Shortest Path Trees Computation in Dynamic Graphs

Edward P.F. Chan & Yaya Yang
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
epfchan@uwaterloo.ca
yaya_yang@hotmail.com
http://sdb.uwaterloo.ca

March 21, 2005

Abstract

Let $G = (V, E, w)$ be a simple digraph, in which all edge weights are non-negative real numbers. Let G' be obtained from G by the application of a set of edge weight updates to G. Let $s \in V$, and let T_s and T'_s be a *Shortest Path Tree* (SPT) rooted at s in G and G', respectively. The *Dynamic Shortest Path* (DSP) problem is to compute T'_s from T_s. For the DSP problem, we correct and extend a few existing SPT algorithms to handle multiple edge weight updates. We prove that these extended algorithms are correct. The complexity of these algorithms is also analyzed. To evaluate the proposed algorithms, we compare them with the well-known static *Dijkstra* algorithm. Extensive experiments are conducted with both real-life and artificial data sets. The real-life data are road system graphs obtained from the Connecticut road system and are relatively sparse. The artificial data are randomly generated graphs and are relatively dense. The experimental results suggest the most appropriate algorithms to be used under different circumstances.

Keywords: Dynamic Shortest Path, Shortest Paths, Shortest Path Trees, Dynamic Graphs, Incremental Algorithms, Fully- and Semi-Dynamic Algorithms.

1 Introduction

Consider an application in which there are a number of distribution centers that are scattered around a metropolitan area, and it is useful to know the least-cost traffic routes from each location to all major intersections. Taking intersections as vertices, blocks between two intersections as edges, and traffic latencies as edge weights, the city traffic map is a digraph with non-negative edge weights. The least-cost route query between two intersections is to find a shortest path between two vertices in the corresponding graph. Since the traffic condition changes rapidly, least-cost routes may not be correct a few minutes after they are computed. One could apply *Dijkstra*'s algorithm [15] repeatedly to compute the shortest paths. However, this well-studied static algorithm may become ineffective when only a small number of the city roads experience latency changes. Therefore, researchers have been studying incremental algorithms to minimize shortest paths re-computation time.

Computing shortest paths efficiently in a dynamic graph environment also finds its application in a spatial database system. In such a system, it is essential to provide the functionality of finding an optimal route in a network. A graph in a route query system in general is of an arbitrary size and is too huge to be main-memory resident. In the past decade, a popular approach to solve the scalability problem is based on graph partitioning [10, 4, 21, 23, 33]. The whole graph is first partitioned into smaller sized *fragments*, each of which can fit into the main-memory. Because the size of a graph could be arbitrarily large, to speed up the search process and to minimize the I/O activity, a common technique is to materialize, in each fragment, the (local) shortest-distance information between the so-called *border vertices* (those shared by more than one fragment). In a real-time traffic information system, an edge weight in a fragment could be updated dynamically, the shortest-distance information between border vertices has to be re-computed fast for it to be useful in a route query evaluation. This can be accomplished by materializing, for each border vertex, a *Shortest Path Tree (SPT)* to all other border nodes in a fragment; and re-computing each SPT whenever some edge weights in the fragment have been changed.

We call the problem of re-computing SPTs in a dynamic environment the *DSP* problem. Let $G = (V, E, w)$ be a simple digraph, in which all edge weights are non-negative real numbers. Let $G' = (V, E, w')$ be obtained from G by the application of a set of edge weight updates (increases and/or decreases) to G. Let $s \in V$; let T_s and T'_s be SPTs rooted at s in G and G', respectively. The *DSP* problem is to compute T'_s from T_s.

For the *DSP* problem, the input edge weight changes could come in three forms: increases only, decreases only, and a mixture of both. We denote an algorithm as *semi-dynamic* if the input is either a set of edge weight increases or a set of edge weight decreases, but not both. An algorithm is said to be *fully-dynamic* if the input can be a set of mixed edge weight changes. We shall investigate the performance of both semi- and fully-dynamic algorithms in this work.

An intelligent approach to solve the *DSP* problem is proposed in [28]. We denote their semi-dynamic algorithms as *BallString* since they are based on a ball-and-string model. Unfortunately, the semi-dynamic algorithm for edge weight increases case is incorrect. We amend *BallString* by proposing *MBallStringInc* that updates SPTs correctly in the case of multiple edge weight increases. We propose a dynamic version of *Dijkstra*, which we call *DynDijkstra*. *DynDijkstra* are two semi-dynamic algorithms that can handle multiple edge weight increases and decreases, respectively. A fully-dynamic algorithm called *DynamicSWSF-FP* is proposed in [31]. However, a problem with *DynamicSWSF-FP* is that some of its computation is inefficient. We modify *DynamicSWSF-FP* by applying some optimizations on re-computing the so-called *rhs* values and adding SPT tree structure maintenance. We call the resulting more efficient algorithm *MFP*.

For each of the following proposed algorithms: *DynDijkstra*, *MBallStringInc*, and *MFP*, we prove its correctness and analyze its complexity. In addition, we derive general frameworks for describing *DynDijkstra* and *MBallString*.[1] Furthermore, we conduct extensive experiments, with both real-life as well as artificial data sets, on all proposed algorithms and compare them with the static algorithm *Dijkstra*. As a result, we identify the preferred algorithm to compute an SPT in a dynamic environment under different input mixes.

In Section 2, we define some basic notation. In Section 3, we survey related work and highlight our contributions. In Sections 4 and 5, we describe the proposed semi- and fully-dynamic algorithms. We give the correctness proofs of some of these algorithms in the *Appendix*. In Section 6, we analyze the complexity of the proposed algorithms. In Section 7, we present experimental results and analysis. Finally, we give our conclusion in Section 8.

2 Preliminary

2.1 Definition and Notation

Before proceeding to the description of algorithms, let us examine the definitions of frequently used terms. Terms not defined here are common concepts in graph theory (such as vertices, edges, paths, and trees), which can be found in any graph theory resource [8].

Let $G = (V, E, w)$ be a simple digraph with non-negative edge weights, where V is the set of vertices, $E = \{e | e = (u, v), u, v \in V\}$, and $w : E \to \Re^{\geq 0}$, *i.e.,* w is a function from the set of edges to non-negative real numbers. Sometimes we use $V(G)$ and $E(G)$ to denote the set of vertices and edges in G, respectively. Let $e = (u, v) \in E$; then u is the *tail* of e denoted as $t(e)$, and v is the *head* of e denoted as $h(e)$.

Let $u \in V$; the set of *outgoing edges* of u is defined as $Out_u = \{e | e \in E$ and $t(e) = u\}$, and the set of *incoming edges* of u is defined as $In_u = \{e | e \in E$ and $h(e) = u\}$. Correspondingly, the *children* of u are defined as $c(u) = \{v | v = h(e)$ and $e \in Out_u\}$, and the *parents* of u are defined as $p(u) = \{v | v = t(e)$ and $e \in In_u\}$.

For $U \subseteq V$, $AllOut_U = \{e | e \in E$ and $t(e) \in U\}$, and $Out_U = \{e | e \in E$ and $t(e) \in U$ and $h(e) \notin U\}$; and $AllIn_U = \{e | e \in E$ and $h(e) \in U\}$, and $In_U = \{e | e \in E$ and $h(e) \in U$ and $t(e) \notin U\}$. We can easily observe that $Out_U \subseteq AllOut_U$ and $In_U \subseteq AllIn_U$.

Let P_{uv} be a path from u to v in G; then v is *reachable* from u. All vertices reachable from u including itself in G are u's *descendants*, denoted as $des(G, u)$ or $des(u)$ if G is understood from the context.

A path P_{uv} is said to be a *shortest path*, denoted as SP_{uv}, if it is not longer than any other possible path P^*_{uv}. Given any $v \in V$, v could have more than one shortest path from a vertex u in G, and all v's shortest paths are of the same shortest distance. The shortest distance from u to v in G is denoted as d_{uv} when G is understood from the context. Given a digraph $G = (V, E, w)$, an SPT rooted at a vertex or source s, denoted as T_s, is a tree with root s and $\forall v \in des(s)$, $v \neq s$, T_s contains an SP_{sv}. Due to the structure of trees, $\forall v \in des(s)$, $v \neq s$, T_s contains only one shortest path SP_{sv}. Let $e \in E$; e is a *tree edge* wrt T_s if

[1] From now on, *MBallString* refers to *MBallStringInc* and the original *BallStringDec*. *BallStringDec* is the *BallString* algorithm in [28] when the input is a set of edge weight decreases.

$e \in E(T_s)$; otherwise it is a *non-tree edge*. Given $v \in V(T_s)$, the subtree rooted at v in T_s is denoted as $SubT_{sv}$.

Given G and $s \in V$, let T_s be an SPT. For any vertex $v \in V(T_s)$, the *shortest path parent* of v in T_s, denoted as $spp_{T_s}(v)$, is the parent of v in T_s; the *shortest path children* of v in T_s, denoted as $spc_{T_s}(v)$, contain all the children of v in T_s. Let $v \in V(T_s)$ and $v \neq s$. If s is understood in the context, then SP_{sv}, d_{sv}, $spp_{T_s}(v)$, $spc_{T_s}(v)$, and $SubT_{sv}$ are simply denoted as SP_v, d_v, $spp(v)$, $spc(v)$, and $SubT_v$.

Given the simple digraphs $G = (V, E, w)$ and $G' = (V, E, w')$, such that $w' \neq w$, G' is denoted as the *updated digraph*. In this paper, $G \to G'$ is achieved by an application of a set ε of edge weight updates: $\varepsilon = \{\langle e_i, \tau_i\rangle | e_i \in E \text{ and } -w(e_i) \leq \tau_i < \infty\}$, $\forall\langle e_i, \tau_i\rangle \in \varepsilon$, and $w'(e_i) = w(e_i) + \tau_i$.

Given G, G', $s, v \in V$, let $\mathbf{SP_v}$ and $\mathbf{SP'_v}$ be the sets of shortest paths from s to v in G and G', respectively. We define v as *unaffected* if $\mathbf{SP_v} = \mathbf{SP'_v}$; otherwise it is *affected*. More specifically, let SP'_v and d'_v denote any shortest path and the shortest distance from s to v in G', respectively; we say SP'_v *equals* SP_v if $V(SP'_v) = V(SP_v)$ and $\forall u \in V(SP'_v)$, $d'_u = d_u$.

2.2 Problem Definition

In order to solve the *DSP* problem, a brute-force solution is to run *Dijkstra*'s algorithm for a source s over G'. It is straight-forward but may not be effective all the times, since no previously-computed result is re-used. When T'_s is not much different from T_s, in terms of structural changes, it is more beneficial to construct T'_s from T_s than from scratch. In this paper, we study algorithms that can solve the *DSP* problem efficiently. In particular, we are interested in fast algorithms that solve the problem incrementally by re-using information in the outdated SPT.

2.3 Algorithmic Notation and Basic SPT Properties

In addition to the definitions we have introduced in this section so far, there are some notation used in the description of the coming algorithms.

Given G and $s \in V$, let T_s be an SPT rooted at s in G. Any vertex $v \in V(T_s)$ has four associated properties: d_v, *spp*(v), *spc*(v), and *status*(v). The first three properties indicate v's shortest distance from s and v's shortest path parent and children in T_s. The last one, *status*(v), usually has two states: *open* or *closed*. Some algorithms use *status*(v) to indicate whether v needs to be processed (*open*) and whether v is consolidated[2] (*closed*).

In the *DSP* problem, let T'_s be an SPT rooted at s in G'. Our *SPT* algorithms, except *Dijkstra*, compute T'_s from T_s, and thus all these are *incremental* algorithms. More specifically, they take T_s as an input, update the properties of some affected vertices in T_s, and then, at the end, return the updated T_s, which is T'_s.

In the description of all the incremental algorithms discussed in this work, we use a `hat`($\widehat{\ }$) over an object

[2] The term "consolidated" will be defined later.

to indicate the current state of that object. For example, $\widehat{T}$ denotes any intermediate tree, in which some vertices' properties are being updated during the execution of an algorithm; $\widehat{d_v}$ and $\widehat{spp_v}$ denote the shortest distance and shortest path parent of v from the source in $\widehat{T}$. In addition, we use a `prime`($'$) with an object to indicate its final status in the modified graph G'. For example, d'_{sv} is the new shortest distance from s to v in G', and $w(e)'$ is the new weight of e in G'. We prove the following properties.

Lemma 2.1 *When edge weights are only increased, for any vertex v, either $d'_v > d_v$ or, $d'_v = d_v$ and $\mathbf{SP_v} \supseteq \mathbf{SP'_v}$.*

Proof Since all edge weight changes are increases, $d'_v \geq d_v$ must hold. Let P'_v be any shortest path from s to v in G'; in other words, $P'_v \in \mathbf{SP'_v}$ and the length of P'_v equals d'_v. If $d'_v = d_v$, P'_v must contain no modified edges and P'_v must also be in SP_v. Therefore, if $d'_v = d_v$, $\mathbf{SP_v} \supseteq \mathbf{SP'_v}$. On the other hand, if $d'_v \neq d_v$, then $d'_v > d_v$. Thus, Lemma 2.1 holds. ∎

Lemma 2.2 *When edge weights are only decreased, for any vertex v, either $d'_v < d_v$ or, $d'_v = d_v$ and $\mathbf{SP_v} \subseteq \mathbf{SP'_v}$.*

Proof This can be proven with a similar argument as in Lemma 2.1. ∎

All incremental algorithms in this paper only process affected vertices: some process all of them, whereas others process only some of them. Any processed vertex v is *consolidated* if the distance assigned by the algorithm equals the final optimal value d'_v and the path constructed by the algorithm is an SP'_v.

At any instant of algorithm executions, an affected but non-consolidated vertex is denoted as a *boundary vertex* if it has either at least one unaffected parent or an affected but consolidated parent; otherwise, it is an *inner vertex.* A *boundary edge* is an incoming edge of a boundary vertex that has either an unaffected tail or an affected and consolidated tail. The *candidate parent* of a boundary vertex v is the tail u^* of a v's boundary edge, such that $d_{u^*} + w(u^*, v)$ is minimum among all tails of v's boundary edges.[3] The *candidate distance* of a boundary vertex v is provided by $d_{u^*v} + w(u^*, v)$, given that u^* is the candidate parent of v. The *candidate path* SP^*_v for boundary vertex v is the shortest path SP_{u^*} concatenated by (u^*, v).

2.4 Data Structures

There are a few important data structures that are shared by all algorithms: Graph G; SPT T_s, rooted at vertex s; and minimum-priority queue Q.

Conceptually, G contains a vertex set V and an edge set E. Each vertex v is identified by a key (the ID of v), and so is each edge e. Each vertex in a graph G has a list of incoming and a list of outgoing edges. Each edge e in a graph is assigned with a weight $w(e)$. T_s is represented by the vertices' auxiliary information

[3]If more than one tail provides the same minimum distance to v, any one of them can be taken as the candidate parent of v.

set, which is identified by the ID of the vertex; the auxiliary information, aux, for vertex v contains $spp(v)$, $spc(v)$, d_v, and $status(v)$.

An entry in Q is of the format $\langle ver,\ data,\ key\rangle$, in which ver is a vertex and is unique among all entries, $data$ contains some useful information of ver but is optional, key is the value on which entries are ranked. In those algorithms, key could be a pair of values, $\langle value1, value2\rangle$. Entries are ranked on $value1$ first; then on $value2$. If more than one entry has same key, then the sequence among them is arbitrary.

Q supports four associated instructions. The instruction $ENQUEUE(Q, \langle ver,\ data,\ key\rangle)$ adds one entry of vertex ver to Q. If ver is already in Q, the entry will replace the old ones only if the new key is smaller. In other words, at any instant, only one entry is maintained for each ver in Q. The instruction $EXTRACTMIN(Q)$ selects and removes an entry $\langle ver,\ data,\ key\rangle$ with the minimum key. The vertex ver is said to be *extracted* in this operation. $ADJUST(Q, \langle ver,\ data,\ key\rangle)$ enqueues this entry if no entry of ver exists in Q, or sets $data$ and key of ver's entry in Q as specified. The last instruction $REMOVE(Q, ver)$ removes the entry of ver from Q.

3 Related Work

There are many research efforts reported in the literature of maintaining shortest paths on dynamic graphs. We are interested in algorithms for graphs of non-negative edge weights. For all pairs shortest paths problem, we refer to the algorithms and experimental results such as those in [14, 13, 11]. Due to the requirement of returning exact shortest paths, previously suggested randomized or approximate algorithms [24, 36, 13] are not directly applicable to our problem. Among all the deterministic incremental algorithms, some require special properties on the graph which are less general than what is assumed in our work. For example, some maintain shortest paths in planar graphs [25, 16]; some require unweighted graphs, such that all edges have a weight of 1 [7, 18]; and some allow only integer edge weights that are less than a certain constant C [24, 6].

Over the past few decades, plenty of deterministic algorithms, which require no specific properties on graphs, have been proposed for this problem [34, 19, 32, 31, 28, 35, 29, 9]. Moreover, plentiful empirical studies have been conducted [12, 20, 17, 5, 22, 9]. Work has also been done on speeding up the search process by reducing the size of heap required in some SPT algorithms [9]. In the rest of this section, we review some proposals that fit into categories of our interests and highlight our contribution at the end.

3.1 *FMN*

Frigioni, Marchetti-Spaccamela and Nanni in [18] propose a complexity model to evaluate the theoretical performance of an SPT algorithm, which specifies using a function of the number of "locally-affected" vertices. This model captures the intrinsic cost required by any incremental algorithm after each input update.[4] Following that, in [19], the same authors propose a semi-dynamic algorithm *FMN* for maintaining

[4]In this work, we employ their model in our complexity analysis.

an SPT in a dynamic graph. *FMN* uses the notion of the *level* of an edge and the notion of the *ownership* of a vertex. Ownership information is used to bound the number of edges scanned each time a vertex changes its distance from s. Every edge (x, y) has an owner that must be either x or y. *FMN* follows the similar flow of *Dijkstra*, except that it tries to visit a smaller number of edges. The authors claim that *FMN* has the best theoretical complexity, but it maintains complex data structures, *e.g.*, levels of edges.

3.2 *RR*

Ramalingam and Reps in [30, 32] propose a semi-dynamic algorithm *RR*, which maintains all shortest paths from the source in a dynamic graph. However, *RR* handles single edge weight update only. After the shortest distances of all vertices have been computed, $\forall(x, y) \in E$, the "side-track value" r_{xy}, which is defined as $d_x + w(x, y) - d_y$, is computed. A side-track value of zero indicates that the edge is on at least one shortest path SP_y in G. A *Shortest Path Graph* (*SPG*) is constructed that contains all edges with zero side-track value. The advantage of maintaining an SPG is that, in the case of edge weight increases, SPG enables one to process only vertices that have to be processed because their shortest distances are increased. However, the trade-off lies in the maintenance of an SPG. The side-track value of any edge (x, y) needs to be re-computed once $w(x, y)$, d_x or d_y is updated. Nevertheless, Demetrescu et al. in [12] and Frigioni et al. in [20] illustrate by experimental results that *RR* performs better than *FMN* in most cases.

3.3 *BallString*

There are also incremental shortest paths algorithms that handle multiple edge weight changes.

Narváez, Siu and Tzeng in [27] propose a general framework for several well-known SPT algorithms when the update is a single edge weight update. The idea is to re-compute only the affected part of an SPT. An intelligent approach to re-compute shortest paths in the case of multiple edge weight updates is proposed by the same authors in [28]. We denote their algorithm as *BallString* since it is based on a ball-and-string model. This model illustrates how affected balls re-arrange themselves in a natural way into their optimal positions when the length of a string is increased or decreased. By simulating the dynamics of the balls in this model, *BallString* processes affected vertices in the most economical way: it always chooses the vertex of least distance increase (in the case of edge weight increases) or most distance decrease (in the case of edge weight decreases) to consolidate. In addition, it always tends to consolidate as many vertices as possible in an iteration.[5]

This approach greatly reduces the number of iterations required for the same set of affected vertices and totally eliminates unnecessary structural changes. However, this idea induces "duplicate distance updates" in the case of edge weight decreases. At the same time, unfortunately, *BallString* is wrong for a certain case of multiple edge weight increases. The following is a simple example to show its incorrectness.

[5]In [28], the authors denote the set of vertices consolidated in an iteration as a *branch*.

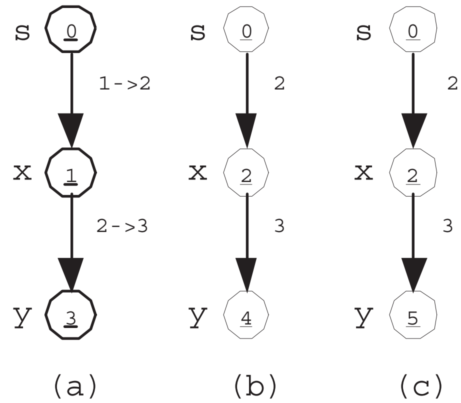

Figure 1: Counter-example of *BallString*. (a) G and T_s; (b) G' and the incorrect T'_s returned by *BallString*; (c) G' and T'_s

Example 3.1 In this example, $V = \{s, x, y\}$, $E = \{(s, x), (x, y)\}$ (thus $SP_y = s \to x \to y$); both $w(s, x)$ and $w(x, y)$ are increased by 1. According to *BallString*, in the initialization step, both x and y are marked as "floating", and only x is enqueued. The entry for x in the priority queue is $\langle x, s, \langle 1, 2\rangle\rangle$, indicating that x's shortest distance will be increased by 1 and its new distance will be 2, after s becomes x's shortest path parent. The queue entry format, in this case, is $\langle vertex, candidate\ parent, \langle \delta, candidate\ distance\rangle\rangle$.[6] However, the distance change of y caused by the increase of $w(x, y)$ is not enqueued, because x is y's only parent, and x is also "floating" at this point of time. Then *BallString* goes into iterations. In the first iteration, the entry of x is extracted from Q, $SubT_x$ (containing vertices x and y) is consolidated, such that x's and y's distances are increased by 1, and both are marked back to "anchored" (meaning that both have obtained their final optimal distances). Since no more entry exists in the priority queue, the algorithm ends. The result is given in Figure 1 (b), in which d'_y is wrong. The correct T'_s is given in Figure 1 (c). The error is due to some edge weight increase is being ignored. ∎

3.4 *DynamicSWSF-FP*

FMN, *RR*, *BallString* are all semi-dynamic shortest paths algorithms. Ramalingam and Reps in [31] propose a fully-dynamic algorithm, *DynamicSWSF-FP*. The main idea is as follows. At any instant, a "right hand side value" (rhs), denoted as $rhs(v)$, is maintained for every vertex v in G. The value records the shortest distance v could get, based on all parents p of v at that time. Given the shortest distance information d_v for each vertex v in G, we have $d_v = rhs(v)$ before any input edge weight updates. After the input edge weight updates are applied to G, *DynamicSWSF-FP* gradually updates the affected vertices' shortest distances, and, at the end, all vertices' shortest distances are equal to their $rhs(v)$ again.

A disadvantage of *DynamicSWSF-FP* is that it computes the rhs value too often, which leads to a

[6] When a vertex is enqueued, its parent must be "anchored".

high number of edge visits. In the same paper [31], the authors suggest some improvement on computing rhs values incrementally. Notice that in *DynamicSWSF-FP*, the rhs values are computed from scratch per request. The authors maintain a heap for each affected vertex. The improved algorithm is proven to be correct, but too many heaps are not practical.

3.5 Contributions

Our contribution in this work is that we propose a few incremental SPT algorithms based on previous work. We amend *BallString* by proposing *MBallStringInc* that updates SPTs correctly in the case of multiple edge weight increases. We propose a dynamic version of *Dijkstra*, which we call *DynDijkstra*. *DynDijkstra* can be considered as a generalization of the dynamic version of *Dijkstra* proposed in [27] by allowing multiple edge updates. *DynDijkstra* are two semi-dynamic algorithms that can handle multiple edge weight increases and decreases. For *DynamicSWSF-FP*, we suggest *MFP* by applying some optimizations on re-computing rhs values and to compute an SPT. In addition, we derive general frameworks for describing *DynDijkstra* and *MBallString*.

For each of the following proposed algorithms: *DynDijkstra*, *MBallStringInc*, and *MFP*, we prove its correctness and analyze its complexity. Furthermore, we conduct extensive experiments on all proposed algorithms. The set of experimental results is our another contribution. We test our algorithms on two types of graphs. One is graphs that are extracted from the real-life Connecticut road system [1]. These graphs are relatively sparse. The other one is randomly generated graphs, which are relatively dense. We evaluate a few factors that might affect the performances of proposed algorithms. We vary the graph size, the percentage of changed edges, and the percentage of weight changed. We first show that the weight changes have little effect on the performance of the incremental algorithms investigated. We also show that, in general, *MBallStringInc* and *DynDijkDec* have the best performance for the increases and decreases cases, respectively. We then combine these two algorithms together to form a new incremental algorithm *MBSDD*. We show experimentally that *MBSDD* and *DynDijkstra* have the best overall performance for the road and random mixed cases, respectively.

4 Semi-Dynamic Algorithms

In this section, we introduce a few semi-dynamic SPT algorithms for *DSP* problem. In Sections 4.1 and 4.2, algorithms *DynDijkstra* and *MBallString* are presented.[7] The correctness proofs of these algorithms are given in the *Appendix*.

There are some properties of input changes, which are shared by all SPT algorithms for dynamic graphs. We can break down the input changes to four cases: *tree edge increase*; *tree edge decrease*; *non-tree edge increase*; and *non-tree edge decrease*. Among these, non-tree edge increase has no effect on an SPT. In the

[7]The original *BallStringDec* algorithm will not be repeated in this paper.

following discussion, we are interested in the remaining three cases in computing an SPT after edge weight changes.

According to Lemmas 2.1 and 2.2, there may exist some affected vertices v such that $d'_v = d_v$ and some shortest path SP_v remains the same in G'. Since an SPT only records one shortest path for each vertex, if T_s happens to contain that unchanged SP_v, then from T_s's point of view, v is more like unaffected than affected. Thus, we define an affected vertex v as *locally-not-affected* in T_s if SP_v in T_s remains the same in G'; otherwise, it is *locally-affected*. Let *Affected* be the set of affected vertices based on G and G', and let $\mathit{Affected}_{T_s}$ be the set of locally-affected vertices based on one T_s and G'. It is straight-forward by definition that $\mathit{Affected}_{T_s} \subseteq \mathit{Affected}$.

As we will soon see, all algorithms in this section only process locally-affected vertices, for ease of description and discussion, we denote unaffected vertices and locally-not-affected vertices together as *not-locally-affected*; and the definitions of boundary vertices, inner vertices, candidate parents, and candidate distances apply only to locally-affected vertices. For instance, a locally-affected but non-consolidated vertex is a *boundary vertex* if it has at least one not-locally-affected or locally-affected but consolidated parent, otherwise is an *inner* vertex. Note that not-locally-affected vertices in T_s all keep their optimal shortest paths and distances as in T_s. For any modified edge e, we denote $h(e)$ as *affected-head* if it is locally-affected, and as *affected-mini-root* if it is locally-affected but it has no locally-affected ancestor (except for itself) in T_s.

Both *DynDijkstra* and *MBallString* contain two individual algorithms corresponding to weight increases and decreases, respectively: *DynDijkInc* and *DynDijkDec*; *MBallStringInc* and *BallStringDec*. Two algorithms for increases fit into a general SPT computation framework, and so do the two for decreases.

All these algorithms consist of an initialization, follows by n iterations of a number of steps, where $n \geq 0$. We say an algorithm *executes* or *runs* n iterations. Similarly, the i^{th} iteration of an algorithm refers to the i^{th} iteration of these steps.

4.1 Algorithms *DynDijkstra* and *MBallString*: Edge Weight Increases

Given a graph G, a source vertex s, an SPT T_s, and a set of edges ε^+, such that $\forall e \in \varepsilon^+$, $w(e)$ is going to be increased, we are going to compute a new SPT T'_s on G'.

We propose two algorithms that compute a new valid T'_s by only processing locally-affected vertices in T_s. With T_s and ε^+, we are able to locate all locally-affected vertices first, then compute new shortest paths and distances for them. Now we prove the following.

Lemma 4.1 *In the case of edge weight increases, for each $v \in V(T_s)$, v is locally-affected in T_s if and only if it is a descendant of an affected-mini-root in T_s.*

Proof "If" If v is a descendant of an affected-mini-root, then at least one edge on SP_v is a modified tree edge. The vertex v is affected and SP_v will not remain the same in G', therefore it is locally-affected.

"Only if" If v is not a descendant of an affected-mini-root, then no edges on SP_v are modified tree edges. Since all input changes are edge weight increases, v cannot get a shorter distance in G'. Thus SP_v must also be a shortest path in G'. In other words, if v is affected, then it is locally-not-affected. ∎

Let us define following phases of operations as *framework F1*, that computes T'_s in case of edge weight increases:

Framework F1:

***Phase 1*:** We locate locally-affected vertices:

1.1 Given T_s and ε^+, construct $\widehat{T_s}$ from T_s by removing modified tree edges;
1.2 We locate locally-affected vertices v in $\widehat{T_s}$;

***Phase 2*:** We compute candidate distances of boundary vertices;

***Phase 3*:** We compute new shortest paths for locally-affected vertices:

As long as there are boundary vertices left, process them according to a certain sequence by repeating the following:
3.1 We consolidate locally-affected vertices and maintain tree edges;
3.2 We compute candidate distances for new boundary vertices.

DynDijkInc and *MBallStringInc* are two instances of framework $F1$; they locate the same set of locally-affected vertices using the same method. The difference between them is in Phase 3.1. *DynDijkInc* conducts "*vertex consolidation by distance*" – it consolidates locally-affected vertices one by one according to a non-decreasing order of new distances; whereas *MBallStringInc* conducts "*branch consolidation by δ (distance change)*" – it consolidates locally-affected vertices set by set according to a non-decreasing order of distance changes.

In Phase 1.2, we locate all locally-affected vertices by calling procedure *findLocallyAffectedVertices*. In Phase 2, boundary vertices are initially identified from the set of locally-affected vertices[8]. In Phase 3, the new SPT is computed by consolidating locally-affected vertices.

4.1.1 *DynDijkInc*

DynDijkInc first locates all locally-affected vertices, then it conducts vertex consolidation by distance.

DynDijkInc(G, s, $\widehat{T_s}$, ε^+)

Input: G is a simple directed graph, s is the source vertex, $\widehat{T_s}$ is an SPT rooted at s in G, and ε^+ is a set of edges whose weights are increased, such that $\forall e_i \in \varepsilon^+$, $w(e_i)$ is increased by $\tau_i > 0$.
Output: The SPT $\widehat{T_s}$ is a new SPT rooted at s in the updated graph G'.

[8]In our implementation, the set of boundary edges (and thus boundary vertices) are initially found as follows: if the number of locally-affected vertices is less the number of unaffected ones, then search incoming edges of locally-affected vertices for boundary edges, otherwise search outgoing edges of unaffected vertices.

Notation: For any vertex v, the notations of $spp(v)$, $spc(v)$, and d_v are wrt $\widehat{T_s}$. All the other notations are wrt G.

Step 1: Apply the set of edge weight changes to G, remove modified tree edges from $\widehat{T_s}$ and locate all locally-affected vertices.

1: $\varepsilon \leftarrow \emptyset$
2: **for each** $e_i \in \varepsilon^+$ **do**
3: $\quad w(e_i)' \leftarrow w(e_i) + \tau_i$
$\quad$/* If e_i is an edge in $\widehat{T_s}$, then remove it from $\widehat{T_s}$ and add it to ε.*/
4: $\quad t \leftarrow t(e_i),\ h \leftarrow h(e_i)$
5: $\quad$**if** $t = spp(h)$ **then**
6: $\quad\quad \widehat{spc(t)} \leftarrow \widehat{spc(t)} - \{h\},\ \widehat{spp(h)} \leftarrow \emptyset$
7: $\quad\quad \varepsilon \leftarrow \varepsilon \cup \{e_i\}$
8: $\quad$**end if**
9: **end for**
/* Find the set of locally-affected vertices based on $\widehat{T_s}$.*/
10: $\bar{N} \leftarrow findLocallyAffectedVertices(\widehat{T_s}, \varepsilon)$

Step 2: Enqueue boundary vertices with candidate distances. A vertex a is a boundary vertex iff $\widehat{d_a} \neq \infty$.

11: **for each** vertex $a \in \bar{N}$ **do**
12: $\quad \widehat{d_a} \leftarrow min(\{d_b + w(b,a)' | (b,a) \in In_a$ **and** $b \notin \bar{N}\} \cup \{\infty\})$
13: $\quad$**if** $\widehat{d_a} \neq \infty$ **then**
14: $\quad\quad ENQUEUE(Q, \langle a, b, \widehat{d_a} \rangle)$
15: $\quad$**end if**
16: **end for**

Step 3: Consolidate and relax locally-affected vertices one by one.

17: **while** $Q \neq \emptyset$ **do**
18: $\quad \langle y, x, d \rangle \leftarrow EXTRACTMIN(Q)$
$\quad$/* Re-assign the shortest path parent of y to x.[9]*/
19: $\quad \widehat{spc(x)} \leftarrow \widehat{spc(x)} \cup \{y\}$
20: $\quad p \leftarrow \widehat{spp(y)},\ \widehat{spc(p)} \leftarrow \widehat{spc(p)} - \{y\}$
21: $\quad \widehat{spp(y)} \leftarrow x$
$\quad$/* Relax outgoing edges of the consolidated vertex y.*/
22: $\quad$**for each** $e \in Out_y$ **do**
23: $\quad\quad q \leftarrow h(e)$
24: $\quad\quad$**if** $\widehat{d_y} + w(e)' < \widehat{d_q}$ **then**
25: $\quad\quad\quad \widehat{d_q} \leftarrow \widehat{d_y} + w(e)'$
26: $\quad\quad\quad ENQUEUE(Q, \langle q, y, \widehat{d_q} \rangle)$
27: $\quad\quad$**end if**
28: $\quad$**end for**
29: **end while**
30: **return** $\widehat{T_s}$

In Step 1, *DynDijkInc* update edges' weights, remove modified tree edges from $\widehat{T_s}$, locates all locally-affected vertices by calling *findLocallyAffectedVertices*. In Step 2, all locally-affected vertices a are examined: if a is a boundary vertex, d_a is updated to its candidate distance, and a is enqueued into Q in the format of $\langle a,\ candidate\ parent,\ candidate\ distance \rangle$; if a is an inner vertex, d_a is set to ∞. From this point on, whenever a shorter candidate distance is located for any vertex y, $\widehat{d_y}$ and candidate parent are updated. In Step 3, *DynDijkInc* goes into iterations. Each iteration consolidates one locally-affected vertex y of the minimum candidate distance, updates y's incoming tree edge, and also relaxes y. In the relaxation part, after y is consolidated, for each child q, *DynDijkInc* updates q's distance and candidate parent information

[9] Line 20 is skipped if $spp(y)$ does not exist. This applies to all other algorithms.

if a shorter distance is computed. The iterations end when Q becomes empty, which indicates no boundary vertices left. Now we look at an increase example.

Example 4.1 As shown in Figure 2 (a), the weights of edges (c, g) and (g, j) are increased. In Step 1, *DynDijkInc* removes modified tree edges and locates all locally-affected vertices, *i.e.,* $\{g, k, o, p, j, i, n\}$. In Step 2, *DynDijkInc* computes candidate distances for boundary vertices, *i.e.,* $\{g, i, j, k, p\}$, and enqueues one entry for each. After that, *DynDijkInc* consolidates locally-affected vertices by distance, and it also updates the incoming tree edge to each consolidated vertex. Since it is just like *Dijkstra*, we leave out the detail of intermediate steps. As shown in Figure 2 (b), all affected vertices are processed. ∎

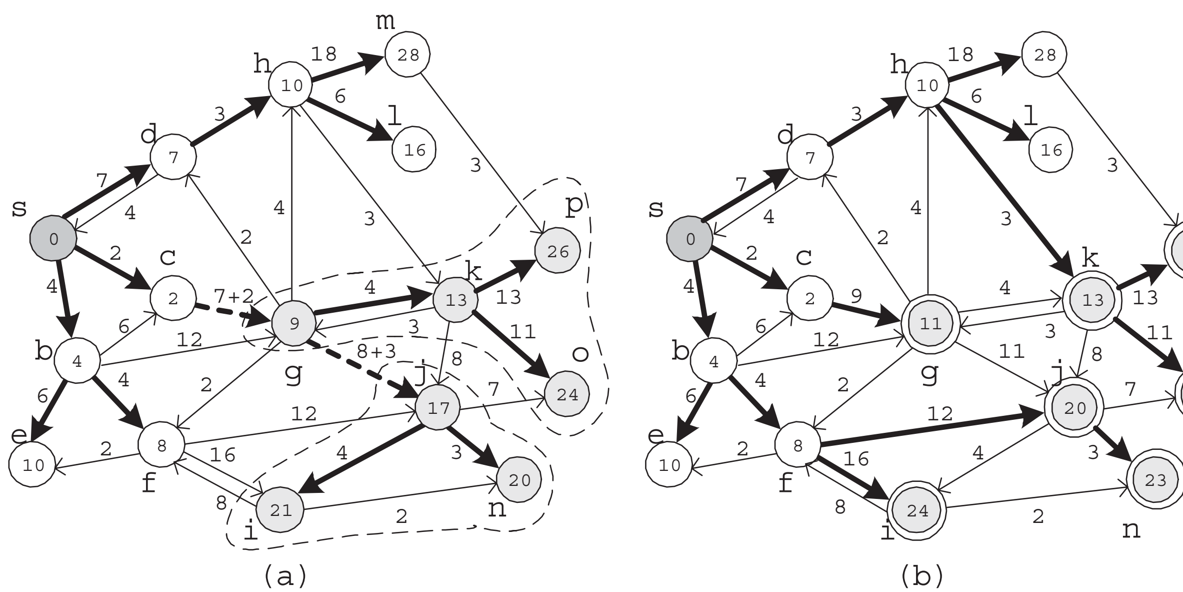

Figure 2: *DynDijkInc* on an example. (a) An SPT T_s rooted at s in a graph G. A vertex is denoted by a single circle, and it is denoted by a letter, an edge is denoted by an arrow from the tail to the head, and the weight is numbered beside. In an SPT, a tree edge is highlighted by thick arrow, vertex's shortest distance is inside the vertex. In this example, $w(c, g)$ is increased by 2 and $w(g, j)$ is increased by 3, $\widehat{T_s}$ which is obtained from T_s by removing edges (c, g) and (g, j), is a forest of three trees: one rooted at s, one rooted at g and the other rooted at j. The latter two subtrees are circled by dashed line; (b) G' and T'_s. Legends: dashed arrow represents removed tree edge in G; locally-affected vertices are lightly shaded and extracted vertices are doubly circled.

From Figure 2 (a) and (b), we see that vertices $\{o, p\}$ are extracted despite that they have the same shortest distance and shortest parent in T'_s and T_s; and vertex i is linked to vertex f, even though i could have stayed with its old shortest parent j for the identical new shortest distance 24. In following subsection, we will see how *MBallStringInc* gets around the above undesirable result.

4.1.2 *MBallStringInc*

Paolo Narváez *et al.* propose an intelligent semi-dynamic algorithm *BallString* in [27, 26, 28]. Unfortunately, as illustrated in Section 3.3, their algorithm *BallStringInc* for multiple edge weight increases is not correct. Here, we propose an algorithm *MBallStringInc*, which is a slight modification of theirs, that correctly and efficiently computes a new SPT for multiple edge weight increases, by adapting the same branch closing idea.

MBallStringInc is another instance of framework F1. Unlike *DynDijkInc*, in Phase 3.1 of $F1$, *MBallStringInc* conducts branch consolidation by δ: it consolidates locally-affected vertices according to the non-decreasing sequence of distance changes δ's. For each locally-affected vertex v, the distance change of v, denoted as δ_v, is defined as $d'_v - d_v$. At the same time, *MBallStringInc* is more aggressive in that it consolidates a whole branch (a subtree) instead of one vertex. In addition, *MBallStringInc* does not set any tentative distances to locally-affected vertices (as *DynDijkInc* does) until they are consolidated; because the old distances of locally-affected vertices are required for the computation of δ values. This is also why *MBallStringInc* needs the status of *open* to trace the remaining locally-affected vertices.

MBallStringInc$(G,\ s,\ \widehat{T_s},\ \varepsilon^+)$

Input: G is a simple directed graph, s is the source vertex, $\widehat{T_s}$ is an SPT rooted at s in G, and ε^+ is a set of edges whose weights are increased. All vertices in $\widehat{T_s}$ are initially *closed*.
Output: The SPT $\widehat{T_s}$ is a new SPT rooted at s in the updated graph G'.
Notation: For any vertex v, the notations of $spp(v)$, $spc(v)$, d_v and $status(v)$ are wrt $\widehat{T_s}$. All the other notations are wrt G.

Step 1: Apply the set of edge weight changes to G; if a modified edge is a tree edge, remove the edge from $\widehat{T_s}$; and locate all locally-affected vertices.

$\varepsilon \leftarrow \emptyset$
for each $e_i \in \varepsilon^+$ **do**
  $w(e_i)' \leftarrow w(e_i) + \tau_i$
  $t \leftarrow t(e_i),\ h \leftarrow h(e_i)$
  /* If e_i is an edge in $\widehat{T_s}$, then remove it from $\widehat{T_s}$ and add it to ε.*/
  if $t = spp(h)$ **then**
    $\widehat{spc(t)} \leftarrow \widehat{spc(t)} - \{h\},\ \widehat{spp(h)} \leftarrow \emptyset$
    $\varepsilon \leftarrow \varepsilon \cup \{e_i\}$
  end if
end for
/* Find the set of locally-affected vertices based on $\widehat{T_s}$.*/
$\bar{N} \leftarrow findLocallyAffectedVertices(\widehat{T_s}, \varepsilon)$

Step 2: Find candidate distances/parents for boundary vertices.

for each vertex $a \in \bar{N}$ **do**
  $status(a) \leftarrow open$
  $newdist \leftarrow min(\{d_b + w(b,a)' | (b,a) \in In_a$ **and** $b \notin \bar{N}\} \cup \{\infty\})$
  if $newdist \neq \infty$ **then**
    $\delta \leftarrow newdist - d_a$
    $ENQUEUE(Q, \langle a, b, \langle \delta, newdist\rangle\rangle)$
  end if
end for

Step 3: Consolidate and relax locally-affected vertices set by set.

while $Q \neq \emptyset$ **do**
  $\langle y, x, \langle \delta, d\rangle\rangle \leftarrow EXTRACTMIN(Q)$

/* Re-assign the shortest path parent of y to x.*/

21: $\widehat{spc(x)} \leftarrow \widehat{spc(x)} \cup \{y\}$

22: $p \leftarrow \widehat{spp(y)},\ \widehat{spc(p)} \leftarrow \widehat{spc(p)} - \{y\}$

23: $\widehat{spp(y)} \leftarrow x$

/* Consolidate all descendants of y.*/

24: $N \leftarrow des(\widehat{T_s}, y)$

25: **for each** $v \in N$ **do**

26: $\widehat{d_v} \leftarrow d_v + \delta$

27: $\widehat{status(v)} \leftarrow closed$

28: **if** $v \in Q$ **then**

29: $REMOVE(v, Q)$

30: **end if**

31: **end for**

/* Relax outgoing edges of just consolidated vertices.*/

32: **for each** $e \in Out_N$ **do**

33: **if** $\widehat{status(h(e))} = open$ **then**

34: $newdist \leftarrow \widehat{d_{t(e)}} + w(e)'$

35: $\delta \leftarrow newdist - \widehat{d_{h(e)}}$

36: $ENQUEUE(Q, \langle h(e), t(e), \langle \delta, newdist \rangle \rangle)$

37: **end if**

38: **end for**

39: **end while**

40: **return** $\widehat{T_s}$

Step 1 and Step 2 of *MBallStringInc* are almost the same as those of *DynDijkInc*, except that in Step 2, *MBallStringInc* sets all locally-affected vertices to *open*, and for each boundary vertex, it enqueues $\langle boundary\ vertex,\ candidate\ parent,\ \langle \delta, candidate\ distance \rangle \rangle$.

In Step 3, *MBallStringInc* extracts the boundary vertex y of the least shortest distance increase δ in line 20, updates y's new shortest path parent; it also selects all vertices in $des(\widehat{T_s}, y)$ into N to consolidate in the next step.

Then, *MBallStringInc* consolidates vertices $v \in N$: it updates the shortest distance of vertex v by adding δ in line 26, and changes v to *closed* in line 27. In lines 28 - 30, if v is still in Q, then v is removed from Q, because v's optimal distance has been found, therefore no need to process it again.

Finally, *MBallStringInc* relaxes consolidated vertices. All remaining *open* vertices adjacent to any vertex in N now become boundary vertices, and the information (candidate parent, candidate distance, and δ) of each boundary vertex is enqueued into Q. *MBallStringInc* repeats consolidation and relaxation until no locally-affected vertices left.

Branch consolidation by δ enables that, locally-affected vertices of less distance increase are processed earlier. For each locally-affected vertex v, δ_v is defined as $d'_v - d_v$. In each branch (a subtree), the vertex without an incoming tree edge is denoted as *mini-root* in [28]. Basically, the algorithm computes T'_s by re-arranging the position of each branch, and applying the δ of the mini-root to all vertices in that branch. *MBallStringInc* yields the SPT that is least different from T_s in terms of tree structure. It is an efficient algorithm because it avoids unnecessary computation inside a branch.

Example 4.2 As shown in Figure 3 (a), after modified tree edges (c, g) and (g, j) are removed from T_s, vertices $\{g, k, p, o, j, i, n\}$ are located as locally-affected in Step 1. Then entries of all boundary vertices $\{g, i, j, k, p\}$ are enqueued in Step 2: $\langle g, c, \langle 2, 11\rangle\rangle$, $\langle i, f, \langle 3, 24\rangle\rangle$, $\langle j, f, \langle 3, 20\rangle\rangle$, $\langle k, h, \langle 0, 13\rangle\rangle$, and $\langle p, m, \langle 5, 31\rangle\rangle$. In Step 3, in the first iteration, k, whose entry has the minimum δ, is extracted, vertices in $des(\widehat{T_s}, k)$, *i.e.,* $\{k, o, p\}$ are selected into N in line 24, and the whole branch is cut from g and linked to h. Vertices in N are consolidated and entry of p is removed from Q. Since the open vertex j does not get a smaller δ from the just consolidated vertex k, there is no change in Q. In the following iteration, entry of g is extracted, and only $\{g\}$ is returned by $des(\widehat{T_s}, g)$ because modified tree edge (g, j) was removed in Step 1, and tree edge (g, k) was removed after k is extracted. At this time, two entries exist in Q: $\langle i, f, \langle 3, 24\rangle\rangle$ and $\langle j, f, \langle 3, 20\rangle\rangle$. Hence in the next iteration, the entry of j is extracted, and all current descendants of j are consolidated, including i (the entry of i is removed from Q). The resulting T'_s is given in Figure 3(b). ∎

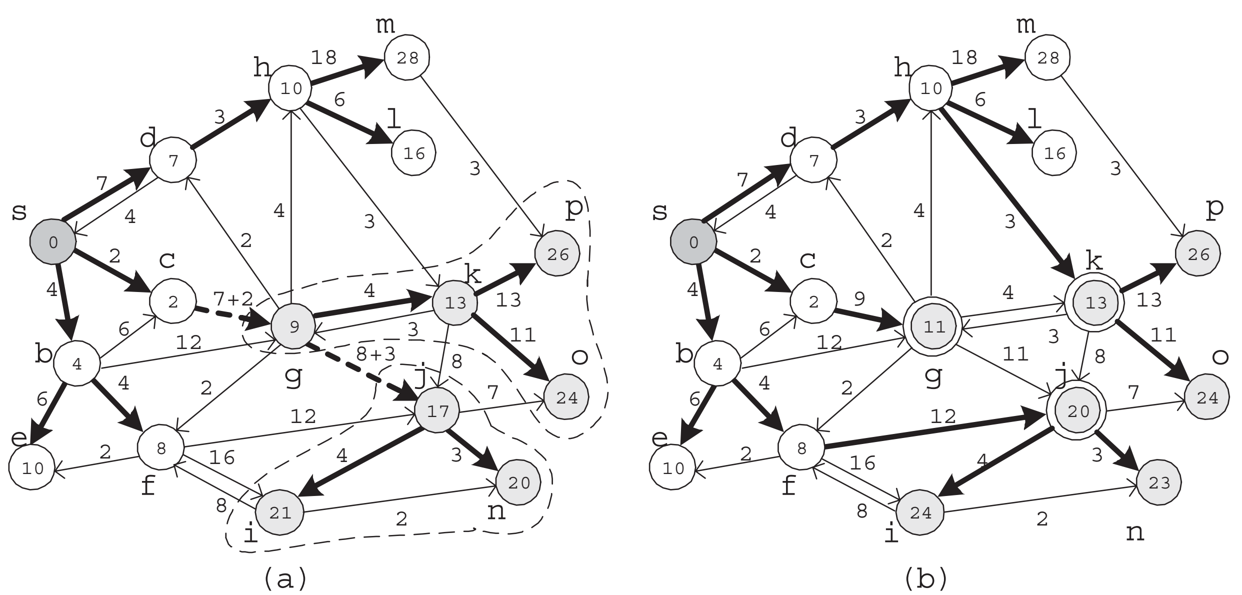

Figure 3: *MBallStringInc* on an example. (a) Graph G and the forest $\widehat{T_s}$ after modified tree edges (c, g) and (g, j) are removed, and dashed circle denotes a set of locally-affected vertices; (b) the final SPT T'

Compared with *DynDijkInc*, *MBallStringInc* has three advantages. Firstly, *MBallStringInc* runs fewer iterations than *DynDijkInc* does on the same set of locally-affected vertices. The reason is that *MBallStringInc* (the same as in *BallStringInc*) removes an entry directly from Q if the corresponding vertex v is already consolidated. By contrasting Figure 2 (b) and Figure 3 (b), we see *DijktraInc* has 7 iterations corresponding to all 7 affected vertices, whereas *MBallStringInc* only has 3. Secondly, *MBallStringInc* consumes much smaller number of tree edge updates because it changes only the incoming tree edges of mini-roots. In this example, *DynDijkInc* updates 7 tree edges, while *MBallStringInc* updates only 3. Thirdly, *MBallStringInc* changes a locally-affected vertex's shortest path parent only when compulsory, as exemplified by vertices p, o, and i, in Figures 2 and 3.

4.2 Algorithms *DynDijkstra* and *MBallString*: Edge Weight Decreases

Given a graph G, a source vertex s, an SPT T_s, and a set of edges ε^-, such that $\forall e \in \varepsilon^-$, $w(e)$ is going to be decreased, we are going to compute a new SPT T'_s.

Lemma 4.2 *In the case of edge weight decreases, for any locally-affected vertex v in T_s, $d'_v < d_v$.*

Proof According to the definition of a locally-affected vertex, SP_v does not remain the same in G'. Since v must be an affected vertex, according to Lemma 2.2, it is not possible to have $d'_v = d_v$. Thus $d'_v < d_v$ must stand. ∎

Unlike in the increases case, we cannot predict the set of locally-affected vertices without computing the new distances for them, because for each modified edge e, all vertices reachable from $h(e)$ in G might be locally-affected.

To compute T'_s in this case, we start from all affected-heads, then we traverse all reachable vertices until no shorter distances are located. In other words, we locate locally-affected vertices and consolidate them in an *interleaved* manner, as stated in [30].

Let us define following phases of operations as *framework F2* which computes T'_s in case of edge weight decreases:

Framework F2:

Phase 1: We compute new candidate distances for all affected-heads;

Phase 2: We compute new shortest paths for all locally-affected vertices:

As long as there are locally-affected vertices left, we process them according to a certain sequence by repeating the following:

2.1 We consolidate locally-affected vertices and maintain tree edges;

2.2 We compute candidate distances for remaining locally-affected vertices.

4.2.1 *DynDijkDec*

Algorithm *DynDijkDec* does precisely what framework F2 describes, and as in *DynDijkInc*, *DynDijkDec* conducts, in Phase 2, vertex consolidation by distance.

In Step 1, *DynDijkDec* checks each modified edge e, if $h = h(e)$ is an affected-head, then a shorter distance, $\widehat{d_{t(e)}} + w(e)'$, is given to h, and h is enqueued. In Step 2, *DynDijkDec* greedily examines all descendants v of h in G' for locally-affected vertices. If v is locally-affected, then all its children will be examined as well. Otherwise, v will not induce its children to be examined. By iterating this process, *DynDijkDec* eventually locates all locally-affected vertices. *DynDijkDec* updates the distance of each locally-affected vertex v whenever a shorter distance is located, and it updates v's incoming tree edge when v is extracted.

DynDijkDec$(G,\, s,\, \widehat{T_s},\, \varepsilon^-)$

Input: G is a simple directed graph, s is the source vertex, $\widehat{T_s}$ is an SPT rooted at s in G, and ε^- is a set of edges whose weights are decreased, such that $\forall e_i \in \varepsilon^-$, $w(e_i)$ is decreased by $-w(e_i) \le \tau_i < 0$.
Output: The SPT $\widehat{T_s}$ is a new SPT rooted at s in the updated graph G'.
Notation: For any vertex v, the notations of $spp(v)$, $spc(v)$ and d_v are wrt $\widehat{T_s}$. All others are wrt G.

Step 1: Apply the set of edge weight changes to G and enqueue affected-heads.
for each $e_i \in \varepsilon^-$ **do**
  $w(e_i)' \leftarrow w(e_i) + \tau_i$
  $t \leftarrow t(e_i),\ h \leftarrow h(e_i)$
  /* If the head of a modified edge is affected, update its distance and enqueue in Q.*/
  if $\widehat{d_t} + w(e_i)' < \widehat{d_h}$ **then**
    $\widehat{d_h} \leftarrow \widehat{d_t} + w(e_i)'$
    $ENQUEUE(Q, \langle h, t, \widehat{d_h} \rangle)$
  end if
end for
Step 2: Consolidate and relax locally-affected vertices.
while $Q \neq \emptyset$ **do**
  $\langle y, x, d \rangle \leftarrow EXTRACTMIN(Q)$
  /* Re-assign the shortest path parent of y to x.*/
  $\widehat{spc(x)} \leftarrow \widehat{spc(x)} \cup \{y\}$
  $p \leftarrow \widehat{spp(y)},\ \widehat{spc(p)} \leftarrow \widehat{spc(p)} - \{y\}$
  $\widehat{spp(y)} \leftarrow x$
  /* Relax outgoing edges of consolidated vertex y.*/
  for each $e \in Out_y$ **do**
    $q \leftarrow h(e)$
    if $\widehat{d_y} + w(e)' < \widehat{d_q}$ **then**
      $\widehat{d_q} \leftarrow \widehat{d_y} + w(e)'$
      $ENQUEUE(Q, \langle q, y, \widehat{d_q} \rangle)$
    end if
  end for
end while
return $\widehat{T_s}$

Example 4.3 In Figure 4 (a), the weight of edges (c, g) and (g, j) will be decreased. In Step 1 of *DynDijkDec*, the weight of each modified edge is decreased, and entries of g and j are enqueued, because both g and j get shorter distances. In Step 2, the entry of g is extracted first, then g is relaxed so that entries of k and j are enqueued. Then entries of k, j, n, i, o, and p are extracted sequentially. The new shortest path parent for each extracted vertex is set to the candidate parent. For instance, j becomes o's new shortest path parent. As shown in Figure 4 (b), vertices g, k, j, n, i, o, and p turn out to be locally-affected, and all of them are processed by *DynDijkDec*. ∎

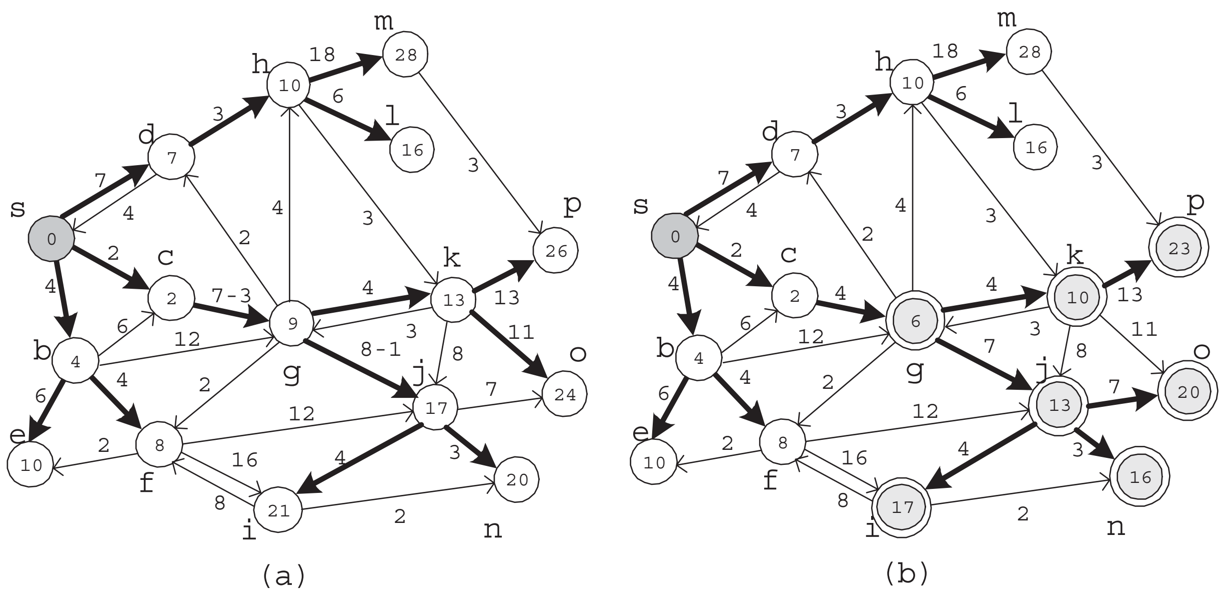

Figure 4: *DynDijkDec* on an example. (a) G and T_s, in which (c, g)'s weight will be decreased by 3 and (g, j)'s weight will be decreased by 1; (b) G' and T'_s.

4.2.2 *BallStringDec*

BallStringDec is presented in [28], therefore we do not repeat it here. It is totally in accordance with framework F2. More specifically, for each boundary vertex, the potential distance and corresponding δ are computed; in each iteration, the boundary vertex v with the minimum δ is extracted, and the distances of vertices in $\widehat{SubT_v}$ are decreased by δ. Here, we run *BallStringDec* with our decrement example. After that, we provide some further discussion of this algorithm.

Example 4.4 In Phase 1 of framework F2, two entries are enqueued: $\langle g, c, \langle -3, 6\rangle\rangle$ and $\langle j, g, \langle -1, 16\rangle\rangle$. Then in Phase 2, the entry of g is extracted first. Vertex g keeps its old shortest path parent; all its descendants in T_s, *i.e.,* N=$\{g, k, j, n, i, o, p\}$, are processed – their distances are decreased by 3. When *BallStringDec* relaxes vertices in N, it enqueues a new entry for j, *i.e.,* $\langle j, g, \langle -1, 13\rangle\rangle$. In the next iteration, the entry of j is extracted. Vertex j's shortest path parent remains to be g; all descendants of j in $\widehat{T_s}$, *i.e.,* $\{j, i, n\}$, are processed - their distances are decreased by 1. The relaxation on j enqueues entry for o, *i.e.,* $\langle o, j, \langle -1, 20\rangle\rangle$. In the last iteration, the entry of o is processed. Vertex o's shortest path parent switches to j and o's shortest distance is now 20. The new SPT T'_s is in Figure 5 (b). Contrasted with Figure 4 (b), *DynDijkDec* extracts all 7 affected vertices, whereas here *BallStringDec* only extracts 3 vertices, *i.e.,* g, j, and o. ∎

The above example again illustrates the advantage of branch consolidation by δ: a fewer number of iterations and also a lesser number of tree edge updates. However, it also exemplifies *duplicate distance*

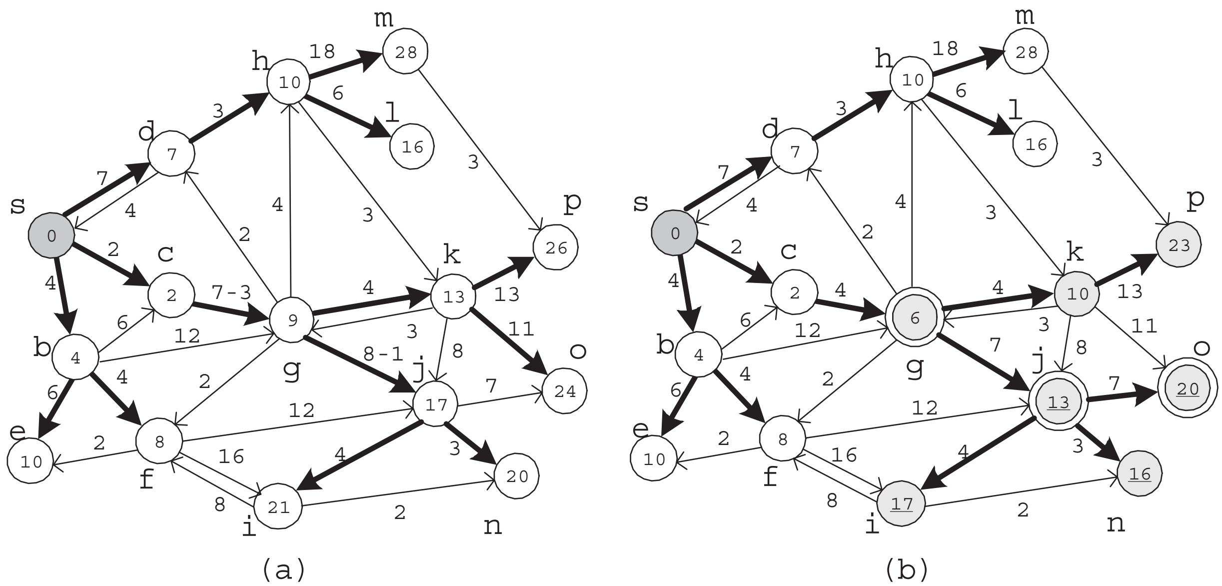

Figure 5: *BallStringDec* on an example. (a) G and T_s, in which (c, g)'s weight will be decreased by 3 and (g, j)'s weight will be decreased by 1; (b) G' and T'_s.

updates, *i.e.,* the distances of some vertices z are updated more than once, and vertex z is said to be *duplicate-updated*. In the above example, vertices j, n, i, and o are duplicate-updated. Moreover, a locally-affected vertex such as j could be enqueued more than once.

5 A Fully-Dynamic Algorithm

In the previous section, we introduce a few semi-dynamic SPT algorithms for the *DSP* problem. In this section, we present a fully-dynamic algorithm that can handle "multiple heterogeneous modifications"[31].

At any instant of the execution of *DynamicSWSF-FP*, we denote the *right hand side value* of v ($rhs(v)$) as $\min_{x \in p(v)}\{\widehat{d_x} + w(x,v)'\}$. We say parent x of v *satisfies* v, if $rhs(v) = \widehat{d_x} + w(x,v)'$. For any vertex $x \in p(v)$, x is a *satisfying-parent* of v, if x satisfies v, and in that case, v is a *satisfying-child* of x. Any affected vertex is processed differently according to whether $rhs(v)$ is greater than (*under-consistent*), equal to (*consistent*), or less than $\widehat{d_v}$ (*over-consistent*).

Let us analyze how this algorithm performs when changes are either increases or decreases, but not both. When all the input updates are edge weight increases, no vertices can have shorter distances. Hence, any affected vertex v is initially under-consistent, and $\widehat{d_v}$ will first be assigned the value of ∞, and then back to its correct value. Therefore, the affected vertices are enqueued and extracted twice. Similarly, when all the input updates are edge weight decreases, no vertices can have longer distances. Because of this, any enqueued vertex u can only be over-consistent, and $\widehat{d_u}$ is directly set to its rhs value and will not be processed again.

Consequently, the affected vertices are processed only once. From this analysis, we conclude that the case of the edge weight increases is always the worst scenario of *DynamicSWSF-FP*.

5.1 *MFP*

DynamicSWSF-FP conducts frequent edge visits and computations to maintain *rhs* values. Here, we apply some simpler optimizations, and their correctness can be verified easily from the transformation. Our optimizations are based on avoiding the unnecessary *rhs* value re-computation, and also simplifying the computation when it is possible. The first optimization is that, when an over-consistent vertex v is extracted ($\widehat{d_v}$ is decreased to $rhs(v)$), for each child q, *DynamicSWSF-FP* re-evaluates $rhs(q)$ according to the definition $\widehat{rhs(q)} = \min_{z \in p(q)}\{\widehat{d_z} + w(z,q)'\}$, whereas *MFP* incrementally re-computes $\widehat{rhs(q)}$ as $\min\{\widehat{rhs(q)}, \widehat{d_v} + w(v,q)'\}$. The second optimization is that, when an under-consistent vertex u is extracted ($\widehat{d_u}$ is set to ∞), *DynamicSWSF-FP* re-evaluates the rhs values of all u's children, whereas *MFP* re-evaluates the rhs values of all u's satisfying-children only. The reason is as follows. According to the definition, $\widehat{rhs(q)} = \min_{u \in p(q)}\{\widehat{d_u} + w(u,q)'\}$. We need to re-evaluate $rhs(q)$, if and only if q is a satisfying-child of u (before $\widehat{d_u}$ is set to ∞).

Furthermore, the original *DynamicSWSF-FP* computes *the shortest distance values only* without maintaining an SPT. To properly evaluate this with other incremental algorithms, *MFP* is designed to accept an outdated SPT with a set of edge weight changes, and to return a new SPT. Note that, for the tree structure, it is sufficient to maintain $spp(v)$ only for each vertex v in SPT. [10] We define a function $sap(v)$ that returns v's tentative satisfying parent when $sap(v)$ is called. In *MFP*, $spp(v)$ is updated by $sap(v)$ whenever $rhs(v)$ is re-computed. The following is an outline of *MFP*.

MFP$(G,\ s,\ \widehat{T_s},\ \varepsilon)$

Input: G is a simple directed graph, s is the source vertex in G, T_s is an SPT rooted at s in G, and ε is a set of edges such that $\forall e_i \in \varepsilon$, $w(e_i)$ will be increased by τ_i, where $\tau_i < 0$ or $\tau_i > 0$.
Output: The changed graph G' and the updated SPT $\widehat{T_s}$.
Notation: For any vertex v, the notations of d_v, $rhs(v)$, and $key(v)$ are wrt $\widehat{T_s}$. All the other notations are wrt G.
Step 1: updates G and enqueues inconsistent heads of modified edges, as in previous algorithms.
Step 2: process inconsistent vertices
while $Q \neq \emptyset$ **do**
 $\langle y, key\rangle \leftarrow EXTRACTMIN(Q)$
 if $\widehat{d_y} > \widehat{rhs(y)}$ **then**
  $\widehat{d_y} \leftarrow \widehat{rhs(y)}$ /*y is over-consistent*/
  /*check children to propagate the updates*/
  for each $e \in \mathbf{Out}_y$ **do**
   $q \leftarrow h(e)$
   $\widehat{rhs(q)} \leftarrow \min\{\widehat{rhs(q)}, \widehat{d_y} + w(e)'\}$ /*==***the first optimization***==*/
   $\widehat{spp(q)} \leftarrow sap(q)$
   if $\widehat{rhs(q)} \neq \widehat{d_q}$ **then**

---

[10] In DynDijkstra and MBallStringInc, for v, we need to maintain $spc(v)$ as well, because we need shortest path descendants information.

10: $\widehat{key(q)} \leftarrow \min\{\widehat{d}_q, \widehat{rhs(q)}\}$
11: $ADJUST(Q, \langle q, \widehat{key(q)} \rangle)$ /*enqueue q if it becomes inconsistent*/
12: **else**
13: $REMOVE(Q, q)$ /*removes q if it becomes consistent*/
14: **end if**
15: **end for**
16: **else**
17: $d \leftarrow \widehat{d}_y$
18: $\widehat{d}_y \leftarrow \infty$ /*y is under-consistent*/
19: **if** $\widehat{rhs(y)} \neq \infty$ **then**
20: $\widehat{key(y)} \leftarrow \widehat{rhs(y)}$
21: $ENQUEUE(Q, \langle y, \widehat{key(y)} \rangle)$ /*enqueue q if it becomes inconsistent*/
22: **else**
23: /*This is the similar to lines 5-15 in above. The second optimization is also applied here by checking satisfying-children of y for more inconsistent vertices. */
24: **end if**
25: **end if**
26: **end while**
27: **return** $\widehat{T}_s$

Example 5.1 In Figure 6, we apply the following edge weight updates: $w(c, g)$ is decreased by 1, $w(g, j)$ is increased by 3, and $w(f, i)$ is decreased by 8.

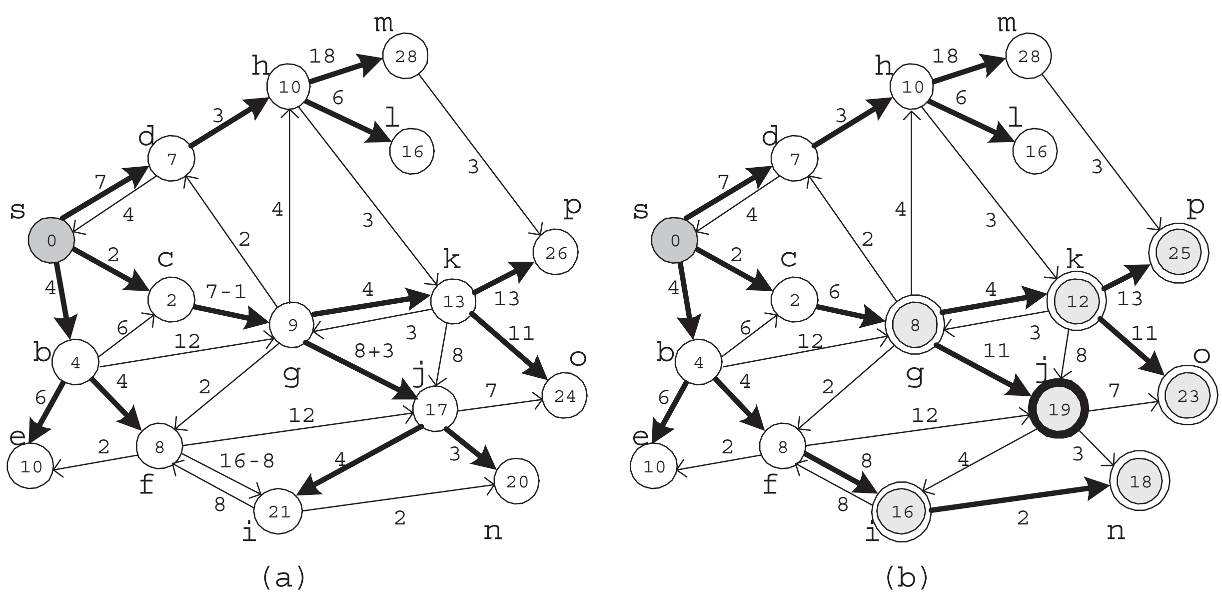

Figure 6: *MFP* on our mixed example. (a) G and T_s, in which $w(c, g)$ will be decreased by 1, $w(f, i)$ will be decreased by 8, and $w(g, j)$ will be increased by 3. (b) The final G' and T'_s, in which all the vertices are consistent and of the correct shortest distances. Legend: the vertices that are processed twice are circled by a heavy dark line.

In this example, two edges' weights will be decreased, and one edge's weight will be increased. *MFP* first examines that all modified heads, g, j, and i, become inconsistent. Their associated variables in the format of $\langle v, \widehat{d}_v, \widehat{rhs(v)}\rangle$ are $\langle g, 9, 8\rangle$, $\langle j, 17, 20\rangle$, and $\langle i, 21, 16\rangle$. Vertices g and i are over-consistent, and j is under-consistent. Therefore, *MFP* enqueues entries $\{\langle g, 8\rangle, \langle j, 17\rangle, \langle i, 16\rangle\}$ in Step 1. Then, in Step 2, *MFP* runs iteratively. It first extracts $\langle g, 8\rangle$. Since g is over-consistent, d'_g is set to 8, and g is consolidated. By checking all the children of g, k is found to be inconsistent, and thus, the entry $\langle k, 12\rangle$ is enqueued. In addition, $\widehat{rhs(j)}$ is changed to 19, although $\widehat{key(j)}$ is still 17. *MFP* conducts similar processes to the subsequent over-consistent vertices k and i. When the under-consistent vertex j is extracted, $\widehat{d}_j = 17$ and $\widehat{rhs(j)} = 19$, *MFP* sets $\widehat{d}_j$ to ∞, which changes j to over-consistent. Since j had no satisfying-children, no vertices require their rhs values to be re-evaluated. [11] Later, j is extracted again and consolidated with the distance of 19. Figure 6 depicts G' and T'_s in which only vertex j is processed by *MFP* twice. ∎

As illustrated in Figure 6 (b), we observe that the affected vertices, whose new distances are increased, could be processed twice.

6 Complexity Analysis

Here we analyze, for the increase and decrease cases, the complexity of algorithms *DynDijkstra*, *MBallString*, and *MFP*. The complexity model used is the one proposed in [18]. For this purpose, we define a set of metrics that represent the important operations in these algorithms.

Given a graph G, an SPT T_s rooted at vertex $s \in V(G)$, and a set ε of edges, in which either all edges get their weights increased or all edges get their weights decreased, we let A be the set of affected vertices, and $\delta_A = |A|$. More specifically, for algorithms *DynDijkstra* and *MBallString*, A denotes the set of locally-affected vertices in T_s; and for the *MFP*, A denotes the set of dist-affected vertices in G. An affected vertex v is said to be *dist-affected* if $d'_v \neq d_v$; otherwise, v is *dist-not-affected*.

We let $\delta_A^{out} = |Out_A|$ be the number of outgoing edges from vertices in A, and $\delta_A^{in} = |In_A|$ be the number of incoming edges to vertices in A.[12] We let δ_m be the number of modified edges, and δ_m^{in} be the number of incoming edges to all heads of modified edges. Finally, we let δ_x be the number of branches or subtrees processed by *MBallString*. We consider δ_x to be much less than δ_A although they could possibly be the same.

[11] Vertex f is i's satisfying-parent; i is n's satisfying-parent; k is o's satisfying-parent.

[12] In complexity analysis, we do not differentiate between $In_N\, and AllIn_N$, because based on the data structure presented in this paper, they have the same complexity. Similarly for Out_N and $AllOut_N$.

Unit Operation	description
edge visit	access any edge in G
distance update	update shortest distance of vertex
link visit	access shortest path parent/child relation
link update	update shortest path parent/child relation
status update	update vertex's status (open/closed)
enqueue	enqueue a new entry
decrease-key	decrease an existing entry's key
increase-key	increase an existing entry's key
extract-min	extract an existing entry
removal	remove an existing entry

Table 1: Unit operations

Specially, for the *MFP*, we let C be the set of not-dist-affected children of dist-affected vertices in G and δ_C^{in} [13]$= |In_C|$ be the number of incoming edges to vertices C in G.

We consider unit operations as listed in Table 1.[14] Edge weight updates are ignored in complexity analysis; and all operations related to one vertex's shortest path parent modification are counted as one link update.[15] Note that Table 1 gives the maximum set of major operations that could be involved in any algorithm discussed in this paper; not every algorithm requires all these operations.

Queue operations are important for each algorithm. To evaluate algorithms fairly, all of them use the same queue implementation. In this paper, a queue is realized with an *ArrayHeap* (in [3]) since this is the only implementation that supports all five queue operations required by the algorithms studied in this work.

6.1 Edge Weight Increases

In this subsection, we analyze the complexities of *DynDijkInc*, *MBallStringInc*, and *MFP*. We summarize the complexities in Table 2 and prove the complexity of each algorithm individually.

Unit Operation	*DynDijkInc*	*MBallStringInc*	*MFP*
edge visit	$\delta_m + \delta_A^{in} + \delta_A^{out}$	$\delta_m + \delta_A^{in} + \delta_A^{out}$	$\delta_m + \delta_m^{in} + 2 \times (\delta_A^{out} + \delta_A^{in} + \delta_C^{in})$
distance update	$\leq \delta_A + \delta_A^{out}$	δ_A	$2 \times \delta_A$
link visit	$\leq \delta_A$	$\leq 2 \times \delta_A$	0
link update	$\leq \delta_m + \delta_A$	$\leq \delta_m + \delta_x$	$\leq \delta_m + 2 \times \delta_A^{out}$
status update	0	$2 \times \delta_A$	0
enqueue	δ_A	$\leq \delta_A$	$2 \times \delta_A$
decrease-key	$\leq \delta_A^{out}$	$\leq \delta_A^{out}$	$\leq \delta_A^{out}$
increase-key	0	0	0
extract-min	δ_A	δ_x	$2 \times \delta_A$
removal	0	$\leq \delta_A - \delta_x$	0

Table 2: Unit operations of *DynDijkInc*, *MBallStringInc*, and *MFP*

[13]This notation is used in the analysis of edge weight increases case.

[14]*REMOVE* and *EXTRACTMIN* operations in the pseudo-code are the same as removal and extract-min in Table 1. However, *ENQUEUE* is enqueue + decrease-key while *ADJUST* is enqueue + decrease-key+ increase-key.

[15]For example, lines $19-21$ in *DynDijkInc* are counted as one link update; so are lines $21-23$ in *MBallStringInc*.

Lemma 6.1 *After a set of edge weight increases, the worst-case number of unit operations of DynDijkInc is as listed in Table 2.*

Proof In Step 1, δ_m modified edges are visited. All modified edges could be tree edges in T_s and be removed, so, there are at most δ_m link updates. Method *findLocallyAffectedVertices* has to traverse $\widehat{T_s}$ from modified heads. Therefore, according to the tree structure, at most δ_A links are visited.

In Step 2, all incoming edges to locally-affected vertices are examined, so, there are δ_A^{in} edge visits. All locally-affected vertices get the distances updated in line 12, thus, there are δ_A distance updates.

In Step 3, according to Lemma A.5, only locally-affected vertices are processed by *DynDijkInc*. According to lines $18-21$, *DynDijkInc* extracts exactly one locally-affected vertex in each iteration; it also updates one shortest parent link in each iteration according to the extracted vertex. Therefore, there are δ_A enqueues, extractions, and link updates. In addition, all outgoing edges for each locally-affected vertex are visited, and each edge visit might induce a distance update and a decrease-key. Therefore, there are exactly δ_A^{out} edge visits and at most that number of distance updates and decrease-keys. ∎

Next, we analyze the complexity of *MBallStringInc*, in which locally-affected vertices are processed branch by branch. For any locally-affected vertex v, its distance is updated only when the final optimal distance is located.

Lemma 6.2 *After a set of edge weight increases, the worst-case number of unit operations of MBallStringInc is as listed in Table 2.*

Proof Step 1 of *MBallStringInc* contains exactly the same unit operations as Step 1 of *DynDijkInc*; therefore, we skip the analysis.

In Step 2, δ_A^{in} edges are visited. No distance update happens, but all locally-affected vertices get their status updated to *open*. Therefore, there are δ_A status updates. Since a locally-affected vertex can be enqueued at most once, then there are no more than δ_A enqueues.

In Step 3, *MBallStringInc* extracts only mini-roots. Correspondingly, it updates only the shortest path parents of these mini-roots; thus, there are δ_x extractions and link updates. All locally-affected vertices are selected by $des(\widehat{T_s}, y)$ in line 24 once and then get the distance and status updated also exactly once. Therefore, there are at most δ_A link visits, and exactly δ_A distance and status updates. In addition, *MBall-StringInc* examines the outgoing edges of all locally-affected vertices, and each edge visit might induce a decrease-key; thus, there are exactly δ_A^{out} edge visits and at most δ_A^{out} decrease-keys.

Finally, since all entries in Q that are not extracted are removed and there are at most δ_A entries in Q, *MBallStringInc* conducts no more than $\delta_A - \delta_x$ extractions. ∎

Lemma 6.3 *After a set of edge weight increases, the worst-case number of unit operations of MFP is summarized in Table 2.*

Proof In this algorithm, a vertex v is enqueued iff v is dist-affected. This follows from the requirement that the $rhs(v)$ is not equal from d_v before a vertex v is enqueued.[16]

In Step 1, all the modified edges are visited; thus, δ_m edges are visited and the same number of link updates. In addition, all the incoming edges to each modified head h are visited to re-evaluate $rhs(h)$. Therefore, there are δ_m^{in} more edge visits.

In Step 2, each dist-affected vertex y is first extracted to be under-consistent and processed: its distance is first updated to ∞; then all the outgoing edges e of y are visited. However, there will be no increase-/decrease-key or remove operation. Therefore, there are δ_A extractions and distance updates, δ_A^{out} edge visits, and at most δ_A^{out} link updates. Furthermore, in order to obtain $\widehat{rhs(h(e))}$, all the incoming edges of $h(e)$ are visited. If $h(e)$ is affected, then there are δ_A^{in} edge visits overall. If $h(e)$ is unaffected, then there are, in total, δ_C^{in} edge visits.

According to the algorithm, each vertex in A is then extracted to be over-consistent and is finalized. Therefore, there are δ_A extractions and distance updates, δ_A^{out} edge visits, and, at most, that number of link updates and decrease-keys. Furthermore, in order to obtain $\widehat{rhs(h(e))}$, all the incoming edges of $h(e)$ are visited. If $h(e)$ is affected, then there are δ_A^{in} edge visits overall. If $h(e)$ is unaffected, then there are δ_C^{in} edge visits overall.

Due to $2 \times \delta_A$ extractions, *MFP* must conduct at least that number of enqueues. ∎

6.2 Edge Weight Decreases

In this subsection, we analyze the complexity of *DynDijkDec* and *MFP*. We summarize the complexities in Table 3 and prove the complexity of each algorithm individually. In addition, we give informal discussion of *BallStringDec*'s complexity in our metrics.

Unit Operation	*DynDijkDec*	*BallStringDec*	*MFP*
edge visit	$\delta_m + \delta_A^{out}$	$\gg \delta_A^{out}$	$\delta_m + \delta_m^{in} + \delta_A^{out}$
distance update	$\leq \delta_m + \delta_A^{out}$	$\gg \delta_A$	δ_A
link visit	0	$\gg \delta_A$	0
link update	δ_A	$\gg \delta_x$	$\leq \delta_m + \delta_A^{out}$
status update	0	0	0
enqueue	δ_A	$\gg \delta_x$	δ_A
decrease-key	$\leq \delta_A^{out}$	$\gg \delta_x$	$\leq \delta_A^{out}$
increase-key	0	0	0
extract-min	δ_A	$\gg \delta_x$	δ_A
removal	0	$\gg \delta_x$	0

Table 3: Unit operations of *DynDijkDec*, *BallStringDec*, and *MFP*

Lemma 6.4 *After a set of edge weight decreases, the worst-case number of unit operations of DynDijkDec is as listed in Table 3.*

[16]This holds for both the increase and decrease cases.

Proof In Step 1, modified edges are visited, and each edge visit might induce a distance update. Therefore, there are exactly δ_m edge visits and at most δ_m distance updates.

In Step 2, according to Lemma A.6, only locally-affected vertices are processed by *DynDijkDec*. According to line 10, *DynDijkDec* extracts exactly one locally-affected vertex in each iteration, and updates one shortest parent link in each iteration according to the extracted vertex. Thus, there are exactly δ_A enqueues, extractions, and link updates. In addition, the outgoing edges of all locally-affected vertices are visited, and each edge visit might induce a link update and decrease-key. Therefore, there are exactly δ_A^{out} edge visits and at most δ_A link updates and decrease-keys. ∎

The official proof of *BallStringDec*'s complexity can be found in [28], and here we analyze *BallStringDec*'s complexity according to our metrics. As we have discussed in the algorithm description, very likely, *BallStringDec* conducts duplicate distance updates. Unlike *MBallStringInc*, which processes each branch exactly once, *BallStringDec* might process some branches multiple times. Whenever a branch is processed, edge visits, distance updates, enqueues, and removals are induced. Therefore, there is no way to predict the numbers of all these unit operations. In Table 3, we use $\gg f(n)$ to indicate that the number of operations is dependent on $f(n)$ but it cannot be established precisely due to the unpredictable effect of duplicate distance updates.

Lemma 6.5 *After a set of edge weight decreases, the worst-case number of unit operations of MFP is summarized in Table 3.*

Proof The proof of Lemma 6.5 is the same as that of Lemma 6.3, except for Step 2.

In Step 2, all vertices in A are found to be over-consistent and are processed (and finalized). According to the algorithm, each vertex y in A is extracted only once, and the distance of each vertex is updated. All the outgoing edges of y are examined, and each edge that is visited could induce a decrease-key. Thus, there are δ_A extractions and distance updates, δ_A^{out} edge visits, and at most δ_A^{out} number of decrease-keys. ∎

7 Experiments

The main purposes of this section are to detail how the algorithms presented in this work perform, and to identify the best solution for different scenarios. In Section 7.1, we introduce our experimental framework, present the problem instance generators, describe the performance indicators, and give some relevant implementation details. In Section 7.2, we present the experimental results.

7.1 Experimental Setup

7.1.1 System Environment and Data Sets

Our experiments are performed on a personal computer with a Pentium IV 2.56 GHz processor and 1 GB of main memory, running Microsoft Windows XP Professional Version 2002. We use Java 1.4.2 to implement all programs. To make a homogeneous execution environment for every test case, we set the Java Virtual Machine (JVM) to 1 GB.

We use two types of graph data: a real-life data set and an artificial data set. The former one is from the Connecticut road system extracted from the U.S. Census Bureau Tiger/Line files [1], denoted as *road system graphs* in short. Road system graphs have 5 different sizes: $1K$, $2K$, $4K$, $8K$, and $15K$. The size is determined by the number of vertices inside the graph. For each size, 2 graphs are extracted from Connecticut road system. Each graph F is originally undirected, so we construct a directed graph G by replacing an undirected edge in F with two directed edges as follows: $\forall v \in V(F)$, v is added to G; $\forall (u, v) \in E(F)$, (u, v) is added to G, and a new edge (v, u) with a random weight is also added to G, such that (u, v) is u's outgoing and v's incoming edge, and (v, u) is v's outgoing and u's incoming edge. The weight of an edge (u, v) is the length of the edge in F while the weight of the newly added edge (v, u) is chosen arbitrarily from the weights in the original graph. The weights, for instance, could denote the time needed to travel over a street block in a road system graph.

Due to the nature of the road system and the way we construct a directed graph, the directed graphs G's are relatively sparse such that $|E(G)| \leq 3 \times |V(G)|$. Moreover, they are all strongly-connected, such that there exists a path between any pair of vertices in G. The statistics are given in Table 4.

Graph size	No. of vertices	No. of edges
$1K$	1194	2970
$1K$	1181	2798
$2K$	2280	5364
$2K$	2034	4784
$4K$	4320	9826
$4K$	4165	9674
$8K$	8350	20474
$8K$	8146	20396
$15K$	15001	38346
$15K$	15002	36814

Table 4: Road System Graphs Statistics.

Graph Size	No. of Vertices	No. of Edges
100	100	4950
200	200	17300
400	400	61400
800	800	220400

Table 5: Artificial Random Graphs Statistics

The other type of graphs is *artificially generated*. With the random graph generator from [2], we generate directed graphs, given the number of vertices, the number of edges, and a certain range of edge weights. The generator assigns edges to vertices such that the outgoing degrees of vertices follow quasi-power law distribution. The weight of an edge is randomly selected from the input range of 1 to 1,000,000. According to this random generator, $|E(G)|_{max} = \frac{|V(G)| \times (|V(G)|-1)}{2}$. Therefore, we are able to generate random graphs much denser than road system graphs. The data on random graphs generated are shown in Table 5.

7.1.2 Problem Instances

In each testing graph G, we randomly select a vertex s as the source, and a set ε of edges whose weights are to be increased or decreased. We denote ε as *non-mixed* if it contains either increased edges or decreased edges, but not both; we denote ε as *mixed* if it contains both. Given an outdated SPT rooted at s in G, we run SPT algorithms to update ε accordingly and compute a new SPT. Note that the outdated SPT is already residing in the main memory before an algorithm starts executing.

In a non-mixed case, we vary the percentage of changed edges and the percentage of weight changed; in a mixed case, we vary the percentage of changed edges and the percentage of increased edges. More details follow in Section 7.1.4.

7.1.3 Performance Indicator

In this work, we are interested in the total number of operations performed and the CPU running time for each solution. The types of operations interested are listed in Table 1.

In the mixed cases, the semi-dynamic algorithms need to divide ε into ε^+ and ε^- so that ε^+ contains all edges in ε whose weights are increased, and ε^- contains the rest. The CPU time for dividing ε into ε^+ and ε^- is also counted as part of the cost. In addition, when we run semi-dynamic algorithms for mixed cases, we first run the decreases routine for ε^-, and then the increases routine for ε^+. This order, however, could be arbitrary.

7.1.4 Factors Evaluated

Since different algorithms may have different properties, in order to draw a meaningful conclusion, we examine which algorithm works the best in different scenarios. We extract some factors from the general situations. Table 6 lists these factors and their sample values used in the experiments.

Factor	Samples for Road System Graphs	Samples for Random Graphs
graphsize (increase and decrease cases)	$1K$, $2K$, $4K$, $8K$, $15K$	100, 200, 400, 800
(mixed case)	$2K$, $8K$, $15K$	200, 400, 800
pce (increase and decrease cases) %	0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10	0.1, 0.5, 2, 10
(mixed cases) %	0.5, 2, 5	0.5, 1, 2, 5
pcw (increase cases) %	100, 200, 1000, 2000, 5000, 10000	100, 200, 1000, 2000, 5000, 10000
(decrease cases) %	5, 10, 20, 40, 60, 90	5, 10, 20, 40, 60, 90
pie(*mixed cases only*) %	10, 30, 50, 70, 90	10, 30, 50, 70, 90

Table 6: Samples of evaluated factors in the *DSP* problem

Graph Size (*graphsize*) It is the number of vertices in it. The samples are listed in Table 6.

Percentage of Changed Edges (*pce*) It is the percentage of changed edges. The samples are listed in Table 6. The changed edges are randomly selected from the graph. For example, in a $4K$ sized road system graph which has approximately $8K$ edges, when *pce* is 1%, 80 edges get their weights updated.

Percentage of Changed Weight (*pcw*) It is the percentage of the changed edge's weight that will be added to or deducted from the its original weight. There are two groups of samples: the edge weight increases and the edge weight decreases. For the mixed case, percentage of increased edges (*pie*) is used instead. Table 6 provides the samples.

For example, if a person is driving at 100 kilometers per hour (100km/h) on a highway, it takes him 30 seconds (30s) to travel from one intersection to the next one. If, due to a traffic congestion, he/she slows down to 10km/h, he/she needs 300s to drive the same distance. In the graph presentation, the corresponding edge's weight is increased by nine times from 30s to 300s. For the opposite situation, the edge's weight is decreased by 90% from 300s to 30s.

Percentage of Increased Edges (*pie*) In the mixed cases, after randomly selecting a group of modified edges, we vary the ratio between the number of the increased edges and the number of the decreased edges in this group. The samples are given in Table 6. For instance, the value 10 stands for that 10% of modified edges have their weights increased while 90% have their weights decreased.[17]

For all cases, given a group of sample values, a *run* consists of 150 ($2 \times 3 \times 25$) SPT computations. For example, for the road system graphs, let $graphsize = 4K$, $pce = 0.1$, and $pcw = 100$ be a group of sample values. In a run, 2 graphs (both are $4K$ in size) are selected. For each graph, we randomly select 3 groups of edges, each of which contains 4 edges ($pce = 0.1$) whose weights are increased by 100%, and we randomly select 25 vertices as sources whose SPTs are computed. Thus, in a run, we have 150 SPT computations. For each algorithm and for each group of sample values, the average number of unit operations and the average execution time of a run are used as the average data, and they are the y-values in our plots.

7.1.5 Implementation Details

In this subsection, we take a closer look at the data structures used in the implementation. There are a few important data structures that are shared by the algorithms: Graph G; SPT T_s, rooted at vertex s; and priority queue Q.

Conceptually, G contains a vertex set V and an edge set E. Each vertex v is identified by a key (the ID of v), and so is each edge e. T_s is denoted by the vertices' auxiliary information set. In this set, a vertex is identified by its ID. Each vertex has an auxiliary information, aux, which contains $spp(v)$, $spc(v)$, d_v, and $status(v)$. We use Java 1.4.2's `HashMap` to implement the containers of V, E, and T_s. During the execution period in which we observe the performance, our algorithms update the auxiliary information of the vertices in T_s. As pointed out before, a priority queue is implemented with the `ArrayHeap` in JDSL [3].

Besides implementing our algorithms in this work, we also implement *Dijkstra* as a reference. To obtain a fair comparison, we modify *Dijkstra* to take a group of modified edges as its input, as all the other incremental algorithms do, to modify these edges' weights, and to compute a new SPT for the updated graph.

[17] The weight changes, in this case, are randomly set.

7.2 Experimental Results

In Section 7.2.1, we show how *pcw* affects the algorithms investigated. Then in the rest of Section 7.2, we focus on the other factors and see how they influence the performance of an algorithm.

7.2.1 Factor *pcw*

Figure 7 shows, for the increases case, the effect of *pcw* on various algorithms with road system graphs.[18] Results on other road system graphs and random graphs are similar, and therefore are not included here.[19] The plot on the left shows the execution time while the one on the right records the total number of operations performed by an algorithm. From the figure, it is observed that, for all algorithms, both the unit operations and the CPU time remain relatively constant, regardless of the changed weights. The reason for this phenomenon is due to the nature of the incremental algorithms. In the increases cases, given a

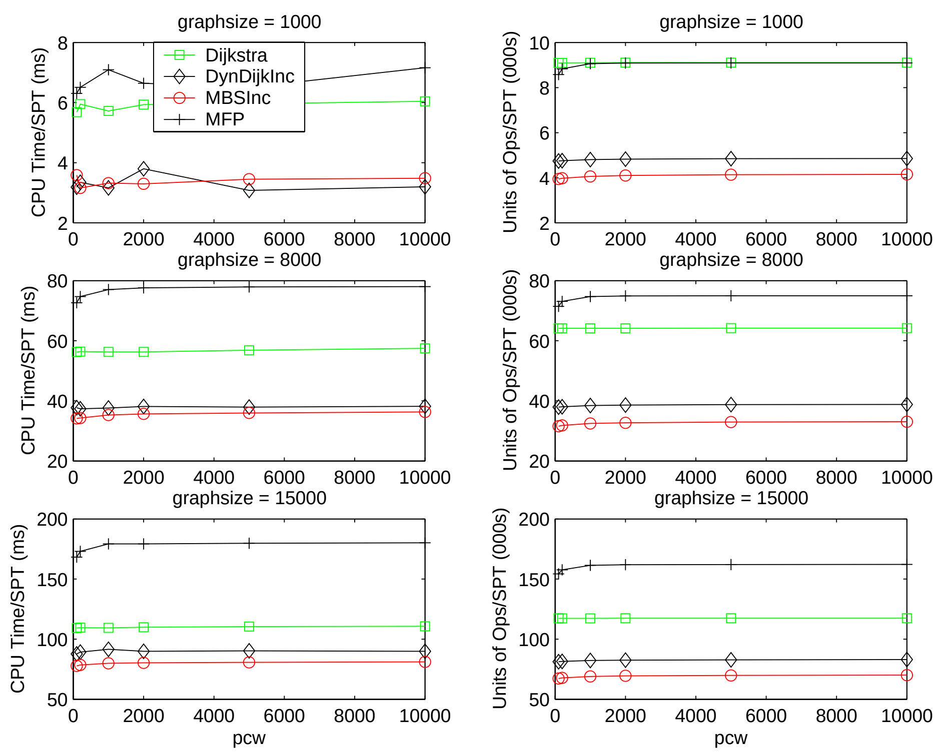

Figure 7: Comparison in Edge Weight Increases on *pcw* with Road System Graphs

graph, an SPT, and a set of changed edges, the set of *locally-affected* vertices in algorithm *DynDijkInc* and *MBallStringInc* remains unchanged, regardless of the weight changes. At the beginning of execution, these algorithms both invoke a function called *findLocallyAffectedVertices*. The set of vertices returned by this function solely depends on the set of changed edges, but not on the increases in their weights. For

[18]The substring *BS* in a legend denotes *BallString*.

[19]From now on, due to space limitation, only a subset of exemplifying plots are presented.

DynDijkInc, the number of iterations is the number of *locally-affected* vertices. For *MBallStringInc*, even though the number of iterations is the number of branches processed, which is much smaller, the amount of work required is proportional to the number of locally-affected vertices. For algorithm *MFP*, similar to the two other algorithms, the number of dist-affected vertices, which need to be processed, remains relatively constant. Therefore, the CPU time and the units of operations remain flat. The performance differences among these algorithms will be explained in subsequent discussion. In summary, for the increases case, *pcw* has little influence on these incremental algorithms.

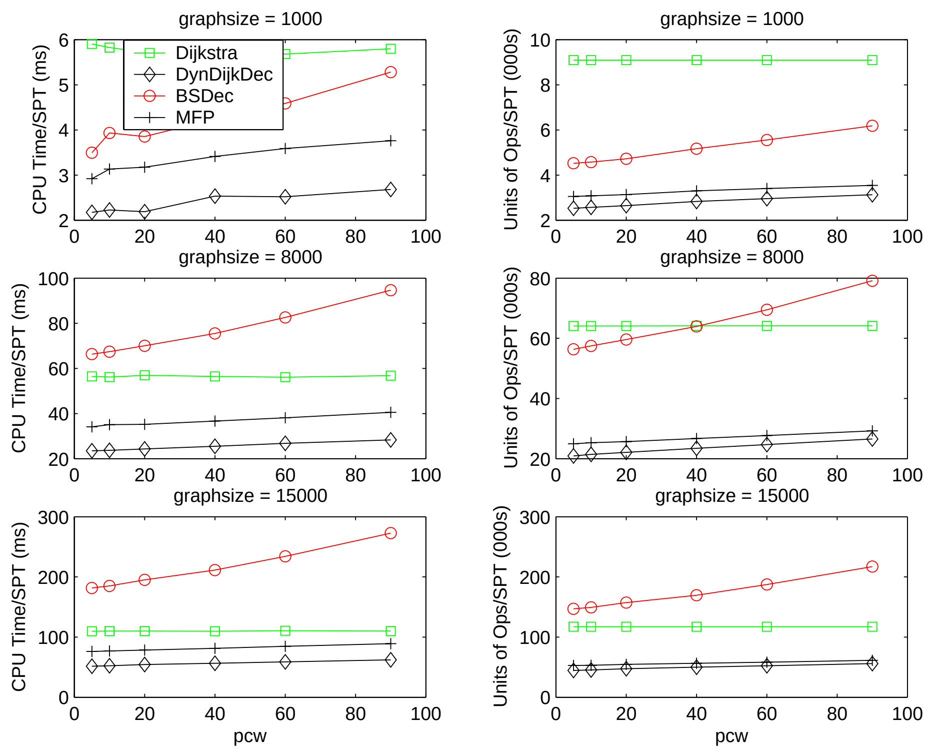

Figure 8: Comparison in Edge Weight Decreases on *pcw* with Road System Graphs

On the other hand, *pcw* has a more noticeable influence in the decreases case. Figures 8 and 9 show how the weight decreases affect the performance of these algorithms. Contrary to the increases case, the set of locally-affected vertices cannot be determined initially. As a result, the more the edges' weights are decreased, the more likely a vertex is locally-affected. The larger the number of locally-affected vertices, the more processing is required. In addition, this effect is amplified in the random case due to a larger number of edge visits.

It is worth noting that algorithm *Dijkstra* performs significantly better, relatively to the incremental algorithms, in road system graphs than in random graphs. This is due to the nature of the data sets and the sample points chosen, as well as the characteristics of these algorithms. Because the density of tree edges is much lower in random graphs than in road system graphs, the same *pce* results in a smaller affected subgraph when it is a random graph than it is a road system graph. For instance, consider the edge weight increases case and a *pce* value, 2%. In random graphs case, 11% vertices in 100-node graphs and 25% vertices in 800-node graphs are locally-affected. In contrast, in road system graphs case, 55% vertices in 1K-node graphs and 75% vertices in 15K-node graphs are locally-affected. Since an incremental algorithm processes locally-affected vertices only, it performs better in road system graphs than in random graphs, in general. Another reason for *Dijkstra*'s better performance in road system graph is its complexity. Recall that the complexity of *Dijkstra* is $O(n \times log\ n + m)$, where n and m are the number of vertices and edges in a graph G, respectively. In a random graph, m is dominant.

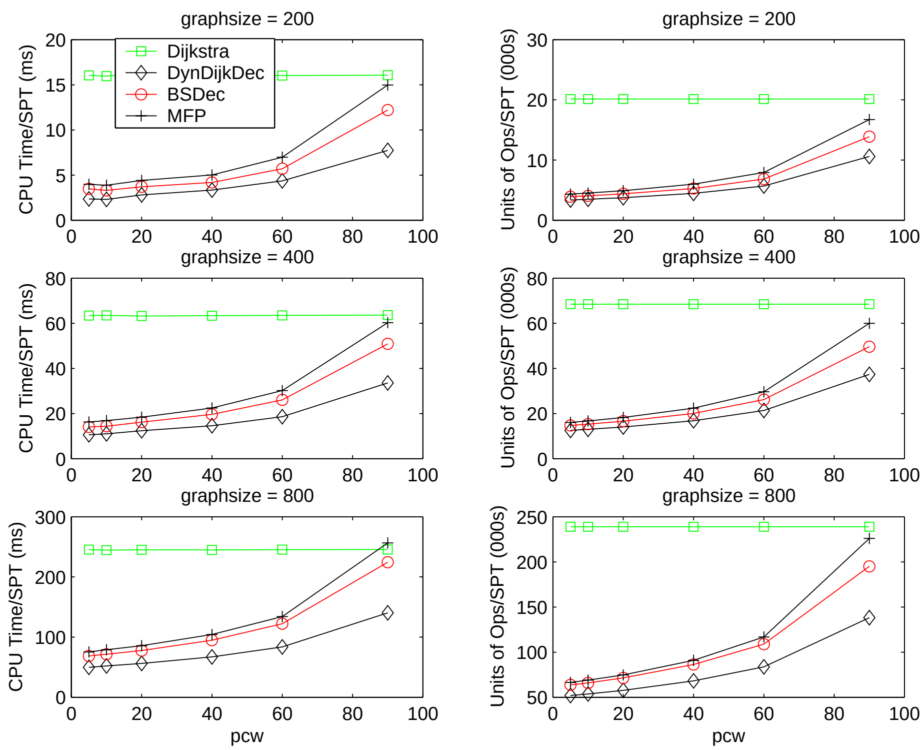

Figure 9: Compariosn in Edge Weight Decreases on *pcw* with Random Graphs

For the rest of the experimental results, we shall focus on other factors and their influences on the performance of various algorithms. We shall present and analyze the results in three parts: the edge weight increases, the edge weight decreases, and the mixed edge weight changes. Let us call a *pce* x the *pce-threshold* of an incremental algorithm I, if for any value $y \geq x$, I no longer, in term of time, outperforms *Dijkstra*.

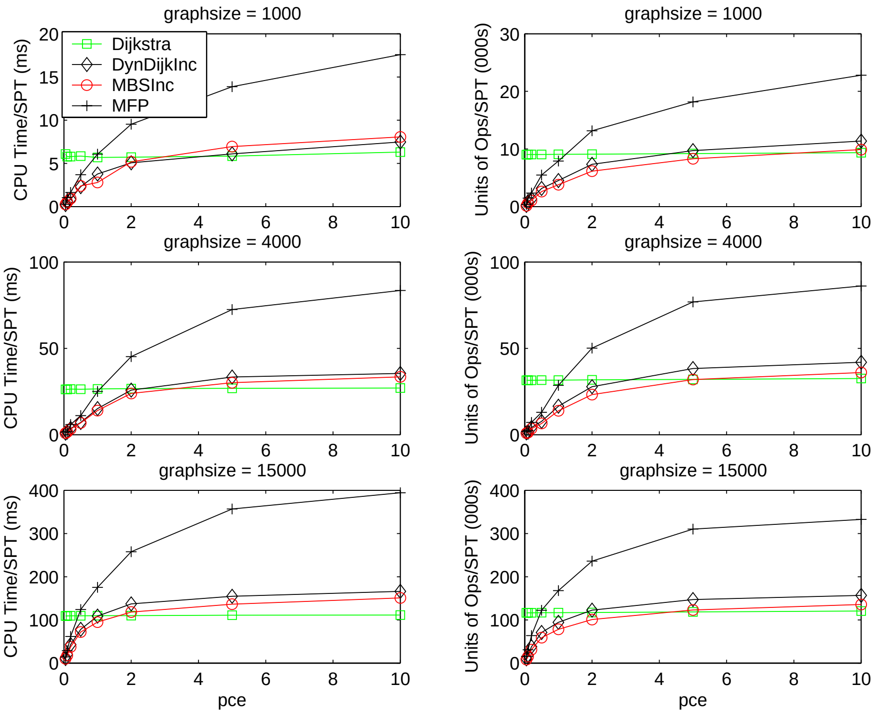

Figure 10: Comparison in Edge Weight Increases on *pce* with Road System Graphs

7.2.2 Edge Weight Increases

For road system graphs, as shown in Figure 10, *graphsize*'s increase lowers the incremental algorithms' *pce*-thresholds. In fact, this holds for all test data sets. In the increases case, all incremental algorithms outperform *Dijkstra* when *pce* is small, say when it is less than 1%. As *pce* increases from zero to some *pce*-threshold, the performance gap between a incremental algorithm and *Dijkstra* narrows. Thus, the advantage of an incremental algorithm over *Dijkstra* reduces as *pce* increases; after some *pce*-threshold is reached, no more advantage exists.

In general, *MBallStringInc* outperforms all other incremental algorithms, in terms of the total number of operations and the CPU execution time, regardless of *graphsize* and *pce*. Although *MBallStringInc* processes the same set of affected vertices as *DynDijkInc* does, *MBallStringInc*'s better performance is due to the branch consolidation by δ. Branch consolidation results in fewer queue operations when compared to *DynDijkInc*. Since *MFP* processes each dist-affected vertex twice, it requires a larger number of edges visits and more queue-related operations. Consequently, it performs the worst among all three incremental algorithms. In sum, if *pce* is less than a certain *pce*-threshold, which depends on *graphsize*, *MBallStringInc* should be applied; otherwise, *Dijkstra* should be applied. According to our tests, the range of the *pce*-threshold for *MBallStringInc* is between 2% to 4%.

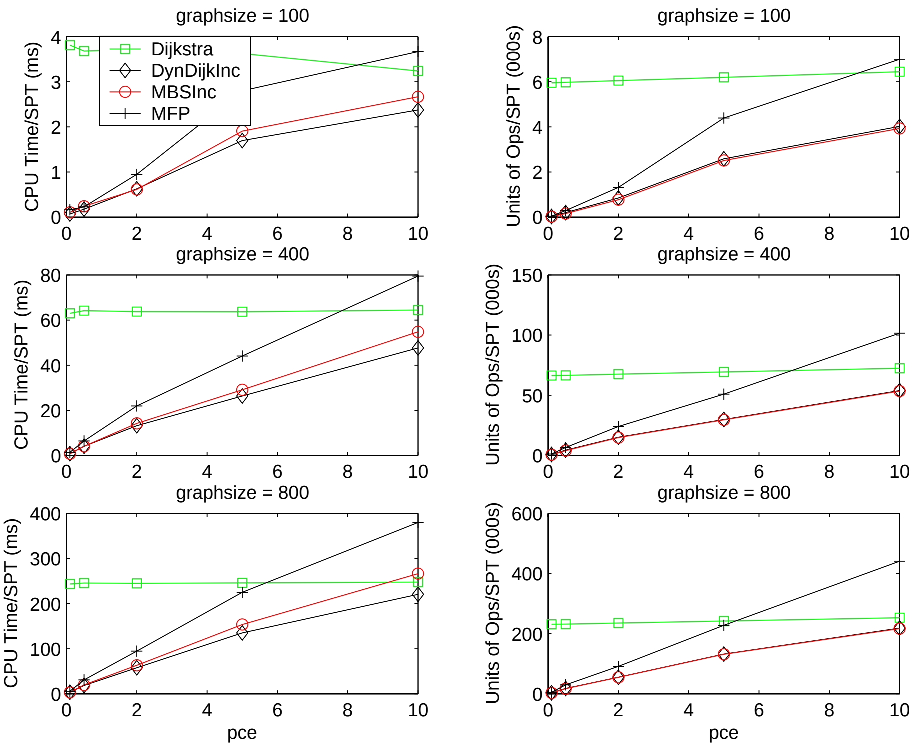

Figure 11: Comparison in Edge Weight Increases on *pce* with Random Graphs

For random graphs, as shown in Figure 11, the relative performance is similar to that in road system graphs, except that *DynDijkInc* performs a bit better in term of CPU time. We observe that *DynDijkInc* and *MBallStringInc* have almost the same number of unit operations. In fact, they both have similar numbers over all major categories of operations. However, *MBallStringInc* is a more complex algorithm than *DynDijkInc*. When an SPT is small, the benefit of branch consolidation of *MBallStringInc* could be out-weighted by its overhead. As a result, *DynDijkInc* has a better time performance than *MBallStringInc*. It is worth noting that the *pce*-thresholds for all incremental algorithms in random graphs are significantly higher than those in road system graphs, due to the reason given at the end of Section 7.2.1.

7.2.3 Edge Weight Decreases

Figures 12 and 13 show the experimental results, in the decreases case, for road system graphs and for random graphs, respectively.

We observe that, for road system graphs, as in the increases case, all incremental algorithms outperform *Dijkstra* when *pce* is small, but under-perform *Dijkstra* once the *pce* passes some *pce*-threshold. *DynDijkDec* outperforms all other incremental algorithms, regardless of *graphsizes* and *pce*. The *pce*-threshold of *DynDijkDec* is noticeably higher than that of *DynDijkInc*. This is due mainly to the far fewer number of operations involving distance update, link visit, and link update in *DynDijkDec*. *BallStringDec* performs significantly worse than other incremental algorithms due to the duplicate distance updates. The performance deteriorates rapidly as *pce* increases, because more and more subtrees in an SPT are repeatedly processed by *BallStringDec*. On the contrary, *MFP* performs better, compare to the increases case, due to that each affected vertex is enqueued only once in the decreases case, which results in far fewer edge visits and queue-related operations. In sum, for road system graphs, if *pce* is less than a certain threshold, *DynDijkDec* should be applied; otherwise, *Dijkstra* should be employed. The range of the *pce*-thresholds is above 10%, depending on the size of a graph.

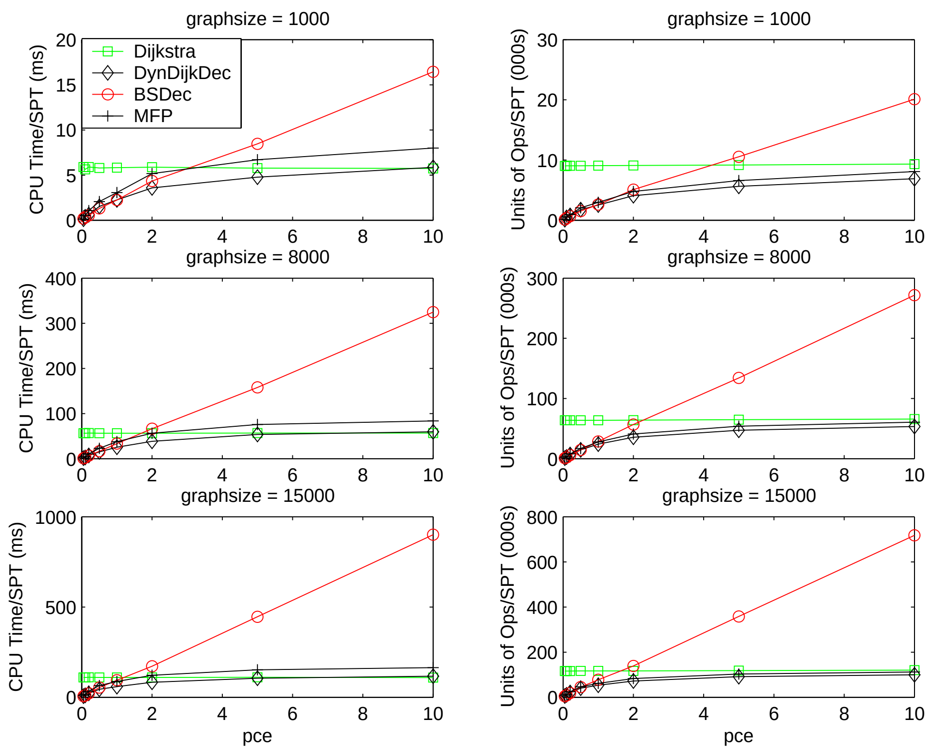

Figure 12: Comparison in Edge Weight Decreases on *pce* with Road System Graphs

For random graphs, as shown in Figure 13, *DynDijkDec* still has the best overall performance among all incremental algorithms while *MFP* performs the worst. Although each vertex is enqueued and extracted once, the number of adjust key operations by *MFP* is much larger than that of the two other incremental algorithms, resulting in *MFP*'s deteriorating performance. In general, an SPT in a random graph is much smaller than that in a road system graph. The chance of duplicate distance update is smaller in a small SPT than in a large SPT. This explains why *BallStringDec* does not perform as bad as in the road system case.

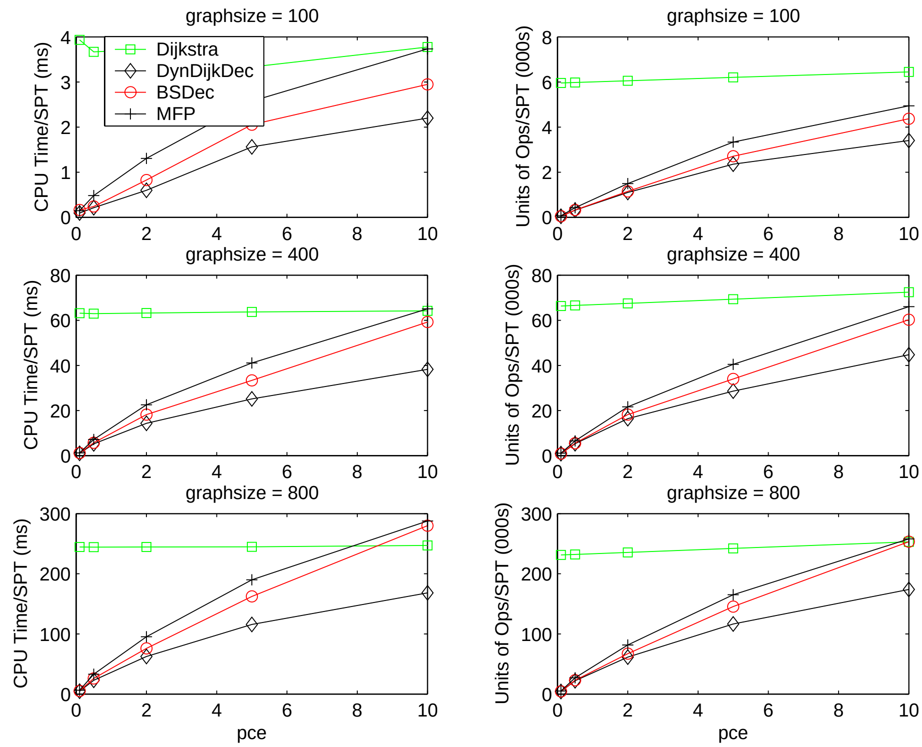

Figure 13: Comparison in Edge Weight Decreases on *pce* with Random Graphs

7.2.4 Mixed Edge Weight Changes

All semi-dynamic algorithms such as *MBallString* and *DynDijkstra* can be used to process a mixed edge weight changes. The mixed edge weight changes are first divided into two sets: increase and decrease. They are then processed by the corresponding semi-dynamic routines. In the previous subsections, we have shown experimentally that the overall best performed semi-dynamic algorithms for increases and decreases cases are *MBallStringInc* and *DynDijkDec*, respectively. We construct an algorithm for the mixed case, which we call *MBSDD*, by combining these two semi-dynamic algorithms together. Algorithm *MBSDD* is also included in our evaluation.

Here, we provide the experimental results in the case of the mixed edge weight changes. Similar to what is reported in the previous two sections, *graphsize* and *pce* affect the performance of all the algorithms in the same manner, and, as can be expected, combining the edge weight increases and the edge weight decreases does not reverse the trend. Consequently, in this part, we choose less samples, and focus on testing *pie*. The *graphsize* chosen are $2K$, $8K$, and $15K$ while the *pce* examined are 0.5%, 2%, and 5%. Our tested samples cover almost the full range of all the possible values for *pie*, *i.e.,* from 10% to 90%.

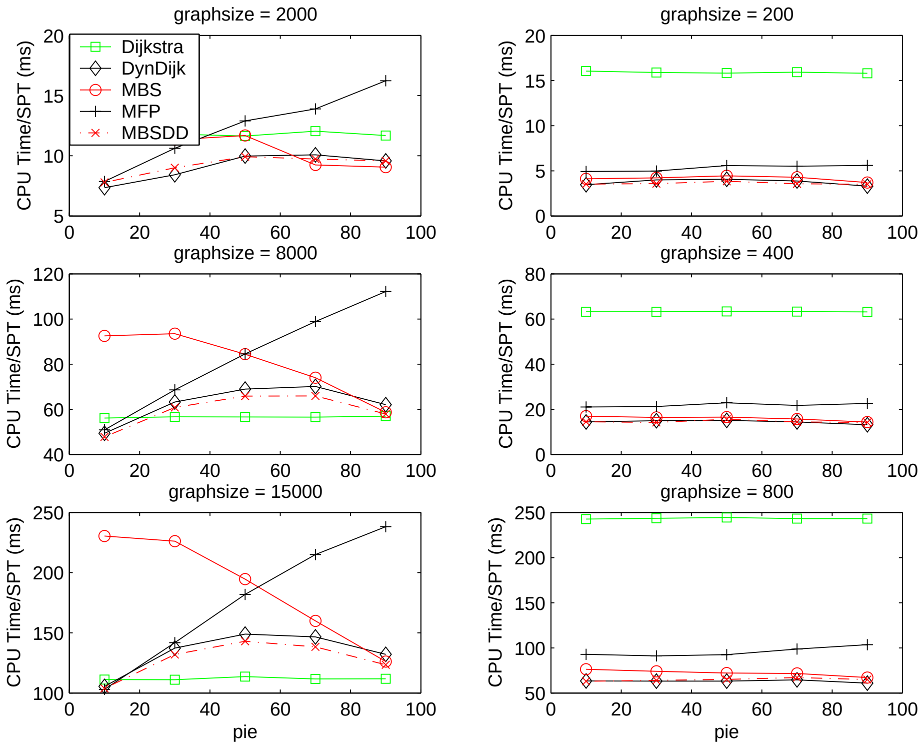

Figure 14: Time Comparison in Mixed Edge Weight Changes on *pie* with Road System Graphs (left) and Random Graphs (right)

Figure 14 shows how the mixes of edge weight increases/decreases affect the performance of the algorithms in term of CPU running time. The plots for the unit operations are very similar to the corresponding time plots and therefore are not included here.

Let us first look at the result on the road system graphs. The figure shows that *pie* does not influence the incremental algorithms uniformly. *DynDijkstra* and *MBSDD* have more or less the same trend. For the same set of modified edges, as *pie* increases from 10% to 90%, two algorithms' performance is initially bad, and then gets improved after a certain threshold. This trend can be explained by considering the behavior of *MBSDD*. When the *pie* is very large (90%) or very small (10%), the best semi-dynamic algorithm, *MBallStringInc* or *DynDijkDec* respectively, is invoked to handle the dominating set of edge weight changes. Thus, the performance of *MBSDD* at the two extremes of *pie* will be very close to (but still slightly better than) two best semi-dynamic algorithms respectively. For the rest of the *pie* sample values, *MBSDD* performs certainly better than all other incremental algorithms. For the reasons stated in the increase and decrease cases, as *pie* increases, *MBallString* performs better while *MFP* performs worse.

The result on random graphs is slightly different from road system graphs. We first observe that the lines look relatively flat, but this mainly due to the poor performance of *Dijkstra*. We also observe that *MFP* performs not as good, relative to road system graphs, mainly because of its larger number of queue-related operations.

The test result on *pce* for road system graphs is summarized in Figure 15. As can be observed, and except for 2K graph and small *pce*'s, *MBSDD* performs no worse than all other incremental algorithms. In fact, *MBSDD* performs just slightly better than *DynDijkstra*. Depending on *graphsize* and *pie*, *MBSDD* should be applied, if *pce* is below a certain *pce*-threshold; otherwise, *Dijkstra* should be applied. The threshold range in this case is about between 1.5% and 3.5%.

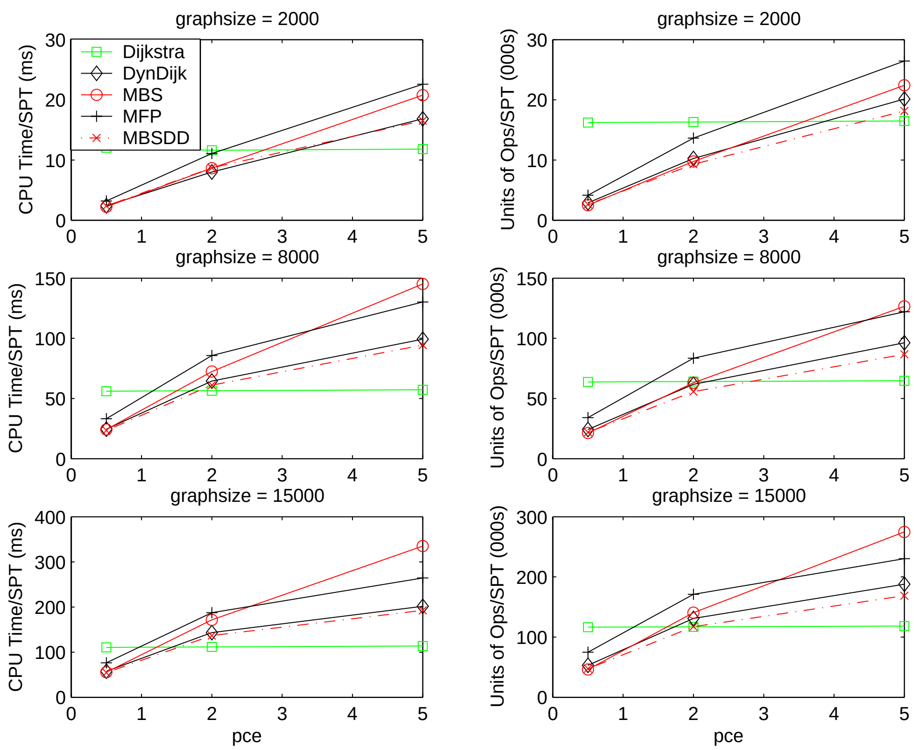

Figure 15: Comparison in Mixed Edge Weight Changes on *pce* with Road System Graphs

For random graphs, the result is summarized in Figures 16. There is not much surprise in this figure. However, unlike the mixed case of road system graphs, *DynDijkstra* edges out *MBSDD* in almost every case. The reason is that *MBallStringInc* does not perform as well as *DynDijkInc* in the random increases case. Consequently, the combined algorithm *MBSDD* is not as good as *DynDijkstra*.

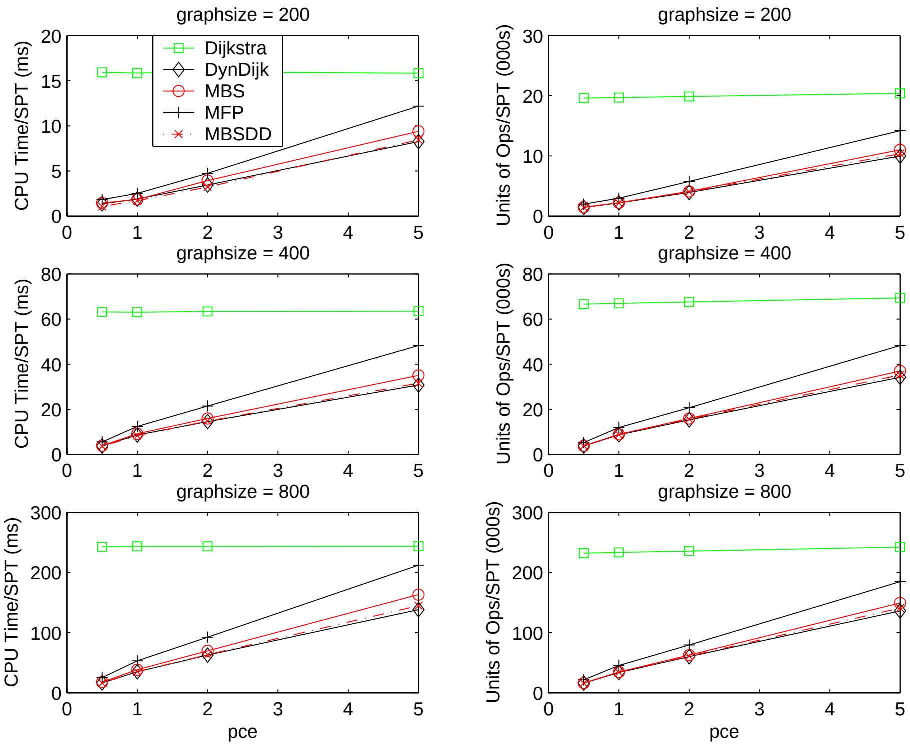

Figure 16: Comparison in Mixed Edge Weight Changes on *pce* with Random Graphs.

8 Conclusion

For the *DSP* problem, we reviewed the previous investigations and discovered that many of them either process a single edge weight update or fail to correctly process the multiple edge weight updates. Therefore, we proposed a few semi-dynamic algorithms by correcting, extending, and optimizing some of the previously studied algorithms. More specifically, *DynDijkstra* and *MBallString* are two semi-dynamic SPT algorithms, whereas *MFP* is a fully-dynamic SPT algorithm. In addition, we derived two frameworks for describing the modified semi-dynamic algorithms: one for increases case and the other for decreases case. We analyzed the complexity and proved the correctness of these algorithms. We conducted experiments to evaluate their performance, both in terms of CPU execution time and total number of operations. We also compared them with the well-known static algorithm *Dijkstra*. The purpose of this study is to understand how these algorithms behave and to determine the best algorithms for different graph sizes and for various mixes of modified edges. We used both real-life and artificial data sets in our experiments. The real-life data sets are road systems in Connecticut and are sparse in nature. The artificial data set are randomly generated graph and are relatively dense. We tried to eliminate the experimental anomalies by conducting a large number of tests. We also identified and evaluated factors that could affect the algorithms' performances.

The factors we investigated in this work are *graph size*, *pce*, and *pcw*. We first showed that, for the increases case, *pcw* has very little effect on the performance of all incremental algorithms studied. However, there are some effects on their performance when the changed-weights are decreases. As expected, incremental algorithms should be used in place of the static *Dijkstra* algorithm when the *pce* is smaller than certain threshold. These thresholds vary on the input mixes and on the graph size. We concluded the following for all incremental algorithms examined in this work. In the increases case, for road system graphs and for random graphs, *MBallStringInc* and *DynDijkInc* have the best overall performance, respectively. In the decreases case, *DynDijkDec* performs the best. For the mixed case, *MBSDD* is the best choice for road system graphs while *DynDijkstra* outperforms others for random graphs.

Acknowledgement

The authors wish to thank the financial support of the Natural Sciences and Engineering Research Council of Canada.

References

[1] Tiger/line files. US Department of Commerce Economics and Statistics Administration, Bureau of Census, 1998.

[2] Java Universal Network/Graph Framework, June 2004.

[3] The Data Structures Library in Java, June 2004.

[4] R. Agrawal and H. V. Jagadish. Algorithms for Searching Massive Graphs. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):225–238, 1994.

[5] D. Alberts, G. Cattaneo, and G. F. Italiano. An Empirical Study of Dynamic Graph Algorithms. *ACM Journal of Experimental Algorithms*, 2:5, 1997.

[6] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–638, 1991.

[7] S. Baswana, R. Hariharan, and S. Sen. Improved Decremental Algorithms for Maintaining Transitive Closure and All-pairs Shortest Paths. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 117–123. ACM Press, 2002.

[8] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier North-Holland, 1976.

[9] L.S. Buriol, M.G.C. Resende, and M. Thorup. Speeding Up Dynamic Shortest Path Algorithms. *AT&T Labs Research, TR TD-5RJ8B*, September 2003.

[10] E. P. F. Chan and N. Zhang. Finding Shortest Paths in Large Network Systems. In *Proceedings of the ninth ACM international symposium on Advances in geographic information systems*, pages 160–166. ACM Press, 2001.

[11] C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 369–378. Society for Industrial and Applied Mathematics, 2004.

[12] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study. In *Proceedings of the 4th International Workshop on Algorithm Engineering*, pages 218–229. Springer-Verlag, 2001.

[13] C. Demetrescu and G. F. Italiano. Fully Dynamic All Pairs Shortest Paths with Real Edge Weights. In *IEEE Symposium on Foundations of Computer Science*, pages 260–267, 2001.

[14] C. Demetrescu and G.F. Italiano. A New Approach to Dynamic All Pairs Shortest Paths. In *Proceedings of the 35th ACM STOC*, pages 159–166, 2003.

[15] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerical Mathematics*, 1:269–271, 1959.

[16] J. Fakcharoemphol and S. Rao. Planar graphs, Negative Weight Edges, Shortest Paths, and Near Linear Time. In *42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01)*, pages 232–241, Las Vegas, Nevada, 2001.

[17] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasquale. Experimental Analysis of Dynamic Algorithms for the Single-Source Shortest-Path Problem. *ACM Journal of Experimental Algorithms*, 3:5, 1998.

[18] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semidynamic algorithms for maintaining single-source shortest path trees. *Algorithmica*, 22(3):250–274, November 1998.

[19] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic Algorithms for Maintaining Shortest Paths Trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[20] D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Shaefer, and C. D. Zaroliagis. An Experimental Study of Dynamic Algorithms for Directed Graphs. In *ESA*, pages 320–331, 1998.

[21] David Hutchinson, Anil Maheshwari, and Norbert Zeh. An External Memory Data Structure for Shortest Path Queries (Extended Abstract). *Lecture Notes in Computer Science*, 1627, 1999.

[22] R. Iyer, D. Karger, H. Rahul, and M. Thorup. An Experimental Study of Poly-logarithmic Fully-dynamic Connectivity Algorithms, 2000.

[23] N. Jing, Y.W. Huang, and E. A. Rundensteiner. Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):409–432, 1998.

[24] V. King. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *IEEE Symposium on Foundations of Computer Science*, pages 81–91, 1999.

[25] P. Klein, S. Rao, M. Rauch, and S. Subramanian. Faster Shortest-path Algorithms for Planar Graphs. pages 27–37, 1994.

[26] P. Narváez. Routing Reconfiguration in IP Networks. *Doctoral Thesis, MIT*, May 2000.

[27] P. Narváez, K. Siu, and H. Tzeng. New Dynamic Algorithms for Shortest Path Tree Computation. *ACM Transactions on Networking*, 8(6):734–746, 2000.

[28] P. Narváez, K. Siu, and H. Tzeng. New Dynamic SPT Algorithm Based on a Ball-and-String Model. *ACM Transactions on Networking*, 9(6):706–718, 2001.

[29] S. Nguyen, S. Pallottino, and M. G. Scutellá. A New Dual Algorithm for Shortest Path Reoptimization. *Transportation and Nework Analysis: Current Trends*, pages 221–235, 2002. Kluwer.

[30] G. Ramalingam. *Bounded Incremental Computation*. Number 1089. Lecture Notes in Computer Science, Springer-Verlag, 1996.

[31] G. Ramalingam and T. W. Reps. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[32] G. Ramalingam and T. W. Reps. On the Computational Complexity of Dynamic Graph Problems. *Theoretical Computer Science*, 158(1-2):233–277, 1996.

[33] S. Shekhar, A. Fetterer, and B. Goyal. Materialization Trade-Offs in Hierarchical Shortest Path Algorithms. In *ACM International Symposium on Large Spatial Databases*, pages 94–111, 1997.

[34] R. E. Tarjan. Sensitivity Analysis of Minimum Spanning Trees and Shortest Path Trees. *Information Processing Letters*, 14(1):30–33, March 1982.

[35] B. Xiao, Q. Zhuge, and E. H.-M. Sha. Efficient Algorithms for Dynamic Update of Shortest Path Tree in Networking. *Journal for Computers and Their Applications*, 11(1), 2003.

[36] U. Zwick. All Pairs Shortest Paths in Weighted Directed Graphs Exact and Almost Exact Algorithms. In *IEEE Symposium on Foundations of Computer Science*, pages 310–319, 1998.

Appendix

A Correctness Proofs of *DynDijkstra* and *MBallStringInc*

We now prove the correctness of *DynDijkstra* and *MBallStringInc*. A simple observation of the two algorithms for the increases case.

Proposition A.1 *Let $\bar{N}$ be the set of vertices returned in line 10 of either DynDikjstraInc or MBallString-Inc. If $\bar{N}$ is non-empty, then there is some vertex $a \in \bar{N}$ that satisfies the condition $(b, a) \in In_a$ and $b \notin \bar{N}$. That is, for both algorithms, there is some boundary vertex enqueued in Q before Step 3 begins execution.*

Proof By Lemma 4.1, $\bar{N}$ is $V(T_s) - des(\widehat{T_s},\ s)$, where $\widehat{T_s}$ is the forest obtained from T_s by removing the modified tree edges. Search the tree T_s, starting with the root s, until a vertex b in $des(\widehat{T_s},\ s)$, but a is not in $des(\widehat{T_s},\ s)$ and $(b,\ a)$ is an edge in G. Such an edge $(b,\ a)$ guarantees to exist since $\bar{N}$ is non-empty, all vertices in $\bar{N}$ are reachable from s in T_s, and $s \notin \bar{N}$. ∎

A.1 *DynDijkstra*

In this part, we prove the correctness of *DynDijkInc* and *DynDijkDec*. Since major part in *DynDijkInc* (Step 3) is the same as that in *DynDijkDec* (Step 2), we are going to prove the correctness of these two algorithms together, and we use *DynDijkstra* when no need to differentiate them.

We prove that *DynDijkstra* computes a correct new SPT T'_s eventually. First we argue that *DynDijkstra* correctly updates only the shortest distances/paths of locally-affected vertices in T_s. Thereafter, we prove that T'_s is a valid new SPT. We break the proof into two cases: one without iteration and the other with iterations during the execution of the algorithm.

Lemma A.2 *If DynDijkstra executes 0 iteration, then $\forall v \in V(T'_s)$, $\widehat{d_v}$ is optimal.*

Proof If *DynDijkInc* executes 0 iteration, no entry is enqueued in Step 2. If $\bar{N}$ is empty at the end of Step 1, then none of modified edges is a tree edge in T_s, thus all vertices are unaffected. Therefore, all vertices keep their optimal distances. If $\bar{N}$ is non-empty, by Proposition A.1, the queue is non-empty and the algorithm will execute at least once.

If *DynDijkDec* executes 0 iteration, no entries are enqueued in Step 1. It means that there are no affected-heads. Since no vertex gets a shorter distance after the updates, all vertices must be unaffected and keep their optimal distances. ∎

Suppose *DynDijkstra* executes k iterations, where $k \geq 1$, and let $y_1, y_2, \ldots, y_k$ be the sequence of vertices extracted from the priority queue Q and are processed in line 18 in *DynDijkInc* or in line 10 in *DynDijkDec*.

Let $\widehat{d_{y_i}}$ be the distance assigned by *DynDijkstra* when y_i is extracted, where $1 \le i \le k$. We prove by induction that $\widehat{d_{y_i}}$ is optimal. First, we shall prove a few preliminary Lemmas and Theorems.

Lemma A.3 *The sequence of $\widehat{d_{y_i}}$, where $1 \le i \le k$, is in a non-decreasing order.*

Proof Suppose in any two consecutive iterations, i and $i+1$, where $i > 0$, y_i and y_{i+1} are the extracted vertices, respectively. To prove Lemma A.3, we want to show that $\widehat{d_{y_i}} \le \widehat{d_{y_{i+1}}}$.

In the i^{th} iteration, vertex y_i is extracted in line 18 in *DynDijkInc* or in line 10 in *DynDijkDec*, which means $\widehat{d_{y_i}}$ is less than or equal to any entry currently enqueued. Then, in line 26 of *DynDijkInc* or line 18 of *DynDijkDec*, a new entry q with distance $\widehat{d_q}$ is enqueued. According to the previous line in *DynDijkstra* and the non-negative edge weight, $\widehat{d_q} \ge \widehat{d_{y_i}}$. Therefore, all entries existing in the priority queue after q is enqueued have a distance of no less than $\widehat{d_{y_i}}$ and so is the entry with the minimum distance, that is going to be extracted in $i+1^{th}$ iteration. In other words, $\widehat{d_{y_i}} \le \widehat{d_{y_{i+1}}}$. Thus, the sequence of extracted entries from the queue is in a non-decreasing order. ∎

Lemma A.4 *In the sequence of $y_1, y_2, \ldots, y_k$, $\forall i \ne j$, where $k \ge i, j \ge 1$, $y_i \ne y_j$.*

Proof It suffices to prove that, any extracted vertex v in some iteration will not be enqueued again in later iterations. Let $\widehat{d_v}$ be the distance of v when it is extracted in some iteration. In any later iteration, inequality $\widehat{d_y} + w(e)' < \widehat{d_q}$ in line 24 of *DynDijkInc* and in line 16 of *DynDijkDec* will not hold, where $\widehat{d_q}$ is $\widehat{d_v}$ in our context. The reason is $w(e)' \ge 0$, and, according to Lemma A.3, $\widehat{d_y} \ge \widehat{d_v}$. Therefore, v will not be enqueued anymore. Thus Lemma A.4 holds. ∎

Now we prove that *DynDijkstra* processes a vertex v in some iteration if and only if v is locally-affected.

Lemma A.5 *Given a graph G, an SPT T_s, and input edge increases ε^+, DynDijkInc processes v if and only if v is locally-affected.*

Proof By Lemma 4.1, it suffices to show that the set of processed vertices is exactly the set of vertices $\bar{N}$ in line 10.

"If" If $v \in \bar{N}$, then either v is a boundary vertex in Step 2 and is enqueued, or its distance is set to ∞. In the former case, v is clearly processed by the algorithm. In the latter case, consider the shortest path SP_v in T_s. Using an argument similar to Proposition A.1, there exists a boundary vertex w such that all ancestor vertices of v, except w, in the sub-path SP_{wv} have their distances set to ∞ in Step 2. Since w is enqueued before Step 3 starts executing, together with lines 22-28, all vertices in the sub-path SP_{wv} will eventually be enqueued in line 26 due to the condition in line 24. Thus v will eventually be processed.

"Only if" If v is a not-locally-affected vertex, then v is either not reachable from s in G or $v \in des(\widehat{T_s}, s)$, where $\widehat{T_s}$ is the forest obtained from T_s in Step 1 after the set of modified tree edges are removed. If v is not reachable from s, then v will not be processed. If $v \in des(\widehat{T_s}, s)$, then $SP_v = SP'_v$ since no affected-head is in the path SP_v and weight changes are only increases. In this case, vertex v is not in $\bar{N}$, thus it is not enqueued in Step 2. Because of the condition in line 24, it is not enqueued in Step 3 either. Therefore v is not processed by the algorithm. ∎

Lemma A.6 *Given a graph G, an SPT T_s, and input edge changes ε^-, DynDijkDec processes v if and only if v is locally-affected.*

Proof By Lemma 4.2, the set of locally-affected vertices is exactly the set of vertices the distance of which are changed with the update.

"If" If $d'_v < d_v$, then there is an affected-mini-root in the path SP'_v. Step 1 guarantees all affected-mini-roots are enqueued. Lines 14 to 20 guarantee v will eventually be enqueued and processed since v is reachable from an enqueued affected-mini-root.

"Only if" By line 4 and line 16, all enqueued and processed vertices have their distances decreased. Thus all processed vertices are locally-affected. ∎

Theorem A.7 *DynDijkstra terminates after finite k iterations.*

Proof According to Lemmas A.5 and A.6, only locally-affected vertices will be processed by *DynDijkstra*. There are at most $|V(T_s)| - 1$ locally-affected vertices caused by the input modified edges. In other words, the worst case is that all vertices in T_s, except the source, are locally-affected. According to lines 18 in *DynDijkInc* and 10 in *DynDijkDec*, in each iteration exactly one vertex is processed. Based on Lemma A.4, all processed vertices are distinct. Thus, *DynDijkstra* will terminate after a finite k iterations, where $k \leq |V(T_s)| - 1$. ∎

At any instant of *DynDijkstra*'s execution, we say vertex v's distance is *finalized* (or simply v is finalized) if it is not-locally-affected, or if it is locally-affected and has already been consolidated by *DynDijkstra*. In general, let $v \in V$ and $q \in p(v)$; $d_q + w(q, v)$ is denoted as the distance of v *induced* by q. Now we prove that the finalized distances of all locally-affected vertices are optimal.

Theorem A.8 *In DynDijkstra, at the end of each iteration, all consolidated vertices are assigned with optimal distances.*

Proof We prove the theorem by induction on the number i of iterations. We want to prove that, if at the beginning of i^{th} iteration, where $i \geq 1$, the inductive hypothesis holds, then it is also true at the end of i^{th} iteration.

At the beginning of the first iteration, no vertices are consolidated in both *DynDijkInc* and *DynDijkDec*. Thus, the inductive hypothesis holds trivially before the first iteration begins.

We now prove that, if at the beginning of any i^{th} iteration, where $i \geq 1$, the inductive hypothesis holds, then at the end of i^{th} iteration, all vertices consolidated are also given the optimal distances. We want to prove that if y_i is extracted in i^{th} iteration (in line 18 in *DynDijkInc* or line 10 in *DynDijkDec*), then it gets its optimal distance at the end of i^{th} iteration.

Suppose in G', a path P'_{sy_i} has a shorter distance than $\widehat{d_{y_i}}$, which is computed by *DynDijkstra*. Let z be the first vertex along P'_{sy_i} such that z is locally-affected but not consolidated before the current iteration begins, and let x be z's shortest path parent on P'_{sy_i}. Vertex z guarantees to exist since y_i is locally-affected but not consolidated before the current iteration begins. We want to show that, before the iteration begins, z is an enqueued vertex with a candidate distance induced by x.

There are two possible cases for x: Case (1): x is not-locally-affected. In the increase case, z is an enqueued boundary vertex in Step 2 of *DynDijkInc*. In the decrease case, the edge (x, z) must be a modified edge and z is an affected head. Thus z is enqueued with a candidate distance induced by x. Case (2): x is locally-affected. By assumption on z, x is extracted and consolidated in some previous iteration. Therefore x is assigned with its optimal distance before the current iteration begins. Before the relaxation of x, z is either not in the queue, or if it is in the queue, then the induced distance cannot be smaller than the one induced by x. In either case, after the relaxation of x, z is enqueued with a candidate distance induced by x.

Now we are ready to show that $\widehat{d_{y_i}}$ is the optimal distance for y_i. According to our assumption that P'_{sy_i} has a shorter distance, $\widehat{d_z} < \widehat{d_{y_i}}$ stands. However, $\widehat{d_{y_i}}$ is minimum among all enqueued boundary vertices, $\widehat{d_z} \geq \widehat{d_{y_i}}$ must stand. A contradiction. Therefore, $\widehat{d_{y_i}}$ is the optimal distance for y_i.

Thus we can conclude Theorem A.8 is correct. ∎

Lemma A.9 *DynDijkstra maintains tree edges correctly.*

Proof According to Lemmas A.5 and A.6, *DynDijkstra* only processes locally-affected vertices. Since *DynDijkstra* only updates the shortest path parent of processed vertex, the tree edges headed at not-locally-affected vertices remain unchanged. Thus it suffices to prove that, each locally-affected vertex v gets its correct shortest path parent when it is consolidated.

According to lines 14 and 26 in *DynDijkInc*, lines 6 and 18 in *DynDijkDec*, the parent p that induces v's current tentative distance is always enqueued with v. At line 18 in *DynDijkInc* and 10 in *DynDijkDec*, v is extracted with parent p. The next three lines make sure v's shortest path parent is correctly set to p. According to Lemma A.3 and Theorem A.8, p is always consolidated before v, and when v is consolidated, both p and v are with optimal distances. Therefore, the correctness of Lemma A.9 follows. ∎

Corollary A.10 *Let T_s be a valid SPT rooted at vertex s in graph G. The graph G is modified into G' by a set of edge weight increases or decreases. Algorithm DynDijkstra computes a new valid SPT T'_s rooted at s in graph G'.*

Proof According to Lemmas A.2, A.5 and A.6, not-locally-affected vertices keep their optimal distances. For locally-affected vertices, the correctness follows from Lemmas A.5, A.6, Theorems A.7 and A.8. For tree edges, the correctness follows from Lemma A.9. Therefore, we conclude the correctness of Corollary A.10. ∎

A.2 *MBallStringInc*

The main part of *MBallStringInc* contains iterations. Each iteration consolidates a set of locally-affected vertices. We use inductive reasoning to prove that, at the end of k iterations ($k \geq 0$), all vertices, whose statuses are set to *closed*, get their final optimal shortest distances. Again, we break the proof into two cases: one with and the other without iterations.

Lemma A.11 *If MBallStringInc runs no iteration,* $\forall v \in V(T_s')$, $\widehat{d_v}$ *is optimal.*

Proof If *MBallStringInc* runs no iteration, no entry is enqueued in Step 2. If $\bar{N}$ is empty at the end of Step 1, then no modified edges were tree edges in T_s. Thus, all vertices are unaffected, keeping their optimal distances. If $\bar{N}$ is non-empty, by Proposition A.1, the queue is non-empty and the algorithm executes at least once. Therefore, if *MBallStringInc* runs no iteration, all vertices must be unaffected and keep their optimal distances. ∎

Lemma A.12 *In an SPT* T_s *rooted at vertex* s *in graph* G*, let* u *and* v *be two vertices that are not* s*, suppose* $d_u \geq d_v$*, then* $d_{vu} \geq d_u - d_v$*.*[20]

Proof It is trivially proven by triangle inequality. ∎

Lemma A.13 *In any iteration of MBallStringInc, if* δ_{old} *is the* δ *extracted in line 20 and* δ_{new} *is any* δ *enqueued in line 36, then* $\delta_{old} \leq \delta_{new}$*.*

Proof Let e be any edge examined in line 32 such that $y = t(e)$, $x = h(e)$ and $status(x) = open$. According to *MBallStringInc*, the distance of y is updated in line 26, and a *newdist* is computed for x in line 34. To facilitate the present discussion, we denote the shortest distance of y before line 26 as d_y^{old} and after line 26 as d_y^{new}; we also denote the shortest distance of x before line 34 as d_x^{old}.

In T_s of G, we have $d_y + w(y,x) \geq d_x$ (1). According to the algorithm, the shortest distance of an *open* vertex is updated only in line 26, and its status is set to *closed* right after that in line 27. Since both y and x are *open* before line 26 in this iteration, inequality (1) can also be stated as $d_y^{old} + w(y,x) \geq d_x^{old}$ (2). After line 26, the shortest distance of y is set to d_y^{new}. Therefore, according to line 34, $d_y^{new} + w(y,x)' = newdist$ (3). By combining (2) and (3), we obtain $d_y^{new} - d_y^{old} + w(y,x)' - w(y,x) \leq newdist - d_x^{old}$ (4). According to line 26, $d_y^{new} - d_y^{old}$ is actually δ_{old}, and, according to line 35, $newdist - d_x^{old}$ is actually δ_{new}. Therefore, inequality (4) is in fact $\delta_{old} + w(y,x)' - w(y,x) \leq \delta_{new}$ (5). At the same time, since all input edge changes are increases, we have $w(y,x)' - w(y,x) \geq 0$. Thus, (5) turns out to be $\delta_{old} \leq \delta_{new}$. ∎

[20] d_{vu} refers to the shortest distance from v to u in G.

Lemma A.14 *If MBallStringInc runs k iterations, where $k \geq 1$, the extracted δ's follow a non-decreasing order.*

Proof According to line 20 of *MBallStringInc*, the minimum δ in priority queue is extracted in each iteration. Therefore, the non-decreasing order follows from repeated applications of Lemma A.13. ∎

Lemma A.15 *During the i^{th} iteration, $i \geq 1$, in line 24 and at the end of the iteration, the structure $\widehat{T_s}$ is a forest.*

Proof After Step 2 is executed, $\widehat{T_s}$ is a forest since it is obtained from T_s by removing some tree edges. During an iteration of Step 3, the only place that changes the parent-child relationship in a tree in $\widehat{T_s}$ is in lines 21 to 23. It assigns a new parent to a subtree of a tree in the forest. Thus the Lemma follows. ∎

Lemma A.16 *At the end of the i^{th} iteration, $i \geq 1$, and given any open vertex v, $des(\widehat{T_s}, v)$ is a subset of the initially open vertices in Step 2. Moreover, all vertices in $des(\widehat{T_s}, v)$ are open.*

Proof We prove this by induction on i. Before the first iteration, the inductive hypothesis holds simply by the Lemma 4.1. Assume the hypothesis holds before the i^{th} iteration, we want to show that it is true after the i^{th} iteration, where $i \geq 1$.

In the i^{th} iteration, let vertex y_i be extracted in line 20. By lines 12 and 33, y_i is *open*. By the inductive hypothesis, the set N_i=$des(\widehat{T_s}, y_i)$ of vertices returned in line 24 are all *open*. From line 27, all vertices in N_i become *closed*. By lines 21 and 23, the subtree rooted at y_i is connected to a *closed* vertex. At the the end of i^{th} iteration, if there exists an *open* vertex u such that $des(\widehat{T_s}, u)$ contains some *closed* vertex c, then, by the inductive hypothesis, the vertex c must be in N_i. A contradiction to Lemma A.15 that $\widehat{T_s}$ at the end of i^{th} iteration is a set of disjoint trees. Thus, vertices in $des(\widehat{T_s}, u)$ are all *open* at the end of the i^{th} iteration. Since no vertex is set to *open* after Step 2, $des(\widehat{T_s}, u)$ is a subset of the initially *open* vertices. Therefore, the inductive hypothesis holds after the i^{th} iteration. ∎

Lemma A.17 *MBallStringInc processes v if and only if v is locally-affected.*

Proof According to Steps 1 and 2 of *MBallStringInc*, the set of locally-affected vertices is exactly the set of vertices with status set to *open*. During the execution of the algorithm, no vertex has its status set to *open*. Thus it is equivalent to proving that the set of processed vertices is exactly the set of *open* vertices.

"If" We want to show that if a vertex is *open*, then it will be consolidated and *closed* eventually. Consider the set of initially *open* vertices before Step 3 begins. Let v be an *open* vertex. Consider the path SP_v in T_s. Since v is *open*, there is some *open* vertex u in the path SP_v which is a boundary vertex and is enqueued in the priority queue before Step 3 starts executing. Moreover, all vertices in the sub-path SP_{uv} are *open*. We want to show that, after the execution of an iteration, either v is *closed* or there is some ancestor vertex of v in SP_{uv} is enqueued. We prove this by induction on the number of iterations. Because of the existence of u, the inductive hypothesis holds before the execution of the first iteration. Suppose the inductive hypothesis holds before the i^{th} iteration, we want to show that it is true after the execution of the i^{th} iteration, $\forall\ i \geq 1$. Consider the execution of an iteration, there are two cases to be considered:

Case (1): Some enqueued ancestor vertex w of v in the sub-path SP_{uv} is extracted and processed. By lines 24-31, a subtree rooted at w is consolidated and all vertices in the subtree are *closed*. Since v is a descendant of w, v is *processed* and *closed* in the iteration.

Case (2): No vertex in SP_{uv} is processed. In this case, none or some of the vertices in the path SP_{uv} are enqueued during the iteration. In any case, the inductive hypothesis holds trivially after the iteration.

In sum, if the inductive hypothesis holds before i^{th} iteration, then after the execution of i^{th} iteration, either v is *closed* or some ancestor vertex in the sub-path SP_{uv} is enqueued.

Since only *open* vertices are enqueued and at least one *open* vertex is *closed* in an iteration, v will eventually be consolidated and *closed* in an iteration.

"Only if" By Lemma A.16, the set of vertices returned in line 24 are all *open* vertices. Thus only initially *open* vertices are processed and *closed*. Since the set of *open* vertices is the set of locally-affected vertices, all processed and *closed* vertices are locally-affected. ∎

Lemma A.18 *If MBallStringInc runs k iterations, where $k \geq 1$, let N_1, N_2, ..., N_k be the sequence of sets of vertices processed over iterations in line 24, then $\forall i \neq j$, where $1 \leq i, j \leq k$, $N_i \cap N_j = \emptyset$.*

Proof By Lemma A.16, each N_i is a subset of the initial set of *open* vertices. Since once a vertex is set to *closed*, it is not *open* again. By Lemma A.16, the Lemma follows. ∎

Lemma A.19 *MBallStringInc terminates after finite k iterations.*

Proof By Lemma A.17, *MBallStringInc* only processes locally-affected vertices. There are at most $|V(T_s)| - 1$ locally-affected vertices caused by the input modified edges. In other words, the worst case is all vertices in T_s except for the source are locally-affected. According to line 24, in each iteration, at least one *open* vertex is selected into N and is consolidated, and from Lemma A.18, no locally-affected vertex will be processed more than once. Therefore at most $|V(T_s)| - 1$ iterations will be processed. ∎

Now we prove *MBallStringInc* correctly updates the distances of all locally-affected vertices after k iterations, where $k \geq 1$.

Theorem A.20 *If MBallStringInc runs k iterations, where $k \geq 1$, at the end of each iteration, all closed vertices get their optimal distances, and all boundary vertices are enqueued in Q with the candidate distances.*

Proof We want to prove that if at the beginning of i^{th} iteration, where $1 \leq i \leq k$, the inductive hypothesis holds, that is,

(1) all consolidated vertices get their final optimal shortest distances; and

(2) all boundary vertices are enqueued with their candidate distances,

then at the end of i^{th} iteration, the inductive hypothesis also holds.

In Step 1, *MBallStringInc* selects all locally-affected vertices into $\bar{N}$. In Step 2, *MBallStringInc* marks all locally-affected vertices to *open*, therefore *closed* vertices are not-locally-affected vertices, which are with their optimal distances. Also, in Step 2, *MBallStringInc* computes a minimum distance *newdist* for each affected vertex a based on a's parents: if a is a boundary vertex, $newdist < \infty$ must stand, thus a must be enqueued. Therefore, before the first iteration, all *closed* vertices are with their optimal distances, and all boundary vertices are enqueued.

Next, we want to show that the inductive hypothesis holds after the i^{th} iteration, assuming that the hypothesis holds before the iteration. Firstly, we want to prove the distance optimality of the consolidated set of vertices in the i^{th} iteration.

At the beginning of i^{th} iteration, all boundary vertices are enqueued with their candidate distances. Then in line 20, the entry $\langle y, x, \langle \delta, newdist \rangle \rangle$ of boundary vertex y with the least δ is extracted. In lines 21-23, y's shortest path parent is set to x. In line 24, vertices returned by $des(\widehat{T_s}, y)$ are selected into N. In lines 26-27, vertices in N get their shortest distances incremented by δ and get their status set to *closed*. Now we prove that their distances are optimal.

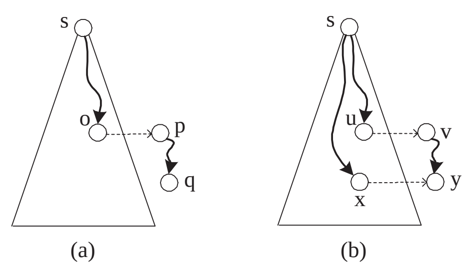

Figure 17: The illustration of possible shortest paths. (a) Any possible shortest path SP'_q in G' is a shortest path SP'_o located so far, concatenated with a boundary edge (o, p) that is again concatenated with a shortest path SP'_{pq}; (b) Suppose y is the vertex with the minimum δ in Q, and its candidate shortest path parent is x, we are going to argue that the shortest path $s \rightharpoonup x \rightharpoonup y$ is not longer than any other possible path $s \rightharpoonup u \rightharpoonup v \rightharpoonup y$.

As shown in Figure 17(a), at any instant of *MBallStringInc*, for any remaining *open* vertex q, the optimal shortest path from s in G' must contain three consecutive parts: the shortest path from s to some vertex o *closed* so far: SP'_{so}; a boundary edge (o, p); the shortest path from p to q in G': SP'_{pq}. Among them, both the first and the third parts may be a single vertex.

In Figure 17(b), y is the vertex to be extracted, which means that, among all *open* vertices directly connected to *closed* vertices, edge (x, y) provides the minimum δ. By the inductive hypothesis, the *closed* vertex x has already got its optimal distance, $\widehat{d_x}$ equals d'_x. After y is extracted, its shortest distance is updated to $d'_x + w(x, y)$. Now we prove by contradiction that this distance is y's optimal distance.

Let SP stand for the shortest path from s to y that is computed by *MBallStringInc*. Assume some other path SP^* is shorter than SP. As shown in Figure 17(b), let SP^* be composed by a shortest path from s to another *closed* vertex u, a boundary edge (u, v), and a shortest path from v to y. As we assume that SP^* is shorter than P, we have $d'_u + w(u, v) + d'_{vy} < d'_x + w(x, y)$ (1). Since all input edge changes are increases, the shortest distance between any two vertices in G' could only be increased, thus $d'_{vy} \geq d_{vy}$ (2). By combining (1) and (2), we get $d'_u + w(u, v) + d_{vy} < d'_x + w(x, y)$ (3). Meanwhile according to Lemma A.12, $d_{vy} \geq d_y - d_v$ (4). By combining (3) and (4), we get $d'_u + w(u, v) + d_y - d_v < d'_x + w(x, y)$, which leads to $d'_u + w(u, v) - d_v < d'_x + w(x, y) - d_y$ (5). According to *MBallStringInc*, an *open* vertex's distance is updated only in line 26, and right after that its status is set back to *closed* in line 27. Since v and y are still *open*, their distances have not been updated yet. Therefore according to the definition of δ, inequality (5) is actually $\delta_v < \delta_y$. However we know $\delta_v \geq \delta_y$ because y is extracted before v. A contradiction. Thus, no other path from s to y is shorter than P located by algorithm *MBallStringInc*.

Now we prove that, besides y, other consolidated vertices w also get their optimal shortest distances. See Figure 18 for the explanation. Basically, we apply the same strategy here.

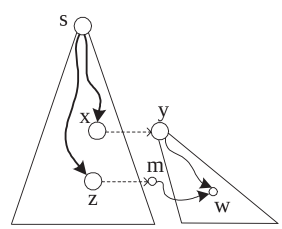

Figure 18: The illustration of possible shortest paths for other consolidated vertices. Legend: the triangle with s on the top stands for the subtree rooted at s that is consolidated so far; the triangle with y on the top stands for the set of vertices returned by $des(\widehat{T_s}, y)$.

For any vertex w in N returned by $des(\widehat{T_s}, y)$ in line 24, *MBallStringInc* updates its shortest distance to $\widehat{d_w} + \delta_y$, which is $\widehat{d_w} + d'_x + w(x, y) - \widehat{d_y}$; and locates the corresponding path SP as the shortest path from s to w. Suppose there is a path SP^* from s to w that is shorter than SP. Based on the same argument as before, let SP^* be composed of the shortest path from s to a *closed* vertex z, a boundary edge (z, m), and the shortest path from m to w in G'. As assumed, $d'_z + w(z, m) + d'_{mw} < \widehat{d_w} + d'_x + w(x, y) - \widehat{d_y}$ (6). Since $d'_{mw} \geq d_{mw}$, inequality (6) can be extended to $d'_z + w(z, m) + d_{mw} < \widehat{d_w} + d'_x + w(x, y) - \widehat{d_y}$ (7). Meanwhile, we have $d_m + d_{mw} \geq\ d_w$ (8), according to Lemma A.12. By combining (7) and (8), we get $d'_z + w(z, m) - d_m < d'_x + w(x, y) - d_y$ (9). Based on the same argument as before, inequality (9) is actually $\delta_m < \delta_y$. A contradiction. So there is no other path from s to w that is shorter than SP located by *MBallStringInc*. In addition, if w is also in Q, in line 28 of *MBallStringInc*, w is removed from Q, which is correct because w's optimal distance has been found.

In lines 32-38 of i^{th} iteration, *MBallStringInc* relaxes consolidated vertices. If any new boundary vertices are induced, they will be located and their candidate paths will be computed in this step. Also, if better candidate paths are induced, the information will be updated. Therefore, all boundary vertices will be enqueued with a candidate distance at the end of i^{th} iteration. We conclude that the inductive hypothesis holds after the i^{th} iteration. Therefore, Theorem A.20 stands. ∎

Lemma A.21 *MBallStringInc maintains tree edges correctly.*

Proof According to Lemma A.17, *MBallStringInc* only processes locally-affected vertices. Since *MBallStringInc* only updates the shortest path parent of a processed vertex, the tree edges headed at not-locally-affected vertices remain unchanged. For locally-affected vertices, *MBallStringInc* conducts branch consolidation. In each branch, all vertices' distances are updated by the same amount, therefore, tree edges in each branch remain unchanged. The root of each branch (except for the branch containing the root s) is connected to another branch by a tree edge in T'_s. Accordingly to lines 16 and 36, when a root v is enqueued, its candidate parent p is also enqueued. At line 20, v is extracted with parent p. The next three lines make sure that v's shortest path parent is correctly set to p. Therefore, *MBallStringInc* maintains all tree edges correctly. ∎

Corollary A.22 *Let T_s be a valid SPT rooted at vertex s in graph G. The graph G is modified into G' by a set of edge weight increases. Algorithm MBallStringInc computes a new valid SPT T'_s rooted at s in G'.*

Proof According to Lemma A.17, not-locally-affected vertices keep their optimal distances. For locally-affected vertices, the correctness follows from Lemmas A.19 and A.11, and Theorem A.20. For tree edges, the correctness follows Lemma A.21. Therefore, we conclude the correctness of Corollary A.22. ∎