9

Dynamic Graph Algorithms

Camil Demetrescu
University of Rome "La Sapienza"

David Eppstein
University of California

Zvi Galil
Tel Aviv University

Giuseppe F. Italiano
University of Rome "Tor Vergata"

9.1 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 decades 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 describe fully dynamic algorithms for graph problems. We present dynamic algorithms for undirected graphs in Section 9.2. Dynamic algorithms for directed graphs are next described in Section 9.3. Finally, in Section 9.4 we describe some open problems.

9.2 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 aforementioned 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 be 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 describe is **sparsification** by Eppstein et al. [16]. 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 second technique we present in this section is due to Henzinger and King [39], 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 $(i + 1)$.

The last technique presented in this section is due to Holm et al. [46], which is still based on graph decomposition, but derandomizes the previous approach by Henzinger and King. Besides derandomizing the bounds, this technique is able to achieve faster running times.

One particular dynamic graph problem that has been thoroughly investigated is the maintenance of a minimum spanning forest. This is an important problem on its own, but it also has an impact on other problems. 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 [16,25,31,37,38,44]. Thus, we will focus our attention to the fully dynamic maintenance of minimum spanning trees.

Extensive computational studies on dynamic algorithms for undirected graphs appear in [1,2,49].

9.2.1 Sparsification

In this section we describe a general technique for designing dynamic graph algorithms, due to Eppstein et al. [16], which is known as 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 $T(n, O(n))$; 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 known as 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 $\log(m/n)$* 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 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 9.1 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.

Nagamochi and Ibaraki [62] 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 9.2 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 9.2.

FACT 9.1 *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.*

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

FACT 9.2 *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 9.1 In the sparsification tree described earlier, 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. □

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 9.1 *[16] 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 9.1 and 9.2, 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 $O(f(n, O(n))) + g(n, cn)$. □

This technique is very effective in 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 9.3 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 9.2 *[16] 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 earlier. 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 earlier 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. □

Theorem 9.1 can be used to dynamize static algorithms, while Theorem 9.2 can be used to speed up existing fully dynamic algorithms. In order to apply effectively Theorem 9.1 we only need to compute efficiently sparse certificates, while for Theorem 9.2 we need to maintain efficiently stable sparse certificates. Indeed stability plays an important role in the proof of Theorem 9.2. 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. [16] 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 9.2 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. □

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 9.2, proves that they satisfy the definition of stability:

LEMMA 9.3 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 9.2, and part (2) follows from Lemma 9.3.

We use Frederickson's algorithm [24,25] that maintains a minimum spanning trees 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 9.2, with $f(n, m) = g(n, m) = O(m^{1/2})$.

THEOREM 9.3 *[16] 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 [16]. 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 [16,38,44].

9.2.2 Randomized Algorithms

All the previous techniques yield efficient deterministic algorithms for fully dynamic problems. Recently, Henzinger and King [39] proposed a new approach that, exploiting the power of randomization, is able to achieve faster update times for some problems. For the fully dynamic connectivity problem, the randomized technique of Henzinger and King 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. The interested reader is referred to the chapter on Randomized Algorithms for basic definitions on randomized algorithms.

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 $E \setminus F$ nontree 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 occurrences 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 aforementioned primitives require the following operations: (1) splicing out an interval from a sequence, (2) inserting an interval into a sequence, (3) inserting a single occurrence into a sequence, or (4) 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 nontree 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 nontree edges incident to v is stored in the active occurrence of v: each node in the $ET(T)$-tree contains the number of nontree 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 nontree edges incident to T.
- *sample_n_test(T)*: select one nontree 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 nontree 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 nontree edge.

Using a balanced binary search tree for representing $ET(T)$, yields the following running times for the aforementioned 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, a spanning forest is easily maintained throughout edge insertions, and deleting a nontree 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 be 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 nontree 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 nontree 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 nontree 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 find no candidate among the sampled edges, we check explicitly all the nontree 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 nontree 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 nonempty 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. Nontree 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 nontree 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 although 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 9.4 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 nontree edge is incident to a small tree at most $\log n$ times in a given level i and the lemma follows. □

LEMMA 9.5 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 9.4, $\sum w(T_1) \leq 2m_{i-1} \log n$, where m_{i-1} is the number of edges ever in level $(i-1)$. The lemma now follows from the induction step. □

The following is an easy corollary of Lemma 9.5:

COROLLARY 9.1 The sum over all levels of the total number of edges is $\sum_i m_i = O(m)$.

Lemma 9.5 gives immediately a bound on the number of levels:

LEMMA 9.6 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 9.5 implies that all edges of G are contained in some E_i, $i \leq \lceil \log m - \log \log n \rceil + 1$. □

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 nontree edges with the proper active occurrences into level 1, and compute the number of nontree 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 nontree 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 $j \geq i$, and then call *replace(u, v, i)*. This yields the following bounds.

THEOREM 9.4 *[39] 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 aforementioned argument. Let e be an edge to be deleted. If e is a nontree 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, $w(T_1) \leq w(T_2)$. 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 nontree 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, $|S| \geq w(T_1)/(8 \log n)$: 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 9.6 and Corollary 9.1 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 toward the sequence of updates. □

If there are also insertions, however, the analysis in Theorem 9.4 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 nontree edges in level j, $j > i$, back to level i. Moreover, 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 nontree 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 representation 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 9.4.

LEMMA 9.7 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 9.8 $m'_i < n^2/2^{i-1}$.

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 9.5, using this time Lemma 9.7, 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 $l = \lceil 2 \log n \rceil$, $m'_i < n^2/2^{i-1}$. □

Lemma 9.8 implies that $m'_l \leq 2$, and thus edges never need to be passed to a level higher than l.

COROLLARY 9.2 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 9.5 *[39] 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 9.4. 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 nontree 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 9.8, 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 9.7. By Lemma 9.8, 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. □

9.2.3 Faster Deterministic Algorithms

In this section we give a high level description of the fastest deterministic algorithm for the fully dynamic connectivity problem in undirected graphs [46]: the algorithm, due to Holm, de Lichtenberg

and Thorup, answers connectivity queries in $O(\log n/\log\log n)$ worst-case running time while supporting edge insertions and deletions in $O(\log^2 n)$ amortized time.

Similarly to the randomized algorithm in [39], the deterministic algorithm in [46] maintains a spanning forest F of the dynamically changing graph G. As aforementioned, we will refer to the edges in F as *tree edges*. Let e be a tree edge of forest F, and let T be the tree of F containing it. When e is deleted, the two trees T_1 and T_2 obtained from T after the deletion of e can be reconnected if and only if there is a nontree edge in G with one endpoint in T_1 and the other endpoint in T_2. 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 creates a new connected component in G.

To accommodate systematic search for replacement edges, the algorithm associates to each edge e a level $\ell(e)$ and, based on edge levels, maintains a set of sub-forests of the spanning forest F: for each level i, forest F_i is the sub-forest induced by tree edges of level $\geq i$. If we denote by L denotes the maximum edge level, we have that:

$$F = F_0 \supseteq F_1 \supseteq F_2 \supseteq \cdots \supseteq F_L,$$

Initially, all edges have level 0; levels are then progressively increased, but never decreased. The changes of edge levels are accomplished so as to maintain the following invariants, which obviously hold at the beginning.

Invariant (1): F is a maximum spanning forest of G if edge levels are interpreted as weights.
Invariant (2): The number of nodes in each tree of F_i is at most $n/2^i$.

Invariant (1) should be interpreted as follows. Let (u, v) be a nontree edge of level $\ell(u, v)$ and let $u \cdots v$ be the unique path between u and v in F (such a path exists since F is a spanning forest of G). Let e be any edge in $u \cdots v$ and let $\ell(e)$ be its level. Due to (1), $\ell(e) \geq \ell(u, v)$. Since this holds for each edge in the path, and by construction $F_{\ell(u,v)}$ contains all the tree edges of level $\geq \ell(u, v)$, the entire path is contained in $F_{\ell(u,v)}$, i.e., u and v are connected in $F_{\ell(u,v)}$.

Invariant (2) implies that the maximum number of levels is $L \leq \lfloor \log_2 n \rfloor$.

Note that when a new edge is inserted, it is given level 0. Its level can be then increased at most $\lfloor \log_2 n \rfloor$ times as a consequence of edge deletions. When a tree edge $e = (v, w)$ of level $\ell(e)$ is deleted, the algorithm looks for a replacement edge at the highest possible level, if any. Due to invariant (1), such a replacement edge has level $\ell \leq \ell(e)$. Hence, a replacement subroutine `Replace((u,w),ℓ(e))` is called with parameters e and $\ell(e)$. We now sketch the operations performed by this subroutine.

`Replace((u,w),ℓ)` finds a replacement edge of the highest level $\leq \ell$, if any. If such a replacement does not exist in level ℓ, we have two cases: if $\ell > 0$, we recurse on level $\ell - 1$; otherwise, $\ell = 0$, and we can conclude that the deletion of (v, w) disconnects v and w in G.

During the search at level ℓ, suitably chosen tree and nontree edges may be promoted at higher levels as follows. Let T_v and T_w be the trees of forest F_ℓ obtained after deleting (v, w) and let, w.l.o.g., T_v be smaller than T_w. Then T_v contains at most $n/2^{\ell+1}$ vertices, since $T_v \cup T_w \cup \{(v, w)\}$ was a tree at level ℓ and due to invariant (2). Thus, edges in T_v of level ℓ can be promoted at level $\ell + 1$ by maintaining the invariants. Nontree edges incident to T_v are finally visited one by one: if an edge does connect T_v and T_w, a replacement edge has been found and the search stops, otherwise its level is increased by 1.

We maintain an Euler tour tree, as described before, for each tree of each forest. Consequently, all the basic operations needed to implement edge insertions and deletions can be supported in $O(\log n)$ time. In addition to inserting and deleting edges from a forest, ET-trees must also support

operations such as finding the tree of a forest that contains a given vertex, computing the size of a tree, and, more importantly, finding tree edges of level ℓ in T_v and nontree edges of level ℓ incident to T_v. This can be done by augmenting the ET-trees with a constant amount of information per node: we refer the interested reader to [46] for details.

Using an amortization argument based on level changes, the claimed $O(\log^2 n)$ bound on the update time can be finally proved. Namely, inserting an edge costs $O(\log n)$, as well as increasing its level. Since this can happen $O(\log n)$ times, the total amortized insertion cost, inclusive of level increases, is $O(\log^2 n)$. With respect to edge deletions, cutting and linking $O(\log n)$ forest has a total cost $O(\log^2 n)$; moreover, there are $O(\log n)$ recursive calls to `Replace`, each of cost $O(\log n)$ plus the cost amortized over level increases. The ET-trees over $F_0 = F$ allows it to answer connectivity queries in $O(\log n)$ worst-case time. As shown in [46], this can be reduced to $O(\log n / \log\log n)$ by using a $\Theta(\log n)$-ary version of ET-trees.

THEOREM 9.6 *[46] A dynamic graph G with n vertices can be maintained upon insertions and deletions of edges using $O(\log^2 n)$ amortized time per update and answering connectivity queries in $O(\log n / \log\log n)$ worst-case running time.*

9.3 Fully Dynamic Algorithms on Directed Graphs

In this section, we survey fully dynamic algorithms for maintaining path problems on general directed graphs. In particular, we consider two fundamental problems: fully dynamic transitive closure and fully dynamic all pairs shortest paths (APSP).

In the fully dynamic transitive closure problem we wish to maintain a directed graph $G = (V, E)$ under an intermixed sequence of the following operations:

Insert(x, y): insert an edge from x to y;

Delete(x, y): delete the edge from x to y;

Query(x, y): return *yes* if y is reachable from x, and return *no* otherwise.

In the fully dynamic APSP problem we wish to maintain a directed graph $G = (V, E)$ with real-valued edge weights under an intermixed sequence of the following operations:

Update(x, y, w): update the weight of edge (x, y) to the real value w; this includes as a special case both edge insertion (if the weight is set from $+\infty$ to $w < +\infty$) and edge deletion (if the weight is set to $w = +\infty$);

Distance(x, y): output the shortest distance from x to y.

Path(x, y): report a shortest path from x to y, if any.

Throughout the section, we denote by m and by n the number of edges and vertices in G, respectively.

Although research on dynamic transitive closure and dynamic shortest paths problems spans over more than three decades, in the last couple of years we have witnessed a surprising resurge of interests in those two problems. The goal of this section is to survey the newest algorithmic techniques that have been recently proposed in the literature. In particular, we will make a special effort to abstract some combinatorial properties, and some common data-structural tools that are at the base of those techniques. This will help us try to present all the newest results in a unifying framework so that they can be better understood and deployed also by nonspecialists.

We first list the bounds obtainable for dynamic transitive closure with simple-minded methods. If we do nothing during each update, then we have to explore the whole graph in order to answer

reachability queries: this gives $O(n^2)$ time per query and $O(1)$ time per update in the worst case. On the other extreme, we could recompute the transitive closure from scratch after each update; as this task can be accomplished via matrix multiplication [60], this approach yields $O(1)$ time per query and $O(n^\omega)$ time per update in the worst case, where ω is the best known exponent for matrix multiplication (currently $\omega < 2.736$ [5]).

For the incremental version of transitive closure, the first algorithm was proposed by Ibaraki and Katoh [48] in 1983: its running time was $O(n^3)$ over any sequence of insertions. This bound was later improved to $O(n)$ amortized time per insertion by Italiano [50] and also by La Poutré and van Leeuwen [57]. Yellin [72] gave an $O(m^*\delta_{\max})$ algorithm for m edge insertions, where m^* is the number of edges in the final transitive closure and $\delta_{\max}$ is the maximum out-degree of the final graph. All these algorithms maintain explicitly the transitive closure, and so their query time is $O(1)$.

The first decremental algorithm was again given by Ibaraki and Katoh [48], with a running time of $O(n^2)$ per deletion. This was improved to $O(m)$ per deletion by La Poutré and van Leeuwen [57]. Italiano [51] presented an algorithm which achieves $O(n)$ amortized time per deletion on directed acyclic graphs. Yellin [72] gave an $O(m^*\delta_{\max})$ algorithm for m edge deletions, where m^* is the initial number of edges in the transitive closure and $\delta_{\max}$ is the maximum out-degree of the initial graph. Again, the query time of all these algorithms is $O(1)$. More recently, Henzinger and King [39] gave a randomized decremental transitive closure algorithm for general directed graphs with a query time of $O(n/\log n)$ and an amortized update time of $O(n \log^2 n)$.

Despite fully dynamic algorithms were already known for problems on undirected graphs since the earlier 1980s [24], directed graphs seem to pose much bigger challenges. Indeed, the first fully dynamic transitive closure algorithm was devised by Henzinger and King [39] in 1995: they gave a randomized Monte Carlo algorithm with one-side error supporting a query time of $O(n/\log n)$ and an amortized update time of $O(n\widehat{m}^{0.58} \log^2 n)$, where $\widehat{m}$ is the average number of edges in the graph throughout the whole update sequence. Since $\widehat{m}$ can be as high as $O(n^2)$, their update time is $O(n^{2.16} \log^2 n)$. Khanna, Motwani, and Wilson [52] proved that, when a lookahead of $\Theta(n^{0.18})$ in the updates is permitted, a deterministic update bound of $O(n^{2.18})$ can be achieved.

The situation for dynamic shortest paths has been even more dramatic. Indeed, the first papers on dynamic shortest paths date back to 1967 [58,61,64]. In 1985, Even and Gazit [20] and Rohnert [67] presented algorithms for maintaining shortest paths on directed graphs with arbitrary real weights. Their algorithms required $O(n^2)$ per edge insertion; however, the worst-case bounds for edge deletions were comparable to recomputing APSP from scratch. Moreover, Ramalingam and Reps [63] considered dynamic shortest path algorithms with arbitrary real weights, but in a different model. Namely, the running time of their algorithm is analyzed in terms of the output change rather than the input size (output bounded complexity). Frigioni et al. [27,28] designed fast algorithms for graphs with bounded genus, bounded degree graphs, and bounded treewidth graphs in the same model. Again, in the worst case the running times of output-bounded dynamic algorithms are comparable to recomputing APSP from scratch.

Until recently, there seemed to be few dynamic shortest path algorithms which were provably faster than recomputing APSP from scratch, and they only worked on special cases and with small integer weights. In particular, Ausiello et al. [3] proposed an incremental shortest path algorithm for directed graphs having positive integer weights less than C: the amortized running time of their algorithm is $O(Cn \log n)$ per edge insertion. Henzinger et al. [43] designed a fully dynamic algorithm for APSP on planar graphs with integer weights, with a running time of $O(n^{9/7} \log(nC))$ per operation. Fakcharoemphol and Rao in [22] designed a fully dynamic algorithm for single-source shortest paths in planar directed graphs that supports both queries and edge weight updates in $O(n^{4/5} \log^{13/5} n)$ amortized time per operation.

Quite recently, many new algorithms for dynamic transitive closure and shortest path problems have been proposed.

Dynamic transitive closure. For dynamic transitive closure, King and Sagert [55] in 1999 showed how to support queries in $O(1)$ time and updates in $O(n^{2.26})$ time for general directed graphs and $O(n^2)$ time for directed acyclic graphs; their algorithm is randomized with one-side error. The bounds of King and Sagert were further improved by King [55], who exhibited a deterministic algorithm on general digraphs with $O(1)$ query time and $O(n^2 \log n)$ amortized time per update operations, where updates are insertions of a set of edges incident to the same vertex and deletions of an arbitrary subset of edges. All those algorithms are based on reductions to fast matrix multiplication and tree data structures for encoding information about dynamic paths.

Demetrescu and Italiano [11] proposed a deterministic algorithm for fully dynamic transitive closure on general digraphs that answers each query with one matrix look-up and supports updates in $O(n^2)$ amortized time. This bound can be made worst-case as shown by Sankowski in [68]. We observe that fully dynamic transitive closure algorithms with $O(1)$ query time maintain explicitly the transitive closure of the input graph, in order to answer each query with exactly one lookup (on its adjacency matrix). Since an update may change as many as $\Omega(n^2)$ entries of this matrix, $O(n^2)$ seems to be the best update bound that one could hope for this class of algorithms. This algorithm hinges upon the well-known equivalence between transitive closure and matrix multiplication on a closed semiring [23,30,60].

In [10] Demetrescu and Italiano show how to trade off query times for updates on directed acyclic graphs: each query can be answered in time $O(n^{\epsilon})$ and each update can be performed in time $O(n^{\omega(1,\epsilon,1)-\epsilon} + n^{1+\epsilon})$, for any $\epsilon \in [0, 1]$, where $\omega(1, \epsilon, 1)$ is the exponent of the multiplication of an $n \times n^{\epsilon}$ matrix by an $n^{\epsilon} \times n$ matrix. Balancing the two terms in the update bound yields that ϵ must satisfy the equation $\omega(1, \epsilon, 1) = 1 + 2\epsilon$. The current best bounds on $\omega(1, \epsilon, 1)$ [5,47] imply that $\epsilon < 0.575$. Thus, the smallest update time is $O(n^{1.575})$, which gives a query time of $O(n^{0.575})$. This subquadratic algorithm is randomized, and has one-side error. This result has been generalized to general graphs within the same bounds by Sankowski in [68], who has also shown how to achieve an even faster update time of $O(n^{1.495})$ at the expense of a much higher $O(n^{1.495})$ query time. Roditty and Zwick presented an algorithm [65] with $O(m\sqrt{n})$ update time and $O(\sqrt{n})$ query time and another algorithm [66] with $O(m + n \log n)$ update time and $O(n)$ query time.

Techniques for reducing the space usage of algorithms for dynamic path problems are presented in [56]. An extensive computational study on dynamic transitive closure problems appears in [29].

Dynamic shortest paths. For dynamic shortest paths, King [54] presented a fully dynamic algorithm for maintaining APSP in directed graphs with positive integer weights less than C: the running time of her algorithm is $O(n^{2.5}\sqrt{C \log n})$ per update. As in the case of dynamic transitive closure, this algorithm is based on clever tree data structures. Demetrescu and Italiano [12] proposed a fully dynamic algorithm for maintaining APSP on directed graphs with arbitrary real weights. Given a directed graph G, subject to dynamic operations, and such that each edge weight can assume at most S different *real* values, their algorithm supports each update in $O(S \cdot n^{2.5} \log^3 n)$ amortized time and each query in optimal worst-case time. We remark that the sets of possible weights of two different edges need not be necessarily the same: namely, any edge can be associated with a different set of possible weights. The only constraint is that throughout the sequence of operations, each edge can assume at most S different real values, which seems to be the case in many applications. Differently from [54], this method uses dynamic reevaluation of products of real-valued matrices as the kernel for solving dynamic shortest paths. Finally, the same authors [8] have studied some combinatorial properties of graphs that make it possible to devise a different approach to dynamic APSP problems. This approach yields a fully dynamic algorithm for general directed graphs with nonnegative real-valued edge weights that supports any sequence of operations in $O(n^2 \log^3 n)$ amortized time per update and unit worst-case time per distance query, where n is the number of vertices. Shortest paths can be reported in optimal worst-case time. The algorithm is deterministic, uses simple data structures, and appears to be very fast in practice. Using the same approach, Thorup [69] has shown how to achieve $O(n^2(\log n + \log^2((m + n)/n)))$ amortized time per update and $O(mn)$ space. His

algorithm works with negative weights as well. In [70], Thorup has shown how to achieve worst-case bounds at the price of a higher complexity: in particular, the update bounds become $\widetilde{O}(n^{2.75})$, where $\widetilde{O}(f(n))$ denotes $O(f(n) \cdot \text{polylog } n)$.

An extensive computational study on dynamic APSP problems appears in [9].

9.3.1 Combinatorial Properties of Path Problems

In this section we provide some background for the two algorithmic graph problems considered here: transitive closure and APSP. In particular, we discuss two methods: the first is based on a simple doubling technique which consists of repeatedly concatenating paths to form longer paths via matrix multiplication, and the second is based on a Divide and Conquer strategy. We conclude this section by discussing a useful combinatorial property of long paths.

9.3.1.1 Logarithmic Decomposition

We first describe a simple method for computing X^* in $O(n^{\mu} \cdot \log n)$ worst-case time, where $O(n^{\mu})$ is the time required for computing the product of two matrices over a closed semiring. The algorithm is based on a simple path doubling argument. We focus on the $\{+, \cdot, 0, 1\}$ semiring; the case of the $\{\min, +\}$ semiring is completely analogous.

Let $\mathcal{B}_n$ be the set of $n \times n$ Boolean matrices and let $X \in \mathcal{B}_n$. We define a sequence of $\log n + 1$ polynomials $P_0, \ldots, P_{\log n}$ over Boolean matrices as:

$$P_i = \begin{cases} X & \text{if } i = 0 \\ P_{i-1} + P_{i-1}^2 & \text{if } i > 0 \end{cases}$$

It is not difficult to see that for any $1 \leq u, v \leq n$, $P_i[u, v] = 1$ if and only if there is a path $u \rightsquigarrow v$ of length at most 2^i in X. We combine paths of length ≤ 2 in X to form paths of length ≤ 4, then we concatenate all paths found so far to obtain paths of length ≤ 8 and so on. As the length of the longest detected path increases exponentially and the longest simple path is no longer than n, a logarithmic number of steps suffices to detect if any two nodes are connected by a path in the graph as stated in the following theorem.

THEOREM 9.7 *Let X be an $n \times n$ matrix. Then $X^* = I_n + P_{\log n}$.*

9.3.1.2 Long Paths

In this section we discuss an intuitive combinatorial property of long paths. Namely, if we pick a subset S of vertices at random from a graph G, then a sufficiently long path will intersect S with high probability. This can be very useful in finding a long path by using short searches.

To the best of our knowledge, this property was first given in [35], and later on it has been used many times in designing efficient algorithms for transitive closure and shortest paths (see e.g., [12,54,71,73]). The following theorem is from [71].

THEOREM 9.8 *Let $S \subseteq V$ be a set of vertices chosen uniformly at random. Then the probability that a given simple path has a sequence of more than $\frac{cn \log n}{|S|}$ vertices, none of which are from S, for any $c > 0$, is, for sufficiently large n, bounded by $2^{-\alpha c}$ for some positive α.*

As shown in [73], it is possible to choose set S deterministically by a reduction to a hitting set problem [4,59]. A similar technique has also been used in [54].

9.3.1.3 Locality

Recently, Demetrescu and Italiano [8] proposed a new approach to dynamic path problems based on maintaining classes of paths characterized by local properties, i.e., properties that hold for all proper subpaths, even if they may not hold for the entire paths. They showed that this approach can play a crucial role in the dynamic maintenance of shortest paths. For instance, they considered a class of paths defined as follows:

DEFINITION 9.4 A path π in a graph is *locally shortest* if and only if every proper subpath of π is a shortest path.

This definition is inspired by the optimal-substructure property of shortest paths: all subpaths of a shortest path are shortest. However, a locally shortest path may not be shortest.

The fact that locally shortest paths include shortest paths as a special case makes them an useful tool for computing and maintaining distances in a graph. Indeed, paths defined locally have interesting combinatorial properties in dynamically changing graphs. For example, it is not difficult to prove that the number of locally shortest paths that may change due to an edge weight update is $O(n^2)$ if updates are partially dynamic, i.e., increase-only or decrease-only:

THEOREM 9.9 *Let G be a graph subject to a sequence of increase-only or decrease-only edge weight updates. Then the amortized number of paths that start or stop being locally shortest at each update is $O(n^2)$.*

Unfortunately, Theorem 9.9 may not hold if updates are fully dynamic, i.e., increases and decreases of edge weights are intermixed. To cope with pathological sequences, a possible solution is to retain information about the history of a dynamic graph, considering the following class of paths:

DEFINITION 9.5 A *historical shortest path* (in short, *historical path*) is a path that has been shortest at least once since it was last updated.

Here, we assume that a path is updated when the weight of one of its edges is changed. Applying the locality technique to historical paths, we derive locally historical paths:

DEFINITION 9.6 A path π in a graph is *locally historical* if and only if every proper subpath of π is historical.

Like locally shortest paths, also locally historical paths include shortest paths, and this makes them another useful tool for maintaining distances in a graph:

LEMMA 9.9 If we denote by *SP*, *LSP*, and *LHP* respectively the sets of shortest paths, locally shortest paths, and locally historical paths in a graph, then at any time the following inclusions hold: $SP \subseteq LSP \subseteq LHP$.

Differently from locally shortest paths, locally historical paths exhibit interesting combinatorial properties in graphs subject to fully dynamic updates. In particular, it is possible to prove that the number of paths that become locally historical in a graph at each edge weight update depends on the number of historical paths in the graph.

THEOREM 9.10 [8] *Let G be a graph subject to a sequence of update operations. If at any time throughout the sequence of updates there are at most $O(h)$ historical paths in the graph, then the amortized number of paths that become locally historical at each update is $O(h)$.*

To keep changes in locally historical paths small, it is then desirable to have as few historical paths as possible. Indeed, it is possible to transform every update sequence into a slightly longer equivalent sequence that generates only a few historical paths. In particular, there exists a simple smoothing strategy that, given any update sequence Σ of length k, produces an operationally equivalent sequence $F(\Sigma)$ of length $O(k \log k)$ that yields only $O(\log k)$ historical shortest paths between each pair of vertices in the graph. We refer the interested reader to [8] for a detailed description of this smoothing strategy. According to Theorem 9.10, this technique implies that only $O(n^2 \log k)$ locally historical paths change at each edge weight update in the smoothed sequence $F(\Sigma)$.

As elaborated in [8], locally historical paths can be maintained very efficiently. Since by Lemma 9.9 locally historical paths include shortest paths, this yields the fastest known algorithm for fully dynamic APSP.

9.3.2 Algorithmic Techniques

In this section we describe some algorithmic techniques which are the kernel of the best-known algorithms for maintaining transitive closure and shortest paths.

We start with some observations which are common to all the techniques considered here. First of all, we note that the algebraic structure of path problems allows one to support insertions in a natural fashion. Indeed, insertions correspond to the $\oplus$ operation on closed semirings. This perhaps can explain the wealth of fast incremental algorithms for dynamic path problems on directed graphs [48,50,57,72]. However, there seems to be no natural setting for deletions in this algebraic framework. Thus, in designing fully dynamic algorithms it seems quite natural to focus on special techniques and data structures for supporting deletions.

The second remark is that most fully dynamic algorithms are surprisingly based on the same decompositions used for the best static algorithms (see Section 9.3.1). The main difference is that they maintain dynamic data structures at each level of the decomposition. As we will see, the definition of a suitable interface between data structures at different levels plays an important role in the design of efficient dynamic algorithms.

In the remainder of this section we describe data structures that are able to support dynamic operations at each level. In Sections 9.3.3 and 9.3.4, the data structures surveyed here will be combined with the decompositions shown in Section 9.3.1 to obtain efficient algorithms for dynamic path problems on directed graphs.

9.3.2.1 Tools for Trees

In this section we describe a tree data structure for keeping information about dynamic path problems. The first appearance of this tool dates back to 1981, when Even and Shiloach [21] showed how to maintain a breadth-first tree of an undirected graph under any sequence of edge deletions; they used this as a kernel for decremental connectivity on undirected graphs. Later on, Henzinger and King [39] showed how to adapt this data structure to fully dynamic transitive closure in directed graphs. Recently, King [54] designed an extension of this tree data structure to weighted directed graphs for solving fully dynamic APSP.

The Problem. The goal is to maintain information about breadth-first search (BFS) on an undirected graph G undergoing deletions of edges. In particular, in the context of dynamic path problems, we are interested in maintaining BFS trees of depth up to d, with $d \leq n$. For the sake of simplicity, we describe only the case where deletions do not disconnect the underlying graph. The general case can

be easily handled by means of "phony" edges (i.e., when deleting an edge that disconnects the graph, we just replace it by a phony edge).

It is well known that BFS partitions vertices into levels, so that there can be edges only between adjacent levels. More formally, let r be the vertex where we start BFS, and let level ℓ_i contains vertices encountered at distance i from r ($\ell_0 = \{r\}$): edges incident to a vertex at level ℓ_i can have their other endpoints either at level ℓ_{i-1}, ℓ_i, or ℓ_{i+1}, and no edge can connect vertices at levels ℓ_i and ℓ_j for $|j - i| > 1$. Let $\ell(v)$ be the level of vertex v.

Given an undirected graph $G = (V, E)$ and a vertex $r \in V$, we would like to support any intermixed sequence of the following operations:

- `Delete`(x, y): delete edge (x, y) from G.
- `Level`(u): return the level $\ell(u)$ of vertex u in the BFS tree rooted at r (return $+\infty$ if u is not reachable from r).

In the remainder, to indicate that an operation `Y()` is performed on a data structure `X`, we use the notation `X.Y()`.

Data structure. We maintain information about BFS throughout the sequence of edge deletions by simply keeping explicitly those levels. In particular, for each vertex v at level ℓ_i in T, we maintain the following data structures: UP(v), SAME(v), and DOWN(v) containing the edges connecting v to level ℓ_{i-1}, ℓ_i, and ℓ_{i+1}, respectively. Note that for all $v \neq r$, UP(v) must contain at least one edge (i.e., the edge from v to its parent in the BFS tree). In other words, a nonempty UP(v) witnesses the fact that v is actually entitled to belong to that level. This property will be important during edge deletions: whenever UP(v) gets emptied because of deletions, v looses its right to be at that level and must be demoted at least one level down.

Implementation of operations. When edge (x, y) is being deleted, we proceed as follows. If $\ell(x) = \ell(y)$, simply delete (x, y) from SAME(y) and from SAME(x). The levels encoded in UP, SAME, and DOWN still capture the BFS structure of G. Otherwise, without loss of generality let $\ell(x) = \ell_{i-1}$ and $\ell(y) = \ell_i$. Update the sets UP, SAME, and DOWN by deleting x from UP(y) and y from DOWN(x). If UP(y) $\neq \emptyset$, then there is still at least one edge connecting y to level ℓ_{i-1}, and the levels will still reflect the BFS structure of G after the deletion.

The main difficulty is when UP(y) $= \emptyset$ after the deletion of (x, y). In this case, deleting (x, y) causes y to loose its connection to level ℓ_{i-1}. Thus, y has to drop down at least one level. Furthermore, its drop may cause a deeper landslide in the levels below. This case can be handled as follows.

We use a FIFO queue Q, initialized with vertex y. We will insert a vertex v in the queue Q whenever we discover that UP(v) $= \emptyset$, i.e., vertex v has to be demoted at least one level down. We will repeat the following demotion step until Q is empty:

Demotion step:

1. Remove the first vertex in Q, say v.
2. Delete v from its level $\ell(v) = \ell_i$ and tentatively try to place v one level down, i.e., in ℓ_{i+1}.
3. Update the sets UP, SAME, and DOWN consequently:
   a. For each edge (u, v) in SAME(v), delete (u, v) from SAME(u) and insert (u, v) in DOWN(u) and UP(v) (as UP(v) was empty, this implies that UP(v) will be initialized with the old set SAME(v)).
   b. For each edge (v, z) in DOWN(v), move edge (v, z) from UP(z) to SAME(z) and from DOWN(v) to SAME(v); if the new UP(z) is empty, insert z in the queue Q. Note that this will empty DOWN(v).
   c. If UP(v) is still empty, insert v again into Q.

Analysis. It is not difficult to see that applying the Demotion Step until the queue is empty will maintain correctly the BFS levels. Level queries can be answered in constant time. To bound the total time required to process any sequence of edge deletions, it suffices to observe that each time an edge (u, v) is examined during a demotion step, either u or v will be dropped one level down. Thus, edge (u, v) can be examined at most $2d$ times in any BFS levels up to depth d throughout any sequence of edge deletions. This implies the following theorem.

THEOREM 9.11 *Maintaining BFS levels up to depth d requires $O(md)$ time in the worst case throughout any sequence of edge deletions in an undirected graph with m initial edges.*

This means that maintaining BFS levels requires d times the time needed for constructing them. Since $d \leq n$, we obtain a total bound of $O(mn)$ if there are no limits on the depth of the BFS levels.

As it was shown in [39,54], it is possible to extend the BFS data structure presented in this section to deal with weighted directed graphs. In this case, a shortest path tree is maintained in place of BFS levels: after each edge deletion or edge weight increase, the tree is reconnected by essentially mimicking Dijkstra's algorithm rather than BFS. Details can be found in [54].

9.3.3 Dynamic Transitive Closure

In this section we consider algorithms for fully dynamic transitive closure. In particular, we describe the algorithm of King [54], whose main ingredients are the logarithmic decomposition of Section 9.3.1.1 and the tools for trees described in Section 9.3.2.1. This method yields $O(n^2 \log n)$ amortized time per update and $O(1)$ per query. Faster methods have been designed by Demetrescu and Italiano [11] and by Sankowski [68].

We start with a formal definition of the fully dynamic transitive closure problem.

The Problem. Let $G = (V, E)$ be a directed graph and let $TC(G) = (V, E')$ be its transitive closure. We consider the problem of maintaining a data structure for graph G under an intermixed sequence of update and query operations of the following kinds:

- `Insert`(v, I): perform the update $E \leftarrow E \cup I$, where $I \subseteq E$ and $v \in V$. This operation assumes that all edges in I are incident to v. We call this kind of update a v-Centered insertion in G.
- `Delete`(D): perform the update $E \leftarrow E - D$, where $D \subseteq E$.
- `Query`(x, y): perform a query operation on $TC(G)$ and return 1 if $(x, y) \in E'$ and 0 otherwise.

We note that these generalized `Insert` and `Delete` updates are able to change, with just one operation, the graph by adding or removing a whole set of edges, rather than a single edge. Differently from other variants of the problem, we do not address the issue of returning actual paths between nodes, and we just consider the problem of answering reachability queries.

We now describe how to maintain the Transitive Closure of a directed graph in $O(n^2 \log n)$ amortized time per update operation. The algorithm that we describe has been designed by King [54] and is based on the tree data structure presented in Section 9.3.2.1 and on the logarithmic decomposition described in Section 9.3.1.1. To support queries efficiently, the algorithm uses also a counting technique that consists of keeping a count for each pair of vertices x, y of the number of insertion operations that yielded new paths between them. Counters are maintained so that there is a path from x to y if and only if the counter for that pair is nonzero. The counting technique has been first introduced in [55]: in that case, counters keep track of the number of different distinct paths between

pairs of vertices in an acyclic directed graph. We show the data structure used for maintaining the Transitive Closure and how operations `Insert`, `Delete`, and `Query` are implemented.

Data structure. Given a directed graph $G = (V, E)$, we maintain $\log n + 1$ levels. On each level i, $0 \leq i \leq \log n$ we maintain the following data structures:

- A graph $G_i = (V, E_i)$ such that $(x, y) \in E_i$ if there is a path from x to y in G of length $\leq 2^i$. Note that the converse is not necessarily true: i.e., $(x, y) \in E_i$ may not necessarily imply, however, that there is a path from x to y in G of length $\leq 2^i$. We maintain G_0 and $G_{\log n}$ such that $G_0 = G$ and $G_{\log n} = TC(G)$.
- For each $v \in V$, a BFS tree $Out_{i,v}$ of depth 2 of G_i rooted at v and a BFS tree $In_{i,v}$ of depth 2 of $\widehat{G_i}$ rooted at v, where $\widehat{G_i}$ is equal to G_i, except for the orientation of edges, which is reversed. We maintain the BFS trees with instances of the data structure presented in Section 9.3.2.1.
- A matrix $Count_i[x, y] = |\{ v : x \in In_{i,v} \wedge y \in Out_{i,v} \}|$. Note that $Count_i[x, y] > 0$, if there is a path from x to y in G of length $\leq 2^{i+1}$.

Implementation of operations. Operations can be realized as follows:

- `Insert`(v, I): for each $i = 0$ to $\log n$, do the following: add I to E_i, rebuild $Out_{i,v}$ and $In_{i,v}$, updating $Count_i$ accordingly, and add to I any (x, y) such that $Count_i[x, y]$ flips from 0 to 1.
- `Delete`(D): for each $i = 0$ to $\log n$, do the following: remove D from E_i, for each $(x, y) \in D$ do $Out_{i,v}$`.Delete`(x, y) and $In_{i,v}$`.Delete`(x, y), updating $Count_i$ accordingly, and add to D all (x, y) such that $Count_i[x, y]$ flips from positive to zero.
- `Query`(x, y): return 1 if $(x, y) \in E_{\log n}$ and 0 otherwise.

We note that an `Insert` operation simply rebuilds the BFS trees rooted at v on each level of the decomposition. It is important to observe that the trees rooted at other vertices on any level i might not be valid BFS trees of the current graph G_i, but are valid BFS trees of some older version of G_i that did not contain the newly inserted edges. A `Delete` operation, instead, maintains dynamically the tree data structures on each level, removing the deleted edges as described in Section 9.3.2.1 and propagating changes up to the decomposition.

Analysis. To prove the correctness of the algorithm, we need to prove that, if there is a path from x to y in G of length $\leq 2^i$, then $(x, y) \in E_i$. Conversely, it is easy to see that $(x, y) \in E_i$ only if there is a path from x to y in G of length $\leq 2^i$. We first consider `Insert` operations. It is important to observe that, by the problem's definition, the set I contains only edges incident to v for $i = 0$, but this might not be the case for $i > 0$, since entries $Count_i[x, y]$ with $x \neq v$ and $y \neq v$ might flip from 0 to 1 during a v-centered insertion. However, we follow a lazy approach and we only rebuild the BFS trees on each level rooted at v. The correctness of this follows from the simple observation that any new paths that appear due to a v-centered insertion pass through v, so rebuilding the trees rooted at v is enough to keep track of these new paths. To prove this, we proceed by induction. We assume that for any new paths $x \rightsquigarrow v$ and $v \rightsquigarrow y$ of length $\leq 2^i$, $(x, v), (v, y) \in E_i$ at the beginning of loop iteration i. Since $x \in In_{i,v}$ and $y \in Out_{i,v}$ after rebuilding $In_{i,v}$ and $Out_{i,v}$, $Count_i[x, y]$ is increased by one if no path $x \rightsquigarrow v \rightsquigarrow y$ of length $\leq 2^{i+1}$ existed before the insertion. Thus, $(x, y) \in E_{i+1}$ at the beginning of loop iteration $i + 1$. To complete our discussion of correctness, we note that deletions act as "undo" operations that leave the data structures as if deleted edges were never inserted. The running time is established as follows.

THEOREM 9.12 *[54] The fully dynamic transitive closure problem can be solved with the following bounds. Any* `Insert` *operation requires* $O(n^2 \log n)$ *worst-case time, the cost of*

`Delete` operations can be charged to previous insertions, and `Query` operations are answered in constant time.

PROOF The bound for `Insert` operations derives from the observation that reconstructing BFS trees on each of the $\log n + 1$ levels requires $O(n^2)$ time in the worst case. By Theorem 9.11, any sequence of `Delete` operations can be supported in $O(d)$ times the cost of building a BFS tree of depth up to d. Since $d = 2$ in the data structure, this implies that the cost of deletions can be charged to previous insertion operations. The bound for `Query` operations is straightforward. □

The previous theorem implies that updates are supported in $O(n^2 \log n)$ amortized time per operation.

Demetrescu and Italiano [7,11] have shown how to reduce the running time of updates to $O(n^2)$ amortized, by casting this dynamic graph problem into a dynamic matrix problem. Again based on dynamic matrix computations, Sankowski [68] showed how to make the $O(n^2)$ amortized bound worst-case.

9.3.4 Dynamic Shortest Paths

In this section we survey the best-known algorithms for fully dynamic shortest paths. Those algorithms can be seen as a natural evolution of the techniques described so far for dynamic transitive closure. They are not a trivial extension of transitive closure algorithms, however, as dynamic shortest path problems look more complicated in nature. As an example, consider the deletion of an edge (u, v). In the case of transitive closure, reachability between a pair of vertices x and y can be reestablished by *any* replacement path avoiding edge (u, v). In case of shortest paths, after deleting (u, v), we have to look for the *best* replacement path avoiding edge (u, v).

We start with a formal definition of the fully dynamic APSP problem.

The problem. Let $G = (V, E)$ be a weighted directed graph. We consider the problem of maintaining a data structure for G under an intermixed sequence of update and query operations of the following kinds:

- `Decrease`(v, w): decrease the weight of edges incident to v in G as specified by a new weight function w. We call this kind of update a v-Centered decrease in G.
- `Increase`(w): increase the weight of edges in G as specified by a new weight function w.
- `Query`(x, y): return the distance between x and y in G.

As in fully dynamic transitive closure, we consider generalized update operations where we modify a whole set of edges, rather than a single edge. Again, we do not address the issue of returning actual paths between vertices, and we just consider the problem of answering distance queries.

Demetrescu and Italiano [8] devised the first deterministic near-quadratic update algorithm for fully dynamic APSP. This algorithm is also the first solution to the problem in its generality. The algorithm is based on the notions of historical paths and locally historical paths in a graph subject to a sequence of updates, as discussed in Section 9.3.1.3.

The main idea is to maintain dynamically the locally historical paths of the graph in a data structure. Since by Lemma 9.9 shortest paths are locally historical, this guarantees that information about shortest paths is maintained as well.

To support an edge weight update operation, the algorithm implements the smoothing strategy mentioned in Section 9.3.1.3 and works in two phases. It first removes from the data structure all maintained paths that contain the updated edge: this is correct since historical shortest paths, in view of their definition, are immediately invalidated as soon as they are touched by an update. This

means that also locally historical paths that contain them are invalidated and have to be removed from the data structure. As a second phase, the algorithm runs an all-pairs modification of Dijkstra's algorithm [13], where at each step a shortest path with minimum weight is extracted from a priority queue and it is combined with existing historical shortest paths to form new locally historical paths. At the end of this phase, paths that become locally hustorical after the update are correctly inserted in the data structure.

The update algorithm spends constant time for each of the $O(zn^2)$ new locally historical path (see Theorem 9.10). Since the smoothing strategy lets $z = O(\log n)$ and increases the length of the sequence of updates by an additional $O(\log n)$ factor, this yields $O(n^2 \log^3 n)$ amortized time per update. The interested reader can find further details about the algorithm in [8].

THEOREM 9.13 *[8] The fully dynamic APSP problem can solved in $O(n^2 \log^3 n)$ amortized time per update.*

Using the same approach, but with a different smoothing strategy, Thorup [69] has shown how to achieve $O(n^2(\log n + \log^2(m/n)))$ amortized time per update and $O(mn)$ space. His algorithm works with negative weights as well.

9.4 Research Issues and Summary

In this chapter we have described the most efficient known algorithms for maintaining dynamic graphs. Despite the bulk of work on this area, several questions remain still open. For dynamic problems on directed graphs, can we reduce the space usage for dynamic shortest paths to $O(n^2)$? Second, and perhaps more importantly, can we solve efficiently fully dynamic *single-source* reachability and shortest paths on general graphs? Finally, are there any general techniques for making increase-only algorithms fully dynamic? Similar techniques have been widely exploited in the case of fully dynamic algorithms on undirected graphs [40–42].

9.5 Further Information

Research on dynamic graph algorithms is published in many computer science journals, including *Algorithmica*, *Journal of ACM*, *Journal of Algorithms*, *Journal of Computer and System Science*, *SIAM Journal on Computing*, and *Theoretical Computer Science*. 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* (STOC), the *IEEE Symposium on Foundations of Computer Science* (FOCS), and the *International Colloquium on Automata, Languages and Programming* (ICALP). More specialized conferences devoted exclusively to algorithms are the *ACM–SIAM Symposium on Discrete Algorithms* (SODA) and the *European Symposium on Algorithms* (ESA).

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.

Acknowledgments

The work of the first author has been partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT ("Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments").

The work of the fourth author has been partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet") and by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT ("Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments").

References

1. D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. *ACM Journal on Experimental Algorithmics*, 2, 1997.
2. G. Amato, G. Cattaneo, and G. F. Italiano, Experimental analysis of dynamic minimum spanning tree algorithms. In *Proceedings of the 8th ACM-SIAM Annual Symposium on Discrete Algorithms (SODA 97)*, New Orleans, LA, pp. 314–323, January 5–7, 1997.
3. G. Ausiello, G. F. Italiano, A. Marchetti Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12:615–638, 1991.
4. V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.
5. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd edition. The MIT Press, Cambridge, MA, 2001.
7. C. Demetrescu. Fully dynamic algorithms for path problems on directed graphs. PhD thesis, Department of Computer and Systems Science, University of Rome "La Sapienza," Rowa, Italy, February 2001.
8. C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the Association for Computing Machinery (JACM)*, 51(6):968–992, 2004.
9. C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. in *ACM Transactions on Algorithms*, 2(4):578–601, 2006.
10. C. Demetrescu and G. F. Italiano. Trade-offs for fully dynamic reachability on dags: Breaking through the $O(n^2)$ barrier. *Journal of the Association for Computing Machinery (JACM)*, 52(2):147–156, 2005.
11. C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proceedings of the 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS'00)*, Re dondo Beach, CA, pp. 381–389, 2000.
12. C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72(5):813–837, August 2006.
13. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
14. D. Eppstein. Dynamic generators of topologically embedded graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD, pp. 599–608, 2003.

15. D. Eppstein. All maximal independent sets and dynamic dominance for sparse graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, Vancouver, BC, pp. 451–459, 2005.
16. D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification—A technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44:669–696, 1997.
17. D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Separator based sparsification I: Planarity testing and minimum spanning trees. *Journal of Computer and System Science*, 52(1):3–27, 1996.
18. D. Eppstein, Z. Galil, G. F. Italiano, T. H. Spencer. Separator based sparsification II: Edge and vertex connectivity. *SIAM Journal on Computing*, 28:341–381, 1999.
19. 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. *Journal of Algorithms*, 13:33–54, 1992.
20. S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.
21. S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28:1–4, 1981.
22. J. Fakcharoemphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proceedings of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01)*, Las Vegas, NV, pp. 232–241, 2001.
23. M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *Conference Record 1971 12th Annual Symposium on Switching and Automata Theory*, East Lansing, MI, pp. 129–131, October 13–15 1971.
24. G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM Journal on Computing*, 14:781–798, 1985.
25. G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing*, 26:484–538, 1997.
26. M. L. Fredman and M. R. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.
27. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single source shortest paths trees. *Algorithmica*, 22(3):250–274, 1998.
28. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:351–381, 2000.
29. D. Frigioni, T. Miller, U. Nanni, and C. D. Zaroliagis. An experimental study of dynamic algorithms for transitive closure. *ACM Journal of Experimental Algorithms*, 6(9), 2001.
30. M. E. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Mathematics–Doklady*, 11(5):1252, 1970. English translation.
31. Z. Galil and G. F. Italiano. Fully dynamic algorithms for 2-edge-connectivity. *SIAM Journal on Computing*, 21:1047–1069, 1992.
32. Z. Galil and G. F. Italiano. Maintaining the 3-edge-connected components of a graph on-line. *SIAM Journal on Computing*, 22:11–28, 1993.
33. Z. Galil, G. F. Italiano, and N. Sarnak. Fully dynamic planarity testing. *Journal of the ACM*, 48:28–91, 1999.
34. D. Giammarresi and G. F. Italiano. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica*, 16(3):263–287, 1996.
35. D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhäuser, Boston, Cambridge, MA, 1982.
36. F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.
37. M. R. Henzinger. Fully dynamic biconnectivity in graphs. *Algorithmica*, 13(6):503–538, 1995.
38. M. R. Henzinger. Improved data structures for fully dynamic biconnectivity. *SIAM Journal on Computing*, 29(6):1761–1815, 2000.
39. M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–536, 1999.

40. M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th IEEE Symposium Foundations of Computer Science*, Milulavkee, WI, pp. 664–672, 1995.
41. M. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM Journal on Computing*, 31(2):364–374, 2001.
42. M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.
43. M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, August 1997.
44. M. R. Henzinger and J. A. La Poutré. Certificates and fast algorithms for biconnectivity in fully dynamic graphs. In *Proceedings of the 3rd European Symposium on Algorithms*. *Lecture Notes in Computer Science* 979, Springer-Verlag, Berlin, pp. 171–184, 1995.
45. M. R. Henzinger and M. Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures and Algorithms*, 11(4):369–379, 1997.
46. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.
47. X. Huang and V. Y. Pan. Fast rectangular matrix multiplication and applications. *Journal of Complexity*, 14(2):257–299, June 1998.
48. T. Ibaraki and N. Katoh. On-line computation of transitive closure for graphs. *Information Processing Letters*, 16:95–97, 1983.
49. R. D. Iyer Jr., D. R. Karger, H. S. Rahul, and M. Thorup. An experimental study of poly-logarithmic fully-dynamic connectivity algorithms. *ACM Journal of Experimental Algorithmics*, 6, 2001.
50. G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.
51. G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28:5–11, 1988.
52. S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead in dynamic graph problems. *Algorithmica*, 21(4):377–394, 1998.
53. P. N. Klein and S. Sairam. Fully dynamic approximation schemes for shortest path problems in planar graphs. In *Proceedings of the 3rd Workshop Algorithms and Data Structures*. *Lecture Notes in Computer Science* 709, Springer-Verlag, Berlin, pp. 442–451, 1993.
54. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, New York, pp. 81–99, 1999.
55. V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150–167, 2002.
56. V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON)*. *Lecture Notes in Computer Science 2108*, Springer-Verlag, Berlin, pp. 268–277, 2001.
57. J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Proceedings of the Workshop on Graph-Theoretic Concepts in Computer Science*. *Lecture Notes in Computer Science* 314, Springer-Verlag, Berlin, pp. 106–120, 1988.
58. P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pp. 96–109, 1967.
59. L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.
60. I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.
61. J. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical Report, LBS-TNT-26, London Business School, Transport Network Theory Unit, London, U.K., 1967.

62. H. Nagamochi and T. Ibaraki. Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmmica*, 7:583–596, 1992.
63. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.
64. V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(5):336–343, 1968.
65. L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of the 43th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, Vancouver, BC, pp. 679–688, 2002.
66. L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, Chicago, IL, pp. 184–191, 2004.
67. H. Rohnert. A dynamization of the all-pairs least cost problem. In *Proceedings of the 2nd Annual Symposium on Theoretical Aspects of Computer Science, (STACS'85). Lecture Notes in Computer Science* 182, Springer-Verlag, Berlin, pp. 279–286, 1985.
68. P. Sankowski. Dynamic transitive closure via dynamic matrix inverse. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, Rome, Italy, pp. 509–517, 2004.
69. M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, Humleback, Denmark, pp. 384–396, 2004.
70. M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the 37th ACM Symposium on Theory of Computing (STOC 2005)*, Baltimore, MD, pp. 112–119, 2005.
71. J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.
72. D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.
73. U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.