The Dynamic Single-Source Shortest-Path problem: A Survey

Arie Slobbe

May 8, 2015

1 Introduction

The shortest path problem occurs naturally in many applied problems, such as internet data routing and routing in road networks. With the rise of network science, another application of shortest-paths has been to find betweenness centralities of nodes in a graph. Whereas internet data routing and geographical routing are computationally tractable, computing betweenness centrality in a large graph is extremely computationally demanding, and it is in this application that efficiency becomes very important.

The fastest known algorithm for the static Single-Source Shortest-Path (SSSP) problem is an implementation of Dijkstra's shortest path finding algorithm using Fibonacci heaps for its priority queue. The original algorithm was published by computer scientist Edsger W. Dijkstra in 1956 [1] and the Fibonacci heap implementation is due to Tarjan and Fredman [2].

The dynamic SSSP problem is a variation of the SSSP problem in which a precomputed graph with the distances from the source node to the others is given. We consider situations in which the graph undergoes changes, given as a set of edge updates, and recompute the distances for the updated graph. Instances of the dynamic SSSP problem occur in road network routing when traffic jams and road work affect travel times, in data routing when routers fail or are compromised, and more generally in any network whose structure evolves over time.

Brauer and Wagner [3] ran an experimental study in which a range of existing dynamic SSSP algorithms were run on a broad set of test instances. First Incremental Dijkstra and Tuned SWSF are two such algorithms that appeared in their study. Both are characterized by their ability to handle batch updates. Pseudocodes of these algorithms, as well as an informal discussion, is provided in Section 2. In Section 3, we report on an implementation in Python of First Incremental Dijkstra and Tuned SWSF. In Section 4 we briefly report on their time complexity results, and in Section 5 discuss how the algorithms performed in Bauer and Wagner's experiments. We find that the performance of these algorithms is strongly dependent on the underlying data and that one algorithm does not always outperform the other.

2 Algorithms

2.1 Definitions

In our discussion of the algorithms, the introduction of some notation will prove invaluable. $Distance(x)$ denotes the distance of a node x to the source node, and $Parent(x)$ is the first neighbor of x through which x routes its shortest path to the source node. $Subtree(x)$ is the subset of nodes that includes x and all its descendants in the shortest-path tree. That is, all nodes in $Subtree(x)$ have in common that x occurs in their shortest path to the source. Corresponding to each subtree are two sets of edges, which we denote $I(subtree)$ and $O(subtree)$. $I(subtree)$ contains all edges whose target is in the subtree (all edges leading *into* the subtree). $O(subtree)$ contains all edges whose source is in the subtree (all edges leading *out* from the subtree). Q shall stand for a priority queue. In our implementations, we will repeatedly insert nodes into Q and extract them in order of "highest priority", which we express as a measure of shortest distance to the source node. Finally, $neighborhood(x)$ represents all nodes that can be reached from x in one hop.

2.2 First Incremental Dijkstra

First Incremental Dijkstra is an algorithm from the Narvaez framework of dynamic algorithms for shortest-path tree (SPT) computation [4], which was published in 2000 by Narvaez, Siu, and Tzeng. The framework consists of eight algorithms that are each a variation of a *basic* algorithm for computing an SPT. An SPT is stored as a parent attribute and a distance attribute for each node. The *basic* algorithm consists of 4 steps, which are outlined below. The pseudocode for the *basic* algorithm contains two ambiguities: In step 2, which element do we extract from the queue, and in step 3, how do we select a subset of descendants? These ambiguities can be resolved in various ways, and each combination of resolutions represents one algorithm in the framework. We will implement First incremental Dijkstra. That means that in step 2, we always extract the highest priority node from the queue. In step 3, we select no descendants and include only the node that is currently being processed. A pseudocode for First Incremental Dijkstra is provided in the figure *Algorithm 1.*

Step 1 Initialization. If an edge weight is increased, identify all nodes whose shortest path will be affected and enqueue the nodes that may find a different shortest path. If an edge weight is decreased (and this edge is in the SPT), then all its descendants decrease their distance to the source correspondingly. There may now be other nodes that can route their shortest path through the edge whose weight was decreased, and we enqueue all such nodes for further processing.

Step 2 Node Selection. If the queue is empty, terminate. Else, extract the highest priority node from the queue in style of the Dijkstra algorithm, and update the node's tree attributes (distance to source, and parent).

Step 3 Distance Update. Take the element that was extracted from the queue and update its distance.

```
input : graph $G = (V, E)$
        $Distance$
        $Parents$
        source node $s$
        list of updated edges $U$
output: updated $Distance$
        updated $G = (V, E)$
/* Initialization                                                  */
for $\forall(u, v) \in U$ do
    if weight has increased then
        $\Delta \leftarrow$ weight increase
        $weight(u, v) \leftarrow weight(u, v) + \Delta$
        if $(u, v) \in Subtree(s)$ then
            for $\forall n \in Subtree(v)$ do
                $Distance(n) \leftarrow Distance(n) + \Delta$
            for $\forall(x, y) \in I(Subtree(v))$ do
                if $Distance(y) > Distance(x) + weight(x, y)$ then
                    $newDistance(y) \leftarrow Distance(x) + weight(x, y)$
                    $newParent(y) \leftarrow x$
                    Insert $y$ into $Q$ with priority $newDistance(y)$
    if weight has decreased then
        $\Delta \leftarrow$ weight decrease
        $weight(u, v) \leftarrow weight(u, v) - \Delta$
        if $Distance(v) > Distance(u) + weight(u, v)$ then
            $\Delta\prime \leftarrow Distance(v) - (Distance(u) + weight(u, v))$
            $Parent(v) \leftarrow u$
            for $\forall n \in Subtree(v)$ do
                $Distance(n) \leftarrow Distance(n) - \Delta\prime$
            for $\forall(x, y) \in O(Subtree(v))$ do
                if $Distance(y) > Distance(x) + weight(x, y)$ then
                    $newDistance(y) \leftarrow Distance(x) + weight(x, y)$
                    $newParent(y) \leftarrow x$
                    Insert $y$ into $Q$ with priority $newDistance(y)$
/* Main Phase                                                      */
while there are nodes in $Q$ do
    $u \leftarrow$ Extract node with highest priority
    $\Delta \leftarrow newDistance(u) - Distance(u)$
    if $\Delta < 0$ then
        $Parent(u) \leftarrow newParent(u)$
        $Distance(u) \leftarrow newDistance(u)$
        for $\forall(u, v) \in neighborhood(u)$ do
            if $Distance(v) > Distance(u) + weight(u, v)$ then
                $newDistance(v) \leftarrow Distance(u) + weight(u, v)$
                $newParent(v) \leftarrow u$
                Insert $v$ into $Q$ with priority $newDistance(v)$
```

Algorithm 1: First Incremental Dijkstra

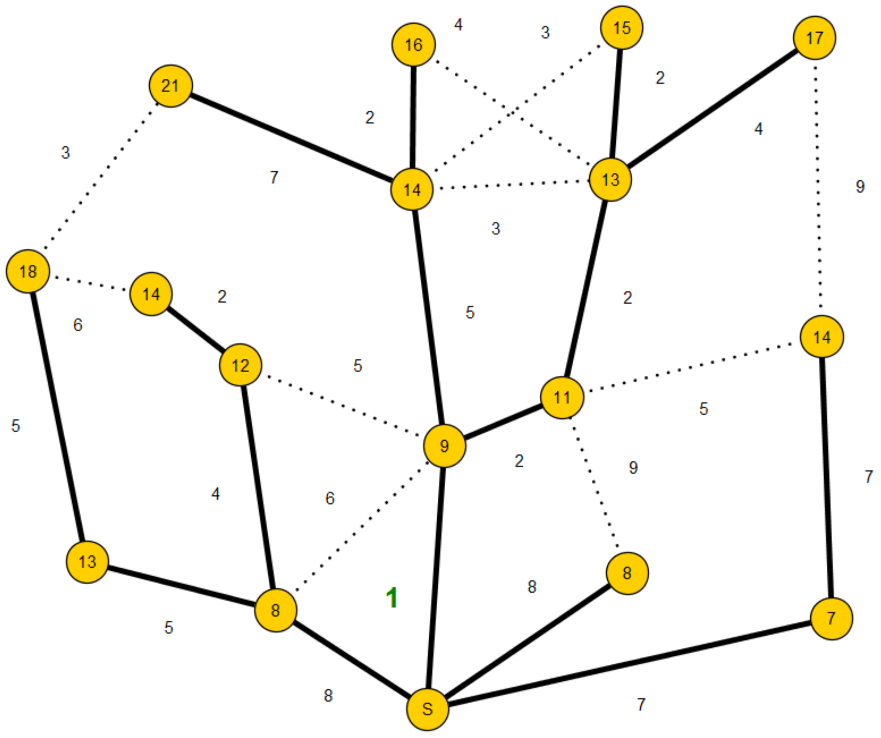

Figure 1: **First Incremental Dijkstra.** Each node is labeled with its distance to the source node S, and edges belonging to the shortest-path tree are given in bold. Suppose that the weight of edge (S, 9) is decreased from 9 to 1. Thus we have $\Delta = 8$.

Step 4 Node Search. For the element that was extracted from the queue, check if there are neighbors that could find a new shortest path by routing through that element. Enqueue these neighbors and return to step 2.

When several updates must be processed, the algorithm can be run once for each instance of an update. However, some optimization can be achieved by processing updates in batches. When a batch update is performed with First Incremental Dijkstra, a separate initialization must be done for each update. First we run the initialization phase for each update that consists of a weight increase, and then we process all weight increases jointly in a single main phase. Subsequently, we initialize all weight decreases individually. The final step of the batch version is to run the main phase for all weight decreases.

Fig. 1, fig. 2 and fig. 3 show how the algorithm is initialized after the weight of an edge has been decreased. A similar procedure is used when a weight is increased: Δ is propagated along the subtree of the affected node, and subsequently we enqueue all nodes for which we can find a shorter path.

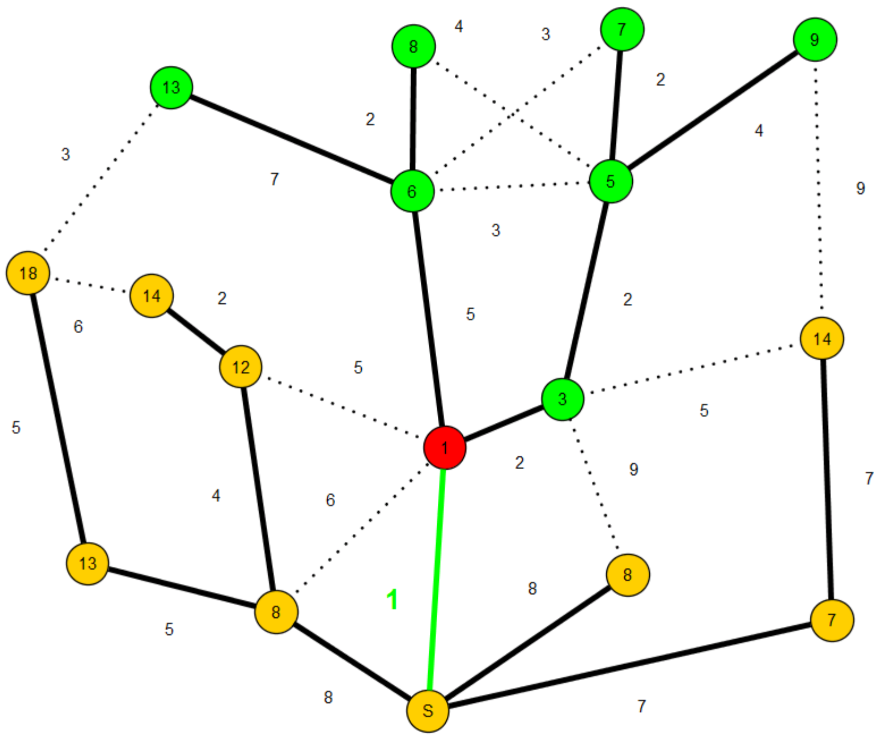

Figure 2: **First Incremental Dijkstra (cont.)** During the initialization phase, we decrease the distance of all nodes in the subtree of node 9 by Δ.

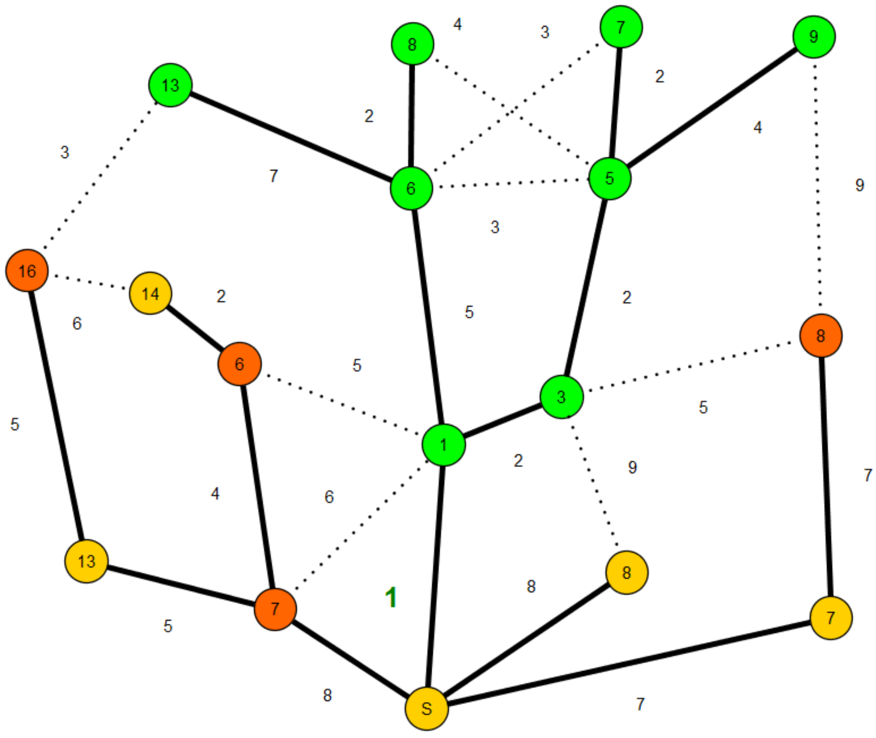

Figure 3: **First Incremental Dijkstra (cont.)** Subsequently, we loop over all edges emanating out from the subtree of node 9 and enqueue each node for which a better shortest path can be found. Then we enter the main phase, in which the shortest paths are recomputed in Dijkstra-like fashion.

2.3 Tuned SWSF

An algorithm under the name SWSF-FP (for Strict Weakly Superior Function - Fixed Point) was proposed by Ramalingam and Reps in 1996 [5]. Interestingly, the algorithm as introduced by Ramalingam and Reps was proposed as a solution to a problem in formal language theory that involves the minimum cost derivation of a string in a particular class of grammars. Their solution applies to cost functions that are strict weakly superior functions, hence the name SWSF. The graph algorithm comes out of this problem as a special case of the grammar algorithm when one non-terminal variable is introduced for each node and one production rule and a terminal symbol are introduced for each edge. Tuned SWSF is a more efficient implementation of SWSF-FP that was introduced in 2009 by R. Bauer and D. Wagner [3]. A pseudocode for Tuned SWSF is provided in the figure *Algorithm 2*.

As was the case for First Incremental Dijkstra, batches of updates are allowed, meaning that we can process many changes simultaneously. Yet unlike First Incremental Dijkstra, Tuned SWSF carries out a single procedure for batch updates that consist of edge weight increases as well as decreases. When Tuned SWSF runs weight increases and decreases through the same steps simultaneously, some complications are introduced. Tuned SWSF needs to do more bookkeeping than First Incremental Dijkstra in order to track which node should get processed first. In order to achieve a proper order for node processing, each node is given an extra piece of information, which is called a label.

The formulation of Tuned SWSF bears traces from its origins as a solution for a grammar problem. We will discuss the algorithm in the language in which it was conceived. First, we re-introduce the shortest-path problem as a problem of equations.

Let u be a vertex in the given graph G with source node s and let $d(u)$ be the length of the shortest path from s to u. The SSSP problem induces a collection of equations, called the *Bellman-Ford equations*, in the set of unknowns $\{d(u) \mid u \in V(G)\}$:

$$d(u) = \begin{cases} 0 & \text{if u=source(G)} \\ \min_{v \in Pred(u)}[d(v) + length(v \to u)] & \text{otherwise} \end{cases}$$

Here, $length(v \to u)$ is the weight corresponding to the edge $(v \to u)$, and the set $Pred(u)$ is the set of vertices that have an arc going to u. For a vertex u that is not reachable from s, we set $d(u) = +\infty$.

Let $d[u]$ be the distance to u that is maintained by the program, and let $rhs(u) = \min_{v \in Pred(u)}[d[v] + length(v \to u)]$. When the SSSP problem is solved, $d[u] = rhs(u)$ for all u in G. We say that all such vertices are **consistent**.

We say that u is **over-consistent** if $d[u] > rhs(u)$, and that u is **under-consistent** if $d[u] < rhs(u)$. Under-consistent vertices can arise, for instance, when some edge on some shortest path is deleted.

```
input  :  graph $G = (V, E)$
          $Distance$
          $Parents$
          source node $s$
          list of updated edges $U$
output:  updated $Distance$
          updated $G = (V, E)$
/* Initialization                                                  */
$Labels \leftarrow Distance$
for $\forall(u, v) \in U$ do
    if $weight\ has\ increased$ then
        update edge weight
        if $Label(v) > Distance(u) + weight(u, v)$ then
            $Label(v) \leftarrow Distance(u) + weight(u, v)$
    if $weight\ has\ decreased$ then
        update edge weight
        $Label(v) \leftarrow rhs(v)$
    if $Label(v) \neq Distance(v)$ then
        Insert v into Q with priority $\min(Distance(v), Label(v))$
/* Main Phase                                                      */
while $there\ are\ nodes\ in\ Q$ do
    $v \leftarrow$ Extract node with highest priority
    if $Label(v) < Distance(v)$ then
        $Distance(v) \leftarrow Label(v)$
        for $(v, w) \in E$ do
            if $Distance(v) + weight(v, w) < Distance(w)$ then
                $Distance(w) \leftarrow Distance(v) + weight(v, w)$
                Insert $w$ into $Q$ with priority
                $\min(Distance(w), label(w))$ or update the priority.
    if $Label(v) > Distance(v)$ then
        $D_{old}(v) \leftarrow Distance(v)$
        $Distance(v) \leftarrow \infty$
        $Label(v) \leftarrow rhs(v)$
        insert $v$ with priority $Label(v)$ into $Q$
        for $(v, w) \in E\ with\ D_{old}(v) + weight(v, w) == Label(w)$ do
            $Label(w) \leftarrow con(w)$
            Insert $w$ into $Q$ with priority $\min(Distance(w), Label(w))$
```

Algorithm 2: Tuned SWSF

Inconsistencies in the graph should be processed in increasing order of key, where the key of an inconsistent vertex u is defined as follows: $key(u) = min(d[u], rhs(u))$. In practice, the key of an over-consistent vertex u is $rhs(u)$, while the key of an under-consistent vertex u is $d[u]$.

In Dijkstra-like fashion, we process inconsistent vertices one by one, starting with the vertex u that has the lowest key. If u is over-consistent, and has the lowest key value, then its rhs value is (provably) its correct value. The reason for this is the same as the reason why the shortest-distance unprocessed node in the Dijkstra algorithm has the correct value: none of the other unprocessed nodes can possibly help it make a shorter path to the source. On the other hand, if u has the lowest key and is under-consistent, then $d[u]$ is outdated (possibly because an edge on the shortest path to u was removed). We then assign $d[u] = \infty$, thereby converting it into an over-consistent vertex, and re-insert it into the queue.

Fig. 4 shows how Tuned SWSF initializes a graph with a batch update. Fig. 5 and fig. 6 show how the main phase of the algorithm deals with a node whose shortest distance has become worse ($label > distance$), and a node whose shortest distance has improved ($label < distance$), respectively.

3 Data Structures and Implementation

We implemented, in Python, the algorithms First Incremental Dijkstra and Tuned SWSF, from the pseudocodes which were provided in their respective original papers. We ran the algorithms on the $USAirLines$ data set from the Pajek data sets webpage [6]. The graph is stored as a dictionary of dictionaries. For example, we access the weight w of a directed edge $u \to v$ with the statement $Graph[u][v]$. In undirected graphs we store each edge twice: once under $Graph[u][v]$, and once under $Graph[v][u]$.

We use the following auxilliary data structures. A distance attribute for each node is stored in a dictionary *distance*, which will hold all shortest path lengths. Another dictionary, *parents*, stores the parent attribute for each node. The parent attribute of a node u is the parent node of u in the shortest path tree rooted at the source. Each node except the source node has exactly one parent. The parent attribute of the source node is set to $None$.

Tuned SWSF and First Incremental Dijkstra each load additional auxilliary data structures which are unique to their respective implementations. Tuned SWSF assigns a label to each node and stores it in a dictionary of the same name. The label of a node represents the prospective shortest path length of a node as it is being processed, and its value may fluctuate during the execution of the algorithm before it settles on a final value, which is then consolidated in *distance*. Ramalingam and Reps suggest an implementation of SWSF with a binary heap for maintaining the priority queue. However, it was found during implementation that it is key for the proper functioning of the algorithm that the priority of a node u always be based on the most up-to-date values of $Distance(u)$ and $Label(u)$. Now, suppose that u is in the queue. As particularly

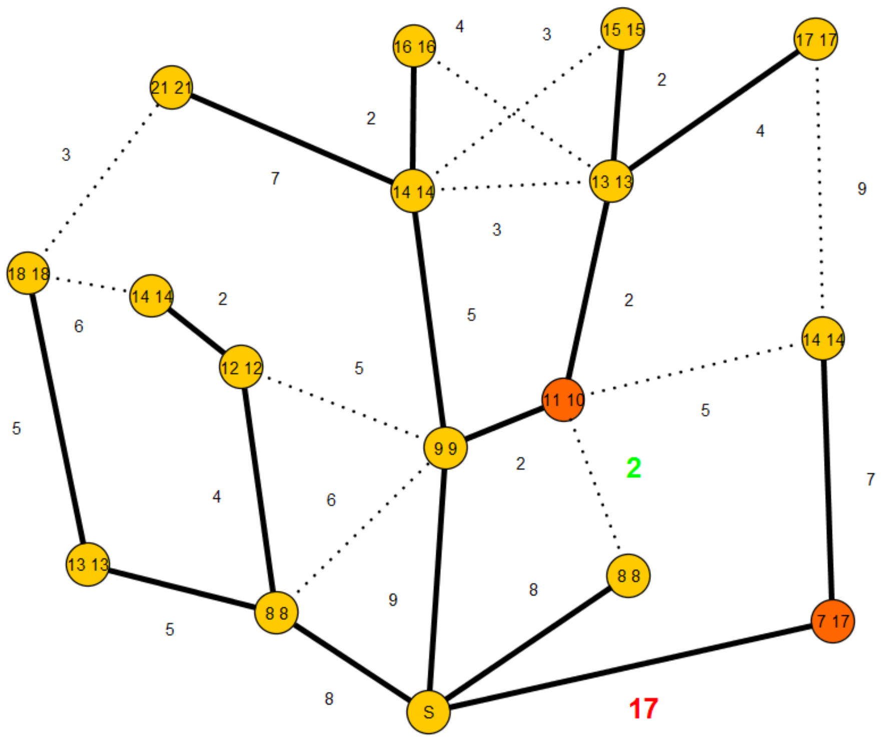

Figure 4: **Tuned SWSF** Suppose that the weight of edge (8 8, 11 11) is decreased from 9 to 2, and the weight of edge (S, 7 7) is increased from 7 to 17. At the end of the initialization phase, each node is assigned a label (which is printed on the right of its distance). The labels of all nodes but the ones in orange are initially equal to distance. The nodes in orange are endpoints of an updated edge, so we set their labels to the best shortest path distance that is currently available and enqueue them for further processing.

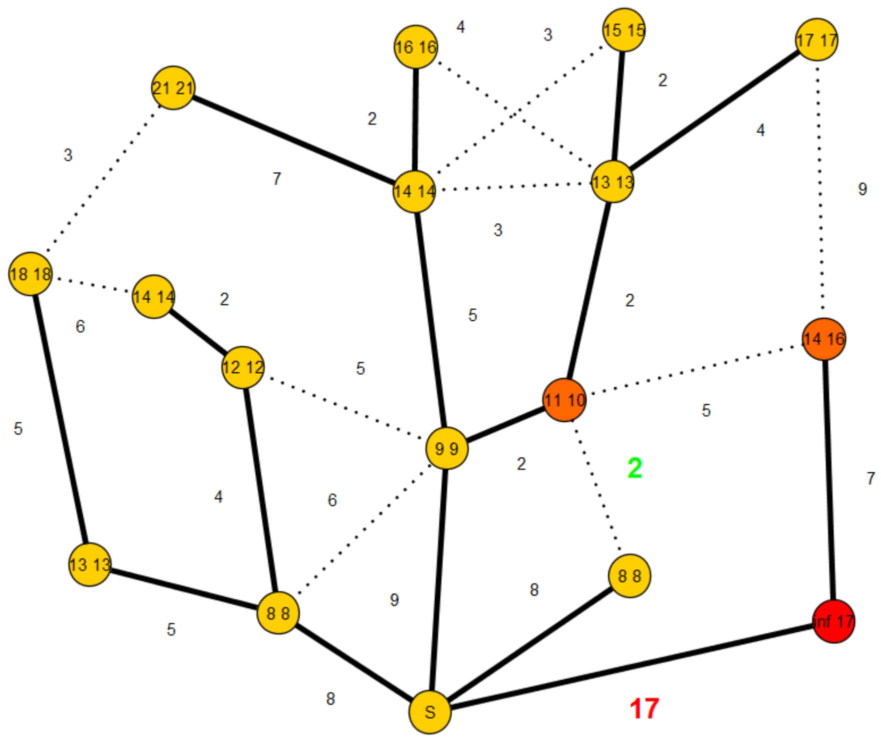

Figure 5: **Tuned SWSF (cont.)**From the enqueued nodes, we process in order of lowest overall value for either label or distance. If *label* > *distance*, we set distance to ∞ and reinsert the node into the queue. As always, we enqueue the neighbors of the node that is currently being processed for which a better shortest path can be found.

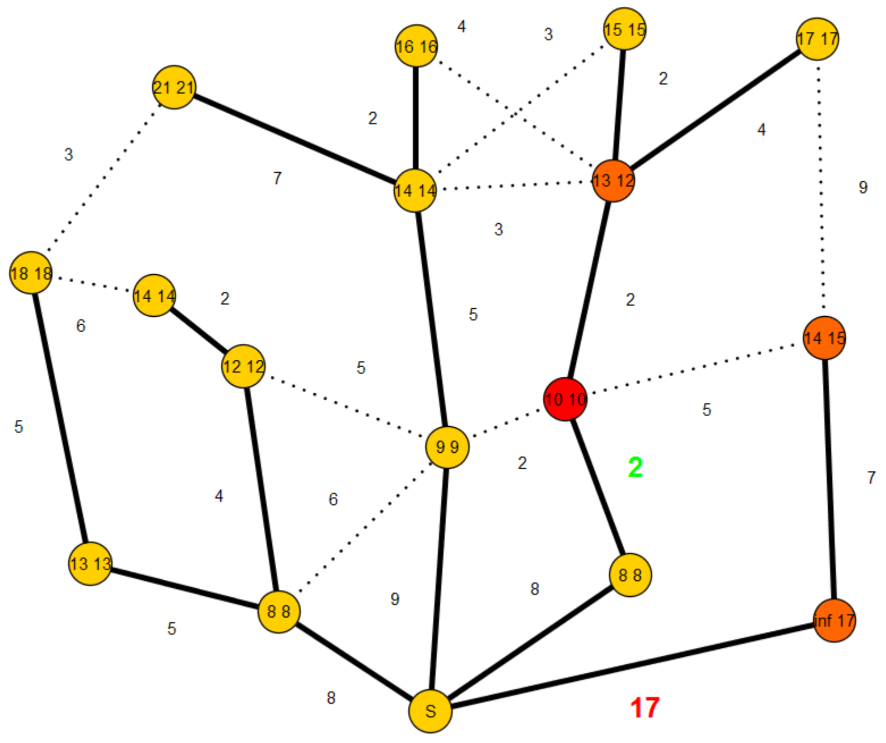

Figure 6: **Tuned SWSF (cont.)** If *label* < *distance*, we have (provably) found the best possible distance for the node. We set distance equal to label and consider the node processed.

the value of of $Label(u)$ may change for u during a run of the main phase of the algorithm, we would have to update its priority in the queue and subsequently re-establish the heap property before the next instance of a node extraction. In order to avoid the associated computational overhead, the queue for Tuned SWSF was implemented as a list.

On the other hand, priorities of nodes in the queue for First Incremental Dijkstra remain fixed on the values with which they were inserted. Thus, the queue for First Incremental Dijkstra is implemented as a binary heap in order to benefit from the binary heap's superior asymptotic performance over linear data structures.

4 Complexity Analysis

The time complexities of the two algorithms are hard to compare to one another, because their authors measure time complexity in slightly different ways. First Incremental Dijkstra's worst-case bound involves maximum degree D_{max}, and δ, which is the minimum number of changes made to the shortest path tree. No complexity analysis was provided for Tuned SWSF, but some information about it can be inferred from the time complexity analysis of SWSF-FP, which was provided by Ramalingam and Reps in its original paper. SWSF-FP's time complexity involves a different δ, which represents the number of nodes changed plus the number of edges adjacent to nodes that were changed, and M, which denotes the time required to find the consistent value for a changed node. Tuned SWSF improves on SWSF-FP by reducing the number of times that the consistent value for a node must be computed. Therefore we take the time complexity of SWSF-FP as an upper bound on the time complexity of Tuned SWSF.

Time complexity results :
First Incremental Dijkstra: $O(D_{max} * \delta * \log(\delta))$
SWSF-FP: $O(\delta * (\log \delta + M))$

5 Algorithm Performance

Bauer and Wagner studied the performance of First Incremental Dijkstra and Tuned SWSF, and many other dynamic shortest-path finding algorithms. They were interested in how these algorithms would perform under various testing conditions. Among the situations that were simulated were single edge updates, batch updates, and node failures. Tests were run on various synthetic networks as well as on various real-world networks such as the internet router network and road networks. We select a subset of these experiments for discussion and refer to the original paper of Bauer and Wagner for a comprehensive review.

For each test, the static Dijkstra algorithm was run and its completion time used as a baseline for the performance of the other algorithms. The performance of Tuned SWSF and First Incremental Dijkstra is expressed in terms of speedup factor as opposed to static Dijkstra.

5.1 Data Sets

The CAIDA data set represents the internet on the router level. Nodes are routers and edges represent connections between routers. The network contains 190,914 vertices and 1,215,220 edges.

The NLD and LUX data sets consist of road networks of the Netherlands and Luxembourg, respectively. The NLD data set has 946,632 nodes and 12,358,226 edges, and LUX has 30,647 nodes and 75,576 edges.

GRID 100 is a fully synthetic graph based on a 100 × 100 two-dimensional square grid. Nodes of the graph correspond to crossings on the grid, and each node is linked only to its immediate neighbors. Edge weights are randomly chosen integer values between 1 and 1000.

5.2 Node Failure and Recovery

On the CAIDA network of internet routers, a scenario was simulated in which a router fails and all the weights of all its edges are set to infinity. Subsequently, the router recovers and all the edges are reinstated to their original values. The test thus consists of two runs for each algorithm (one for the edge failure, and one for the recovery). The results below apply to combined performance over node failure and subsequent recovery.

Node degree	1-10	10-100	100-500
Tuned SWSF	9651	2203	128
First Incremental Dijkstra	4315	761	75

For nodes with degree between 1 and 10, the dynamic algorithms are literally thousands of times faster than the static Dijkstra algorithm which recomputes the complete shortest path tree from scratch. When bigger, and likely more central routers were knocked out, the difference with Dijkstra diminished. In all likelihood, this is because hub-like nodes are tend to be found on many more shortest paths. For all node sizes, Tuned SWSF clearly outperforms First Incremental Dijkstra. One main reason for that is that the edge-weight propagation of First Incremental Dijkstra creates extra effort which may be overwritten later on. In such cases, First Incremental Dijkstra must visit affected nodes twice where Tuned SWSF must visit only once.

5.3 Single-Edge Update and Batch Update Experiments

In the following tables, we present the results of a single edge update experiment and a batch update experiment.

Single Edge Updates	LUX	NLD	GRID 100	CAIDA
Tuned SWSF	152	5140	105	16406
First Incremental Dijkstra	250	5192	146	6438

Batch Size 25	LUX	NLD	GRID 100	CAIDA
Tuned SWSF	7	58	6	1207
First Incremental Dijkstra	12	73	9	738

We note that Tuned SWSF consistently outperforms First Incremental Dijkstra on the CAIDA data sets. However, on the road networks and on GRID 100, First Incremental Dijkstra is superior. Bauer and Wagner suggest that this is because the structure of the shortest-path tree stored by the algorithms hardly changes in these experiments. Therefore, the early-propagation of the weight change works well for First Incremental Dijkstra.

Overall, neither of the algorithms seems be more particularly suited than the other for single edge updates. Despite Tuned SWSF's innovative approach to handling multiple edge updates at once, it does not do significantly better than First Incremental Dijkstra on the batch version of the SSSP problem.

6 Conclusion

In this work we focused on two approaches to the dynamic SSSP problem. Tuned SWSF and First Incremental Dijksttra have their strengths and their weaknesses. Experiments turned out to be required to get the big picture of the efficiency of these algorithms. Tuned SWSF performs very well on the CAIDA data set, and a question for further research would be whether Tuned SWSF is generally the superior choice for any network data set whose degree distribution obeys a power law. First Incremental Dijkstra performed very well on data sets in which the shortest-path tree is robust under edge modifications. However, it performed poorly on the CAIDA data set, where its edge weight propagation technique proved counter-productive.

7 Bibliography

References

[1] Dijkstra, E. V., *A note on two problems in connexion with graphs*, Numerische Mathematik 1 pp. 269-271 (1959)

[2] Fredman, M. L., Tarjan, R. E. *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the Association for Computing Machinery 34 (3): 596–615 (1987)

[3] Bauer, R., Wagner, D. *Batch dynamic single-source shortest-path algorithms: An experimental study* 8th International symposium on experimental algorithms (SEA '09), volume 5526 of LNCS, pp. 51-62 (2009)

[4] Narvaez, P., Siu, K.Y., Tzeng, H.Y., *New Dynamic Algorithms for Shortest Path Tree Computation*, IEEE/ACM Transactions on Networking 8 pp. 734-746 (2000)

[5] Reps, T., Ramalingam, G., *An Incremental Algorithm for a Generalization of the Shortest-Path Problem*, Journal of Algorithms 21 (1996)

[6] http://vlado.fmf.uni-lj.si/pub/networks/data/default.htm