SPEEDING UP DYNAMIC SHORTEST PATH ALGORITHMS

L.S. BURIOL, M.G.C. RESENDE, AND M. THORUP

ABSTRACT. Dynamic shortest path algorithms update the shortest paths to take into account a change in an edge weight. This paper describes a new technique that allows the reduction of heap sizes used by several dynamic shortest path algorithms. For unit weight change, 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.

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 edge set, and $w \in \mathbb{R}^{|E|}$ is the edge 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 edge 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*. An algorithm is referred to as *fully-dynamic* if both weight increment (increase of arc weight by $\triangle$) and decrement (decrease of arc weight by $\triangledown$) are supported, and *semi-dynamic* incremental (decremental) if only increment (decrement) of weights is supported.

Many algorithms were proposed for solving this problem, but the algorithm of Ramalingam and Reps [20] (RR), seems to be the most used [1, 10, 13].

As previous work on algorithms for the dynamic shortest path problem, we refer to Murchland [19], Goto and Sangiovanni-Vincentelli [17] and Dionne [8]. Considering semi-dynamic decremental algorithms, previous work was done by Gallo [16] and Fujishige [15], while for the incremental case (arc deletion), we refer to Even and Shiloach [9].

Recently, Demetrescu, Frigioni, and Nanni [5] proposed a specialization of the Ramalingam and Reps algorithm for updating a shortest path tree, which is a revision of their previous work [13]. In the new version, they propose a fully dynamic algorithm, while

Date: September 19, 2003.

Key words and phrases. Shortest path algorithms, Dijkstra's algorithm, heaps, graphs, trees.

AT&T Labs Research Technical Report TD-5RJ8B.

in the earlier paper, a semi-dynamic incremental algorithm was proposed. Their specialized algorithm did not make use of the special tree proposed by King and Thorup [18]. In graphs where only a few affected nodes have alternative shortest paths, the incremental algorithm proposed by Demetrescu [3] usually has better performance.

For maintaining all pairs shortest paths in directed graphs with real-valued edge weights, we refer the fully dynamic algorithms of Demetrescu and Italiano [6, 7] and the experimental results in Demetrescu, Emiliozzi, and Italiano [4].

In this paper, we show that the Ramalingam and Reps algorithm is not the best algorithm for all applications. However, one of its main advantages is having good performance in most situations. First of all, it updates a shortest path graph, rather than a shortest path tree, although it can be easily specialized for updating a tree [5]. Even and Shiloach [9] proposed a semi-dynamic incremental algorithm that works in cascades, which can be computationally expensive for large arc weight increments. RR has good performance independent of $\triangle$. The semi-dynamic incremental algorithm of Demetrescu [3], 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, Marchetti-Spaccamela, and Nanni [12], that theoretically is better than RR, was usually outperformed by RR in computational testing [12].

Many theoretical studies of dynamic shortest path algorithms have been carried out, but few experimental results are known. Frigioni et al. [11] compared the algorithm of Ramalingam and Reps with the algorithm of Frigioni et al. [12], to update a 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 [21]. Demetrescu et al. [5] compare the incremental algorithms of [5, 11, 20] and a specialization of [5] described in [3]. 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 describes a new technique that allows the reduction of heap sizes in several dynamic shortest path algorithms. For unit weight change, the updates can be done without heaps. These reductions almost always reduce the computational times for these algorithms. In the incremental case, when this idea is applied to RR, it speeds up the standard RR for both large and small weight changes. 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.

Pseudo-codes for the algorithms are described for the single-destination problem. In these pseudo-codes, the heap function names follow [20]. These functions are:

- `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 this paper, we use the terms edge, arc, and link as synonyms. 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) if it refers to an incremental or decremental algorithm, respectively. When the name of the algorithm is used without the sign, it refers to both the incremental and

decremental cases or it is followed by the `Incr` or `Decr` indication. We add `rh` before the name to indicate the reduced heap variant. The terms `std` and `rh` are used to refer to the standard and reduced heap variants of the algorithms.

The paper is organized as follows. In Section 2, the data structures used to represent the graph and the solution (graph or tree) are presented and some implementation tricks described. Next, incremental dynamic shortest path algorithms, as well as the heap-reduction technique for this case, are presented in Section 3. A discussion of the standard and reduced heaps incremental algorithms with respect to heap size and memory usage is given in Section 4. This is followed, in Section 5, by the weight decrease case. A discussion of the `std` and `rh` decremental algorithms with respect to heap size and memory usage is given in Section 6. Computational results are reported in Section 7 and concluding remarks are made in Section 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 described.

2.1. **Data structures.** We first describe two data structures used in the implementations of the algorithms. The first data structure is used to represent the input graph, while the other represents the solution. Figure 1 shows an example with a graph and its corresponding data structures. In the left side of the figure, a graph with node indices is shown, while on the right, the corresponding data structures are given. The data structure on top are used for representing the input graph, while the data structures on the bottom represent a graph solution (`w`, `d`, g^{SP}, and δ) and a tree solution (`w`, `d`, and t^{SP}).

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 *i*-th position of the $|E|+1$-array `point` indicates the initial position in array `forward` of the list of outgoing links from node *i*. By assumption, the last position in array `forward` of the list of outgoing links 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 array `forward`. The *i*-th position of the $|E|+1$-array `rpoint` indicates the end position in array `reverse` of the list of incoming links into node *i*. By assumption, the last position in array `reverse` of the list of incoming links into node *i* is $\texttt{rpoint}[i+1]-1$.

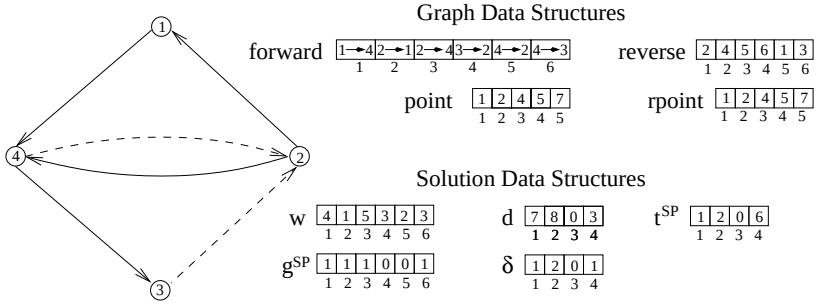

FIGURE 1. Graph representation.

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

In the case of trees, the i-th position of the $|V|$-array $\mathtt{t}^{\mathtt{SP}}$ indicates the index of the outgoing arc of node i in the shortest path tree.

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

2.2. **Implementation.** High-level pseudo-codes are given in Sections 3 and 5 for the algorithms. In this subsection, we indicate how some of the basic operations referred to in the pseudo-codes were implemented.

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

In the King and Thorup algorithm [18] presented in the next sections, a special tree is used. When the algorithm needs to determine if a node u has an alternative shortest path (not in the tree), the algorithm does not scan the entire set of outgoing nodes $\mathtt{OUT}(u)$. Rather, it scans edges $e = (\overrightarrow{u,v}) \in \mathtt{OUT}_e^{KT}(u) \subseteq \mathtt{OUT}(u)$. Let $e = (\overrightarrow{u,v}) \in \mathtt{OUT}(u)$ be the edge corresponding to entry $\mathtt{t}_{\mathtt{u}}^{\mathtt{SP}}$. The set $\mathtt{OUT}_e^{KT}(u)$ is made up of the arcs $\mathtt{forward}[e+1], \ldots, \mathtt{forward}[\mathtt{point}[u+1]-1]$.

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

3. Algorithms for Arc Weight Increase

In this section, we describe four incremental dynamic shortest path algorithms. First, we consider the Ramalingam and Reps algorithm for updating a shortest path graph [20] and its specialization for updating trees. In addition, we also use the special tree proposed by King and Thorup [18] in another specialization of the Ramalingam and Reps algorithm for updating trees. Finally, we consider the algorithm of Demetrescu [3] for trees. We refer to these standard (std) algorithms as G_+^{RR}, T_+^{RR}, T_+^{KT}, and T_+^{D}, respectively.

We introduce a technique to reduce the use of heaps and apply the technique to the four dynamic shortest path algorithms above. We refer to the reduced heap (rh) variants of these algorithms as rhG_+^{RR}, rhT_+^{RR}, rhT_+^{KT}, and rhT_+^{D}, respectively. These algorithms were originally designed to update a single-destination shortest path graph in directed graphs with real positive arc weights. 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 the algorithms. They receive as input the arc a, which has its 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 a shortest path graph, g^{SP} and δ are input. The reduced heap variants also receive as input the increment $\triangle$. As output they produce the updated input arrays.

All these algorithms are based on the same idea: a set Q of affected nodes is determined and the changes are only applied to these nodes and their incoming and outgoing arcs. Changes can occur in the distance labels of the nodes in Q and removal or addition of the

arcs incoming to or outgoing from these nodes. For the update, the standard versions of the incremental algorithms insert into heap H all nodes initially in set Q, while the reduced heap variants use heaps to update only a subset of Q. In the reduced heaps variants, the set Q is identified and, initially, all nodes $u \in Q$ have their distances increased by $\triangle$. After that, the actual increase of the distance label of node u (the tail node of arc a) is computed. Next, all nodes $u \in Q$ have their distances adjusted downwards by an appropriate amount $\bigtriangledown$. Only those nodes that have a shorter path, and can thus have their distance label further decreased, are inserted into heap H. As we will see in the experimental section of this paper, the number of nodes inserted into the heap is usually much smaller than the number of nodes in Q. In the next subsections, the pseudo-code of the standard and reduced heaps versions of these algorithms are presented.

3.1. **Incremental algorithm of Ramalingam and Reps for updating a shortest path graph.** G_+^{RR} updates a shortest path graph $g^{SP} = (V, E^{SP})$ when the weight of arc a is increased by $\triangle$. Figure 2 shows pseudo-code for G_+^{RR}.

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) and δ_u is decremented by one (line 3). If u, the tail node of arc a, has an alternative path (i.e. $\delta_u > 0$) 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 the set Q. For each node $u \in Q$, its distance is set to ∞ (line 7). For each incoming arc $e = (\overrightarrow{s,u})$ into node u, if $e \in g^{SP}$, e is removed from g^{SP} (line 10) and δ_s is updated (line 11). If s, the tail node of arc e, has no alternative path to the destination node, it is inserted into set Q (line 12).

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), in case its distance was decreased. The main objective of this loop is to update distances of nodes which have an alternative shorter path linking nodes outside set Q.

The loop in lines 22 to 36 updates the distances of nodes in Q using the heap H (lines 24 to 29) and restores g^{SP} (lines 30 to 35). Nodes u with minimal distance from the destination node are removed from H (line 23) and all arcs $e = (\overrightarrow{s,u})$ incoming into node u are traversed. In case d_s can be reduced, the new distance is set (line 26) and heap H is adjusted (line 27). In fact, this loop is Dijkstra's algorithm applied to the nodes in Q. Next, all outgoing arcs $e = (\overrightarrow{u,v})$ from node u are traversed. If d_u is the shortest distance to the destination node, then e is added to g^{SP} (line 32) and δ_u is updated (line 33).

Figure 3 shows pseudo-code for the rhG_+^{RR}, the reduced heap variant for the algorithm.

The first 15 lines are identical to the first 15 in Figure 2, with 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 16 to 42 are not executed and the heap H is not used.

Lines 17 to 21 calculate the amount that the distances of nodes $u \in Q$ will decrease. We denote by Q_0 the first element inserted into set Q, which is the tail node of arc a (inserted into Q in line 5). First, the current distance of node Q_0 is stored (line 17). Following, all arcs outgoing Q_0 are traversed and if an alternative shortest path is identified, d_{Q_0} is updated (line 19). Next, the amount $\bigtriangledown$ that the distance of node u will be decreased (line 21) is computed.

In the loop from line 22 to 32, all nodes from set Q, excluding node Q_0, have their distances decreased by $\bigtriangledown$ (line 23). Furthermore, the distances of nodes $u \in Q \setminus \{Q_0\}$ with a shortest path linking nodes $v \notin Q$ are updated. In the loop from line 25 to 30, all outgoing

```
procedure G_+^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, d);
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 G_+^RR.
```

FIGURE 2. Pseudo-code of procedure G_+^{RR}.

arcs from nodes $u \in Q \setminus \{Q_0\}$ are traversed. If a shortest path linking nodes outside set Q is found, the distance of node u is updated (line 27) and later (line 31) node u is inserted into heap H.

In the loop from line 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. In case node s has a shortest 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. **Incremental algorithm of Ramalingam and Reps for updating a shortest path tree.** The main difference with G_+^{RR} is that T_+^{RR} is specialized to update a shortest path tree rather than a shortest path graph. Instead of updating g^{SP}, this algorithm updates a tree t^{SP}: each node u stores an arc $a = (\overrightarrow{u,v})$, which is any arc in the outgoing adjacency list of node u belonging to a shortest path. The pseudo-code for the standard algorithm is described in Figure 4. In case arc $a = (\overrightarrow{u,v})$ is not in the shortest path tree, the algorithm stops (line 1). Otherwise, the outgoing arcs from node u are traversed. If one alternative shortest path is found, t_u^{SP} is updated (line 4) and the algorithm stops (line 5). Otherwise, set Q is initialized with node u (line 8).

The loop in lines 9 to 22 identifies the set of affected nodes. For each node $u \in Q$, d_u is set to ∞ (line 10), and each incoming arc $e = (\overrightarrow{s,u})$ is traversed. In case e is the outgoing link of node s stored in the shortest path tree, i.e. $t_s^{SP} = e$, all outgoing links from node s are scanned. If an alternative shortest path is found, t_s^{SP} is set (line 15) and the remaining arcs outgoing node s are not scanned (line 16). Otherwise, if an alternative path was not found, s is inserted into set Q (line 19). The loop in lines 23 to 31 updates the distances of nodes $u \in Q$ which have a shortest path linking nodes outside Q. Each arc $e = (\overrightarrow{u,v})$ outgoing from node $u \in Q$ is traversed. If d_u can be decreased, its new distance is set (line 26) and t_u^{SP} is updated (line 27). All nodes $u \in Q$ whose distances have decreased are inserted into heap H (line 30).

The loop in lines 32 to 41 updates the distances of the nodes in Q using heap H. Nodes with minimum distance to the destination node are removed from H and all incoming arcs $e = (\overrightarrow{s,u})$ are traversed. In case d_s can be decreased, the new distance is set (line 36), t_s^{SP} is updated (line 37), and heap H is adjusted (line 38).

In the reduced heaps variant, described in Figure 5, the increase amount $\triangle$ is an input parameter. The first 22 lines of the pseudo-code are identical to the standard T_+^{RR}, with exception of line 10, that in the reduced heaps variant d_u is increased by $\triangle$ instead of set to ∞. As in G_+^{RR}, the idea is to increment the distance of nodes $u \in Q$ by $\triangle$, and later decrement them by $\bigtriangledown$. In the case of unit increment ($\triangle = 1$) the algorithm stops in line 23. In lines 24 to 31, the value of $\bigtriangledown$ is calculated. In line 28, the outgoing link of node u is stored in t^{SP}, since the shortest path tree can be restored while the distances are updated, as was not the case for G_+^{RR}, whose last loop has this purpose.

The loop in lines 32 to 43 updates the distances of nodes $u \in Q \setminus \{Q_0\}$ with a shortest path not traversing arc a. Recall that Q_0 is the first node inserted into set Q (line 8). Each node $u \in Q$, with the exception of node Q_0, has its distance decremented by $\bigtriangledown$ (line 33). Each arc $e = (\overrightarrow{u,v})$ outgoing from node u is traversed and if d_u can be decreased, then d_u and t_u^{SP} are updated (lines 37 and 38, respectively), and later (line 42) node u is inserted into heap H.

The loop in lines 44 to 53 is the same as the loop in lines 32 to 41 in the pseudo-code of algorithm T_+^{RR} (Figure 4).

3.3. **The incremental algorithm for updating the special tree proposed by King and Thorup.** The main difference with T_+^{RR} is that T_+^{KT} updates a special shortest path tree. In this special tree, t^{SP} stores for each node u the first link $a = (\overrightarrow{u,v})$ in the outgoing adjacency list of node u, belonging to a shortest path. The advantage of storing this special tree is to be able to find an alternative path with no need to explore all outgoing arcs in the set

```
procedure rhG+RR(a = (u→,v→), w, d, δ, gSP, △)
1     if gaSP = 0 return;
2     gaSP = 0;
3     δu = δu − 1;
4     if δu > 0 then return;
5     Q = {u};
6     for u ∈ Q do
7           du = du + △;
8           for e = (s→,u→) ∈ IN(u) do
9                 if geSP = 1 then
10                      geSP = 0;
11                      δs = δs − 1;
12                      if δs = 0 then Q = Q ∪ {s};
13                end if
14          end for
15    end for
16    if △ > 1 then
17          dist = dQ0;
18          for e = (Q0→,v→) ∈ OUT(Q0) do
19                if dQ0 > dv + we then dQ0 = dv + we;
20          end for
21          ▽ = dist − dQ0;
22          for u ∈ Q \ Q0 do
23                du = du − ▽;
24                flag = 0;
25                for e = (u→,v→) ∈ OUT(u) do
26                      if du > dv + we then
27                            du = dv + we;
28                            flag = 1;
29                      end if
30                end for
31                if flag = 1 then InsertIntoHeap(H, u, du);
32          end for
33          while HeapSize(H) > 0 do
34                u = FindAndDeleteMin(H, d);
35                for e = (s→,u→) ∈ IN(u) do
36                      if ds > du + we then
37                            ds = du + we;
38                            AdjustHeap(H, s, ds);
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 du = we + dv then
46                      geSP = 1;
47                      δu = δu + 1;
48                end if
49          end for
50    end for
end rhG+RR.
```

FIGURE 3. Pseudo-code of procedure rhG_+^{RR}.

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

FIGURE 4. Pseudo-code of procedure T_+^{RR}.

```
procedure rhT+^RR(a = (u→,v), w, d, t^SP, △)
1     if t_u^SP ≠ a return;
2     for e = (u→,v) ∈ OUT(u) do
3           if d_u = d_v + w_e then
4                 t_u^SP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          d_u = d_u + △;
11          for e = (s→,u) ∈ IN(u) do
12                if t_s^SP = e do
13                      for a = (s→,v) ∈ OUT(s) do
14                            if d_s = d_v + w_a then
15                                  t_s^SP = A;
16                                  break;
17                            end if
18                      end for
19                      if t_s^SP = e then Q = Q ∪ {s};
20                end if
21          end for
22    end for
23    if △ = 1 then return;
24    dist = d_Q0;
25    for e = (Q0→,v) ∈ OUT(Q0) do
26          if d_Q0 > d_v + w_e then
27                d_Q0 = d_v + w_e;
28                t_Q0^SP = e;
29          end if
30    end for
31    ▽ = dist − d_Q0;
32    for u ∈ Q \ Q0 do
33          d_u = d_u − ▽;
34          flag = 0;
35          for e = (u→,v) ∈ OUT(u) do
36                if d_u > d_v + w_e then
37                      d_u = d_v + w_e;
38                      t_u^SP = e;
39                      flag = 1;
40                end if
41          end for
42          if flag = 1 then InsertIntoHeap(H, u, d_u);
43    end for
44    while HeapSize(H) > 0 do
45          u = FindAndDeleteMin(H, d);
46          for e = (s→,u) ∈ IN(u) do
47                if d_s > d_u + w_e then
48                      d_s = d_u + w_e;
49                      t_s^SP = e;
50                      AdjustHeap(H, s, d_s);
51                end if
52          end for
53    end while
end rhT+^RR.
```

FIGURE 5. Pseudo-code of procedure rhT_+^{RR}.

$\texttt{OUT}(u)$. For example, in Figure 1, t_2^{SP} must equal two for T^{KT}, but it could equal two or three for T^{RR}. The pseudo-code for this algorithm is described in Figure 6.

In case arc $a \notin t^{SP}$, the algorithm stops (line 1). Otherwise, the existence of an alternative shortest path is verified traversing the arcs in the outgoing adjacency list of node u, starting at the next position after arc a. In the pseudo-code of Figure 6, we denote by $\texttt{OUT}^{KT}(u) \subseteq \texttt{OUT}(u)$ the set of arcs which are located after t_u^{SP} in the outgoing adjacency list of node u. If another shortest path is found, arc e is stored in t_u^{SP} (line 4) and the algorithm stops (line 5). Otherwise, set Q is initialized with node u (line 8).

The loop in lines 9 to 22 identifies the remaining affected nodes of t^{SP} and adds them to set Q. For each node $u \in Q$, its distance is set to ∞ (line 10), and all arcs $e = (\overrightarrow{s,u})$ incoming into node u are scanned. In case $t_s^{SP} = e$, the arcs in the set $\texttt{OUT}^{KT}(s)$ are traversed. If node s has an alternative shortest path, t_s^{SP} is updated (line 15) and the subsequent arcs in $\texttt{OUT}^{KT}(u)$ are not scanned (line 16). Otherwise, if an alternative shortest path for s is not found, node s is inserted into set Q (line 19).

The loop in lines 23 to 31 scans all links e outgoing each node $u \in Q$. The distances and t^{SP} of nodes $u \in Q$ are updated (lines 26 and 27), and $t_u^{SP} = e$ (line 27). In case d_u is decremented, node u is inserted into heap H (line 30). The main objective of this loop is to update distances of nodes $u \in Q$ which have an alternative shorter path linking nodes $v \notin Q$.

The loop in lines 32 to 42 updates the distances of the nodes in Q using heap H. This loop is identical to the loop in lines 32 to 41 of T_+^{RR}. Furthermore, in case arc $e = (\overrightarrow{s,u})$ is in a shortest path, i.e. $d_s = d_u + w_e$, and its position in the outgoing adjacency list is prior to the current shortest path information stored, i.e. $t_s^{SP} > e$, then t_s^{SP} is updated (line 40).

The reduced heap variant rhT_+^{KT} is executed in two phases. The first phase identifies set Q, the set of affected nodes. The second phase updates distances of nodes $u \in Q$ and restores the shortest path tree t^{SP}.

The first phase is not presented in pseudo-code since it is identical to the first 22 lines of the standard T_+^{KT}. The only difference is that in rhT_+^{KT} the distance of node u (line 10) is incremented by $\triangle$, while in T_+^{KT} d_u is set to ∞. Figure 7 shows the pseudo-code for the second phase $rhT_+^{KT}Ph2$ of rhT_+^{KT}.

The loop in lines 1 to 11 is only executed for unit increment. Considering unit increment, the affected nodes already had their distances increased by one in the first phase of the algorithm. The shortest path tree t^{SP} only needs to be restored to maintain the special tree. We denote by $\texttt{OUT}^*(u) \subseteq \texttt{OUT}(u)$ the set of arcs outgoing node u prior to the arc t_u^{SP}, i.e. $\texttt{OUT}^*(u) = OUT(u) \setminus \texttt{OUT}^{KT}(u) \setminus \{t_u^{SP}\}$. For each node $u \in Q$, the set of outgoing links $\texttt{OUT}^*(u)$ is traversed. In case an arc $e = (\overrightarrow{u,v})$ is in the shortest path tree, t_u^{SP} is updated (line 5) and the algorithm stops (line 10). Note that in the case of unit weight increase, the reduced heap variant rhT_+^{KT} stops (line 10) without using heaps.

In lines 12 to 19, each arc $e = (\overrightarrow{Q_0,v})$ outgoing from node Q_0, the head node of arc a which had its weight originally increased, is traversed. This loop is done in backward order to allow the correct update of t^{SP} without testing if $t_{Q_0}^{SP} > e$. To do this, however, we test if $d_{Q_0} \geq d_v + w_e$. If the inequality holds, then d_{Q_0} and $t_{Q_0}^{SP}$ are updated. The value of $\bigtriangledown$ is calculated in line 19.

The loop in lines 20 to 32 has the objective of updating the distances and t^{SP} of nodes $u \in Q$ which have a shortest path linking nodes outside Q. Again, the backward order for loop is used to avoid testing if $t_s^{SP} > e$ in line 29.

The Loop in lines 33 to 43 is identical to lines 32 to 42 in Figure 6.

```
procedure T_+^KT(a = (u→,v→), w, d, t^SP)
1     if t_u^SP ≠ a then return;
2     for e = (u→,v→) ∈ OUT^KT(u) do
3           if d_u = d_v + w_e then
4                 t_u^SP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          d_u = ∞;
11          for e = (s→,u→) ∈ IN(u) do
12                if t_s^SP = e do
13                      for a = (s→,v→) ∈ OUT^KT(s) do
14                            if d_s = d_v + w_a then
15                                  t_s^SP = a;
16                                  break;
17                            end if
18                      end for
19                      if t_s^SP = e then Q = Q ∪ {s};
20                end if
21          end for
22    end for
23    for u ∈ Q do
24          for e = (u→,v→) ∈ OUT(u) do
25                if d_u > d_v + w_e then
26                      d_u = d_v + w_e;
27                      t_u^SP = e;
28                end if
29          end for
30          if d_u ≠ ∞ then InsertIntoHeap(H,u,d_u);
31    end for
32    while HeapSize(H) > 0 do
33          u = FindAndDeleteMin(H,d);
34          for e = (s→,u→) ∈ IN(u) do
35                if d_s > d_u + w_e then
36                      d_s = d_u + w_e;
37                      t_s^SP = e;
38                      AdjustHeap(H,s,d_s);
39                end if
40                else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
41          end for
42    end while
end T_+^KT.
```

FIGURE 6. Pseudo-code of procedure T_+^{KT}.

```
procedure rhT_+^KT Ph2(a = (u→,v), w, d, t^SP, △)
1    if △ = 1 then
2        for u ∈ Q do
3            for e = (u→,v) ∈ OUT*(u) do
4                if d_u = d_v + w_e then
5                    t_u^SP = e;
6                    break;
7                end if
8            end for
9        end for
10       return;
11   end if
12   dist = d_{Q_0};
13   backward for e = (Q_0→,v) ∈ OUT(Q_0) do
14       if d_{Q_0} ≥ d_v + w_e then
15           d_{Q_0} = d_v + w_e;
16           t_{Q_0}^SP = e;
17       end if
18   end for
19   ▽ = dist − d_{Q_0};
20   for u ∈ Q \ Q_0 do
21       d_u = d_u − ▽;
22       flag = 0;
23       backward for e = (u→,v) ∈ OUT(u) do
24           if d_u > d_v + w_e then
25               d_u = d_v + w_e;
26               t_u^SP = e;
27               flag = 1;
28           end if
29           else if d_u = d_v + w_e then t_u^SP = e;
30       end for
31       if flag = 1 InsertIntoHeap(H, u, d_u);
32   end for
33   while HeapSize(H) > 0 do
34       u = FindAndDeleteMin(H, d);
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               t_s^SP = e;
39               AdjustHeap(H, s, d_s);
40           end if
41           else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
42       end for
43   end while
end rhT_+^KT.
```

FIGURE 7. Pseudo-code of second phase of rhT_+^{KT}.

3.4. **Incremental algorithm of Demetrescu.** 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. However, $|Q^{RR}| = |Q^{KT}| \leq |Q^D|$. Supposing arc $a = (\overrightarrow{u,v})$ had its weight increased by $\triangle$, set Q^{RR} is composed only of the nodes such that all their shortest paths traverse arc a, while set Q^D is composed of those nodes but also can 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 more than one shortest path $|Q^D| \approx |Q^{RR}|$, with the advantage of Q^D being identifi ed using a simpler mechanism.

The pseudo-code of this algorithm is described in Figure 8. This pseudo-code is identical to algorithm T_+^{RR}, differing only in the loop which identifi es the set Q. The loop in lines 9 to 14 identifi es the *affected* nodes in a manner that is simpler to implement than in all incremental algorithms presented so far in this paper. However, some of the nodes $u \in Q$ may be unaffected but treated as though they were. For each node $u \in Q$, d_u is set to ∞ (line 10), and each incoming arc e is traversed. In case e is the outgoing link 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 edge weights is spread out and the probability of ties is low.

In the reduced heaps variant, described in Figure 9, the increase amount $\triangle$ is an input parameter. This algorithm is identical to rhT_+^{RR}, only differing in how they identify set Q and rhT_+^D does not stop in the case of unit increment like rhT_+^{RR} does in line 23 of pseudo-code of Figure 5. The reduced heaps variant identifi es set Q in the same manner the standard algorithm does. Note that in the case of unit weight increase only the unaffected nodes from set Q are inserted into heap H by rhT_+^D, while rhT_+^{RR} and rhT_+^{KT} do not insert any node into H.

4. Reduced heap vs. standard versions of the incremental algorithms

In this section, we make a few observations comparing the reduced heaps variants with the respective standard versions of the incremental algorithms. Figures 10 and 11 illustrate the heap size used by the incremental algorithms discussed in the previous section. The set Q of affected nodes is linked to the remaining part of the graph by arc a, which had its weight increased. Node t is the destination node. The shaded part of set Q is composed of the nodes that were inserted into heap H.

As we can see in the fi gures, in the case of unit increment for G_+^{RR}, T_+^{RR}, and T_+^{KT}, heap H is empty, while for T_+^D some nodes $u \in Q$ are inserted into H. These nodes are the ones treated as affected while in fact they were not. Due to this, comparing any single case of algorithms `std`, `rh`, and `unit increment`, $Q^D \supseteq Q^{RR} = Q^{KT}$. For all of these algorithms, $H^{std} \supseteq H^{rh}_{random} \supseteq H^{rh}_{unit}$.

In terms of memory, no additional space is required to implement the reduced heap variants of all incremental algorithms.

5. Algorithms for Arc Weight Decrease

In this section, we describe three algorithms for arc weight decrease, as well the idea of reduced heaps applied to them. The weight can be decreased by any amount. We refer to these algorithms as G_-^{RR}, T_-^{RR}, and T_-^{KT}. The reduced heap variants of these algorithms are denoted by rhG_-^{RR}, rhT_-^{RR}, and rhT_-^{KT}.

Let $a = (\overrightarrow{u,v}) \in E$ be the arc whose weight is to be decreased. For these algorithms, the set Q is the set of nodes with at least one shortest path traversing arc a. A different set of

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

FIGURE 8. Pseudo-code of procedure T_+^D.

nodes, U, is composed of nodes whose shortest paths originally did not traverse arc a, but do after the decrement of w_a. The standard algorithms insert into the heap H all nodes in sets Q and U. The idea of reduced heaps is applied to avoid the use of heaps for updating nodes in Q and only inserts into the heap nodes from set U.

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

```
procedure rhT+D(a = (u→,v), △, w, d, tSP)
1     if tuSP ≠ a return;
2     for e = (u→,v) ∈ OUT(u) do
3           if du = dv + we then
4                 tuSP = e;
5                 return;
6           end if
7     end for
8     Q = {u};
9     for u ∈ Q do
10          du = du + △;
11          for e = (s→,u) ∈ IN(u) do
12                if tsSP = e then Q = Q ∪ {s};
13          end for
14    end for
15    dist = dQ0;
16    for e = (Q0→,v) ∈ OUT(Q0) do
17          if dQ0 > dv + we then
18                dQ0 = dv − we;
19                tQ0SP = e;
20          end if
21    end for
22    ▽ = dist − dQ0;
23    for u ∈ Q \ Q0 do
24          du = du − ▽;
25          flag = 0;
26          for e = (u→,v) ∈ OUT(u) do
27                if du > dv + we then
28                      du = dv + we;
29                      tuSP = e;
30                      flag = 1;
31                end if
32          end for
33          if flag = 1 then InsertIntoHeap(H, u, du);
34    end for
35    while HeapSize(H) > 0 do
36          u = FindAndDeleteMin(H, d);
37          for e = (s→,u) ∈ IN(u) do
38                if ds > du + we then
39                      ds = du + we;
40                      tsSP = e;
41                      AdjustHeap(H, s, ds);
42                end if
43          end for
44    end while
end T+D.
```

FIGURE 9. Pseudo-code of procedure rhT_+^D.

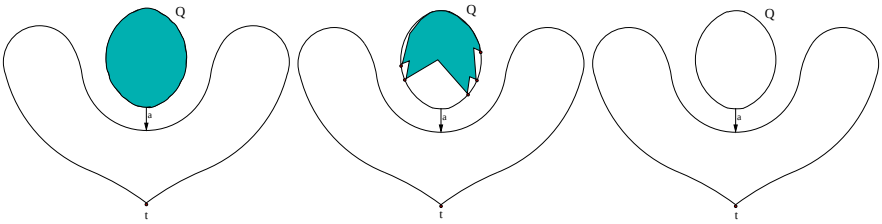

FIGURE 10. Heap size used by the incremental algorithms G_+^{RR}, T_+^{RR}, and T_+^{KT} and their reduced heap variants. Graph on the left represents the standard algorithms. The graph in the middle represents the reduced heaps variants for random weight increase. On the right, for the reduced heaps variants with unit weight increment, the heap is empty.

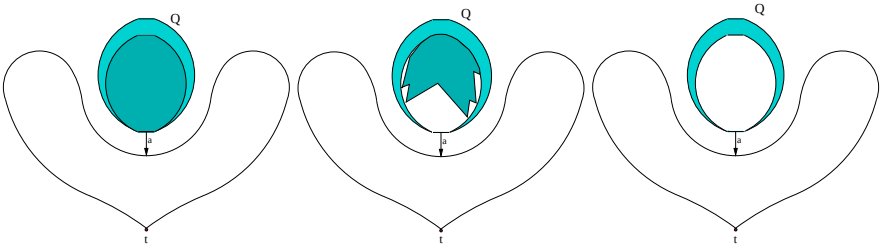

FIGURE 11. Heap size used by the incremental algorithms T_+^D and rhT_+^D. Graph on the left represents the standard algorithm. The graph in the middle represents the reduced heaps variant for random weight increase. On the right, for the reduced heaps variant with unit weight increment, the heap contains only unaffected nodes that were inserted into set Q.

nodes $u \in U$. In the next subsections, we present pseudo-codes of the standard and reduced heaps versions of the decrease algorithms.

5.1. **Ramalingam and Reps arc weight decrease for updating a shortest path graph.** The Ramalingam and Reps arc weight decrease algorithm [20] updates a shortest path graph considering an arc weight decrease of any amount. The pseudo-code for this procedure is given in Figure 12.

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} (line 3), the variable δ_u is updated (line 4), and the algorithm stops (line 5). Otherwise, the distance d_u is updated (line 7) and heap H is initialized with node u (line 8).

In the loop in lines 9 to 29, nodes from H are removed, one by one, from the smallest to the largest distance to the destination node, and their respective distances and incoming/outgoing arcs are updated. Node u is remove from H (line 10) and δ_u is set to zero (line 11). All outgoing arcs from node u are scanned and inserted into g^{SP} (line 15), in case they have the minimum distance to destination node. Otherwise, $g_e^{SP} = 0$ (line 17). Next, each incoming arc $e = (\overrightarrow{s,u})$ into node $u \in Q$ is traversed. If d_s can be decreased, the new distance is set (line 21) and H adjusted (line 22). If arc e is not in the current shortest path

```
procedure G_-^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    InserIntoHeap(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 G_-^RR.
```

FIGURE 12. Pseudo-code of procedure G_{-}^{RR}.

graph and $d_s = d_u + w_e$, then the loop in lines 24 to 27 inserts arc e in g^{SP} (line 25) and updates δ_s (line 26).

We now consider the Ramalingam and Reps weight decrease algorithm with reduced heaps. The first phase of rhG_{-}^{RR} is described in Figure 13. The first six lines of this pseudo-code are identical to the first six lines of the standard version algorithm. In line 7 the amount $\bigtriangledown$, by which the distance of node u will be decreased, is calculated. The distance of node u is decreased by $\bigtriangledown$ (line 8), it is inserted into set Q (line 9) and $degree_u$ is initialized (line 10). The array *degree* is used to avoid inserting a node u into set Q more than once. Furthermore, *degree* is used to identify nodes in Q having alternative shortest paths not traversing arc a.

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. For each node $u \in Q$, all incoming links $e = (\overrightarrow{s,u})$ are scanned. If e is in the shortest path graph and if $degree_s = 0$, then vertex s

has its distance decreased by $\bigtriangledown$ (line 15), s is inserted into set Q (line 16), and $degree_s$ receives the value $\delta_s - 1$ (line 17). Otherwise, if $e \in g^{SP}$ but $degree_s > 0$, then $degree_s$ is decremented by one unit (line 19). In the case of unit decrement, g^{SP} and δ are updated in lines 22 and 23, respectively.

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. If one or more alternative paths from node u are detected, then $degree_u$ is reset to 0 (line 29), and in the loop in line 30 to 35, all outgoing links $e = (\overrightarrow{u,v})$ from u are scanned. If arc e is no longer in the shortest path, it is removed (line 32) and the number of outgoing links from u (δ_u) is updated (line 33).

In the case of unit decrement, the algorithm stops in line 38.

In the loop from line 39 to 52, all incoming arcs $e = (\overrightarrow{s,u})$ into node $u \in Q$ are scanned. The test in line 41 permits the algorithm to avoid testing edges $e = (\overrightarrow{s,u})$ where $s \in Q$ and $e \in g^{SP}$. We chose not to maintain an indicator array of nodes in set Q, which would allow us to avoid testing edges $e = (\overrightarrow{s,u})$ where $s \in Q$ and $e \notin g^{SP}$ because any savings associated with its use does not appear to outweigh the computational effort associated with its housekeeping. In lines 42 to 45, if node s has an alternative shortest path traversing arc e, e is inserted into g^{SP} (line 43) and δ is updated (line 44). If d_s decreases transversing arc a, then it is updated (line 47) and H adjusted (line 48).

Recall that phase 1 of the algorithm determined the set Q, containing all nodes that have at least one shortest path traversing arc a, and updated the part of the graph which 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 the 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 a node in Q. This procedure is identical to the pseudo-code in lines 9 to 29 of Figure 12.

5.2. **Ramalingam and Reps weight decrease 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 in [14]. Algorithm T_-^{RR} is described in Figure 14.

Let arc $a = (\overrightarrow{u,v})$ be the arc whose weight is decreased. In case the distance of node u does not change, the algorithm stops (line 1). Otherwise, d_u and t_u^{SP} are updated in lines 2 and 3, respectively, and heap H is initialized with node u (line 4). Nodes are removed, one by one, from heap H (line 6), and each incoming arc $e = (\overrightarrow{s,u})$ into node u is scanned. If d_s decreases, d_s and t_s^{SP} are updated in lines 9 and 10, respectively, and H is adjusted (line 11).

Figure 15 shows the pseudo-code for the reduced heap variant rhT_-^{RR}. The algorithm initially identifies the sets Q and U. Next, the nodes in U are inserted into the heap and the remaining part of graph is updated. First, the decrease amount $\bigtriangledown$ is calculated (line 1). If $\bigtriangledown \leq 0$ the algorithm stops (line 2) since the decrease amount $\bigtriangledown$ is not enough to decrease the distance of node u. Otherwise, arc a is in the new shortest path from node u to the destination node. The distance d_u and t_u^{SP} are updated (lines 3 and 4, respectively), t_u^{SP} is set (line 4) and u is inserted into set Q (line 5). The set U is initialized empty in line 6.

The $|V|$-array *maxdiff* is used to store the maximum decrease amount for nodes $u \in U$. The loop in lines 7 to 24 identifies nodes from sets Q and U. For each node $u \in Q$, all incoming links $e = (\overrightarrow{s,u})$ are scanned. The amount *diff* that d_s decreases when scanning arc

```
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 13. Pseudo-code of procedure $rhG_{-}^{RR}Ph1$.

```
procedure T_-^RR(a = (u→,v→), w, d, δ, g^SP)
1     if d_u ≤ d_v + w_a then return;
2     d_u = d_v + w_a;
3     t_u^SP = a;
4     InserIntoHeap(H, u, d_u);
5     while HeapSize(H) > 0 do
6           u =FindAndDeleteMin(H, d);
7           for e = (s→,u→) ∈ IN(u) do
8                 if d_s > d_u + w_e then
9                       d_s = d_u + w_e;
10                      t_s^SP = e;
11                      AdjustHeap(H, s, d_s);
12                end if
13          end for
14    end while
end T_-^RR.
```

FIGURE 14. Pseudo-code of procedure T_-^{RR}.

e, is calculated in line 9. Only positive values of *diff* are considered (line 10). If $diff = \bigtriangledown$ then node *s* has a shortest path traversing arc *a* and *u* is identified as belonging to set *Q*. Once a node *s* is identified as belonging to set *Q*, its distance is updated (line 12), t_u^{SP} is set (line 13), node *s* is inserted into *Q* and and its *maxdiff* is reset to zero (line 15). Otherwise, if *s* is not in *U*, it is inserted in *U* (line 18). If *diff* is larger than $maxdiff_s$, $maxdiff_s$ is updated with *diff* (line 19) and t_u^{SP} is updated (line 20). In case of unit decrement, i.e. $\bigtriangledown = 1$, the algorithm stops in line 25. Otherwise, *maxdiff* of nodes $u \in U$ are tested. Note that the nodes *u* that were initially added to set *U* can later be identified as belonging to set *Q*. In this case, node *u* is inserted into set *Q* and not removed from set *U*, since the computational effort for removing it can be expensive and these nodes can be identified later using the array *maxdiff*. For each node *u* previously inserted into set *U*, the algorithm makes use of the variable

$$maxdiff_u = \begin{cases} 0, & \text{if } u \in U \cap Q; \\ maxdiff_u > 0, & \text{if } u \in U \setminus (U \cap Q), \end{cases}$$

to determine if node $u \in U \setminus (U \cap Q)$. If $maxdiff_u > 0$, then it is the amount by which the distance of *u* will be reduced. Nodes $u \in U \cap Q$ are discarded in the test of line 27. In case $maxdiff_u > 0$, the distances of node *u* are updated (line 28), its *maxdiff* value is reset (line 29), and node *u* is inserted into *H* (line 30). The loop in lines 33 to 42 updates t^{SP} and the distances for nodes $u \in U$. Nodes are removed, one by one, from heap *H*, and each incoming arc $e = (\overrightarrow{s,u})$ is scanned. In case d_s can be decreased, d_s and t_s^{SP} are updated (lines 37 and 38, respectively), and *H* in adjusted (line 39).

5.3. **The decremental algorithm for updating the special tree proposed by King and Thorup.** We present in this subsection, an algorithm for arc weight decrease that uses the special tree proposed by King and Thorup [18]. The difference from T_-^{RR} is that T_-^{KT} updates a special shortest path tree. Algorithm T_-^{KT} stores, for each node, the first arc from the outgoing adjacency list belonging to one of the shortest paths. Due to this simple

```
procedure rhT_-^RR(a = (u→,v→), w_E, d_V, δ_V, g^SP)
1     ▽ = d_u − d_v − w_a;
2     if ▽ ≤ 0 then return;
3     d_u = d_u − ▽;
4     t_u^SP = a;
5     Q = {u};
6     U = ∅;
7     for u ∈ Q do
8           for e = (s→,u→) ∈ IN(u) do
9                 diff = d_s − d_u − w_e;
10                if diff > 0 then
11                      if diff = ▽ then
12                            d_s = d_s − ▽;
13                            t_s^SP = e;
14                            Q = Q ∪ {s};
15                            maxdiff_s = 0;
16                      end if
17                      else if diff > maxdiff_s then
18                            if maxdiff_s = 0 then U = U ∪ {s};
19                            maxdiff_s = diff;
20                            t_s^SP = e;
21                      end if
22                end if
23          end for
24    end for
25    if ▽ = 1 then return;
26    for u ∈ U do
27          if maxdiff_u > 0 then
28                d_u = d_u − maxdiff_u;
29                maxdiff_u = 0;
30                InserIntoHeap(H,u,d_u);
31          end if
32    end for
33    while HeapSize(H) > 0 do
34          u = FindAndDeleteMin(H,d);
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                      t_s^SP = e;
39                      AdjustHeap(H,s,d_s);
40                end if
41          end for
42    end while
end rhT_-^RR.
```

FIGURE 15. Pseudo-code of procedure rhT_{-}^{RR}.

```
procedure T_-^KT(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           if t_u^SP > a then t_u^SP = a;
4           return;
5     end if
6     d_u = d_v + w_a;
7     t_u^SP = a;
8     InserIntoHeap(H,u,d_u);
9     while HeapSize(H) > 0 do
10          u =FindAndDeleteMin(H,d);
11          for e = (s→,u→) ∈ IN(u) do
12                if d_s > d_u + w_e then
13                      d_s = d_u + w_e;
14                      t_s^SP = e;
15                      AdjustHeap(H,s,d_s);
16                end if
17                else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
18          end for
19    end while
end T_-^KT.
```

FIGURE 16. Pseudo-code of procedure T_-^{KT}.

difference, both algorithms are similar. Two variants are presented: one that does not use the reduced heaps technique and another one that does.

The pseudo-code of algorithm T_-^{KT} is shown in Figure 16. The algorithm stops if d_u does not decrease (line 1). Otherwise, if node u has an alternative shortest path traversing arc a, then t^{SP} is updated if necessary (line 3) and the algorithm stops (line 4). Lines 6 to 19 are identical to lines 2 to 14 from the pseudo-code of algorithm T_-^{RR} in Figure 16. Line 17 is needed for this algorithm since in case there is an alternative shortest path from node s, the first arc belonging to a shortest path from the outgoing list should be the arc stored in t^{SP}.

The last pseudo-code presented in this section is the reduced heap variant rhT_-^{KT} presented in Figure 17. Let $a = (\overrightarrow{u,v})$ be the arc whose weight is decreased. If $\bigtriangledown < 0$ the algorithm stops (line 2). Otherwise, if node u has an alternative shortest path traversing arc a, the tree t^{SP} is updated if needed (line 4) and the algorithm stops (line 5). The distance d_u is updated (line 7), t_u^{SP} is set (line 8), and node u is inserted into set Q (line 9). The set U is initialized empty in line 10.

In this algorithm, the $|V|$-array *maxdiff* is used in the same manner that it was in T_-^{RR}. In T_-^{KT}, non-negative values for *diff* are considered. In case $diff = 0$ (line 15), as well as if *diff* = $maxdiff_s$ (line 29), then node s has an alternative shortest path. In this case, the special tree should be updated. For this reason, the update is tested in line 49.

```
procedure rhT_-^KT(a = (u→,v→), w_E, d_V, δ_V, g^SP)
1    ▽ = d_u − d_v − w_a;
2    if ▽ < 0 then return;
3    if ▽ = 0 then
4        if t_a^SP > v then t_a^SP = v;
5        return;
6    end if
7    d_u = d_v − ▽;
8    t_u^SP = a;
9    Q = {u};
10   U = ∅;
11   for u ∈ Q do
12       for e = (s→,u→) ∈ IN(u) do
13           diff = d_s − d_u − w_e;
14           if diff ≥ 0 then
15               if diff = 0 then
16                   if t_s^SP > e then t_s^SP = e;
17               end if
18               else if diff = ▽ then
19                   d_s = d_s − ▽;
20                   t_s^SP = e;
21                   Q = Q ∪ {s};
22                   maxdiff_s = 0;
23               end if
24               else if diff > maxdiff_s then
25                   if maxdiff_s = 0 then U = U ∪ {s};
26                   maxdiff_s = diff;
27                   t_s^SP = e;
28               end if
29               else if diff = maxdiff_s and t_s^SP > e then t_s^SP = e;
30           end if
31       end for
32   end for
33   if ▽ = 1 then return;
34   for u ∈ U do
35       if maxdiff_u > 0 then
36           d_u = d_u − maxdiff_u;
37           maxdiff_u = 0;
38           InserIntoHeap(H,u,d_u);
39       end if
40   end for
41   while HeapSize(H) > 0 do
42       u = FindAndDeleteMin(H,d);
43       for e = (s→,u→) ∈ IN(u) do
44           if d_s > d_u + w_e then
45               d_s = d_u + w_e;
46               t_s^SP = e;
47               AdjustHeap(H,s,d_s);
48           end if
49           else if d_s = d_u + w_e and t_s^SP > e then t_s^SP = e;
50       end for
51   end while
end rhT_-^KT.
```

FIGURE 17. Pseudo-code of procedure rhT_{-}^{KT}.

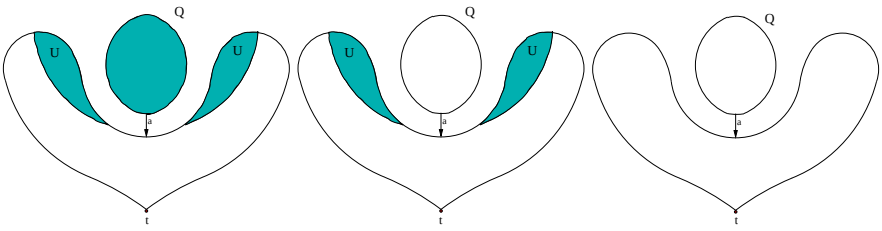

FIGURE 18. Heap size used by the decremental algorithms G_{-}^{RR}, T_{-}^{RR}, and T_{-}^{KT} and their reduced heap variants. Graph on the left represents the standard algorithms. The graph in the middle represents the reduced heaps variants for random weight decrease. On the right, for the reduced heaps variants with unit decrement, the heap is empty.

6. REDUCED HEAP VS. STANDARD VERSIONS OF DECREMENTAL ALGORITHMS

In this section, we make a few observations comparing the reduced heaps (`rh`) variants with the respective standard (`std`) versions of the decremental algorithms. Figure 18 illustrates the heap size used by the decremental algorithms discussed in the previous section. The set Q of affected nodes is linked to the remaining part of the graph by arc a, which had its weight increased. Node t is the destination node. The shaded part of set Q is composed of the nodes that were inserted into heap H.

As we can see in the fi gure, in the standard algorithms all nodes $u \in Q$ are inserted into H, while for the reduced heaps variants no node $u \in Q$ is inserted into H.

Nodes $u \in U$ are inserted into H for the standard and reduced heaps algorithms for random weight decrease, while H remains empty for unit decrement in the reduced heaps variant.

In terms of memory, the reduced heaps variants require extra space for their implementation. 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 set 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 measure with the system function `getrusage`.

The experiments were performed on nine classes of instances. The fi rst class, `Internet`, is from traffi c engineering problems studied in [1, 10]. This class is composed of four subclasses: `att`, `hier`, `rand`, and `wax`. These sub-classes were originally proposed in [10]. The fi rst sub-class is taken from a real-world AT&T IP network with old data, while the other three are synthetic internetwork instances. According to [10], `hier` are realistic models for internetworks, while `rand` and `wax` are derived from a mathematical perspective. The `rand` instances are purely random graphs. The probability of having an arc between two nodes is given by a constant parameter used to control the density of the graph. In `wax` instances, nodes are uniformily distributed points in a unit square and the existence of an arc between each pair of nodes depends on a probability p, which is a function of two

parameters that control the graph density, the Euclidean distances between the two nodes, and the maximum distances between two nodes in the graph. In this paper, arc weights are generated at random in the range [1,μ], where μ is 20 on the unit change experiments and 10,000 on 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, while instances from subclasses `hier`, `rand` and `was` have $|V|$ destination nodes.

The other eight classes are taken from Cherkassky, Goldberg, and Radzik [2]. They are `Grid-SSquare-S`, `Grid-SWide`, `Grid-SLong`, `Grid-PHard`, `Rand-4`, `Rand-1:4`, `Rand-Len`, and `Acyc-Pos`. They constitute all of the instances in [2] that only have non-negative arc lengths. The problem generators of Cherkassky, Goldberg, and Radzik are available on-line [1]. These instances were originally for single source shortest path, 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 arc lengths selected uniformly at random in the interval [0,10000]. The instances from class `Grid-PHard` are non-planar and constructed with a complex layer structure. In these instances, a path between two nodes with many arcs is likely to have shorter length than a path with fewer arcs. 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 length while the lengths of the remaining arcs are chosen randomly in the interval [0,μ]. The size of μ is fixed at 10,000 for classes `Rand-4` and `Rand-1:4`, and varies in class `Rand-Len`. For `Rand-Len`, the arc length is fixed at 1 for the first problem in the family, and selected in the interval [0,μ] for the others problems, with μ varying from 10 to 1,000,000. Because the lengths of the path arcs are set to 1, the structure of the shortest paths 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 lengths of the path arcs are set to 1 and the remaining arc lengths are selected in the interval [0,10000]. Since the dynamic shortest path algorithms update a graph with positive weights, we set to 1 all weights that originally were zero.

A summary of each class is given next.

- `Internet`: graphs from traffic engineering problems;
  - `att`: AT&T IP network;
  - `hier`: 2-level hierarchical graphs;
  - `rand`: purely random graphs.
  - `wax`: uniformily distributed points in a unit square;
- `Grid-SSquare-S`: generated on a square grid;
- `Grid-SWide`: generated on a wide grid;
- `Grid-SLong`: generated on a long grid;
- `Grid-PHard`: difficult grid instances;
- `Rand-4`: sparse graphs;
- `Rand-1:4`: dense graphs;
- `Rand-Len`: dependency on arc length;
- `Acyc-Pos`: acyclic networks.

[1] `http://www.avglab.att.com/~andrew/soft.html`

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 number generator seeds. The seeds we used for generating the instances of Cherkassky, Goldberg, and Radzik are those distributed with the generators. Instances from each grid graph group have identical edge sets, differing only with respect to arc weights. Instances from the remaining groups have identical dimensions, but differ both with respect to their edges 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 edge sets and differ only with respect to arc weights. The problem generators for the class `Internet` are available on-line [2].

For each instance, we ran 10,000 weight increases using an incremental algorithm followed by 10,000 weight decreases using a decremental algorithm, resulting in the end in the original graph. Times are measured for the incremental and decremental algorithms separately. The following pairs of incremental/decremental algorithms were used: (G_+^{RR}, G_-^{RR}), (rhG_+^{RR}, rhG_-^{RR}), (T_+^{RR}, T_-^{RR}), (rhT_+^{RR}, rhT_-^{RR}), (T_+^{KT}, T_-^{KT}), and (rhT_+^{KT}, rhT_-^{KT}). For algorithms T_+^D and rhT_+^D only the 10,000 weight increases were done since these shortest path algorithms are semi-dynamic. 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 only run Dijkstra's algorithm 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 10,000 changes multiplying the running time by 100.

For each problem instance and range of weight increase permited, all algorithms use the same sequence of arcs and weight change values per each instance. 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 $\overline{w}$ be the average arc length in the problem instance. The following procedure is repeated for $k = 1,\ldots,10,000$ to produce a sequence of arcs $a_1,\ldots,a_{10,000}$ and weight change values $\triangle_1,\ldots,\triangle_{10,000}$ to be 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 $[1,\overline{w}]$;
(3) add $\triangle_k$ to the current length 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: computational results for random weight changes; and for unit weight changes. Subsection 7.1 presents results for random weight changes, while Subsection 7.2 presents results for unit weight changes. All algorithms used in the experiments were implemented by us and are available for download [3].

7.1. **Random weight changes.** This subsection presents computational results for random weight changes applied on all instances of each group from each of the nine problem classes. The increase is a random value between 1 and the average arc weight from the original graph, e.g, $\triangle = [1,\overline{w}]$.

Tables 1 to 28 present, for each class of instances, averages computed for the five instances in each group. Each instance class has three tables: total CPU times (in seconds) for the 10,000 updates of the incremental algorithms, total CPU times (in seconds) for the

[2] `http://www.research.att.com/~mgcr/src/dspa-gen.tar.gz`
[3] `http://www.research.att.com/~mgcr/src/dspa.tar.gz`

10,000 updates of the decremental algorithms, and heap sizes for both types of algorithms. For each instance, the heap size is the average number of nodes inserted into the heap during each of the 10,000 updates. For each table, the first column identifies the group of instances (`Group`) which is being considered. The next two columns indicate the number of nodes $|V|$ and arcs $|E|$ for each group. The remaining columns present the results for the standard and reduced heaps versions of the algorithms G^{RR}, T^{RR} and T^{KT}, while for algorithm T^{D} only incremental results are given.

Since the set of nodes inserted into the heap by G^{RR}, T^{RR}, and T^{KT} are identical, the tables with heap sizes these entries are repeated in the tables. Likewise, for the reduced heap versions of these algorithms, entries are not repeated. Note that heap sizes in the incremental and decremental variants of G^{RR}, T^{RR}, and T^{KT} are identical.

The tables with CPU times also present a last column that contains results for Dijkstra's algorithm (`Dij`). The heap tables do not present results for this algorithm since Dijkstra's algorithm always inserts into the heap all nodes of the graph.

TABLE 1. CPU time in seconds for 10^4 random arc weight increases on `Internet` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1: att	90	274	0.16	0.09	0.16	0.09	0.16	0.09	0.14	0.06	2.00
GR2: hier50a	50	148	0.29	0.18	0.29	0.17	0.28	0.17	0.26	0.14	3.00
GR3: hier50b	50	212	0.25	0.16	0.25	0.16	0.25	0.16	0.22	0.12	2.60
GR4: hier100	100	280	0.88	0.48	0.87	0.46	0.85	0.44	0.76	0.33	11.40
GR5: hier100a	100	360	0.61	0.38	0.58	0.36	0.58	0.35	0.52	0.28	11.60
GR6: rand50	50	228	0.19	0.13	0.17	0.12	0.18	0.13	0.17	0.11	2.00
GR7: rand50a	50	245	0.17	0.12	0.17	0.13	0.17	0.12	0.16	0.10	3.00
GR8: rand100	100	403	0.52	0.36	0.49	0.33	0.49	0.32	0.45	0.26	11.80
GR9: rand100a	100	410	0.52	0.35	0.49	0.33	0.48	0.32	0.44	0.26	12.00
GR10: wax50	50	169	0.24	0.16	0.23	0.15	0.23	0.15	0.22	0.13	3.00
GR11: wax50a	50	230	0.19	0.13	0.18	0.13	0.18	0.13	0.17	0.11	2.40
GR12: wax100	100	391	0.53	0.34	0.50	0.33	0.49	0.33	0.45	0.27	11.20
GR13: wax100a	100	476	0.47	0.32	0.44	0.31	0.44	0.30	0.40	0.25	11.60

TABLE 2. CPU time in seconds for 10^4 random arc weight decreases on `Internet` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1: att	90	274	0.12	0.08	0.09	0.06	0.09	0.11	2.80
GR2: hier50a	50	148	0.20	0.16	0.15	0.14	0.16	0.22	2.60
GR3: hier50b	50	212	0.17	0.14	0.14	0.12	0.14	0.22	2.40
GR4: hier100	100	280	0.59	0.42	0.48	0.33	0.50	0.61	11.60
GR5: hier100a	100	360	0.44	0.35	0.34	0.29	0.36	0.47	11.40
GR6: rand50	50	228	0.13	0.13	0.11	0.11	0.11	0.17	2.40
GR7: rand50a	50	245	0.13	0.12	0.10	0.11	0.11	0.16	2.60
GR8: rand100	100	403	0.39	0.34	0.30	0.28	0.32	0.42	11.60
GR9: rand100a	100	410	0.39	0.33	0.29	0.28	0.31	0.41	11.80
GR10: wax50	50	169	0.17	0.14	0.14	0.13	0.14	0.19	3.00
GR11: wax50a	50	230	0.13	0.12	0.11	0.11	0.12	0.17	2.60
GR12: wax100	100	391	0.40	0.34	0.30	0.28	0.32	0.42	11.20
GR13: wax100a	100	476	0.35	0.32	0.27	0.26	0.29	0.40	11.00

TABLE 3. Heap size for each update (averaged over 10^4 random weight changes) on `Internet` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^D	rhT^D
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1: att	90	274	36.98	36.98	1.50	1.49	36.98	1.50
GR2: hier50a	50	148	70.55	70.55	2.10	2.08	70.55	2.10
GR3: hier50b	50	212	53.59	53.59	1.94	1.92	53.59	1.94
GR4: hier100	100	280	228.31	228.31	7.57	7.53	228.31	7.58
GR5: hier100a	100	360	124.88	124.88	9.37	9.32	124.88	9.37
GR6: rand50	50	228	30.60	30.60	2.22	2.22	30.60	2.23
GR7: rand50a	50	245	27.39	27.39	2.23	2.22	27.39	2.23
GR8: rand100	100	403	89.64	89.64	9.59	9.58	89.64	9.59
GR9: rand100a	100	410	87.18	87.18	9.76	9.76	87.18	9.76
GR10: wax50	50	169	47.70	47.70	2.13	2.12	47.70	2.13
GR11: wax50a	50	230	30.22	30.22	2.30	2.30	30.22	2.30
GR12: wax100	100	391	92.65	92.65	9.78	9.78	92.65	9.78
GR13: wax100a	100	476	73.10	73.10	9.45	9.44	73.10	9.45

TABLE 4. CPU time in seconds for 10^4 random arc weight increases on `Grid-SSquare-S` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	4098	16385	0.35	0.30	0.34	0.31	0.34	0.32	0.30	0.26	34.40
GR2	16386	65537	1.00	0.91	0.95	0.89	0.95	0.92	0.85	0.75	185.60
GR3	65538	262145	2.33	2.10	2.28	2.11	2.30	2.18	1.97	1.73	851.80
GR4	262146	1048577	6.11	5.44	5.99	5.44	6.11	5.63	5.28	4.54	4480.40
GR5	1048578	4194305	15.35	13.57	14.91	13.41	15.53	14.02	13.17	11.37	27690.60

TABLE 5. CPU time in seconds for 10^4 random arc weight decreases on `Grid-SSquare-S` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	4098	16385	0.24	0.21	0.19	0.16	0.19	0.18	35.00
GR2	16386	65537	0.66	0.62	0.48	0.43	0.49	0.47	186.40
GR3	65538	262145	1.46	1.35	1.02	0.94	1.05	1.00	852.20
GR4	262146	1048577	3.35	3.11	2.41	2.20	2.50	2.34	4470.60
GR5	1048578	4194305	7.81	7.09	5.42	4.87	5.63	5.20	27711.80

TABLE 6. Heap size for each update (averaged over 10^4 random weight changes) on `Grid-SSquare-S` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	4098	16385	40.67	40.67	31.03	29.50	40.67	31.03
GR2	16386	65537	88.67	88.67	72.56	71.02	88.67	72.57
GR3	65538	262145	171.15	171.15	140.75	137.50	171.16	140.75
GR4	262146	1048577	365.31	365.31	300.31	294.16	365.34	300.35
GR5	1048578	4194305	727.44	727.44	607.92	589.33	727.45	607.93

TABLE 7. CPU time in seconds for 10^4 random arc weight increases on `Grid-SWide` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	0.10	0.07	0.09	0.07	0.09	0.07	0.09	0.06	71.60
GR2	16385	49152	0.13	0.10	0.12	0.10	0.12	0.10	0.11	0.08	173.20
GR3	32769	98304	0.15	0.12	0.13	0.11	0.13	0.11	0.12	0.10	439.60
GR4	65537	196608	0.15	0.12	0.14	0.12	0.14	0.12	0.13	0.11	1251.00
GR5	131073	393216	0.17	0.15	0.16	0.14	0.16	0.13	0.15	0.12	3526.80
GR6	262145	786432	0.19	0.14	0.17	0.14	0.17	0.14	0.16	0.12	9700.20
GR7	524289	1572864	0.18	0.15	0.17	0.14	0.17	0.13	0.15	0.12	24350.20

TABLE 8. grid_swide

TABLE 9. CPU time in seconds for 10^4 random arc weight decreases on `Grid-SWide` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	0.08	0.07	0.05	0.04	0.05	0.05	71.80
GR2	16385	49152	0.10	0.09	0.07	0.06	0.07	0.07	173.60
GR3	32769	98304	0.11	0.10	0.08	0.07	0.08	0.08	440.20
GR4	65537	196608	0.12	0.11	0.08	0.08	0.08	0.08	1250.00
GR5	131073	393216	0.13	0.12	0.09	0.09	0.10	0.09	3527.40
GR6	262145	786432	0.14	0.13	0.10	0.09	0.10	0.10	9703.80
GR7	524289	1572864	0.14	0.13	0.10	0.10	0.10	0.10	24344.80

TABLE 10. Heap size for each update (averaged over 10^4 random weight changes) on `Grid-SWide` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^D	rhT^D
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8193	24576	11.13	11.13	4.62	4.60	11.13	4.62
GR2	16385	49152	11.25	11.25	4.59	4.56	11.26	4.59
GR3	32769	98304	11.36	11.36	4.81	4.78	11.36	4.81
GR4	65537	196608	11.14	11.14	4.64	4.61	11.15	4.64
GR5	131073	393216	11.49	11.49	4.67	4.64	11.49	4.67
GR6	262145	786432	11.58	11.58	4.50	4.46	11.58	4.50
GR7	524289	1572864	11.57	11.57	4.47	4.43	11.57	4.47

TABLE 11. CPU time in seconds for 10^4 random arc weight increases on `Grid-SLong` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	3.00	1.91	2.97	1.94	3.03	2.05	2.69	1.56	43.60
GR2	16385	49152	6.53	4.05	6.51	4.09	6.62	4.33	5.78	3.23	87.20
GR3	32769	98304	14.53	9.25	14.53	9.30	14.87	9.87	12.91	7.45	177.40
GR4	65537	196608	30.70	19.28	30.72	19.42	31.44	20.61	27.56	15.65	356.00
GR5	131073	393216	63.63	38.71	63.55	38.87	64.88	41.16	57.38	31.54	710.00
GR6	262145	786432	132.49	80.13	132.36	80.37	134.98	85.09	120.44	66.25	1427.60
GR7	524289	1572864	284.43	173.92	283.07	172.33	289.39	182.47	258.25	143.20	2847.00

TABLE 12. CPU time in seconds for 10^4 random arc weight decreases on `Grid-SLong` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	1.80	1.32	1.34	0.98	1.40	1.31	43.00
GR2	16385	49152	3.75	2.67	2.76	1.93	2.83	2.66	87.20
GR3	32769	98304	8.11	5.87	5.82	4.21	6.01	5.59	176.80
GR4	65537	196608	16.29	11.74	11.78	8.45	12.12	11.27	356.20
GR5	131073	393216	32.30	23.02	23.37	16.54	24.02	22.37	709.80
GR6	262145	786432	64.25	45.69	47.18	33.45	48.49	44.89	1428.80
GR7	524289	1572864	133.12	95.40	96.34	68.43	98.94	91.72	2848.40

TABLE 13. Heap size for each update (averaged over 10^4 random weight changes) on `Grid-SWide` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8193	24576	326.31	326.31	159.40	158.68	326.32	159.41
GR2	16385	49152	650.45	650.45	302.50	300.30	650.46	302.50
GR3	32769	98304	1342.16	1342.16	668.65	665.15	1342.34	668.84
GR4	65537	196608	2683.29	2683.29	1319.14	1313.03	2683.30	1319.15
GR5	131073	393216	5347.72	5347.72	2569.13	2563.32	5347.72	2569.13
GR6	262145	786432	10767.60	10767.60	5172.76	5159.84	10767.60	5172.76
GR7	524289	1572864	22094.40	22094.40	10901.09	10844.20	22094.40	10901.10

TABLE 14. CPU time in seconds for 10^4 random arc weight increases on `Grid-PHard` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	63808	12.08	10.20	12.26	10.52	11.94	10.35	10.05	8.16	122.20
GR2	16385	129344	30.39	25.97	30.94	26.73	30.28	26.47	25.75	21.13	245.20
GR3	32769	260416	71.24	61.04	71.96	62.42	70.62	61.68	60.36	49.42	503.00
GR4	65537	522560	128.67	111.00	127.41	111.15	125.31	110.08	107.81	89.25	774.40
GR5	131073	1046848	286.33	248.09	283.35	247.14	278.38	245.54	239.81	199.27	1528.40
GR6	262145	2095424	587.38	511.89	583.30	514.92	572.67	508.82	496.90	414.58	3131.00

TABLE 15. CPU time in seconds for 10^4 random arc weight decreases on `Grid-PHard` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	63808	7.44	6.46	5.03	4.34	5.33	5.39	122.00
GR2	16385	129344	17.05	15.14	11.72	10.27	12.43	12.90	245.60
GR3	32769	260416	38.99	35.42	24.84	22.13	26.37	28.06	502.80
GR4	65537	522560	62.82	57.97	38.00	35.27	40.46	44.71	776.20
GR5	131073	1046848	134.85	123.28	79.91	75.03	84.81	95.32	1528.20
GR6	262145	2095424	254.48	236.82	166.31	152.96	176.48	192.45	3128.80

TABLE 16. Heap size for each update (averaged over 10^4 random weight changes) on `Grid-PHard` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8193	63808	693.42	693.42	532.18	487.34	695.73	534.71
GR2	16385	129344	1699.61	1699.61	1316.92	1196.78	1706.92	1325.28
GR3	32769	260416	3873.01	3873.01	3028.37	2773.73	3895.42	3053.46
GR4	65537	522560	8597.23	8597.23	6842.77	6242.90	8656.73	6907.45
GR5	131073	1046848	18707.41	18707.41	15027.50	13694.50	18817.40	15148.60
GR6	262145	2095424	38572.20	38572.20	31242.00	28564.57	38810.60	31489.00

TABLE 17. CPU time in seconds for 10^4 random arc weight increases on Rand-4 instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	32768	0.39	0.33	0.38	0.32	0.38	0.32	0.32	0.25	90.40
GR2	16384	65536	0.94	0.83	0.93	0.84	0.97	0.86	0.79	0.68	276.60
GR3	32768	131072	2.00	1.80	2.10	1.94	2.20	1.95	1.73	1.55	817.40
GR4	65536	262144	4.41	4.00	4.76	4.43	4.98	4.40	3.92	3.56	2428.60
GR5	131072	524288	8.16	7.66	8.86	8.48	9.29	8.41	7.37	6.94	6266.60
GR6	262144	1048576	11.14	10.50	11.96	11.46	12.51	11.33	10.09	9.54	14752.40
GR7	524288	2097152	15.02	14.19	15.98	15.33	16.74	15.17	13.53	12.81	33346.60
GR8	1048576	4194304	20.54	19.28	21.42	20.47	22.53	20.23	18.39	17.20	73487.00

TABLE 18. CPU time in seconds for 10^4 random arc weight decreases on Rand-4 instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8192	32768	0.33	0.28	0.19	0.17	0.19	0.19	90.40
GR2	16384	65536	0.77	0.64	0.48	0.47	0.49	0.48	276.80
GR3	32768	131072	1.56	1.32	0.99	0.99	1.01	1.00	819.00
GR4	65536	262144	3.15	2.70	2.01	2.00	2.04	2.03	2430.60
GR5	131072	524288	5.42	4.64	3.41	3.40	3.45	3.43	6269.00
GR6	262144	1048576	7.29	6.26	4.54	4.50	4.59	4.53	14739.20
GR7	524288	2097152	9.57	8.33	5.97	5.99	6.05	6.06	33415.20
GR8	1048576	4194304	12.84	11.08	7.94	7.82	8.10	7.94	73593.40

TABLE 19. Heap size for each update (averaged over 10^4 random weight changes) on `Rand-4` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8192	32768	25.33	25.33	17.62	17.61	25.38	17.66
GR2	16384	65536	41.46	41.46	33.15	33.15	41.50	33.20
GR3	32768	131072	67.19	67.19	55.99	55.98	67.31	56.12
GR4	65536	262144	117.05	117.05	102.34	102.33	117.34	102.63
GR5	131072	524288	184.52	184.52	168.38	168.37	185.22	169.09
GR6	262144	1048576	229.97	229.97	214.25	214.24	231.00	215.28
GR7	524288	2097152	278.77	278.77	268.56	268.55	280.07	269.86
GR8	1048576	4194304	331.87	331.87	309.41	309.41	333.66	311.22

TABLE 20. CPU time in seconds for 10^4 random arc weight increases on `Rand-1:4` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	512	65536	0.95	0.71	0.72	0.60	0.80	0.64	0.65	0.51	69.20
GR2	1024	262144	4.16	3.40	3.71	3.25	3.89	3.30	3.53	3.03	528.40
GR3	2048	1048576	12.69	10.95	12.03	10.82	12.49	10.85	11.80	10.54	2421.20
GR4	4096	4194304	37.18	32.70	37.02	33.07	38.32	32.96	36.90	33.30	9798.60

TABLE 21. CPU time in seconds for 10^4 random arc weight decreases on `Rand-1:4` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	512	65536	1.06	0.83	0.48	0.47	0.56	0.53	69.40
GR2	1024	262144	4.42	3.49	2.61	2.60	2.73	2.67	528.40
GR3	2048	1048576	12.84	10.14	7.87	7.87	8.08	7.99	2421.00
GR4	4096	4194304	39.26	30.84	23.47	23.10	23.80	23.42	9797.60

TABLE 22. Heap size for each update (averaged over 10^4 random weight changes) on `Rand-1:4` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	512	65536	3.98	3.98	2.58	2.58	3.99	2.59
GR2	1024	262144	5.53	5.53	4.04	4.03	5.57	4.08
GR3	2048	1048576	7.22	7.22	5.64	5.63	7.37	5.79
GR4	4096	4194304	9.58	9.58	7.81	7.80	10.08	8.32

TABLE 23. CPU time in seconds for 10^4 random arc weight increases on `Rand-Len` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	131072	524288	0.97	0.68	0.96	0.68	1.01	0.67	0.83	0.55	7127.40
GR2	131072	524288	0.44	0.37	0.44	0.37	0.46	0.37	0.44	0.37	4189.20
GR3	131072	524288	0.56	0.42	0.56	0.43	0.58	0.42	0.49	0.36	6236.20
GR4	131072	524288	0.86	0.60	0.86	0.60	0.90	0.59	0.74	0.49	7138.00
GR5	131072	524288	0.84	0.59	0.83	0.59	0.87	0.59	0.72	0.48	7169.60

TABLE 24. CPU time in seconds for 10^4 random arc weight decreases on `Rand-Len` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	131072	524288	0.73	0.59	0.50	0.41	0.50	0.41	7128.80
GR2	131072	524288	0.34	0.31	0.22	0.21	0.22	0.22	4188.00
GR3	131072	524288	0.43	0.37	0.29	0.26	0.29	0.26	6236.60
GR4	131072	524288	0.65	0.53	0.45	0.36	0.45	0.37	7129.20
GR5	131072	524288	0.63	0.52	0.44	0.36	0.44	0.36	7166.40

TABLE 25. Heap size for each update (averaged over 10^4 random weight changes) on `Rand-Len` instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	131072	524288	16.04	16.04	6.87	6.87	16.04	6.87
GR2	131072	524288	8.79	8.79	5.04	5.04	10.31	6.56
GR3	131072	524288	11.02	11.02	5.09	5.09	11.20	5.27
GR4	131072	524288	14.61	14.61	6.23	6.23	14.62	6.23
GR5	131072	524288	14.31	14.31	6.30	6.30	14.31	6.30

TABLE 26. CPU time in seconds for 10^4 random arc weight increases on Acyc-Pos instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	131072	1.06	0.82	0.97	0.79	0.96	0.75	0.86	0.65	395.40
GR2	16384	262144	2.07	1.74	2.02	1.75	1.95	1.63	1.78	1.49	998.20
GR3	32768	524288	4.02	3.51	4.25	3.78	4.05	3.44	3.63	3.17	2653.20
GR4	65536	1048576	7.85	6.93	8.66	7.91	8.31	6.89	7.29	6.45	6586.20
GR5	131072	2097152	13.24	12.29	14.87	14.00	14.44	12.29	12.55	11.68	15605.20

TABLE 27. CPU time in seconds for 10^4 random arc weight decreases on Acyc-Pos instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8192	131072	1.05	0.89	0.60	0.57	0.64	0.60	396.40
GR2	16384	262144	1.97	1.61	1.16	1.14	1.22	1.17	998.20
GR3	32768	524288	3.77	3.04	2.22	2.19	2.29	2.23	2649.80
GR4	65536	1048576	7.00	5.85	4.17	4.16	4.33	4.18	6593.20
GR5	131072	2097152	11.25	9.25	6.66	6.63	6.90	6.67	15596.60

TABLE 28. Heap size for each update (averaged over 10^4 random weight changes) on Acyc-Pos instances

			G^{RR}, T^{RR}, T^{KT}		$rhG^{RR}, rhT^{RR}, rhT^{KT}$		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Decr	Incr	Incr
GR1	8192	131072	15.90	15.90	10.90	10.90	15.92	10.92
GR2	16384	262144	25.11	25.11	19.68	19.68	25.15	19.72
GR3	32768	524288	40.52	40.52	33.49	33.49	40.62	33.59
GR4	65536	1048576	66.86	66.86	57.11	57.11	67.00	57.24
GR5	131072	2097152	97.67	97.67	88.01	88.01	97.97	88.31

TABLE 29. Time ratio between the standard and reduced heaps implementations of the algorithms for updating 10^4 random weight changes

Class	G_+^{RR}	G_-^{RR}	T_+^{RR}	T_-^{RR}	T_+^{KT}	T_-^{KT}	T_+^{D}
Internet	1.55	1.21	1.55	1.11	1.57	0.74	1.79
Grid-SSquare-S	1.13	1.09	1.10	1.12	1.07	1.06	1.15
Grid-SWide	1.26	1.08	1.22	1.08	1.22	1.01	1.29
Grid-SLong	1.61	1.39	1.60	1.40	1.54	1.07	1.78
Grid-PHard	1.16	1.11	1.15	1.11	1.14	0.93	1.21
Rand-4	1.10	1.17	1.08	1.02	1.12	1.01	1.11
Rand-1:4	1.22	1.27	1.14	1.01	1.18	1.03	1.17
Rand-Len	1.37	1.19	1.34	1.17	1.43	1.16	1.42
Acyc-Pos	1.17	1.21	1.13	1.02	1.21	1.04	1.17
Average	1.29	1.19	1.26	1.12	1.28	1.01	1.34

In the remainder of this subsection, we discuss the experimental results for random weight change.

7.1.1. *Improvement obtained using reduced heap algorithms.* Table 29 compares CPU times for the standard and reduced heaps versions of all dynamic shortest paths algorithms described in this paper. Each value in the table is the average ratio of the CPU times of the standard and reduced heaps versions. More precisely, let ts_i and tr_i be the average CPU times for the five instances of the i-th group in the class for the standard and reduced heaps versions, respectively. Then, the table entries are

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

where n_g is the number of groups in the class. The last line lists the average ratio for each algorithm. Note that the tr_i and ts_i values above are the ones displayed in the CPU time tables of the instance classes.

On average, the reduced heap algorithms run faster than their standard version counterparts, as shown in the last line of the table. For all incremental variants, the reduced heap version was faster than the standard version, for all instance classes. This was also true for the decremental algorithms, with the exception of the T_-^{KT} algorithm, for which the reduced heap version took longer, on average, than the standard version 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. From the last row, we observe that the incremental algorithms benefited more from the reduced heap idea than did the decremental algorithms. This is expected, since applying this idea in the decremental algorithms requires extra computational effort: the incoming links to nodes in set Q are scanned twice, while in the standard algorithm they are scanned only once. The stardard incremental algorithms took, on average, 29.25% longer than their reduced heap counterparts, while for the decremental algorithms, the standard algorithms took, on average, 10.67% longer than their reduced heaps counterparts.

Table 30 compares average heap sizes. Each value in the table is the average ratio of the heap size of the standard and reduced heaps versions. More precisely, let hs_i and hr_i be the average heap sizes for the five instances of the i-th group in the class for the standard and

TABLE 30. Ratio between heap sizes of standard and reduced heaps implementations of the algorithms for updating 10^4 random weight changes

	G^{RR}, T^{RR}, T^{KT}		T^D
Class	Incr	Decr	Incr
Internet	17.42	17.50	17.42
Grid-SSquare-S	1.23	1.27	1.23
Grid-SWide	2.46	2.48	2.46
Grid-SLong	2.06	2.07	2.06
Grid-PHard	1.27	1.39	1.27
Rand-4	1.16	1.16	1.16
Rand-1:4	1.35	1.36	1.35
Rand-Len	2.17	2.17	2.13
Acyc-Pos	1.24	1.24	1.24
Average	3.37	3.40	3.37

reduced heaps versions, respectively. Then, the table entries are

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

The last line lists the average ratio for each algorithm. Note that the hr_i and hs_i values above are the ones displayed in the heap time table of the instance classes. For all combinations of algorithm and instance class, the idea of reduced heaps was successful in reducing the heap size. On average, the heap size was reduced by more than one third.

7.1.2. *Time comparison between recomputing from scratch (`Dij`) and using a dynamic shortest path graph algorithm (G^{RR}).* Table 31 shows the ratios between the total times spent by Dijkstra's algorithm and G^{RR} for each class of instances. Each value in the table is the average ratio of the CPU times of the `Dij` and G^{RR} algorithms. The table entries are computed in a fashion similar to (1).

The last line lists the average ratio for each algorithm. In this table, the CPU time for each algorithm is the sum of the CPU times of the incremental and decremental phases. For instance classes `Internet`, `Grid-SSquare-S`, `Grid-SWide`, `Rand-4`, `Rand-14`, and `Acyc-Pos`, the ratio increases with instances size, while the opposite occurs in classes `Grid-SLong` and `Grid-PHard`. Class `Rand-Len` consists of instances of the same size. The smallest ratio is 7.26, for group `GR5` of class `Grid-PHard`. The largest is over 149 thousand, 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 subsection, we show that for the instances considered in this experiment, any gain that could be achieved while scanning the outgoing links in T^{KT}, is washed out by the additional computational effort associated with maintaining the special tree proposed by King and Thorup [18]. Table 32 presents results for this comparison.

Each value in the table is the average ratio of the CPU times of the T^{KT} and T^{RR}. The table entries are computed in a fashion similar to (1). The last line lists the average ratio for each algorithm.

TABLE 31. Ratio between the time spent by the Dij and G^{RR} algorithms for updating 10^4 random weight changes 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	8395.88	374.91
GR2	11.43	223.83	1507.83	16.96	10.35	324.77	123.29	10712.53	493.67
GR3	11.79	449.37	3383.85	15.64	9.12	459.40	189.64	12522.89	681.27
GR4	15.65	945.59	9262.96	15.16	8.10	643.26	256.36	9435.98	887.62
GR5	22.03	2391.75	23204.61	14.80	7.26	922.96		9739.13	1274.06
GR6	13.75		60260.87	14.52	7.44	1600.20			
GR7	18.42		149371.17	13.64		2715.88			
GR8	25.88					4406.24			
GR9	26.10								
GR10	14.56								
GR11	15.53								
GR12	24.24								
GR13	27.63								

TABLE 32. Ratio between the time spent by the T^{KT} and T^{RR} algorithms for updating 10^4 random weight changes

	std		rh	
Class	Incr	Decr	Incr	Decr
Internet	1.00	1.06	1.56	1.60
Grid-SSquare-S	1.02	1.03	1.09	1.09
Grid-SWide	1.00	1.04	1.10	1.11
Grid-SLong	1.02	1.03	1.30	1.34
Grid-PHard	0.98	1.06	1.22	1.26
Rand-4	1.04	1.02	1.02	1.03
Rand-1:4	1.06	1.06	1.04	1.05
Rand-Len	1.05	1.01	1.01	1.02
Acyc-Pos	0.97	1.05	0.99	1.02
Average	1.02	1.04	1.15	1.17

TABLE 33. Ratio between the time spent by the T^{D} and T^{RR} algorithms for updating 10^4 random weight changes

	Time		Heap Size	
Class	std	rh	std	rh
Internet	0.91	0.79	1.00	1.00
Grid-SSquare-S	0.88	0.84	1.00	1.00
Grid-SWide	0.93	0.89	1.00	1.00
Grid-SLong	0.90	0.81	1.00	1.00
Grid-PHard	0.84	0.80	1.01	1.01
Rand-4	0.84	0.81	1.00	1.00
Rand-1:4	0.96	0.94	1.02	1.03
Rand-Len	0.90	0.85	1.04	1.07
Acyc-Pos	0.86	0.83	1.00	1.00
Average	0.89	0.84	1.01	1.01

The second column of the table presents results for the comparison between the standard implementations, while the third column shows the comparison between reduced heaps implementations. For both standard and reduced heaps columns, results are presented comparing the incremental and decremental algorithms. On average, for the incremental and decremental algorithms for the standard and reduced heaps implementations, algorithm T^{RR} is faster than algorithm T^{KT}. For the standard algorithms, the performance is almost the same, while for reduced heaps variants, algorithm T^{KT}, on average, takes 15% and 17% longer than T^{RR}.

7.1.4. *Time comparison between algorithms T^{RR} and T^{D}.* We next, compare algorithms T^{RR} and T^{D}. Table 33 presents the results for this comparison. The second column of the table present results for the time comparison, while the third column shows the comparison between heap sizes. For both, results are presented comparing the standard and reduced heaps implementations. Since T^{D} only has an incremental version, there is no entry for decremental versions of the algorithms.

TABLE 34. Ratio between the time spent by the G^{RR} and T^{RR} algorithms for updating 10^4 random weight changes

	std		rh	
Class	Incr	Decr	Incr	Decr
Internet	1.04	1.04	1.29	1.18
gridi_ssquare_s	1.03	1.00	1.39	1.42
grid_swide	1.10	1.08	1.45	1.45
grid_slong	1.00	0.99	1.37	1.38
grid_phard	1.00	0.99	1.56	1.57
rand_4	0.96	0.94	1.61	1.41
rand1:4	1.13	1.06	1.80	1.44
rand_len	1.01	0.99	1.48	1.44
acyc_pos	0.97	0.94	1.70	1.43
Average	1.03	1.00	1.52	1.41

Each value in the table is the average ratio of the CPU times of the T^{RR} and T^{D}. For the second column, the table entries are computed in a fashion similar to (1), while that for the third column the table entries are computed in a fashion similar to (2). The last line lists the average ratio for each algorithm.

On average, T_+^{KT} takes more than 10% longer than T_+^{D} for both standard and reduced heaps implementations. With respect to heap size, we can observe that 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 identifi cation 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 updating a shortest path graph with time spent updating a shortest path tree. To do this, we use CPU times for 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} since it was faster than T^{KT} in our experiments, and the running times of T^{D} are less predictable.

Table 34 shows results for the standard and the reduced heap versions of the algorithms. Each value in the table is the average ratio of the CPU times of the G^{RR} and T^{RR}. The table entries are computed in a fashion similar to (1). The last line lists the average ratio for each algorithm.

The second column of the table present results comparing the standard versions, while the third column shows the comparison between the reduced heaps variants. In both cases, results are presented comparing the incremental and decremental algorithms. As we can observe, for the standard implementations of the algorithms, updating a tree is slightly faster than updating a graph for the incremental version, and about the same for the decremental version. For the reduced heaps comparison, updating a graph takes 52% and 41% longer than updating a shortest path tree, for incremental and decremental algorithms, respectively.

7.1.6. *Time comparison between incremental and decremental implementations of the dynamic algorithms.* In this subsection, we explore the time difference between the incremental and the decremental algorithms. Table 35 presents the results for this comparison. Each value in the table is the average ratio of the CPU times of the algorithm pairs (G_+^{RR},

TABLE 35. Time ratio between the incremental and decremental implementations of the algorithms spent for updating 10^4 random weight changes

Class	G^{RR}	rhG^{RR}	T^{RR}	rhT^{RR}	T^{KT}	rhT^{KT}
Internet	1.38	1.08	1.72	1.23	1.62	0.76
Grid-SSquare-S	1.67	1.62	2.26	2.30	2.23	2.19
Grid-SWide	1.33	1.14	1.75	1.53	1.68	1.39
Grid-SLong	1.89	1.63	2.59	2.27	2.57	1.79
Grid-PHard	1.95	1.85	3.06	2.94	2.83	2.31
Rand-4	1.41	1.51	2.38	2.26	2.45	2.20
Rand-1:4	0.94	0.99	1.50	1.33	1.50	1.30
Rand-Len	1.32	1.15	1.93	1.68	2.01	1.63
Acyc-Pos	1.08	1.13	1.92	1.73	1.78	1.54
Average	1.44	1.34	2.12	1.92	2.07	1.68

G^{RR}_{-}), (rhG^{RR}_{+}, rhG^{RR}_{-}), (T^{RR}_{+}, T^{RR}_{-}), (rhT^{RR}_{+}, rhT^{RR}_{-}), (T^{KT}_{+}, T^{KT}_{-}), and (rhT^{KT}_{+}, rhT^{KT}_{-}). The table entries are computed in a fashion similar to (1). The last line lists the average ratio for each algorithm.

In almost all cases, the decremental algorithm is faster than the incremental counterpart. This performance was expected since the number of links scanned by the decremental algorithm is smaller than the correspondent scanned by the incremental algorithm.

We observe that the in reduced heap variants, the ratios are smaller than in the standard versions of the algorithms.

7.2. **Unit weight changes.** This subsection presents computational results for unit weight changes applied on all instances of each group from each of the nine problem classes. The increase is equal to one, e.g, $\triangle = 1$.

Tables 36 to 62 present, for each class of instances, averages computed for the five instances in each group.

This process is the same used by the random weight changes, with the only difference that now the increase is unitary. As before, we run each algorithm updating 10^4 arc weight changes. For this experiment, instances from class `Internet` have weights generated 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 $w_a = 1 + (20 * \lfloor w_a \div 20 \rfloor)$.

Since the set of nodes inserted into the heap by G^{RR}, T^{RR}, and T^{KT} are identical, the tables with heap sizes these entries are repeated in the tables. Heap sizes is only presented for the reduced heap variant rhT^{D}, since it is the only variant which insertes nodes into the heap for unit weight increase.

TABLE 36. CPU time in seconds for 10^4 unit arc weight increases on `Internet` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1: att	90	274	0.15	0.06	0.15	0.06	0.15	0.06	0.13	0.06	2.00
GR2: hier50	50	148	0.29	0.14	0.29	0.12	0.29	0.13	0.26	0.13	2.60
GR3: hier50b	50	212	0.26	0.12	0.26	0.12	0.26	0.12	0.23	0.12	2.40
GR4: hier100	100	280	0.88	0.35	0.88	0.33	0.87	0.31	0.79	0.34	11.60
GR5: hier100a	100	360	0.60	0.28	0.59	0.26	0.59	0.26	0.54	0.27	10.60
GR6: rand50	50	228	0.18	0.10	0.17	0.09	0.17	0.10	0.16	0.10	2.00
GR7: rand50a	50	245	0.16	0.10	0.16	0.09	0.16	0.10	0.15	0.10	2.20
GR8: rand100	100	403	0.49	0.25	0.47	0.22	0.46	0.23	0.42	0.24	10.00
GR9: rand100a	100	410	0.48	0.25	0.46	0.23	0.46	0.23	0.42	0.24	10.80
GR10: wax50	50	169	0.23	0.12	0.23	0.11	0.22	0.11	0.20	0.12	2.60
GR11: wax50a	50	230	0.16	0.10	0.17	0.09	0.16	0.10	0.16	0.10	2.60
GR12: wax100	100	391	0.49	0.25	0.47	0.23	0.47	0.24	0.44	0.25	10.80
GR13: wax100a	100	476	0.43	0.24	0.41	0.21	0.41	0.22	0.38	0.22	9.60

TABLE 37. CPU time in seconds for 10^4 unit arc weight decreases on `Internet` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1: att	90	274	0.11	0.05	0.08	0.06	0.09	0.10	2.00
GR2: hier50a	50	148	0.20	0.12	0.16	0.13	0.17	0.22	2.40
GR3: hier50b	50	212	0.18	0.11	0.14	0.12	0.15	0.23	2.40
GR4: hier100	100	280	0.60	0.31	0.50	0.32	0.53	0.61	11.60
GR5: hier100a	100	360	0.44	0.26	0.35	0.27	0.37	0.47	10.80
GR6: rand50	50	228	0.13	0.10	0.10	0.11	0.11	0.16	2.40
GR7: rand50a	50	245	0.12	0.09	0.09	0.10	0.10	0.15	2.60
GR8: rand100	100	403	0.37	0.25	0.28	0.26	0.31	0.41	9.60
GR9: rand100a	100	410	0.37	0.25	0.28	0.26	0.31	0.40	11.00
GR10: wax50	50	169	0.17	0.11	0.13	0.12	0.14	0.19	2.60
GR11: wax50a	50	230	0.12	0.09	0.10	0.10	0.10	0.15	2.20
GR12: wax100	100	391	0.38	0.25	0.29	0.26	0.31	0.41	10.40
GR13: wax100a	100	476	0.34	0.24	0.25	0.24	0.27	0.39	8.80

TABLE 38. Heap size for each update (averaged over 10^4 unit weight changes) on `Internet` instances

			G^{RR}, T^{RR}, T^{KT}		T^D	rhT^D
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1: att	90	274	35.77	35.77	36.03	0.28
GR2: hier50a	50	148	70.05	70.05	70.55	0.50
GR3: hier50b	50	212	56.07	56.07	56.48	0.42
GR4: hier100	100	280	230.86	230.86	232.47	1.61
GR5: hier100b	100	360	125.21	125.21	127.04	1.86
GR6: rand50	50	228	28.17	28.17	28.66	0.52
GR7: rand50a	50	245	23.84	23.84	24.30	0.50
GR8: rand100	100	403	84.72	84.72	86.61	1.93
GR9: rand100b	100	410	82.02	82.02	83.95	1.98
GR10: wax50	50	169	45.34	45.34	45.85	0.53
GR11: wax50a	50	230	26.12	26.12	26.59	0.50
GR12: wax100	100	391	86.80	86.80	88.75	1.98
GR13: wax100a	100	476	68.09	68.09	69.85	1.83

TABLE 39. CPU time in seconds for 10^4 unit arc weight increases on `Grid-SSquare-S` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	4098	16385	0.01	0.00	0.00	0.00	0.01	0.01	0.01	0.01	9.00
GR2	16386	65537	0.02	0.02	0.02	0.01	0.02	0.01	0.02	0.01	49.20
GR3	65538	262145	0.03	0.02	0.03	0.02	0.03	0.02	0.03	0.02	188.20
GR4	262146	1048577	0.03	0.02	0.03	0.02	0.03	0.02	0.02	0.02	748.80
GR5	1048578	4194305	0.04	0.03	0.04	0.03	0.04	0.04	0.04	0.03	2927.00

TABLE 40. CPU time in seconds for 10^4 unit arc weight decreases on `Grid-SSquare-S` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	4098	16385	0.00	0.01	0.01	0.00	0.00	0.01	7.60
GR2	16386	65537	0.02	0.01	0.01	0.01	0.01	0.01	48.80
GR3	65538	262145	0.03	0.02	0.02	0.02	0.02	0.01	187.80
GR4	262146	1048577	0.03	0.03	0.02	0.02	0.02	0.02	749.40
GR5	1048578	4194305	0.03	0.03	0.02	0.02	0.02	0.02	2927.60

TABLE 41. Heap size for each update (averaged over 10^4 unit weight changes) on `Grid-SSquare-S` instances

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	4098	16385	0.90	0.90	0.96	0.01
GR2	16386	65537	0.92	0.92	0.96	0.00
GR3	65538	262145	0.97	0.97	0.98	0.00
GR4	262146	1048577	0.99	0.99	0.99	0.00
GR5	1048578	4194305	1.00	1.00	1.00	0.00

TABLE 42. CPU time in seconds for 10^4 unit arc weight increases on `Grid-SWide` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	0.07	0.03	0.06	0.02	0.06	0.03	0.06	0.03	44.00
GR2	16385	49152	0.09	0.05	0.08	0.05	0.09	0.05	0.08	0.05	108.40
GR3	32769	98304	0.11	0.07	0.10	0.06	0.10	0.06	0.10	0.06	298.00
GR4	65537	196608	0.11	0.07	0.10	0.06	0.10	0.06	0.10	0.07	884.80
GR5	131073	393216	0.12	0.08	0.12	0.07	0.11	0.07	0.11	0.07	2388.40
GR6	262145	786432	0.13	0.08	0.12	0.07	0.12	0.07	0.12	0.08	5326.40
GR7	524289	1572864	0.15	0.08	0.14	0.07	0.13	0.08	0.13	0.07	11361.00

TABLE 43. CPU time in seconds for 10^4 unit arc weight decreases on `Grid-SWide` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	0.05	0.03	0.03	0.02	0.03	0.04	43.80
GR2	16385	49152	0.08	0.05	0.05	0.04	0.05	0.04	108.60
GR3	32769	98304	0.09	0.06	0.06	0.05	0.06	0.06	298.00
GR4	65537	196608	0.10	0.07	0.07	0.05	0.07	0.06	885.00
GR5	131073	393216	0.10	0.07	0.07	0.06	0.08	0.07	2389.40
GR6	262145	786432	0.11	0.08	0.07	0.06	0.08	0.07	5328.80
GR7	524289	1572864	0.12	0.08	0.08	0.06	0.08	0.07	11351.40

TABLE 44. Heap size for each update (averaged over 10^4 unit weight changes) on `Grid-SWide` instances

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8193	24576	10.06	10.06	10.73	0.77
GR2	16385	49152	10.20	10.20	10.90	0.78
GR3	32769	98304	10.27	10.27	10.89	0.66
GR4	65537	196608	10.44	10.44	11.05	0.67
GR5	131073	393216	10.57	10.57	11.15	0.60
GR6	262145	786432	10.49	10.49	11.06	0.58
GR7	524289	1572864	10.38	10.38	10.90	0.53

TABLE 45. CPU time in seconds for 10^4 unit arc weight increases on `Grid-SLong` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	24576	2.03	0.49	2.00	0.45	2.02	0.52	1.92	0.54	29.40
GR2	16385	49152	4.52	1.12	4.51	0.99	4.56	1.25	4.18	1.07	63.00
GR3	32769	98304	10.13	2.45	10.05	2.10	10.18	2.76	9.32	2.36	128.80
GR4	65537	196608	21.29	4.85	21.18	4.18	21.50	5.55	20.14	4.89	251.20
GR5	131073	393216	44.50	9.59	43.94	8.26	44.70	11.01	41.78	9.56	498.00
GR6	262145	786432	91.41	18.74	90.24	16.21	91.52	21.60	86.21	19.22	1012.20
GR7	524289	1572864	258.94	49.06	255.45	42.53	258.06	54.36	242.56	47.26	2637.60

TABLE 46. CPU time in seconds for 10^4 unit arc weight decreases on `Grid-SLong` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	24576	1.24	0.45	0.89	0.36	0.94	0.80	29.60
GR2	16385	49152	2.62	1.10	1.85	0.79	1.95	1.66	63.00
GR3	32769	98304	5.63	2.39	3.93	1.74	4.15	3.46	128.60
GR4	65537	196608	11.17	4.82	7.80	3.47	8.24	6.83	251.40
GR5	131073	393216	22.28	9.55	15.59	6.90	16.40	13.58	497.80
GR6	262145	786432	43.53	18.81	30.72	13.61	32.32	26.63	1012.40
GR7	524289	1572864	121.74	46.20	86.58	35.89	91.09	75.18	2637.40

TABLE 47. Heap size for each update (averaged over 10^4 unit weight changes) on `Grid-SLong` instances

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8193	24576	301.20	301.20	325.95	27.97
GR2	16385	49152	600.38	600.38	643.40	41.89
GR3	32769	98304	1231.78	1231.78	1316.80	90.95
GR4	65537	196608	2439.44	2439.44	2647.14	194.49
GR5	131073	393216	4858.52	4858.52	5211.27	349.32
GR6	262145	786432	9569.81	9569.81	10272.00	708.83
GR7	524289	1572864	19765.20	19765.20	20884.20	1142.01

TABLE 48. CPU time in seconds for 10^4 unit arc weight increases on `Grid-PHard` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8193	63808	0.16	0.08	0.15	0.07	0.15	0.07	0.18	0.12	55.60
GR2	16385	129344	0.27	0.13	0.25	0.12	0.26	0.13	0.34	0.23	112.40
GR3	32769	260416	0.48	0.22	0.46	0.20	0.46	0.23	0.70	0.49	200.00
GR4	65537	522560	0.71	0.29	0.68	0.28	0.70	0.31	1.06	0.66	438.40
GR5	131073	1046848	1.28	0.48	1.28	0.48	1.30	0.55	2.05	1.29	833.20
GR6	262145	2095424	2.47	0.84	2.52	0.88	2.55	1.00	3.85	2.56	1556.20

TABLE 49. CPU time in seconds for 10^4 unit arc weight decreases on `Grid-PHard` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8193	63808	0.14	0.12	0.08	0.07	0.09	0.08	55.40
GR2	16385	129344	0.23	0.20	0.13	0.12	0.14	0.14	112.80
GR3	32769	260416	0.37	0.34	0.22	0.19	0.24	0.24	199.80
GR4	65537	522560	0.52	0.48	0.30	0.24	0.33	0.32	438.00
GR5	131073	1046848	0.88	0.84	0.49	0.40	0.53	0.55	833.00
GR6	262145	2095424	1.55	1.78	0.85	0.73	0.94	1.01	1555.80

TABLE 50. Heap size for each update (averaged over 10^4 unit weight changes) on `Grid-PHard` instances

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8193	63808	7.35	7.35	11.45	4.08
GR2	16385	129344	10.80	10.80	17.81	6.83
GR3	32769	260416	18.73	18.73	35.26	16.60
GR4	65537	522560	28.10	28.10	54.84	22.66
GR5	131073	1046848	53.16	53.16	107.66	50.38
GR6	262145	2095424	99.33	99.33	192.10	100.56

TABLE 51. CPU time in seconds for 10^4 unit arc weight increases on `Rand-4` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	32768	0.17	0.06	0.16	0.06	0.16	0.07	0.15	0.07	81.00
GR2	16384	65536	0.26	0.11	0.25	0.11	0.25	0.12	0.24	0.12	262.40
GR3	32768	131072	0.33	0.15	0.33	0.16	0.34	0.17	0.33	0.19	733.60
GR4	65536	262144	0.72	0.31	0.72	0.33	0.74	0.35	0.67	0.34	1926.40
GR5	131072	524288	0.53	0.24	0.55	0.27	0.57	0.29	0.55	0.33	4430.60
GR6	262144	1048576	1.07	0.45	1.09	0.48	1.13	0.53	1.03	0.53	9422.00
GR7	524288	2097152	1.18	0.50	1.20	0.53	1.24	0.58	1.20	0.67	21037.40
GR8	1048576	4194304	0.79	0.35	0.82	0.39	0.83	0.42	0.84	0.49	44635.80

TABLE 52. CPU time in seconds for 10^4 unit arc weight decreases on `Rand-4` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8192	32768	0.14	0.10	0.08	0.06	0.08	0.07	80.60
GR2	16384	65536	0.22	0.17	0.13	0.11	0.14	0.12	261.80
GR3	32768	131072	0.28	0.21	0.17	0.15	0.18	0.16	732.80
GR4	65536	262144	0.52	0.37	0.33	0.28	0.34	0.29	1927.00
GR5	131072	524288	0.41	0.30	0.26	0.22	0.27	0.23	4426.40
GR6	262144	1048576	0.74	0.51	0.47	0.38	0.48	0.40	9425.00
GR7	524288	2097152	0.80	0.56	0.51	0.42	0.52	0.43	20992.60
GR8	1048576	4194304	0.58	0.41	0.35	0.31	0.36	0.32	44661.60

TABLE 53. Heap size for each update (averaged over 10^4 unit weight changes) on Rand-4 instances

			G^{RR}, T^{RR}, T^{KT}		T^D	rhT^D
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8192	32768	9.90	9.90	11.49	1.64
GR2	16384	65536	10.36	10.36	12.35	1.98
GR3	32768	131072	10.40	10.40	12.45	2.12
GR4	65536	262144	15.42	15.42	17.62	2.18
GR5	131072	524288	12.27	12.27	14.77	2.56
GR6	262144	1048576	18.82	18.82	21.62	2.76
GR7	524288	2097152	19.64	19.64	23.64	4.08
GR8	1048576	4194304	14.22	14.22	17.40	3.15

TABLE 54. CPU time in seconds for 10^4 unit arc weight increases on `Rand-1:4` instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	512	65536	0.12	0.03	0.09	0.03	0.10	0.03	0.13	0.09	7.40
GR2	1024	262144	0.15	0.05	0.12	0.05	0.13	0.05	0.30	0.20	28.40
GR3	2048	1048576	0.12	0.05	0.11	0.05	0.10	0.05	0.23	0.16	91.20
GR4	4096	4194304	0.19	0.07	0.16	0.07	0.16	0.08	0.38	0.29	183.80

TABLE 55. CPU time in seconds for 10^4 unit arc weight decreases on `Rand-1:4` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	512	65536	0.12	0.11	0.05	0.06	0.07	0.07	7.40
GR2	1024	262144	0.16	0.15	0.09	0.09	0.10	0.10	28.40
GR3	2048	1048576	0.13	0.12	0.08	0.07	0.08	0.08	91.40
GR4	4096	4194304	0.19	0.19	0.11	0.11	0.12	0.12	183.20

TABLE 56. Heap size for each update (averaged over 10^4 unit weight changes) on `Rand-1:4` instances

			G^{RR}, T^{RR}, T^{KT}		T^D	rhT^D
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	512	65536	0.43	0.43	0.81	0.35
GR2	1024	262144	0.18	0.18	0.45	0.23
GR3	2048	1048576	0.06	0.06	0.14	0.07
GR4	4096	4194304	0.04	0.04	0.11	0.06

TABLE 57. CPU time in seconds for 10^4 unit arc weight increases on Rand-Len instances

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	131072	524288	0.72	0.31	0.72	0.32	0.74	0.34	0.65	0.33	4835.40
GR2	131072	524288	0.42	0.20	0.42	0.21	0.44	0.22	0.42	0.24	4257.60
GR3	131072	524288	0.59	0.27	0.59	0.27	0.61	0.30	0.55	0.30	4812.20
GR4	131072	524288	0.44	0.21	0.44	0.22	0.46	0.23	0.42	0.24	4880.80
GR5	131072	524288	0.62	0.28	0.62	0.28	0.65	0.31	0.57	0.30	4904.00

TABLE 58. CPU time in seconds for 10^4 unit arc weight decreases on Rand-Len instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	131072	524288	0.53	0.34	0.35	0.27	0.36	0.27	4834.00
GR2	131072	524288	0.32	0.23	0.21	0.17	0.22	0.18	4259.00
GR3	131072	524288	0.44	0.30	0.29	0.23	0.30	0.24	4809.00
GR4	131072	524288	0.34	0.24	0.22	0.19	0.23	0.19	4880.20
GR5	131072	524288	0.46	0.31	0.31	0.24	0.32	0.25	4903.40

TABLE 59. Heap size for each update (averaged over 10^4 unit weight changes) on Rand-Len instances

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	131072	524288	12.78	12.78	13.77	0.99
GR2	131072	524288	8.45	8.45	9.91	1.49
GR3	131072	524288	11.29	11.29	12.39	1.16
GR4	131072	524288	9.25	9.25	10.14	0.92
GR5	131072	524288	11.65	11.65	12.65	0.99

TABLE 60. Uacyc

Group	$\|V\|$	$\|E\|$	G_+^{RR}	rhG_+^{RR}	T_+^{RR}	rhT_+^{RR}	T_+^{KT}	rhT_+^{KT}	T_+^{D}	rhT_+^{D}	Dij
GR1	8192	131072	0.38	0.12	0.33	0.13	0.33	0.13	0.34	0.16	258.40
GR2	16384	262144	0.47	0.17	0.42	0.17	0.42	0.17	0.45	0.23	663.00
GR3	32768	524288	0.85	0.30	0.85	0.35	0.85	0.33	0.87	0.41	1630.80
GR4	65536	1048576	1.29	0.46	1.38	0.59	1.38	0.54	1.36	0.65	4018.60
GR5	131072	2097152	1.10	0.39	1.19	0.54	1.21	0.51	1.23	0.65	9223.60

TABLE 61. CPU time in seconds for 10^4 unit arc weight decreases on `Acyc-Pos` instances

Group	$\|V\|$	$\|E\|$	G_-^{RR}	rhG_-^{RR}	T_-^{RR}	rhT_-^{RR}	T_-^{KT}	rhT_-^{KT}	Dij
GR1	8192	131072	0.38	0.35	0.22	0.21	0.24	0.22	258.40
GR2	16384	262144	0.46	0.42	0.28	0.27	0.29	0.28	661.80
GR3	32768	524288	0.82	0.70	0.50	0.48	0.52	0.50	1630.80
GR4	65536	1048576	1.19	0.99	0.74	0.71	0.76	0.72	4018.80
GR5	131072	2097152	1.01	0.84	0.63	0.61	0.65	0.63	9238.40

TABLE 62. Heap size for each update (averaged over 10^4 unit weight changes) on `Acyc-Pos` instances

			G^{RR}, T^{RR}, T^{KT}		T^{D}	rhT^{D}
Group	$\|V\|$	$\|E\|$	Incr	Decr	Incr	Incr
GR1	8192	131072	5.36	5.36	6.37	1.07
GR2	16384	262144	5.29	5.29	6.33	1.07
GR3	32768	524288	7.73	7.73	9.09	1.44
GR4	65536	1048576	10.10	10.10	11.69	1.62
GR5	131072	2097152	8.06	8.06	9.53	1.48

TABLE 63. Time ratio between the standard and reduced heaps 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

In the remainder of this subsection, we discuss the experimental results for unit weight change. The kind of comparisons are the same done for random weight change and, because of this, the ratios are computed in the same fashion.

7.2.1. *Improvement obtained using reduced heaps algorithms.* Table 63 compares times for standard and reduced heaps versions of all dynamic shortest paths algorithms.

On average, for each class of instances, the reduced heap algorithms were able to reduce the computacional time in almost all simulations. For all the incremental algorithms G_+^{RR}, T_+^{RR} and T_+^{D} the reduced heaps variants were faster. The ratio varied from 1.33 for T_+^{D} algorithm in class `Grid-SSquare-S` to 5.11 for T_+^{RR} in class `Grid-SLong`. Considering the decremental algorithms, only three ratios were not favorable to the reduced heaps reduced heaps variant. Looking in the last row, on average all algorithms were able to reduce time using the reduced heaps technique. The average ratio varied from 1.05 for T_-^{KT} to 2.51 for G_+^{RR}.

7.2.2. *Time comparison between recomputing from scratch (`Dij`) and using a dynamic shortest path graph algorithm (G^{RR}).* Table 64 presents results for the comparison between the `Dij` and G^{RR} algorithm. As expected, the `Dij` takes much longer than the G^{RR}, varying from 10.33 times on Group `GR2` from class `Internet` to 83,637.14 times on Group `GR5` of class `Grid-SSquare-S`. It is interesting to observe that the ratios have changed considerable when compared with the ones found by random increments. This behavior is due to the range of weights chosen for tests. In [2], it was shown experimentaly that the times depends on the weight range that the algorithm is working on. Observe, for instance, the times presented in Tables 4 and 39. The instances are exactly the same, only their arc weight ranges are different.

7.2.3. *Comparison of performance for the dynamic shortest paths algorithms.* The tables comparing algorithms G^{RR} and T^{RR}, and T^{RR} and T^{KT} show similar results than the results presented on Tables 34 and 32, respectively. Table 65 presents results for the comparison between the performance of T_+^{RR} and T_+^{D} algorithms. The second column presents results for the time comparison and the third column compares heap sizes between algorithms. The heap comparison first compares the heap size for the standard versions of each algorithm and, then the heap size for standard and reduced heaps versions of algorithm T_+^{D}. This table shows explicitly some examples where algorithm T^{D} does not perform well

TABLE 64. Ratio between the time spent by algorithms Dij and G^{RR} for updating 10^4 unit weight changes on each group 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	1383.33	719.67	18.01	362.75	521.29	61.67	7747.92	672.92
GR2	10.33	2722.22	1261.63	17.64	452.21	1092.08	182.05	11416.35	1430.67
GR3	10.91	6962.96	3104.17	16.33	468.15	2427.81	730.40	9340.97	1955.40
GR4	15.59	26753.57	8348.11	15.48	709.06	3117.64	976.06	12482.10	3230.47
GR5	20.58	83637.14	21329.46	14.91	770.68	9422.34		9030.76	8749.76
GR6	14.47		44769.75	15.00	773.36	10424.23			
GR7	17.14		86031.82	13.86		21163.14			
GR8	22.84					65180.58			
GR9	25.41								
GR10	13.27								
GR11	16.67								
GR12	24.31								
GR13	23.96								

TABLE 65. Ratio between the time spent by the T^D and T^{RR} algorithms for updating 10^4 unit weight changes

	Time		Heap Size	
Class	std	rh	std	rh
Internet	0.91	1.06	1.02	0.02
Grid-SSquare-S	1.17	1.55	1.03	0.00
Grid-SWide	0.97	1.10	1.06	0.06
Grid-SLong	0.95	1.15	1.07	0.07
Grid-PHard	1.47	2.35	1.83	0.44
Rand-4	0.98	1.15	1.18	0.16
Rand-1:4	2.09	3.39	2.26	0.50
Rand-Len	0.94	1.09	1.11	0.10
Acyc-Pos	1.04	1.21	1.18	0.16
Average	1.17	1.56	1.30	0.17

TABLE 66. Time ratio between the incremental and decremental implementations of the algorithms spent for updating 10^4 unit arc weight changes

Class	G^{RR}	rhG^{RR}	T^{RR}	rhT^{RR}	T^{KT}	rhT^{KT}
Internet	1.36	1.08	1.72	0.93	1.59	0.59
Grid-SSquare-S	2.00	0.91	1.39	1.11	2.43	1.29
Grid-SWide	1.21	1.06	1.72	1.20	1.67	1.00
Grid-SLong	1.90	1.03	2.67	1.21	2.56	0.77
Grid-PHard	1.34	0.61	2.29	1.12	2.13	0.96
Rand-4	1.31	0.78	2.14	1.18	2.14	1.19
Rand-1:4	0.97	0.35	1.46	0.61	1.35	0.56
Rand-Len	1.33	0.89	2.02	1.19	2.04	1.25
Acyc-Pos	1.05	0.42	1.68	0.74	1.63	0.68
Average	1.39	0.79	1.90	1.03	1.95	0.92

when compared with other dynamic shortest paths algorithms. Since these experiments are applied on instances in a small range, algorithm T_+^D handles many unaffected nodes as though they were. The worst performance of this algorithm was observed on `Rand-1:4`. The algorithms took more than twice the time than T_+^{RR} spent in the standard implementation and 3.39 times more for the reduced heaps algorithm. Looking at the heap size values, we can see that for the standard algorithm, its heap is 2.26 larger, on average, than the heap of algorithm T_+^R. For the reduced heaps implementation, while algorithm rhT_+^{RR} does not insert any node into the heap, algorithm T^D inserts up to 50% of the nodes inserted by its standard implemetation.

7.2.4. *Time comparison between incremental and decremental implementations of the dynamic algorithms.* Table 66 presents the ratio between times spent by the incremental and decremental implementations of the algorithms.

For the standard algorithms, the results presented in this table are similar to the results presented for random weight increments in Table 35. For the reduced heaps implementations, on average the decremental algorithms were faster for algorithms G_{-}^{RR} and T^{KT} and slightly slower for algorithm T^{RR}.

8. CONCLUDING REMARKS

This paper introduces a technique for reducing the heap size of dynamic shortest path algorithms (DSP). This technique can be used for both incremental and decremental algorithms. The basic idea is to make the updates with a smaller set of nodes than in standard implementations of these algorithms. Consider a change in the weight of arc $a = (\overrightarrow{u,v})$ and let the distance of node u to the destination change by an amount $\triangle$. The standard implementations of these algorithms insert into the heap all nodes whose distance labels have changed. We denote by Q this set of nodes. The reduced heap variants of these algorithms insert into the heap only the subset of Q whose distance labels have changed by an amount smaller than $\triangle$. 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 1.79, while for unit weight changes, the largest speedup was 5.11.

Comparing Dijkstra's algorithm with 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.

The comparison between dynamic shortest path algorithms has shown that updating trees is lightly faster than updating graphs for random and unit weight changes. Furthermore, for the instances considered in this paper, on average any gain that could be achieved while scanning the outgoing links in T^{KT}, is washed out by the additional computational effort associated with maintaining the special tree proposed by King and Thorup [18].

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 incremental and decremental algorithms, even if in a few examples the reduced heap variant took longer than the corresponding standard version. In case the application does not need a shortest path graph, updating a shortest path tree is faster. The best choice would be the combination of the incremental algorithm T_{+}^{D} with the decremental algorithm T_{-}^{RR}, if the instance is generated with weights from a wide range. Considering weights from a short range, the combination of T_{+}^{RR} and T_{-}^{RR} is recommended. Finally, we conclude that the performance of an algorithm depends largely on the instance, and also on its size.

REFERENCES

[1] L. S. Buriol, M. G. C. Resende, C. C. Ribeiro, and M. Thorup. A hibrid genetic algorithm for the weight setting problem in ospf/is-is routing. *Networks*, 2003. under review.

[2] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[3] C. Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs*. PhD thesis, Department of Computer and Systems Science, University of Rome "La Sapienza", April 2001.
[4] C. Demetrescu, S. Emiliozzi, and G. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. Technical report, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Rome, Italy, 2003.
[5] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. *Algorithm Engineering*, pages 218–229, 2000.
[6] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *FOCS 2001*, pages 260–267, 2001.
[7] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03)*, pages 159–166, 2003.
[8] R. Dionne. Etude et extension d'un algorithme de murchland. *INFOR*, 16:132–146, 1978.
[9] S. Even and Y. Shiloach. An on-line edge-deletion problem. *J. of the Asso. For Comp.*, 28:1–4, March 1981.
[10] B. Fortz and M. Thorup. Increasing internet capacity using local search. Technical report, AT&T Labs Research, 180 Park Avenue, Florham Park, NJ 07932 USA, 2000.
[11] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest path problem. *ACM J. of Exp. Alg.*, 3, 1998. article 5.
[12] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem. *Proceedings of the 7th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 212–221, 1996.
[13] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single-source shortest path trees. *Algorithmica*, 22(3):250–274, 1998.
[14] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:351–381, 2000.
[15] S. Fujishige. A note on the problem of updating shortest paths. *Networks*, 11:317–319, 1981.
[16] G. Gallo. Reoptimization procedures in shortest path problems. *Rivista di Matematica per le Scienze Economiche e Sociali*, 3:3–13, 1980.
[17] S. Goto and A. Sangiovanni-Vincentelli. A new shortest path updating algorithm. *Networks*, 8:341–372, 1978.
[18] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Computing and Combinatorial Conference COCOON in LNCS 2108*, pages 268–277. Springer-Verlag, 2001.
[19] J. D. Murchland. *A fixed matrix method for all shortest distances in a directed graph and for the inverse problem*. PhD thesis, University of Karlsrube, 1970.
[20] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. of Algorithms*, 21:267–305, 1996.
[21] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

(L.S. Buriol) UNICAMP, CAMPINAS, SP, BRAZIL
E-mail address, L.S. Buriol: buriol@densis.fee.unicamp.br

(M.G.C. Resende) INTERNET AND NETWORK SYSTEMS RESEARCH, AT&T LABS RESEARCH, 180 PARK AVENUE, ROOM C241, FLORHAM PARK, NJ 07932 USA.
E-mail address, M.G.C. Resende: mgcr@research.att.com

(M. Thorup) INTERNET AND NETWORK SYSTEMS RESEARCH, AT&T LABS RESEARCH, 180 PARK AVENUE, ROOM B288, FLORHAM PARK, NJ 07932 USA.
E-mail address, M. Thorup: mthorup@research.att.com