INFORMS Journal on Computing
Vol. 20, No. 2, Spring 2008, pp. 191–204
ISSN 1091-9856 | EISSN 1526-5528 | 08 | 2002 | 0191

DOI 10.1287/ijoc.1070.0231

Speeding Up Dynamic Shortest-Path Algorithms

Luciana S. Buriol
Instituto de Informática, Universidade Federal do Rio Grande do Sul,
Porto Alegre, Rio Grande do Sul 91501, Brazil, buriol@inf.ufrgs.br

Mauricio G. C. Resende, Mikkel Thorup
Algorithms and Optimization Research Department, AT&T Labs Research,
Florham Park, New Jersey 07932 {mgcr@research.att.com, mthorup@research.att.com}

Dynamic shortest-path algorithms update the shortest paths taking into account a change in an arc weight. This paper describes a new generic technique that allows the reduction of heap sizes used by several dynamic single-destination shortest-path algorithms. For unit weight changes, the updates can be done without heaps. These reductions almost always reduce the computational times for these algorithms. In computational testing, several dynamic shortest-path algorithms with and without the heap-reduction technique are compared. Speedups of up to a factor of 1.8 were observed using the heap-reduction technique on random weight changes and of over a factor of five on unit weight changes. We compare as well with Dijkstra's algorithm, which recomputes the paths from scratch. With respect to Dijkstra's algorithm, speedups of up to five orders of magnitude are observed.

Key words: shortest-path algorithms; Dijkstra's algorithm; dynamic shortest-path algorithms; heaps; graphs; trees
History: Accepted by William Cook, former Area Editor for Design and Analysis of Algorithms; received September 2003; revised August 2004, February 2006, April 2007; accepted May 2007. Published online in *Articles in Advance* December 11, 2007.

1. Introduction

Finding a shortest path is a fundamental graph problem, which besides being a basic component in many graph algorithms, has numerous real-world applications. Consider a weighted directed graph $G = (V, E, w)$, where V is the vertex set, E is the arc set, and $w \in \mathbb{R}^{|E|}$ is the arc-weight vector. Given a source vertex $s \in V$, the single-source shortest-path problem is to find a shortest-path graph g^{SP} from source s to every vertex $v \in V$. By reversing the direction of each arc in the graph, we transform the single-source into the single-destination shortest-path problem.

There are applications where g^{SP} is given and must be updated after a weight change. Considering a single weight change, usually only a small part of the graph is affected. For this reason, it is sensible to avoid the computation of g^{SP} from scratch, but only update the part of the graph affected by the arc weight change. This problem is known as the *dynamic shortest-path (DSP) problem*. DSP algorithms for the single-source shortest-path problem are the focus of this paper. An algorithm is referred to as *fully dynamic* if both arc deletion (arc weight is set as ∞) and insertion are supported, and *semidynamic* incremental (decremental) if only arc deletion (insertion) is supported.

Previous work on algorithms for the dynamic shortest-path problem include Murchland (1970), Goto and Sangiovanni-Vincentelli (1978), and Dionne (1978). Considering semidynamic decremental algorithms, previous work was done by Gallo (1980) and Fujishige (1981), whereas for the incremental case (arc deletion), we refer to Even and Shiloach (1981).

Many algorithms were proposed for solving this problem, but the algorithm RR, of Ramalingam and Reps (1996a), seems to be the most used (Buriol et al. 2005, Fortz and Thorup 2004, Frigioni et al. 1998). Their algorithm is not the best for all applications. However, one of its main advantages is to have good performance in most situations. First of all, it updates the shortest-path graph, rather than a shortest-path tree, although it can be easily specialized for updating a tree (Demetrescu et al. 2000). Even and Shiloach (1981) proposed a semidynamic incremental algorithm that works in cascades, which can be computationally expensive for large arc-weight increments. RR has good performance independent of the increment range. The semidynamic incremental algorithm of Demetrescu (2001), for updating a shortest-path tree, can have good performance if most of the affected nodes have no alternative shortest paths, but its performance can be poor otherwise. Again, RR has good performance in both situations. Even the algorithm of Frigioni et al. (1996), which theoretically is better than RR, was usually outperformed by RR in computational testing (Frigioni et al. 1996).

Recently, Demetrescu et al. (2000) proposed a specialization of the Ramalingam and Reps algorithm for updating a shortest-path tree, which is a revision and extension of their previous work (Frigioni et al. 1998). In the new version, they present a fully dynamic algorithm, whereas in the earlier paper, they proposed a semidynamic incremental algorithm. Their specialized algorithm did not make use of the special tree proposed by King and Thorup (2001). In graphs where only a few affected nodes have alternative shortest paths, the incremental algorithm of Demetrescu (2001) usually has better performance.

For maintaining all pairs shortest paths in directed graphs with real-valued arc weights, we refer to the fully dynamic algorithms of Demetrescu and Italiano (2001, 2003) and the experimental results in Demetrescu et al. (2003).

Many theoretical studies of dynamic shortest-path algorithms have been carried out, but few experimental results are known. Frigioni et al. (1998) compared the algorithm of Ramalingam and Reps (1996a) with the algorithm of Frigioni et al. (1996) to update the single-source shortest-path graph. They concluded that the algorithm of Ramalingam and Reps is usually better in practice, with respect to running times, although their algorithm has a better worst-case time complexity (Ramalingam and Reps 1996b). Demetrescu et al. (2000) compare the incremental algorithms of Demetrescu et al. (2000), Frigioni et al. (1998), Ramalingam and Reps (1996a), and a specialization of Demetrescu et al. (2000) described in Demetrescu (2001). For the set of instances used in their study, the results show that their new idea speeds up the running times for updating a tree.

This paper presents a new generic technique that allows the reduction of heap (priority queue) sizes in several dynamic shortest-path algorithms. For unit weight changes, the updates are done without heaps for most of the algorithms. Suppose an arc weight is increased by $\triangle$. In the standard implementations of DSP algorithms, all affected nodes (whose distances have changed) will be placed in a heap, on which a Dijkstra subroutine is run. However, a subset of these nodes have their distances increased by exactly $\triangle$. In this case, an update without heaps can be applied. The basic idea of the reduced-heap technique is to apply the Dijkstra subroutine for only those nodes whose distances increase by an amount smaller than $\triangle$. In the worst case the Dijkstra subroutine may, of course, be applied to all affected nodes. But often, in practice, the subset of affected nodes whose distances increase by exactly $\triangle$ is big. Avoiding the use of heaps on this set almost always results in substantial savings, reducing the computational times for DSP algorithms. In computational testing, several dynamic shortest-path algorithms with and without the heap-reduction technique are compared. We also compare with Dijkstra's algorithm, which recomputes the paths from scratch.

In this paper, we named the algorithms *increase* and *decrease*, instead of adopting the nomenclature *incremental* and *decremental* used for arc insertion and deletion, respectively. We note that deleting an arc, as in a decremental algorithm, can be viewed as increasing its weight to infinity. However, the focus here is on smaller weight changes. The dynamic algorithms to update the graph after an arc *weight increase* are called *increase algorithms*, while *decrease algorithms* update a graph after an arc *weight decrease*. The algorithms are named with the letter `G` or `T` if they update a shortest-path graph or tree, respectively. The names also indicate the originators of the algorithms (in superscript) and the sign $+$ or $-$ (in subscript) is used if it refers to an increase or decrease algorithm, respectively. When the name of the algorithm is used without the sign, it refers to both increase and decrease cases, or it is followed by the `Incr` or `Decr` indication. The terms `std` and `rh` are used to refer to the standard and reduced-heap variants of the algorithms, respectively. The standard algorithms were originally designed to update the single-source shortest paths in directed graphs with real positive arc weights. In this paper, the algorithms update the single-destination shortest paths with integer positive arc weights (most of the real-world applications use positive weights).

The pseudocodes of all algorithms discussed in this paper, as well as the tables with detailed presentation of the results, are in the Online Supplement to this paper (available at http://joc.pubs.informs.org/ecompanion.html).

In §2, the data structures used to represent the graph and the solution (graph or tree) are presented and some implementation tricks described. Next, some standard increase dynamic shortest-path algorithms, as well as the heap-reduction technique for this case, are presented in §3. The weight decrease case is presented in §5. A discussion of the standard and reduced-heap increase algorithms with respect to heap size and memory usage is given in §4. A similar discussion for the arc-weight-decrease case is given in §6. Computational results are reported in §7 and concluding remarks are made in §8.

2. Implementation Issues

In this section, the data structures used to represent the graph and the solution (graph or tree) are presented and some implementation tricks are described.

2.1. Data Structures

We first describe two sets of data structures used in the implementations of the algorithms.

The input graph is stored in forward and reverse representations. It is represented by four arrays. The $|E|$ array `forward` stores the arcs, where each arc consists of its node indices *tail* and *head*. The arcs in this array are sorted by their tails, with ties broken by their heads. The ith position of the $|V|+1$ array `point` indicates the initial position in `forward` of the list of outgoing arcs from node i. By assumption, the last position in `forward` of the list of outgoing arcs from node i is $\texttt{point}[i+1]-1$.

The $|E|$ array `reverse` stores the arcs, where the arcs are sorted by their heads, with ties broken by their tails. To save space, each arc in `reverse` is represented by the index of this arc in `forward`. The ith position of the $|V|+1$ array `rpoint` indicates the initial position in `reverse` of the list of incoming arcs into node i. By assumption, the last position in `reverse` of the list of incoming arcs into node i is $\texttt{rpoint}[i+1]-1$.

Solutions are represented in different algorithms either as trees or graphs. For both cases, the $|E|$ array `w` stores the arc weights, and the $|V|$ array `d` stores the distances of the nodes to the destination.

In the case of trees, the ith position of the $|V|$ array $\texttt{t}^{\text{SP}}$ indicates the index of the outgoing arc of node i in the shortest-path tree. The destination node t stores the value 0, i.e., $t_t^{\text{SP}}=0$.

In the case of graphs, two arrays are used. The $|E|$ array $\texttt{g}^{\text{SP}}$ is a $0-1$ indicator array whose ith position is 1, if and only if arc i is in the shortest-path graph. Finally, the ith position of the $|V|$ array δ stores the number of arcs in the shortest-path graph outgoing node i.

2.2. Implementation

In this section, we indicate how some of the basic operations referred to in the pseudocodes were implemented.

In these pseudocodes, the heap-function names follow Ramalingam and Reps (1996a). These functions are as follows:

- `HeapMember`(H, u): returns 1 if element u is in heap H, and 0 otherwise;
- `HeapSize`(H): returns the number of elements in heap H;
- `FindAndDeleteMin`(H): returns the item in heap H with minimum key and deletes it from H;
- `InsertIntoHeap`(H, u, k): inserts an item u with key k into heap H;
- `AdjustHeap`(H, u, k): if $u \in H$, changes the key of element u in heap H to k and updates H. Otherwise, u is inserted into H.

In several points in the algorithm one must scan all outgoing or all incoming arcs of a node. To scan the outgoing arcs of node u, i.e., $e=(\overrightarrow{u,v}) \in \texttt{OUT}(u)$, simply scan positions $\texttt{point}[u], \ldots, \texttt{point}[u+1]-1$ of array `forward`. Similarly, to scan the incoming arcs of node u, i.e., $e=(\overrightarrow{s,u}) \in \texttt{IN}(u)$, simply scan positions $\texttt{reverse}[\texttt{rpoint}[u]], \ldots, \texttt{reverse}[\texttt{rpoint}[u+1]-1]$ of array `forward`.

Set Q is stored as a $|V|$ array where Q_i is the ith element of the set. Set U is represented is a similar fashion.

3. Standard and Reduced-Heap Versions of the Increase Algorithms

This section introduces the general technique of reducing the heap size used by the increase algorithms and describes four increase dynamic shortest-path algorithms. All standard algorithms are based on the same idea: considering that the weight of an arc $a=(\overrightarrow{u,v})$ has increased, a set Q of affected nodes is determined. This set is composed of the nodes that have all their shortest paths traversing arc a. The changes are applied only to these nodes and their incoming and outgoing arcs.

The set Q can remain empty in two situations: if a does not belong to a shortest path, or if it does but u has an alternative shortest path to the destination node. In the first case nothing is done, and in the second case a local update is applied.

If the weight increase affects the distance label of u, the tail node of arc a, a more complex update is required. All nodes that have their distances increased are inserted into a set Q and have their distances set to ∞. Q is initialized with u. All incoming arcs e to these nodes are traversed. If the tail node of an arc e has no alternative shortest path to the destination, it is also inserted into Q.

In a second phase, the algorithm updates the distances of nodes in Q. First, by traversing the outgoing arcs of each node $u \in Q$, their distances are updated considering the arcs directly linking nodes outside set Q. Next, Dijkstra's algorithm is applied considering nodes in Q, taking into account their current distance labels. Thus, all nodes $u \in Q$ are inserted into a heap H.

In the case of updating the shortest-path graph, a third phase is required. To identify the arcs that belong to g^{SP}, all outgoing arcs of nodes in $u \in Q$ are traversed.

The standard versions of the increase algorithms insert into a heap H all nodes initially in set Q, while the reduced-heap variants use heaps to update only a subset of Q.

In the proposed *reduced-heap* variants of these algorithms, the distances of nodes in Q are increased by $\triangle$ (instead of being set to ∞), which is the total amount of the original increment of arc $a=(\overrightarrow{u,v})$. After that, the actual decrease $\triangledown$ of the distance label of node u is computed. Next, all nodes $u \in Q$ have their distances adjusted downward by an appropriate amount $\triangle - \triangledown$. Dijkstra's algorithm is applied only for those nodes

that have a shorter path and can thus have their distance label further decreased. As we will see in §7, the number of nodes inserted into the heap is usually much smaller than the number of nodes in Q.

We present the increase algorithms in §§3.1–3.4. The weight can be increased by any amount. If an explicit arc removal is required, one can simply change its weight to infinity and apply one of the algorithms. They receive as input the arc a, which has the weight increased, the vector w of weights (updated with the weight increase of arc a), and the distance vector d. Furthermore, in the case of an algorithm for updating a shortest-path tree, the vector t^{SP} is also an input parameter. When updating the shortest-path graph, g^{SP} and δ are the inputs, instead of t^{SP}. The reduced-heap variants also receive as input the increment $\triangle$ on arc a. As output they produce the updated input arrays.

In §§3.1–3.4, the standard and reduced-heap versions of these algorithms are presented.

3.1. G_+^{RR}: Increase Algorithm of Ramalingam and Reps for Updating the Shortest-Path Graph

G_+^{RR} updates the shortest-path graph $g^{SP} = (V, E^{SP})$ when the weight of an arc a is increased by $\triangle$. Figure 1 shows both the standard and reduced-heap variants.

The pseudocode on the left side of Figure 1 presents $stdG_+^{RR}$, the standard algorithm. Clearly, if arc a is not in the current graph g^{SP}, the algorithm stops (line 1). Otherwise, arc a is removed from g^{SP} (line 2). If node u has an alternative shortest path to the destination node, then the algorithm stops (line 4). Otherwise, the set Q is initialized with node u (line 5). The loop in lines 6 to 15 identifies the remaining affected nodes of g^{SP} and adds them to Q. The loop in lines 16 to 21 updates distances from nodes $u \in Q$ (line 18) and inserts these nodes into heap H (line 20) if its distance was decreased. The main objective of this loop is to update distances of nodes that have an alternative shorter path linking nodes outside Q. The order in which nodes are considered in this loop can affect running times, but not the correctness of the algorithm. This is true for this algorithm, as well as for all increase algorithms presented in this paper. The loop in lines 22 to 36 updates the distances of nodes in Q using heap H (lines 24 to 29) and restores g^{SP} (lines 30 to 35).

The pseudocode on the right side of Figure 1 presents rhG_+^{RR}, the reduced-heap variant for the algorithm. The first 15 lines of rhG_+^{RR} are identical to the first 15 of $stdG_+^{RR}$, with the exception of line 7. Instead of setting the distance of node u to ∞, it is just increased by $\triangle$. In the case of unit weight increase ($\triangle = 1$) the commands from lines 17 to 41 are not executed and heap H is not used. Lines 17 to 21 calculate the maximum amount that the distances of nodes $u \in Q$ will decrease. We denote by Q_0 the first element inserted into Q, and by $\triangledown$ the amount that the distance of node u decreases. In the loop from lines 22 to 32, all nodes from Q, excluding node Q_0, have their distances decreased by $\triangledown$ (line 23). Furthermore, the distances of nodes $u \in Q \backslash \{Q_0\}$ with a shorter path linking nodes $v \notin Q$ are updated. In the loop from lines 33 to 42, nodes u from heap H are removed one by one. All arcs $e = (\overrightarrow{s, u})$ incoming into node u are traversed. If node s has a shorter path traversing node u, its distance is updated (line 37) and heap H is adjusted (line 38). The loop in lines 43 to 50 restores g^{SP} adding the missing arcs in the shortest paths from nodes $u \in Q$.

3.2. T_+^{RR}: Specialization of Ramalingam and Reps Increase Algorithm for Updating a Shortest-Path Tree

The main difference with respect to G_+^{RR} is that T_+^{RR} is specialized to update a shortest-path tree t^{SP} rather than the shortest-path graph g^{SP}. The shortest-path tree t^{SP} maintained by t_+^{RR} stores an arc $a = (\overrightarrow{u, v})$ associated with each node $u \in V$. The arc a is any one from the outgoing adjacency list of node u belonging to a shortest path. In the standard algorithm, t^{SP} is updated while the distances are updated, instead of employing a loop just for this purpose, as $stdG_+^{RR}$ does. Furthermore, the set Q of affected nodes is identical to the one in $stdG_+^{RR}$, but the procedure for identification is slightly different. The main reason is that this algorithm does not maintain a variable δ associated with each node, and the set of outgoing arcs of a node is traversed to verify whether it has an alternative shortest path. On the other hand, if an arc $a = (\overrightarrow{u, v})$, incoming into an affected node v, is not the one in t_u^{SP}, i.e, $t_u^{SP} \neq a$, nothing needs to be done, even if it is a shortest-path arc, because it is known that the arc in t_u^{SP} represents an alternative shortest path.

In the reduced-heap variant, the increase amount $\triangle$ is an input parameter. As the algorithm updates t^{SP} at the same time that Q is identified, in the case of unit increment ($\triangle = 1$) the distances of nodes in Q increase by exactly one unit, and the algorithm stops as soon as Q is totally identified. Otherwise, if $\triangle > 1$, the algorithm uses the idea of the reduced heap, similar to rdG_+^{RR}.

3.3. T_+^{KT}: Incremental Algorithm for Updating the Special Shortest-Path Tree Proposed by King and Thorup

The main difference with respect to T_+^{RR} is that T_+^{KT} updates a special shortest-path tree. It stores, for each node u, the first shortest-path arc $a = (\overrightarrow{u, v})$ belonging to the outgoing adjacency list of node u. The advantage of storing this special tree is to be able

```
procedure stdG+RR(a = (u→, v→), w, d, δ, g^SP)
 1  if g_a^SP = 0 return;
 2  g_a^SP = 0;
 3  δ_u = δ_u − 1;
 4  if δ_u > 0 then return;
 5  Q = {u};
 6  for u ∈ Q do
 7    d_u = ∞;
 8    for e = (s→, u→) ∈ IN(u) do
 9      if g_e^SP = 1 then
10        g_e^SP = 0;
11        δ_s = δ_s − 1;
12        if δ_s = 0 then Q = Q ∪ {s};
13      end if
14    end for
15  end for
16  for u ∈ Q do
17    for e = (u→, v→) ∈ OUT(u) do
18      if d_u > d_v + w_e then d_u = d_v + w_e;
19    end for
20    if d_u ≠ ∞ then InsertIntoHeap(H, u, d_u);
21  end for
22  while HeapSize(H) > 0 do
23    u = FindAndDeleteMin(H);
24    for e = (s→, u→) ∈ IN(u) do
25      if d_s > d_u + w_e then
26        d_s = d_u + w_e;
27        AdjustHeap(H, s, d_s);
28      end if
29    end for
30    for e = (u→, v→) ∈ OUT(u) do
31      if d_u = w_e + d_v then
32        g_e^SP = 1;
33        δ_u = δ_u + 1;
34      end if
35    end for
36  end while
end stdG+RR.
```

```
procedure rhG+RR(a = (u→, v→), w, d, δ, g^SP, △)
 1  if g_a^SP = 0 return;
 2  g_a^SP = 0;
 3  δ_u = δ_u − 1;
 4  if δ_u > 0 then return;
 5  Q = {u};
 6  for u ∈ Q do
 7    d_u = d_u + △;
 8    for e = (s→, u→) ∈ IN(u) do
 9      if g_e^SP = 1 then
10        g_e^SP = 0;
11        δ_s = δ_s − 1;
12        if δ_s = 0 then Q = Q ∪ {s};
13      end if
14    end for
15  end for
16  if △ > 1 then
17    dist = d_Q0;
18    for e = (Q0→, v→) ∈ OUT(Q0) do
19      if d_Q0 > d_v + w_e then d_Q0 = d_v + w_e;
20    end for
21    ▽ = dist − d_Q0;
22    for u ∈ Q\{Q0} do
23      d_u = d_u − ▽;
24      flag = 0;
25      for e = (u→, v→) ∈ OUT(u) do
26        if d_u > d_v + w_e then
27          d_u = d_v + w_e;
28          flag = 1;
29        end if
30      end for
31      if flag = 1 then InsertIntoHeap(H, u, d_u);
32    end for
33    while HeapSize(H) > 0 do
34      u = FindAndDeleteMin(H);
35      for e = (s→, u→) ∈ IN(u) do
36        if d_s > d_u + w_e then
37          d_s = d_u + w_e;
38          AdjustHeap(H, s, d_s);
39        end if
40      end for
41    end while
42  end if
43  for u ∈ Q do
44    for e = (u→, v→) ∈ OUT(u) do
45      if d_u = w_e + d_v then
46        g_e^SP = 1;
47        δ_u = δ_u + 1;
48      end if
49    end for
50  end for
end rhG+RR.
```

Figure 1 Pseudocodes of Procedures $stdG_+^{RR}$ (Left) and rhG_+^{RR} (Right)

to find an alternative shortest path for a node s without exploring all outgoing arcs in the set $\mathtt{OUT}(s)$. Therefore, during the process of identification of set Q, while searching for an alternative shortest path for an affected node s, only a subset $\mathtt{OUT}(s) \supset \mathtt{OUT}^{KT}(s)$ is traversed.

The drawback is that some extra computational effort is needed to maintain this special tree. When the distances of nodes in Q are updated, making use of heap H, the alternative shortest paths also should be identified to make sure that, in this case, the shortest-path arc stored is the first one in the outgoing adjacency list.

In the reduced-heap variant of this algorithm, extra effort is required to store the correct arc in t^{SP}. In the case of unit increment, the subset of arcs $\mathtt{OUT}(s) \supset \mathtt{OUT}^{*}(s)$ of each node $s \in Q$ should be traversed. The subset $\mathtt{OUT}^{*}(s)$ is composed of outgoing arcs that are

located before the stored arc t_s^{SP} in the outgoing adjacency list. If there is at least one alternative shortest path using arcs of this subset, the first shortest-path arc found replaces the current t_s^{SP}. Additional tests are done to ensure that the special shortest-path tree is being maintained correctly.

3.4. T_+^D: Incremental Algorithm of Demetrescu for Updating a Shortest-Path Tree

As in T_+^{RR} and T_+^{KT}, T_+^D also updates a shortest-path tree. The main difference is that T_+^D uses a simpler mechanism for detecting the affected nodes Q^D. In other words, it relaxes the notion of affected. However, $|Q^{RR}| = |Q^{KT}| \leq |Q^D|$ for the standard implementations.

If some arc a has its weight increased by $\triangle$, then set Q^{RR} is composed only of nodes such that all their shortest paths traverse arc a. In T_+^D, set Q^D is not only composed of those nodes, but can also have some of the nodes that have an alternative shortest path not traversing arc a. The idea is that in graphs where only a few nodes (or none) have alternative shortest paths $|Q^D| \approx |Q^{RR}|$, with the advantage of Q^D being identified with a simpler mechanism. This algorithm is similar to $stdT_+^{RR}$, differing only in the loop that identifies Q. If e is the outgoing arc of node u in the shortest-path tree, i.e., $t_s^{SP} = e$, node s is inserted into Q, even if it has an alternative shortest path to the destination node. The idea of this algorithm is not to waste time looking for an alternative path. This pays off when the distribution of arc weights is spread out and the probability of ties is low.

As in the standard variant, the reduced-heap variant of this algorithm is similar to rhT_+^{RR}, differing only in how it identifies Q. Furthermore, even in the case of unit increment, the additional nodes inserted into Q are updated making use of a heap.

4. Reduced-Heap vs. Standard Versions of the Increase Algorithms

In this section, we make a few observations comparing the reduced-heap variants with the respective standard versions of the increase algorithms discussed in §3. The heap sizes of these algorithms are illustrated in Figures 2 and 3. The set Q of affected nodes is linked to the remaining part of the shortest-path graph by arc a, the first arc whose weight was increased. Node t is the destination node. The shaded part of Q is composed of the nodes that were inserted into heap H.

As observed in the figures, in the case of unit increment for rhG_+^{RR}, rhT_+^{RR}, and rhT_+^{KT}, the heap H is empty, whereas for rhT_+^D, the nodes $u \in (Q^D \backslash Q^{RR})$ are inserted into H. These nodes are the ones treated as affected while in fact they were not. For this reason, comparing any single case of algorithms `std`, `rh`, and `unit increment`, $Q^D \supseteq Q^{RR} = Q^{KT}$. For all of these algorithms, $H^{std} \supseteq H^{rh}_{\text{random}} \supseteq H^{rh}_{\text{unit}}$.

No additional memory is required to implement the reduced-heap variants of the increase algorithms.

5. Standard and Reduced-Heap Versions of the Decrease Algorithms

This section introduces the idea of reducing the heap size used by the algorithms to update a graph after an arc-weight decrease. Moreover, we describe three algorithms for arc-weight decrease, as well as the idea of reduced heap applied to them. The weight can be decreased by any amount.

Let $a \in E$ be the arc whose weight is to be decreased. All standard algorithms are based on the same idea: considering that the weight of an arc a has decreased, Q of affected nodes is determined. In this case, the set Q is composed by all nodes that have at least one shortest-path-traversing arc a after its weight decreases. The changes are applied only to nodes of Q and their incoming and outgoing arcs. For that, all nodes $u \in Q$ are inserted into a heap H.

In the reduced-heap variants, Q is divided in two subsets, Q and U. Q is composed of all nodes that have at least one shortest-path-traversing arc a. The subset U is composed of nodes whose shortest paths originally did not traverse arc a but do so after the decrement of w_a. The idea of reduced heap avoids the

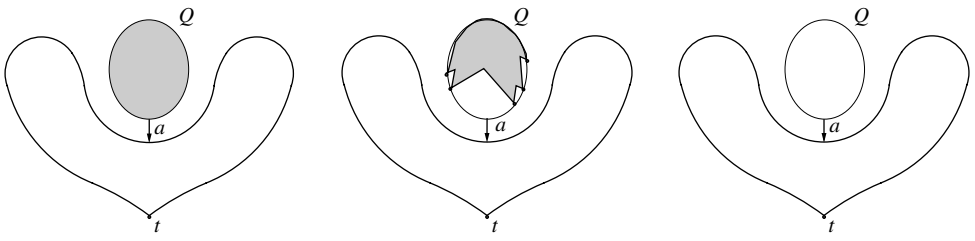

Figure 2 Heap Size Used by the Standard and Reduced-Heap Increase Algorithms G_+^{RR}, T_+^{RR}, and T_+^{KT}

Notes. The left graph represents the standard algorithms. The middle graph represents the reduced-heap variants for random weight increase. In the right graph, for the reduced-heap variants with unit weight increment, the heap is empty.

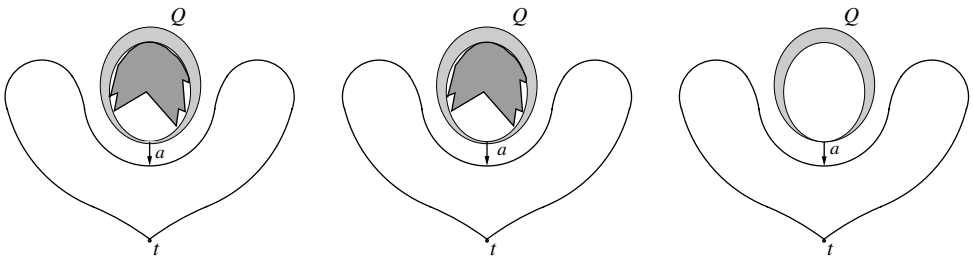

Figure 3 Heap Size Used by the Increase Algorithms $stdT_+^D$ and rhT_+^D

Notes. The left graph represents the standard algorithm. The middle graph represents the reduced-heap variant for random weight increase. In the right graph, for the reduced-heap variant with unit weight increment, the heap contains only unaffected nodes that were inserted into Q.

use of heaps for updating nodes in Q and only inserts into the heap nodes from set U.

For both the standard and reduced-heap algorithms, Q remains empty if a is not in the shortest-path tree before its weight decreases. In this case, Q remains empty if the decrease does not affect the distance label of any node. Also, subset U is used only if Q is not empty. If $|Q| > 0$, U remains empty in the case that no node $u \notin Q$ has an alternative shortest path linking nodes $u \in Q$.

The reduced-heap algorithms have two phases. Initially, the shortest paths are updated considering nodes in Q. Next, they are updated considering nodes belonging to U. The decrease amount ∇ is computed prior to determining Q. Because Q contains all nodes with at least one shortest-path-traversing arc a, we decrease the distance label of these nodes by exactly ∇, without using heaps. However, heaps are needed to compute distance labels of nodes $u \in U$.

In the §§5.1–5.3, we briefly describe the standard and reduced-heap versions of the decrease algorithms.

5.1. G_-^{RR}: Decrease Algorithm of Ramalingam and Reps for Updating the Shortest-Path Graph

The Ramalingam and Reps arc-weight-decrease algorithm (Ramalingam and Reps 1996a) updates the shortest-path graph considering an arc-weight decrease. Figure 4 presents the standard and reduced-heap variants of these algorithms.

The pseudocode for the standard procedure is given on the left side of Figure 4. Recall that $a = (\overrightarrow{u, v}) \in E$ is the arc whose weight is decreased. If d_u remains unchanged, the algorithm stops (line 1). Otherwise, if node u has an alternative shortest-path-traversing arc a, a is inserted in g^{SP} and the algorithm stops (lines 3 to 5). Otherwise, H is initialized with node u.

In the loop in lines 9 to 29, nodes from H are removed, one by one, in order of the smallest to the largest distance to the destination node, and their respective distances and incoming/outgoing arcs are updated in g^{SP}.

We now consider the Ramalingam and Reps weight-decrease algorithm with reduced heap. The first phase of rhG_-^{RR} is described on the left side of Figure 4. In line 7 the amount ∇, by which the distance of node u will be decreased, is calculated. The array *degree* is used to avoid inserting a node u into set Q more than once. Moreover, *degree* is used to identify nodes in Q having alternative shortest paths not traversing arc a. Furthermore, we consider that all positions of vector *degree* are initialized with zero when this vector is created.

The loop in lines 11 to 26 identifies nodes $u \in Q$, making use of the array *degree*, and updates g^{SP} in the case of unitary decrement.

If a node $u \in Q$ has one or more alternative shortest paths not traversing arc a, these paths are no longer shortest after the weight of arc is reduced and must be removed from the shortest-path graph. The existence of an alternative path is determined making use of array *degree*. The loop in lines 27 to 37 removes these paths by analyzing each node $u \in Q$, one at a time.

In the case of unit decrement, the algorithm stops in line 38. In the loop from lines 39 to 52, all incoming arcs $e = (\overrightarrow{s, u})$ into node $u \in Q$ are scanned. The distances of nodes s and the shortest-path graph g^{SP} are updated considering these traversed links.

Recall that phase 1 of the algorithm identifies the set Q, containing all nodes that have at least one shortest-path-traversing arc a, and updates the part of the graph that contains these nodes. Furthermore, all nodes $s \in U$ with an alternative shortest-path-linking set Q are identified and if d_s is reduced, s is inserted into heap H. The second phase updates the remaining affected part of the graph, i.e., the nodes that now have a shortest path through arc a, which had its weight decreased, but do not have an arc linking directly to a node in Q. The second phase of this algorithm is identical to lines 9 to 29 in procedure $stdG_-^{RR}$.

5.2. T_-^{RR}: Specialization of Ramalingam and Reps Decrease Algorithm for Updating a Shortest-Path Tree

Algorithm T_-^{RR} is a specialization of the Ramalingam and Reps algorithm (G_-^{RR}) restricted to updating a shortest-path tree. A similar algorithm was proposed

```
procedure stdG_-^RR(a = (u→, v→), w, d_V, δ_V, g^SP)
 1  if d_u < d_v + w_a then return;
 2  if d_u = d_v + w_a then
 3     g_a^SP = 1;
 4     δ_u = δ_u + 1;
 5     return;
 6  end if
 7  d_u = d_v + w_a;
 8  InsertIntoHeap(H, u, d_u);
 9  while HeapSize(H) > 0 do
10     u = FindAndDeleteMin(H, d);
11     δ_u = 0;
12     for e = (u→, v→) ∈ OUT(u) do
13        if d_u = d_v + w_e then
14           δ_u = δ_u + 1;
15           g_e^SP = 1;
16        end if
17        else g_e^SP = 0;
18     end for
19     for e = (s→, u→) ∈ IN(u) do
20        if d_s > d_u + w_e then
21           d_s = d_u + w_e;
22           AdjustHeap(H, s, d_s);
23        end if
24        else if g_e^SP = 0 and d_s = d_u + w_e then
25           g_e^SP = 1;
26           δ_s = δ_s + 1;
27        end if
28     end for
29  end while
end stdG_-^RR.
```

```
procedure rhG_-^RR Ph1(a = (u→, v→), w_E, d_V, δ_V, g^SP)
 1  if d_u < d_v + w_a then return;
 2  if d_u = d_v + w_a then
 3     g_a^SP = 1;
 4     δ_u = δ_u + 1;
 5     return;
 6  end if
 7  ▽ = d_u − d_v − w_a;
 8  d_u = d_u − ▽;
 9  Q = {u};
10  degree_u = δ_u − 1;
11  for u ∈ Q do
12     for e = (s→, u→) ∈ IN(u) do
13        if g_e^SP = 1 then
14           if degree_s = 0 then
15              d_s = d_s − ▽;
16              Q = Q ∪ {s};
17              degree_s = δ_s − 1;
18           end if
19           else degree_s = degree_s − 1;
20        end if
21        else if ▽ = 1 and d_s = d_u + w_e then
22           g_e^SP = 1;
23           δ_s = δ_s + 1;
24        end if
25     end for
26  end for
27  for u ∈ Q do
28     if degree_u > 0 then
29        degree_u = 0;
30        for e = (u→, v→) ∈ OUT(u) do
31           if g_e^SP = 1 and d_u < d_v + w_e then
32              g_e^SP = 0;
33              δ_u = δ_u − 1;
34           end if
35        end for
36     end if
37  end for
38  if ▽ = 1 return;
39  for u ∈ Q do
42     for e = (s→, u→) ∈ IN(u) do
41        if g_e^SP = 0 then
42           if d_s = d_u + w_e then
43              g_e^SP = 1;
44              δ_s = δ_s + 1;
45           end if
46           else if d_s > d_u + w_e then
47              d_s = d_u + w_e;
48              AdjustHeap(H, s, d_s);
49           end if
50        end if
51     end for
52  end for
end rhG_-^RR Ph1.
```

Figure 4 **Pseudocodes of Procedures $stdG_-^{RR}$ (Left) and $rhG_-^{RR}Ph1$ (Right)**

in Frigioni et al. (2000). This algorithm is simpler than G_-^{RR} because t^{SP} can be updated while Q is being identified. The set Q is identified with use of a heap H, and the procedure only requires one scan in each incoming arc into the affected nodes.

The reduced-heap variant requires a more complex implementation. To differentiate the nodes in U from those in Q, a variable *maxdiff* is used. The value $maxdiff_u$ of a node u is zero if the node belongs to Q, and *diff* if it belongs to U. The value *diff* corresponds to the amount of its distance that was decreased. Initially a node can belong to U, but as soon as its decreased distance is equal to ∇, it is inserted into Q. In this case, the node is not removed from U and

the corresponding value of *maxdiff* is set to zero. Next, only the nodes from U with a positive value of *maxdiff* are updated using a heap.

5.3. T_{-}^{KT}: Decrease Algorithm for Updating the Special Shortest-Path Tree Proposed by King and Thorup

T_{-}^{RR} is similar to T_{-}^{KT}, the algorithm for arc weight decrease that uses the special tree proposed by King and Thorup (2001). The main difference is that $stdT_{-}^{\mathrm{KT}}$ and rhT_{-}^{KT} have additional tests to guarantee that the correct tree is updated.

6. Reduced Heap vs. Standard Versions of Decrease Algorithms

In this section, we make a few observations comparing the reduced-heap (`rh`) variants with the respective standard (`std`) versions of the decrease algorithms. Figure 5 illustrates the heap size used by the decrease algorithms discussed in the previous section. The set Q of affected nodes is linked to the remaining part of the shortest-path graph by arc a, which had its weight decreased. Node t is the destination node. The shaded part of Q is composed of the nodes that were inserted into heap H. As we can see in the figure, in the standard algorithms all nodes $u \in Q$ are inserted into H, whereas for the reduced-heap variants no node $u \in Q$ is inserted into H. Nodes $u \in U$ are inserted into H for the standard and reduced-heap algorithms for random weight decrease, whereas H remains empty for unit decrement in the reduced-heap variant.

The reduced-heap variants require extra memory. Algorithm rhG_{-}^{RR} uses the $|V|$ array *degree* not required by its standard implementation. Algorithms rhT_{-}^{RR} and rhT_{-}^{KT} require two extra $|V|$ arrays, *maxdiff* and U, not used by their standard implementations.

7. Computational Results

In this section, we describe experimental results comparing the algorithms presented in this paper. The experiments were performed on a 1.7 GHz Intel Pentium IV computer with 256 MB of RAM, running RedHat Linux 8.0. The codes were written in C and compiled with the `gcc` compiler version 3.2, using the `-O3` optimization option. CPU times were measured with the system function `getrusage`.

The experiments were performed on nine classes of instances. The first class, `Internet`, is from traffic-engineering problems studied in Buriol et al. (2005) and Fortz and Thorup (2004). This class is composed of four subclasses: `att`, `hier`, `rand`, and `wax`. These subclasses were originally proposed in Fortz and Thorup (2004). The first subclass is taken from a real-world AT&T IP network with old data, whereas the other three are synthetic internetwork instances. In this paper, arc weights are integers, generated uniformly in the range $[1, \mu]$, where μ is 20 in the unit change experiments and 10^4 in the random weight change experiments. Instances from `Internet` class have multiple destination nodes; i.e., once an arc weight changes, the shortest-path graph (or tree) for each destination node must be updated. Instances from group `att` have 17 destinations nodes, whereas instances from subclasses `hier`, `rand`, and `was` have $|V|$ destination nodes.

The other eight classes are taken from Cherkassky et al. (1996). They constitute all of the instances in Cherkassky et al. (1996) that have only nonnegative arc weights. These instances were originally for single-source shortest paths, but all arcs were inverted (heads and tails swapped) and the source nodes were redefined as destination nodes. Unlike the instances of class `Internet`, which have multiple destination nodes, these instances have a single destination node. The instances from classes `Grid-SSquare-S`, `Grid-SWide`, and `Grid-SLong` are generated on rectangular grid networks with integer arc weights selected uniformly in the interval $[0, 10^4]$. The instances from class `Grid-PHard` are nonplanar and constructed with a complex layer structure. Weights are selected from a wide range of integers, with some multiplied by a function of the layer x-coordinate difference. Instances from classes `Rand-4`, `Rand-1:4`, and `Rand-Len` are constructed by first creating a Hamiltonian cycle. Arcs of the cycle have unit weight while the integer weights of the remaining arcs are chosen uniformly in the interval $[0, \mu]$. The size of μ is fixed at 10^4 for classes

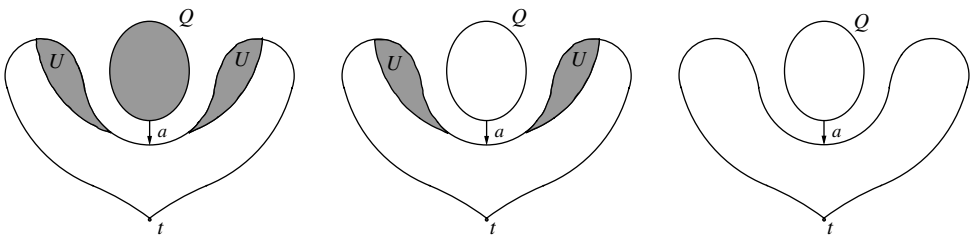

Figure 5 Heap Size Used by the Standard and Reduced-Heap Decrease Algorithms G_{-}^{RR}, T_{-}^{RR}, and T_{-}^{KT}

Notes. The left graph represents the standard algorithms. The middle graph represents the reduced-heap variants for random weight decrease. In the right graph, for the reduced-heap variants with unit decrement, the heap is empty.

Rand-4 and Rand-1:4, and varies in class Rand-Len. For Rand-Len, the integer arc weight is fixed at 1 for the first problem in the family, and selected uniformly in the interval $[0, \mu]$ for the others problems, with μ varying from 10 to 10^6. Because the weights of the path arcs are set to 1, the structure of the shortest-path tree changes as μ increases. For bigger values of μ, the path arcs are more likely to be in the tree and the tree is likely to be taller. The Acyc-Pos class is composed of acyclic networks. The weights of the path arcs are set to 1 and the remaining arc weights are selected in the interval $[0, 10^4]$. Because the dynamic shortest-path algorithms update a graph with positive weights, we set to 1 all weights that originally were zero.

Each class is composed of groups of instances, varying from four groups for class Rand-4 to 13 groups for class Internet. Each group consists of five instances generated with different random seeds. The seeds we used for generating the instances of Cherkassky et al. (1996) are those distributed with the generators. Instances from each grid graph group have identical arc sets, differing only with respect to arc weights. Instances from the remaining groups have identical dimensions, but differ both with respect to their arc sets as well as arc weights. The seeds used for generating the Internet class were $[1001, \ldots, 1005]$. Instances from each group in class Internet have identical arc sets and differ only with respect to arc weights.

For each instance, we ran 10^4 weight increases using an increase algorithm followed by 10^4 weight decreases, applied in the reverse order, using a decrease algorithm, resulting in the end in the original graph. Times are measured for the increase and decrease algorithms separately. The following pairs of increase/decrease algorithms were used: $(stdG_+^{\mathrm{RR}}, stdG_-^{\mathrm{RR}})$, $(rhG_+^{\mathrm{RR}}, rhG_-^{\mathrm{RR}})$, $(stdT_+^{\mathrm{RR}}, stdT_-^{\mathrm{RR}})$, $(rhT_+^{\mathrm{RR}}, rhT_-^{\mathrm{RR}})$, $(stdT_+^{\mathrm{KT}}, stdT_-^{\mathrm{KT}})$, and $(rhT_+^{\mathrm{KT}}, rhT_-^{\mathrm{KT}})$. For algorithms $stdT_+^{\mathrm{D}}$ and rhT_+^{D}, only the 10^4 weight increases were done because these shortest-path algorithms are semidynamic. Besides the dynamic shortest-path algorithms, results are presented for Dijkstra's algorithm. We use a simple mechanism that locally updates g^{SP} when there is no change in the distance label of any node and run Dijkstra's algorithm only if the distance of at least one node changes. The times reported for Dijkstra's algorithm are estimated. We run the algorithm for the first 100 changes and estimate the time for 10^4 changes multiplying the running time by 100. Tests were carried out to ensure that this estimate was accurate.

For each problem instance and range of weight increase permitted, all algorithms use the same sequence of arcs and weight-change values. These arc sequences and weight-change values are generated once by a separate program and stored in a file. The generation process is as follows. Given a shortest-path graph g_1^{SP} for a particular instance, let $\bar{w}$ be the average arc weight in the problem instance. The following procedure is repeated for $k = 1, \ldots, 10^4$ to produce a sequence of arcs $a_1, \ldots, a_{10^4}$ and weight change values $\triangle_1, \ldots, \triangle_{10^4}$ used in the simulations:

1. Select arc a_k at random from g_k^{SP}.
2. Choose at random a value $\triangle_k$ from the interval $[1, \bar{w}]$.
3. Add $\triangle_k$ to the current weight of arc a_k.
4. Recompute the shortest-path graph g_{k+1}^{SP}.

The experiments described in this section are grouped in two categories. Section 7.1 presents results for random weight changes, whereas §7.2 discusses results for unit weight changes. All algorithms used in the experiments as well as the complete set of results (for all instances of all classes presented) are in the Online Supplement.

7.1. Random Weight Changes

This section presents computational results for random weight changes applied on all instances of each group from each of the nine problem classes. The increase is an integer value chosen uniformly between one and the average arc weight from the original graph, e.g, $\triangle \in [1, \bar{w}]$. We consider the total CPU times (in seconds) for the 10^4 updates of the increase algorithms, the total CPU time for the 10^4 updates of the decrease algorithms, and the heap sizes for both algorithms. In the following, comparisons are done among algorithms to identify the best algorithm for diverse scenarios.

7.1.1. Improvement Obtained Using Reduced-Heap Algorithms. Running times were compared between the standard and reduced-heap versions of all algorithms described in this paper. For each dynamic shortest-path algorithm we compared the average ratio of the CPU times of the standard and reduced-heap versions. More precisely, let ts_i and tr_i be the average CPU times for the five instances of the ith group in the class for the standard and reduced-heap versions, respectively. Then, for each of the algorithms G_+^{RR}, G_-^{RR}, T_+^{RR}, T_-^{RR}, T_+^{KT}, T_-^{KT}, and T_+^{D}, we calculated the ratio as

$$\frac{\sum_{i=1}^{n_g} ts_i/tr_i}{n_g}, \tag{1}$$

where n_g is the number of groups in the class.

We observed that the reduced-heap algorithms run faster than their standard-version counterparts in almost all comparisons. For all four increase variants, the reduced-heap version was faster than the standard version for all nine instance classes. This was also true for the decrease algorithms, with the exception of the T_-^{KT} algorithm, for which the reduced-heap

version took longer, on average, in two of the nine classes.

The largest average gain was obtained for T_+^{D}, whose average ratio was 1.79 on the Internet class. On the other hand, the lowest ratio was 0.74, obtained for T_-^{KT}, on the same instance class. It was also observed that the increase algorithms benefited more from the reduced-heap idea than did the decrease algorithms. This behavior is expected because applying this idea in the decrease algorithms requires extra computational effort: the incoming arcs to nodes in set Q are scanned twice, whereas in the standard algorithm they are scanned only once. The standard increase algorithms took, on average, 29.25% longer than their reduced-heap counterparts, while for the decrease algorithms, the standard algorithms took, on average, 10.67% longer than their reduced-heap counterparts.

The average heap sizes for the standard and reduced-heap version of the dynamic shortest-path algorithms were also compared. We calculated the average ratio of the heap size of the standard and reduced-heap versions as

$$\frac{\sum_{i=1}^{n_g} hs_i/hr_i}{n_g}. \tag{2}$$

More precisely, let hs_i and hr_i be the average heap sizes for the five instances of the ith group in the class for the standard and reduced-heap versions, respectively. For all combinations of algorithm and instance class, the idea of reduced heap was successful in reducing the heap size. On average, the heap size was reduced by more than one third. In particular, the ratio for the class Internet was over 17 times, showing that, in the case of multiple shortest-path graphs, only a few graphs are affected by a single weight change (recalling that instances from this class have more than one destination node each).

7.1.2. Time Comparison Between Recomputing from Scratch (Dij) and Using a Dynamic Shortest-Path Graph Algorithm ($stdG^{\mathrm{RR}}$). Table 1 shows the ratios between the total times spent by Dijkstra's algorithm and $stdG^{\mathrm{RR}}$ for each class of instances. Each value in the table is the average ratio of the CPU times of the Dij and $stdG^{\mathrm{RR}}$ algorithms. The table entries are computed in a fashion similar to (1).

In this table, the CPU time considered for each algorithm is the sum of the CPU times of the increase and decrease phases. For instance classes Internet, Grid-SSquare-S, Grid-SWide, Rand-4, Rand-1:4, and Acyc-Pos, the ratio increases with instance size, while the opposite occurs in classes Grid-SLong and Grid-PHard.

The smallest ratio is 7.26, for group GR5 of class Grid-PHard. The largest is over 149,000, for group GR7 of class Grid-SWide. From this table, it is obvious that dynamic shortest-path algorithms should be used in place of Dijkstra's algorithm.

7.1.3. Time Comparison Between T^{KT} and T^{RR}. In this section, we show that, for the instances considered in this experiment, any gain that could be achieved while scanning the outgoing arcs in T^{KT} is washed out by the additional computational effort associated with maintaining the special tree proposed by King and Thorup (2001). We compute the average ratios of the CPU times of the T^{KT} and T^{RR} in a fashion similar to (1). For both standard and reduced-heap cases, results are compared for the increase and decrease algorithms. On average, for the standard and reduced-heap implementations, algorithm T^{RR} is faster than algorithm T^{KT}. For the standard algorithms, the performance is almost the same, whereas for reduced-heap variants, algorithm rhT^{KT}, on average, takes 15% and 17% longer than rhT^{RR}. The ratios for the reduced-heap algorithms are greater than those of the standard versions because for the reduced-heap variants even more computa-

Table 1 Ratio Between the Time Spent by the *Dij* and $stdG^{\mathrm{RR}}$ Algorithms for Updating 10^4 Random Weight Increases and 10^4 Random Weight Decreases on Each Group of All Classes of Instances

Group	Internet	Grid-SSquare-S	Grid-SWide	Grid-SLong	Grid-PHard	Rand-4	Rand-1:4	Rand-Len	Acyc-Pos
GR1	17.27	117.63	814.77	18.03	12.51	249.72	68.89	8,395.88	374.91
GR2	11.43	223.83	1,507.83	16.96	10.35	324.77	123.29	10,712.53	493.67
GR3	11.79	449.37	3,383.85	15.64	9.12	459.40	189.64	12,522.89	681.27
GR4	15.65	945.59	9,262.96	15.16	8.10	643.26	256.36	9,435.98	887.62
GR5	22.03	2,391.75	23,204.61	14.80	7.26	922.96		9,739.13	1,274.06
GR6	13.75		60,260.87	14.52	7.44	1,600.20			
GR7	18.42		149,371.17	13.64		2,715.88			
GR8	25.88					4,406.24			
GR9	26.10								
GR10	14.56								
GR11	15.53								
GR12	24.24								
GR13	27.63								

tional effort is needed to store the special shortest-path tree maintained by rhT^{KT} algorithm.

7.1.4. Time Comparison Between Algorithms T^{RR} and T^{D}. We next compare algorithms T^{RR} and T^{D} observing running times and heap sizes for the standard and reduced-heap implementations. Because T^{D} has only an increase version, there is no comparison for the decrease versions of the algorithms. We computed the average ratios between the values found by the algorithms T^{RR} and T^{D}. For the running-time comparison, the values are computed in a fashion similar to (1), whereas for the heap-size comparison the ratios are computed in a fashion similar to (2).

On average, T_{+}^{RR} is over 10% longer than T_{+}^{D} for both standard and reduced-heap implementations. With respect to heap size, the average heap of T_{+}^{D} was only 1% larger than that of T_{+}^{RR}. We conclude that for large ranges of weight changes, the larger heap size of T_{+}^{D} is compensated by its simpler identification of the set Q.

7.1.5. Time Comparison Between the Algorithms for Updating Shortest-Path Graphs G^{RR} and Trees T^{RR}. We next compare computational times spent to update shortest-path graphs and trees. For this comparison, we use CPU times of G^{RR} and T^{RR}. Among the three algorithms for updating shortest-path trees, T^{RR}, T^{KT}, and T^{D}, we selected T^{RR} because it was faster than T^{KT} in our experiments, and the running times of T^{D} are less predictable.

We compared results for the standard and the reduced-heap versions of the algorithms analyzing the average ratio of their CPU times, computed in a fashion similar to (1). For the increase algorithms, updating a t^{SP} is slightly faster than updating a g^{SP}. These results show that the extra computational effort G_{+}^{RR} needs to identify all shortest paths (last loop of the algorithm) is about the same effort T_{+}^{RR} needs to verify if a node has an alternative shortest path.

The decrease algorithms show more interesting results. In this case, to update a graph takes 52% and 41% longer than updating a shortest-path tree, for standard and reduced-heap algorithms, respectively. These results were expected because T_{-}^{RR} just scans once the links incoming to the affected nodes, whereas G_{-}^{RR} scans once the incoming and once the outgoing links of these nodes.

7.1.6. Time Comparison Between the Increase and Decrease Implementations of the Dynamic Algorithms. In this section, we explore the time difference between the increase and the decrease algorithms. We computed the average ratio, in a fashion similar to (1), of the CPU times of the algorithms $stdG^{RR}$, rhG^{RR}, $stdT^{RR}$, rhT^{RR}, $stdT^{KT}$, and rhT^{KT}. The average for each algorithm above, over the nine instance classes, are 1.44, 1.34, 2.12, 1.92, 2.07, and 1.68, respectively. Considering each algorithm and instance class separately, in only three of the 54 comparisons is the decrease algorithm faster than its increase counterpart. This performance was expected because the number of arcs scanned by the decrease algorithm is smaller than the corresponding number of arcs scanned by the increase algorithm. We observe that in the reduced-heap variants, the ratios are smaller than in the standard versions of the algorithms.

7.2. Unit Weight Changes

This section presents computational results for unit weight changes on the same instances used by the random-weight-changes experiment. The process of changes is the same, but now the increase and the decrease of the weight are equal to one, e.g., $\triangle = 1$ and $\triangledown = 1$. As before, we run each algorithm updating 10^4 arc weight changes. For this experiment, instances from class `Internet` have integer weights generated uniformly in the range $[1, 20]$. Instances from the other eight classes have the arc weights modified. The function `mod` was used to have weights in the interval $[1, 20]$. Each weight is computed as $w_a = 1 + (w_a \bmod 20)$.

We found two kinds of results. Either the heap sizes and times are much shorter than those generated by random weight changes, or they are about the same. For example, classes `Internet`, `Grid-SLong`, and `Rand-Len` have about the same heap size and running time, whereas the opposite was found in the experiments with classes `Grid-SSquare-S`, `Grid-PHard`, `Rand-4`, `Rand-1:4`, and `Acyc-Pos`.

In the remainder of this section, we discuss the experimental results for unit weight changes. Because the kinds of comparisons are similar to those presented for random weight changes, the ratios are computed similarly to those experiments.

Table 2 **Time Ratio Between the Standard and Reduced-Heap Implementations of the Algorithms for Updating 10^4 Unit Weight Changes**

Class	G_{+}^{RR}	G_{-}^{RR}	T_{+}^{RR}	T_{-}^{RR}	T_{+}^{KT}	T_{-}^{KT}	T_{+}^{D}
Internet	2.00	1.58	2.12	1.15	2.05	0.75	1.83
Grid-SSquare-S	1.58	0.91	1.50	1.17	1.43	0.97	1.33
Grid-SWide	1.73	1.51	1.87	1.29	1.74	1.05	1.63
Grid-SLong	4.50	2.45	5.11	2.32	4.02	1.20	4.21
Grid-PHard	2.40	1.07	2.41	1.18	2.19	0.99	1.51
Rand-4	2.36	1.39	2.19	1.20	2.10	1.16	1.86
Rand-1:4	2.98	1.04	2.38	1.00	2.49	1.01	1.47
Rand-Len	2.20	1.46	2.13	1.26	2.04	1.25	1.85
Acyc-Pos	2.87	1.15	2.39	1.04	2.52	1.05	2.05
Average	2.51	1.40	2.46	1.29	2.29	1.05	1.97

7.2.1. Improvement Obtained Using Reduced-Heap Algorithms. Table 2 compares times for standard and reduced-heap versions of the dynamic shortest-path algorithms.

On average, for each class of instances, the reduced-heap algorithms were able to reduce the computational time in almost all simulations. For all the increase algorithms, i.e., G_+^{RR}, T_+^{RR}, and T_+^{D}, the reduced-heap variants were faster. The ratio varied from 1.33 for T_+^{D} in class Grid-SSquare-S to 5.11 for T_+^{RR} in class Grid-SLong. Considering the decrease algorithms, only three ratios were not favorable to the respective reduced-heap variant. The last row of the table shows that, on average, all algorithms were able to reduce running times using the reduced-heap technique. The ratios varied from 1.05 for T_-^{KT} to 2.51 for G_+^{RR}. On average, for unit weight changes the gains obtained using the reduced-heap idea are bigger than those presented for random weight changes.

7.2.2. Time Comparison Between Recomputing from Scratch (Dij) and Using a Dynamic Shortest-Path Graph Algorithm ($stdG^{RR}$). Table 3 presents results for the comparison between the Dij and $stdG^{RR}$ algorithm. As expected, Dij takes much longer than the $stdG^{RR}$, with ratios varying from 10.33 on group GR2 from class Internet to 83,637.14 on group GR5 of class Grid-SSquare-S.

7.2.3. Comparison of Performance for the Dynamic Shortest-Paths Algorithms. The comparisons between algorithms G^{RR} and T^{RR}, and algorithms T^{RR} and T^{KT} for unit weight change and random weight change, presented similar results. However, this is not the case for the comparison between the performance of T_+^{RR} and T_+^{D}. In this last case, we observed some examples where T^{D} does not perform well when compared with other dynamic shortest-path algorithms. Because these experiments are applied to instances in a small range, T_+^{D} treats many unaffected nodes as though they were affected. The worst performance for this algorithm was observed on Rand-1:4. The algorithms took more than twice the time that T_+^{RR} spent in the standard implementation and 3.39 times more for the reduced-heap algorithm. Looking at the heap-size values, we can see that the heap size for the standard algorithm is 2.26 times larger, on average, than the heap of $stdT_+^{RR}$. For the reduced-heap implementation, while rhT_+^{RR} does not insert any node into the heap, rhT^{D} inserts up to 50% of the nodes inserted by $stdT^{D}$.

7.2.4. Time Comparison Between Increase and Decrease Implementations of the Dynamic Algorithms. For the standard algorithms, the results computed for the unit and random weight changes are similar. For the reduced-heap implementations, the decrease algorithms were faster, on average, for G^{RR} and T^{KT} and slightly slower for T^{RR}. The averages over the nine instances, for $stdG^{RR}$, rhG^{RR}, $stdT^{RR}$, rhT^{RR}, $stdT^{KT}$, and rhT^{KT}, are 1.39, 0.79, 1.90, 1.03, 1.95, and 0.92, respectively.

8. Concluding Remarks

This paper introduces a technique for reducing the heap size of DSP algorithms. This technique can be used for both increase and decrease algorithms. For unit arc-weight changes, the heaps are not used for all but one algorithm. Computational experiments were conducted for random and unit weight changes.

On average, all reduced-heap variants were faster than their corresponding standard implementations. For random weight changes, the speedups were up to a factor of 1.79, whereas for unit weight changes, the largest speedup was a factor of 5.11.

Comparing Dijkstra's algorithm with the Ramalingam and Reps algorithm showed that dynamic shortest-path algorithms are preferable. Speedups varied from 7.26 to 149,371.17 for random weight changes and from 10.33 to 86,031.82 for unit weight changes.

Table 3 **Ratio Between the Time Spent by Algorithms *Dij* and $stdG^{RR}$ for Updating 10^4 Unit Weight Increases and 10^4 Unit Weight Decreases on Each Group of All Classes of Instances**

Group	Internet	Grid-SSquare-S	Grid-SWide	Grid-SLong	Grid-PHard	Rand-4	Rand-1:4	Rand-Len	Acyc-Pos
GR1	15.38	1,383.33	719.67	18.01	362.75	521.29	61.67	7,747.92	672.92
GR2	10.33	2,722.22	1,261.63	17.64	452.21	1,092.08	182.05	11,416.35	1,430.67
GR3	10.91	6,962.96	3,104.17	16.33	468.15	2,427.81	730.40	9,340.97	1,955.40
GR4	15.59	26,753.57	8,348.11	15.48	709.06	3,117.64	976.06	12,482.10	3,230.47
GR5	20.58	83,637.14	21,329.46	14.91	770.68	9,422.34		9,030.76	8,749.76
GR6	14.47		44,769.75	15.00	773.36	10,424.23			
GR7	17.14		86,031.82	13.86		21,163.14			
GR8	22.84					65,180.58			
GR9	25.41								
GR10	13.27								
GR11	16.67								
GR12	24.31								
GR13	23.96								

The comparison between dynamic shortest-path algorithms has shown that, for the increase case, updating trees is slightly faster than updating graphs for random and unit weight changes. On the other hand, for the decrease algorithm to update a tree is about 50% faster than updating a graph.

For the instances considered in this paper, on average any gain that could be achieved while scanning the outgoing arcs in T^{KT} is washed out by the additional computational effort associated with maintaining the special tree proposed by King and Thorup (2001).

Algorithm T^{D} can be considered the fastest for graphs with weights selected in a wide range, but its performance is not predictable for instances with weights taken from a small range.

As a final conclusion, there is no dynamic shortest-path algorithm that can be considered the best for all situations. Clearly, however, any one of them is a better choice than recomputing the graph from scratch using Dijkstra's algorithm. The reduced-heap idea can be applied in both increase and decrease algorithms, even if in a few examples the reduced-heap variant took longer than the corresponding standard version. If the application does not need the shortest-path graph, updating a shortest-path tree is faster. The best choice would be the combination of the increase algorithm rhT_{+}^{D} with the decrease algorithm rhT_{-}^{RR}, if the instance is generated with weights from a wide range. Considering weights from a narrow range, the combination of rhT_{+}^{RR} and rhT_{-}^{RR} is recommended. Finally, we conclude that the performance of an algorithm depends largely on the instance, and also on its size.

Acknowledgments

The authors thank the reviewers and editors for the careful revisions and suggestions that much improved the final version of the paper.

References

Buriol, L. S., M. G. C. Resende, C. C. Ribeiro, M. Thorup. 2005. A hybrid genetic algorithm for the weight setting problem in OSPF/IS-IS routing. *Networks* **46** 36–56.

Cherkassky, B. V., A. V. Goldberg, T. Radzik. 1996. Shortest paths algorithms: Theory and experimental evaluation. *Math. Programming* **73** 129–174.

Demetrescu, C. 2001. Fully dynamic algorithms for path problems on directed graphs. PhD thesis, Department of Computer and Systems Science, University of Rome "La Sapienza," Rome.

Demetrescu, C., G. F. Italiano. 2001. Fully dynamic all pairs shortest with real edge weights. *Proc. 42nd Annual Sympos. Foundations Comput. Sci. (FOCS 2001)*. IEEE Computer Society, Washington, D.C., 260–267.

Demetrescu, C., G. F. Italiano. 2003. A new approach to dynamic all pairs shortest paths. *Proc. 35th Annual ACM Sympos. Theory Comput. (STOC'03)*. ACM Press, New York, 159–166.

Demetrescu, C., S. Emiliozzi, G. Italiano. 2003. Experimental analysis of dynamic all pairs shortest path algorithms. Technical report, Dipartimento di Informatica e Sistemistica, University of Rome "La Sapienza," Rome.

Demetrescu, C., D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. 2000. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. *Proc. Algorithm Engrg.: 4th Internat. Workshop, WAE 2000, Saarbrücken, Germany*. S. Näher, D. Wagner, eds. *Lecture Notes Compu. Sci.*, Vol. 1982. Springer, Berlin, 218–229.

Dionne, R. 1978. Etude et extension d'un algorithme de Murchland. *INFOR* **16** 132–146.

Even, S., Y. Shiloach. 1981. An on-line edge-deletion problem. *J. ACM* **28** 1–4.

Fortz, B., M. Thorup. 2004. Increasing internet capacity using local search. *Comput. Optim. Appl.* **29** 13–48.

Frigioni, D., A. Marchetti-Spaccamela, U. Nanni. 1996. Fully dynamic output bounded single source shortest path problem. *Proc. 7th Annual ACM-SIAM Sympos. Discrete Algorithms (SODA)*. SIAM, Philadelphia, 212–221.

Frigioni, D., A. Marchetti-Spaccamela, U. Nanni. 1998. Semi-dynamic algorithms for maintaining single-source shortest path trees. *Algorithmica* **22** 250–274.

Frigioni, D., A. Marchetti-Spaccamela, U. Nanni. 2000. Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms* **34** 351–381.

Frigioni, D., M. Ioffreda, U. Nanni, G. Pasqualone. 1998. Experimental analysis of dynamic algorithms for the single source shortest path problem. *ACM J. Experiment. Algorithmics* **3** Article 5, 1–20.

Fujishige, S. 1981. A note on the problem of updating shortest paths. *Networks* **11** 317–319.

Gallo, G. 1980. Reoptimization procedures in shortest path problems. *Rivista di Matematica per le Scienze Economiche e Sociali* **3** 3–13.

Goto, S., A. Sangiovanni-Vincentelli. 1978. A new shortest path updating algorithm. *Networks* **8** 341–372.

King, V., M. Thorup. 2001. A space saving trick for directed dynamic transitive closure and shortest path algorithms. *Proc. 7th Annual Internat. Comput. Combin. Conf. (COCOON). Lecture Notes Comput. Sci.*, Vol. 2108. Springer Verlag Italia, Milano, Italy, 268–277.

Murchland, J. D. 1970. A fixed matrix method for all shortest distances in a directed graph and for the inverse problem. PhD thesis, University of Karlsruhe, Karlsruhe, Germany.

Ramalingam, G., T. Reps. 1996a. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms* **21** 267–305.

Ramalingam, G., T. Reps. 1996b. On the computational complexity of dynamic graph problems. *Theoret. Comput. Sci.* **158** 233–277.