Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study [⋆]

Reinhard Bauer and Dorothea Wagner

Karlsruhe Institute of Technology (KIT), Germany
{rbauer,wagner}@ira.uka.de

Abstract. A dynamic shortest-path algorithm is called a batch algorithm if it is able to handle graph changes that consist of multiple edge updates at a time. In this paper we focus on fully-dynamic batch algorithms for single-source shortest paths in directed graphs with positive edge weights. We give an extensive experimental study of the existing algorithms for the single-edge and the batch case, including a broad set of test instances. We further present tuned variants of the already existing SWSF-FP-algorithm being up to 15 times faster than SWSF-FP. A surprising outcome of the paper is the astonishing level of data dependency of the algorithms.

1 Introduction

The *single-source shortest-path problem* is a fundamental graph problem with many real-world applications, such as routing in road networks, routing/data harvesting in sensor networks and internet routing using link state protocols (for example OSPF and IS-IS). In these applications shortest-path trees are stored and have to be updated whenever the underlying graph undergoes changes [1–4].

Algorithms that update the trees without a full recomputation from scratch are called *dynamic single-source shortest-path algorithms*. Such algorithms slightly differ in the type of their output. Some store only the distances from the source, while others additionally store a shortest-path tree or the shortest-path subgraph. Some of the algorithms known in the literature are only able to cope with the update of one edge at a time, while others can perform *batch updates*, i.e. update the shortest-path information after multiple edges have simultaneously changed their weight. Batch updates naturally arise in real-world applications: traffic jams usually affect a set of edges, updating the information in sensor networks usually requires flooding, which is done in intervals big enough so that more than one link in the network has changed.

We consider edge insertions and deletions as special cases of weight changes: Deletions correspond to weight increments to infinity, while insertions are weight decrements from infinity. An algorithm is called *fully dynamic* if both weight increases and decreases are supported, and *semi-dynamic* if only weight decreases or only increases are supported.

Aims. In this paper we focus on fully-dynamic batch updates for directed graphs with positive edge weights. In order to compare the different approaches, the only requirement that we make regarding the tested algorithms is that they update the distance vector. We furthermore demand that the algorithms be able to cope with edge insertions and deletions. For our experimental study, we apply integer edge weights.

Up to now, the experimental knowledge on this topic is quite sparse. The existing experimental work focuses on very specific datasets. Hence, our first interest is to study the performance when applying single-edge updates to graphs with different structural properties. Do algorithms behave uniformly or is there a high degree of data dependency? This question will be answered in an experimental study, including a broad set of real-world and synthetic instances. For batch updates more fundamental open questions exist: it is not even known if it is useful to process a set of updates as a batch. Intuition tells us that edge updates that are far away from each other do not interfere regarding their impact on the shortest paths in the graph. So it seems that these updates can be handled iteratively. On the other hand, updated edges with a strong interference should be processed in a batch (paying with some computational overhead). We will

---

show that this intuition is right, and how to formalize the interference of updated edges through a simple approach. Finally, we want to pay some attention to the already-existing SWSF-FP-algorithm. This algorithm has been stated with regard to mainly theoretical considerations. We want to test if it can be implemented more efficiently and if combinations with other algorithms yield additional speed-up.

Related work. Ramalingam and Reps [5] introduce the batch algorithm SWSF-FP, Narvaez et al. [1] propose the NARVÁEZ-framework containing six single-edge update algorithms and a modification to the framework leading to the according batch algorithms. Pure single-edge update algorithms are RR [6] (due to Ramalingam and Reps) and FMN [7] (by Frigioni et al). All these algorithms are described shortly in Section 3. Buriol et al [8] present a heuristic technique to speed up RR-like approaches. The technique is similar to techniques used in the NARVÁEZ-framework but does not support edge insertions or deletions. Furthermore, in [8] the RR algorithm is adapted to maintain a special (shortest-path) tree proposed in [9].

There is no algorithm known in the literature for which the worst case is asymptotically better than recomputing the new solution from scratch. In the original works the algorithms described in Section 3 are theoretically analyzed with respect to different measures. These measures mostly depend on the size of the subgraph for which the shortest-path subgraph changes. An overview of these complexity results can be found in Appendix A.

There is some work on the variant of the problem where edge weights may also be negative. In [6] the algorithm RR is adapted to cope with the existence of negative cycles, in [10] the same is done for the algorithm FMN. In [11] Demetrescu gives some algorithms for that problem. These algorithms use the reweighting technique, which incorporates a complete Dijkstra run on the graph (with changed edge weights). Hence, this approach is impractical for the problem with non-negative edges.

A well-studied related problem is the *fully dynamic all-pairs shortest-path problem*, in which the distances between all pairs of nodes have to be maintained while the graph undergoes changes. See [12] for a survey on the problem.

There is only few experimental work on this topic, all concentrating on single-edge updates. In [13] the algorithms FMN, RR and a full recomputation from scratch are compared on two instance classes: Erdös-Rényi graphs, where updates are chosen uniformly at random and a graph representing the internet on the AS-level, where updates simulate the failure and recovery of the links. In [1] the algorithms of the NARVÁEZ-framework are evaluated on graphs originating from a generator. This generator randomly places nodes on a grid and connects them by edges with probability that exponentially decreases with the distance of the nodes. The generator does not seem to be available any more. In [14] the algorithms SWSF-FP, RR, FMN, NARVÁEZ and full recomputation from scratch using DIJKSTRA, BELLMAN FORD, D'ESOPO PAPE are evaluated with single-edge updates on Erdös-Rényi-like graphs. In [3] one algorithm of the NARVÁEZ-framework is evaluated on random single-edge updates on a graph representing the road-network of Western Europe. In [8], the algorithm RR as well as seven variants thereof are evaluated on a real world AT&T IP network, synthetic internet-related graphs and a large set of other synthetic instances, namely those of [15] with non-negative edge lengths.

Overview. This paper is organized as follows. Section 2 states basic definitions and formally introduces the problem. Section 3 reviews the existing algorithms. Section 4 presents our tuned variants of the SWSF-FP-algorithm, while an extensive experimental study of these algorithms on synthetic and real-world data is given in Section 5. The paper ends with a conclusion in Section 6.

2   Problem Statement

Let $G = (V, E)$ be a directed graph with n nodes and m edges and a non-negative length function len : $V \times V \rightarrow \mathbb{R}^+ \cup \{\infty\}$. Let $s \in V$ be an arbitrary but fixed *source*. With $d(v)$ we denote the length of a shortest s-v-path in G for any $v \in V$.

A *batch update* is a set of *edge modifications* on G which can be edge insertions, edge deletions, edge weight increases and edge weight decreases (that keep the length function non-negative). We want to maintain a distance vector $D[]$ containing $d(v)$ for each node v in a dynamic environment where G is undergoing batch updates. After

each batch update, $D[]$ (and possible required auxiliary data needed by the recomputation algorithm) has to be updated accordingly.

Throughout the text, we will cope with the recomputation of $D[]$ and the auxiliary data when *one* concrete batch update is given (because of the recomputation of the auxiliary data the algorithms are able to handle following updates). We write len_{old} for length function and d_{old} for distance before the update. Accordingly we write len for length function and d for distance after the update. For notational convenience, we consider inserted or deleted edges to be existing in the original and the updated graph and set the edge length to infinity, if necessary.

Some of the following algorithms are designed to handle only one edge modification at a time. Obviously, repeated application of these algorithms also solves the batch case. We call such algorithms *iterative algorithms* while the others are called *batch algorithms*. Iterative algorithms can be split into two parts: the *incremental* part handles edge insertions and weight decreases while the *decremental* part handles edge deletions and weight increases. This terminology can be unintuitive on a first glance but originates from the point of view that the graph increases when edges are inserted.

3   Description of Algorithms

In this section, we describe the algorithms evaluated in our experimental study. Each algorithm includes a main phase in which a min-based priority queue Q is used to recompute the distances in a Dijkstra-like fashion but on a smaller subgraph. We only give a rather short description for each algorithm. Complete descriptions including pseudocodes can be found in the original papers.

The input of the algorithms is the outdated distance vector $D[]$, the graph G, the original length function len_{old}, the batch update $U = (u_1, \ldots u_k)$ and some auxiliary data which will be described for each algorithm separately. The output is the updated distance vector $D[]$ and the updated auxiliary data.

Notation. Given the outdated distance vector $D[]$, we say we *relax* an edge (u, v) when we check if $D[v] > D[u] + \text{len}(u, v)$. We say we *relax and update* an edge (u, v) when we set $D[v] := \min\{D[v], D[u] + \text{len}(u, v)\}$. An edge (u, v) is said to be *consistent* if $D[v] = \text{len}(u, v) + D[u]$ and *underconsistent* if $D[v] > \text{len}(u, v) + D[u]$. The *consistent value* $\text{con}(v)$ of a node v is

$$\text{con}(v) := \begin{cases} \min_{(u,v) \in E} \{D[u] + \text{len}(u,v)\} & , v \neq s \\ 0 & , v = s \end{cases}$$

A node is said to be *consistent* if $D[v] = \text{con}(v)$ and to be *over-consistent* if $D[v] > \text{con}(v)$. As convention, we use $\min \emptyset := \infty$.

3.1   Algorithm of Ramalingam and Reps

Ramalingam and Reps [6] describe the iterative algorithm RR that handles only edge insertions and deletions. It can be directly transferred to an algorithm that works with weight increases and decreases. We will state this variant.

Auxiliary Data. The approach maintains the following auxiliary data: for each edge e, the information if e lies on the shortest-path subgraph *SD* rooted at s and for each node n, the number $\text{indeg}(n)$ of incoming edges of n in *SD* (indeg is adjusted whenever *SD* changes).

Incremental Part. Given the edge (x, y) with weight decrease, we first update $\text{len}(x, y)$ and relax (x, y). If (x, y) is consistent we insert the edge (x, y) in *SD*. If (x, y) is underconsistent we set $D[y] := D[x] + \text{len}(x, y)$ and insert y with priority $D[y]$ into Q.

Main phase. We perform as follows until Q is empty: We extract and delete the minimum node v from Q. Then, each edge with target v is removed from *SD* and each consistent incoming edge (u, v) is inserted into *SD*. Afterwards, for each outgoing consistent edge (v, w) we insert (v, w) into *SD*. For each outgoing underconsistent edge (v, w) we set $D[w] := D[v] + \text{len}(v, w)$ and insert w with priority $D[w]$ in Q.

Decremental Part. Given the edge (x,y) with weight increase, we first update len(x,y). If $(x,y) \notin SD$, nothing is to do. Otherwise, we remove (x,y) from SD. Let $B \subseteq SD$ be the subgraph of SD that is not connected to s any more. We delete each edge $e \in B$ from SD. For each node $b \in B$, we set $D[b] := \min\{D[a] + \text{len}(a,b) \mid (a,b) \in E, a \notin B\}$. If $D[b]$ is not infinity we insert b with priority $D[b]$ into Q.

Main phase. Now, we perform as follows until Q is empty: We extract and delete the minimum node v from Q. Each incoming consistent edge (u,v) is inserted into SD. For each outgoing underconsistent edge (v,w), we assign $D[w] := D[v] + \text{len}(v,w)$ and insert w with priority $D[w]$ into Q.

3.2 Algorithm of Frigioni et al

The FMN-algorithm of Frigioni et al. [7] is an iterative algorithm similar to the algorithm RR that uses more complex auxiliary data to obtain better theoretical worst case bounds. The approach relies on the existence of a *k-bounded accounting function* on G, which is a mapping $K : E \to V$ such that for each edge (u,v) the node $K(u,v)$ is either u or v and such that for each node n, no more than k edges are n-valued. We use the constructive 2-approximation algorithm described in [10] for finding a k-bounded accounting function on G.

Auxiliary Data. The algorithm stores a k-bounded accounting function K of the graph. Given a node x, the set of edges (x,y) with $K(x,y) = x$ is called ownership(x). The set of the other edges adjacent to x is called not_ownership(x). The *backward level* of an edge (z,q) and of vertex q relative to vertex z, is the value b_level$_z(q) := D(q) - \text{len}(z,q)$. The *forward level* of an edge (z,q) and of vertex q relative to vertex z, is the value f_level$_z(q) := D(q) + \text{len}(z,q)$.

For each node x, the algorithm stores two priority queues, each containing the edges in not_ownership(x). The queue B_x is max-based. The priority of an edge (x,y) is b_level$_x(y)$. The queue F_x is min-based. The priority of an edge (x,y) is f_level$_x(y)$. In the original version a shortest-path tree is additionally maintained by storing a parent node $P[n]$ for each node $n \neq s$.

Incremental Part. Given the edge (x,y) with weight decrease, we first update len(x,y) and the queues $B(x)$, $F(x)$, $B(y)$ and $F(y)$. Then, if (x,y) is underconsistent we set $D[y] = D[x] + \text{len}(x,y)$ and insert y with priority $D[y]$ into Q.

In the main phase we perform as follows until Q is empty: We extract and delete the minimum node v from Q and update the queues $B(v)$ and $F(v)$. Afterwards we check for each edge (v,w) in not_ownership(v) and for each edge (v,w) in ownership(v) with b_level$_v(w) > D[v]$ if (v,w) is underconsistent. In that case we set $D[w] = D[v] + \text{len}[v,w]$ and insert w with priority $D[w]$ into Q.

Decremental Part. Given the edge (x,y) with weight increase, we first update len(x,y) and the queues $B(x)$, $F(x)$, $B(y)$ and $F(y)$. Let SD be the tentative shortest-path subgraph that is given implicitly by all consistent edges. Let $B \subseteq SD$ be the subgraph of SD that is not connected to s. The computation of B exploits the auxiliary data in a way similar to the main phase of the incremental part. We leave out a detailed description for that part and refer to [7] for a full description. For each node $b \in B$, we set $D[b] := \min\{D[a] + \text{len}(a,b) \mid (a,b) \in E, a \notin B\}$ and insert b with priority $D[b]$ into Q.

Main phase. Now, we perform as follows until Q is empty: We extract and delete the minimum node v from Q and update the queues $B(v)$ and $F(v)$. For each outgoing underconsistent edge (v,w) we assign $D[w] := D[v] + \text{len}(v,w)$ and update the priority of w in q to $D[w]$.

3.3 Algorithm of Narvaez et al

Narvaez et al. [1] propose a batch algorithm incorporating two degrees of freedom. One degree of freedom is the choice of Q which does not necessarily need to be a priority queue but only has to maintain the operations INSERT and EXTRACT. Narvaez et al propose a FIFO queue (Bellman-Ford like approach), a heap (implemented as binary heap or linked list) and a D'Esopo-Pape like approach. The other degree of freedom consists of two different variants for the main phase of the algorithm which we will describe below. We will refer to the diffent variants as NAR{1st, 2nd}{HEAP, BF, PAP}. The main idea of the NARVÁEZ-framework is to early-propagate distance changes through the tentative shortest-path tree.

Auxiliary Data. The algorithm maintains a shortest-path tree T of the graph by storing the parent node $P[v]$ for each node v. With $B(u,v)$ we denote the set of nodes that is contained in the branch of T that starts with (u,v) minus $\{u\}$. During the execution of the algorithm, a tentative parent $P'[n]$ is stored for each node n in Q.

Algorithm. *Initialization Phase.* In this phase edge updates are handled iteratively. First, for each edge (u,v) with weight increase λ, we update $\text{len}(u,v)$ and set $D[x] := D[x] + \lambda$ for each x in $B(u,v)$. By N_{inc} we denote the set of all vertices with previously incremented weight. Afterwards we relax each edge with target in N_{inc}, insert each overconsistent node $n \in N_{inc}$ with priority $\text{con}(n)$ in Q and update $P'[n]$ accordingly.

Now, for each edge (u,v) with weight decrease λ, we update $\text{len}(u,v)$ and set $D[x] := D[x] - \lambda$ for each x in $B(u,v)$. By N_{dec} we denote the set of all vertices with previously decremented weight. Afterwards we relax each edge with source in N_{dec}, insert each node v for which $d_v := \min\{D[u] + \text{len}(u,v) \mid u \in N_{dec}\} < D[v]$ with priority d_v in Q and update $P'[v]$ accordingly.

Main Phase, 1st Variant. We perform as follows until Q is empty: We extract and delete the next node v (according to the actual choice of Q) from Q, set $D[v]$ to be priority of v in Q and $P(v)$ to be $P'(v)$. Afterwards we relax each outgoing edge (v,w). If (v,w) is underconsistent we insert w with priority $D[v] + \text{len}(v,w)$ in Q and set $P'[w] = v$. If w is already in Q we only update the priority and $P'[w]$.

Main Phase, 2nd Variant. We perform as follows until Q is empty: We extract and delete the next node v (according to the actual choice of Q) from Q. With $\text{key}(v)$ we denote the priority of v in Q. We set $\lambda := \text{key}(v) - D[v]$ and $P(v) := P'(v)$. Now, we identify the set N of all descendants of v in T. Subbranches that start with nodes u that are already with $\text{key}(u) < D[u] + \lambda$ in Q are not included in N. Each node in N which is with $\text{key}(u) >= D[u] + \lambda$ in Q is removed from Q. Afterwards we relax each edge (v,w) outgoing from a node $v \in N$. If (v,w) is underconsistent we insert w with priority $D[v] + \text{len}(v,w)$ in Q and set $P'[w] = v$. If w is already in Q we only update the priority and $P'[w]$.

4 Tuning SWSF-FP

In this section we will review the algorithm SWSF-FP which is due to Ramalingam and Reps [5] and give some tuned variants of it.

4.1 SWSF-FP.

For each node v, a label $d[v]$ is given. Initially, $d[]$ equals $D[]$ (in order to save time for the copy process we implemented $d[]$ as auxiliary data). We say we *adjust* an inconsistent node v when we set $d[v] := \text{con}(v)$ and insert v with priority $\min(D[v], d[v])$ in Q. In case v is already in Q we only update its priority. We adjust a consistent node v when we remove it from Q. If v is not in Q we do nothing.

Initially, we adjust each node which is target of an edge in U. *Main Phase.* While Q is not empty, we perform as follows: We extract and delete the minimum node w from Q. If $d[w] < D[w]$ we set $D[w] := d[w]$ and adjust each outgoing neighbor of w. If $d[w] > D[w]$ we set $D[w] := \infty$ and adjust w and each of its outgoing neighbors.

4.2 Tuned SWSF

This algorithms basically works like the SWSF-FP-algorithm, but with less computational effort. When performing SWSF-FP we have to relax all incoming edges of a node n in order to compute $\text{con}(n)$. Tuned SWSF relaxes fewer of such incoming edges: When we adjust an outgoing neighbor v of a node w with $d[w] < D[w]$, we compute $\text{con}(v)$ by $\min\{d[w] + \text{len}(v,w), d[v]\}$. The same strategy works in the initialization phase when we compute $\text{con}(n)$ for a node n that is the target node of an edge with decreased edge weight. When we adjust an outgoing neighbor v of a node w with $d[w] > D[w]$, we set $D_{old} := D[w]$ and $D[w] := \infty$. We can skip v when $D_{old} + \text{len}(w,v) \neq d[v]$. The same strategy holds in the initialization phase for target nodes of edges with increased weight. The pseudocode of this algorithm can be found in Appendix B.

4.3 TUNED SWSF RR

This variant enhances the algorithm TUNED SWSF with a technique adapted from the RR-algorithm. For each node v, a label indegree(v) is given indicating the number of edges (u,v) with $D[u] + \text{len}(u,v) = d[v]$. Further, for each edge (u,v) a boolean label $DAG(u,v)$ is given indicating if $D[u] + \text{len}(u,v) = d[v]$. The labels indegree and DAG are directly updated whenever len, $D[]$ or $d[]$ change. The algorithm performs like TUNED SWSF with the following difference: After a node v with $d[v] > D[v]$ is extracted from Q only those edges (v,w) have to processed for which indegree$[w] = 0$.

4.4 TUNED SWSF NAR

This variant enhances the algorithm TUNED SWSF with a technique adapted from the NARVÁEZ-algorithm. For each node v that is not the source, a label $P(v)$ is given pointing at another node, such that $D[P(v)] + \text{len}(P(v),v) = d(v)$. At the beginning a shortest-path tree T on the original graph is given implicitly by this label. The main phase of the algorithm works like the main phase of TUNED SWSF. The initialization phase works as follows: First, we update the edge weights. We denote by A the set of all nodes that lie behind a target node of an updated edge. Then, we update the distances $D[]$ of nodes in T according to the new edge weights (but to the original shortest-path tree T). This can be implemented such that for each node $v \in A$, the distance $D[w]$ is updated at most once. Then, we set $d[v] = \text{con}(v)$ for each node v which either is contained in A or has a neighbor in A. Finally we insert each node with $d[v] \neq D[v]$ with priority $\min\{d[v], D[v]\}$ in Q.

5 Experiments

In this section, we present an experimental evaluation of the algorithms described above. Our implementation is written in C++ (using the STL at some points). Our tests were executed on one core of an AMD Opteron 2218, running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 32 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2, using optimization level 3.

For each experiment, 1000 update instances were generated. To properly measure the speed-ups, a full Dijkstra run is performed directly after each update and the speed-up compared to that run (i.e. the time needed by Dijkstra's algorithm divided by the time needed by the update algorithm) is computed. Finally we compute the mean value of these speed-ups. Thus, measurement disturbances due to background processes etc are avoided as much as possible. The absolute running times of Dijkstra's algorithm for each instance is given in Table 12. For Tables 1-4 we showed in bold letters all algorithms whose performance was at least 85% of the best observed performance.

In our experiments we evaluated all previously described algorithms. We did not include the heuristic of Buriol et al [8] because it does not support edge insertions or deletions. Further, we did not include the D'Esopo-Pape variants of the NARVÁEZ-framework because pretests had revealed some instances with extremely bad performance with this approach. To gain further insights in the performance of the batch-algorithms (NARVÁEZ and TUNED SWSF), we executed these two times: one time with processing the edges in batch, as stated originally and one time with iteratively processing the edges one after another. We refer to these approaches as ITNAR and ITTUNED SWSF. Note that we refer to the NARVÁEZ-framework as a batch algorithm while it actually does not perform updates completely in a batch: its initialization phase handles edge updates iteratively but the following main phase handles all updates in a batch.

5.1 Graph Instances

We use the following test instances for our study.

UNIT DISK. During the last years, the field of sensor networks has received wide attention. We evaluate so called unit disk graphs, which are widely used for experimental evaluations in that field. Given n and m, a unit disk graph is generated by randomly assigning each of the n nodes to a point in the unit square of the Euclidean plain. Two nodes are connected by an edge in case their Euclidean distance is below a given radius. This radius is adjusted such that the resulting graph has approximately m edges. As edge weights we use the Euclidean distance to the power of 0 (hop length), 1 (Euclidean distance) and 2 (energy). All tested graphs consist of 15 000 nodes.

RAILWAY. The graph RAIL represents the condensed railway network of Europe, based on timetable information, provided by the company HaCon [16] for scientific use. Nodes represent stations while edges represent direct connections between the stations. The edge weight corresponds to the average travel time between two stations. The graph has 29 578 nodes and 159 914 edges.

AS-GRAPH. The graph AS-HOP represents the internet as of 2008/3/26 on the AS-level, i.e. each node corresponds to an autonomous system and edges represent connections between autonomous systems. This graph is taken from the Routeviews project page [17]. It has 27 909 nodes and 114 474 edges. The edge weight is 1 for each edge. The same graph with edge weights chosen uniformly at random from the interval $[1, 1000]$ is called AS-RAN.

CAIDA. This dataset represents the internet on the router level, i.e. nodes are routers and edges represent connections between routers. The network is taken from the CAIDA webpage [18] and has 190 914 nodes and 1 215 220 edges. The edge weight is 1 for each edge.

ROAD. We evaluate three road networks provided by the PTV AG [19]. DEU represents Germany with 4 378 447 nodes and 10 968 884 edges, NLD the Netherlands with 946 632 nodes and 2 358 226 edges and LUX represents Luxembourg with 30 647 nodes and 75 576 edges. The edge weights are the corresponding travel times with speed profile 'slow car'.

GRID. These are fully synthetic graphs based on two-dimensional square grids. The nodes of the graph correspond to the crossings in the grid. There is an edge between two nodes if these are neighbors on the grid. Edge weights are randomly chosen integer values between 1 and 1000. GRID 100 is a 100x100 grid graph while GRID 300 is a 300x300 grid graph.

5.2 Data Structures

For our tests we always applied integer valued edge weights. We always apply a binary heap when a priority queue is needed. We use the graph datastructure that is also applied in [20–22]. There, the datastructure has experimentally shown to perform well in the context of shortest-path computation on sparse graphs.

The input graph $G = (V, E, \text{len})$ is stored in forward and reverse representation. It is represented by two arrays. The array EDGES stores the edges. Each entry $e = (n, \text{len}, \text{reverse}, for, back)$ in EDGES consists of the following data elements: a target node n, an edge weight len, the index of another entry in EDGES reverse, a binary value for and a binary value back. Moreover, we associate (but do not store) with each entry e a node source(e). For each edge (u, v) in G, there are two entries in EDGES: $e_1 = (v, \text{len}(u, v), \text{false}, \text{true}, id(e_2))$ with source$(e_1) = u$ and $e_2 = (u, \text{len}(u, v), \text{true}, \text{false}, id(e_1))$ with source$(e_2) = v$ where $id(e_i)$ denotes the index of the entry e_i in EDGES. The entries in EDGES are ordered by source$()$, ties are broken arbitrarily.

The array NODES stores the nodes. For each node, there is one entry in NODES. An entry of a node n consists of the index of the first entry e in EDGES for which source$(e) = n$. Nodes are identified by their index in NODES while we identify an edge (u, v) by the index of its corresponding entry in EDGES for which source$(u, v) = u$.

In case of undirected graphs or directed graphs in which the updates are always performed on a pair of edges (u, v) and (v, u) by the same weight difference we always use the following compression: Two entries of the form $e_1 = (v, len(u, v), false, true, id_1)$ and $e_2 = (v, len(u, v), true, false, id_2)$ with source$(e_1) = $ source(e_2) are compressed to one entry $e_3 = (v, len(u, v), true, true, id_3)$ where id_3 is the index of the edge that results from compressing the edges with index id_1 and id_2.

In order to dynamically insert or delete edges one has to rearrange the array EDGES and adjust the information in NODES. To prevent that as far as possible some dummy edges are inserted in EDGES and some extra information is maintained to organize the dummy edges. We do not describe that in full detail as this has to be equally done for each update algorithm and for a full recomputation from scratch.

5.3 Assessing the Performance of the Algorithms

Let $U = \{u_1, \ldots, u_k\}$ be a set of updated edges. By $\Delta(G, U)$ we denote the number of vertices in V for which the distance from the source changes due to the update. The *expected speed-up* of an update is the number of vertices in the graph divided by $\Delta(G, U)$. This value is roughly the speed-up we expect from a good update algorithm. Of course, speed-ups can even be higher for special instances. It experimentally turned out that when the topology of the original shortest-path tree does not change, the propagation of the updated edge's weights through the tree can gain a large speed-up.

When we want to measure the difficulty of an update for an iterative algorithm we consider $U = (u_1, \ldots, u_k)$ to be ordered. We perform the updates u_i iteratively in the given ordering (always additional to the former updates) obtaining a sequence of graphs $G = G_0, G_1, \ldots, G_k$. We write $\delta(G, (u_1, \ldots, u_k)) := \sum_{i=0}^{k-1} \Delta(G_i, \{u_{i+1}\})$. We have the following hypothesis: the smaller the difference between $\Delta(G, U)$ and $\delta(G, U)$ is, the less do the contained single-edge updates interfere and it is reasonable to use an iterative algorithm for the update. If the difference is great, an iterative algorithm would change the distance of many nodes multiple times. Hence, it is more appropriate to use a batch algorithm. The experimental evaluation will support our hypothesis.

The algorithm ITTUNED SWSF can be seen as a very simple iterative approach incorporating no extra features like early edge-weight propagation, etc. Figure 1 shows the runtime improvement of TUNED SWSF over ITTUNED SWSF compared with $\Delta(G, U)/\delta(G, U)$ for all batch experiments performed.
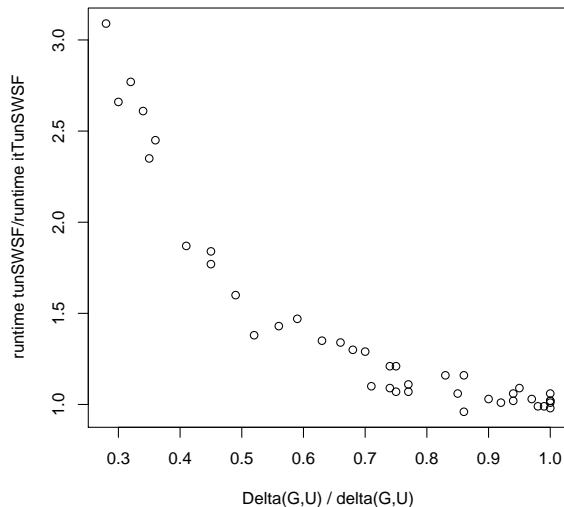
Fig. 1. Value of $\Delta(G, U)/\delta(G, U)$ (x-axis) and runtime of TUNED SWSF / runtime of ITTUNED SWSF (y-axis) for all batch-experiments

5.4 Space-Saving Implementation of RR

The algorithm RR needs to maintain the shortest-path subgraph. This subgraph is implicitly given by each edge (u, v) with $d[u] + len(u, v) = d(v)$. We implemented the algorithm doubly. One time with explicitly storing the subgraph (RR DAG) and one time with reconstructing it when needed (RR). It turned out that there are only small differences between both implementations, with no variant being clearly superior. We therefore only report the results for the space-saving implementation RR.

5.5 Single-Edge Update Experiments

We start our experimental study by single edge updates. Because of space restrictions and a different focus of our paper we do not carry out a separate analysis for the decremental and the incremental case. An update consists of choosing an edge uniformly at random and multiplying its weight by a random value in $(0, 2)$. The results can be seen in Table 1 , extended tables in Table 5 and corresponding experiments for edge insertions and deletions in Table 6.

We observe that the algorithms of the NARVÁEZ-framework have only tiny differences in performance with Nar-1st BF being slightly (but not significantly) faster most times (see Fig. 5). There is no such uniform behavior for the SWSF-FP-like algorithms. TUNED SWSF is always faster (between 1.3 and 6 times) than SWSF-FP. The algorithm TUNED SWSF RR is always at least as fast as SWSF-FP and up to 5.5 times faster. The algorithm TUNED SWSF NAR seems to be very volatile being between half as fast and 4 times faster than SWSF-FP.

Comparing the different classes of algorithms, we find the algorithms to perform quite differently, but within the same order of magnitude. The algorithm FMN is most times much slower than the other ones. This is due to the overhead caused by maintaining and reading the priority queues used by this algorithm. The technique used in this algorithm can pay off in case nodes with high degree exists (for which many edge-relaxations can be saved). This is not the case for the test instances used. Exceptions are the INTERNET instances CAIDA, AS-HOP and AS-RAN. Here, the gap to the other algorithms is much smaller, (which meets the theoretical considerations). Hence, it is to be expected that there are dense graph classes for which FMN is the superior algorithm. On the ROAD and GRID instances, the NARVÁEZ-framework is superior. This is because the structure of the shortest-path tree stored by the algorithm hardly changes on these experiments. Therefore, the early-propagation of the weight change works well. On the INTERNET instances, RR is the fastest algorithm. Looking at the small value of $\Delta(G, U)$, we can see that updates hardly have any impact on these instances, which favors the RR-algorithm with its small computational overhead and the early detection of edge weight increases that do not change distances on the graph.

The achieved speed-ups vary greatly between the instances. This is mainly due to the different structure of the underlying graphs, which results in greatly differing expected speed-ups. It is interesting to see that in nearly all cases the best actual speed-ups are close to the expected speed-ups or even higher. This, in combination with the small absolute runtimes in the range of microseconds, makes us expect that there is not much space for further improvement for the single-edge update case.

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GR100	GR300	UNIT H	UNIT E
FMN	42	1504	29087	151	22702	1624	2182	25	142	327	36
SWSF-FP	112	3759	65404	366	12429	416	691	59	351	1613	31
tun SWSF-FP	152	**5140**	84873	**562**	16406	893	**3442**	105	598	**2436**	**186**
tun SWSF-NAR	147	3354	70245	215	9306	614	695	94	523	748	129
tun SWSF-RR	118	3798	66068	412	26093	2148	**3766**	74	430	2096	102
RR	155	**4666**	74857	**510**	**34586**	**2599**	**4057**	103	568	**2519**	137
Nar-1st BF	**284**	**5335**	**100944**	357	6578	417	305	**138**	**784**	1176	20
$\Delta(G, U)$	130.42	140.52	70.72	30.68	0.21	0.41	0.74	59	113	0.01	93
expected speed-up	236	6762	62549	986	inf	inf	inf	169	804	inf	163

Table 1. Speed-ups of experiments with single-edge updates

5.6 Experiments on Batch Updates

Multiple Randomly Chosen Edges. In this experiment we chose 25 edges uniformly at random. For each edge, we chose uniformly at random a value from the interval $(0, 2)$ and multiplied the weight of the edge with that value. The results can be seen in Table 7. For each graph there is hardly any difference between $\Delta(G, U)$ and $\delta(G, U)$. Therefore, the single-edge updates did only interfere marginally with each other. Hence, not much news is to be expected by this setting regarding the comparison of the algorithms. This has been confirmed by the experiments.

However, we ran the batch-algorithms (NARVÁEZ and TUNED SWSF) twice. One time with processing the edges in batch as stated in the description and one time with iteratively processing the edges one after another. Nearly no runtime differences were observed between the iterative and the batch variants, which indicates a low overhead with batch updates.

Node Failure and Recovery. This update class uses the two parameters deg_{min} and deg_{max}. First, a node v with degree between deg_{min} and deg_{max} is chosen uniformly at random. The update consists of two steps. In the first step, v fails, i.e. the weights of all edges adjacent to v are set to infinity. In the second step, v recovers, i.e. the weights of all edges adjacent to v are reset to their original values. The results can be found in Tables 2 and 3, extended data in Tables 8 and 9.

degree	AS-HOP			AS-RAN			CAIDA		
	1-10	10-100	100-500	1-10	10-100	100-500	1-10	10-100	100-500
FMN	784	173	23	1368	129	3	7824	2284	185
ittun SWSF-FP	912	228	26	1320	235	11	**12874**	4212	382
SWSF-FP	273	68	8	389	28	1	9651	2203	128
tun SWSF-FP	967	250	24	1417	252	15	**14042**	4693	405
tun SWSF-NAR	407	92	9	410	50	4	9785	2187	122
tun SWSF-RR	**1272**	**528**	**130**	**2475**	**433**	**21**	12395	**6839**	**969**
RR	**1438**	**576**	**142**	**2623**	**490**	**17**	**13915**	**7075**	**1163**
Nar-1st Heap	53	21	9	86	59	16	4315	761	75
itNar-1st Heap	52	18	6	71	30	8	4060	573	63
$\delta(G,U)$	1.26	12.16	82.54	1.47	45.01	1365.85	1.97	7.4	90.99
$\Delta(G,U)$	1.07	8.59	71.28	1.1	34.6	712.3	1.45	5.7	85.73
expected speedup	27909	3489	393	27909	821	86	190914	38183	2246

Table 2. Speed-ups of experiments with node failure and recovery updates on INTERNET-instances

We now take a look at the INTERNET instances. The most remarkable result is the bad performance of the Narvaez-framework, which clearly is the inferior algorithm for that testset. One main reason for that is, that on this testset the edge-weight propagation in the initialization phase creates useless extra effort which gets overwritten later on. The gap between $\delta(G,U)$ and $\Delta(G,U)$ is small to mid-size, favoring RR with its small overhead, but big enough such that TUNED SWSF RR is nearly as fast. This difference also manifests in the small difference between ITTUNED SWSF and TUNED SWSF.

The situation is similar, but a bit clearer, for UNIT DISK graphs. When applying hop distance, $\delta(G,U)$ and $\Delta(G,U)$ are still quite near to each other, TUNED SWSF and RR are the best-performing algorithms (with RR being slightly better). When applying Euclidean or energy edge weights updates, the difference between $\delta(G,U)$ and $\Delta(G,U)$ is much bigger, and TUNED SWSF clearly is the superior algorithm. We also observe the advantage of TUNED SWSF against SWSF-FP being between 2 and 15 times faster.

Traffic Jams. This update class models real-world traffic jams. It derives from the observation that traffic jams often occur along shortest paths. The number k of updated edges is given as a parameter. Initially, a node v is chosen uniformly at random. Then a shortest path SP ending at v and containing exactly k edges is chosen uniformly at random. The update consists of two steps: in the first step, the weights of edges in SP are multiplied by 10. In the second step, the edge weights are reset to their original values. The results can be found in Tables 4 and 10.

We observe that this update class consists of strongly interfering single-edge updates: there is a big difference between $\delta(G,U)$ and $\Delta(G,U)$. TUNED SWSF and TUNED SWSF NAR are the best-performing algorithms for this testset. This is because pure batch algorithms avoid processing nodes many times. With an increasing number of edges, the interference between the updated edges increases and the advantage of these two algorithms grows.

metric	hop			euclidean			energy		
average degree	7	10	15	7	10	15	7	10	15
FMN	30	40	55	27	21	24	12	14	20
ittun SWSF-FP	238	398	485	116	95	98	56	66	91
SWSF-FP	128	214	236	60	32	36	28	24	22
tun SWSF-FP	**260**	**462**	**561**	**158**	**115**	**141**	**75**	**86**	**110**
tun SWSF-NAR	106	116	147	101	77	97	57	61	67
tun SWSF-RR	223	395	527	105	75	89	49	54	67
RR	**289**	**504**	**628**	111	91	106	55	63	84
Nar-1st Heap	70	87	131	84	62	111	52	62	74
itNar-1st Heap	55	71	100	64	50	66	36	46	52
$\delta(G,U)$	19	8	6	86	107	99	194	174	132
$\Delta(G,U)$	18	7	5	54	79	55	128	119	98
expected speedup	833	2500	3750	283	190	273	117	126	153

Table 3. Speed-ups of experiments with node failure and recovery updates on UNIT DISK-instances

	GRID			LUX			NLD			DEU		
edges	10	20	30	5	10	20	10	20	30	10	20	30
FMN	3	2	1	4	2	1	11	5	2	185	30	7
ittun SWSF-FP	15	9	5	15	7	2	39	17	6	755	100	23
SWSF-FP	13	10	6	15	9	5	75	32	12	873	173	40
tun SWSF-FP	**23**	**16**	**9**	20	**12**	6	**107**	**44**	**17**	**1210**	235	55
tun SWSF-NAR	**22**	**16**	**9**	22	**13**	**7**	**107**	41	**17**	**1402**	**342**	**79**
tun SWSF-RR	16	12	7	15	9	5	72	31	12	957	181	42
RR	17	10	5	20	9	3	43	18	6	924	149	36
Nar-1st Heap	16	9	5	20	10	4	37	15	5	1120	196	35
itNar-1st Heap	19	12	6	**24**	**12**	4	57	24	8	**1231**	219	54
$\delta(G,U)$	4367	7909	15552	1178	3052	8616	12910	32088	93725	7885	39260	142191
$\Delta(G,U)$	2591	3564	6412	821	1366	2567	4134	10899	26153	3884	13701	51252
expected speed-up	35	25	14	37	22	12	229	87	36	1127	320	85

Table 4. Speed-ups of experiments with traffic jam updates

For a small number of edges in the jam, the NARVÁEZ-framework is comparable to TUNED SWSF. The framework slows down with a growing number of updated edges. This is because the initialization phase processes many nodes one time for each updated edge. It is astonishing to see that the NARVÁEZ-framework is not able to take advantage of the batch-character of the update. This can be seen through a comparison with ITNARVAEZ. The iterative variant is even faster than the batch one, which could be a hint at space for improvement. Again, FMN is much slower than the other algorithms, as its overhead does not pay of on these instances.

6 Conclusion

In this work we focused on the single-source shortest-path problem with non-negative weights. We gave the first experimental study evaluating the performance for single-edge updates that contains all current algorithms and incorporates a broad set of instance classes. It turned out that the algorithms perform quite differently, but within the same order of magnitude. For road networks and grid graphs, the NARVÁEZ-framework performed best while RR was superior for internet-instances. The TUNED SWSF-algorithm was up to 6 times faster than its base algorithm SWSF-FP. Together with RR it was the best approach for the railway graph and unit disk graphs. Due to its overhead on the graphs used, FMN was always the slowest algorithm. Furthermore, the achieved speed-ups varied greatly between different instances. This can be explained by measuring the impact of the updates on the graphs. These measurements also propose that there is not much space for further improvements when applying the instances used in our study.

Moreover, we presented the first experimental study at all for the case of multiple edge changes at a time. One experiment was to choose a set of edges uniformly at random. It turned out that this way the single-edge updates did almost not interfere. Therefore, the results deviated not much from the single-edge case. Interestingly, nearly no runtime differences could be observed between the iterative and the batch variants of TUNED SWSF and the NARVÁEZ-framework, indicating a low overhead for batch updates. We also used two more realistic types of batch updates. One is the simulation of node failure and recovery, which affects all incident edges. The single-edge updates interfered for that class, but not very strongly. For internet instances, the best performing algorithms were RR and TUNED SWSF RR with RR being slightly faster. For UNIT DISK graphs, TUNED SWSF was the best algorithm with RR being slightly faster for hop distance. The other update class modelled traffic jams. The single-edge updates interfered greatly, TUNED SWSF and TUNED SWSF NAR were the superior algorithms there.

Furthermore, we presented tuned variants for the SWSF-FP-algorithm and evaluated their performance. TUNED SWSF requires only simple changes, but yields a great improvement in runtime. TUNED SWSF was never slower than its base algorithm and up to 15 times faster. The algorithm performed very well (and often best) with updates for which $\delta(G,U) - \Delta(G,U)$ was large. The combination of TUNED SWSF and ideas of the NARVÁEZ-framework was slightly faster than TUNED SWSF for traffic jams, but slower on all other datasets. The enhancement of TUNED SWSF with the technique of RR was faster than TUNED SWSF on the INTERNET-instances (were it was slightly slower than the best performing RR) and slower otherwise.

Finally, we gave a simple methodology (based only on Dijkstra's algorithm) to decide if one should try a single-edge or a batch-update algorithm for a given instance class. We compared the 'impact' of the update when processed in batch with the 'impact' when processed iteratively. For updates with a big gap between both values, the algorithms TUNED SWSF or TUNED SWSF RR usually performed best. With a small gap, there was usually a better-performing iterative algorithm.

Concluding, we gave a first experimental overview on the different approaches for the problem, which can be used as a base for further research. The most important information that can be extracted from our experiments is the astonishing level of data dependency within the problem. It turned out that a proper assessment of an algorithm's running time is not possible without full knowledge of the application it is used in. Further, a great amount of experiments is required to get the big picture of an algorithm's efficiency.

References

1. Narváez, P., Siu, K.Y., Tzeng, H.Y.: New Dynamic Algorithms for Shortest Path Tree Computation. IEEE/ACM Transactions on Networking **8** (2000) 734–746
2. Bruera, F., Cicerone, S., D'Angelo, G., Stefano, G.D., Frigioni, D.: Dynamic Multi-level Overlay Graphs for Shortest Paths. Mathematics in Computer Science (2008) To appear.
3. Delling, D., Wagner, D.: Landmark-Based Routing in Dynamic Graphs. In Demetrescu, C., ed.: Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07). Volume 4525 of Lecture Notes in Computer Science., Springer (2007) 52–65
4. Wagner, D., Watternhofer, R., eds.: Algorithms for Sensor and Ad Hoc Networks. Volume 4621 of Lecture Notes in Computer Science. Springer (2007)
5. Reps, T., Ramalingam, G.: An Incremental Algorithm for a Generalization of the Shortest-Path Problem. Journal of Algorithms **21** (1996)
6. Ramalingam, G., Reps, T.: On the computational complexity of dynamic graph problems. Theoretical Computer Science **158** (1996)
7. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully Dynamic Algorithms for Maintaining Shortest Paths trees. Journal of Algorithms **34** (2000)
8. Buriol, L., Resende, M., Thorup, M.: Speeding Up Dynamic Shortest-Path Algorithms. Informs Journal on Computing **20** (2008)
9. King, V., Thorup, M.: A space saving trick for directed dynamic transitive closure and shortest path algorithms. In: Proceedings of the 7th Annual International Conference on Computing Combinatorics (COCOON'01). Volume 2108 of Lecture Notes in Computer Science., Springer (2001) 268–277
10. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully dynamic shortest paths in digraphs with arbitrary arc weights. Journal of Algorithms **49** (2003) 86–113
11. Demetrescu, C.: Fully Dynamic Algorithms for Path Problems on Directed Graphs. PhD thesis, Department of Computer and Systems Science (2001)
12. Demetrescu, C., Italiano, G.F.: Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. Journal of Discrete Algorithms **4** (2006)
13. Frigioni, D., Ioffreda, M., Nanni, U., Pasqualone, G.: Experimental Analysis of Dynamic Algorithms for the single Source Shortest Path Problem. ACM Journal of Experimental Algorithmics **3** (1998)
14. Taoka, S., Takafuji, D., Iguchi, T., Watanabe, T.: Performance Comparison of Algorithms for the Dynamic Shortest Path Problem. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **E90-A** (2007)
15. Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest paths algorithms. Mathematical Programming, Series A **73** (1996) 129–174
16. HaCon - Ingenieurgesellschaft mbH: `http://www.hacon.de` (2008)
17. University of Oregon Routeviews Project: `http://www.routeviews.org/` (2008)
18. CAIDA: The Cooperative Association for Internet Data Analysis: `http://www.caida.org/` (2008)
19. PTV AG - Planung Transport Verkehr: `http://www.ptv.de` (2008)
20. Bauer, R., Delling, D.: SHARC: Fast and Robust Unidirectional Routing. In Munro, I., Wagner, D., eds.: Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08), SIAM (2008) 13–26
21. Bauer, R., Delling, D., Wagner, D.: Experimental Study on Speed-Up Techniques for Timetable Information Systems. In Liebchen, C., Ahuja, R.K., Mesa, J.A., eds.: Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07), Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2007) 209–225
22. Delling, D.: Time-Dependent SHARC-Routing. In: Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08). Volume 5193 of Lecture Notes in Computer Science., Springer (2008) 332–343 Best Student Paper Award - ESA Track B.

A Review on Complexity Results

In this section we report complexity results on the algorithms in Section 3 which are taken out of the original works.

FMN. The algorithm FMN has worst-case runtime $O(k \log n)$ if we consider only weight updates of edges. In case we consider a sequence of updates including insertions and deletions, then each output update requires $O(k \log n)$ amortized time when the graph consisting of all edges that occur during the updates has a k-bounded accounting function.

SWSF-FP. The algorithm SWSF-FP runs in $O(\| \delta \| \cdot (\log \| \delta \| + M_\delta))$ where we use the following notation: A vertex is said to be *modified* if it is not the source and if it is the target node of an updated edge. A vertex is said to be *affected* if its distance changes. A node is said to be *changed* if it is modified or affected. With $|\delta|$ we denote the number of changed nodes. With $\| \delta \|$ we denote the number of changed nodes plus the number of all edges adjacent to a changed node. Finally, M_δ denotes the time required to solve Belman Ford's Equations for a changed node.

RR. The algorithm RR processes a single edge update in time $O(\| \delta \| + |\delta| \log |\delta|)$ where we use the same notation as for the algorithm SWSF-FP.

NARVÁEZ. The algorithms of the Narvaez Framework have the following runtimes:

Queue Type	First Variant	Second Variant
FIFO	$O(D_{max} \cdot \delta_d^2)$	$O(D_{max} \cdot \delta_d^3)$
D'Esopo Pape	no polynomial upper bound	no polynomial upper bound
Priority Queue: Linear List	$O(\delta_d^2 + D_{max} \cdot \delta_d)$	$O(\delta_{pd} \delta_d + \gamma \cdot D_{max} \cdot \delta_d)$
Priority Queue: Binary Heap	$O(D_{max} \cdot \delta_d \cdot \log \delta_d)$	$O(\gamma \cdot D_{max} \cdot \delta_d \cdot \log \delta_d)$
Priority Queue: Fibonacci Heap	$O(\delta_d \cdot \log \delta_d + D_{max} \cdot \delta_d)$	$O(\delta_d \cdot \log \delta_d + \gamma \cdot D_{max} \cdot \delta_d)$

Where the D'Esopo Pape Queue is a linked list. Nodes that already have been in the queue are inserted at the front of the list and the other nodes are inserted at the back. Nodes that are extracted are always extracted from the front.

Symbols mean the following: δ_d is the minimum number of nodes that must change their distance or parent attributes or both, δ_{pd} is the minimum number of nodes that must change their distance and parent attributes. D_{max} denotes the maximum node degree. Finally, γ denotes the redundancy factor, which represents the average time that each node is visited by the algorithm.

B   Pseudocode of TUNED SWSF

Algorithm 1: TUNED SWSF

---

   input : graph $G = (V, E, len)$

             distance vector $D[]=d[]$

             source s

             set of updated edges U

             new length function $len_{new} : U \rightarrow \mathbb{R}^+$

   output: graph $G = (V, E, len)$

             distance vector $D[]=d[]$

1 never change $D[s]$ and $d[s]$

  /* Initialization   */

2 **for** $(u, v) \in U$ **do**

3   **if** $len(u, v) < len_{new}(u, v)$ **then**

4 $len(u, v) = len_{new}(u, v)$

5   **if** $d[v] > D[u] + len(u, v)$ **then** $d[v] = D[u] + len(u, v)$

6   **if** $len(u, v) > len_{new}(u, v)$ **then**

7 $len(u, v) = len_{new}(u, v)$

8 $d[v] := con(v)$

9   **if** $D[v] \neq d[v]$ **then**

10   insert v with priority $\min\{D[w], d[w]\}$ into Q or update the priority

  /* Main Phase   */

11 **while** *there are nodes in Q* **do**

12 $v :=$ extract and delete minimum from Q

13   **if** $d[v] < D[v]$ **then**

14 $D[v] := d[v]$

15   **for** $(v, w) \in E$ **do**

16   **if** $D[v] + len(v, w) < d[w]$ **then**

17 $d[w] := D[v] + len(v, w)$

18   insert w with priority $\min\{D[w], d[w]\}$ into Q or update the priority

19   **if** $d[v] > D[v]$ **then**

20 $D_{old} = D[v]$

21 $D[v] := \infty$

22 $d[v] := con(v)$

23   insert v with priority $d[v]$ into Q

24   **for** $(v, w) \in E$ with $D_{old} + len(v, w) = d[w]$ **do**

25 $d[w] := con(w)$

26   insert w with priority $\min\{D[w], d[w]\}$ into Q or update the priority

---

C Extended Tables of the Experiments

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GRID 100	GRID 300	UNIT HOP	UNIT EUCL
FMN	42	1504	29087	151	22702	1624	2182	25	142	327	36
SWSF-FP	112	3759	65404	366	12429	416	691	59	351	1613	31
tun SWSF-FP	152	5140	84873	562	16406	893	3442	105	598	2436	186
tun SWSF-NAR	147	3354	70245	215	9306	614	695	94	523	748	129
tun SWSF-RR	118	3798	66068	412	26093	2148	3766	74	430	2096	102
RR	155	4666	74857	510	34586	2599	4057	103	568	2519	137
RR dag	149	4340	71293	498	36801	2752	3882	95	534	2801	116
Nar-1st Heap	250	5192	97532	533	6438	410	304	146	821	1117	153
Nar-2nd Heap	274	5189	97035	554	6385	411	303	155	857	1063	110
Nar-1st BF	284	5335	100944	357	6578	417	305	138	784	1176	20
Nar-2nd BF	275	4636	99886	467	6494	411	304	156	871	1061	57
$\Delta(G,U)$	130.42	140.52	70.72	30.68	0.21	0.41	0.74	59	113	0.01	93
expected speed-up	236	6762	62549	986	inf	inf	inf	169	804	inf	163

Table 5. Speed-ups of single edge updates - extended table

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GRID 100	GRID 300	UNIT HOP	UNIT EUCL
FMN	62	1884	690	339	21427	1444	3592	50	304	257	681
SWSF-FP	142	4855	1490	742	12218	347	989	99	737	439	543
tun SWSF-FP	190	6202	1986	1049	15806	767	2151	165	1185	873	1676
tun SWSF-NAR	154	3128	2326	369	13137	462	769	104	618	428	525
tun SWSF-RR	143	5066	1495	763	27671	1798	4201	120	881	1043	1147
RR	322	6172	4619	1139	34537	2427	5046	221	1336	1457	1714
RR dag	307	6250	4479	1078	30886	2237	4680	205	1254	1399	1627
Nar-1st Heap	263	6349	3930	748	9373	274	782	202	1141	897	1234
Nar-2nd Heap	189	5106	2686	642	9183	275	766	143	877	759	1101
Nar-1st BF	55	2504	246	383	9770	280	800	23	105	587	560
Nar-2nd BF	26	1219	148	237	9805	282	792	10	51	394	331
$\Delta(G,U)$	52	59	977	7	0.23	0.22	0.12	19	35	2	2
expected speedup	589	16045	4486	4930	inf	inf	inf	556	2647	7500	15000

Table 6. Speed-ups of single edge updates, edge failure and recovery - Extended Table

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GRID 100	GRID 300	UNIT HOP	UNIT EUCL
FMN	2	12	184	7	1571	69	49	1	4	18	1
ittun SWSF-FP	6	56	784	35	1183	49	136	5	19	222	7
SWSF-FP	5	40	534	18	773	28	9	3	11	101	1
tun SWSF-FP	7	58	767	35	1207	49	139	6	19	236	7
tun SWSF-NAR	7	50	839	15	636	21	12	5	16	29	6
tun SWSF-RR	5	40	595	25	3913	409	183	4	14	400	4
RR	7	47	783	32	5501	592	236	6	18	618	5
RR dag	7	47	764	30	5556	627	220	5	17	658	4
Nar-1st Heap	12	73		31	738	9	33	9	25	71	6
Nar-2nd Heap	13	70		33	721	9	33	9	27	65	4
Nar-1st BF	12	63		19	753	9	33	8	17	73	1
Nar-2nd BF	13	73		29	729	9	33	9	27	64	2
itNar-1st Heap	12	72		30	728	9	33	8	25	70	6
itNar-2nd Heap	12	73		31	709	9	33	9	27	63	4
$\delta(G,U)$	3081	10833	5414	727	8	6	90	1420	4160	4	2612
$\Delta(G,U)$	2886	10468	5414	723	8	6	90	1304	4059	4	2340
expected speedup	11	90	809	41	23864	4652	310	8	22	5000	6

Table 7. Speed-ups of experiments with 25 edges chosen uniformly at random

degree	AS-HOP			AS-RAN			CAIDA		
	1-10	10-100	100-500	1-10	10-100	100-500	1-10	10-100	100-500
FMN	784	173	23	1368	129	3	7824	2284	185
ittun SWSF-FP	912	228	26	1320	235	11	12874	4212	382
SWSF-FP	273	68	8	389	28	1	9651	2203	128
tun SWSF-FP	967	250	24	1417	252	15	14042	4693	405
tun SWSF-NAR	407	92	9	410	50	4	9785	2187	122
tun SWSF-RR	1272	528	130	2475	433	21	12395	6839	969
RR	1438	576	142	2623	490	17	13915	7075	1163
RR dag	1377	550	116	2351	462	16	12723	6228	938
Nar-1st Heap	53	21	9	86	59	16	4315	761	75
Nar-2nd Heap	53	21	9	86	58	15	4294	751	73
Nar-1st BF	53	21	9	87	58	13	4386	762	74
Nar-2nd BF	53	21	9	86	53	9	4329	747	71
itNar-1st Heap	52	18	6	71	30	8	4060	573	63
itNar-2nd Heap	51	18	6	71	29	7	3976	564	60
$\delta(G,U)$	1.26	12.16	82.54	1.47	45.01	1365.85	1.97	7.4	90.99
$\Delta(G,U)$	1.07	8.59	71.28	1.1	34.6	712.3	1.45	5.7	85.73
expected speedup	27909	3489	393	27909	821	86	190914	38183	2246

Table 8. Speed-ups of experiments with node failure and recovery on INTERNET-instances - extended table

metric	hop			euclidean			energy		
average degree	7	10	15	7	10	15	7	10	15
FMN	30	40	55	27	21	24	12	14	20
ittun SWSF-FP	238	398	485	116	95	98	56	66	91
SWSF-FP	128	214	236	60	32	36	28	24	22
tun SWSF-FP	260	462	561	158	115	141	75	86	110
tun SWSF-NAR	106	116	147	101	77	97	57	61	67
tun SWSF-RR	223	395	527	105	75	89	49	54	67
RR	289	504	628	111	91	106	55	63	84
RR dag	255	422	561	102	81	94	52	58	76
Nar-1st Heap	70	87	131	84	62	111	52	62	74
Nar-2nd Heap	62	76	120	73	55	94	44	52	62
Nar-1st BF	49	63	109	7	2	8	2	2	3
Nar-2nd BF	32	44	87	5	1	4	1	1	2
itNar-1st Heap	55	71	100	64	50	66	36	46	52
itNar-2nd Heap	50	64	93	56	44	52	31	39	44
$\delta(G,U)$	19	8	6	86	107	99	194	174	132
$\Delta(G,U)$	18	7	5	54	79	55	128	119	98
expected speedup	833	2500	3750	283	190	273	117	126	153

Table 9. Speed-ups of experiments with node failure and recovery on UNIT DISK-instances - extended table

	GRID			LUX			NLD			DEU		
	10 edges	20 edges	30 edges	5 edges	10 edges	20 edges	10 edges	20 edges	30 edges	10 edges	20 edges	30 edges
FMN	3	2	1	4	2	1	11	5	2	185	30	7
ittun SWSF-FP	15	9	5	15	7	2	39	17	6	755	100	23
SWSF-FP	13	10	6	15	9	5	75	32	12	873	173	40
tun SWSF-FP	23	16	9	20	12	6	107	44	17	1210	235	55
tun SWSF-NAR	22	16	9	22	13	7	107	41	17	1402	342	79
tun SWSF-RR	16	12	7	15	9	5	72	31	12	957	181	42
RR	17	10	5	20	9	3	43	18	6	924	149	36
RR dag	16	9	5	19	8	3	42	17	6	859	143	34
Nar-1st Heap	16	9	5	20	10	4	37	15	5	1120	196	35
Nar-2nd Heap	16	9	5	22	11	4	37	14	5	1153	197	36
Nar-1st BF	2	1	1	10	7	2	12	7	2	524	71	11
Nar-2nd BF	13	7	4	20	10	5	35	14	5	1041	155	29
itNar-1st Heap	19	12	6	24	12	4	57	24	8	1231	219	54
itNar-2nd Heap	22	13	6	28	13	5	55	23	8	1421	249	67
$\delta(G,U)$	4367	7909	15552	1178	3052	8616	12910	32088	93725	7885	39260	142191
$\Delta(G,U)$	2591	3564	6412	821	1366	2567	4134	10899	26153	3884	13701	51252
expected speed-up	35	25	14	37	22	12	229	87	36	1127	320	85

Table 10. Speed-ups of experiments with traffic jam updates - extended table

	RAIL	LUX
FMN	271	15
ittun SWSF-FP	748	55
SWSF-FP	638	50
tun SWSF-FP	892	66
tun SWSF-NAR	535	77
tun SWSF-RR	556	51
RR	697	75
RR dag	638	72
Nar-1st Heap	312	74
Nar-2nd Heap	295	65
Nar-1st BF	302	22
Nar-2nd BF	261	12
itNar-1st Heap	263	69
itNar-2nd Heap	250	57
$\delta(G,U)$	13.31	281.6
$\Delta(G,U)$	9.5	227.3
expected speedup	3286	135

Table 11. Speed-ups of experiments with node-failure-and-recovery updates on additional graphs

graph	runtime
LUX	7.09
NLD	429.99
DEU	2414.28
RAIL	7.86
CAIDA	157.85
AS-HOP	8.80
AS-RAN	17.53
GRID 100	2.03
GRID 300	26.86
UNIT HOP	8.92
UNIT EUCL	9.13
UNIT DISC - HOP - deg 7	5.39
UNIT DISC - HOP - deg 10	6.61
UNIT DISC - HOP - deg 15	8.91
UNIT DISC - EUCL - deg 7	5.84
UNIT DISC - EUCL - deg 10	7.24
UNIT DISC - EUCL - deg 15	9.34
UNIT DISC - ENER - deg 7	6.12
UNIT DISC - ENER - deg 10	7.85
UNIT DISC - ENER - deg 15	10.35

Table 12. Absolute runtimes (ms) of a full run of Dijkstra's algorithms on the different instances