Updating Directed Minimum Cost Spanning Trees

Gerasimos G. Pollatos*, Orestis A. Telelis**, and Vassilis Zissimopoulos

Dept. of Informatics and Telecommunications,
University of Athens, Hellas, Greece
{gpol, telelis, vassilis}@di.uoa.gr

Abstract. We consider the problem of updating a directed minimum cost spanning tree (DMST), when edges are deleted from or inserted to a weighted directed graph. This problem apart from being a classic for directed graphs, is to the best of our knowledge a wide open aspect for the field of dynamic graph algorithms. Our contributions include results on the hardness of updates, a dynamic algorithm for updating a DMST, and detailed experimental analysis of the proposed algorithm exhibiting a speedup factor of at least 2 in comparison with the static practice.

Keywords: branchings, dynamic graph algorithms, data structures.

1 Introduction

We study the problem of updating a *directed minimum spanning tree* (DMST) efficiently when a directed edge is inserted to or deleted from a weighted digraph. On a digraph $G(V, E)$, of $|V| = n$ vertices and $|E| = m$ edges, each associated with a non-negative cost $c(e)$, a DMST is defined as *a maximal acyclic subset of edges, such that no vertex of the digraph has more than one incoming edge in this set, and the total edge cost is minimum.* If G is strongly connected this definition implies indeed a directed tree (also called arborescence) blossoming out of its root, otherwise it may be a collection of trees (also called a *branching* [1]). Since G can always be made strongly connected by the addition of at most $O(n)$ edges, we can assume a directed tree. Applications of DMST updates range from wireless networks [2, 3] to hardware design [4, 5].

An identical polynomial time algorithm was described for this problem in [1, 6, 7]. For the rest of the discussion we refer to this algorithm as Edmonds' algorithm [1]. Tarjan [8] gave an implementation of $O(\min\{m \log n, n^2\})$ time. Gabow et al. [9] improved the running time to $O(m + n \log n)$ by using a special implementation of Fibonacci heaps. Improved heaps in [10] yielded deterministic $O(m \log \log n)$ and randomized $O(m\sqrt{\log \log n})$ time.

* Author partially supported by the programme $\Pi ENE\Delta 2003$ of the Greek General Secretariat of Research and Technology.

** Author partially supported by the Greek Ministry of Education under the project PYTHAGORAS II.

C. Álvarez and M. Serna (Eds.): WEA 2006, LNCS 4007, pp. 291–302, 2006.

To the best of our knowledge, the *dynamic* DMST problem is a wide open aspect for the area of *dynamic graph algorithms* [11], in contrast to the near optimal achievements seen for the minimum spanning tree in undirected graphs [12]. A *fully* dynamic graph algorithm maintains efficiently a solution to a graph problem when edges are deleted from or inserted to the underlying graph in time less than the time required for re-evaluating a solution from scratch.

Our contributions include a hardness result regarding the complexity of dynamic DMST updates (section 3), the design of a fully dynamic algorithm and its analysis in the *output complexity* model (sections 4-5), and extended experimental investigation of the proposed algorithm (section 6), revealing a speedup factor of at least 2 in comparison with the static re-evaluation practice. In the output complexity model the complexity of a dynamic algorithm is measured with respect to a minimal subset of the previous output that needs to be updated [13, 14, 15, 16].

2 Preliminaries

From now on we assume as input a strongly connected digraph $G(V, E)$, with edge costs $c(e) \geq 0$. If the input digraph is not strongly connected, we add a vertex v_∞ and $2n$ edges of infinite (very large) weight, (v_∞, v_i) and (v_i, v_∞) for each $v_i \in V$, so as to make it strongly connected. These edges will never be affected by dynamic edge operations, so that strong connectivity of the underlying digraph is always assured. For each directed edge $e = (u, v) \in E$, we refer to $t(e) = u$ as the tail vertex of e, and $h(e) = v$ as its head vertex. For $S \subset V$, let $\delta_E(S) = \{e \in E | h(e) \in S, t(e) \in V - S\}$ be the "in" cut-set of S w.r.t. E. The algorithm of Edmonds greedily produces an edge set $H \subseteq E$ and prunes it to obtain the DMST T:

1. set $H = \emptyset$
2. set $\hat{c}(e) = c(e)$ for every edge e
3. while there are more than one vertices, pick a vertex v
   (a) let $e^\star$ be the incoming edge of v with the minimum $\hat{c}(e)$
   (b) set $\hat{c}(e) = \hat{c}(e) - \hat{c}(e^\star)$ for every incoming edge of v
   (c) insert edge $e^\star$ in H
   (d) if a directed cycle occurred, *contract* the cycle into a single vertex.
4. create T from H by removing redundant edges.

The loop (lines (a)-(d)) creates an edge set $H \subseteq E$, and the final DMST T is produced from H, by removal of redundant edges. This removal can be performed in $O(n)$ time [8], thus making the loop a complexity bottleneck for the algorithm. In a strongly connected digraph the algorithm will eventually contract the vertex set into a single vertex. At most $n - 1$ contractions will take place, since each contraction absorbs at least one of the original digraph's vertices. In the sequel we refer to vertices emerged by contraction as *c-vertices* and to vertices of the original digraph as *simple* vertices.

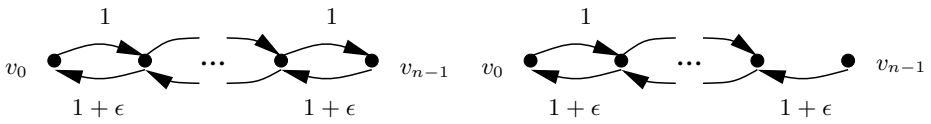


Fig. 1. Edges of cost 1 form the DMST for the left digraph. Deletion of (v_{n-2}, v_{n-1}) causes the DMST to change entirely (right) into a new one consisting of edges of cost $1+\epsilon$ because inclusion of any of the 1-weighted edges cannot yield a maximal edge set with DMST properties.

3 On the Hardness of Updates

We consider the hardness of DMST updates when the only information retained and used is the DMST itself. We use the framework presented in [17] which assumes that the unit operation of an algorithm is evaluation and positivity testing of an analytic function over the edge weights of the underlying digraph. Such an algorithm is called an *analytic tree program*. A lower bound on the verification complexity of a DMST is obtained:

Lemma 1. *Given a directed acyclic graph G of m edges with positive edge costs and a subset T of edges, an analytic tree program verifying that T is a DMST of G incurs $\Omega(m)$ complexity.*

Proof. A feasible tree (or a collection of trees - a branching) in a DAG, is any assignment of a unique incoming edge to each vertex. This can be checked in $O(n)$ time. The cost of T is minimum if and only if for every $e \notin T$ there is $e' \in T$ with $h(e) = h(e')$ and $c(e') \leq c(e)$, which translates to testing that each vertex is assigned its minimum cost incoming edge. This implies testing a set of $\Theta(m)$ inequalities for analytic functions of edge weights. A classic result of Rabin [18] states that *all* these inequalities must be evaluated in the worst case. □

The $\Omega(m)$ lower bound for verification holds for general digraphs in the worst case. This leads to the following result:

Theorem 1. *Dynamic maintenance of a DMST under edge deletions and/or insertions is as hard as recomputing a DMST from scratch if only the DMST information is retained and used between updates.*

Proof. Consider a digraph G of n vertices $v_0, \ldots, v_{n-1}$. Let edges (v_i, v_{i+1}), for $i = 0, \ldots, n-2$ have cost 1 and edges (v_i, v_{i-1}), $i = 1, \ldots, n-1$ have cost $1+\epsilon$ for some $\epsilon > 0$. Set all other edges to some cost $M > 1+\epsilon$. Then a DMST of this digraph is the *directed line* $\{(v_i, v_{i+1}) | i = 0 \ldots n-2\}$. Removal of edge (v_{n-2}, v_{n-1}) from this set, causes the DMST to change completely to another optimal set of edges $\{(v_i, v_{i-1}) | i = 1, \ldots, n-1\}$. Re-insertion of the removed edge causes the DMST to change entirely to its former state (see fig. 1 for an example). Every algorithm using only DMST information to update the DMST per edge operation requires at least the time given by lemma 1, which is $\Omega(n^2)$ for dense digraphs. □

A similar result was derived in [17] for shortest paths tree updates. In the next section we take the approach of maintaining intermediate information related to construction of a solution (also suggested in [17] and investigated later in [14] for shortest paths tree updates). Note that when the underlying digraph is restricted to remain a DAG in between edge operations, a simple application of Fibonacci heaps yields an $O(\log n)$ update time dynamic algorithm.

4 Dynamic Algorithm

The algorithm maintains as many contractions as possible per edge operation, along with the selected edges (edges of H). The purpose of this practice is to efficiently initialize and execute the implementation of Edmonds' algorithm known from [9] on a maintained partially contracted digraph, so as to process less vertices and edges per edge operation. We show that such a partially contracted digraph can be recognized in $O(n)$ time by using simple operations over an appropriate data structure, and a modified version of the implementation of [9].

4.1 An Augmented Structure

We present a data structure, namely the *Augmented Tree* (ATree), which appropriately encodes the redundant edge set H along with all vertices (c-vertices and simple ones) processed during execution of Edmonds' algorithm. Simple vertices are represented in the ATree by *simple nodes* while c-vertices are represented by *c-nodes*. For the rest we denote simple nodes with N_i^s, where $v_i \in V$ and c-nodes with N_j^c. We use unsuperscripted N to refer to ATree nodes regardless of their type. Six records are maintained at each node N of the ATree:

1. $e(N)$ is the edge selected by the algorithm for the represented vertex. If no edge was selected we set $e(N) = null$ and call N a *root node*. The root node will be unique as discussed below.
2. $y_N = \hat{c}(e)$ is the cost of edge $e(N)$ at the time it was selected for the vertex represented by N.
3. $children(N)$ is a list holding the children of N in the ATree.
4. $parent(N)$ is the parent node of N in the ATree (which equals to $null$ if N is the root node).
5. *contracted-edges*(N^c) is a list holding all edges contracted during creation of the corresponding c-vertex represented by N^c (that is, edges having both their end-vertices on the contracted cycle).
6. $kind(N)$ is the kind of node N (simple node or c-node).

Since the digraph is strongly connected, all vertices will be eventually contracted to a single c-vertex by the end of the algorithm's execution. This c-vertex is represented by the root node of the ATree. The parent of each other node N is the intermediate c-node N^c to which it was contracted. Since the parent of each node is unique, the described structure is indeed a tree.

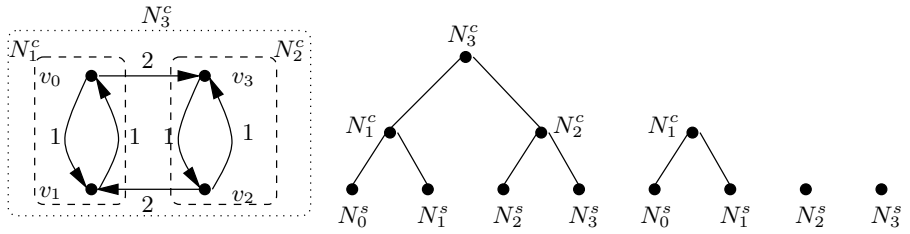


Fig. 2. Execution of Edmonds' algorithm on the digraph on the left performs contractions marked with dashed lines. The representative ATree appears in the middle. The decomposed ATree after deletion of edge (v_2, v_3) represents a partially contracted digraph of three vertices (on the right).

The ATree has at most $O(n)$ nodes because the algorithm handles $O(n)$ contractions. Construction of an ATree can be embedded into the implementation of [9], without affecting its complexity. However, maintenance of *contracted-edges* lists requires special manipulation with respect to the implementation of [9], and we defer this discussion to paragraph 4.5. Fig. 2 depicts an ATree example (middle) with respect to execution of Edmonds' algorithm on a digraph (left).

4.2 Deleting Edges

We discuss how to handle edge deletions using the ATree structure. Let $e_{out} \in E$ be an edge we want to remove from the digraph. Two cases must be considered:

1. $e_{out} \notin H$: we only need to remove e_{out} from the digraph and from the *contracted-edges* list to which e_{out} belongs. This can be achieved in $O(1)$ time, if we use an *endogenous* list implementation [9]: each edge has associated pointers in the digraph representation, pointing to the next and previous elements in the list.
2. $e_{out} \in H$, in which case we proceed by *decomposing* the ATree, initializing Edmonds' algorithm w.r.t. the remainders of the ATree and execute it.

Decomposition. The decomposition of the ATree begins from node N such that $e(N) = e_{out}$ and proceeds by following a path from N towards the ATree root and removing all c-nodes on this path except N. Each of the children of a removed c-node is made the root of its own subtree. By the end of this procedure, the initial structure has been decomposed into smaller ATrees, each corresponding to a contracted subset of the original digraph's vertices. Observe that all these ATrees remain intact after decomposition, because e_{out} was not part of their formation. An example of ATree decomposition is shown on the right of fig. 2.

4.3 Recognizing a Partially Contracted Digraph

Having performed the decomposition of the ATree, we proceed by recognizing the partially contracted digraph $G(V', E')$ represented by the remainders (namely

smaller ATrees). Let $V' = \{N_1 \dots N_k\}$ be the roots of ATrees after decomposition. These will constitute the vertex set of the digraph. A BFS on each tree suffices to assign each original digraph vertex v_i to some ATree root in V' in $O(n)$ time. Now we need to identify E' without scanning all edges of the original digraph. E' consists of edges having their end-vertices in different remaining ATrees. Let $R = \{N_1^c \dots N_r^c\}$ be the set of removed c-nodes during decomposition of the ATree. Note that the union of *contracted-edges*(N_i^c) lists, $N_i^c \in R$, is precisely the correct edge set E' and it can be found in $O(n)$ time.

Given the partially contracted digraph $G(V', E')$, each $N \in V'$ associated with a set of incoming edges $\delta_{E'}(N)$, a second aspect concerns consideration of the proper reduced costs $\hat{c}$ for these edges. Let $v_i \in V$ be a vertex of the original digraph represented as a leaf N_i^s of an ATree with root $N \in V'$ (it may occur that N_i^s is the root N itself). Let $e = (u, v_i)$ with $e \in \delta_E(v_i) \cap \delta_{E'}(N)$. Let $\mathcal{P} = [N_i, N_1^c, \dots N_l^c, N]$ denote the path from N_i to N in the ATree. Then by definition of the ATree and by functionality of Edmonds' algorithm described in section 2, we can determine the reduced cost of e:

$$\hat{c}(e) = c(e) - \sum_{N \in [N_i, N_1^c, \dots N_l^c]} y_N$$

As an example in the decomposed ATree of fig 2 we obtain $\hat{c}(e) = c(e) - y_{N_0}$ for all edges $e \in \delta_{E'}(N_1^c) \cap \delta_E(v_0)$. Our practice is to compute a reduction quantity r_i (i.e. the subtracted sum) for each simple node N_i^s of the remaining ATrees with a single BFS on each remaining ATree in a total of $O(n)$ time. Then, we can scan once the edges $e = (u, v_i) \in E'$ and assign them the proper reduced cost $\hat{c}(e) = c(e) - r_i$.

4.4 Inserting Edges

Edge insertion is handled by reduction to edge deletion. Let e_{in} be the edge we want to insert, at cost $c(e_{in})$. We have to check whether e_{in} should replace some edge encoded in the ATree. This check involves *only* c-nodes of the ATree that are ancestors of $N^s_{h(e_{in})}$ and is performed as follows: starting from the node $N^s_{h(e_{in})}$ we follow the path towards the ATree root. For each visited node N, we check whether $c(e(N)) > c(e_{in})$. If this is not the case, we proceed to the parent node. Otherwise, we have found a candidate node N which should have e_{in} as its selected edge, because it is of lower cost. It may be the case that the root node of the ATree is reached: then e_{in} cannot replace any edge of H. In this case we insert it in the digraph and in the *contracted-edges* list associated with the least common ancestor of $N^s_{t(e_{in})}$ and $N^s_{h(e_{in})}$.

Given that we have found a candidate node N which should replace its $e(N)$ with e_{in}, we have to determine whether e_{in} should or should not belong in the "in" cut-set of N. To do so we examine whether the $N^s_{t(e_{in})}$ is hanged in the subtree rooted at N, by engaging a BFS on this subtree. If $N^s_{t(e_{in})}$ is found, it is implied that e_{in} should *not* belong in the "in" cut-set of N, so we simply insert the edge in the digraph and in the *contracted-edges* list of the least common

ancestor of $N^s_{t(e_{in})}$ and $N^s_{h(e_{in})}$. Otherwise, we insert e_{in} in the digraph and engage a *virtual* deletion of $e(N)$, i.e. without actually removing $e(N)$ from the digraph. After this virtual edge deletion recognition of $G(V', E')$ takes place as described in the previous paragraph, and the algorithm of Edmonds is executed over $G(V', E' \cup \{e_{in}\})$.

4.5 Maintaining Contracted Edges

We describe here how to maintain a *contracted-edges*(N^c) list for each c-node N^c of the ATree structure, by introducing a simple modification on the $O(m+n \log n)$ time implementation of Gabow et al. [9], without burdening the complexity. A brief description of the implementation follows.

The loop of Edmonds' algorithm is executed by growing a path, referred to as the *growth path* in [9]. The growth path is constructed as follows: initially, an arbitrary vertex s, called *current root vertex*, is considered and an incoming edge $e = (u, s)$ of minimum cost $\hat{c}(e)$ is selected. Vertex u gets marked, edge e is added in the growth path and the process is repeated by considering vertex u as the current root vertex. If the insertion of e causes a directed cycle (i.e. its tail $t(e)$ is already marked), a contraction of the cycle happens and a new c-vertex replaces all cycle vertices in the growth path. This c-vertex becomes the current root vertex of the growth path.

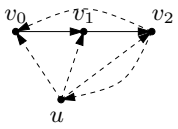
Each vertex $u \in V$ is associated with an *exit list*, which holds outgoing edges of u, incoming to some vertex on the current growth path. If we let $v_0, \ldots, v_l$ be the current growth path, with v_0 its current root vertex, such a list has the following contents:

1. *If u is not on the growth path:* its associated exit list contains only the edges e with $t(e) = u$ and $h(e) = v_j$ such that v_j is on the growth path.
2. *If $u = v_i$ is on the growth path:* only edges e with $t(e) = u$ and $h(e) = v_j$ such that v_j is on the growth path and $j < i$, are contained.

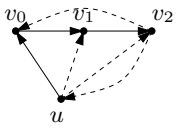
Furthermore, in both cases, the edges are sorted in increasing order of j. When a vertex is either added to the growth path, or takes place in a contraction, its exit list is scanned once (for purposes related to details of [9]) and cleared. The following modified manipulation of exit lists is adopted:

1. When a vertex u is added to the growth path, its exit list is scanned once and cleared as in [9], but each edge (u, v_i) (v_i belonging on the growth path) is added in a list *deprecated*(v_i).
2. When a contraction of vertices $v_0, \ldots, v_k$ happens, the new c-vertex is given an explicit name, say c. The exit lists of the contracted vertices are scanned once and cleared as required in [9], but their contents are merged into a *contracted-edges*(c) list initialized for the new c-vertex c. All *deprecated*(v_i), $i = 0 \ldots k$, are merged into *contracted-edges*(c).

By these modifications all edges contracted due to the emergence of a new c-vertex c (having both their end-vertices in the cycle) are stored in its associated



$exit(u)$:$\{(u, v_0), (u, v_1), (u, v_2)\}$
$exit(v_0)$:$\{\ \}$
$exit(v_1)$:$\{\ \}$
$exit(v_2)$:$\{(v_2, v_0)\}$
$deprecated(v_1)$:$\{(v_0, v_1)\}$
$deprecated(v_2)$:$\{(v_1, v_2)\}$



$exit(u)$:$\{\ \}$ $deprecated(v_0)$:$\{(u, v_0)\}$
$exit(v_0)$:$\{\ \}$ $deprecated(v_1)$:$\{(v_0, v_1), (u, v_1)\}$
$exit(v_1)$:$\{\ \}$ $deprecated(v_2)$:$\{(v_1, v_2), (u, v_2)\}$
$exit(v_2)$:$\{(v_2, v_0), (v_2, u)\}$

$contracted\text{-}edges(c)$:$\{(v_2, v_0), (v_2, u), (u, v_0), (v_0, v_1),$
$(u, v_1), (v_1, v_2), (u, v_2)\}$

Fig. 3. In the upper part, exit lists and deprecated lists are shown for the current growth path v_0, v_1, v_2. When edge (u, v_0) is added on the growth path the updated exit lists and the deprecated lists are as shown in the center part. Augmentation of the growth path with edge (v_2, u) causes contraction of all vertices. The list of contracted edges for the new c-vertex c is the union of exit and deprecated lists, shown in the lower part.

list *contracted-edges*(c). Merging of the lists can be done in k steps and since the algorithm performs k steps anyway for identifying the cycle, its complexity is not burdened. An example of the described manipulation appears in fig. 3.

5 Complexity

In order to study the output complexity of the proposed dynamic scheme, we have to identify the minimal portion of the maintained output that is affected by each edge operation. As mentioned previously, the output consists of all processed vertices (simple and c-vertices). A vertex v (whether a simple or c-vertex) is affected if it takes part in a *different* contraction in the new output after an edge operation. A contraction is defined exactly by the vertices and edges that comprise the directed cycle which caused it. A different contraction is one which was not present in the previous output. We denote the set of affected vertices with ρ, $|\rho|$ being its size. The *extended* set of affected elements (namely vertices and edges incoming to affected vertices) is denoted as $||\rho||$: this definition was introduced in [13] also used in [14, 16]. First we show that:

Lemma 2. *Root nodes of ATrees which emerged after decomposition of the initial ATree represent affected vertices.*

Proof. Let e_{out} be the removed edge, and N be the corresponding ATree node with $e(N) = e_{out}$. Clearly N is affected by definition, since it will change its selected incoming edge. Hence any contraction to which it takes part differs

from previous contractions at least by the new edge. For the rest roots of ATrees one of the following happens: either they take part in at least one contraction not present in the previous output, or they take part in a contraction also present in the previous output. In the latter case however, this contraction also included N, hence in the new output it differs at least by $e(N)$. □

Notice that $|\rho| \leq n$. During edge insertion/deletion all supplementary operations incur $O(n)$ complexity, while re-execution of Edmonds' algorithm processes only affected vertices, given by lemma 2. Hence:

Theorem 2. *Fully dynamic maintenance of a directed minimum cost spanning tree can be done in* $O(n + ||\rho|| + |\rho| \log |\rho|)$ *output complexity per edge operation.*

By the previous discussion and the results of [10]:

Corollary 1. *Fully dynamic maintenance of a directed minimum cost spanning tree can be done in deterministic* $O(n+||\rho|| \log \log |\rho|)$ *and* $O(n+||\rho|| \sqrt{\log \log |\rho|})$ *randomized output complexity per edge operation in sparse digraphs.*

6 Experimental Evaluation

We evaluated the proposed method on sequences of edge operations on digraphs of varying order and density. Implementation was grown in `C++` under version `3.1` of the `gcc` compiler with optimization level 3. Experiments were carried out on an Intel P4 `3.2 GHz` PC with `1 GB` of memory, running `Linux Kernel 2.6`. `CPU` time was measured using the `getrusage()` system call. We implemented the description of [9] for dense digraphs of $O(n^2)$ edges and the deterministic heaps of [10] for sparse digraphs of $O(n)$ edges. Both implementations discourage usage of pre-existing data structure libraries due to the heaps used and due to the *endogenous* nature of other supplementary structures.

6.1 Experimental Setup

A large set of random digraphs divided into three categories was used:

Dense Digraphs: $n = 500i$, $i = 1 \ldots 10$, densities $p = 0.2i$, $i = 1 \ldots 5$.
Sparse Digraphs: $n = 500i$, $i = 1 \ldots 10$, densities $p = \frac{c}{n-1}$, with c taking values in $\{10, 20, 30, 40, 50\}$.
Embedded Cliques: We generated 10 digraphs of order $n = 5000$ and density $\frac{10}{n-1}$, and embedded on each of them a clique of increasing order $500i$, $i = 1 \ldots 10$.

Edge costs were chosen uniformly at random from the range $1 \ldots 1000$. The dynamic algorithm was compared against re-executing Edmonds' algorithm on the whole digraph instance per edge operation (static practice). An edge operation was chosen to be insertion or deletion with probability 0.5. Average `CPU` time and number of iterations performed by each algorithm were derived over 10^4 operations per digraph instance. % Gain for both measures, defined as the relative savings of the dynamic over the static practice, is discussed.

6.2 Discussion

The proposed practice achieves substantial `CPU` time savings over the static DMST re-evaluation, as shown in fig. 4, for both complete and very sparse digraphs. Fig. 5 shows that for general dense digraphs, the savings are stable across orders and densities over 60% on average. For sparse digraphs the gain in `CPU` time increases from about 65% to near 95% when $c = 10$ (very sparse digraphs) as n increases, while the increase becomes more modest when c becomes larger.

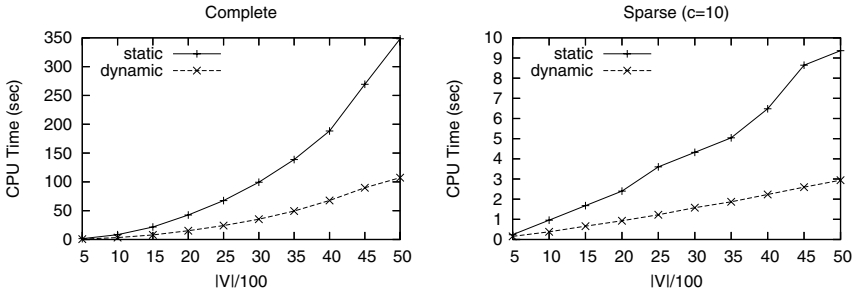


Fig. 4. Performance comparison per edge operation in complete and very sparse digraphs

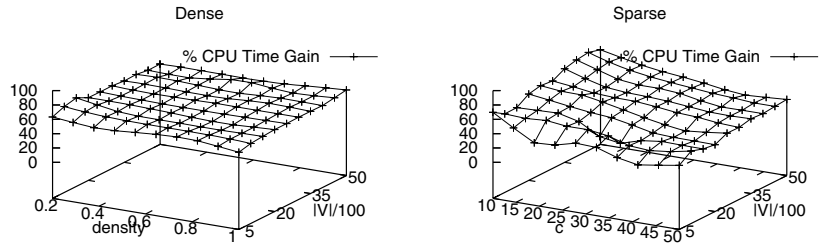


Fig. 5. Almost stable over 60% CPU Time gain of dynamic over static per edge operation in dense digraphs and increasing gain in sparse digraphs

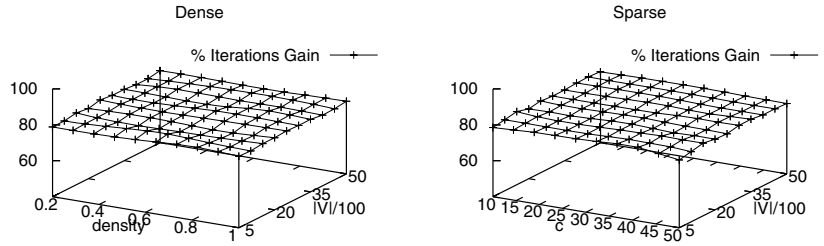


Fig. 6. Stable 80% iterations gain of dynamic over static per edge operation in dense and sparse digraphs

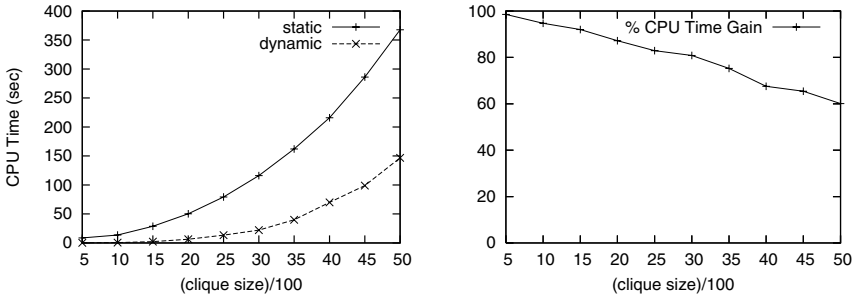


Fig. 7. Performance and % CPU Time gain for embedded cliques of increasing size

A stability of the dynamic practice is observed in fig. 6 in terms of iterations savings to a 80% gain across all dense and sparse graphs. This result combined with the 60% time gain for dense digraphs and the increasing gain observed for large sparse instances, implies a dependance of the overall performance on the density of edges. This dependance was further examined on very sparse instances ($c = 10$) having an embedded clique of increasing size, i.e. on digraphs of increasing non-uniformly distributed density.

The results we obtained confirm this dependance. Initially, when the embedded clique is very small and thus the instance is sparse, the overall gain is approximately 95% as shown in Fig. 7. As the embedded clique grows and the density of the considered digraphs increases, the gain decreases, resulting in a still significant 60% when the whole digraph has become complete.

Conclusively, the proposed dynamic algorithm achieves an update time reduced by a factor of more than 2 as opposed to solving the problem statically on dense digraphs. We believe that the case of sparse digraphs merits theoretical investigation from an average case complexity perspective, since there appears to be an asymptotic improvement on average.

7 Conclusions

We have studied the problem of updating the DMST of a weighted digraph changing by edge insertions and deletions. We provided an $\Omega(n^2)$ complexity lower bound when the only information retained is the DMST itself, and designed a dynamic algorithm of $O(n + ||\rho|| + |\rho| \log |\rho|)$ output complexity, where ρ is a minimal subset of the output that needs to be updated per edge operation. Experimental evaluation of the proposed technique establishes its practical value, and raises an open question regarding average case analysis for sparse digraphs.

Acknowledgements. We thank three anonymous reviewers for comments that helped improve the quality of the paper.

References

1. Edmonds, J.: Optimum branchings. Journal of Research of the National Bureau for Standards **69B** (1967) 125–130
2. Kang, I., Poovendran, R.: Maximizing network lifetime of broadcasting over wireless stationary adhoc networks (to appear). Mobile Networks **11**(2) (2006)
3. Li, N., Hou, J.: Topology Control in Heterogeneous Wireless Networks: Problems and Solutions. In: Proceedings of the 23rd IEEE INFOCOM. (2004)
4. Li, Z., Hauck, S.: Configuration compression for virtex fpgas. In: Proceedings of the 9th IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'01. (2001) 147–159
5. He, L., Mitra, T., Wong, W.: Configuration bitstream compression for dynamically reconfigurable FPGAs. In: Proceedings of the 2004 International Conference on Computer-Aided Design, ICCAD'04. (2004) 766–773
6. Bock, F. In: An algorithm to construct a minimum spanning tree in a directed network. In: Developments in Operations Research. Gordon and Breach (1971) 29–44
7. Chu, Y.J., Liu, T.H.: On the shortest arborescence of a directed graph. Scientia Sinica **14** (1965) 1396–1400
8. Tarjan, R.E.: Finding optimum branchings. Networks **7** (1977) 25–35
9. Gabow, H.N., Galil, Z., Spencer, T.H., Tarjan, R.E.: Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica **6** (1986) 109–122
10. Mendelson, R., Tarjan, R.E., Thorup, M., Zwick, U.: Melding Priority Queues. In: Proceedings of SWAT'04, Springer LNCS 3111. (2004) 223–235
11. Eppstein, D., Galil, Z., Italiano, G.F.: 8: Dynamic graph algorithms. In: Algorithms and Theory of Computation Handbook. CRC Presss (1999)
12. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Journal of the ACM **48** (2001) 723 – 760
13. Alpern, B., Hoover, R., Rosen, B.K., Sweeney, P.F., Zadeck, F.K.: Incremental evaluation of computational circuits. In: Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, SODA'90. (1990) 32–42
14. Ramalingam, G., Reps, T.: On the complexity of dynamic graph problems. Theoretical Computer Science **158**(1&2) (1996) 233–277
15. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully dynamic algorithms for maintaining shortest paths trees. Journal of Algorithms **34** (2000) 251–281
16. Pearce, D.J., Kelly, P.H.J.: A Dynamic Algorithm for Topologically Sorting Directed Acyclic Graphs. In: Proceedings of the 3rd Workshop on Efficient and Experimental Algorithms, WEA'04, Springer LNCS 3059. (2004) 383–398
17. Spira, P.M., Pan, A.: On Finding and Updating Spanning Trees and Shortest Paths. SIAM Journal on Computing **4**(3) (1975) 364–380
18. Rabin, M.O.: Proving simultaneous positivity of linear forms. Journal of Computers and Systems Sciences **6** (1972) 639–650