

1

Dynamic graphs

	1.1	Introduction.....	1-1
	1.2	Tools for undirected graphs..... Clustering • Sparsification • Randomization	1-2
	1.3	Tools for directed graphs..... Kleene closures • Locality • Long paths • Matrices	1-5
	1.4	Connectivity..... Updates in $O(\log^2 n)$ time	1-9
	1.5	Minimum spanning tree..... Deletions in $O(\log^2 n)$ time • Updates in $O(\log^4 n)$ time	1-11
Camil Demetrescu <i>DIS, Università di Roma "La Sapienza"</i>	1.6	Transitive closure..... Updates in $O(n^2 \log n)$ time • Updates in $O(n^2)$ time	1-13
Irene Finocchi <i>DISP, Università di Roma "Tor Vergata"</i>	1.7	All-pairs shortest paths..... Updates in $O(n^{2.5} \sqrt{C \log n})$ time • Updates in $O(n^2 \log^3 n)$ time	1-15
Giuseppe F. Italiano <i>DISP, Università di Roma "Tor Vergata"</i>	1.8	Conclusions.....	1-17

1.1 Introduction

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

An *update on a graph* is an operation that inserts or deletes edges or vertices of the graph or changes attributes associated with edges or vertices, such as cost or color. Throughout this chapter by *dynamic graph* we denote a graph that is subject to a sequence of updates. In a typical dynamic graph problem one would like to answer queries on dynamic graphs, such as, for instance, whether the graph is connected or which is the shortest path between any two 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. In particular, a dynamic graph problem is said to be *fully dynamic* if the update operations include unrestricted insertions and deletions of edges or vertices. A dynamic graph problem

is said to be *partially dynamic* if only one type of update, either insertions or deletions, is allowed. More specifically, a dynamic graph problem is said to be *incremental* if only insertions are allowed, while it is said to be *decremental* if only deletions are allowed.

In the first part of this chapter we will present the main algorithmic techniques used to solve dynamic problems on *undirected* graphs. To illustrate those techniques, we will focus particularly on dynamic minimum spanning trees and on connectivity problems. In the second part of the chapter we will deal with dynamic problems on *directed* graphs, and we will investigate as paradigmatic problems the dynamic maintenance of transitive closure and shortest paths. Interestingly enough, dynamic problems on directed graphs seem much harder to solve than their counterparts on undirected graphs, and require completely different techniques and tools.

1.2 Tools for undirected graphs

Many of the algorithms proposed in the literature use the same general techniques, and hence we begin by describing these techniques. In this section we focus on tools for undirected graphs, while tools for directed graphs will be discussed in Section 1.3. Typically, most of these techniques use some sort of graph decomposition, and partition either the vertices or the edges of the graph to be maintained. Moreover, data structures that maintain properties of dynamically changing trees, such as the ones described in Chapter 1 (linking and cutting trees, topology trees, and Euler tour trees), are often used as building blocks by many dynamic graph algorithms.

1.2.1 Clustering

The clustering technique has been introduced by Frederickson [16] and is based upon partitioning the graph into a suitable collection of connected subgraphs, called *clusters*, such that each update involves only a small number of such clusters. Typically, the decomposition defined by the clusters is applied recursively and the information about the subgraphs is combined with the topology trees described above. A refinement of the clustering technique appears in the idea of *ambivalent data structures* [17], in which edges can belong to multiple groups, only one of which is actually selected depending on the topology of the given spanning tree.

As an example, we briefly describe the application of clustering to the problem of maintaining a minimum spanning forest [16]. Let $G = (V, E)$ be a graph with a designated spanning tree S . Clustering is used for partitioning the vertex set V into subtrees connected in S , so that each subtree is only adjacent to a few other subtrees. A topology tree, as described in Chapter 1, is then used for representing a recursive partition of the tree S . Finally, a generalization of topology trees, called *2-dimensional topology trees*, are formed from pairs of nodes in the topology tree and allow it to maintain information about the edges in $E \setminus S$ [16].

Fully dynamic algorithms based only on a single level of clustering obtain typically time bounds of the order of $O(m^{2/3})$ (see for instance [20, 36]). When the partition can be applied recursively, better $O(m^{1/2})$ time bounds can be achieved by using 2-dimensional topology trees (see, for instance, [16, 17]).

THEOREM 1.1 (Frederickson [16]) *The minimum spanning forest of an undirected graph can be maintained in time $O(\sqrt{m})$ per update, where m is the current number of edges in the graph.*

We refer the interested reader to references [16, 17] for details about Frederickson's algorithm. With the same technique, an $O(\sqrt{m})$ time bound can be obtained also for fully dynamic connectivity and 2-edge connectivity [16, 17]

The type of clustering used can vary problem-dependent, however, and makes this technique difficult to be used as a black box.

1.2.2 Sparsification

Sparsification is a general technique due to Eppstein *et al.* [12] that can be used as a black box (without having to know the internal details) in order to design and dynamize graph algorithms. It is a divide-and-conquer technique that allows it 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. More precisely, 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 $T(n, m) = O(\sqrt{m})$, we get a better bound of $O(\sqrt{n})$. The technique itself is quite simple. A key concept is the notion of certificate.

DEFINITION 1.1 For any graph property P and graph G , a *certificate* for G is a graph G' such that G has property P if and only if G' has the property.

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 a sparse certificate. Certificates are then merged 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 $O(\log(m/n))$ ¹ graphs with $O(n)$ edges each, instead of one graph with m edges.

There exist two variants of sparsification. The first variant is used in situations where no previous fully dynamic algorithm is known. A static algorithm is used for recomputing 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, certificates are maintained using a dynamic data structure. For this to work, a *stability* property of certificates is needed, to ensure that a small change in the input graph does not lead to a large change in the certificates. We refer the interested reader to [12] for a precise definition of stability. This variant transforms time bounds of the form $O(m^p)$ into $O(n^p)$.

DEFINITION 1.2 A time bound $T(n)$ is *well-behaved* if, for some $c < 1$, $T(n/2) < cT(n)$. Well-behavedness eliminates strange situations in which a time bound fluctuates wildly with n . For instance, all polynomials are well-behaved.

THEOREM 1.2 (Eppstein *et al.* [12]) *Let P be a property for which we can find sparse*

¹Throughout this chapter, we assume that $\log x$ stands for $\max\{1, \log_2 x\}$, so $\log(m/n)$ is never smaller than 1 even if $m < 2n$.

certificates in time $f(n, m)$ for some well-behaved f , and such that we can construct a data structure for testing property 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 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 $q(n, O(n))$.

THEOREM 1.3 (Eppstein et al. [12]) *Let 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 P with update time $g(n, m)$ and query time $q(n, m)$. Then P can be maintained in time $O(f(n, O(n))) + g(n, O(n))$ per update, with query time $q(n, O(n))$.*

Basically, the first version of sparsification (Theorem 1.2) can be used to dynamize static algorithms, in which case we only need to *compute* efficiently *sparse* certificates, while the second version (Theorem 1.3) can be used to speed up existing fully dynamic algorithms, in which case we need to *maintain* efficiently *stable sparse* certificates.

Sparsification applies to a wide variety of dynamic graph problems, including minimum spanning forests, edge and vertex connectivity. As an example, for the fully dynamic minimum spanning tree problem, it reduces the update time from $O(\sqrt{m})$ [16, 17] to $O(\sqrt{n})$ [12].

Since sparsification works on top of a given algorithm, we need not to know the internal details of this algorithm. Consequently, it can be applied orthogonally to other data structuring techniques: in a large number of situations both clustering and sparsification have been combined to produce an efficient dynamic graph algorithm.

1.2.3 Randomization

Clustering and sparsification allow one to design efficient deterministic algorithms for fully dynamic problems. The last technique we present in this section is due to Henzinger and King [24], and allows one to achieve faster update times for some problems by exploiting the power of randomization.

We now sketch how the randomization technique works taking the fully dynamic connectivity problem as an example. Let $G = (V, E)$ be a graph to be maintained dynamically, and let F be a spanning forest of G . We call edges in F *tree edges*, and edges in $E \setminus F$ *non-tree edges*. The algorithm by Henzinger and King is based on the following ingredients.

Maintaining spanning forests

Trees are maintained using the Euler Tours data structure (ET trees) described in Chapter 1: this allows one to obtain logarithmic updates and queries within the forest.

Random sampling

Another key idea is the following: when e is deleted from a tree T , use random sampling among the non-tree edges incident to T , in order to find quickly a replacement edge for e , if any.

Graph decomposition

The last key idea is to combine randomization with a suitable graph decomposition. We maintain an edge decomposition of the current graph G using $O(\log n)$ edge disjoint subgraphs $G_i = (V, E_i)$. These subgraphs are hierarchically ordered. The lower levels con-

tain 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 also maintained.

Note that the hard operation in this problem is the deletion of a tree edge: indeed, a spanning forest is easily maintained with the help of the linking and cutting trees described in Chapter 1 throughout edge insertions, and deleting a non-tree edge does not change the forest.

The goal is an update time of $O(\log^3 n)$: after an edge deletion, in the quest for a replacement edge, we can afford a number of sampled edges of $O(\log^2 n)$. However, if the candidate set of edge e is a small fraction of all non-tree edges which are adjacent to T , it is unlikely to find a replacement edge for e among this small sample. If we found no candidate among the sampled edges, we must check explicitly all the non-tree edges adjacent to T . After random sampling has failed to produce a replacement edge, we need to perform this check explicitly, otherwise we would not be guaranteed to provide correct answers to the queries. Since there might be a lot of edges which are adjacent to T , this explicit check could be an expensive operation, so it should be made a low probability event for the randomized algorithm. This can produce pathological updates, 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 graph decomposition is used to prevent the undesirable behavior described above. If a spanning forest edge e is deleted from a tree at some level i , random sampling is used to quickly find a replacement for e at that level. If random sampling succeeds, the tree 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 $i + 1$.

THEOREM 1.4 (Henzinger and King [24]) *Let G be a graph with m_0 edges and n vertices subject to edge deletions only. A spanning forest of G can be maintained in $O(\log^3 n)$ expected amortized time per deletion, if there are at least $\Omega(m_0)$ deletions. The time per query is $O(\log n)$.*

1.3 Tools for directed graphs

In this section we discuss the main techniques used to solve dynamic path problems on directed graphs. We first address combinatorial and algebraic properties, and then we consider some efficient data structures, which are used as building blocks in designing dynamic algorithms for transitive closure and shortest paths. Similarly to the case of undirected graphs seen in Section 1.2, data structures that maintain properties of dynamically changing trees, such as the ones described in Chapter 1 (reachability trees), are often used as building blocks by many dynamic graph algorithms.

1.3.1 Kleene closures

Path problems such as transitive closure and shortest paths are tightly related to matrix sum and matrix multiplication over a closed semiring (see [7] for more details). In particular, the transitive closure of a directed graph can be obtained from the adjacency matrix of the graph via operations on the semiring of Boolean matrices, that we denote by $\{+, \cdot, 0, 1\}$. In this case, $+$ and \cdot denote the usual sum and multiplication over Boolean matrices.

LEMMA 1.1 Let $G = (V, E)$ be a directed graph and let $TC(G)$ be the (reflexive) transitive closure of G . If X is the Boolean adjacency matrix of G , then the Boolean adjacency matrix of $TC(G)$ is the Kleene closure of X on the $\{+, \cdot, 0, 1\}$ Boolean semiring:

$$X^* = \sum_{i=0}^{n-1} X^i.$$

Similarly, shortest path distances in a directed graph with real-valued edge weights can be obtained from the weight matrix of the graph via operations on the semiring of real-valued matrices, that we denote by $\{\oplus, \odot, \mathcal{R}\}$, or more simply by $\{\min, +\}$. Here, \mathcal{R} is the set of real values and \oplus and \odot are defined as follows. Given two real-valued matrices A and B , $C = A \odot B$ is the matrix product such that $C[x, y] = \min_{1 \leq z \leq n} \{A[x, z] + B[z, y]\}$ and $D = A \oplus B$ is the matrix sum such that $D[x, y] = \min\{A[x, y], B[x, y]\}$. We also denote by AB the product $A \odot B$ and by $AB[x, y]$ entry (x, y) of matrix AB .

LEMMA 1.2 Let $G = (V, E)$ be a weighted directed graph with no negative-length cycles. If X is a weight matrix such that $X[x, y]$ is the weight of edge (x, y) in G , then the distance matrix of G is the Kleene closure of X on the $\{\oplus, \odot, \mathcal{R}\}$ semiring:

$$X^* = \bigoplus_{i=0}^{n-1} X^i.$$

We now briefly recall two well-known methods for computing the Kleene closure X^* of X . In the following, we assume that X is an $n \times n$ matrix.

Logarithmic decomposition

A simple method to compute X^* , based on repeated squaring, requires $O(n^\mu \cdot \log n)$ worst-case time, where $O(n^\mu)$ is the time required for computing the product of two matrices over a closed semiring. This method performs $\log_2 n$ sums and products of the form $X_{i+1} = X_i + X_i^2$, where $X = X_0$ and $X^* = X_{\log_2 n}$.

Recursive decomposition

Another method, due to Munro [32], is based on a Divide and Conquer strategy and computes X^* in $O(n^\mu)$ worst-case time. Munro observed that, if we partition X in four submatrices A, B, D, C of size $n/2 \times n/2$ (considered in clockwise order), and X^* similarly in four submatrices E, F, H, G of size $n/2 \times n/2$, then X^* is defined recursively according to the following equations:

$$E = (A + BD^*C)^* \quad | \quad F = EBD^* \quad | \quad G = D^*CE \quad | \quad H = D^* + D^*CEBD^*$$

Surprisingly, using this decomposition the cost of computing X^* starting from X is asymptotically the same as the cost of multiplying two matrices over a closed semiring.

1.3.2 Locality

Recently, Demetrescu and Italiano [10] 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 1.3 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 1.5 *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 1.5 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 1.4 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 1.5 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 1.3 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 1.6 (Demetrescu and Italiano [10]) *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 [10] for a detailed description of this smoothing strategy. According to Theorem 1.6, 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 [10], locally historical paths can be maintained very efficiently. Since by Lemma 1.3 locally historical paths include shortest paths, this yields the fastest known algorithm for fully dynamic all pairs shortest paths.

1.3.3 Long paths

In this section we recall an intuitive combinatorial property of long paths in a graph. 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, the long paths property was first given in [21], and later on it has been used many times in designing efficient algorithms for transitive closure and shortest paths (see e.g., [9, 28, 41, 42]). The following theorem is from [41].

THEOREM 1.7 (Ullman and Yannakakis [41]) *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 $(cn \log n)/|S|$ 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 .*

Zwick [42] showed it is possible to choose set S deterministically by a reduction to a hitting set problem [5, 31]. King used a similar idea for maintaining fully dynamic shortest paths [28].

1.3.4 Matrices

Another useful data structure for keeping information about paths in dynamic directed graphs is based on matrices subject to dynamic changes. As we have seen in Section 1.3.1, Kleene closures can be constructed by evaluating polynomials over matrices. It is therefore natural to consider data structures for maintaining polynomials of matrices subject to updates of entries, like the one introduced in [8].

In the case of Boolean matrices, the problem can be stated as follows. Let P be a polynomial over $n \times n$ Boolean matrices with constant degree, constant number of terms, and variables $X_1 \dots X_k$. We wish to maintain a data structure for P subject to any intermixed sequence of update and query operations of the following kind:

SetRow($i, \Delta X, X_b$): sets to one the entries in the i -th row of variable X_b of polynomial

P corresponding to one-valued entries in the i -th row of matrix ΔX .

SetCol($i, \Delta X, X_b$): sets to one the entries in the i -th column of variable X_b of polynomial P corresponding to one-valued entries in the i -th column of matrix ΔX .

Reset($\Delta X, X_b$): resets to zero the entries of variable X_b of polynomial P corresponding to one-valued entries in matrix ΔX .

Lookup(ΔX): returns the maintained value of P .

We add to the previous four operations a further update operation especially designed for maintaining path problems:

LazySet($\Delta X, X_b$): sets to 1 the entries of variable X_b of P corresponding to one-valued entries in matrix ΔX . However, the maintained value of P might not be immediately affected by this operation.

Let C_P be the correct value of P that we would have by recomputing it from scratch after each update, and let M_P be the actual value that we maintain. If no **LazySet** operation is ever performed, then always $M_P = C_P$. Otherwise, M_P is not necessarily equal to C_P , and we guarantee the following weaker property on M_P : if $C_P[u, v]$ flips from 0 to 1 due to a **SetRow**/**SetCol** operation on a variable X_b , then $M_P[u, v]$ flips from 0 to 1 as well. This means that **SetRow** and **SetCol** always correctly reveal new 1's in the maintained value of P , possibly taking into account the 1's inserted through previous **LazySet** operations. This property is crucial for dynamic path problems, since the appearance of new paths in a graph after an edge insertion, which corresponds to setting a bit to one in its adjacency matrix, is always correctly recorded in the data structure.

LEMMA 1.4 (Demetrescu and Italiano [8]) Let P be a polynomial with constant degree of matrices over the Boolean semiring. Any **SetRow**, **SetCol**, **LazySet**, and **Reset** operation on a polynomial P can be supported in $O(n^2)$ amortized time. **Lookup** queries are answered in optimal time.

Similar data structures can be given for settings different from the semiring of Boolean matrices. In particular, in [9] the problem of maintaining polynomials of matrices over the $\{\min, +\}$ semiring is addressed.

The running time of operations for maintaining polynomials in this semiring is given below.

THEOREM 1.8 (Demetrescu and Italiano [9]) Let P be a polynomial with constant degree of matrices over the $\{\min, +\}$ semiring. Any **SetRow**, **SetCol**, **LazySet**, and **Reset** operation on variables of P can be supported in $O(D \cdot n^2)$ amortized time, where D is the maximum number of different values assumed by entries of variables during the sequence of operations. **Lookup** queries are answered in optimal time.

1.4 Connectivity

In this section and in Section 1.5 we consider dynamic problems on undirected graphs, showing how to deploy some of the techniques and data structures presented in Section 1.2 to obtain efficient algorithms. These algorithms maintain efficiently some property of an undirected graph that is undergoing structural changes defined by insertion and deletion of

edges, and/or updates of edge costs. To check the graph property throughout a sequence of these updates, the algorithms must be prepared to answer queries on the graph property efficiently. In particular, in this section we address the *fully dynamic connectivity* problem, where we are interested in algorithms that are capable of inserting edges, deleting edges, and answering a query on whether the graph is connected, or whether two vertices are in the same connected component. We recall that the goal of a dynamic algorithm is to minimize the amount of recomputation required after each update. All the dynamic algorithms that we describe are able to maintain dynamically the graph property at a cost (per update operation) which is significantly smaller than the cost of recomputing the graph property from scratch.

1.4.1 Updates in $O(\log^2 n)$ time

In this section we give a high level description of the fastest deterministic algorithm for the fully dynamic connectivity problem in undirected graphs [26]: 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 [24], the deterministic algorithm in [26] maintains a spanning forest F of the dynamically changing graph G . As above, 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 non-tree 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:

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

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 non-tree 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 $\text{Replace}((u, w), \ell(e))$ is called with parameters e and $\ell(e)$. We now sketch the operations performed by this subroutine.

$\text{Replace}((u, w), \ell)$ 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 non-tree 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. Non-tree 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 ET-tree, as described in Chapter 1, 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 non-tree 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 [26] 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 [26], this can be reduced to $O(\log n / \log \log n)$ by using a $\Theta(\log n)$ -ary version of ET-trees.

THEOREM 1.9 (Holm et al. [26]) *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.*

1.5 Minimum spanning tree

One of the most studied dynamic problem on undirected graphs is the *fully dynamic minimum spanning tree* problem, which consists of maintaining a minimum spanning forest of a graph during insertions of edges, deletions of edges, and edge cost changes. In this section, we show that a few simple changes to the connectivity algorithm presented in Section 1.4 are sufficient to maintain a minimum spanning forest of a weighted undirected graph upon deletions of edges [26]. A general reduction from [23] can then be applied to make the deletions-only algorithm fully dynamic.

1.5.1 Deletions in $O(\log^2 n)$ time

In addition to starting from a *minimum* spanning forest, the only change concerns function `Replace`, that should be implemented so as to consider candidate replacement edges of level ℓ in order of increasing weight, and not in arbitrary order. To do so, the ET-trees from Chapter 1 can be augmented so that each node maintains the minimum weight of a non-tree edge incident to the Euler tour segment below it. All the operations can still be supported in $O(\log n)$ time, yielding the same time bounds as for connectivity.

We now discuss the correctness of the algorithm. In particular, function `Replace` returns a replacement edge of minimum weight on the highest possible level: it is not immediate that such a replacement edge has the minimum weight among all levels. This can be proved by first showing that the following invariant, proved in [26], is maintained by the algorithm:

Invariant (3): Every cycle \mathcal{C} has a non-tree edge of maximum weight and minimum level among all the edges in \mathcal{C} .

Invariant (3) can be used to prove that, among all the replacement edges, the lightest edge is on the maximum level. Let e_1 and e_2 be two replacement edges with $w(e_1) < w(e_2)$, and let \mathcal{C}_i be the cycle induced by e_i in F , $i = 1, 2$. Since F is a minimum spanning forest, e_i has maximum weight among all the edges in \mathcal{C}_i . In particular, since by hypothesis $w(e_1) < w(e_2)$, e_2 is also the heaviest edge in cycle $\mathcal{C} = (\mathcal{C}_1 \cup \mathcal{C}_2) \setminus (\mathcal{C}_1 \cap \mathcal{C}_2)$. Thanks to Invariant (3), e_2 has minimum level in \mathcal{C} , proving that $\ell(e_2) \leq \ell(e_1)$. Thus, considering non-tree edges from higher to lower levels is correct.

LEMMA 1.5 (Holm et al. [26]) There exists a deletions-only minimum spanning forest algorithm that can be initialized on a graph with n vertices and m edges and supports any sequence of edge deletions in $O(m \log^2 n)$ total time.

1.5.2 Updates in $O(\log^4 n)$ time

The reduction used to obtain a fully dynamic algorithm is a slight generalization of the construction proposed by Henzinger and King [23] and works as follows.

LEMMA 1.6 [23, 26] Suppose we have a deletions-only minimum spanning tree algorithm that, for any k and l , can be initialized on a graph with k vertices and l edges and supports any sequence of $\Omega(l)$ deletions in total time $O(l \cdot t(k, l))$, where t is a non-decreasing function. Then there exists a fully-dynamic minimum spanning tree algorithm for a graph with n nodes starting with no edges, that, for m edges, supports updates in time

$$O\left(\log^3 n + \sum_{i=1}^{3+\log_2 m} \sum_{j=1}^i t(\min\{n, 2^j\}, 2^j)\right)$$

We refer the interested reader to references [23] and [26] for the description of the construction that proves Lemma 1.6. From Lemma 1.5 we get $t(k, l) = O(\log^2 k)$. Hence, combining Lemmas 1.5 and 1.6, we get the claimed result.

THEOREM 1.10 (Holm et al. [26]) There exists a fully-dynamic minimum spanning forest algorithm that, for a graph with n vertices, starting with no edges, maintains a minimum spanning forest in $O(\log^4 n)$ amortized time per edge insertion or deletion.

1.6 Transitive closure

In the rest of this chapter we survey the newest results for dynamic problems on directed graphs. In particular, we focus on two of the most fundamental problems: transitive closure and shortest paths. These problems play a crucial role in many applications, including network optimization and routing, traffic information systems, databases, compilers, garbage collection, interactive verification systems, industrial robotics, dataflow analysis, and document formatting. In this section we consider the best known algorithms for fully dynamic transitive closure. Given a directed graph G with n vertices and m edges, the problem consists of supporting any intermixed sequence of operations of the following kind:

Insert(u, v): insert edge (u, v) in G ;

Delete(u, v): delete edge (u, v) from G ;

Query(x, y): answer a reachability query by returning “yes” if there is a path from vertex x to vertex y in G , and “no” otherwise;

A simple-minded solution to this problem consists of maintaining the graph under insertions and deletions, searching if y is reachable from x at any query operation. This yields $O(1)$ time per update (**Insert** and **Delete**), and $O(m)$ time per query, where m is the current number of edges in the maintained graph.

Another simple-minded solution would be to maintain the Kleene closure of the adjacency matrix of the graph, rebuilding it from scratch after each update operation. Using the recursive decomposition of Munro [32] discussed in Section 1.3.1 and fast matrix multiplication [6], this takes constant time per reachability query and $O(n^\omega)$ time per update, where $\omega < 2.38$ is the current best exponent for matrix multiplication.

Despite many years of research in this topic, no better solution to this problem was known until 1995, when Henzinger and King [24] proposed a randomized Monte Carlo algorithm with one-sided error supporting a query time of $O(n/\log n)$ and an amortized update time of $O(n\hat{m}^{0.58} \log^2 n)$, where \hat{m} is the average number of edges in the graph throughout the whole update sequence. Since \hat{m} can be as high as $O(n^2)$, their update time is $O(n^{2.16} \log^2 n)$. Khanna, Motwani and Wilson [27] 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.

King and Sagert [29] 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-sided error. The bounds of King and Sagert were further improved by King [28], 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. Using a different framework, in 2000 Demetrescu and Italiano [8] obtained a deterministic fully dynamic algorithm that achieves $O(n^2)$ amortized time per update for general directed graphs. We note that each update might change a portion of the transitive closure as large as $\Omega(n^2)$. Thus, if the transitive closure has to be maintained explicitly after each update so that queries can be answered with one lookup, $O(n^2)$ is the best update bound one could hope for.

If one is willing to pay more for queries, Demetrescu and Italiano [8] showed how to break the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure: building on a previous path counting technique introduced by King and Sagert [29], they devised a randomized algorithm with one-sided error for directed acyclic graphs that achieves $O(n^{1.58})$ worst-case time per update and $O(n^{0.58})$ worst-case time per query. Other recent results for dynamic transitive closure appear in [39].

1.6.1 Updates in $O(n^2 \log n)$ time

In this section we address the algorithm by King [28], who devised the first deterministic near-quadratic update algorithm for fully dynamic transitive closure. The algorithm is based on the reachability tree data structure considered in Chapter 1 and on the logarithmic decomposition discussed in Section 1.3.1.

It maintains explicitly the transitive closure of a graph G in $O(n^2 \log n)$ amortized time per update, and supports inserting and deleting several edges of the graph with just one operation. Insertion of a bunch of edges incident to a vertex and deletion of any subset of edges in the graph require asymptotically the same time of inserting/deleting just one edge.

The algorithm maintains $\log n + 1$ levels: level i , $0 \leq i \leq \log n$, maintains a graph G_i whose edges represent paths of length up to 2^i in the original graph G . Thus, $G_0 = G$ and $G_{\log n}$ is the transitive closure of G .

Each level graph G_i is built on top of the previous level graph G_{i-1} by keeping two trees of depth ≤ 2 rooted at each vertex v : an out-tree $OUT_i(v)$ maintaining vertices reachable from v by traversing at most two edges in G_{i-1} , and an in-tree $IN_i(v)$ maintaining vertices that reach v by traversing at most two edges in G_{i-1} . Trees $IN_i(v)$ can be constructed by considering the orientation of edges in G_{i-1} reversed. An edge (x, y) will be in G_i if and only if $x \in IN_i(v)$ and $y \in OUT_i(v)$ for some v . In/out trees are maintained with the deletions-only reachability tree data structure considered in Chapter 1.

To update the levels after an insertion of edges around a vertex v in G , the algorithm simply rebuilds $IN_i(v)$ and $OUT_i(v)$ for each i , $1 \leq i \leq \log n$, while other trees are not touched. This means that some trees might not be up to date after an insertion operation. Nevertheless, any path in G is represented in at least the in/out trees rooted at the latest updated vertex in the path, so the reachability information is correctly maintained. This idea is the key ingredient of King's algorithm.

When an edge is deleted from G_i , it is also deleted from any data structures $IN_i(v)$ and $OUT_i(v)$ that contain it. The interested reader can find further details in [28].

1.6.2 Updates in $O(n^2)$ time

In this section we address the algorithm by Demetrescu and Italiano [8]. The algorithm is based on the matrix data structure considered in Section 1.3.4 and on the recursive decomposition discussed in Section 1.3.1.

It maintains explicitly the transitive closure of a graph in $O(n^2)$ amortized time per update, supporting the same generalized update operations of King's algorithm, i.e., insertion of a bunch of edges incident to a vertex and deletion of any subset of edges in the graph with just one operation. This is the best known update bound for fully dynamic transitive closure with constant query time.

The algorithm maintains the Kleene closure X^* of the $n \times n$ adjacency matrix X of the graph as the sum of two matrices X_1 and X_2 . Let V_1 be the subset of vertices of the graph corresponding to the first half of indices of X , and let V_2 contain the remaining vertices. Both matrices X_1 and X_2 are defined according to Munro's equations of Section 1.3.1, but in such a way that paths appearing due to an insertion of edges around a vertex in V_1 are correctly recorded in X_1 , while paths that appear due to an insertion of edges around a vertex in V_2 are correctly recorded in X_2 . Thus, neither X_1 nor X_2 encode complete information about X^* , but their sum does. In more detail, assuming that X is decomposed in sub-matrices A, B, C, D as explained in Section 1.3.1, and that X_1 , and X_2 are similarly decomposed in sub-matrices E_1, F_1, G_1, H_1 and E_2, F_2, G_2, H_2 , the algorithm maintains X_1 and X_2 with the following 8 polynomials using the data structure discussed in Section 1.3.4:

$Q = A + BP^2C$	$E_2 = E_1BH_2^2CE_1$
$F_1 = E_1^2BP$	$F_2 = E_1BH_2^2$
$G_1 = PCE_1^2$	$G_2 = H_2^2CE_1$
$H_1 = PCE_1^2BP$	$R = D + CE_1^2B$

where $P = D^*$, $E_1 = Q^*$, and $H_2 = R^*$ are Kleene closures maintained recursively as smaller instances of the problem of size $n/2 \times n/2$.

To support an insertion of edges around a vertex in V_1 , strict updates are performed on polynomials Q , F_1 , G_1 , and H_1 using `SetRow` and `SetCol`, while E_2 , F_2 , G_2 , and R are updated with `LazySet`. Insertions around V_2 are performed symmetrically, while deletions are supported via `Reset` operations on each polynomial in the recursive decomposition. Finally, P , E_1 , and H_2 are updated recursively. The interested reader can find the low-level details of the method in [8].

1.7 All-pairs shortest paths

In this section we survey the best known algorithms for fully dynamic all pairs shortest paths (in short APSP). Given a weighted directed graph G with n vertices and m edges, the problem consists of supporting any intermixed sequence of operations of the following kind:

Update(u, v, w): updates the weight of edge (u, v) in G to the new value w (if $w = +\infty$ this corresponds to edge deletion);

Query(x, y): returns the distance from vertex x to vertex y in G , or $+\infty$ if no path between them exists;

The dynamic maintenance of shortest paths has a remarkably long history, as the first papers date back to 35 years ago [30, 33, 37]. After that, many dynamic shortest paths algorithms have been proposed (see, e.g., [13, 18, 19, 34, 35, 38]), but their running times in the worst case were comparable to recomputing APSP from scratch.

The first dynamic shortest path algorithms which are provably faster than recomputing APSP from scratch, only worked on graphs with small integer weights. In particular, Ausiello *et al.* [3] proposed a decrease-only 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.* [25] designed a fully dynamic algorithm for APSP on planar graphs with integer weights, with a running time of $O(n^{4/3} \log(nC))$ per operation. Fakcharoemphol and Rao in [15] 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 edge operation.

The first big step on general graphs and integer weights was made by King [28], who presented a fully dynamic algorithm for maintaining all pairs shortest paths 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.

Demetrescu and Italiano [9] gave the first algorithm for fully dynamic APSP on general directed graphs with real weights assuming that each edge weight can attain a limited number S of different *real* values throughout the sequence of updates. In particular, the algorithm supports each update in $O(n^{2.5} \sqrt{S \log^3 n})$ amortized time and each query in $O(1)$ worst-case time. The same authors discovered the first algorithm that solves the fully dynamic all pairs shortest paths problem in its generality [10]. The algorithm maintains explicitly information about shortest paths supporting any edge weight update in $O(n^2 \log^2 n)$

amortized time per operation in directed graphs with non-negative real edge weights. Distance queries are answered with one lookup and actual shortest paths can be reconstructed in optimal time. We note that each update might change a portion of the distance matrix as large as $\Omega(n^2)$. Thus, if the distance matrix has to be maintained explicitly after each update so that queries can be answered with one lookup, $O(n^2)$ is the best update bound one could hope for. Other deletions-only algorithms for APSP, in the simpler case of unweighted graphs, are presented in [4].

1.7.1 Updates in $O(n^{2.5}\sqrt{C\log n})$ time

In this section we consider the dynamic shortest paths algorithm by King [28]. The algorithm is based on the long paths property discussed in Section 1.3.3 and on the reachability tree data structure Chapter 1.

Similarly to the transitive closure algorithms described in Section 1.6 generalized update operations are supported within the same bounds, i.e., insertion (or weight decrease) of a bunch of edges incident to a vertex, and deletion (or weight increase) of any subset of edges in the graph with just one operation.

The main idea of the algorithm is to maintain dynamically all pairs shortest paths up to a distance d , and to recompute longer shortest paths from scratch at each update by stitching together shortest paths of length $\leq d$. For the sake of simplicity, we only consider the case of unweighted graphs: an extension to deal with positive integer weights less than C is described in [28].

To maintain shortest paths up to distance d , similarly to the transitive closure algorithm by King described in Section 1.6, the algorithm keeps a pair of in/out shortest paths trees $IN(v)$ and $OUT(v)$ of depth $\leq d$ rooted at each vertex v . Trees $IN(v)$ and $OUT(v)$ are maintained with the decremental data structure mentioned in Chapter 1. It is easy to prove that, if the distance d_{xy} between any pair of vertices x and y is at most d , then d_{xy} is equal to the minimum of $d_{xv} + d_{vy}$ over all vertices v such that $x \in IN(v)$ and $y \in OUT(v)$. To support updates, insertions of edges around a vertex v are handled by rebuilding only $IN(v)$ and $OUT(v)$, while edge deletions are performed via operations on any trees that contain them. The amortized cost of such updates is $O(n^2d)$ per operation.

To maintain shortest paths longer than d , the algorithm exploits the long paths property of Theorem 1.7: in particular, it hinges on the observation that, if H is a random subset of $\Theta((n \log n)/d)$ vertices in the graph, then the probability of finding more than d consecutive vertices in a path, none of which are from H , is very small. Thus, if we look at vertices in H as “hubs”, then any shortest path from x to y of length $\geq d$ can be obtained by stitching together shortest subpaths of length $\leq d$ that first go from x to a vertex in H , then jump between vertices in H , and eventually reach y from a vertex in H . This can be done by first computing shortest paths only between vertices in H using any cubic-time static all-pairs shortest paths algorithm, and then by extending them at both endpoints with shortest paths of length $\leq d$ to reach all other vertices. This stitching operations requires $O(n^2|H|) = O((n^3 \log n)/d)$ time.

Choosing $d = \sqrt{n \log n}$ yields an $O(n^{2.5}\sqrt{\log n})$ amortized update time. As mentioned in Section 1.3.3, since H can be computed deterministically, the algorithm can be derandomized. The interested reader can find further details on the algorithm in [28].

1.7.2 Updates in $O(n^2 \log^3 n)$ time

In this section we address the algorithm by Demetrescu and Italiano [10], who devised the first deterministic near-quadratic update algorithm for fully dynamic all-pairs shortest

paths. 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 1.3.2.

The main idea is to maintain dynamically the locally historical paths of the graph in a data structure. Since by Lemma 1.3 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 1.3.2 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 potentially uniform 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 [11], 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 potentially uniform paths. At the end of this phase, paths that become potentially uniform after the update are correctly inserted in the data structure.

The update algorithm spends constant time for each of the $O(zn^2)$ new potentially uniform path (see Theorem 1.6). 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 [10].

1.8 Conclusions

In this chapter we have surveyed the algorithmic techniques underlying the fastest known dynamic graph algorithms for several problems, both on undirected and on directed graphs. Most of the algorithms that we have presented achieve bounds that are close to optimum. In particular, we have presented fully dynamic algorithms with polylogarithmic amortized time bounds for connectivity and minimum spanning trees [26] on undirected graphs. It remains an interesting open problem to show whether polylogarithmic update bounds can be achieved also in the worst case: we recall that for both problems the current best worst-case bound is $O(\sqrt{n})$ per update, and it is obtained with the sparsification technique [12] described in Section 1.2.

For directed graphs, we have shown how to achieve constant-time query bounds and nearly-quadratic update bounds for transitive closure and all pairs shortest paths. These bounds are close to optimal in the sense that one update can make as many as $\Omega(n^2)$ changes to the transitive closure and to the all pairs shortest paths matrices. However, if the problem is just to maintain reachability or shortest paths between two fixed vertices s and t , no solution better than the static is known. Furthermore, if one is willing to pay more for queries, Demetrescu and Italiano [8] have shown how to break the $O(n^2)$ barrier on the single-operation complexity of fully dynamic transitive closure for directed acyclic graphs. It remains an interesting open problem to show whether effective query/update tradeoffs can be achieved for general graphs and for shortest paths problems. Furthermore, can we solve efficiently fully dynamic single-source shortest paths on general graphs?

Finally, dynamic algorithms for other fundamental problems such as matching and flow problems deserve further investigation.

Acknowledgement

This work has been supported in part by the IST Programmes of the EU under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33555 (COSIN), and by the Italian Ministry of University and Scientific Research (Project “ALINWEB: Algorithmics for Internet and the Web”).

References

- [1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th Int. Colloquium on Automata, Languages and Programming (ICALP 97)*, LNCS 1256, pages 270–280, 1997.
- [2] S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT 00)*, pages 46–56, 2000.
- [3] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–38, 1991.
- [4] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proc. 34th ACM Symposium on Theory of Computing (STOC’02)*, pages 117–123, 2002.
- [5] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.
- [6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9, 1990.
- [7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, Second Edition. The MIT Press, 2001.
- [8] C. Demetrescu and G.F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proc. of the 41st IEEE Annual Symposium on Foundations of Computer Science (FOCS’00)*, pages 381–389, 2000.
- [9] C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS’01), Las Vegas, Nevada*, pages 260–267, 2001.
- [10] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proc. 35th Symp. on Theory of Computing (STOC’03)*, 2003.
- [11] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [12] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. *J. Assoc. Comput. Mach.*, 44:669–696, 1997.
- [13] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.
- [14] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. Assoc. Comput. Mach.*, 28:1–4, 1981.
- [15] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS’01), Las Vegas, Nevada*, pages 232–241, 2001.
- [16] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.
- [17] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2): 484–538, 1997.

- [18] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single source shortest paths trees. *Algorithmica*, 22(3):250–274, 1998.
- [19] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:251–281, 2000.
- [20] Z. Galil and G. F. Italiano. Fully-dynamic algorithms for 2-edge connectivity. *SIAM J. Comput.*, 21:1047–1069, 1992.
- [21] D. H. Greene and D.E. Knuth. *Mathematics for the analysis of algorithms*. Birkhäuser, 1982.
- [22] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.
- [23] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th Int. Colloquium on Automata, Languages and Programming (ICALP 97)*, pages 594–604, 1997.
- [24] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with poly-logarithmic time per operation. *J. Assoc. Comput. Mach.*, 46(4):502–536, 1999.
- [25] 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.
- [26] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. Assoc. Comput. Mach.*, 48(4):723–760, 2001.
- [27] S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead on dynamic graph problems. In *Algorithmica* 21(4): 377-394 (1998).
- [28] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40-th Symposium on Foundations of Computer Science (FOCS 99)*, 1999.
- [29] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proc. 31-st ACM Symposium on Theory of Computing (STOC 99)*, pages 492–498, 1999.
- [30] P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pages 96–109, 1967.
- [31] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.
- [32] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.
- [33] 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, UK, 1967.
- [34] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest path problem. *Journal of Algorithms*, 21:267–305, 1996.
- [35] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.
- [36] M. Rauch. Fully dynamic biconnectivity in graphs. In *Algorithmica*, 13:503-538, 1995.
- [37] V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Math. and Math. Phys.*, 8(5):336–343, 1968.
- [38] H. Rohnert. A dynamization of the all-pairs least cost problem. In *Proc. 2nd Annual Symposium on Theoretical Aspects of Computer Science, (STACS 85), LNCS 182*, pages 279–286, 1985.
- [39] 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'02)*, p. 679, Vancouver, Canada, 2002.

- [40] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comp. Syst. Sci.*, 24:362–381, 1983.
- [41] J.D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.
- [42] U. Zwick. All pairs shortest paths in weighted directed graphs - exact and almost exact algorithms. In *Proc. of the 39th IEEE Annual Symposium on Foundations of Computer Science (FOCS'98)*, pages 310–319, Los Alamitos, CA, November 8–11 1998.