Maintaining dynamic graph properties deterministically

Moreno Marzolla
Università "Ca' Foscari" di Venezia , Venezia, Italy
E-mail: mmarzoll@dsi.unive.it

Abstract

In this paper we present deterministic fully dynamic algorithms for maintaining several properties on undirected graphs subject to edge insertions and deletions, in polylogarithmic time per operation. Combining techniques from [6, 10], we can maintain a minimum spanning forest of a graph with k different edge weights in $O(k \log^2 n)$ amortized time per update; maintain an $1+\epsilon$-approximation of the minimum spanning forest in $O(\log^2 n \log(U(1+\epsilon))/\log(1+\epsilon))$ amortized time per update, where edge weights are between 1 and U; test if a graph is bipartite in $O(1)$ worst-case time, supporting updates in $O(\log^2 n)$ amortized time; test if the removal of k given edges disconnect the graph (*k-edge witness problem*) in $O(k \log^2 n)$ amortized time, supporting updates in $O(\log^2 n)$ amortized time; maintain a maximal spanning forest decomposition of order k in $O(k \log^2 n)$ amortized time per update. For all these problems, our algorithms match the previous best randomized bounds, and improve substantially over the best deterministic bounds.

1 Introduction

Graph algorithms are fundamental tools in many areas of computer science, including communication networks, development of optimizing compilers, VLSI design, and planning. In recent years there has been a growing interest in *dynamic* graph algorithms, that is, algorithms that maintain some properties on graphs subject to edge (or vertex) insertions or deletions more efficiently than recomputing them from scratch after each change. This interest is motivated by theoretical as well as practical reasons [15]. If only edge insertions (resp.

deletions) are allowed, the algorithm is said *incremental* (resp. *decremental*); if both are allowed, the algorithm is *fully dynamic*. In this paper, we will focus on *edge-dynamic* graph algorithms, in which a given property is maintained on a graph subject to the following operations: (1) insertion of an edge e; (2) deletion of an edge e, and (3) test if the graph fulfills the property.

In the last years, a lot of work has been done in fully-dynamic graph algorithms for undirected graphs [1, 3, 6, 7, 8, 9, 10]. In [6] was described a randomized connectivity algorithm for maintaining a spanning forest of a dynamically changing graph in polylogarithmic expected time per update. In the same paper were also presented randomized algorithms for maintaining dynamic graph properties in polylogarithmic (expected) time per update, by reduction to the dynamic connectivity problem. These reductions were quite general, and independent from the connectivity algorithm. In [9], the same reductions were applied with an improved randomized connectivity algorithm, strengthening the previous bounds by a logarithmic factor. Very recently, Holm, de Lichtenberg and Thorup [10] gave a deterministic algorithm for maintaining a spanning forest of a dynamic graph in $O(\log^2 n)$ amortized time per update, matching the previous best randomized bound and improving substantially over the previous best deterministic bound of $O(n^{1/3} \log n)$ [7].

In this paper, the general techniques from Henzinger and King [6] are applied together with the deterministic dynamic connectivity algorithm of Holm et. al. [10], resulting in deterministic algorithms for maintaining the following properties on dynamic graphs with n vertices: k-weight Minimum Spanning Forest in $O(k \log^2 n)$[1] amortized time per update; $1+\epsilon$-approximate Minimum Spanning Forest in $O(\log^2 n \log(U(1+\epsilon))/\log(1+\epsilon))$ amortized time per update; bipartiteness in $O(\log^2 n)$ amortized time per update, and $O(1)$ worst-case time per query, improving the previous best deterministic bound of $O(n^{1/3} \log n)$ per update [7]; k-edge witness problem in $O(\log^2 n)$ amortized time per update, and $O(k \log^2 n)$ amortized time per query; the previous best deterministic bounds were $O(n^{1/3} \log n)$ amortized time per update and $O(kn^{1/3} \log n)$ amortized time per query [7]; cycle-equivalence in $O(\log^2 n)$ amortized time per update and query, improving the previous best deterministic bound of $O(n^{1/3} \log n)$ amortized time per update and query [7]; maximal spanning forest decomposition of order k in $O(k \log^2 n)$ amortized time per update, improving the previous best deterministic bound of $O(kn^{1/3} \log n)$ amortized

[1] Unless otherwise stated, all logarithms are base 2, and we let $\log x = \max\{1, \log_2 x\}$, so $\log x$ is never less than 1

time per update [7]. All these results match the best known randomized bounds. For graphs with n vertices and m edges, the algorithms use $O(m + n \log n)$ space. Note that our algorithm for maintaining an $1+\epsilon$-approximate Minimum Spanning Forest is, to our best knowledge, the first deterministic algorithm for solving this specific problem.

This work is based on [13].

2 Preliminaries and Notations

We define a graph property as follows: let $G = (V, E)$ be an undirected graph; a *property* $\mathcal{P}$ on G is a function which either:

1. maps $G = (V, E)$ to *true* or *false* (e.g.: $\mathcal{P}(G)$ = true iff G is connected), or
2. maps the tuple (G, v, w) to *true* or *false* for each $u, v \in V$ (e.g.: $\mathcal{P}(G, v, w)$ = true iff u and v are connected in G), or
3. maps G into a subgraph G' (e.g.: $\mathcal{P}(G) = F$, where F is a spanning forest of G).

If G is an undirected graph, we let $V(G)$ be the vertex set of G, and $E(G)$ the edge set of G. A graph G' is a *subgraph* of G, indicated as $G' \subseteq G$, if $E(G') \subseteq E(G)$ and $V(G') \subseteq V(G)$. A *spanning forest* F of G is an acyclic subgraph of G with the same connected components as G. The edges in $E(F)$ will be referred to as *tree-edges*, and those in $E(G) \setminus E(F)$ as *nontree-edges*. If G is a graph with weight function $w : E \to \mathbf{R}$, a *Minimum Spanning Forest* (in short, MSF) for G is a spanning forest F with minimum total weight $w(F) = \sum_{e \in E(F)} w(e)$. In the static case, a MSF can be computed applying any combination of the following properties [17]:

Cut Property The MSF is constructed one edge at a time. At each step, find a cut in G which contains no edges in the current forest, and add the edge with minimum weight crossing the cut.

Cycle Property The MSF is constructed removing one edge at a time from G. At each step, find a cycle in the remaining graph, and remove the edge in the cycle with maximum weight.

These rules applies also to the dynamic case, and will be exploited in section 4.1.

3 Deterministic connectivity algorithm

One of the main components of our algorithms is the deterministic dynamic connectivity algorithm developed by Holm, de Lichtenberg

and Thorup [10]. This algorithm can maintain a spanning forest of a graph, subject to edge insertions and deletions, in polylogarithmic time per update; it is deterministic, and uses only simple data structures. The result of Holm et al. can be stated by the following

Theorem 1 (Holm, de Lichtenberg and Thorup [10]) *Given a graph G with m edges and n vertices, there exists a deterministic fully dynamic algorithm that answers connectivity queries in $O(\log n / \log\log n)$ time worst case, and uses $O(\log^2 n)$ amortized time per insertion or deletion of edges. The space required is $O(m + n \log n)$.*

We refer the interested reader to the given reference for details.

4 Deterministic Algorithms for Dynamic Graph Problems

As shown in [6], many dynamic graph problems can be solved by reduction to the connectivity problem; in this section we exploit some of these reductions to present deterministic algorithms for maintaining properties on dynamic graphs.

4.1 k-weight Minimum Spanning Forest

In the k-weight Minimum Spanning Forest problem, we are given a graph $G = (V, E)$ with at most k different edge weights at any time, for a fixed $k \geq 1$. We are asked to maintain a spanning forest of minimum weight as edges are inserted and deleted from G.

Instead of using directly the reduction from [6], it is preferable to consider a simplified version of the problem, in which the weights are limited to the set $\{1, 2, \ldots, k\}$. Let F be a minimum spanning forest of G, and for every $e \in E(G)$, let $w(e)$ denote the weight of e (here we are assuming $w(e) \in \{1, 2, \ldots, k\}$). For $i = 1, 2, \ldots, k$, we let $E_i = \{e \in E(G) \mid w(e) = i\} \cup E(F)$, and let $G_i = (V, E_i)$. After each update, the invariant that $E(G_i) = \{\text{edges of weight } i\} \cup E(F)$ is maintained.

For every i, we can maintain a spanning forest F_i of G_i using the dynamic connectivity algorithm (note that, in general $F_i \neq F$); F_i and the minimum spanning forest F are also stored in dynamic trees [16]. Now we describe how to deal with insertions and deletions of edges.

To insert an edge $\{u, v\}$: If u and v are not connected in F, the new spanning forest is $F \cup \{u, v\}$; to preserve the invariant, then new edge is inserted in G_i, for each $i = 1, 2, \ldots, k$.

If u and v are connected in F, then using the dynamic tree for F, we can find the maximum weight edge e on the path from u to v in F. If the weight of e is greater than the weight of $\{u, v\}$, we can replace e by $\{u, v\}$ in F, updating the E_i to reflect the changes in F; otherwise, $\{u, v\}$ is just added to $G_{w\{u,v\}}$.

To delete an edge $\{u, v\}$: Remove $\{u, v\}$ from all data structures containing it (updating the local spanning forests). If $\{u, v\}$ was in F, the tree T containing it is split in two subtrees T_u and T_v. In that case, we first, we find the minimum l such that u and v are connected in G_l. Then, we can find a replacement edge, if one exists, performing a binary search on the path connecting u and v in F_l. Let x be a midpoint of the path: if u and x are connected in F, then recurse on the path from x to v, else recurse on the path from u to x, until the edge f of F_l crossing the cut in T is found. Then f is inserted into F and into every G_i, $i = 1, 2, \ldots, k$.

Correctness: We start by observing that the above algorithm maintains the invariant: for $i = 1, 2, \ldots, k$, $E_i = \{\text{edges of weight } i\} \cup E(F)$. When an edge is inserted, we can decide if and how F changes using the dynamic tree representing it. When an edge $\{u, v\}$ is deleted, u and v are connected in F_j iff there is a replacement edge of weight j. The minimum weight replacement edge can be found using a binary search on the path connecting u and v in F_j.

Running time: After an edge insertion, we can compute in $O(\log n)$ time how the MSF changes, using the dynamic tree representing it. After that, the data structures can be updated in $O(k \log^2 n)$ amortized time. The amortized complexity of an edge insertion is thus $O(k \log^2 n)$.
After the deletion of an edge $\{u, v\}$, we may have to update all G_i, $i = 1, 2, \ldots, k$, for a total cost of $O(k \log^2 n)$. Then, it takes $O(k \log n / \log \log n)$ time to find the minimum l such that u and v are connected in F_l (remember that a connectivity query on a graph G_i takes $O(\log n / \log \log n)$ worst-case time). The midpoint of the tree path between u and v in F_l can be found in $O(\log n)$ time using the dynamic tree representation of F_l (this is not a standard dynamic tree operation, but can be implemented easily [8]); the algorithm recurses at most $\log n$ times to determine the replacement edge for $\{u, v\}$, for a total cost of $O(\log^2 n)$. The amortized complexity of an edge deletion is thus $O(k \log^2 n)$.

Generalization: The above algorithm, with few changes, can be generalized for the case in which the graph G has at most k arbitrary different edge weights at any time (thus obtaining the reduction in [6]). Suppose we start with a graph G with $l \leq k$ different edge weights

$w_1, w_2, \ldots, w_l$. After computing a minimum spanning forest F, we let $E_i = \{\text{edges with weight of rank } i\} \cup E(F)$ for $i = 1, 2, \ldots, k$, and we let $G_i = (V, E_i)$; if there are $l < k$ different edge weights, $l - k$ subgraphs will be coincident with the MSF F. Adopting the same terminology used in [6], we call these subgraphs *extras*. The subgraphs G_i are kept ordered according to the weight of their edge set and stored in a balanced binary tree.
When an edge e with a new weight is inserted into G, then a new subgraph is created by adding e to an extra, and inserting that extra into the ordering of the other G_i. Symmetrically, when e is the only edge of its weight and is removed from G, then the subgraph containing it becomes an extra. These two operations can be performed in $O(\log k)$ worst-case time, so they have no impact on the overall running time of insertions and deletions of edges in G. Note that every update to F causes an update to every G_i (including every extra). Note also that, after the deletion of a tree-edge $\{u, v\}$, we can find the replacement edge, if any, performing a binary search on the sequence of graphs, rather than a linear search. So, we can maintain a MSF of a graph with at most k arbitrarily different edge weights, supporting insertions and deletions of edges in $O(k \log^2 n)$ amortized time per update.

4.2 1+ϵ-Approximate Minimum Spanning Forest

Definition 1 *Let G be an undirected graph with weight function $w : E \to [1, U]$, for a given $U > 1$. Let F_m be a minimum spanning forest for G. A spanning forest F is a 1+ϵ-approximate minimum spanning forest, for a given $\epsilon > 0$, if*

$$w(F) \leq (1 + \epsilon) w(F_m)$$

The problem of maintaining an 1+ϵ-approximation of a MSF can be reduced to the k-weight MSF. We present a reworked version of the reduction presented in [6], with a complete proof of correctness.

Theorem 2 *Let $G = (V, E)$ be an undirected graph with weight function $w : E \to [1, U]$, for some constant $U > 1$. Given $\epsilon > 0$, let $N = \lfloor \log U / \log(1 + \epsilon) \rfloor$. Define the function $h(x) : [1, U] \to \{0, 1, \ldots, N\}$ as*

$$h(x) = \lfloor \log_{(1+\epsilon)} x \rfloor$$

If F is a MSF for G w.r.t. the weight function $h \circ w$ (where $(h \circ w)(x)$ is, by definition, $h(w(x))$), then F is an 1+ϵ-approximate MSF for G w.r.t. the weight function w.

Proof. If $F = F_m$, the theorem follows. Suppose $F \neq F_m$; we will prove the theorem by transforming F_m into F, replacing one edge of $E(F_m) \setminus E(F)$ with one of $E(F) \setminus E(F_m)$ at a time. We call the edges in $E(F) \setminus E(F_m)$ *white edges*, those in $E(F_m) \setminus E(F)$ *black edges* and those in $E(F) \cap E(F_m)$ *grey edges*. So, edges in F are white or grey, and those in F_m are black or grey.

The idea behind the transformation is to replace every black edge in F_m with a white edge, until no more black edges exist. The following algorithm compute the transformation.

Require: F is a MSF w.r.t. $h \circ w$, F_m is a MSF w.r.t w.
Ensure: $F^* = F$
begin
    $F^* := F_m$
    for all white edges $e \in E(F) \setminus E(F_m)$
(*) **Let** f be the heaviest black edge, w.r.t $h \circ w$,
        on the cycle induced by e in F^*
(**) $F^* := F^* \setminus \{f\} \cup \{e\}$
    end for
end

The algorithm maintains a spanning forest F^* of the graph, starting with $F^* = F_m$; at each step, it finds a black edge in F^* which then replaces with a white edge not already in F^*, until no more black edges exist. At that point, $F^* = F$.

In the following, we will use the following notation: given a set of edges S, an edge $e \in S$ is *w-heaviest* if it is the heaviest edge in S with respect to the weight function w. e is *$h \circ w$-heaviest* if it is the heaviest edge in S w.r.t the weight function $h \circ w$.

Observe the following facts:

Fact 1 On line (*) we have to select the $h \circ w$-heaviest black edge in the cycle C induced by the insertion of e in F^* (such a cycle exists because F^* is a spanning forest of G, and e is an edge of G not already in F^*). Note that C must contain at least one black edge; if it did not, all edges in C would be white or grey, that is, all edges in C would be in $E(F)$, but this is impossible because F is a forest.

The black edge f has the property that, for every edge $e \in C$, $h(w(f)) \geq h(w(e))$, because the $h \circ w$-heaviest edge in C can't be white or gray.

Fact 2 $w(e) - w(f) \leq \epsilon w(f)$, where e, f are the edges considered on line (**) of the algorithm. That is because f is the $h \circ w$-heaviest edge in C; so, for every $x \in C$

$$h(w(f)) \geq h(w(x)) \tag{1}$$

But f is a black edge, so it can't be the w-heaviest edge in C; there exists a white edge $g \in C$ such that, for every $y \in C$

$$w(g) \geq w(y) \tag{2}$$

Observe that h is monotone, so from (2) we can deduce that $h(w(g)) \geq h(w(y))$. Substituting g for y in (1), and f for x in (2), we obtain

$$h(w(g)) = h(w(f))$$

which immediately implies $w(g) - w(f) \leq \epsilon w(f)$, and because $w(e) \leq w(g)$ (by (2)), we conclude

$$w(e) - w(f) \leq w(g) - w(f) \leq \epsilon w(f) \tag{3}$$

The algorithm implicitly defines a one-to-one mapping $\phi : E(F) \rightarrow E(F_m)$ as:

$$\phi(e) = \begin{cases} e & \text{if } e \text{ is a grey edge (and thus never replaced in } F^* \\ f & \text{if } f \text{ is replaced by } e \text{ in } F^* \text{ on line (**)} \end{cases}$$

The function ϕ has the property that $w(e) - w(\phi(e)) \leq \epsilon w(\phi(e))$ (see (3)), so

$$\begin{aligned} w(F) - w(F_m) &= \textstyle\sum_{e \in E(F)} w(e) - \sum_{e \in E(F_m)} w(e) \\ &= \textstyle\sum_{e \in E(F)} (w(e) - w(\phi(e)) \\ &\leq \epsilon \textstyle\sum_{e \in E(F)} w(\phi(e)) = \epsilon w(F_m) \end{aligned} \tag{4}$$

which concludes the proof. □

4.3 Bipartiteness

A graph $G = (V, E)$ is *bipartite* iff its vertex set V can be partitioned in two nonempty, disjoint subsets V_1, V_2, such that every edge $e \in E$ has exactly one endpoint in V_1 and the other in V_2. In the fully dynamic bipartiteness problem we have to answer the query "is G bipartite?", as edges are inserted and deleted.

As shown in [6], this problem can be reduced to the 2-weight minimum spanning forest problem: a graph is bipartite iff, given any spanning forest F, each nontree-edge forms a cycle of even length with F.

So, first we compute a spanning forest F of G; the edges in F, and those who induce an even cycle with F, are given weight 0 and are called *even edges*. The remaining edges, called *odd edges*, are given weight 1.

When an edge e is inserted, we can compute the parity of e using the dynamic tree representation of F; e is then inserted into the graph as in the 2-weight MSF problem.

When an edge e is deleted, if it is a nontree-edge, it is simply removed. If e is a tree-edge, and is replaced by an odd edge, then the parity of all edges crossing the cut in F have to change (see [13] for a proof); to do so, we delete the replacement edge and search for the next replacement, delete it and so on until no more replacements exist. Then, we give all replacement edges weight 0, and reinsert them back into G.

Correctness: When an edge is inserted, we can determine its parity using the dynamic trees representing F. When an edge is deleted, we replace it with an even edge if possible; if this is not possible, then the parity of all the edges crossing the cut have to be changed. We can do that deleting all replacement edges, and reinserting them with weight 0.

Running Time: For the analysis of the running time, note that an even edge never becomes odd. Every edge can be inserted into the data structure at most twice, once with weight 1 and once with weight 0. Thus, an edge insertion or deletion can be performed in $O(\log^2 n)$ amortized time.

A faster algorithm for planar graphs: It turns out that if we can maintain a minimum spanning forest of a (two) weighted graph G with n vertices, supporting edge insertions and deletions in time $O(f(n))$, then there exists an algorithm for testing if the graph is bipartite in $O(1)$ time, supporting updates in $O(f(n) + \log n)$ time per operation. For the general case described before, we had $f(n) = \log^2 n$, so we can perform insertions and deletions of edges in $O(\log^2 n)$ amortized time per operation. But for plane graphs, the deterministic algorithm by Eppstein et al. [2] can maintain a minimum spanning forest in $O(\log n)$ time per update. So, applying the same reduction, we can test if a plane graph is bipartite in $O(1)$ worst-case time, supporting updates in $O(\log n)$ time per operation.

4.4 k-edge Witness Problem

The dynamic connectivity algorithm can be used to solve the k-edge Witness Problem: for a given graph G, does the removal of k given

edges $e_1, e_2, \ldots, e_k$ disconnect two vertices u and v? If the graph G is kept in a fully dynamic connectivity data structure, to solve this problem:

1. Remove the edges $e_1, e_2, \ldots, e_k$ in $O(k \log^2 n)$ time;
2. Test if u and v are connected, in $O(\log n / \log \log n)$ worst-case time;
3. Reinsert the edges $e_1, e_2, \ldots, e_k$ in $O(k \log^2 n)$ time.

So, given a graph G with n vertices, we can test if the removal of k given edges disconnects two vertices u and v in $O(k \log^2 n)$ amortized time, supporting insertions and deletions of edges in $O(\log^2 n)$ amortized time.

4.5 Cycle-equivalence

Definition 2 *Let $G = (V, E)$ be an undirected graph. Two edges e_1 and e_2 are cycle-equivalent iff every cycle containing e_1 also contains e_2.*

As pointed out by Henzinger [5], computing cycle-equivalence is useful in many compilation problems (see for example [4, 11, 12]). The cycle-equivalence problem can be reduced to the 2-edge witness problem by the following

Lemma 1 (Henzinger [5]) *Let G be an undirected graph. Two edges $e_1 = \{x, y\}$ and $e_2 = \{u, v\}$ are cycle-equivalent iff x, y are disconnected in $G \setminus \{e_1, e_2\}$, and u, v are disconnected in $G \setminus \{e_1, e_2\}$.*

Thus, given a graph G with n vertices, we can test if two edges are cycle-equivalent in $O(\log^2 n)$ amortized time, supporting edge insertions and deletions in $O(\log^2 n)$ amortized time per operation.

4.6 Maximal Spanning Forest Decomposition

Given a graph G with n vertices, and an integer $k \geq 1$, a *maximal spanning forest decomposition* of order k for G is a sequence of forests $F_1, F_2, \ldots, F_k$ such that F_i is a spanning forest of the graph $G_i = G \setminus \cup_{j<i} F_j$. This decomposition is important because the graph $\cup_i F_i$ has $O(kn)$ edges and has the same k-edge connected components as G [14]. We can maintain each graph G_i and each spanning forest F_i using a dynamic connectivity data structure.

When an edge e is inserted, it is inserted in every G_i, starting from G_1, updating the local spanning forest F_i. If e is inserted into F_i, then for $j > i$, e can't appear in G_j, so the procedure stops; otherwise, e is inserted into G_{i+1}.

When an edge e is deleted, it is removed from all structures containing it, starting from G_1. At each step, if e was in F_i, then it did not appear in $G_j, j > i$, so the procedure stops; otherwise, e is removed from G_{i+1}.

5 Conclusions

In this paper we have presented deterministic fully dynamic algorithms for maintaining special types of Minimum Spanning Forests, and for connectivity-related problems on graphs. The algorithms are obtained combining a recent result by Holm et al. [10], with general reductions by Henzinger and King [7]. This gives the first deterministic algorithms for maintaining these properties in polylogarithmic time per update. Also, only simple data structures are required, so the algorithms may be easily implemented in practice.

Of course, there is still room for further work on this topic: primarily, other problems could be solved by reduction to the dynamic connectivity problem. Developing the right reduction could yield to faster algorithms.

Also, it should be noted that the reductions used here are fairly independent from the underlying connectivity algorithm. However, these reductions introduce an overhead which we could not afford if faster connectivity algorithms were discovered. In particular, in the k-weight MSF algorithm it takes $O(\log^2 n)$ time to find the correct replacement after an edge deletion; do there exist more efficient reductions which could be applied on top of faster dynamic connectivity algorithms?

Acknowledgments

I am grateful to Giuseppe F. Italiano for his useful comments and suggestions on this paper.

References

[1] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - A technique for speeding up dynamic

graph algorithms. *Journal of ACM*, 44(1):669–696, 1997.

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

[3] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[4] R. Gupta and M. L. Soffa. Region scheduling. In *Proc. 2nd International Conference on Supercomputing*, pages 141–148, 1987.

[5] M. Rauch Henzinger. Fully dynamic cycle-equivalence in graphs. In Shafi Goldwasser, editor, *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 744–757, Los Alamitos, CA, USA, November 1994. IEEE Computer Society Press.

[6] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 519–527, Las Vegas, Nevada, 29 May–1 June 1995.

[7] Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning trees in dynamic graphs. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium*, volume 1256 of *Lecture Notes in Computer Science*, pages 594–604, Bologna, Italy, 7–11 July 1997. Springer-Verlag.

[8] Monika Rauch Henzinger and Valerie King. Fully Dynamic 2-Edge Connectivity Algorithm in Polylogarithmic Time per Operation. Technical Note 1997-014a, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, July 1997.

[9] Monika Rauch Henzinger and Mikkel Thorup. Improved sampling with applications to dynamic graph algorithms. In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium*, volume 1099 of *Lecture Notes in Computer Science*, pages 290–299, Paderborn, Germany, 8–12 July 1996. Springer-Verlag.

[10] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Polylogarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC'98)*, Dallas, Texas, 24 May–26 May 1998.

[11] R. Johnson, D. Pearson, and K. Pingali. Finding regions fast: Single entry single exit and control regions in linear time. In *Proc. Sigplan'94 PLDI*, 1994.

[12] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proc. Sigplan'93 PLDI*, pages 78–89, 1993. Published as ACM SIGPLAN Notices 28(6).

[13] Moreno Marzolla. Algoritmi deterministici per il mantenimento di proprietà su grafi dinamici. Master's thesis, Università degli Studi "Ca' Foscari" di Venezia, Facoltà di Scienze Matematiche, Fisiche e Naturali, Corso di Laurea in Scienze dell'Informazione, March 1998.

[14] N. Nagamochi and T. Ibaraki. Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.

[15] G. Ramalingam. *Bounded incremental computation*, volume 1089 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[16] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26:362–390, 1983.

[17] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.