result of these definitions, G is k–edge–connected if and only any two vertices of G are k–edge–connected. An edge set $E' \subseteq E$ is an *edge–cut for vertices x and y* if the removal of all the edges in E' disconnects G into two graphs, one containing x and the other containing y. An edge set $E' \subseteq E$ is an *edge–cut for G* if the removal of all the edges in E' disconnects G into two graphs. An edge–cut E' for G [for x and y respectively] is *minimal* if removing any edge from E' reconnects G [x and y respectively]. The cardinality of an edge–cut E', denoted by $|E'|$, is given by the number of edges in E'. An edge–cut E' for G [for x and y respectively] is said to be a *minimum cardinality edge–cut* or in short a *connectivity edge–cut* if there is no other edge–cut E'' for G [for x and y respectively] such that $|E''| < |E'|$. Connectivity edge–cuts are of course minimal edge–cuts. Note that G is k–edge–connected if and only if a connectivity edge–cut for G contains at least k edges, and $x \equiv_k y$ if and only if a connectivity edge–cut for x and y contains at least k edges. A connectivity edge–cut of cardinality 1 is called a *bridge*.

3 Dynamic Problems on Trees

This section presents data structures that maintain properties of a dynamically changing forest of trees. The basic operations that we will consider are edge insertions and edge deletions. Many properties of dynamically changing trees have been considered in the literature. The basic property is tree membership: namely, while the forest of trees is dynamically changing, we would like to know at any time which tree contains a given vertex, or whether two vertices are in the same tree. Dynamic tree membership is a special case of dynamic connectivity in undirected graphs, and indeed we will later use some of the data structures developed here for trees to solve the more general problem on graphs. The partially dynamic tree membership problem by means of the well known set union data structures [50] described in Chapter 8. We will thus focus on the fully dynamic tree membership problem. Other properties that have been considered are finding the parent of a vertex, or finding the least common ancestor of two vertices [44]. When costs are associated either to vertices or to edges, one could also ask what is the minimum (or maximum) cost in a given path.

The fully dynamic tree membership problem consists of maintaining a forest of unrooted trees under insertion of edges (which merge two trees into one), deletion of edges (which split one tree into two), and membership queries. Typical queries return the name of the tree containing a given vertex, or ask whether two vertices are in a same tree. Most of the solutions presented here will root each tree arbitrarily at one of its vertices; by keeping extra information at the root (such as the name of the tree), membership queries are equivalent to finding the root of the tree containing the given vertex.

There are two fully dynamic data structures for this problem: the dynamic trees of Sleator and Tarjan [44], and the topology trees of Frederickson [14]. Both data structures follow the common idea of partitioning a tree into a set of vertex–disjoint paths. However, they are very different in how this partition is chosen, and in the data structures they use to represent the paths inside the partition. Indeed, Sleator and Tarjan [44] use a simple partition of the trees based upon a careful choice of sophisticated data structures to represent paths. On the contrary, Frederickson [14] uses a more sophisticated partition that is based upon the topology of the tree; this implies more complicated algorithms but simpler data structures for representing paths.

The dynamic trees of Sleator and Tarjan [44] are able to maintain maintain a collection of rooted trees, each of whose vertices has a real–valued cost, under an arbitrary sequence of the following operations:

maketree(v)*:*	initialize a new tree consisting of single vertex v with cost zero.
findroot(v)*:*	return the root of the tree containing vertex v.
findcost(v)*:*	return a vertex of minimum cost in the path from v to findroot(v).
addcost(v, δ)*:*	add the real number δ to the cost of every vertex in the path from v to findroot(v).
link(v, w)*:*	Merge the trees containing vertices v and w by inserting edge (v, w). This operation assumes that v and w are in different trees and that v is a tree root.
cut(v)*:*	Delete the edge leaving v, thus splitting into two the tree containing vertex v. This operation assumes that v is not a tree root.
evert(v)*:*	Make v the root of its tree.

Theorem 1 (Sleator and Tarjan [44]) *Each of the above operations can be supported in $O(\log n)$ worst–case time.*

We do not give the details of the method here, and refer the interested reader to reference [44]. We only mention that some extensions of the dynamic trees of Sleator and Tarjan are the edge–ordered dynamic trees [12], designed to handle compressions and expansions of edges efficiently, and the dynamic expression trees [5]. Rather than the dynamic trees of Sleator and Tarjan, we describe in more detail the topology trees of Frederickson. The reason for this choice is that the topology trees are often used as building blocks by many dynamic graph algorithms.

3.1 Topology Trees

In this section we consider trees with maximum vertex degree 3. This is without loss of generality, as we can convert any tree T to a tree T' with maximum vertex degree 3 by means of a standard transformation [23], which will be defined in a more general sense for graphs in Section 5.1. Let T be a tree of maximum degree 3 to be maintained dynamically. Before definining formally a topology tree, we need a little terminology. A *vertex cluster* with respect to T is a set of vertices that induces a connected subgraph on T. The *cardinality* of a cluster is the number of vertices in it. An edge is *incident* to a cluster if exactly one of its endpoints is inside the cluster. Two clusters are *adjacent* if there is a tree edge that is incident to both. A *boundary vertex* of a cluster is a vertex that is adjacent in T to some vertex not in the cluster. The *external degree* of a cluster is the number of tree edges incident to it. A *restricted partition* of T is a partition of its vertex set V into vertex clusters such that:

(1) Each cluster of external degree 3 is of cardinality 1.

(2) Each cluster of external degree less than 3 is of cardinality at most 2.

(3) No two adjacent clusters can be combined and still satisfy the above.

There can be several restricted partitions for a given tree T, based upon different choices of the vertices to be unioned. Because of (3), the restricted partition implements a cluster–forming scheme according to a locally greedy heuristic, which does not always obtain the minimum number of clusters. However, this greedy method has the advantage of requiring only local adjustments during updates. Thus, although not optimal from the viewpoint of number of clusters generated, this partition is well suited for dynamic operations. To illustrate more this point, we sketch how to update the clusters of a restricted partition when the underlying tree is subject to updates.

Assume we want to delete an edge e from T. First, removing e splits T into two trees, say T_1 and T_2. T_1 and T_2 inherit all of the clusters of T, possibly with the following exceptions. If e is entirely contained in a cluster, this cluster is no longer connected and therefore must be split. After the split, we must check whether each of the two resulting clusters is adjacent to a cluster of tree degree at most 2, and if these two adjacent clusters together have at most 2 vertices. If so, we combine these two clusters in order to maintain condition (3) above. If e is between two clusters, then no split is needed. However, since the tree degree of the clusters containing the endpoints of e has been decreased, we must check if each cluster should be combined with an adjacent cluster, again because of condition (3). This completes the updates needed for the deletion of e.

Next, we show how to combine two trees T_1 and T_2 into one tree T by adding an edge f. Again, T inherits the clusters of T_1 and T_2 with the only following exceptions. If f increases the tree degree of a cluster from 1 to 2, in order to preserve condition (3) we must again check whether this cluster must be combined with the cluster newly adjacent to it. If f increases the tree degree of a cluster containing more than one vertex from 2 to 3, condition (1) is violated and we have to split the cluster. For each cluster formed after the split, we check whether it must be combined with an adjacent cluster.

A *restricted multi-level partition* consists of a collection of restricted partitions of V satisfying the following:

(1) The clusters at level 0 (known as *basic clusters*) contain one vertex each.

(2) The clusters at level $\ell \geq 1$ form a restricted partition with respect to the tree obtained after shrinking all the clusters at level $\ell - 1$.

(3) There is exactly one vertex cluster at the topmost level.

From this definition, it follows that any cluster at level $\ell \geq 1$ is either (a) the union of two *adjacent* clusters of level $(\ell - 1)$ such that the external degree of one cluster is 1 or the external degree of both is 2, or (b) one cluster of level $(\ell - 1)$, if rule (a) does not apply. Thus, any cluster obtained by unioning two clusters at a lower level, has always tree degree at most 2. This implies that at any level a cluster of tree degree 3 consists always of a single vertex. Once again, several multi–level partitions are possible. But each restricted multi–level partition has the nice property of having only logarithmic depth, as implied by the following lemma of Frederickson [15].

Lemma 1 (Frederickson [15]) *For any $\ell \geq 0$, the number of clusters at level $\ell + 1$ is at most $5/6$ times the number of clusters at level ℓ.*

The *topology tree* is a hierarchical representation of T. Each level of the topology tree partitions the vertices of T into connected subsets called *clusters*. More precisely, given a restricted multi–level partition for T, a *topology tree* for T is a tree satisfying the following:

(1) A topology tree node at level ℓ represents a vertex cluster at level ℓ in the restricted multi–level partition.

(2) A node at level $\ell \geq 1$ has at most two children, representing the vertex clusters at level $\ell - 1$ whose union gives the vertex cluster the node represents.

Since by Lemma 1 a restricted multi–level partition reduces the number of clusters at each level by a constant fraction, the height of the topology tree is $O(\log n)$. We now sketch how to update a topology tree during edge insertions or deletions. If there are updates in the spanning tree T, the restricted multi–level partition of T may be required to change. The changes in the topology tree are caused by the changes in the restricted multi–level partition it represents: at each level of the topology tree, we apply few locally greedy adjustments similar to the ones described before for one–level restricted partitions. As shown in [15], the topology tree update actually consists of two subtasks. First, a constant number of basic clusters (corresponding to leaves in the topology tree) have to be examined and possibly updated. Second, the changes in these basic clusters percolate up in the topology tree, possibly causing vertex clusters in the multi–level partition to be regrouped in different ways. This is handled by rebuilding portions of the topology tree in a bottom–up fashion, and involves a constant amount of work to be done on at most $O(\log n)$ topology tree nodes.

Lemma 2 (Frederickson [14]) *The update of a topology tree because of an edge insertion or deletion can be supported in time $O(\log n)$.*

4 Partially Dynamic Problems on Undirected Graphs

For undirected graphs, most of the partially dynamic problems considered are incremental, namely they support edge insertions only. We first show how to maintain a minimum spanning forest when the underlying graph G is subject to insertions of edges. We represent each tree in the minimum spanning forest as a dynamic tree (as defined in Section 3). Let $e = (x, y)$ be the edge to be inserted, and let F be a minimum spanning forest of G before inserting e. If the endpoints of e are in two different connected components of G, then the new forest F' will be $F \cup \{e\}$. Note that all this can be done in $O(\log n)$ worst–case time with the dynamic trees: we find the root of x, the root of y, and then link the two trees. Otherwise, x and y are in the same tree of the forest F, and inserting e in F introduces one cycle λ. To compute the new spanning forest F' we apply the cycle property in λ as follows. Let T be the tree of F containing both x and y: perform an evert on T and root it at x. Next, find the maximum cost edge f in the path from y to the root. If the cost of e is greater than the cost of f, then $F' = F$. Otherwise, swap e and f. Note that again this implies a constant number of operations to be executed on a forest of dynamic trees and therefore can be accomplished in $O(\log n)$ worst–case time.

Theorem 2 *A minimum spanning forest of a graph G subject to edge insertions only can be maintained in $O(\log n)$ worst–case time per operation, where n is the number of vertices in G.*

The previous theorem gives implicitly an incremental algorithm for maintaining the connected components of a graph, since a minimum spanning forest of G is trivially a spanning forest of G (we assume that each edge has cost 1). A query on whether two vertices x and y are in a same connected component can be answered in $O(\log n)$ time by finding and comparing the roots of x and y. A better bound can be achieved by representing the trees in the forest using the set union data structures of Chapter 8. A query on whether two vertices x and y are connected can be answered by checking whether find(x) equals find(y). When inserting edge $e = (x, y)$, we first perform $A \leftarrow$ find(x) and $B \leftarrow$ find(y). If $A = B$, x and y were already connected and the insertion of e does not change the connected components of G. If $A \neq B$ then A and B have to be merged into a new connected component: we do this by executing a union(A, B). Since each operation can be implemented by a constant number of set union operations, we have the following theorem.

Theorem 3 *The connected components a graph G subject to edge insertions only can be maintained in $O(\alpha(q,n))$ amortized time per query or update operation, where q is the total number of queries and n is the number of vertices in G.*

Set union data structures can be used for the partially dynamic maintenance of other graph properties, such as bipartiteness, and edge and vertex connectivity. For lack of space, we describe here only how to solve the partially dynamic 2–edge connectivity problem. This problem consists of maintaining a graph G under an intermixed sequence of operations of the following kinds.

Same2EdgeBlock(u,v):	Return *true* if vertices u and v are in the same 2–edge–connected component. Return *false* otherwise.
InsertEdge(x,y):	Insert a new edge between the two vertices x and y.

Westbrook and Tarjan [52] presented one algorithm that runs in a total of $O(q\alpha(q,n))$ time, where q is the total number of *Same2EdgeBlock* and *InsertEdge* operations, and n is the number of vertices. For sake of simplicity, we describe in details only the algorithm that operates on connected graphs, and refer the interested reader to [52] for the full details of the method on general unconnected graphs.

For any vertex x in G denote by $C_{2E}(x)$ the 2–edge–connected component of G containing x. The main data structure maintained by the algorithm is the bridge–block tree of G, which is defined as follows. Nodes in this tree correspond to 2–edge–connected components of G, and any two nodes are connected by an edge if and only if there is a bridge connecting the corresponding 2–edge–connected components. We assume that the bridge–block tree is rooted arbitrarily at one node, and we denote by *parent*(σ) the parent of a tree node σ in the bridge–block tree.

Besides maintaining the bridge–block tree of G, we maintain the actual 2–edge–connected components of G as disjoint sets, subject to *union* and *find* operations. For this, we use any of the fast set–union data structures described in Chapter 8, which are able to process any sequence of q *union* and *find* operations on a collection of n elements in $O(q\alpha(q,n))$ worst–case time.

We define the name of each disjoint set in the set–union data structure as a pointer to the tree node associated to the corresponding 2–edge–connected component. Thus, we can assume that for each vertex x in G, *find*(x)= $C_{2E}(x)$. With this data structure, a *Same2EdgeBlock*(x,y) can be simply performed by checking whether *find*(x)=*find*(y). The *union* operations will be used to update efficiently the 2–edge–connected components during *InsertEdge* operations. The effect of an *InsertEdge*(u,v) on the bridge–block tree depends on whether u and v are in the same 2–edge–connected component or in different 2–edge–connected components. Let $C_{2E}(u)$ and $C_{2E}(v)$ be the 2–edge–connected components containing u and v respectively, before inserting the edge (u,v).

We now describe the changes caused in the bridge–block tree by the insertion of edge (u,v). If $C_{2E}(u) = C_{2E}(v)$, the bridge–block tree is unaffected by the new edge and hence no change is needed in the data structure. If $C_{2E}(u) \neq C_{2E}(v)$, the inserted edge creates a new cycle that reduces the number of 2–edge–connected components and bridges of G. This cycle consists of (u,v) plus all the edges in the tree path between node $C_{2E}(u)$ and node $C_{2E}(v)$. All the nodes in this tree path must be replaced by a single node, and every node previously adjacent to one of the nodes in the tree path becomes adjacent to the new single node. This process is called *path condensation.*

To implement *InsertEdge* operations, the bridge–block tree is maintained as a data structure that supports the following primitives:

FindPath(σ_1,σ_2)*:* Given two tree nodes σ_1 and σ_2, return the tree path between σ_1 and σ_2. If $\sigma_1 = \sigma_2 = \sigma$, return σ.

CondensePath(π)*:* Perform path condensation on the tree path π, unioning the corresponding disjoint sets associated with the encountered tree nodes. Return the modified bridge–block tree. Note that a *CondensePath* does nothing if π is an empty path consisting of a single node.

A *FindPath*(σ_1,σ_2) can be implemented as follows: we proceed from u and from v towards the tree root, alternating one step at the time, until the paths traced from u and from v intersect at their nearest common ancestor. The number of steps required to return a path π is at most $2|\pi|$. Since they are performed by following parent pointers, we call these *parent steps.* The path π is returned as a list of nodes (not in order along the path), with the nearest common ancestor at the end. Let $\widetilde{\sigma}$ be the parent of the nearest common ancestor. To perform a *CondensePath*(π), π is condensed into a single node σ, which is made child of $\widetilde{\sigma}$. All the disjoint sets associated with nodes in π are unioned. Since we perform a union at each step, we call these *union steps.*

Lemma 3 *In any sequence of operations, there are at most $O(n)$ parent and union steps.*

Proof: Suppose a path π is being condensed. To generate π, $O(|\pi|)$ parent steps are required, and $|\pi|-1$ nodes are condensed. Since after $(n-1)$ condensations the graph becomes 2–edge–connected, there can be at most $O(n)$ parent steps and $(n-1)$ union steps. □

With the help of these primitives, we support an *InsertEdge*(u,v) as follows:

```
$InsertEdge(u,v)$ **begin**
        **return** $CondensePath(FindPath(find(u), find(v)))$;
**end**
```

Theorem 4 (Westbrook and Tarjan [52]) *Given an initially connected graph $G_0 = (V_0, E_0)$ and $O(|E_0|)$ preprocessing time, a sequence of q* InsertEdge *and* Same2EdgeBlock *operations can be processed in $O(q\alpha(q,n))$ time.*

Proof: The 2–edge–connected components and the bridge–block tree of G_0 can be found in $O(|E_0|)$ using the algorithm of Tarjan [49]. By Lemma 3, the total number of parent and union steps is $O(n)$, giving a total of $O(q)$ finds and $O(n)$ unions in the set union data structure. □

To complete this section, we mention that all the incremental algorithms proposed in the literature for edge and vertex connectivity (see e.g. [7, 8, 20, 33, 35, 36, 38, 52]) follow the approach outlined here for 2–edge connectivity. Namely, connectivity queries are answered by maintaining a tree that reflects the structure of the connectivity cuts of the graph. When a new edge (u,v) is added to the graph, some connectivity cuts may be invalidated; these connectivity cuts can be easily found since they all lie in a tree path, whose endpoints correspond to u and v. After locating the path, some form of path condensation takes place. To support efficiently path condensation, set–union based data structures are used. The tree structure of the $(k-1)$–cuts is fairly complicated, however, as k increases, and it requires sophisticated data structures and a quite delicate analysis.

5 Fully Dynamic Problems on Undirected Graphs

This section describes fully dynamic algorithms for undirected graphs. These algorithms maintain efficiently some property of a graph that is undergoing structural changes defined by insertion and deletion of edges, and/or edge cost updates. For instance, the fully dynamic minimum spanning tree problem consists of maintaining a minimum spanning forest of a graph during the above operations. The typical updates for a fully dynamic problem will therefore be inserting a new edge, and deleting an existing edge. To check the graph property throughout a sequence of these updates, the algorithms must prepared to answer queries on the graph property. Thus, a fully dynamic connectivity algorithm must be able to insert edges, delete edges, and answer a query on whether the graph is connected, or whether two vertices are connected. Similarly, a fully dynamic k–edge connectivity algorithm must be able to insert edges, delete edges, and answer a query on whether the graph is k–edge–connected, or whether two vertices are k–edge–connected. The goal of a dynamic algorithm is to minimize the amount of recomputation required after each update. All of the fully dynamic algorithms that we describe in this section are able to dynamically maintain the graph property at a cost (per update operation) which is significantly smaller than the cost of recomputing the graph property from scratch. Many of the algorithms proposed in the literature use the same general techniques, and so we begin by describing these techniques. All of these techniques use some form of graph decomposition, and partition either the vertices or the edges of the graph to be maintained.

The first technique we present is the *clustering* technique of Frederickson [14], which is based upon partitioning the graph into a suitable collection of connected subgraphs called *clusters*, such that each update involves only a small number of such clusters. The dynamic trees of Sleator and Tarjan [44] can be used to maintain information about the edges of a tree; clusters dually keep track of the edges that are not part of some given spanning tree, by grouping them according to which clusters they connect. Typically, this decomposition is applied recursively and the information about the subgraphs is combined with the topology trees described in Section 3.1. A refinement of the clustering technique appears in the idea of *ambivalent data structures* [15], in which edges can belong to multiple groups, only one of which is actually selected depending on the topology of the given spanning tree.

Another technique we describe is *sparsification* by Eppstein *et al.* [10]. This is a divide–and–conquer technique that can be used to reduce the dependence on the number of edges in a graph, so that the time bounds for maintaining some property of the graph match the times for computing in sparse graphs. Roughly speaking, sparsification works as follows. Let $\mathcal{A}$ be an algorithm that maintains some property of a dynamic graph G with m edges and n vertices in time $T(n, m)$. Sparsification maintains a proper decomposition of G into small subgraphs, with $O(n)$ edges each.

In this decomposition, each update involves applying algorithm $\mathcal{A}$ to few small subgraphs of G, resulting into an improved $T(n, n)$ time bound per update. Thus, throughout a sequence of operations, sparsification makes a graph looks sparse (i.e., with only $O(n)$ edges): hence, the reason for its name. Sparsification works on top of a given algorithm, and need not to know the internal details of this algorithm. Consequently, it can be applied orthogonally to other data structuring techniques; we will actually see a number of situations in which both clustering and sparsification can be combined to produce an efficient dynamic graph algorithm.

The previous two techniques allows one to design efficient deterministic algorithms. The last technique we present in this section is due to Henzinger and King [24], and it is a combination of a suitable graph decomposition and randomization. We now sketch how this decomposition is defined. Let G be a graph whose spanning forest has to be maintained dynamically. The edges of G are partitioned into $O(\log n)$ levels: the lower levels contain tightly–connected portions of G (i.e., dense edge cuts), while the higher levels contain loosely–connected portions of G (i.e., sparse cuts). For each level i, a spanning forest for the graph defined by all the edges in levels i or below is maintained. If a spanning forest edge e is deleted at some level i, random sampling is used to quickly find a replacement for e at that level. If random sampling succeeds, the forest is reconnected at level i. If random sampling fails, the edges that can replace e in level i form with high probability a sparse cut. These edges are moved to level $(i + 1)$ and the same procedure is applied recursively on level $(i + 1)$.

One particular dynamic graph problem that has been thoroughly investigated is the maintenance of a minimum spanning forest [4, 10, 14, 45]. This is an important problem on its own, but it has also impact on other problems as well. Indeed the data structures and techniques developed for dynamic minimum spanning forests have found applications also in other areas, such as dynamic edge and vertex connectivity [10, 15, 19, 26, 41, 42]. Thus, we will focus our attention to the fully dynamic maintenance of minimum spanning trees.

5.1 Clustering and Topology Trees

Let $G = (V, E)$ be a graph, with a designated spanning tree S. *Clustering* is a method of partitioning the vertex set V, into connected subtrees in S, so that each subtree is only adjacent to a few other subtrees. *Topology trees* are a representation of the tree S using a different tree with logarithmic height, formed by recursive clustering, and are a generalization of the topology trees described in Section 3.1. *2–dimensional topology trees* are formed from pairs of nodes in a topology tree, and allow us to maintain information about the edges in $E - S$. Fully dynamic algorithms based only on a single level of clustering obtain typically time bounds of $O(m^{2/3})$ (see for instance [19, 41]). When the partition can be applied recursively, better $O(m^{1/2})$ time bounds can be achieved with the help of topology trees (see for instance [14, 15]). Along with their applications to dynamic graph algorithms, topology trees can be used in situations in which a dynamic tree is part of a static graph; for instance, topology trees can be used to speed up the execution of pivots in the network simplex algorithm for minimum cost circulations [9].

Before defining clustering and topology trees, we describe a standard graph transformation that we use throughout. We convert the graph G into a graph with maximum vertex degree 3 [23]: Suppose $v \in V$ has degree $d(v) > 3$, and is adjacent to vertices $u_1, u_2, \ldots, u_d$. In the transformed graph, v is replaced by a cycle of d *dashed* edges: namely, we substitute v by d vertices $v_1, v_2, \ldots, v_d$. For each edge (v, u) of the original graph, in position i among the list of edges adjacent to v and position j among the edges adjacent to u, we create an *actual* edge (v_i, u_j). We also create *dashed* edges (v_i, v_{i+1}) for $1 \leq i \leq d - 1$, and a *dashed* edge (v_d, v_1) to close the loop. We call these the *dashed edges of* v. As a result of this transformation, the graph keeps its actual edges, and has an additional $O(m)$ dashed edges. Note that this transformation affects the problems we would like to solve. However for most of the problems to which this technique is applied (such as minimum spanning forests, connectivity and 2–edge connectivity), either the two solutions are identical (such as in the case of connectivity and 2–edge connectivity), or we can easily compute the solution in the original graph once we know the solution in the transformed graph (such as in the case of minimum spanning forests). In other cases, although this computation is not easy (such in the case of biconnectivity), it can be handled with some little extra effort.

Throughout the sequence of updates, any cluster–based dynamic graph algorithm maintains a spanning tree T of G. For each vertex v, T contains all of the dashed edges of v except one. The only one dashed edge of v that does not belong to T can be arbitrarily chosen. We now generalize the notion of restricted partition given in Section 3.1. Let $z > 0$ be an integer, to be fixed later on. A *restricted partition of order* z of G is a partition of its vertex set V into $O(m/z)$ vertex clusters such that:

(1) Each set in the partition yields a vertex cluster of external degree at most 3.

(2) Each cluster of external degree 3 is of cardinality 1.

(3) Each cluster of external degree less than 3 is of cardinality less than or equal to z.

(4) No two adjacent clusters can be combined and still satisfy the above.

Note that since any cluster has maximum external degree 3, it can have at most three boundary vertices. A restricted partition of order z can be found in linear time as shown in [15]. We now discuss how to update the clusters of a restricted partition of order z when the underlying graph is subject to updates. The basic update we will consider is a *swap*: a *swap* (e, f) in a spanning tree T replaces a tree edge e by a non–tree edge f, yielding a new spanning tree. This is a basic update operation, since each edge insertion, edge deletion, or edge cost change causes at most one swap in a spanning tree: at most one edge is added to the tree and one edge removed. When a swap (e, f) is performed, we do the following. First, removing e splits T into two trees, say T_1 and T_2. T_1 and T_2 inherit all of the clusters of T, possibly with the following exceptions. If e is entirely contained in a cluster, this cluster is no longer connected and therefore must be split. After the split, we must check whether each of the two resulting clusters can be merged with neighboring clusters in order to maintain condition (4) above. If e is between two clusters, then no split is needed. However, since the tree degree of the clusters containing the endpoints of e has been decreased, we must check if each cluster should be combined with an adjacent cluster, again because of condition (4). This completes the updates needed for the deletion of e.

Next, we show how to combine T_1 and T_2 into one tree T by adding edge f. Again, T inherits the clusters of T_1 and T_2 with the only following exceptions. If f increases the tree degree of a cluster from 1 to 2, in order to preserve condition (4) we must again check whether this cluster must be combined with the cluster newly adjacent to it. If f increases the tree degree of a cluster containing more than one vertex from 2 to 3, condition (2) is violated and we do the following. Let w', w'' and w''' be the endpoints of the three tree edges incident on this cluster. Let x be the common vertex on tree paths between w', w'' and w'''. Make x into a cluster (of tree degree 3) by itself, and take the remaining parts of the old cluster as new clusters. For each cluster so formed, we check whether it must be combined with an adjacent cluster.

Lemma 4 *The time required to update a restricted partition of order z because of a swap is $O(z)$.*

Proof: As described before, at most a constant number of vertex clusters is changed, deleted or created during a swap. Each cluster that is modified in some way, has at most z vertices and edges, and therefore can be handled in time $O(z)$. $\square$

The notion of *restricted multi–level partition* of Section 3.1 can be generalized as follows:

(1) The clusters at level 0 (known as *basic clusters*) form a restricted partition of order z.

(2) The clusters at level $\ell \geq 1$ form a restricted partition of order 2 with respect to the tree obtained after shrinking all the clusters at level $\ell - 1$.

(3) There is exactly one vertex cluster at the topmost level.

We now list some properties of a multi–level restricted partition. First, any basic vertex cluster of tree degree 3 consists of a single vertex. Second, any cluster obtained by unioning two clusters at a lower level, has always tree degree at most 2. These two properties imply that any non–basic cluster of tree degree 3 also consists of a single vertex, and all of its incident edges will be tree edges. Furthermore, there are no non–tree edges having an endpoint in a cluster of tree degree 3. Finally, the restricted multi–level partition has the nice property of having only logarithmic depth. Indeed Frederickson [15] shows that each level of the topology tree has a number of nodes which is a constant fraction of the previous level, from which the following lemma follows.

Lemma 5 (Frederickson [14, 15]) *The number of levels in a restricted multi–level partition is $\Theta(\log n)$.*

The *topology tree* is a hierarchical representation of G based on T. Each level of the topology tree partitions the vertices of G into connected subsets called *clusters*. More precisely, given a restricted multi–level partition for T, a *topology tree* for T is a tree satisfying the following:

(1) A topology tree node at level ℓ represents a vertex cluster at level ℓ in the restricted multi–level partition.

(2) A node at level $\ell \geq 1$ has at most two children, representing the vertex clusters at level $\ell - 1$ whose union gives the vertex cluster the node represents.

As shown in [15], the update of a topology tree because of an edge swap in T consists of two subtasks. First, a constant number of basic clusters (corresponding to leaves in the topology tree) have to be examined and possibly updated. As shown in Lemma 4, this can be supported in $O(z)$ time. Second, the changes in these basic clusters percolate up in the topology tree, possibly causing vertex clusters in the multi–level partition to be regrouped in different ways. This is handled by rebuilding portions of the topology tree in a bottom–up fashion, and involves a constant amount of work to be done on at most $O(\log n)$ topology tree nodes. This yields the following lemma.

Lemma 6 (Frederickson [14, 15]) *The update of a topology tree because of an edge swap can be supported in time $O(z + \log n)$.*

A *2–dimensional topology tree* for a topology tree is defined as follows. For every pair of nodes V_α and V_β at the same level in the topology tree there is a node labeled $V_\alpha \times V_\beta$ in the 2–dimensional topology tree. Let E_T be the tree edges of G (i.e., the edges in the spanning tree T): node $V_\alpha \times V_\beta$ represents all the non–tree edges of G (i.e., the edges of $E - E_T$) having one endpoint in V_α and the other in V_β. The root of the 2–dimensional topology tree is labeled $V \times V$ and represents all the non–tree edges of G. If a node is labeled $V_\alpha \times V_\beta$, and V_α has children V_{α_i}, $1 \leq i \leq p$, and V_β has children V_{β_j}, $1 \leq j \leq q$, in the topology tree, then $V_\alpha \times V_\beta$ has children $V_{\alpha_i} \times V_{\beta_j}$, $1 \leq i \leq p, 1 \leq j \leq q$, in the 2–dimensional topology tree.

Note that a 2–dimensional topology tree corresponds roughly to having $O(m/z)$ topology trees, one for each basic cluster in the restricted multi–level partition. As previously described, updating the basic clusters because of an edge swap would require a total of $O(z)$ time, and then updating these $O(m/z)$ topology trees would require a total of $O((m/z)\log n)$ time. This yields a total of $O(z + (m/z)\log n)$ time. The computational saving of a 2–dimensional topology tree is that it can be updated during a swap in its corresponding topology tree in $O(m/z)$ time only [15]. The crucial point of this analysis is that only $O(m/z)$ nodes in the 2–dimensional topology tree need to be looked at and eventually updated during a swap, and this can be done in constant time per node.

Lemma 7 (Frederickson [14, 15]) *The update of a 2–dimensional topology tree because of an edge swap in the corresponding topology tree can be supported in time $O(m/z)$.*

Note that the bound in Lemma 7 does not take into account the time needed to update the topology tree upon which the 2–dimensional topology tree is based. If this is taken into account, the total time required to perform a swap becomes $O(z + (m/z))$. Typical algorithms will balance this bound by choosing $z = \Theta(m^{1/2})$ to get an $O(m^{1/2})$ total time bound, as shown in the following theorem.

Theorem 5 (Frederickson [14]) *The minimum spanning forest of an undirected graph can be maintained in time $O(m^{1/2})$ per update, where m is the current number of edges in the graph.*

Proof: We maintain a restricted multi–level partition of order z, and the corresponding topology tree and 2–dimensional topology tree as described before. We augment the 2–dimensional topology tree as follows. Each leaf $V_i \times V_j$ of the 2–dimensional topology tree stores the set $E_{i,j}$ of edges having one endpoint in V_i and the other in V_j, as well as the minimum cost edge in this set. This information is stored in a heap–like fashion: internal nodes of the 2–dimensional topology tree have the minimum of the values of their children. This additional information required constant time per node to be maintained. Consequently, as in Lemma 7 the update of this augmented 2–dimensional topology tree because of a swap can be done in $O(m/z)$ time.

Whenever a new edge is inserted or a non–tree edge has its cost decreased, we can find a replacement edge in time $O(\log n)$ with the dynamic trees of Sleator on Tarjan, as described in Section 4. Whenever an edge is deleted, or a tree edge has its cost increased, we can find a a replacement edge as follows. Let e be the edge that has been deleted or increased. We first split the 2–dimensional topology tree at e in $O(z + m/z)$ time by Lemma 7. Suppose this splits the corresponding topology tree into two trees, whose roots are the clusters V_α and V_β, with V_β having no fewer levels than V_α. To find a possible replacement edge for e, we examine the values at the nodes $V_\alpha \times V_\gamma$ for all possible V_γ in the 2–dimensional topology tree, and take the minimum. It takes $O(m/z)$ time to find and examine these nodes.

This yields a total of $O(z+(m/z))$ time for each update. Choosing $z = \lceil m^{1/2} \rceil$ gives an $O(m^{1/2})$ bound. However, m is changing because of insertions and deletions. When the value of z changes because of m, there will be at least $m^{1/2}$ update before z advances to the next value up or down in the same directions. Since there are at most $O(m/z)$ basic clusters that need to be adjusted, we can adjust a constant number of clusters during each update. □

5.2 Sparsification

The techniques described in Section 5.1 allow us to obtain an $O(m^{1/2})$ time bound for the fully dynamic maintenance of a minimum spanning forest, connectivity and 2–edge connectivity [14, 15]. The type of clustering used is very problem–dependent, however, and makes these techniques unsuitable to be used as a *black box*. Namely, whenever we want to apply such techniques to solve a certain problem, we must devise a proper partition of the graph into clusters. Furthermore, we must get into the low–level details of the data structures employed. For instance, we have to make sure that the problem can be represented as a small set of alternatives at each node of the 2–dimensional topology tree, and show how we can select efficiently among those alternatives, or select the information we are interested more directly from the topology tree.

In this section we describe a general technique for designing dynamic graph algorithms, due to Eppstein *et al.* [10], which is called *sparsification*. This technique can be used to speed up many fully dynamic graph algorithms. Roughly speaking, when the technique is applicable it speeds up a $T(n, m)$ time bound for a graph with n vertices and m edges to $T(m, O(n))$; i.e. to the time needed if the graph were sparse. For instance if $T(n, m) = O(m^{1/2})$, we get a better bound of $O(n^{1/2})$. Sparsification applies to a wide variety of dynamic graph problems, including minimum spanning forests, edge and vertex connectivity. Moreover, it is a general technique and can be used as a *black box* (without having to know the internal details) in order to dynamize graph algorithms.

The technique itself is quite simple. Let G be a graph with m edges and n vertices. We partition the edges of G into a collection of $O(m/n)$ sparse subgraphs, i.e., subgraphs with n vertices and $O(n)$ edges. The information relevant for each subgraph can be summarized in an even sparser subgraph, which is called a *sparse certificate*. We merge certificates in pairs, producing larger subgraphs which are made sparse by again computing their certificate. The result is a balanced binary tree in which each node is represented by a sparse certificate. Each update involves $\log(m/n)$[1] graphs with $O(n)$ edges each, instead of one graph with m edges. With some extra care, the $O(\log(m/n))$ overhead term can be eliminated.

We describe two variants of the sparsification technique. We use the first variant in situations where no previous fully dynamic algorithm was known. We use a static algorithm to recompute a sparse certificate in each tree node affected by an edge update. If the certificates can be found

[1]Throughout $\log x$ stands for $\max(1, \log_2 x)$, so $\log(m/n)$ is never smaller than 1 even if $m < 2n$.

in time $O(m+n)$, this variant gives time bounds of $O(n)$ per update. In the second variant, we maintain certificates using a dynamic data structure. For this to work, we need a *stability* property of our certificates, to ensure that a small change in the input graph does not lead to a large change in the certificates. This variant transforms time bounds of the form $O(m^p)$ into $O(n^p)$. We start by describing an abstract version of sparsification. The technique is based on the concept of a *certificate*:

Definition 1 *For any graph property $\mathcal{P}$, and graph G, a* certificate *for G is a graph G' such that G has property $\mathcal{P}$ if and only if G' has the property.*

Cheriyan *et al.* [3] use a similar concept, however they require G' to be a subgraph of G. We do not need this restriction. However, this allows trivial certificates: G' could be chosen from two graphs of constant complexity, one with property $\mathcal{P}$ and one without it.

Definition 2 *For any graph property $\mathcal{P}$, and graph G, a* strong certificate *for G is a graph G' on the same vertex set such that, for any H, $G \cup H$ has property $\mathcal{P}$ if and only if $G' \cup H$ has the property.*

In all our uses of this definition, G and H will have the same vertex set and disjoint edge sets. A strong certificate need not be a subgraph of G, but it must have a structure closely related to that of G. The following facts follow immediately from Definition 2.

Fact 1 *Let G' be a strong certificate of property $\mathcal{P}$ for graph G, and let G'' be a strong certificate for G'. Then G'' is a strong certificate for G.*

Fact 2 *Let G' and H' be strong certificates of $\mathcal{P}$ for G and H. Then $G' \cup H'$ is a strong certificate for $G \cup H$.*

A property is said to have *sparse certificates* if there is some constant c such that for every graph G on an n–vertex set, we can find a strong certificate for G with at most cn edges.

The other key ingredient is a *sparsification tree*. We start with a partition of the vertices of the graph, as follows: we split the vertices evenly in two halves, and recursively partition each half. Thus we end up with a complete binary tree in which nodes at distance i from the root have $n/2^i$

vertices. We then use the structure of this tree to partition the edges of the graph. For any two nodes α and β of the vertex partition tree at the same level i, containing vertex sets V_α and V_β, we create a node $E_{\alpha\beta}$ in the edge partition tree, containing all edges in $V_\alpha \times V_\beta$. The parent of $E_{\alpha\beta}$ is $E_{\gamma\delta}$, where γ and δ are the parents of α and β respectively in the vertex partition tree. Each node $E_{\alpha\beta}$ in the edge partition tree has either three or four children (three if $\alpha = \beta$, four otherwise). We use a slightly modified version of this edge partition tree as our sparsification tree. The modification is that we only construct those nodes $E_{\alpha\beta}$ for which there is at least one edge in $V_\alpha \times V_\beta$. If a new edge is inserted new nodes are created as necessary, and if an edge is deleted those nodes for which it was the only edge are deleted.

Lemma 8 *In the sparsification tree described above, each node $E_{\alpha\beta}$ at level i contains edges inducing a graph with at most $n/2^{i-1}$ vertices.*

Proof: There can be at most $n/2^i$ vertices in each of V_α and V_β. $\square$

We say a time bound $T(n)$ is *well–behaved* if for some $c < 1$, $T(n/2) < cT(n)$. We assume well–behavedness to eliminate strange situations in which a time bound fluctuates wildly with n. All polynomials are well–behaved. Polylogarithms and other slowly growing functions are not well behaved, but since sparsification typically causes little improvement for such functions we will in general assume all time bounds to be well behaved.

Theorem 6 (Eppstein *et al.* **[10])** *Let $\mathcal{P}$ be a property for which we can find sparse certificates in time $f(n,m)$ for some well–behaved f, and such that we can construct a data structure for testing property $\mathcal{P}$ in time $g(n,m)$ which can answer queries in time $q(n,m)$. Then there is a fully dynamic data structure for testing whether a graph has property $\mathcal{P}$, for which edge insertions and deletions can be performed in time $O(f(n,O(n))) + g(n,O(n))$, and for which the query time is $q(n,O(n))$.*

Proof: We maintain a sparse certificate for the graph corresponding to each node of the sparsification tree. The certificate at a given node is found by forming the union of the certificates at the three or four child nodes, and running the sparse certificate algorithm on this union. As shown in Lemmas 1 and 2, the certificate of a union of certificates is itself a certificate of the union, so this gives a sparse certificate for the subgraph at the node. Each certificate at level i can be computed in time $f(n/2^{i-1}, O(n/2^i))$. Each update will change the certificates of at most one node at each level of the tree. The time to recompute certificates at each such node adds in a geometric series to $f(n, O(n))$. This process results in a sparse certificate for the whole graph at the root of the tree. We update the data structure for property $\mathcal{P}$, on the graph formed by the sparse certificate at the root of the tree, in time $g(n, O(n))$. The total time per update is thus $O(f(n, O(n))) + g(n, cn)$. $\square$

This technique is very effective at producing dynamic graph data structures for a multitude of problems, in which the update time is $O(n \log^{O(1)} n)$ instead of the static time bounds of $O(m + n \log^{O(1)} n)$. To achieve sublinear update times, we further refine our sparsification idea.

Definition 3 *Let A be a function mapping graphs to strong certificates. Then A is* stable *if it has the following two properties:*

(1) For any graphs G and H, $A(G \cup H) = A(A(G) \cup H)$.

(2) For any graph G and edge e in G, $A(G - e)$ differs from $A(G)$ by $O(1)$ edges.

Informally, we refer to a certificate as stable if it is the certificate produced by a stable mapping. The certificate consisting of the whole graph is stable, but not sparse.

Theorem 7 (Eppstein *et al.* **[10])** *Let $\mathcal{P}$ be a property for which stable sparse certificates can be maintained in time $f(n, m)$ per update, where f is well behaved, and for which there is a data structure for property $\mathcal{P}$ with update time $g(n, m)$ and query time $q(n, m)$. Then $\mathcal{P}$ can be maintained in time $O(f(n, O(n))) + g(n, O(n))$ per update, with query time $q(n, O(n))$.*

Proof: As before, we use the sparsification tree described above. After each update, we propagate the changes up the sparsification tree, using the data structure for maintaining certificates. We then update the data structure for property $\mathcal{P}$, which is defined on the graph formed by the sparse certificate at the tree root.

At each node of the tree, we maintain a stable certificate on the graph formed as the union of the certificates in the three or four child nodes. The first part of the definition of stability implies that this certificate will also be a stable certificate that could have been selected by the mapping A starting on the subgraph of all edges in groups descending from the node. The second part of the definition of stability then bounds the number of changes in the certificate by some constant s, since the subgraph is changing only by a single edge. Thus at each level of the sparsification tree there is a constant amount of change.

When we perform an update, we find these s changes at each successive level of the sparsification tree, using the data structure for stable certificates. We perform at most s data structure operations, one for each change in the certificate at the next lower level. Each operation produces at most s changes to be made to the certificate at the present level, so we would expect a total of s^2 changes. However, we can cancel many of these changes since as described above the net effect of the update will be at most s changes in the certificate.

In order to prevent the number of data structure operations from becoming larger and larger at higher levels of the sparsification tree, we perform this cancellation before passing the changes in the certificate up to the next level of the tree. Cancellation can be detected by leaving a marker on each edge, to keep track of whether it is in or out of the certificate. Only after all s^2 changes have been processed do we pass the at most s uncancelled changes up to the next level.

Each change takes time $f(n, O(n))$, and the times to change each certificate then add in a geometric series to give the stated bound. $\square$

Theorem 6 can be used to dynamize static algorithms, while Theorem 7 can be used to speed up existing fully dynamic algorithms. In order to apply effectively Theorem 6 we only need to *compute* efficiently *sparse* certificates, while for Theorem 7 we need to *maintain* efficiently *stable sparse* certificates. Indeed stability plays an important role in the proof of Theorem 7. In each level of the update path in the sparsification tree we compute s^2 changes resulting from the s changes in the previous level, and then by stability obtain only s changes after eliminating repetitions and canceling changes that require no update. Although in most of the applications we consider stability can be used directly in a much simpler way, we describe it in this way here for sake of generality.

We next describe the $O(n^{1/2})$ algorithm for the fully dynamic maintenance of a minimum spanning forest given by Eppstein *et al.* [10] based on sparsification. A minimum spanning forest is not a graph property, since it is a subgraph rather than a Boolean function. However sparsification still applies to this problem. Alternately, sparsification maintains any property defined on the minimum spanning trees of graphs. The data structure introduced in this section will also be an important subroutine in some results described later. We need the following analogue of strong certificates for minimum spanning trees:

Lemma 9 *Let T be a minimum spanning forest of graph G. Then for any H there is some minimum spanning forest of $G \cup H$ which does not use any edges in $G - T$.*

Proof: If we use the cycle property on graph $G \cup H$, we can eliminate first any cycle in G (removing all edges in $G - T$) before dealing with cycles involving edges in H. $\Box$

Thus we can take the strong certificate of any minimum spanning forest property to be the minimum spanning forest itself. Minimum spanning forests also have a well–known property which, together with Lemma 9, proves that they satisfy the definition of stability:

Lemma 10 *Let T be a minimum spanning forest of graph G, and let e be an edge of T. Then either $T - e$ is a minimum spanning forest of $G - e$, or there is a minimum spanning forest of the form $T - e + f$ for some edge f.*

If we modify the weights of the edges, so that no two are equal, we can guarantee that there will be exactly one minimum spanning forest. For each vertex v in the graph let $i(v)$ be an identifying number chosen as an integer between 0 and $n - 1$. Let ϵ be the minimum difference between any two distinct weights of the graph. Then for any edge $e = (u, v)$ with $i(u) < i(v)$ we replace $w(e)$ by $w(e) + \epsilon i(u)/n + \epsilon i(v)/n^2$. The resulting MSF will also be a minimum spanning forest for the unmodified weights, since for any two edges originally having distinct weights the ordering between those weights will be unchanged. This modification need not be performed explicitly—the only operations our algorithm performs on edge weights are comparisons of pairs of weights, and this can be done by combining the original weights with the numbers of the vertices involved taken in lexicographic order. The mapping from graphs to unique minimum spanning forests is stable, since part (1) of the definition of stability follows from Lemma 9, and part (2) follows from Lemma 10.

We use Frederickson's algorithm of Theorem 5 that states that minimum spanning trees can be maintained in time $O(m^{1/2})$. We improve this bound by combining Frederickson's algorithm with sparsification: we apply the stable sparsification technique of Theorem 7, with $f(n, m) = g(n, m) = O(m^{1/2})$ by Theorem 5.

Theorem 8 (Eppstein *et al.* **[10])** *The minimum spanning forest of an undirected graph can be maintained in time $O(n^{1/2})$ per update.*

The dynamic spanning tree algorithms described so far produce fully dynamic connectivity algorithms with the same time bounds. Indeed, the basic question of connectivity can be quickly determined from a minimum spanning forest. However, higher forms of connectivity are not so easy. For edge connectivity, sparsification can be applied using a dynamic minimum spanning forest algorithm, and provides efficient algorithms: 2–edge connectivity can be solved in $O(n^{1/2})$ time per update, 3–edge connectivity can be solved in $O(n^{2/3})$ time per update, and for any higher k, k–edge connectivity can be solved in $O(n \log n)$ time per update [10]. Vertex connectivity is not so easy: for $2 \leq k \leq 4$, there are algorithms with times ranging from $O(n^{1/2} \log^2 n)$ to $O(n\alpha(n))$ per update [10, 26, 42].

5.3 Randomized Algorithms

All the previous techniques yield efficient deterministic algorithms, whose best running times are $O(n^{1/2})$. Recently, Henzinger and King [24] proposed a new approach that, exploiting the power of randomization, is able to achieve faster update times for some problems. For instance, for the fully dynamic connectivity problem this technique yields an expected amortized update time of $O(\log^3 n)$ for a sequence of at least m_0 updates, where m_0 is the number of edges in the initial graph, and a query time of $O(\log n)$. It needs $\Theta(m + n \log n)$ space. We now sketch the main ideas behind this technique.

Let $G = (V, E)$ be a graph to be maintained dynamically, and let F be a spanning forest of G. We call edges in F *tree edges*, and edges in $E \setminus F$ *non–tree edges*. First, we describe a data structure which stores all trees in the spanning forest F. This data structure is based on Euler Tours, and allows one to obtain logarithmic updates and queries within the forest. Next, we show how to keep the forest F spanning throughout a sequence of updates. The Euler Tour data structure for a tree T is simple, and consists of storing the tree vertices according to an Euler Tour of T. Each time a vertex v is visited in the Euler Tour, we call this an *occurrence* of v and we denote it by $o(v)$. A vertex of degree Δ has exactly Δ occurrences, except for the root which has $(\Delta + 1)$ occurrences. Furthermore, each edge is visited exactly twice. Given an n–nodes tree T, we encode it with the sequence of $2n - 1$ symbols produced by procedure ET. This encoding is referred to as $ET(T)$. We now analyze how to update $ET(T)$ when T is subject to dynamic edge operations.

If an edge $e = (u, v)$ is deleted from T, denote by T_u and T_v the two trees obtained after the deletion, with $u \in T_u$ and $v \in T_v$. Let $o(u_1)$, $o(v_1)$, $o(u_2)$ and $o(v_2)$ be the occurences of u and v encountered during the visit of (u, v). Without loss of generality assume that $o(u_1) < o(v_1) < o(v_2) < o(u_2)$ so that $ET(T) = \alpha o(u_1) \beta o(v_1) \gamma o(v_2) \delta o(u_2) \epsilon$. $ET(T_u)$ and $ET(T_v)$ can be easily computed from $ET(T)$, as $ET(T_v) = o(v_1)\gamma o(v_2)$, and $ET(T_v) = \alpha o(u_1)\beta\delta o(u_2)\epsilon$. To change the root of T from r to another vertex s, we do the following. Let $ET(T) = o(r)\alpha o(s_1)\beta$, where $o(s_1)$ is any occurrence of s. Then, the new encoding will be $o(s_1)\beta\alpha o(s)$, where $o(s)$ is a newly created occurrence of s that is added at the end of the new sequence. If two trees T_1 and T_2 are joined in a new tree T because of a new edge $e = (u, v)$, with $u \in T_1$ and $v \in T_2$, we first reroot T_2 at v. Now, given $ET(T_1) = \alpha o(u_1)\beta$ and the computed encoding $ET(T_2) = o(v_1)\gamma o(v_2)$, we compute $ET(T) = \alpha o(u_1)o(v_1)\gamma o(v_2)o(u)\beta$, where $o(u)$ is a newly created occurrence of vertex u.

Note that all the above primitives require the following operations: (i) splicing out an interval from a sequence, (ii) inserting an interval into a sequence, (iii) inserting or (iv) deleting a single occurrence from a sequence. If the sequence $ET(T)$ is stored in a balanced search tree of degree d (i.e., a balanced d–ary search tree), then one may insert or splice an interval, or insert or delete an occurrence in time $O(d \log n / \log d)$, while maintaining the balance of the tree. It can be checked in $O(\log n/d)$ whether two elements are in the same tree, or whether one element precedes the other in the ordering. The balanced d–ary search tree that stores $ET(T)$ is referred to as the *ET(T)–tree*.

We augment ET–trees to store non–tree edges as follows. For each occurrence of vertex $v \in T$, we arbitrarily select one occurrence to be the *active occurrence* of v. The list of non–tree edges incident to v is stored in the active occurrence of v: each node in the $ET(T)$–tree contains the number of non–tree edges and active occurrences stored in its subtree; thus the root of the $ET(T)$–tree contains the weight and size of T.

Using these data structures, we can implement the following operations on a collection of trees:

- *tree(x)*: return a pointer to $ET(T)$, where T is the tree containing vertex x.
- *non_tree_edges(T)*: return the list of non–tree edges incident to T.
- *sample_n_test(T)*: select one non–tree edge incident to T at random, where an edge with both endpoints in T is picked with probability $2/w(T)$, and an edge with only endpoint in T is picked with probability $1/w(T)$. Test whether the edge has exactly one endpoint in T, and if so return this edge.
- *insert_tree(e)*: join by edge e the two trees containing its endpoints. This operation assumes that the two endpoints of e are in two different trees of the forest.

- *delete_tree(e)*: remove e from the tree that contains it. This operation assumes that e is a tree edge.
- *insert_non_tree(e)*: insert a non–tree edge e. This operation assumes that the two endpoints of e are in a same tree.
- *delete_non_tree(e)*: remove the edge e. This operation assumes that e is a non–tree edge.

Using a balanced binary search tree for representing $ET(T)$, yields the following running times for the above operations: *sample_n_test(T)*, *insert_tree(e)*, *delete_tree(e)*, *insert_non_tree(e)*, and *delete_non_tree(e)* in $O(\log n)$ time, and *non_tree_edges(T)* in $O(w(T) \log n)$ time.

We now turn to the problem of keeping the forest of G spanning throughout a sequence of updates. Note that the hard operation is a deletion of a tree edge: indeed, as shown in Section 4 a spanning forest is easily maintained throughout edge insertions, and deleting a non–tree edge does not change the forest. Let $e = (u, v)$ be a tree edge of the forest F, and let T_e be the tree of F containing edge e. Let T_u and T_v the two trees obtained from T after the deletion of e, such that T_u contains u and T_v contains v. When e is deleted, T_u and T_v can be reconnected if and only if there is a non–tree edge in G with one endpoint in T_u and one endpoint in T_v. We call such an edge a *replacement edge for e*. In other words, if there is a replacement edge for e, T is reconnected via this replacement edge; otherwise, the deletion of e disconnects T into T_u and T_v. The set of all the replacement edges for e (i.e., all the possible edges reconnecting T_u and T_v), is called the *candidate set of e*.

One main idea behind the technique of Henzinger and King is the following: when e is deleted, use random sampling among the non–tree edges incident to T_e, in order to find quickly whether there exists a replacement edge for e. Using the Euler Tour data structure, a single random edge adjacent to T_e can be selected and tested whether it reconnects T_e in logarithmic time. The goal is an update time of $O(\log^3 n)$, so we can afford a number of sampled edges of $O(\log^2 n)$. However, the candidate set of e might only be a small fraction of all non–tree edges which are adjacent to T. In this case it is unlikely to find a replacement edge for e among the sampled edges. If we found no candidate among the sampled edges, we check explicitly all the non–tree edges adjacent to T. After random sampling has failed to produce a replacement edge, we need to perform this check explicitly, otherwise we would not be guaranteed to provide correct answers to the queries. Since there might be a lot of edges which are adjacent to T, this explicit check could be an expensive operation, so it should be made a low probability event for the randomized algorithm. This is not yet true, however, since deleting all edges in a relatively small candidate set, reinserting them, deleting them again, and so on will almost surely produce many of those unfortunate events.

The second main idea prevents this undesirable behavior: we maintain an edge decomposition of the current graph G into $O(\log n)$ edge disjoint subgraphs $G_i = (V, E_i)$. These subgraphs are hierarchically ordered. Each i corresponds to a *level.* For each level i, there is a forest F_i such that the union $\cup_{i \leq k} F_i$ is a spanning forest of $\cup_{i \leq k} G_i$; in particular the union F of all F_i is a spanning forest of G. A *spanning forest at level i* is a tree in $\cup_{j \leq i} F_j$. The *weight* $w(T)$ of a spanning tree T at level i is the number of pairs (e', v) such that e' is a non–tree edge in G_i adjacent to the node v in T. If T_1 and T_2 are the two trees resulting from the deletion of e, we sample edges adjacent to the tree with the smaller weight. If sampling is unsuccessful due to a candidate set which is non–empty but relatively small, then the two pieces of the tree which was split are reconnected on the next higher level using one candidate, and all other candidate edges are copied to that level. The idea is to have sparse cuts on high levels and dense cuts on low levels. Non–tree edges always belong to the lowest level where their endpoints are connected or a higher level, and we always start sampling at the level of the deleted tree edge. After moving the candidates one level up, they are normally no longer a small fraction of all adjacent non–tree edges at the new level. If the candidate set on one level is empty, we try to sample on the next higher level. There is one more case to mention: if sampling was unsuccessful despite the fact that the candidate set was big enough, which means that we had bad luck in the sampling, we do not move the candidates to the next level, since this event has a small probability and does not happen very frequently. We present the pseudocode for *replace(u,v,i)*, which is called after the deletion of the forest edge $e = (u, v)$ on level i:

replace(u,v,i)

```
1. Let $T_u$ and $T_v$ be the spanning trees at level $i$ containing $u$ and $v$, respectively. Let $T$ be the
   tree with smaller weight among $T_u$ and $T_v$. Ties are broken arbitrarily.

2. If $w(T) > \log^2 n$ then

    (a) Repeat sample_n_test($T$) for at most $16 \log^2 n$ times. Stop if a replacement edge $e$ is
```

```
        found.
    (b) If a replacement edge $e$ is found then do delete_non_tree($e$), insert_tree($e$), and return.
3.  (a) Let $S$ be the set of edges with exactly one endpoint in $T$.
    (b) If $|S| \geq w(T)/(8 \log n)$ then
        Select one $e \in S$, delete_non_tree($e$), and insert_tree($e$).
    (c) Else if $0 < |S| < w(T)/(8 \log n)$ then
        Delete one edge $e$ from $S$, delete_non_tree($e$), and insert_tree($e$) in level $(i+1)$.
        Forall $e' \in S$ do delete_non_tree($e'$) and insert_non_tree($e'$) in level $(i+1)$.

    (d) Else if $i < l$ then replace($u, v, i+1$).
```

Note that edges may migrate from level 1 to level l, one level at the time. However, an upper bound of $O(\log n)$ for the number of levels is guaranteed, if there are only deletions of edges. This can be proved as follows. For any i, let m_i be the number of edges ever in level i.

Lemma 11 *For any level i, and for all smaller trees T_1 on level i, $\sum w(T_1) \leq 2m_i \log n$.*

Proof: Let T be a tree which is split into two trees: if an endpoint of an edge is contained in the smaller split tree, the weight of the tree containing the endpoint is at least halved. Thus, each endpoint of a non–tree edge is incident to a small tree at most $\log n$ times in a given level i and the lemma follows. $\Box$

Lemma 12 *For any level i, $m_i \leq m/4^{i-1}$.*

Proof: We proceed by induction on i. The lemma trivially holds for $i = 1$. Assume it holds for $(i-1)$. When summed over all small trees T_1 on level $(i-1)$, at most $\sum w(T_1)/(8 \log n)$ edges are added to level i. By Lemma 11, $\sum w(T_1) \leq 2m_{i-1} \log n$, where m_{i-1} is the number of edges ever in level $(i-1)$. The lemma now follows from the induction step. $\Box$

The following is an easy corollary of Lemma 12:

Corollary 1 *The sum over all levels of the total number of edges is $\sum_i m_i = O(m)$*

Lemma 12 gives immediately a bound on the number of levels:

Lemma 13 *The number of levels is at most* $l = \lceil \log m - \log\log n \rceil + 1$.

Proof: Since edges are never moved to a higher level from a level with less than $2\log n$ edges, Lemma 12 implies that all edges of G are contained in some E_i, $i \le \lceil \log m - \log\log n \rceil + 1$. $\square$

We are now ready to describe an algorithm for maintaining a spanning forest of a graph G subject to edge deletions. Initially, we compute a spanning forest F of G, compute $ET(T)$ for each tree in the forest, and select active occurrences for each vertex. For each i, the spanning forest at level i is initialized to F. Then, we insert all the non–tree edges with the proper active occurrences into level 1, and compute the number of non–tree edges in the subtree of each node of the binary search tree. This requires $O(m+n)$ times to find the spanning forest and initialize level 1, plus $O(n)$ for each subsequent level to initialize the spanning forest at that level. To check whether two vertices x and y are connected, we test if *tree(x)=tree(y)* on the last level. This can be done in time $O(\log n)$. To update the data structure after the deletion of an edge $e = (u, v)$, we do the following. If e is a non–tree edge, it is enough to perform a *delete_non_tree(e)* in the level where e appears. If e is a tree edge, let i be the level where e first appears. We do a *delete_tree(e)* at level j, for $j \ge i$, and then call *replace*(u, v, i). This yields the following bounds.

Theorem 9 (Henzinger and King [24]) *Let G be a graph with m_0 edges and n vertices subject to edge deletions only. A spanning forest of G can be maintained in $O(\log^3 n)$ expected amortized time per deletion, if there are at least $\Omega(m_0)$ deletions. The time per query is $O(\log n)$.*

Proof: The bound for queries follows from the above argument. Let e be an edge to be deleted. If e is a non–tree edge, its deletion can be taken care of in $O(\log n)$ time via a *delete_non_tree* primitive.

If e is a tree edge, let T_1 and T_2 the two trees created by the deletion of e, $w(T_1) \le w(T_2)$. During the sampling phase we spend exactly $O(\log^3 n)$ time, as the cost of *sample_n_test* is $O(\log n)$ and we repeat this for at most $16\log^2 n$ times.

If the sampling is not successful, collecting and testing all the non–tree edges incident to T_1 implies a total cost of $O(w(T_1)\log n)$. We now distinguish two cases. If we are unlucky in the sampling, $|S| \ge w(T_1)/(8\log n)$: this happens with probability at most $(1 - 1/(8\log n))^{16\log^2 n} =$

$O(1/n^2)$ and thus contributes an expected cost of $O(\log n)$ per operation. If the cut S is sparse, $|S| < w(T_1)/(8\log n)$, and we move the candidate set for e one level higher. Throughout the sequence of deletions, the cost incurred at level i for this case is $\sum w(T_1)\log n$, where the sum is taken over the small trees T_1 on level i. By Lemma 13 and Corollary 1 this gives a total cost of $O(m_0 \log^2 n)$.

In all cases where a replacement edge is found, $O(\log n)$ tree operations are performed in the different levels, contributing a cost of $O(\log^2 n)$. Hence, each tree edge deletion contributes a total of $O(\log^3 n)$ expected amortized cost towards the sequence of updates. $\square$

If there are also insertions, however, the analysis in Theorem 9 does not carry through, as the upper bound on the number of levels in the graph decomposition is no longer guaranteed. To achieve the same bound, there have to be periodical rebuilds of parts of the data structure. This is done as follows. We let the maximum number of levels to be $l = \lceil 2\log n \rceil$. When an edge $e = (u, v)$ is inserted into G, we add it to the last level l. If u and v were not previously connected, then we do this via a *tree_insert*, otherwise we perform a *non_tree_insert*. In order to prevent the number of levels to grow behind their upper bound, a rebuild of the data structure is executed periodically. A *rebuild at level* i, $i \geq 1$, is carried out by moving all the tree and non–tree edges in level j, $j > i$, back to level i. Also, for each $j > i$ all the tree edges in level i are inserted into level j. Note that after a rebuild at level i, $E_j = \emptyset$ and $F_j = F_i$ for all $j > i$, i.e., there are no non–tree edges above level i, and the spanning trees on level $j \geq i$ span G.

The crucial point of this method is deciding when to apply a rebuild at level i. This is done as follows. We keep a counter K that counts the number of edge insertions modulo $2^{\lceil 2\log n \rceil}$ since the start of the algorithm. Let $K_1, K_2, \ldots, K_l$ be the binary rapresentation of K, with K_1 being the most significant bit: we perform a rebuild at level i each time the bit K_i flips to 1. This implies that the last level is rebuilt every other insertion, level $(l-1)$ every four insertions. In general, a rebuild at level i occurs every 2^{l-i+1} insertions. We now show that the rebuilds contribute a total cost of $O(\log^3 n)$ toward a sequence of insertions and deletions of edges.

For the sake of completeness, assume that at the beginning the initialization of the data structures is considered a rebuild at level 1. Given a level i, we define an *epoch for* i to be any period of time starting right after a rebuild at level j, $j \leq i$, and ending right after the next rebuild at level j', $j' \leq i$. Namely, an epoch for i is the period between two consecutive rebuilds below or at level

i: an epoch for i starts either at the start of the algorithm or right after some bit K_j, $j \leq i$, flips to 1, and ends with the next such flip. It can be easily seen that each epoch for i occurs every 2^{l-i} insertions, for any $1 \leq i \leq l$. There are two types of epochs for i, depending on whether it is bit K_i or a bit K_j, $j > i$, that flips to 1: an *empty* epoch for i starts right after a rebuild at j, $j < i$, and a *full* epoch for i starts right after a rebuild at i. At the time of the initialization, a *full* epoch for 1 starts, while for $i \geq 2$, *empty* epochs for i starts. The difference between these two types of epochs is the following. When an *empty* epoch for i starts, all the edges at level i have been moved to some level j, $j < i$, and consequently E_i is empty. On the contrary, when a *full* epoch for i starts, all the edges at level k, $k > i$, have been moved to level i, and thus $E_i \neq \emptyset$.

An important point in the analysis is that for any $i \geq 2$, any epoch for $(i-1)$ consists of two parts, one corresponding to an *empty* epoch for i followed by another corresponding to a *full* epoch for i. This happens because a flip to 1 of a bit K_j, $j \leq 2$, must be followed by a flip to 1 of K_i before another bit $K_{j'}$, $j' \leq i$ flips again to 1. Thus, when each epoch for $(i-1)$ starts, E_i is empty. Define m'_i to be the number of edges ever in level i during an epoch for i. The following lemma is the analogous of Lemma 11.

Lemma 14 *For any level i, and for all smaller trees T_1 on level i searched during an epoch for i, $\sum w(T_1) \leq 2m'_i \log n$.*

Lemma 15 $m'_i < n^2/2^{i-1}$.

Proof: To prove the lemma, it is enough to bound the number of edges that are moved to level i during any one epoch for $(i-1)$, as $E_i = \emptyset$ at the start of each epoch for $(i-1)$ and each epoch for i is contained in one epoch for $(i-1)$. Consider an edge e that is in level i during one epoch for $(i-1)$. There are only two possibilities: either e was passed up from level $(i-1)$ because of an edge deletion, or e was moved down during a rebuild at i. Assume that e was moved down during a rebuild at i: since right after the rebuild at i the second part of the epoch for $(i-1)$ (i.e., the *full* epoch for i) starts, e was moved back to level i still during the *empty* epoch for i. Note that E_k, for $k \geq i$, was empty at the beginning of the epoch for $(i-1)$; consequently, either e was passed up from E_{i-1} to E_i or e was inserted into G during the *empty* epoch for i. In summary, denoting by a_{i-1} the maximum number of edges passed up from E_{i-1} to E_i during one epoch for $(i-1)$, and by b_i the number of edges inserted into G during one epoch for i, we have that $m'_i \leq a_{i-1} + b_i$.

By definition of epoch, the number of edges inserted during an epoch for i is $b_i = 2^{l-i}$. It remains for us to bound a_{i-1}. Applying the same argument as in the proof of Lemma 12, using this time Lemma 14, yields that $a_{i-1} \leq m'_{i-1}/4$. Substituting for a_{i-1} and b_i yields $m'_i \leq m'_{i-1}/4 + 2^{l-i}$, with $m'_1 \leq n^2$. Since $l = \lceil 2 \log n \rceil$, $m'_i < n^2/2^{i-1}$. $\square$

Lemma 15 implies that $m'_l \leq 2$, and thus edges never need to be passed to a level higher than l.

Corollary 2 *All edges of G are contained in some level E_i, $i \leq \lceil 2 \log n \rceil$.*

We are now ready to analyze the running time of the entire algorithm.

Theorem 10 (Henzinger and King [24]) *Let G be a graph with m_0 edges and n vertices subject to edge insertions and deletions. A spanning forest of G can be maintained in $O(\log^3 n)$ expected amortized time per update, if there are at least $\Omega(m_0)$ updates. The time per query is $O(\log n)$.*

Proof: There are two main differences with the algorithm for deletions only described in Theorem 9. The first is that now the actual cost of an insertion has to be taken into account (i.e., the cost of operation *move_edges*). The second difference is that the argument that a total of $O(m_i \log n)$ edges are examined throughout the course of the algorithm when sparse cuts are moved one level higher must be modified to take into account epochs.

The cost of executing *move_edges(i)* is the cost of moving each non–tree and tree edge from E_j to E_i, for all $j > i$, plus the cost of updating all the forests F_k, $i \leq k < j$. The number of edges moved into level i by a *move_edges(i)* is $\sum_{j>i} m'_j$, which by Lemma 15, is never greater than $n^2/2^{i-1}$. Since moving one edge costs $O(\log n)$, the total cost incurred by a *move_edges(i)* operation is $O(n^2 \log n/2^i)$. Note that during an epoch for i, at most one *move_edges(i)* can be performed, since that will end the current epoch and start a new one.

Inserting a tree edge into a given level costs $O(\log n)$. Since a tree edge is never passed up during edge deletions, it can be added only once to a given level. This yields a total of $O(\log^2 n)$ per tree edge.

We now analyze the cost of collecting and testing the edges from all smaller trees T_1 on level i during an epoch for i (when sparse cuts for level i are moved to level $(i+1)$). Fix a level $i \leq l$. If $i = l$, since there are $O(1)$ edges in E_l at any given time, the total cost for collecting and testing on level l will be $O(\log n)$. If $i < l$, the cost of collecting and testing edges on all small trees on level i during an epoch for i is $O(2m'_i \log n \times \log n)$ because of Lemma 14. By Lemma 15, this is $O(n^2 \log^2 n/2^i)$.

In summary, each update contributes $O(\log^3 n)$ per operation plus $O(n^2 \log^2 n/2^i)$ per each epoch for i, $1 \leq i \leq l$. To amortized the latter bound against insertions, during each insertion we distribute $\Theta(\log^2 n)$ credits per level. This sums up to $\Theta(\log^3 n)$ credits per insertion. An epoch for i occurs every $2^{l-i} = \Theta(n^2/2^i)$ insertions, at which time level i has accumulated $\Theta(n^2 \log^2 n/2^i)$ credits to to pay for the cost of *move_edges(i)* and the cost of collecting and testing edges on all small trees. $\square$

We end this section by mentioning that Henzinger and Thorup have recently improved the update bound of Theorem 10 from $O(\log^3 n)$ to $O(\log^2 n)$. We refer the interested reader to [27] for the details.

6 Research Issues and Summary

In this chapter we have described the most efficient known algorithms for maintaining dynamic graphs. Experimental comparison of some of the dynamic connectivity algorithms has recently been performed by Alberts et al. [1], who showed that in the average case for sufficiently random inputs, a simple sparsification tree based on edge subdivision performs as well as the vertex subdivision method we described in Theorem 6. Furthermore, they compared this simplified sparsification algorithm (having a worst–case update bound of $O(n \log(m/n))$) with the randomized method of Henzinger and King (having an expected amortized bound of $O(\log^3 n)$). The sparsification method worked well for small update sequences, but the other method was faster on longer sequences. It remains to see how well stable sparsification would perform in similar experiments.

There are still several open questions. For some of the fully dynamic problems described in this chapter, such as connectivity and 2–connectivity, polylogarithmic randomized algorithms are available. However, the deterministic bounds for the same problems have higher running times. Are there any faster deterministic algorithms for these problems? Furthermore, no randomized algorithm is known for the fully dynamic maintenance of a minimum spanning tree, and the fastest algorithm requires $O(n^{1/2})$ time per update. Is there any faster randomized algorithm for this? Very little is known about lower bounds for fully dynamic graph problems. The only nontrivial lower bounds known are the $\Omega(\log n/\log\log n)$ lower bounds of Fredman and Henzinger [16] for the cell–probe model of computation. While the randomized algorithms described in this chapter are close to these lower bounds, there is still a big gap for the deterministic algorithms. Can the gap between upper and lower bounds be tightened in this case? Furthermore, can we prove nontrivial lower bounds for other fully dynamic problems as well? Can we allow other update operations besides edge insertion and deletion? Usually, isolated vertices can be inserted and deleted in the same times as edge insertion and deletion. Is there some way of allowing rapid deletion of vertices that may still be connected to many edges?

7 Defining Terms

Certificate For any graph property $\mathcal{P}$, and graph G, a certificate for G is a graph G' such that G has property $\mathcal{P}$ if and only if G' has the property.

Fully Dynamic Graph Problem Problem where the update operations include unrestricted insertions and deletions of edges.

Partially Dynamic Graph Problem Problem where the update operations include either edge insertions (*incremental*) or edge deletions (*decremental*).

Sparsification Technique for designing dynamic graph algorithms, which when applicable transform a time bound of $T(n,m)$ into $O(T(n,n))$, where m is the number of edges, and n is the number of vertices of the given graph.

Topology Tree Tree that describes a balanced decomposition of another tree, according to its topology.

References

[1] D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. In *Proc. 7th ACM-SIAM Symp. Discrete Algorithms*, pages 192–201, 1996.

[2] G. Ausiello, G. F. Italiano, A. Marchetti Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *J. Algorithms*, 12:615–638, 1991.

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

[4] F. Chin and D. Houk. Algorithms for updating minimum spanning trees. *J. Comp. Syst. Sci.*, 16:333–344, 1978.

[5] R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *Proc. 2nd ACM-SIAM Symp. Discrete Algorithms*, pages 52–61, 1991.

[6] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proc. 30th IEEE Symp. Foundations of Computer Science*, pages 436–441, 1989.

[7] G. Di Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In *Proc. 17th Int. Colloquium on Automata, Languages and Programming*, pages 598–611. Lecture Notes in Computer Science 443, Springer-Verlag, Berlin, 1990.

[8] E. A. Dinitz. Maintaining the 4-edge-connected components of a graph on-line. In *Proc. 2nd Israel Symp. Theory of Computing and Systems*, pages 88–99, 1993.

[9] D. Eppstein. Clustering for faster network simplex pivots. In *Proc. 5th ACM-SIAM Symp. Discrete Algorithms*, pages 160–166, 1994.

[10] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. Revised manuscript, 1996. See also *Proc. 33rd IEEE Symp. Foundations of Computer Science*, pages 60–69, 1992.

[11] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification for dynamic planar graph algorithms. In *Proc. 25th ACM Symp. Theory of Computing*, pages 208–217, 1993.

[12] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13:33–54, 1992.

[13] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

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

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

[16] M. L. Fredman and M. R. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica.* To appear.

[17] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problems. In *Proc. 7th ACM-SIAM Symp. Discrete Algorithms*, pages 212–221, 1996.

[18] H. N. Gabow and M. Stallman. Efficient algorithms for graphic matroid intersection and parity. In *Proc. 12th Int. Coll. Automata, Languages, and Programming*, pages 210–220. Lecture Notes in Computer Science 194, Springer-Verlag, Berlin, 1985.

[19] Z. Galil and G. F. Italiano. Fully dynamic algorithms for 2-edge-connectivity. *SIAM J. Comput.*, 21:1047–1069, 1992.

[20] Z. Galil and G. F. Italiano. Maintaining the 3-edge-connected components of a graph on-line. *SIAM J. Comput.*, 22:11–28, 1993.

[21] Z. Galil, G. F. Italiano, and N. Sarnak. Fully dynamic planarity testing. In *Proc. 24th ACM Symp. Theory of Computing*, pages 495–506, 1992.

[22] D. Giammarresi and G. F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica.* To appear. See also *Proc. 3rd Scandinavian Workshop on Algorithm Theory*, pages 221–232. Lecture Notes in Computer Science 621, Springer-Verlag, Berlin, 1992.

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

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

[25] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. operation. In *Proc. Proc. 36th IEEE Symp. Foundations of Computer Science*, pages 664–672, 1995.

[26] M. R. Henzinger and J. A. La Poutré. Certificates and fast algorithms for biconnectivity in fully dynamic graphs. In *Proc. 3rd European Symp. on Algorithms*, pages 171–184. Lecture Notes in Computer Science 979, Springer-Verlag, Berlin, 1995.

[27] M. R. Henzinger and M. Thorup. Improved sampling with applications to dynamic graph algorithms. In *Proc. 23rd Int. Colloquium on Automata, Languages and Programming*, 1996.

[28] J. Hershberger, M. Rauch, and S. Suri. Data structures for two-edge connectivity in planar graphs. *Theor. Comp. Sci.*, 130:139–161, 1994.

[29] T. Ibaraki and N. Katoh. On-line computation of transitive closure for graphs. *Inform. Proc. Lett.*, 16:95–97, 1983.

[30] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48:273–281, 1986.

[31] G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Inform. Proc. Lett.*, 28:5–11, 1988.

[32] G. F. Italiano, J. A. La Poutré, and M. Rauch. Fully dynamic planarity testing in planar embedded graphs. In *Proc. 1st European Symp. Algorithms*, pages 212–223. Lecture Notes in Computer Science 726, Springer-Verlag, Berlin, 1993.

[33] A. Kanevsky, R. Tamassia, G. Di Battista, and J. Chen. On-line maintenance of the four-connected components of a graph. In *Proc. 32nd IEEE Symp. Foundations of Computer Science*, pages 793–801, 1991.

[34] P. N. Klein and S. Sairam. Fully dynamic approximation schemes for shortest path problems in planar graphs. In *Proc. 3rd Worksh. Algorithms and Data Structures*, pages 442–451. Lecture Notes in Computer Science 709, Springer-Verlag, Berlin, 1993.

[35] J. A. La Poutré. Maintenance of 2- and 3-connected components of graphs, part II: 2- and 3-edge-connected components and 2-vertex-connected components. Technical Report ALCOM-91-145, Department of Computer Science, Utrecht University, 1991.

[36] J. A. La Poutré. Maintenance of triconnected components of graphs. In *Proc. 19th Int. Colloquium on Automata, Languages and Programming*, pages 354–365. Lecture Notes in Computer Science 623, Springer-Verlag, Berlin, 1992.

[37] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Lecture Notes in Computer Science 314, Springer-Verlag, Berlin, 1988.

[38] J. A. La Poutré, J. van Leeuwen, and M. H. Overmars. Maintenance of 2- and 3-connected components of graphs, part I: 2- and 3-edge-connected components. *Discrete Mathematics*, 114:329–359, 1993.

[39] C. C. Lin and R. C. Chang. On the dynamic shortest path problem. *J. Information Processing*, 13:470–476, 1990. See also *Int. Worksh. Discrete Algorithms and Complexity*, pages 203–212, 1989.

[40] G. Ramalingam. *Bounded incremental compilation.* PhD thesis, Department of Computer Science, University of Wisconsin at Madison, August 1993.

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

[42] M. Rauch. Improved data structures for fully dynamic biconnectivity. In *Proc. 26th Symp. Theory of Computing*, 1994.

[43] H. Rohnert. A dynamization of the all pairs least cost path problem. In *Proc. 2nd Symp. Theoretical Aspects of Computer Science*, pages 279–286. Lecture Notes in Computer Science 182, Springer-Verlag, Berlin, 1985.

[44] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comp. Syst. Sci.*, 24:362–381, 1983.

[45] P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM J. Comput.*, 4:375–380, 1975.

[46] R. Tamassia. A dynamic data structure for planar graph embedding. In *Proc. 15th Int. Colloquium on Automata, Languages and Programming*, pages 576–590. Lecture Notes in Computer Science 317, Springer-Verlag, Berlin, 1988.

[47] R. Tamassia and F. P. Preparata. Dynamic maintenance of planar digraphs, with applications. *Algorithmica*, 5:509–527, 1990.

[48] R. Tamassia and I. G. Tollis. Dynamic reachability in planar digraphs with one source and one sink. *Theor. Comput. Sci.*, 119:331–344, 1993.

[49] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[50] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. Assoc. Comput. Mach.*, 31:245–281, 1984.

[51] J. Westbrook. Fast incremental planarity testing. In *Proc. 19th Int. Colloquium on Automata, Languages and Programming*, pages 342–353. Lecture Notes in Computer Science, Springer-Verlag 623, Berlin, 1992.

[52] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

[53] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.

Further Information

Research on dynamic graph algorithms is published in many computer science journals, including "Algorithmica", "Journal of ACM", "Journal of Algorithms" and "SIAM Journal on Computing". Work on this area is published also in the proceedings of general theoretical computer science conferences, such as the "ACM Symposium on Theory of Computing", the "IEEE Symposium on Foundations of Computer Science" and the "International Colloquium on Automata, Languages and Programming". More specialized conferences devoted exclusively to algorithms are the "ACM–SIAM Symposium on Discrete Algorithms" and the "European Symposium on Algorithms".