10.2 DYNAMIC GRAPH ALGORITHMS

Camil Demetrescu, University of Rome "La Sapienza"
Irene Finocchi, University of Rome "Tor Vergata"
Giuseppe F. Italiano, University of Rome "Tor Vergata"

INTRODUCTION

In many applications of graph algorithms, including communication networks, VLSI design, graphics, and assembly planning, graphs are subject to discrete changes, such as additions or deletions of edges or vertices. In the last two 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.

DEFINITIONS

D1: 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.

D2: A ***dynamic graph*** is a graph that is undergoing a sequence of updates.

REMARKS

R1: 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.

R2: 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.

DEFINITIONS

D3: A dynamic graph problem is said to be *fully dynamic* if the update operations include unrestricted insertions and deletions of edges or vertices.

D4: A dynamic graph problem is said to be *partially dynamic* if only one type of update, either insertions or deletions, is allowed.

D5: A dynamic graph problem is said to be *incremental* if only insertions are allowed.

D6: A dynamic graph problem is said to be *decremental* if only deletions are allowed.

REMARKS

R3: In the first part of this work 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.

R4: In the second part of this work 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.

R5: 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.

DYNAMIC PROBLEMS ON UNDIRECTED GRAPHS

This part considers 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 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.

EXAMPLES

E1: The *fully dynamic minimum spanning tree* problem consists of maintaining a minimum spanning forest of a graph during insertions of edges, deletions of edges, and edge cost changes.

E2: 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 in the same connected component.

REMARKS

R6: The goal of a dynamic algorithm is to minimize the amount of recomputation required after each update.

R7: 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.

In the remainder of this part, first we present general techniques and tools used in designing dynamic algorithms on undirected graphs (Section 10.2.1), and then we survey the fastest algorithms for solving two of the most fundamental graph problems: connectivity (Section 10.2.2) and minimum spanning trees (Section 10.2.3).

10.2.1 General Techniques for Undirected Graphs

Many of the algorithms proposed in the literature use the same general techniques, and hence we begin by describing these techniques. As a common theme, 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 are often used as building blocks by many dynamic graph algorithms. The basic update operations are edge insertions and edge deletions. Many properties of dynamically changing trees have been considered in the literature.

EXAMPLES

E3: The basic query operation is tree membership: while the forest of trees is dynamically changing, we would like to know at any time which tree contains a given vertex, or whether two vertices are in the same tree. Dynamic tree membership is a special case of dynamic connectivity in undirected graphs, and indeed in Section 10.2.2 and in Section 10.2.3 we will see that some of the data structures presented here for trees are useful for solving the more general problem on graphs.

E4: Other properties that have been considered are finding the parent of a vertex, the least common ancestor of two vertices, the center or the diameter of a tree [AlHoDeTh97,AlHoTh00,SlTa83]. When costs are associated either to vertices or to edges, one could also ask what is the minimum or maximum cost in a given path.

In the remainder of this section we first present three different data structures that maintain properties of dynamically changing trees: topology trees, ET trees, and top trees. Next, we discuss techniques that can be applied on general undirected graphs: clustering, sparsification, and randomization. In the course of the presentation, we also highlight how these techniques have been applied to solve the fully dynamic connectivity and/or minimum spanning tree problems, and which update and query bounds can be achieved when they are deployed.

Topology Trees

Topology trees have been introduced by Frederickson [Fr85] to maintain dynamic trees upon insertions and deletions of edges.

DEFINITIONS

D7: Given a tree T of the forest, a *cluster* is a connected subgraph of T.

D8: The *cardinality* of a cluster is the number of its vertices.

D9: The *external degree* of a cluster is the number of tree edges incident to it.

D10: A *topology tree* is a hierarchical representation of a tree T of the forest: each level of the topology tree partitions the vertices of T into clusters. Clusters at level 0 contain one vertex each. Clusters at level $\ell \geq 1$ form a partition of the vertices of the tree T' obtained by shrinking each cluster at level $\ell - 1$ into a single vertex.

REMARK

R8: The basic partition must be suitably chosen so that the topology tree has depth $O(\log n)$ and, during edge insertions and deletions, each level of the topology tree can be updated by applying only a few local adjustments.

ASSUMPTION

In order to illustrate the solution proposed by Frederickson [Fr85,Fr97], we assume that T has maximum vertex degree 3: this is without loss of generality, since a standard transformation can be applied if this is not the case [Ha69].

DEFINITION

D11: A *restricted partition* of T is a partition of its vertex set V into clusters of external degree ≤ 3 and cardinality ≤ 2 such that:

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

(2) Each cluster of external degree < 3 has cardinality at most 2.

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

An example of topology tree, together with the restricted partitions used to obtain its levels, is given in Figure 10.2.1.

REMARKS

R9: There can be several restricted partitions for a given tree T, based upon different choices of the vertices to be unioned.

R10: Because of (3), the restricted partition implements a cluster-forming scheme according to a locally greedy heuristic, which does not always obtain the minimum number of clusters, but has the advantage of requiring only local adjustments during updates.

APPROACH

Edge deletion. We sketch how to update the clusters of a restricted partition when an edge e is deleted from T. First, removing e splits T into two trees, say T_1 and T_2, which inherit all of the clusters of T, possibly with the following exceptions.

♢ If e is entirely contained in a cluster, this cluster is no longer connected and therefore must be split. After the split, we must check whether each of the two resulting clusters is adjacent to a cluster of tree degree at most 2, and if these two adjacent clusters together have cardinality ≤ 2. If so, we combine these two clusters in order to maintain condition (3).

♢ If e is between two clusters, then no split is needed. However, since the tree degree of the clusters containing the endpoints of e has been decreased, we must check if each cluster should be combined with an adjacent cluster, again because of condition (3).

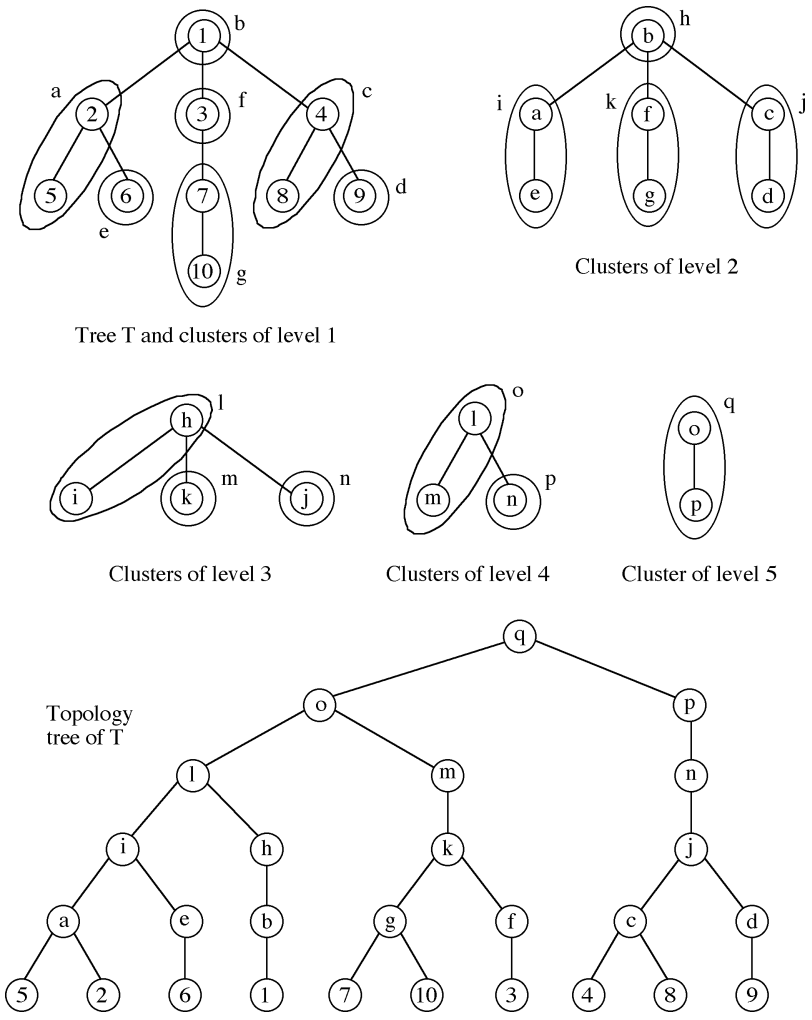

Figure 10.2.1 **Restricted partitions and topology tree of a tree T.**

Edge insertion. Similar local manipulations can be applied to restore invariants (1) - (3) in Definition D11 in case of edge insertions.

Construction of the topology tree. The levels of the topology tree are built in a bottom up fashion by repeatedly applying the locally greedy heuristic.

Update of the topology tree. Each level can be updated upon insertions and deletions of edges in tree T by applying few locally greedy adjustments similar to the ones described before. In particular, a constant number of basic clusters (corresponding to leaves in the topology tree) are examined: the changes in these basic clusters percolate up in the topology tree, possibly causing vertex clusters to be regrouped in different ways.

FACTS

F1: Each level of the topology tree has a number of nodes which is a constant fraction

of the previous level, and thus the number of levels is $O(\log n)$ (see [Fr85,Fr97]).

The fact that only a constant amount of work has to be done on $O(\log n)$ topology tree nodes implies a logarithmic bound on the update time.

F2: (Frederickson's Theorem) [Fr85] The update of a topology tree because of an edge insertion or deletion can be supported in $O(\log n)$ time.

ET Trees

ET trees have been introduced by Henzinger and King [HeKi99] to work on dynamic forests whose vertices are associated with weighted or unweighted keys. Updates allow it to cut arbitrary edges, to insert edges linking different trees of the forest, and to add or remove the weighted key associated to a vertex. Supported queries are the following:

♢ `Connected`(u, v): tells whether vertices u and v are in the same tree.

♢ `Size`(v): returns the number of vertices in the tree that contains v.

♢ `Minkey`(v): returns a key of minimum weight in the tree that contains v; if keys are unweighted, an arbitrary key is returned.

DEFINITIONS

D12: An *Euler tour* of a tree T is a maximal closed walk over the graph obtained by replacing each edge of T by two directed edges with opposite direction. The walk traverses each edge exactly once; hence, if T has n vertices, the Euler tour has length $2n - 2$ (see Figure 10.2.2).

D13: An *ET tree* is a dynamic balanced binary tree (see [CLRS01] for a definition of balanced binary trees) over some Euler tour around T. Namely, leaves of the balanced binary tree are the nodes of the Euler Tour, in the same order in which they appear (see Figure 10.2.2).

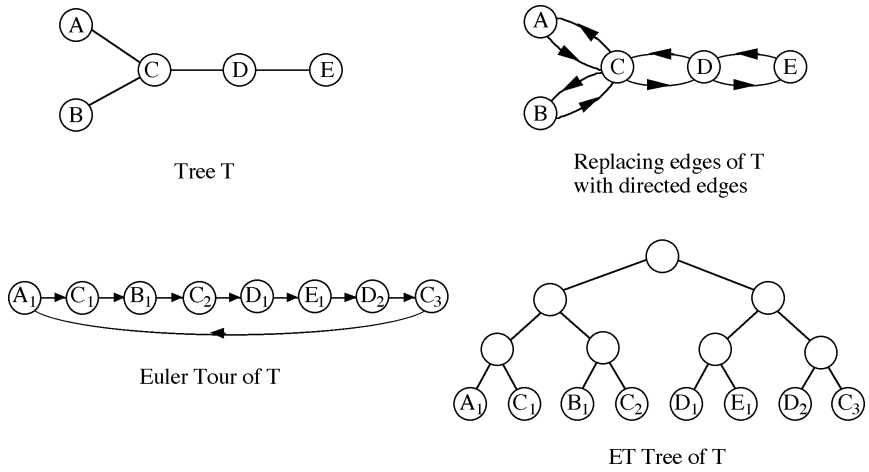

Figure 10.2.2 Euler Tour and ET Tree of a tree T.

REMARK

R11: Despite each vertex of T may occur several times in the Euler tour (an arbitrary occurrence is marked as *representative* of the vertex), an ET tree has $O(n)$ nodes.

APPROACH

Edge insertion and deletion. If trees in the forest are linked or cut, a constant number of splits and concatenations allow it to reconstruct the new Euler tour(s); the ET tree(s) can then be rebalanced by affecting only $O(\log n)$ nodes.

Connectivity queries. The query `Connected`(u, v) can be easily supported in $O(\log n)$ time by finding the roots of the ET trees containing u and v and checking if they coincide.

Size and Minkey queries. To support `Size` and `Minkey` queries, each node q of the ET tree maintains two additional values: the number $s(q)$ of representatives below it and the minimum weight key $k(q)$ attached to a representative below it. Such values can be maintained in $O(\log n)$ time per update and allow it to answer queries of the form `Size`(v) and `Minkey`(v) in $O(\log n)$ time for any vertex v of the forest: the root r of the ET tree containing v is found and values $s(r)$ and $k(r)$ are returned, respectively. We refer the interested reader to [HeKi99] for additional details of the method.

FACT

F3: Both updates and queries can be supported in $O(\log n)$ time using ET trees (see [HeKi99]).

Top trees

Top trees have been introduced by Alstrup et al. [AlHoDeTh97] to maintain efficiently information about paths in trees, such as, e.g., the maximum weight on the path between any pair of vertices in a tree. The basic idea is taken from Frederickson's topology trees, but instead of partitioning vertices, top trees work by partitioning edges: the same vertex can then appear in more than one cluster.

DEFINITIONS

D14: Similarly to [Fr85,Fr97], a *cluster* is a connected subtree of tree T, with the additional constraint that at most two vertices, called *boundary vertices*, have edges out of the subtree.

D15: Two clusters are said to be *neighbors* if their intersection contains exactly one vertex.

D16: A *top tree* of T is a binary tree such that:

- ♢ The leaves and the internal nodes represent edges and clusters of T, respectively.
- ♢ The subtree represented by an internal node is the union of the subtrees represented by its two children, which must be neighbors.
- ♢ The root represents the entire tree T.
- ♢ The height is $O(\log n)$.

We refer to Figure 10.2.3 for an example of top tree.

APPROACH

Top trees can be maintained under edge insert and delete operations in tree T by making use of two basic `Merge` and `Split` operations.

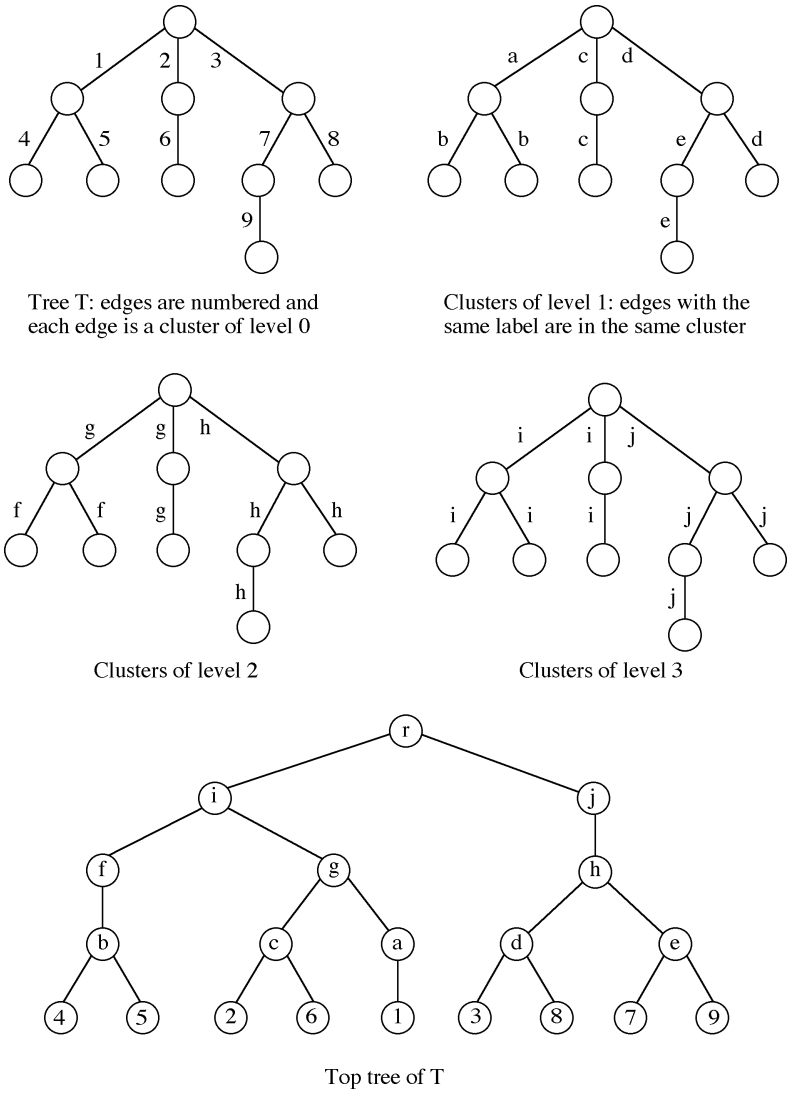

Figure 10.2.3 Clusters and top tree of a tree T.

Merge. It takes two top trees whose roots are neighbor clusters and joins them to form a unique top tree.

Split. This is the reverse operation, deleting the root of a given top tree.

Edge insertion and deletion. The implementation of an edge insertion/deletion starts with a sequence of `Split` of all ancestor clusters of edges whose boundary changes and finishes with a sequence of `Merge`. Since an end-point v of an edge has to be already boundary vertex of the edge if v is not a leaf, each edge insert/delete can change the boundary of at most two edges, excluding the edge being inserted/deleted.

From [AlHoDeTh97,Fr85] we have:

FACT

F4: (Alstrup et al.) [AlHoDeTh97] For a dynamic forest we can maintain top trees

of height $O(\log n)$ supporting edge insertions and deletions with a sequence of $O(\log n)$ `Split` and `Merge`. The sequence itself is identified in $O(\log n)$ time.

REMARKS

R12: Top trees are typically used by attaching extra information to their nodes. A careful choice of the extra information makes it possible to maintain easily path properties of trees, such as the maximum weight of an edge in the unique path between any two vertices. We refer the interested reader to [AlHoDeTh97,AlHoTh00,HoDeTh01] for sample applications.

R13: Top trees are a natural generalization of standard balanced binary trees over dynamic collections of lists that may be concatenated and split, where each node of the balanced binary tree represents a segment of a list. In the terminology of top trees, this is just a special case of a cluster.

Clustering

The clustering technique has been introduced by Frederickson [Fr85] 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.

REMARKS

R14: Typically, the decomposition defined by the clusters is applied recursively and the information about the subgraphs is combined with the topology trees described above.

R15: A refinement of the clustering technique appears in the idea of *ambivalent data structures* [Fr97], in which edges can belong to multiple groups, only one of which is actually selected depending on the topology of the given spanning tree.

EXAMPLE

E5: As an example, we briefly describe the application of clustering to the problem of maintaining a minimum spanning forest [Fr85]. 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 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 $E \setminus S$ [Fr85].

FACTS

F5: 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 [GaIt92,Ra95]).

When the partition can be applied recursively, better $O(m^{1/2})$ time bounds can be achieved by using 2-dimentional topology trees (see, for instance, [Fr85,Fr97]).

F6: (Frederickson's theorem) [Fr85] The minimum spanning forest of an undirected graph can be maintained in time $O(m^{1/2})$ per update, where m is the current number of edges in the graph.

REMARKS

R16: We refer the interested reader to [Fr85,Fr97] for details about Frederickson's algorithm. With the same technique, an $O(m^{1/2})$ time bound can be obtained also for fully dynamic connectivity and 2-edge connectivity [Fr85,Fr97].

R17: The type of clustering used can very problem-dependent, however, and makes this technique difficult to be used as a black box.

Sparsification

Sparsification is a general technique due to Eppstein *et al.* [EpGaItNi97] 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 $T(n, m) = O(m^{1/2})$, we get a better bound of $O(n^{1/2})$. The technique itself is quite simple. A key concept is the notion of certificate.

DEFINITION

D17: 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.

APPROACH

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 $O(\log(m/n))$ graphs with $O(n)$ edges each, instead of one graph with m edges.

NOTATION

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

REMARKS

There exist two variants of sparsification.

R18: 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.

R19: 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 [EpGaItNi97] for a precise definition of stability.) This variant transforms time bounds of the form $O(m^p)$ into $O(n^p)$.

DEFINITION

D18: 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.

FACTS

F7: (Eppstein *et al.*) [EpGaItNi97] Let 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 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 $q(n,O(n))$.

F8: (Eppstein *et al.*) [EpGaItNi97] 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 $q(n,O(n))$.

REMARKS

R20: Basically, the first version of sparsification (Fact F7) can be used to dynamize static algorithms, in which case we only need to *compute* efficiently *sparse* certificates, while the second version (Fact F8) can be used to speed up existing fully dynamic algorithms, in which case we need to *maintain* efficiently *stable sparse* certificates.

R21: 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 $O(m^{1/2})$ [Fr85,Fr97] to $O(n^{1/2})$ [EpGaItNi97].

R22: 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.

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 [HeKi99], and allows one to achieve faster update times for some problems by exploiting the power of randomization.

EXAMPLE

We sketch how the randomization technique works taking the fully dynamic connectivity problem as an example.

NOTATION

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*.

APPROACH

Maintaining spanning forests. Trees in the spanning forests are maintained using the Euler Tours data structure (ET trees) described above, that allows one to obtain logarithmic updates and queries within the forest.

Random sampling. A key idea behind the technique of Henzinger and King 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 second key idea is to combine randomization with a suitable graph decomposition. 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. 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 also maintained.

REMARKS

R23: Note that the hard operation is the deletion of a tree edge: indeed, a spanning forest is easily maintained throughout edge insertions, and deleting a non-tree edge does not change the forest.

R24: 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.

R25: 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.

R26: 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 $i + 1$.

FACT

F9: (Henzinger and King's Theorem) [HeKi99] 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 $O(\log n)$.

10.2.2 Connectivity

In this section we give a high level description of the fastest deterministic algorithm for the fully dynamic connectivity problem in undirected graphs [HoDeTh01]: 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.

REMARKS

R27: Similarly to te randomized algorithm in [HeKi99], the deterministic algorithm in [HoDeTh01] maintains a spanning forest F of the dynamically changing graph G. As above, we will refer to the edges in F as *tree edges*.

FACTS

F10: 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.

F11: 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 $\geq i$.

F12: $F = F_0 \supseteq F_1 \supseteq F_2 \supseteq \ldots \supseteq F_L$, where L denotes the maximum edge level.

F13: 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.

INVARIANTS

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 $n/2^i$.

REMARKS

R28: 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 $F_{\ell(u,v)}$.

R29: Invariant (2) implies that the maximum number of levels is $L \leq \lfloor \log_2 n \rfloor$.

FACTS

F14: 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.

F15: 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`$((u, w), \ell(e))$ is called with parameters e and $\ell(e)$. We now sketch the operations performed by this subroutine.

F16: `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.

F17: 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.

F18: We maintain an ET-tree, as described in Section 10.2.1, 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.

REMARKS

R30: 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 [HoDeTh01] for details.

R31: 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 [HoDeTh01], this can be reduced to $O(\log n / \log \log n)$ by using a $\Theta(\log n)$-ary version of ET-trees.

FACT

F19: (Holm et al.) [HoDeTh01] 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.

10.2.3 Minimum Spanning Trees

A few simple changes to the connectivity algorithm presented in Section 10.2.2 are sufficient to maintain a minimum spanning forest of a weighted undirected graph upon deletions of edges [HoDeTh01]. A general reduction from [HeKi97] can then be applied to make the deletions-only algorithm fully dynamic.

Decremental Minimum Spanning Tree

APPROACH

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 Section 10.2.1 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 [HoDeTh01], is maintained by the algorithm.

INVARIANT

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

FACTS

F20: 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.

F21: (Holm et al.) [HoDeTh01] 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.

Fully Dynamic Minimum Spanning Tree

The reduction used to obtain a fully dynamic algorithm is a slight generalization of the construction proposed by Henzinger and King [HeKi97] and works as follows.

FACTS

F22: (HeKi97,HoDeTh01) 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 [HeKi97] and [HoDeTh01] for the description of the construction that proves Fact F22. From Fact F21 we get $t(k,l) = O(\log^2 k)$. Hence, combining Fact F21 and Fact F22, we get the claimed result.

F23: (Holm et al.) [HoDeTh01] 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.

DYNAMIC PROBLEMS ON DIRECTED GRAPHS

In this section 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.

We first present general techniques and tools used in designing dynamic path problems on directed graphs (Section 10.2.4), and then we address the newest results for dynamic transitive closure and dynamic shortest paths (Section 10.2.5 and Section 10.2.6, respectively). In the first problem, the goal is to maintain reachability information in a directed graph subject to insertions and deletions of edges. The fastest known algorithms support graph updates in quadratic (or near-quadratic) time and reachability queries in constant time [DeIt00,Ki99]. In the second problem, we wish to maintain information about shortest paths in a directed graph subject to insertion and deletion of edges, or updates of edge weights. Similarly to dynamic transitive closure, this can be done in near-quadratic time per update and optimal time per query [DeIt03].

10.2.4 General Techniques 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.

Path Problems and 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 [CoLeRiSt01] 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.

FACT

F24: 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:

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

REMARK

R32: 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 $\{\min, +\}$.

NOTATION

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 $C[x, y] = \min_{1 \le z \le n}\{A[x, z] + B[z, y]\}$ 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.

FACT

F25: 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:

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

APPROACH

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.

FACT

F26: The 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 [Mu71], is based on a Divide and Conquer strategy and computes X^* in $O(n^\mu)$ worst-case time.

FACT

F27: 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:

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

REMARK

R33: 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.

Uniform Paths

In this section we address some combinatorial properties of shortest paths in directed graphs that have been recently discovered by Demetrescu and Italiano [DeIt03]. In particular, we consider shortest paths as a special case of a broader class of paths called *uniform paths*.

DEFINITION

D19: A path π in a graph is *uniform* if every proper subpath of π is a shortest path.

REMARKS

R34: As an alternative equivalent definition, a path π_{xy} is uniform in a graph if every edge (u, v) in π_{xy} satisfies the relation $d_{xu} + w_{uv} + d_{vy} = w(\pi_{xy})$, where d_{xy} denotes the distance between vertex x and vertex y in the graph, w_{uv} is the weight of edge (u, v), and $w(\pi_{xy})$ is the weight of π_{xy}. This accounts for the terminology used in Definition D19.

R35: It is not difficult to prove that the number of uniform 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.

To characterize how uniform paths change in a fully dynamic graph, we consider the notions of *historical shortest path* and *potentially uniform path*.

DEFINITIONS

D20: A *historical shortest path* is a path that has been a shortest path at some point during the sequence of updates, and none of its edges have been updated since then.

Using this notion we can define a superset of uniform paths that are called *potentially uniform paths*.

D21: A path π in a graph is *potentially uniform* if every proper subpath of π is a historical shortest path.

The following lemma addresses the relationship between shortest paths, uniform paths, historical shortest paths, and potentially uniform paths in a dynamic graph:

FACT

F28: (Demetrescu and Italiano [DeIt03]) If we denote by SP, UP, HSP, and PUP respectively the sets of shortest paths, uniform paths, historical shortest paths, and potentially uniform paths in a graph, then at any time the following inclusions hold: $SP \subseteq UP \subseteq PUP$ and $SP \subseteq HSP \subseteq PUP$.

REMARK

R36: Potentially uniform paths exhibit strong combinatorial properties in graphs subject to (fully) dynamic updates. In particular, it is possible to prove that the number of paths that become potentially uniform in a graph at each edge weight update depends on the number of historical shortest paths in the graph.

FACT

F29: (Demetrescu and Italiano [DeIt03]) 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 $O(z)$ historical shortest paths between each pair of vertices, then the amortized number of paths that become potentially uniform at each update is $O(zn^2)$.

REMARK

R37: To keep changes in potentially uniform paths small, it is then desirable to have as few historical shortest paths as possible. Indeed, it is possible to transform every update sequence into a slightly longer equivalent sequence that generates only a few historical shortest 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 [DeIt03] for a detailed description of this smoothing strategy. According to Fact F29, this technique implies that only $O(n^2 \log k)$ potentially uniform paths change at each edge weight update in the smoothed sequence $F(\Sigma)$.

R38: As elaborated in [DeIt03], potentially uniform paths can be maintained very efficiently. Since by Fact F28 potentially uniform paths include shortest paths, this yields the fastest known algorithm for fully dynamic all pairs shortest paths.

Long Paths Property

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.

REMARK

R39: To the best of our knowledge, the long paths property was first given in [GrKn82], and later on it has been used may times in designing efficient algorithms for transitive closure and shortest paths (see e.g., [DeIt01,Ki99,UlYa91,Zw98]).

The following theorem is from [UlYa91].

FACT

F30: (Ullman and Yannakakis [UlYa91]) 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.

REMARK

R40: As shown in [Zw98], it is possible to choose set S deterministically by a reduction to a hitting set problem [Ch79,Lo75]. A similar technique has also been used in [Ki99].

Reachability Trees

In this section we consider a tree data structure that has been widely used to solve dynamic path problems on directed graphs.

REMARKS

R41: The first appearance of this tool dates back to 1981, when Even and Shiloach showed how to maintain a breadth-first tree of an undirected graph under any sequence of edge deletions [EvSh81]; they used this as a kernel for decremental connectivity on undirected graphs.

R42: Later on, Henzinger and King [HeKi99] showed how to adapt this data structure to fully dynamic transitive closure in directed graphs.

R43: King [Ki99] designed an extension of this tree data structure to weighted directed graphs for solving fully dynamic all pairs shortest paths.

PROBLEM

In the unweighted directed version, the goal is to maintain information about breadth-first search (BFS) on a directed 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 $d \leq n$. Given a directed graph $G = (V, E)$ and a vertex $r \in V$, we would like to support any intermixed sequence of the following operations:

DEFINITIONS

D22: `Delete`(x, y): delete edge (x, y) from G.

D23: `Level`(u): return the level of vertex u in the BFS tree of depth d rooted at r (return $+\infty$ if u is not reachable from r within distance d).

In [Ki99] it is shown how to maintain efficiently the BFS levels, supporting any `Level` operation in constant time and any sequence of `Delete` operations in $O(md)$ overall time:

FACT

F31: (King [Ki99]) Maintaining BFS levels up to depth d from a given root requires $O(md)$ time in the worst case throughout any sequence of edge deletions in a directed graph with m initial edges.

REMARKS

R44: Fact F31 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.

R45: As it was shown in [HeKi99,Ki99], 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 [Ki99].

Matrix Data Structures

In this section, we consider matrix data structures for keeping information about paths in dynamic directed graphs. As we have seen above (Path Problems and Kleene Closures), 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 [DeIt00].

PROBLEM

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 $X_1 \ldots X_k$. We wish to maintain a data structure for P subject to any intermixed sequence of update and query operations of the following kind:

DEFINITIONS

D24: `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.

D25: `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.

D26: `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.

D27: `Lookup`(): returns the maintained value of P.

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

D28: `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.

REMARK

R46: 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 in-

serted through previous `LazySet` operations. This property is crucial for dynamic path problems.

FACT

F32: (Demetrescu and Italiano [DeIt00]) 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.

REMARK

R47: Similar data structures can be given for settings different from the semiring of Boolean matrices. In particular, in [DeIt01] 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.

FACT

F33: (Demetrescu and Italiano [DeIt01]) 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.

10.2.5 Dynamic Transitive Closure

In this section we survey 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:

DEFINITIONS

D29: `Insert`(u, v): insert edge (u, v) in G;

D30: `Delete`(u, v): delete edge (u, v) from G;

D31: `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;

FACTS

F34: 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.

F35: 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 [Mu71] discussed in Section 10.2.4 (Path Problems and Kleene Closures) and fast matrix multiplication [CoWi90], 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.

REMARKS

R48: Despite many years of research in this topic, no better solution to this problem was known until 1995, when Henzinger and King [HeKi99] 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 $O(n^{2.16}\log^2 n)$.

R49: Khanna, Motwani and Wilson [KhMoWi96] 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.

R50: King and Sagert [KiSa99] 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.

R51: The bounds of King and Sagert were further improved by King [Ki99], 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.

R52: Using a different framework, in 2000 Demetrescu and Italiano [DeIt00] obtained a deterministic fully dynamic algorithm that achieves $O(n^2)$ amortized time per update for general directed graphs.

R53: 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.

R54: If one is willing to pay more for queries, Demetrescu and Italiano [DeIt00] 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 [KiSa99], 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.

R55: Other recent results for dynamic transitive closure appear in [RoZw02].

King's $O(n^2\log n)$ Update Algorithm

In this section we address the algorithm by King [Ki99], who devised the first deterministic near-quadratic update algorithm for fully dynamic transitive closure.

REMARKS

R56: The algorithm is based on the tree data structure considered in Section 10.2.4 (Tree Data Structures) and on the logarithmic decomposition discussed in Section 10.2.4 (Path Problems and Kleene Closures).

R57: 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.

R58: 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.

APPROACH

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.

FACTS

F36: Each level i is built on top of the previous level $i-1$ by keeping two trees of depth ≤ 2 rooted at each vertex v of G: 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}. 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.

F37: The $2 \log n$ trees $IN_i(v)$ and $OUT_i(v)$ are maintained with instances of the BFS tree data structure considered in Section 10.2.4 (Tree Data Structures).

F38: 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.

F39: 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 [Ki99].

Demetrescu and Italiano's $O(n^2)$ Update Algorithm

In this section we address the algorithm by Demetrescu and Italiano [DeIt00].

REMARKS

R59: The algorithm is based on the matrix data structure considered in Section 10.2.4 (Matrix Data Structures) and on the recursive decomposition discussed in Section 10.2.4 (Path Problems and Kleene Closures).

R60: 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.

R61: This is the best known update bound for fully dynamic transitive closure with constant query time.

APPROACH

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.

NOTATION

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.

FACTS

F40: Both matrices X_1 and X_2 are defined according to Munro's equations of Section 10.2.4 (Path Problems and Kleene Closures), 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.

F41: In more detail, assuming that X is decomposed in sub-matrices A, B, C, D as explained in Section 10.2.4 (Path Problems and Kleene Closures), 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 10.2.4 (Matrix Data Structures):

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

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

F42: 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`.

F43: Insertions around V_2 are performed symmetrically, while deletions are supported via `Reset` operations on each polynomial in the recursive decomposition.

F44: Finally, P, E_1, and H_2 are updated recursively. The interested reader can find the low-level details of the method in [DeIt00].

10.2.6 Dynamic 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:

DEFINITIONS

D32: `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);

D33: `Query`(x, y): returns the distance from vertex x to vertex y in G, or $+\infty$ if no path between them exists;

REMARKS

R62: The dynamic maintenance of shortest paths has a remarkably long history, as the first papers date back to 35 years ago [Lo67,Mu67,Ro68]. After that, many dynamic shortest paths algorithms have been proposed (see, e.g., [EvGa85,FrMaNa98,FrMaNa00, RaRe96a,RaRe96b,Ro85]), but their running times in the worst case were comparable to recomputing APSP from scratch.

R63: The first dynamic shortest path algorithms which are provably faster than recomputing APSP from scratch, only worked on graphs with small integer weights.

R64: In particular, Ausiello *et al.* [AuItMaNa91] 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.

R65: Henzinger *et al.* [HeKiRaSu97] 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.

R66: This bound has been improved by Fakcharoemphol and Rao in [FaRa01], who 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.

R67: The first big step on general graphs and integer weights was made by King [Ki99], 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.

R68: Demetrescu and Italiano [DeIt01] 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.

R69: The same authors discovered the first algorithm that solves the fully dynamic all pairs shortest paths problem in its generality [DeIt03]. 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.

R70: 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.

R71: Other deletions-only algorithms for APSP, in the simpler case of unweighted graphs, are presented in [BaHaSe02].

King's $O(n^{2.5}\sqrt{C \log n})$ Update Algorithm

In this section we consider the dynamic shortest paths algorithm by King [Ki99].

REMARKS

R72: The algorithm is based on the long paths property discussed in Section 10.2.4 (Long Paths Property) and on the tree data structure of Section 10.2.4 (Tree Data Structures).

R73: Similarly to the transitive closure algorithms described in Section 10.2.5, 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.

APPROACH

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 [Ki99].

FACTS

F45: To maintain shortest paths up to distance d, similarly to the transitive closure algorithm by King described in Section 10.2.5, 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 Section 10.2.4 (Tree Data Structures). 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 $O(n^2 d)$ per operation.

F46: To maintain shortest paths longer than d, the algorithm exploits the long paths property of Fact F30: 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.

F47: Choosing $d = \sqrt{n \log n}$ yields an $O(n^{2.5}\sqrt{\log n})$ amortized update time. As mentioned in Section 10.2.4 (Long Paths Property), since H can be computed deterministically, the algorithm can be derandomized. The interested reader can find further details on the algorithm in [Ki99].

Demetrescu and Italiano's $O(n^2 \log^2 n)$ Update Algorithm

In this section we address the algorithm by Demetrescu and Italiano [DeIt03], 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.

REMARK

R74: The algorithm is based on the notions of uniform path, potentially uniform path, and historical shortest paths in a graph subject to a sequence of updates, as discussed in Section 10.2.4 (Uniform Paths).

APPROACH

The main idea is to maintain dynamically the potentially uniform paths of the graph in a data structure. Since by Fact F28 shortest paths are potentially uniform, this guarantees that information about shortest paths is maintained as well.

FACTS

F48: To support an edge weight update operation, the algorithm implements the smoothing strategy mentioned in Section 10.2.4 (Uniform Paths) 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 [Di59], 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.

F49: The update algorithm spends constant time for each of the $O(zn^2)$ new potentially uniform path (see Fact F29). 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 $O(n^2 \log^2 n)$ amortized time per update. The interested reader can find further details about the algorithm in [DeIt03].

RESEARCH ISSUES

In this work 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 [HoDeTh01] 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 [EpGaItNi97] described in Section 10.2.1.

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 that the static is known. Furthermore, if one is willing to pay more for queries, Demetrescu and Italiano [DeIt00] 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.

10.2.7 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).

10.2.8 Acknowledgments

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

[AlHoDeTh97] 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.

[AlHoTh00] 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.

[AuItMaNa91] 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.

[BaHaSe02] 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.

[Ch79] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[CoWi90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9, 1990.

[CoLeRiSt01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, Second Edition. The MIT Press, 2001.

[DeIt00] 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.

[DeIt01] 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.

[DeIt03] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. To appear in *Proc. 35th Symp. on Theory of Computing* (STOC'03), 2003.

[Di59] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[EpGaItNi97] 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.

[EvGa85] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

[EvSh81] S. Even and Y. Shiloach. An on-line edge deletion problem. *J. Assoc. Comput. Mach.*, 28:1–4, 1981.

[FaRa01] J. Fakcharoemphol 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.

[Fr85] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.

[Fr97] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2): 484–538, 1997.

[FrMaNa98] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single source shortest paths trees. *Algorithmica*, 22(3):250–274, 1998.

[FrMaNa00] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:251–281, 2000.

[GaIt92] Z. Galil and G. F. Italiano. Fully-dynamic algorithms for 2-edge connectivity. *SIAM J. Comput.*, 21:1047–1069, 1992.

[GrKn82] D. H. Greene and D.E. Knuth. *Mathematics for the analysis of algorithms.* Birkhäuser, 1982.

[Ha69] F. Harary. *Graph Theory.* Addison-Wesley, Reading, MA, 1969.

[HeKi97] 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.

[HeKi99] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. Assoc. Comput. Mach.*, 46(4):502–536, 1999.

[HeKiRaSu97] 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.

[HoDeTh01] 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.

[KhMoWi96] S. Khanna, R. Motwani, and R. H. Wilson. On certificates and lookahead on dynamic graph problems. In *Algorithmica* 21(4): 377-394 (1998).

[Ki99] 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.

[KiSa99] 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.

[Lo67] P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pages 96–109, 1967.

[Lo75] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[Mu71] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[Mu67] 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.

[RaRe96a] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest path problem. *Journal of Algorithms*, 21:267–305, 1996.

[RaRe96b] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

[Ra95] M. Rauch. Fully dynamic biconnectivity in graphs. In *Algorithmica*, 13:503-538, 1995.

[Ro68] V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Math. and Math. Phys.*, 8(5):336–343, 1968.

[Ro85] 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.

[SlTa83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comp. Syst. Sci.*, 24:362–381, 1983.

[UlYa91] J.D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[Zw98] 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.

GLOSSARY

Certificate: 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.

Clustering: Technique used in the design of dynamic algorithms based on partitioning the graph into a suitable collection of connected subgraphs, called clusters, such that each update involves only a small number of such clusters.

ET-Tree: Dynamic balanced binary tree over some Euler tour around another tree.

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

Historical Shortest Path – of a graph G: Path that has been a shortest path at some point during a sequence of updates of the graph, and such that none of its edges have been updated since then.

In-tree: Tree representing paths of a graph that lead to a given vertex.

Out-tree: Tree representing paths of a graph that originate from a given vertex.

Partially Dynamic Graph Problem: Problem where the update operations include either edge insertions/weight-decreases (*incremental*) or edge deletions/weight-increases (*decremental*).

Potentially Uniform Path – of a graph G: Path such that every proper subpath is a historical shortest path.

Sparsification: Technique for speeding up dynamic graph algorithms, which when applicable transforms 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.

Top Tree: Tree that describes a hierarchical partition of the edges of another tree, well suited to maintaining path information.

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

Uniform Path – of a graph G: Path such that every proper subpath is a shortest path.