The Dissertation Committee for Matteo Pontecorvi
certifies that this is the approved version of the following dissertation:

**Algorithms for Dynamic and Distributed Networks: Shortest Paths, Betweenness Centrality and Related Problems**

Committee:

Vijaya Ramachandran, Supervisor

Lorenzo Alvisi

Meghana Nasre

C. Greg Plaxton

David Zuckerman

# Algorithms for Dynamic and Distributed Networks: Shortest Paths, Betweenness Centrality and Related Problems

by

**Matteo Pontecorvi**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

December 2017

# Acknowledgments

First and foremost, I want to express my sincere gratitude to my advisor, Prof. Vijaya Ramachandran, for her mentorship and support during all my Ph.D. study and related research. I am deeply indebted to Vijaya. Her patience, motivation, and vast knowledge, are the main reasons of all the accomplishments I had in my graduate years. She taught me how to approach and investigate challenging problems, and her guidance was crucial for both my academic and a personal growth. Her profound ideas and reasoning were decisive to surpass seemingly unavoidable barriers during my research path, and largely shaped all the results in this thesis. It has been a great pleasure meeting and working with Vijaya, whom I consider the greatest boost in my entire academic career.

Apart from Vijaya, I would like to thank the rest of my thesis committee: Prof. Lorenzo Alvisi, Prof. Meghana Nasre, Prof. Greg Plaxton and Prof. David Zuckerman. They all provided insightful comments and encouragement, but also central questions related to my research which incented me to improve my abilities in communicating a scientific result to an expert audience, from various perspectives. Moreover, I cannot forget all the beautiful events where I had the opportunity to discover the friendly and generous side of all the members of my committee. I would also like to thank Prof. Anna Gal and Prof. Eric Price for useful discussions and for improving my teaching abilities during their courses.

I thank the Department of Computer Sciences at UT Austin for supporting me with an MCD graduate fellowship. This research was also supported by National Science Foundation under grant NSF-CCF-1320675.

I am very grateful to my colleagues and friends: Abhishek, Adrian, Akshay, Daniel, Fu, George, Harsh, Jia, John, Josh, Justin, Kevin, Matt, Onur, Orestis, Pravesh, Rashish, Ridwan, Rishabh, Serjay, Siddhesh, Sushrut, Trinabh, Udit, Venkata, Xue (and many others). Their presence, daily support and amazing moments spent together, kept me motivated especially when my research was not progressing as expected. Special thanks go to Daniele, Emanuele, Francesco, Lorenzo, Marco and Salvatore for challenging my mind with many interesting discussions, clever games and other ludic activities.

Last but not the least, I would like to thank my family: my parents, my brother and my grandma for supporting me spiritually throughout my academic years and my life in general. This thesis would have been impossible without their unceasing care and love.

Matteo Pontecorvi

*The University of Texas at Austin*
*December 2017*

# Algorithms for Dynamic and Distributed Networks: Shortest Paths, Betweenness Centrality and Related Problems

Publication No. ____________________

Matteo Pontecorvi, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Vijaya Ramachandran

In this thesis we study the problem of computing Betweenness Centrality in dynamic and distributed networks. Betweenness Centrality (BC) is a well-known measure for the relative importance of a node in a social network. It is widely used in applications such as understanding lethality in biological networks, identifying key actors in terrorist networks, supply chain management processes and more. The necessity of computing BC in large networks, especially when they quickly change their topology over time, motivates the study of dynamic algorithms that can perform faster than static ones. Moreover, the current techniques for computing BC requires a deeper understanding of a classic problem in computer science: computing all pairs *all* shortest paths (APASP) in a graph. One of the main contributions of this thesis is a collection of dynamic algorithms for computing APASP and BC scores which are provably faster than static algorithms for several classes of graphs. We use $n = |V|$ and $m = |E|$ to indicate respectively the number of nodes and edges in a directed positively weighted graph $G = (V, E)$. Our bounds depend on the parameter $\nu^*$ that is defined as the maximum number of edges that lie on shortest paths through any single vertex. The main results in the first part of this thesis are listed below.

- A decrease-only algorithm for computing BC and APASP running in time $O(\nu^* \cdot n)$ that is provably faster than recomputing from scratch in sparse graphs.
- An increase-only algorithm for computing BC and APASP that runs in $O(\nu^{*2} \cdot \log n)$ per update for a sequence of at least $\Omega(m^*/\nu^*)$ updates. Here $m^*$ is the number of edges in $G$ that lie on shortest paths. This algorithm uses $O(m^* \cdot \nu^*)$ space.
- An increase-only algorithm for computing BC and APASP that runs in $O(\nu^{*2} \cdot \log n)$ but improves the computational space to $O(m^* \cdot n)$.
- A fully dynamic algorithm for computing BC and APASP that runs in $O(\nu^{*2} \cdot \log^3 n)$ amortized time per update for a sequence of at least $\Omega(n)$ updates.
- A refinement of our fully dynamic algorithm that improves the amortized running time to $O(\nu^{*2} \cdot \log^2 n)$, saving a logarithmic factor.

In the second part of this thesis, we study the case when the input graph is a distributed network of machines and the BC score of each machine, considering its location within the network topology, needs to be computed. In this scenario, each node in the input graph is a self-contained machine with limited knowledge of the

network and communication power. Each machine only knows the (virtual) location of the neighbors machines (adjacent nodes in the input graph). The messages, exchanged in each round between machines, cannot exceed a bounded size of at most $O(\log n)$ bits. In this distributed model, called CONGEST, we present algorithms for computing BC in near-optimal time for unweighted networks, and some classes of weighted networks. Specifically, our main results are:

- A distributed BC algorithm for unweighted undirected graphs completing in at most $\min(2n+O(D_u), 4n)$ rounds, where $D_u$ is the diameter of the undirected network.
- A distributed BC algorithm for unweighted directed graphs completing in at most $\min(2n + O(D), 4n)$ rounds, where $D$ is the diameter of the directed network.
- A distributed APSP algorithm for unweighted directed graphs completing in at most $\min(n + O(D), 2n)$ rounds.
- A distributed BC algorithm for weighted directed acyclic graphs (dag) completing in at most $2n + O(L)$ rounds, where $L$ is the longest length of a path in the dag.
- A distributed APSP algorithm for weighted dags completing in at most $n + O(L)$ rounds.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In this thesis, we consider the problem of designing efficient algorithms for dynamic and distributed networks. In particular, we focus on problems related to shortest paths (SPs) such as computing betweenness centrality (BC) and multiple shortest paths in dynamic and distributed networks. The problems we study are closely related and are used in several applications.

**Betweenness Centrality.** The *betweenness centrality* $BC(v)$ of a node $v$ is often used as a parameter that determines the importance of $v$ in $G$, relative to the presence of $v$ on shortest paths, and is computed for all $v \in V$. As a classical measure, BC is widely used in sociology [Fre77, Ley07], physics [KHP$^+$07] and network analysis [PAE$^+$13, SG05]. In recent years, BC also had a wide impact in the analysis of social networks [GOKK03, Ram04], wireless [MK12] and mobile networks [CFF13], P2P networks [KI12] and more. Others applications of BC include analyzing social interaction networks [KAS$^+$12], identifying lethality in biological networks [PMW05], identifying key actors in terrorist networks [CGM04, Kre02], identify and prevent security attacks on mobile and complex networks [QH10, HKYH02]. BC is also used for identifying community structure in social and biological networks using the Girvan–Newman algorithm [GN02], and for understanding road network patterns of traffic analysis zones [ZWZC11]. Many of the above systems are usually represented as directed networks (see Section 7 in [MV13]), and this motivates our interest in studying solutions for computing BC in directed graphs.

As described later, all the current techniques known for computing BC require to implicitly discover *all* the shortest paths for any pair of nodes in the graph. We call this problem All Pairs All Shortest Paths (APASP), and all our results can also be used to solve this task.

**Dynamic Algorithms.** Computing BC can be an expensive task in terms of computational resources. In fact, the static algorithms are usually too slow for very large networks, especially when the topology (of the network) quickly changes over time because of quick updates. One approach to solve this problem is to use dynamic algorithms. In general, dynamic algorithms are used to quickly recompute a property of the input graph after an update changed only a local region of the graph, and not the entire topology. The efficiency of dynamic algorithms comes from their ability to reuse the data available before the graph was updated, in order to recompute the graph property without processing again the entire dataset. There is a vast literature on dynamic algorithms for shortest paths [DI04, Kin99, FMSN00, AISN91, ACK17, BHS07, Tho04, Tho05]. For this reason, we investigate the possibility of recomputing APASP and BC in a network experiencing dynamic updates, without re-applying the static algorithm after each update. Our dynamic algorithms will improve over the static ones in common classes of graphs that can easily represents real application networks.

**Distributed Model.** As another avenue to explore, we study the problem when the data set is too large to fit a single machine and each node in the networks is considered a complete machine. These models of distributed and decentralized networks are fundamental to understand modern computer systems. Their properties are well studied in the field of Computer Science, especially the computation, required to solve global tasks, that is locally performed at each node of the network [Lyn96a, Gar02, Tel01]. However, understanding only the local properties of a distributed network is usually not sufficient to solve many problems. Important optimization tasks such as computing shortest paths, determining the diameter of the network etc., require the understanding of "global" properties in the sense that the information used by the algorithm must travel to the farthest node in the network. In this direction, we provide efficient algorithms for computing BC in directed unweighted networks and some class of weighted networks. Moreover, our solutions imply algorithms for computing APSP, Reachability, Transitive Closure and SCC in the same class of networks.

## 1.1 Computational Problems and Related Works

In this section, we describe the problems studied in the first part of this thesis. We also give a summary regarding their state-of-art, focusing only on static and dynamic algorithms. We initially review the classic all pair shortest paths problem which is intimately connected to the definition of BC (Section 1.1.2). Then, we focus on the more general all pairs *all* shortest paths problem which is a central tool to compute BC (Section 1.1.3), and finally we review the BC problem itself (Section 1.1.4). We defer problems, definitions and related works for the distributed models to Chapter 7.

### 1.1.1 Basic Definitions

Unless specified, we refer to directed graphs (*digraphs*) $G = (V, E)$, where $V$ is the set of nodes and $E \subseteq V \times V$ is the set of edges. Let $\mathbf{w} : E \to \mathbb{R}^+$ denote the edge weight function on the edges of $G$. Let $\pi_{st}$ denote a path from $s$ to $t$ in $G$.

Define $\mathbf{w}(\pi_{st}) = \sum_{e \in \pi_{st}} \mathbf{w}(e)$ as the weight of the path $\pi_{st}$. We use $d(s,t)$ (or $\delta(s,t)$ when specified by the context) to denote the weight of a shortest path from $s$ to $t$ in $G$, also called its *distance*. We assume $d(s,t) = \infty$ if there is no path from $s$ to $t$. Let $E^*$ be the set of edges in $G$ that lie on shortest paths, let $m^* = |E^*|$ (see, e.g., Karger et al. [KKP93]), and let $\nu^*$ be the maximum number of edges that lie on shortest paths through any single vertex. The *length* $\ell(\pi_{st})$ is the number of edges in $\pi_{st}$. In many cases we will consider *directed acyclic graphs* (dags). We call *out-dag(v)* the single source shortest path (SSSP) dag representing all the shortest paths starting from $v$ to every other node in $V$. Similarly, *in-dag(v)* is the SSSP dag representing all the shortest paths starting from each node in $V$ and ending in $v$. Note that $\nu^*$ also bounds the number of edges that lie on any single-source shortest path dag. For a dag $G$, we call $L$ the length of a longest (in terms of number of edges) path in $G$. For a node $u \in V$ we define $\Gamma_{\text{in}}(u) = \{v \in V \mid (v,u) \in E\}$ as the set of *incoming neighbors* of $u$ and $\Gamma_{\text{out}}(u) = \{v \in V \mid (u,v) \in E\}$ as the set of the *outgoing neighbors* of $u$. Let $U_G$ be the undirected version of $G$. A digraph $G$ is *weakly connected* if $U_G$ is connected. A digraph $G$ is *strongly connected* if it contains at least one directed path $u \rightsquigarrow v$ and at least one directed path $v \rightsquigarrow u$ for each pair of nodes $u, v \in V$. Similarly, we can define a *weakly connected component* (wcc) and *strongly connected component* (scc) in a digraph. All our results also apply to undirected graphs using standard reductions (inserting two directed edges for each indirect one). We use $D$ to denote the diameter of the directed graph $G$, while if the graph is undirected we use $D_u$.

**Discussion of the Parameter $\boldsymbol{\nu^*}$.** Both $\nu^*$ and $m^*$ are typically much smaller than $m$ in dense graphs. For instance, it is known [FG85, HZ85, KKP93, LR89] that $m^* = O(n \log n)$ with high probability in a complete graph where edge weights are chosen from a large class of probability distributions, including the uniform distribution on integers in $[1, n^2]$ or reals in $[0,1]$. Since $\nu^* \leq m^*$, our algorithms will have an amortized bound of $O(n^2 \cdot polylog(n))$ on such graphs. Also, $\nu^* = O(n)$ in any graph with only $k$ shortest paths, where $k$ is a constant, between every pair of vertices. These graphs are called $k$-geodetic [RMRR98], and are well studied in graph theory [SOA88, BM84, Mul82]. In fact $\nu^* = O(n)$ even in some graphs that have an exponential number of SPs between some pairs of vertices. In contrast, $m^*$ can be $\Theta(n^2)$ even in some graphs with unique SPs, for example the complete unweighted graph $K_n$. Another type of graph with $\nu^* \ll m^*$ is one with large clusters of nodes, as described by the *planted $\ell$-partition model* [CK01, Sch07]: consider a graph $H$ with $k$ clusters of size $n/k$ (for some constant $k \geq 1$) with $\delta < w(e) \leq 2\delta$, for some constant $\delta > 0$, for each edge $e$ in a cluster; between the clusters is a sparse interconnect. Then $m^* = \Omega(n^2)$ but $\nu^* = O(n)$.

Note that this section contains only general definitions used in this thesis. Additional specific definitions will be provided in each chapter in order to understand the problems presented.

### 1.1.2 All Pairs Shortest Paths (APSP)

Given a weighted graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, the *APSP (all pairs shortest paths)* problem requires to find a shortest path between every pair of vertices in a graph.

For undirected graphs, the problem was introduced by Shimbel [Shi53] and solved by the author in total time of $O(n^4)$. The best available bound for undirected graphs is given by Pettie and Ramachandran [PR05], where the authors present a $O(mn \log \alpha)$ algorithm for APSP (where $\alpha(m, n)$ is the very slowly growing inverse-Ackerman function). For directed graphs, the best available result is given by Pettie in [Pet04] with a $O(mn + n^2 \log \log n)$ algorithm. For positive integers weights, Thorup [Tho99] shows a $O(mn)$ time algorithm (assuming constant time multiplication) for undirected graphs, while Hagerup [Hag00] gives a $O(mn + n^2 \log \log n)$ for directed graphs.

An interesting question is how to compute the APSP problem on dynamic networks, without running static algorithms from scratch. The early dynamic algorithms that were faster than recomputing APSP from scratch only worked on graphs with small integer weights. Ausiello [AISN91] proposed a decrease-only APSP algorithm for directed graphs with integer weights less than $C$, with an amortized running time of $O(Cn \log n)$ per edge insertion. A randomized increase-only algorithm for APSP, for unweighted graphs, is presented in Baswana et al. [BHS07]: it returns $(1+\epsilon)$-approximate shortest paths in $O(\frac{n \ln n}{\epsilon^2} + \frac{n^2\sqrt{\ln n}}{\epsilon\sqrt{m}})$, where $\epsilon > 0$ is a small constant. A fully dynamic algorithm (which is faster than recomputing from scratch) for computing APSP on general graphs, with positive integer edge weights less than $C$, was developed by King [Kin99]: it has running time $O(n^{2.5}\sqrt{C \log n})$.

Dealing with arbitrary weights required new combinatorial techniques. Demetrescu and Italiano presented a fully dynamic algorithm (the '`DI`' method) for APSP on general graphs with non-negative real weights [DI04]. The `DI` algorithm runs in $O(n^2 \log^3 n)$ amortized time per update. A method that is faster by a logarithmic factor was given by Thorup [Tho04] (the '`Thorup`' method), but this algorithm is considerably more complicated. Thorup also presented a worst case fully dynamic algorithm [Tho05] which recomputes the complete distance matrix in $\tilde{O}(n^{2.75})$. Recently, the worst case time bound was improved by Abraham et al. [ACK17] with a randomized algorithm which runs in $O(n^{2+2/3} \log^{4/3} n)$.

All the dynamic algorithms mentioned above maintains a single shortest path for each pair of nodes. In the next section, we introduce the more general case where *all* the shortest paths are required for each pair of nodes.

### 1.1.3 All Pairs All Shortest Paths (APASP)

Given a directed graph $G = (V, E)$ with positive edge weights, we consider the problem of maintaining *AP$\underline{A}$SP (all pairs $\underline{all}$ shortest paths)* [NPR14b], i.e., the set of all shortest paths between all pairs of vertices. This is a fundamental graph property, and it also enables the efficient computation of *betweenness centrality* (BC) (Section 1.1.4) for every vertex in the graph. A natural way to represent APASP is to maintain an in-dag and an out-dag for each vertex $v \in V$. We use these structures in our algorithms to recompute BC efficiently after maintaining the APASP.

Efficient algorithms for the APASP problem were not known in dynamic settings prior to our results which we describe below. In this thesis we will present semi dynamic and fully dynamic algorithms for APASP (see Section 1.3 of this Chapter).

### 1.1.4 Betweenness Centrality (BC)

Betweenness centrality (BC) is a widely used measure in the analysis of large complex networks. Informally, the BC of a node $v$ in a network is the fraction of all shortest paths in the network that go through $v$, and this measure is often used as an index that determines the relative importance of $v$ in the network. Formally, is defined as follows. Given a directed graph $G = (V, E)$ with $|V| = n$, $|E| = m$ and positive edge weights, let $\sigma_{xy}$ denote the number of shortest paths (SPs) from $x$ to $y$ in $G$, and $\sigma_{xy}(v)$ the number of SPs from $x$ to $y$ in $G$ that pass through $v$, for each pair $x, y \in V$. Then,

$$BC(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

As in [Bra01], we assume positive edge weights to avoid the case where cycles of 0 weight are present in the graph.

In the static case, the widely used algorithm by Brandes [Bra01] runs in $O(mn + n^2 \log n)$ on weighted graphs. We give an overview of the techniques used by Brandes in Section 1.2 below.

Given the changing nature of the networks under consideration, it is desirable to have algorithms that compute BC faster than computing it from scratch after every change. A contribution in this thesis is the first set of dynamic algorithms for computing BC after an update on an edge or on a vertex that are provably faster, on several classes of graphs, than the widely used static algorithm by Brandes [Bra01].

## 1.2 Brandes' Algorithm for BC

Brandes' algorithm is one of the first non-trivial algorithms for computing BC. Most of our results are based on different techniques used in this approach, thus we carefully review them in this section.

### 1.2.1 Preliminaries

The following notation was developed by Brandes [Bra01]. For a source $s$ and a vertex $v$, let $P_s(v)$ denote the predecessors of $v$ on shortest paths from $s$, i.e.,

$$P_s(v) = \{u \in V : (u,v) \in E \text{ and } d(s,v) = d(s,u) + \mathbf{w}(u,v)\} \tag{1.1}$$

Further, let $\sigma_{st}$ denote the number of shortest paths from $s$ to $t$ in $G$ (with $\sigma_{ss} = 1$). Finally, let $\sigma_{st}(v)$ denote the number of shortest paths from $s$ to $t$ in $G$ that pass through $v$. It follows from the definition that,

$$\sigma_{st}(v) = \begin{cases} 0 & \text{if } d(s,t) < d(s,v) + d(v,t) \\ \sigma_{sv} \cdot \sigma_{vt} & \text{otherwise} \end{cases} \tag{1.2}$$

The dependency of the pair $s, t$ on an intermediate vertex $v$ is defined in [Bra01] as the *pair dependency* $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$.

For $v \in V$, the *betweenness centrality* $\mathrm{BC}(v)$ is defined by Freeman [Fre77] as:

$$\mathrm{BC}(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{s \neq v, t \neq v} \delta_{st}(v) \tag{1.3}$$

The following two-step procedure computes BC for all $v \in V$:

1. For every pair $s, t \in V$, compute $\sigma_{st}$.
2. For every vertex $v \in V$, and for every $s, t$ pair, compute $\sigma_{st}(v)$ (Equation 1.2), and then compute $\mathrm{BC}(v)$ (Equation 1.3).

Step 1 above can be achieved by $n$ executions of Dijkstra's single source shortest paths algorithm. Therefore Step 1 takes time $O(mn + n^2 \log n)$ time, if we use a priority queue with $O(1)$ amortized cost for the decrease-key operation. For every vertex $v$, Step 2 takes $O(n^2)$ time, since there are $O(n^2)$ pair dependencies. This gives a $\Theta(n^3)$ time algorithm to compute BC for all vertices. Thus, the bottleneck of the above algorithm is the second step which explicitly sums up the pair dependencies for every vertex. To obtain a faster algorithm for sparse graphs, Brandes [Bra01] defined the dependency of a vertex $s$ on a vertex $v$ as: $\delta_{s\bullet}(v) = \sum_{t \in V \setminus \{v,s\}} \delta_{st}(v)$. Brandes [Bra01] also made the useful observation that the partial sums satisfy a recursive relation. In particular, the dependency of a source $s$ on a vertex $v \in V$ can be written as:

$$\delta_{s\bullet}(v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)) \tag{1.4}$$

(See [Bra01] for a proof of Equation 1.4.) The above equation gives an efficient algorithm for computing BC described in the next section.

### 1.2.2 The Brandes Algorithm

We present a high level overview of Brandes' algorithm. The algorithm begins by initializing the BC score for every vertex to 0. Next, for every $s \in V$, it executes Dijkstra's SSSP algorithm. During this step, for every $t \in V$, it computes $\sigma_{st}$, the number of shortest paths from $s$ to $t$, and $P_s(t)$, the set of predecessors of $t$ on shortest paths from $s$. Additionally, the algorithm stores the vertices $v \in V$ in a stack $S$ in order of non-increasing value of $d(s, v)$. Finally, to compute the BC score, the algorithm accumulates the dependency of $s$. We now elaborate on this final step, which is given in Algorithm 2 (Accumulate-dependency). This algorithm takes as its input a source $s$ for which Dijkstra's SSSP algorithm has been executed and the stack $S$ containing vertices ordered by distance from $s$. The algorithm repeatedly extracts a vertex from $S$ and accumulates the dependency using Equation 1.4. The time taken by Algorithm 2 is linear in the size of the DAG rooted at $s$, i.e., it is $O(m_s^*)$.

Note that in Brandes' algorithm, the set $S$ contains vertices $v \in V$ ordered in non-increasing value of $d(s, v)$. However, for the dependency accumulation it suffices that $S$ contains vertices $v \in V$ ordered in the reverse topological order of the DAG($s$). Such an ordering ensures that the dependency of a vertex $w$ is *accumulated* to any of its predecessor $v$, only after all the successors of $w$ in DAG($s$) have been processed. This observation is useful for our decrease-only algorithm, since topological sort can be performed in linear time.

The static Brandes algorithm computes BC scores in a two phases process. The first phase computes the SP out-dag for every source through $n$ applications of Dijkstra's algorithm. During this phase, it also keeps track of the distances and number of shortest paths for each pair of nodes. The second phase uses an *accumulation technique* that computes all BC scores using these SP dags in $O(mn)$ time. Here, we give a brief summary on the accumulation technique which is also used in our dynamic algorithms. Brandes defined *pair dependency* of $s, t$ on an intermediate vertex $v$ as $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. Thus, for $v \in V$, the *betweenness centrality* BC($v$) is:

$$\mathrm{BC}(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{s \neq v, t \neq v} \delta_{st}(v) \tag{1.5}$$

Let $P_s(v)$ denote the predecessors of $v$ on shortest paths from $s$. The dependency of a vertex $s$ on a vertex $v$ is $\delta_{s\bullet}(v) = \sum_{t \in V \setminus \{v,s\}} \delta_{st}(v)$, and Brandes observed that

$$\delta_{s\bullet}(v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))\,1 \quad \text{and} \quad \mathrm{BC}(v) = \sum_{s \neq v} \delta_{s\bullet}(v) \tag{1.6}$$

We can accumulate the dependency $\delta_{s\bullet}(v)$ for each node $v$ by recursively applying equation 1.6 starting from the most distant (from the root) node in the SSSP dag rooted at $s$, and proceeding in non-increasing distance order up to the root $s$.

**Algorithm 1** Betweenness-centrality($G = (V, E)$) (from [Bra01])

```
1: for every $v \in V$ do BC($v$) $\leftarrow 0$
2: for every $s \in V$ do
3:    run Dijkstra's SSSP from $s$ and compute $\sigma_{st}$ and $P_s(t), \forall\, t \in V \setminus \{s\}$
4:    store the explored nodes in a stack $S$ in non-increasing distance from $s$
5:    accumulate dependency of $s$ on all $t \in V \setminus s$ using Algorithm 2
```

In our dynamic BC algorithms we will keep phase two unchanged, although we will apply our dynamic APASP algorithms to update dags, distances and number of paths maintained in phase one.

**Algorithm 2** Accumulate-dependency$(s, S)$ (from [Bra01])

```
Require: For every $t \in V$: $\sigma_{st}, P_s(t)$
    A stack $S$ containing $v \in V$ in a suitable order (non-increasing $d(s,v)$ in [Bra01])
1: for every $v \in V$ do $\delta_{s\bullet}(v) \leftarrow 0$
2: while $S \neq \emptyset$ do
3:     $w \leftarrow \text{pop}(S)$
4:     for $v \in P_s(w)$ do $\delta_{s\bullet}(v) \leftarrow \delta_{s\bullet}(v) + \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))$
5:     if $w \neq s$ then $\text{BC}(w) \leftarrow \text{BC}(w) + \delta_{s\bullet}(w)$
```

## 1.3 Dynamic BC and APASP

Computing BC (and APASP) in a real world graph using a static algorithm is not the most efficient solutions, especially when the underlying graph topology changes slightly over time. For this reason, algorithms are needed that quickly re-compute the property in the modified graph. In general, algorithms that make use of previous solutions to solve the problem faster than a recomputation from scratch, are called *dynamic graph algorithms.* In our thesis, we will consider different classes of dynamic algorithms:

- *A decrease-only* algorithm (or *incremental* algorithm) allows edge updates where the weight of the edges can only be decreased. A more general version allows node updates where the weights of any subset of edges incident to a given node can only be decreased.

- An *increase-only* algorithm (or *decremental* algorithm) allows edge updates where the weight of the edges can only be increased. Again, a more general version allows node updates where the weights of any subset of edges incident to a given node can only be increased.

- A *fully dynamic* algorithm, which is the more general case, allows node updates where any subset of edges incident to a given node can change their weights arbitrarily.

In this thesis, we will present efficient algorithms in each one of the above classes to compute BC and APASP. In Chapter 2 we present our decrease-only algorithm for BC and APASP problems, where each update in $G$ is exclusively decrease-only. In Chapter 3 we present our increase-only algorithm for BC and APASP problems, where each update in $G$ In Chapters 5 and 6 we show two fully dynamic algorithms for BC and APASP, which support both decrease-only and increase-only updates. We now present a summary of the results already known for computing BC.

### 1.3.1 Related Results

Together with the classic Brandes algorithm, there are many previous results for computing BC. Several BC algorithms for approximation and parallel algorithms have been considered in [BKMM07, GSS08a, RK14] and [MEJ$^+$09] respectively. Heuristics for dynamic betweenness centrality with good experimental performance are given in [GMB12, LLP$^+$12, SGIS13], but none provably improve on Brandes.

**Dynamic Algorithms.** The problem of computing betweenness centrality dynamically has received increasing attention, and several results for decrease-only or increase-only BC are listed in the table below in the section 'Semi Dynamic'.
All of these results except [KWCC13] deal with unweighted graphs as opposed to our results, which are for the weighted case. As mentioned above, BC is also widely used in weighted networks (see [CFF13, KI12, PMW05, PAE$^+$13]); however, only the heuristic in Kas et al. [KWCC13], which has no worst-case bounds, addresses this version.
In the area of fully dynamic algorithms, the results presented in [LLP$^+$12, SGIS13] are heuristics with no theoretical guarantees; the algorithm in [KMB14] has the same worst case as Brandes and works only for unweighted graphs, while the algorithm in [BM15] returns approximated BC scores.

The only exact dynamic BC algorithms that provably improve on Brandes

| **Paper** | **Year** | **Time** | **Weights** | **Update Type** | **DR/UN** | **Result** |
|---|---|---|---|---|---|---|
| [Bra01] | 2001 | $O(mn)$ | NO | Static Alg. | Both | Exact |
| [Bra01] | 2001 | $O(mn + n^2 \log n)$ | YES | Static Alg. | Both | Exact |
| [GSS08b] | 2007 | Heuristic | YES | Static Alg. | Both | Approx. |
| [RK14] | 2014 | depends on $\epsilon$ | YES | Static Alg. | Both | $\epsilon$-Approx. |
| | | | **Semi Dynamic** | | | |
| [GMB12] | 2012 | $O(mn)$ | NO | Edge Inc. | Both | Exact |
| [KWCC13] | 2013 | Heuristic | YES | Edge Inc. | Both | Exact |
| **NPR** Chapter 2 | 2014 | $O(\nu^* \cdot n)$ | YES | Vertex Inc. | Both | Exact |
| **NPRdec** Chapter 3 | 2014 | $O(\nu^{*2} \cdot \log n)$ | YES | Vertex Dec. | Both | Exact |
| [BMS15] | 2015 | depends on $\epsilon$ | YES | Batch (edges) Inc. | Both | $\epsilon$-Approx. |
| | | | **Fully Dynamic** | | | |
| [LLP+12] | 2012 | Heuristic | NO | Edge Update | UN | Exact |
| [SGIS13] | 2013 | Heuristic | NO | Vertex Update | UN | Exact |
| [KMB14] | 2014 | $O(mn)$ | NO | Edge Update | Both | Exact |
| [BM15] | 2015 | depends on $\epsilon$ | YES | Batch (edges) | UN | $\epsilon$-Approx. |
| **PR** Chapter 5 | 2015 | $O(\nu^{*2} \cdot \log^3 n)$ | YES | Vertex Update | Both | Exact |
| **PR(FFD)** Chapter 6 | 2015 | $O(\nu^{*2} \cdot \log^2 n)$ | YES | Vertex Update | Both | Exact |

Table 1.1: BC related results (DR stands for Directed and UN for Undirected). The results in this thesis are highlighted in bold.

on some classes of graphs are the separate decrease-only, increase-only and fully dynamic algorithms in [NPR14a, NPR14b, PR15b] presented in this thesis. Table 1.1 contains a summary of these results.

## 1.4 Distributed Algorithms

In the second part of this thesis (Chapter 7), we investigate our problem BC (and APASP) in the CONGEST model. In this model the network is modeled as a distributed and synchronous set of nodes, where the communication between a pair of nodes is bounded. The challenge here is to provide algorithms which can compute properties on the topology of the network (the input graph) in the fewest number of rounds. All the details are deferred to Chapter 7.

## 1.5 Organization and Main Results

The overall structure of this thesis is described below. We also highlight the main results presented in each chapter.

In Chapter 2, we give a simple decrease-only APASP algorithm (based on a joint work with Nasre and Ramachandran [NPR14a]), the '`NPR`' method, which updates the in-dags and out-dags after each decrease-only update. This algorithm allows a dynamic computation of the BC scores (for decrease-only updates) which runs in $O(\nu^* \cdot n)$ time and is provably faster than recomputing from scratch in sparse graphs.

In Chapter 3, we develop a more involved increase-only APASP algorithm (based on a joint work with Nasre and Ramachandran[NPR14b]), the '`NPRdec`' method, building on and extending the increase-only `DI` method (see Section 1.1.2) to APASP. The algorithm runs in $O(\nu^{*2} \cdot \log n)$ per update for a sequence of at least $\Omega(m^*/\nu^*)$ updates. Thus, this approach gives us a increase-only BC algorithm which is faster than a static recomputation on some classes of graphs including the class of dense graphs (where $m$ is close to $n^2$) with SSSP dags of size linear in the total number of vertices in $G$ (see the discussion in Section 1.1.1 for more details on these classes of graphs).

In Chapter 4 an improved version of `NPRdec` is proposed (based on a joint work with Nasre and Ramachandran), where we significantly reduce the space used by the algorithm using a new set of data structures.

In Chapter 5, we extend our results to the fully dynamic case developing a fully dynamic APASP algorithm (based on a joint work with Ramachandran [PR15b]), the '`PR`' method, which builds on [NPR14b] and is a variant of the fully dynamic `DI` method [DI04] for APSP. The algorithm runs in $O(\nu^{*2} \cdot \log^3 n)$ amortized time per update for a sequence of at least $\Omega(n)$ updates.

In Chapter 6, we refine our fully dynamic algorithm `PR` by reducing the complexity by a logarithmic factor and introducing (based on a joint work with Ramachandran

[PR15a]) the 'FFD' algorithm (for 'faster fully dynamic').

In the last chapter, we move to the widely studied CONGEST distributed model and we show how to efficiently compute BC and APASP in this setting (based on a joint work with Ramachandran) with near-optimal algorithms for unweighted directed (and undirected) networks. For unweighted graphs, we present a distributed BC algorithm which terminates in $\min\{2n + O(D), 4n\}$ rounds if $G$ is directed ($2n+O(D_u)$ rounds if $G$ is undirected). We also present a directed APSP algorithm which terminates in $\min\{n + O(D), 2n\}$ rounds. Finally, we give an algorithm for computing APSP in a weighted dag in $n + O(L)$ rounds. This algorithm is then enhanced to compute BC scores, in a weighted dag, for each node in $2n + O(L)$ rounds.

# Chapter 2

# Decrease-Only Algorithm

In this chapter we present a decrease-only BC algorithm that is provably faster on sparse graphs than current algorithms for the problem[1]. Our result works for both decrease-only edge updates and decrease-only vertex updates. By a *decrease-only edge update* on edge $(u, v)$ we mean the addition of a new edge $(u, v)$ with finite weight if $(u, v)$ is not present in the graph, or a decrease in the weight of an existing edge $(u, v)$; in a *decrease-only vertex update* on node $v$, decrease-only updates can occur on any subset of edges incident to $v$, and this includes adding new edges.

## 2.1 Our Contributions

Recall the definitions of $m^*$ and $\nu^*$ in Section 1.1.1, Chapter 1. Here is our main result:

**Theorem 1.** *After a decrease-only update on an edge or a vertex in a directed or undirected graph with positive edge weights, the betweenness centrality of all vertices can be recomputed in:*

*1. $O(\nu^* \cdot n)$ time using $O(\nu^* \cdot n)$ space;*

[1]The results presented in this chapter appeared in [NPR14a].

*2. $O(m^* \cdot n)$ time using $O(n^2)$ space.*

Since $\nu^* \leq m^*$ and $m^* \leq m$, the worst case time for both results is bounded by $O(mn + n^2)$, which is a $\log n$ factor improvement over Brandes' algorithm on sparse graphs with $m = o(n \log n)$. Our results also have benefits for dense graphs (when $m = \omega(n \log n)$ but $m^*$ remains small) similar to the Hidden Paths algorithm of Karger et al. [KKP93] for the APSP problem (see also McGeogh [McG95]), although our techniques are different. This is through the use of $\nu^*$ or $m^*$ in place of $m$, and we comment more on this below. Our algorithms are simple, and only use stack, queue and linked list data structures. Note that for the random real weights, the first result would give $O(n^2)$ time and space since shortest paths are unique with probability 1 in this setting, hence $\nu^* = O(n)$.

Our paper [NPR14a] also contains an efficient cache-oblivious implementation for our decrease-only results, which avoids the high caching cost of Dijkstra's algorithm that is present in Alg. 1. We will not discuss the above result in this thesis. Other cache-oblivious results for computing betweenness centrality (with a static algorithm) can be found in [AGvW13].

We observe that Alg. 1 (Brandes) can be made to run faster: In a directed graph, by using the Pettie [Pet04] or the Hidden Paths algorithm in place of Dijkstra in Step 3 of Alg. 1, we can compute BC scores in $O(mn + n^2 \log \log n)$ or $O(m^*n + n^2 \log n)$ time, respectively. In an undirected graph, we can obtain $O(mn \cdot \log \alpha(m, n))$ time, where $\alpha$ is an inverse-Ackermann function, using [PR05]. Our decrease-only bounds are better than any of these bounds for sparse graphs.

As seen in Section 1.3.1 Chapter 1, there are several results on dynamic BC algorithms and heuristics [SGIS13, GMB12, KMB14, LLP$^+$12], but our time bounds are better than any of these on sparse graphs. In fact, ours is the first decrease-only BC algorithm that gives a provable improvement over Brandes' algorithm for sparse graphs, which are the type of graphs that typically occur in practice. While the space used by our algorithms is higher than Brandes', which uses only linear space, our second result matches the best space bound obtained by any of these other dynamic BC algorithms and heuristics.

We consider only decrease-only updates in this chapter. Computing increase-only and fully dynamic updates efficiently appears to be more challenging (as is the case for APSP [DI04]). In the following chapters, we will develop increase-only and fully dynamic BC algorithms that build on techniques in [DI04], and run in amortized time $O(\nu^{*2} \cdot polylog(n))$.

**Organization.** Since the algorithm for a single edge update is simpler than that for a vertex update, we first present our edge update result in Section 2.2. We describe the $O(n \cdot \nu^*)$ algorithm, and then the simple changes needed to obtain the second $O(n^2)$ space result. Finally, we present the vertex update result in Section 2.3.

## 2.2 Decrease-Only Edge Update

In this section we present our algorithm to recompute BC scores of all vertices in a directed graph $G = (V, E)$ after a decrease-only edge update (i.e., adding an edge or decreasing the weight of an existing edge). Let $G' = (V, E')$ denote the graph obtained after an edge update to $G = (V, E)$. A path $\pi_{st}$ from $s$ to $t$ in $G$ has *weight* $\mathbf{w}(\pi_{st}) = \sum_{e \in \pi_{st}} \mathbf{w}(e)$. Let $d(s,t), \sigma_{st}$, $\delta_{s\bullet}(t)$ and DAG($s$) denote the distance from $s$ to $t$ in $G$, the number of shortest paths from $s$ to $t$ in $G$, the dependency of $s$ on $t$ and the SSSP DAG rooted at $s$ in $G$ respectively; let $d'(s,t), \sigma'_{st}$, $\delta'_{s\bullet}(t)$ and DAG$'(s)$ denote these parameters in $G'$.

**Lemma 1.** *If weight of edge $(u, v)$ in $G$ is decreased to obtain $G'$, then for any $x \in V$, the set of shortest paths from $x$ to $u$ and from $v$ to $x$ is the same in $G$ and $G'$, and $d'(x, u) = d(x, u)$, $d'(v, x) = d(v, x)$ ; $\sigma'_{xu} = \sigma_{xu}$, $\sigma'_{vx} = \sigma_{vx}$.*

*Proof.* Since edge weights are positive, the edge $(u, v)$ cannot lie on a shortest path to $u$ or from $v$. The lemma follows. $\square$

By Lemma 1, $\text{DAG}(v) = \text{DAG}'(v)$ after the decrease of weight on edge $(u, v)$. The next lemma shows that after the weight of $(u, v)$ is decreased we can efficiently obtain the updated values $d'(s, t)$ and $\sigma'_{st}$ for any $s, t \in V$.

**Lemma 2.** *Let the weight of edge $(u, v)$ be decreased to $\mathbf{w}'(u, v)$, and for any given pair of vertices $s, t$, let $D(s, t) = d(s, u) + \mathbf{w}'(u, v) + d(v, t)$. Then,*

1. *If $d(s, t) < D(s, t)$, then $d'(s, t) = d(s, t)$ and $\sigma'_{st} = \sigma_{st}$.*
   *The shortest paths from $s$ to $t$ in $G'$ are the same as in $G$.*

2. *If $d(s, t) = D(s, t)$, then $d'(s, t) = d(s, t)$ and $\sigma'_{st} = \sigma_{st} + (\sigma_{su} \cdot \sigma_{vt})$.*
   *The shortest paths from $s$ to $t$ in $G'$ are a superset of the shortest paths $G$.*

3. *If $d(s, t) > D(s, t)$, then $d'(s, t) = D(s, t)$ and $\sigma'_{st} = \sigma_{su} \cdot \sigma_{vt}$.*
   *The shortest paths from $s$ to $t$ in $G'$ are new (shorter distance).*

*Proof.* Case 1 holds because the shortest path distance from $s$ to $t$ remains unchanged and no new shortest path is created in this case. In case 2, the shortest path distance from $s$ to $t$ remains unchanged, but there are $\sigma_{su} \cdot \sigma_{vt}$ new shortest paths from $s$ to $t$ created via edge $(u, v)$. In case 3, the shortest path distance from $s$ to $t$ decreases and all new shortest paths pass through $(u, v)$. $\square$

By Lemma 2, the updated values $d'(s, t)$ and $\sigma'_{st}$ can be computed in constant time for each pair $s, t$. Once we have the updated $d'(\cdot)$ and $\sigma'_{(\cdot)}$ values, we need the updated predecessors $P'_s(t)$ for every $s, t$ pair for Alg. 2. The SSSP $\text{DAG}(s)$ rooted at a source $s$ is the union of all the $P_s(t), \forall\ t \in V$. Thus, obtaining $\text{DAG}'(s)$ after the edge update is equivalent to computing the $P'_s(t), \forall\ t \in V$. The next section gives a simple algorithm to maintain the SSSP DAGs rooted at every source $s \in V$, after a decrease-only edge update.

### 2.2.1 Updating an SSSP DAG

For each pair $s, t$ we define $flag(s,t)$ to indicate the specific case of Lemma 2 that is applicable.

$$flag(s,t) = \begin{cases} \text{UN-changed} & \text{if } d'(s,t) = d(s,t) \text{ and } \sigma'_{st} = \sigma_{st} \quad \text{(Lemma 2-1)} \\ \text{NUM-changed} & \text{if } d'(s,t) = d(s,t) \text{ and } \sigma'_{st} > \sigma_{st} \quad \text{(Lemma 2-2)} \\ \text{WT-changed} & \text{if } d'(s,t) < d(s,t) \quad \text{(Lemma 2-3)} \end{cases}$$

By Lemma 2, $flag(s,t)$ can be computed in constant time for each pair $s, t$. Given an input $s$ and the updated edge $(u, v)$, Alg. 3 (Update-DAG) constructs a set of edges $H$ using these $flag$ values, together with DAG($s$) and DAG($v$). We will show that $H$ contains exactly the edges in DAG$'$($s$). The algorithm considers edges in DAG($s$) (Steps 3–5) and edges in DAG($v$) (Steps 6–8), and for each edge $(a, b)$ in either DAG, it decides whether to include it in $H$ based on the value of $flag(s, b)$. For the updated edge $(u, v)$ there is a separate check (Steps 9–10). The algorithm clearly takes time linear in the size of DAG($s$) and DAG($v$), i.e., $O(\nu^*)$ time.

**Algorithm 3** Update-DAG($s, \mathbf{w}'(u, v)$)

```
Require: DAG($s$), DAG($v$), and $flag(s,t), \forall t \in V$.
Ensure: An edge set $H$ after decrease of weight on edge $(u,v)$, and $P'_s(t), \forall t \in$
    $V - \{s\}$.
 1: $H \leftarrow \emptyset$.
 2: for each $v \in V$ do $P'_s(v) = \emptyset$.
 3: for each edge $(a,b) \in$ DAG($s$) and $(a,b) \neq (u,v)$ do
 4:    if $flag(s,b) =$ UN-changed or $flag(s,b) =$ NUM-changed then
 5:       $H \leftarrow H \cup \{(a,b)\}$ and $P'_s(b) \leftarrow P'_s(b) \cup \{a\}$.
 6: for each edge $(a,b) \in$ DAG($v$) do
 7:    if $flag(s,b) =$ NUM-changed or $flag(s,b) =$ WT-changed then
 8:       $H \leftarrow H \cup \{(a,b)\}$ and $P'_s(b) \leftarrow P'_s(b) \cup \{a\}$.
 9: if $flag(s,v) =$ NUM-changed or $flag(s,v) =$ WT-changed then
10:    $H \leftarrow H \cup \{(u,v)\}$ and $P'_s(v) \leftarrow P'_s(v) \cup \{u\}$.
```

**Lemma 3.** *Let $H$ be the set of edges output by Alg. 3. An edge $(a,b) \in H$ if and only if $(a,b) \in DAG'(s)$.*

*Proof.* Since the update is a decrease-only update on edge $(u,v)$, we note that for any $b$, a shortest path $\pi'_{sb}$ from $s$ to $b$ in $G'$ can be of two types:
(i) $\pi'_{sb}$ is a shortest path in $G$. Therefore every edge on such a path is present in DAG($s$) and each such edge is added to $H$ in Steps 3–5 of Alg. 3.
(ii) $\pi'_{sb}$ is not a shortest path in $G$. However, since $\pi'_{sb}$ is a shortest path in $G'$, therefore $\pi'_{sb}$ is of the form $s \rightsquigarrow u \rightarrow v \rightsquigarrow b$. Since shortest paths from $s$ to $u$ in $G$ and $G'$ are unchanged (by Lemma 1), the edges in the sub-path $s \rightsquigarrow u$ are present in DAG($s$) and are added to $H$ in Steps 3–5 of Alg. 3. Finally, shortest paths from $v$ to any $b$ in $G$ and $G'$ remain unchanged. Thus, the edges in the sub-path $v \rightsquigarrow b$ are present in DAG($v$) and are added to $H$ in Steps 6–8 of Alg. 3.

For the other direction, if the edge $(a,b)$ is added to $H$ by Step 5, this implies that the edge $(a,b) \in$ DAG($s$). Thus, there exists a shortest path $\pi_{sb} = s \rightsquigarrow a \rightarrow b$ in $G$. We execute Step 5 when $flag(s,b) =$ UN-changed or $flag(s,b) =$ NUM-changed. Thus every shortest path from $s$ to $b$ in $G$ is also shortest path in $G'$. Therefore, $(a,b) \in$ DAG$'(s)$. If the edge $(a,b)$ is added to $H$ by Step 8, then the edge $(a,b) \in$ DAG($v$). Thus, there exists a shortest path $\pi_{vb} = v \rightsquigarrow a \rightarrow b$ in $G$. Since decreasing the weight of the edge $(u,v)$ does not change shortest paths from $v$ to any other vertex, $\pi_{vb}$ is in $G'$. We execute Step 8 when $flag(s,b) =$ NUM-changed or $flag(s,b) =$ WT-changed. Therefore, there exists at least one shortest path from $s$ to $b$ in $G'$ that uses the updated edge $(u,v)$. Hence the path $\pi'_{sb} = \pi'_{su} \cdot (u,v) \cdot \pi_{vb}$ is shortest in $G'$, and this establishes that $(a,b) \in$ DAG$'(s)$. Finally, edge $(u,v)$ is added to $H$ by Step 10 only if $flag(s,v)$ is NUM-changed or WT-changed, and in either case, there is at least a new shortest path from $s$ to $v$ through $(u,v)$. Hence $(u,v) \in$ DAG$'(s)$. □

### 2.2.2 Updating Betweenness Centrality Scores

The algorithm for updating the BC scores after an edge update (Alg. 4) is similar to Alg. 1, but with the following changes: an extended Step 1 also computes, for every $s, t$ pair, the updated $d'(s,t)$ and $\sigma'_{st}$, as well as $flag(s,t)$. Using Lemma 2, we spend constant time for each $s, t$ pair, hence $O(n^2)$ time for all pairs. In Step 3, instead of Dijkstra's algorithm, we run Alg. 3 to obtain the updated predecessor lists $P'_s(t)$, for all $s, t$. This step requires time $O(\nu^*)$ for a source $s$, and $O(\nu^* \cdot n)$ over all sources. The last difference is in Step 4: we place in the stack $S$ the vertices in reverse topological order in DAG$'(s)$, instead of non-increasing distance from $s$. This requires time linear in the size of the updated DAG. Thus the time complexity of Edge-Update is $O(\nu^* \cdot n)$.

**Algorithm 4** Edge-Update($G = (V, E), \mathbf{w}'(u, v)$)

```
Require: updated edge with $\mathbf{w}'(u,v)$, $d(s,t)$ and $\sigma_{st}$, $\forall\ s,t \in V$; DAG$(s), \forall\ s \in V$.
Ensure: BC$'(v)$, $\forall\ v \in V$; $d'(s,t)$ and $\sigma'_{st}\ \forall\ s,t \in V$; DAG$'(s), \forall\ s \in V$.
1: for every $v \in V$ do BC$'(v) \leftarrow 0$.
   for every $s,t \in V$ do compute $d'(s,t), \sigma'_{st}$, $flag(s,t)$.      // use Lemma 2
2: for every $s \in V$ do
3:    Update-DAG$(s, (u,v))$.                              // use Alg. 3
4:    Stack $S \leftarrow$ vertices in $V$ in a reverse topological order in DAG$'(s)$.
5:    Accumulate-dependency$(s, S)$.                       // use Alg. 2
```

**Undirected Graphs.** For an undirected $G$, we construct the corresponding directed graph $G_D$ in which every undirected edge is replaced with 2 directed edges. A decrease-only update on an undirected edge $(u, v)$ is equivalent to two edge updates on $(u, v)$ and $(v, u)$ in $G_D$. Thus, Theorem 1 holds for undirected graphs.

**Space Efficient Implementation.** In order to obtain $O(n^2)$ space complexity, we do not store the SSSP DAGs rooted at every source. Instead, we only store the edge set $E^*$. After a decrease-only update on edge $(u, v)$ we first construct the updated set $E'^*$ in $O(m^* \cdot n)$ time as follows. For each edge $(a, b) \in E^*$, if $d'(s, b) = d(s, a) + \mathbf{w}(a, b)$ for some source $s \in V$, then $(a, b) \in E'^*$. Using the updated $E'^*$ we can construct $\mathrm{DAG}'(s)$ in $O(m^*)$ time, by using the fact that an edge $(a,b) \in E'^*$ belongs to $\mathrm{DAG}'(s)$ iff $d(s,b) = d(s,a) + \mathbf{w}(a,b)$. Since the construction of each updated DAG takes $O(m^*)$ time and there are $n$ DAGs to be constructed, the $O(m^* \cdot n)$ time complexity follows. The space used is $O(m^* + n^2)$ to store $E^*$ and $d(s,t)$, $\sigma_{st}$, for all $s, t \in V$.

## 2.3 Decrease-Only Vertex Update

We now consider a decrease-only update to a vertex $v$ in $G = (V, E)$, which allows a decrease-only edge update on any subset of edges incoming to and outgoing from $v$. In this algorithm, we use the graph $G$ and the graph $G_R = (V, E_R)$, which is obtained by reversing every edge in $G$, i.e., $(a,b) \in E_R$ iff $(b,a) \in E$. Thus, for every $s \in V$, we also maintain $\mathrm{DAG}_R(s)$, the SSSP DAG rooted at $s$ in $G_R$. We will obtain the same time bound as in Section 2.2.

### 2.3.1 Overview

Let $E_i(v)$ and $E_o(v)$ denote the set of updated edges incoming to $v$ and outgoing from $v$ respectively. Our algorithm is a natural extension, with some new features, of the algorithm for a single edge update, and works as follows. We process $E_i(v)$ in $G$ in Step 1 to form $G'$, $G'_R$, $\mathrm{DAG}'(s)$ and $\mathrm{DAG}'_R(s)$; we then process $E_o(v)$ in $G'_R$ in a complementary Step 2 to obtain the updated $G''$, $\mathrm{DAG}''(s)$ and $\mathrm{DAG}''_R(s)$. Step 1, which processes $E_i(v)$, consists of two phases.

**Step 1, Phase 1:** Constructing the $\mathrm{DAG}'(s)$ for updates in $E_i(v)$.

Since $E_i(v)$ contains updated edges incoming to $v$, $\mathrm{DAG}(v) = \mathrm{DAG}'(v)$ (as in the single edge update case). In order to handle updates to several edges incoming to $v$, we strengthen Lemma 2 by introducing $\hat{\sigma}$, which keeps track of new shortest paths from $s$ to $v$ that go through any of the updated edges in $E_i(v)$. This allows us to efficiently recompute the number of shortest paths from a source to any node in $G'$, and thus update all the DAG$'(s)$ using an algorithm similar to Alg. 3. Parts (A), (B), (C) in Section 2.3.2 describe Phase 1 in detail.

**Step 1, Phase 2:** Constructing the DAG$'_R(s)$ for updates in $E_i(v)$.

We present an efficient algorithm to construct the DAG$'_R(s)$ for all $s$ in $G'$. We construct these reverse graphs because the edges in $E_o(v)$ are in fact incoming edges to $v$ in $G'_R$. Hence our method to maintain DAGs when incoming edges are updated can be applied to $G'_R$ with $E_o$ to obtain DAG$''_R(s)$, for every $s$, in Phase 1 of Step 2 (and then we can obtain the DAG$''(s)$ in Phase 2 of Step 2).

Let $(t, a)$ be the first edge on a shortest path from $t$ to $v$ in $G'$. Then $(t, a)$ is an outgoing edge from $t$ in DAG$'(t)$, and its reverse $(a, t)$ is on a shortest path from $v$ to $t$ in $G'_R$. Further an edge $(a, t)$ is on a new shortest path from $v$ to $t$ in $G'_R$ if and only if its reverse is on a new shortest path from $t$ to $v$ in $G'$. These edges on new shortest paths are the ones we need to keep track of in order to update the reverse DAGs, and to facilitate this we define a collection of sets $R_t$, $t \in V$. The set $R_t$ is the set of (reversed) outgoing edges from $t$ in DAG$'(t)$ that lie on a shortest path from $t$ to $v$ in $G'$ (see also Eqn. 2.3 in the next section). Thus, if a new shortest path $\pi_{sb}$ is present in DAG$'_R(s)$ ($\pi_{sb}$ must pass through $v$), its last edge $(a, b)$ is present in $R_b$. Using the sets $R_t, \forall\, t \in V$, it is possible to quickly build the DAG$'_R(t)$ after Phase 1 as shown in part (D) in section 2.3.2.

**Step 2:** After applying Phase 1 and 2 on the initial DAGs using $E_i$ to obtain the DAG$'_R(s)$ and $G'_R$, Step 2 re-applies Phase 1 and Phase 2 on these updated graphs using $E_o$ in order to complete all of the updates to vertex $v$. We can then apply Alg. 2 to the DAG$''(s)$ to obtain the BC scores for the updated graph $G''$.

### 2.3.2 Vertex Update Algorithm

We now give details of each phase of our algorithm starting with the graph $G$.

**Step 1, Phase 1**

(A) **Compute $d'(s,v)$ and $\sigma'_{sv}$ for any $s$.** We show how to compute in $G'$ the distance and number of shortest paths to $v$ from any $s$. Let $(u_j, v) \in E_i(v)$ and let $D_j(s,v) = d(s,u_j) + \mathbf{w}'(u_j, v)$. Since the updates on edges in $E_i(v)$ are decrease-only, it follows that:

$$d'(s,v) = \min\{d(s,v), \min_{j:(u_j,v)\in E_i(v)} \{D_j(s,v)\}\} \tag{2.1}$$

Further, if $d'(s,v) = d(s,v)$, we define:

$$\widehat{\sigma}'_{sv} = |\{\pi'_{sv} : \pi'_{sv} \text{ is a shortest path in } G' \text{ and } \pi'_{sv} \text{ uses } e \in E_i(v)\}| \tag{2.2}$$

We also need to compute $\sigma'_{sv}$, the number of shortest paths from $s$ to $v$ in $G'$. It is straightforward to compute $d'(s,v)$, $\sigma'_{sv}$, and $\widehat{\sigma}'_{sv}$ in $O(|E_i(v)|)$ time. Alg. 5 gives the details of this step.

**Algorithm 5** Dist-to-$v$ $(s, E_i(v))$

```
Require: $E_i(v)$ with updated weights $\mathbf{w}'$.
    $d(s,t)$ and $\sigma_{st}$, $\forall\ s,t \in V$.
Ensure: $d'(s,v), \sigma'_{sv}, \widehat{\sigma}'_{sv}$.
1: $\hat{\sigma}'_{sv} \leftarrow 0$, $\sigma'_{sv} \leftarrow \sigma_{sv}$, $D' \leftarrow d(s,v)$.
2: for each edge $(u_i, v) \in E_i(v)$ do
3:     if $D' = d(s,u_i) + \mathbf{w}'(u_i, v)$ then
4:         $\sigma'_{sv} \leftarrow \sigma'_{sv} + \sigma_{su_i}$.
5:         $\hat{\sigma}'_{sv} \leftarrow \hat{\sigma}'_{sv} + \sigma_{su_i}$.
6:     else if $D' > d(s,u_i) + \mathbf{w}'(u_i, v)$ then
7:         $D' \leftarrow d(s,u_i) + \mathbf{w}'(u_i, v)$.
8:         $\sigma'_{sv} \leftarrow \sigma_{su_i}$.
9: $d'(s,v) \leftarrow D'$.
```

**Algorithm 6** Upd-Rev-DAG($s$, $E_i(v)$)

```
Require: $\text{DAG}_R(s)$; $R_t, flag(s,t), \forall t \in V$.
Ensure: An edge set $X$ after update on edges in $E_i(v)$.
1: $X \leftarrow \emptyset$.
2: for each edge $(a,b) \in \text{DAG}_R(s)$ do
3:     if $flag(b,s) =$ UN-changed or $flag(b,s) =$ NUM-changed then
4:         $X \leftarrow X \cup (a,b)$ .
5: for each $b \in V \setminus \{s\}$ do
6:     if $flag(b,s) =$ NUM-changed or $flag(b,s) =$ WT-changed then
7:         $X \leftarrow X \cup R_b$ .
```

(B) **Compute $d'(s,t)$ and $\sigma'(s,t)$ for all $s,t$.** After computing $d'(s,v), \sigma'_{sv}$ and $\hat{\sigma}'_{sv}$, we show that the values $d'(s,t)$ and $\sigma'(s,t)$ can be computed efficiently. We state Lemma 4 which captures this computation. The proof of this lemma is similar to Lemma 2 in the edge update case.

**Lemma 4.** *Let $E_i(v)$ be the set of updated edges incoming to $v$. Let $G'$ be the graph obtained by applying the updates in $E_i(v)$ to $G$. For any $s \in V$ and $t \in V \setminus \{v\}$, let $D(s,t) = d'(s,v) + d(v,t)$, $\Sigma_{st} = \sigma_{st} + \widehat{\sigma}'_{sv} \cdot \sigma_{vt}$, $\Sigma'_{st} = \sigma_{st} + \sigma'_{sv} \cdot \sigma_{vt}$.*

1. *If $d(s,t) < D(s,t)$, then $d'(s,t) = d(s,t)$ and $\sigma'_{st} = \sigma_{st}$.*
2. *If $d(s,t) = D(s,t)$ and $d(s,v) = d'(s,v)$, then $d'(s,t) = d(s,t)$ and $\sigma'_{st} = \Sigma_{st}$.*
3. *If $d(s,t) = D(s,t)$ and $d(s,v) > d'(s,v)$, then $d'(s,t) = d(s,t)$ and $\sigma'_{st} = \Sigma'_{st}$.*
4. *If $d(s,t) > D(s,t)$, then $d'(s,t) = D(s,t)$ and $\sigma'_{st} = \sigma'_{sv} \cdot \sigma_{vt}$.*

The value $flag(s,t)$ for every $s,t$ can be computed using the updated distances and number of shortest paths ($flag(s,t)$ is UN-changed for 1, NUM-changed for both 2 and 3, and WT-changed for 4, in Lemma 4).

(C) **Compute DAG$'(s)$ for every $s$.** Given $d'(s,t)$ and $\sigma'(s,t)$ updated for all

$s, t \in V$, the algorithm to compute $\mathrm{DAG}'(s)$ for any $s \in V$ is similar to Alg. 3 in the edge update case. The only modification we need is in Steps 9–10 where instead of a single edge $(u, v)$, we consider every edge $(u_1, v) \in E_i(v)$.

**Step 1, Phase 2**

(D) **Compute $\mathbf{DAG}'_R(s)$ for every $s$.** We update $\mathrm{DAG}_R(s)$, for every $s$, for which we use Alg. 6. Recall the sets $R_t, \forall\, t \in V$ defined as:

$$R_t = \{(a, t) \mid (t, a) \in \mathrm{DAG}'(t) \text{ and } \mathbf{w}'(t, a) + d'(a, v) = d'(t, v)\} \tag{2.3}$$

The set $R_t$ is the set of (reversed) outgoing edges from $t$ in $\mathrm{DAG}'(t)$ that lie on a shortest path from $t$ to $v$ in $G'$. Consider an edge $e = (a, b)$ in the updated $\mathrm{DAG}'_R(s)$. If $e$ is in $\mathrm{DAG}_R(s)$, it is added to $\mathrm{DAG}'_R(s)$ by Steps 2–4. If $e$ lies on a new shortest path present only in $G'_R$, its reverse must also lie on a shortest path that goes through $v$ in $G'$, and it will be added to $\mathrm{DAG}'_R(s)$ by the $R_b$ during Steps 5–7 ($R_b$ could also contain edges on old shortest paths through $v$ already processed in Steps 2–4, but even in that case each edge is added to $\mathrm{DAG}'_R(s)$ at most twice by Alg. 6). Note that we do not need to process edges $(u_j, v)$ in $E_i$ separately (as with edge $(u, v)$ in Alg. 2), because these edges will be present in the relevant $R_{u_j}$. The correctness of Alg. 6 follows from Lemma 5, whose proof is similar to Lemma 3, and is omitted.

**Lemma 5.** *In Alg. 6, an edge $(a, b)$ is placed in $X$ if and only if $(a, b) \in DAG'_R(s)$ after the decrease-only update of the set $E_i(v)$.*

**Step 2:** To process the updates in $E_o(v)$, we re-apply Phase 1 and 2 over $G'_R$. Since we are processing incoming edges in $G'_R$, our earlier steps apply unchanged, and we obtain modified values for $d''(\cdot)$, $\sigma''_{(\cdot)}$, and $\mathrm{DAG}''_R(s)$ for every $s$. Then, using Alg. 6 we obtain the $\mathrm{DAG}''(s)$ for every $s$. Finally, to compute the updated BC scores, we apply Alg. 2.

**Performance:** Computing $d'(s,v), \sigma'_{sv}$ and $\hat{\sigma}'_{sv}$ requires time $O(|E_i(v)|) = O(n)$ for each $s$, and hence $O(n^2)$ time for all sources. Applying Lemma 4 to all pairs of vertices takes time $O(n^2)$. The complexity of modified Alg. 3 applied to all DAGs is again $O(\nu^* \cdot n)$. Creating set $R_t$ requires at most $O(E^* \cap \{\text{outgoing edges of } t\})$, so the overall complexity for all the sets is $O(m^*)$. Finally, we bound the complexity of Algorithm 6: the algorithm adds $(a,b)$ in a reverse DAG edge set $X$ at most twice. Since $\sum_{s \in V} |E(\text{DAG}'(s))| = \sum_{s \in V} |E(\text{DAG}'_R(s))|$, at most $O(\nu^* \cdot n)$ edges can be inserted into all the sets $X$ when Algorithm 6 is executed over all sources. Finally, applying the updates in $E_o(v)$ requires a symmetric procedure starting from the reverse DAGs, the final complexity bound of $O(\nu^* \cdot n)$ follows.

## 2.4 Static betweenness centrality

In this section we present static algorithms that compute betweenness centrality faster than the Brandes algorithm. In Brandes' algorithm, the computation of the predecessors list is completed during the construction of each SSSP dag using Dijkstra. Our main idea is to decouple these two components by computing first the set of edges on shortest paths $E^*$, together with all the distances in the graph (using an APSP algorithm); and then rebuild all the SSSP dags using $E^*$ to efficiently compute number of paths and predecessor lists. With this technique, we can use faster APSP algorithms in the initial phase, obtaining a provable speed-up over the classic Brandes algorithm.

We first consider an algorithm that is based on the Hidden Paths algorithm by Karger et al. [KKP93] together with Brandes' accumulation technique. The Hidden Path algorithm runs Dijkstra's SSSP in parallel from each vertex. It identifies all pairs shortest paths while only examining the edges in $E^*$, the set of edges that actually lie on some shortest path. A similar algorithm with the same running time of $O(m^*n + n^2 \log n)$ was developed independently by McGeoch [McG95]; here $m^* = |E^*|$.

Our Static-BC algorithm is presented as Algorithm 7. In Step 1 we run the Hidden Paths algorithm to compute $E^*$ as well as the shortest path distances for every pair of vertices. This is the step with the dominant cost, while in Steps 2–8 the complexity is strictly related to the size of $E^*$. In Steps 2–4, we identify the edges in each shortest path DAG and in each $P_s(v)$: for every edge $(u, v) \in E^*$, if $d(s, u) + \mathbf{w}(u, v) = d(s, v)$, then we add the edge $(u, v)$ to DAG$(s)$ and the vertex $u$ to $P_s(v)$. The overall time spent for constructing the DAGs and the predecessor lists is bounded by $O(m^*n)$. Step 7 counts the number of shortest paths from $s$ to $v$ for all $v \in V$, by traversing DAG$(s)$ according to the topological order of its vertices, maintained in the double-ended queue $Q$ (created in Step 6) used as a queue. We accumulate the path counts for a vertex $v$ according to the formula $\sigma_{sv} = \sum_{(u,v)\in \mathrm{DAG}(s)} \sigma_{su}$. This takes time linear in the size of DAG$(s)$. Therefore, across all sources, we spend time which is bounded by $O(n^2 + \sum_{s\in V} m^*_s) = O(\bar{m}^*n + n^2)$. Finally in Step 8, using $Q$ as a stack (reverse topological order), we call Accumulate-dependency$(s, Q)$ (Algorithm 2) to accumulate dependencies. Thus the overall running time of this static BC algorithm is $O(m^*n + n^2 \log n)$. The correctness of the algorithm follows from the correctness of the Hidden Paths algorithm and the Brandes' accumulation technique.

**Algorithm 7** Static-BC$(G = (V, E))$

```
1: Using an APSP algorithm, compute $E^*$, and $d(s,t)$ for every $s, t \in V$
2: **for** each node $s \in V$ **do**
3:     **for** each $(u, v) \in E^*$ **do**
4:         **if** $d(s, u) + \mathbf{w}(u, v) = d(s, v)$ **then** add $(u, v)$ to DAG$(s)$ and $u$ to $P_s(v)$
5: **for** each DAG$(s)$ **do**
6:     compute a dequeue $Q$ containing the nodes of DAG$(s)$ in topological order
7:     for all $v \in V$, compute $\sigma_{sv} = \sum_{(u,v)\in \mathrm{DAG}(s)} \sigma_{su}$ by accumulating path counts
       on vertices extracted from $Q$ in queue order (topological order)
8:     Accumulate-dependency$(s, Q)$, using $Q$ in stack order    //use Algorithm 2
```

Algorithm 7 can be expected to run faster than Brandes' on many graphs since $m^*$ is often much smaller than $m$. However, its worst-case running time is asymptotically the same as Brandes'. We observe that if we replace the Hidden Paths algorithm in Step 1 of Algorithm 7 with any other APSP algorithm that identifies a set of edges $E' \supseteq E^*$, we can use $E'$ in place of $E$ in Step 3, and obtain a correct static BC algorithm that runs in time $O(m'n + n^2 + T')$, where $m' = |E'|$ and $T'$ is the running time of the APSP algorithm used in Step 1. In particular, if we use one of the faster APSP algorithms for positive real-weighted graphs (Pettie [Pet04] for directed graphs or [PR05] for undirected graphs) in Step 1, we can obtain asymptotically faster BC algorithms than Brandes' by using $E' = E$. With Pettie's algorithm [Pet04], we obtain an $O(mn + n^2 \log\log n)$ time algorithm for static betweenness centrality in directed graphs, and with the algorithm of Pettie and Ramachandran [PR05], we obtain a static betweenness centrality algorithm for undirected graphs that runs in $O(mn \cdot \log \alpha(m, n))$, where $\alpha$ is an inverse-Ackermann function.

# Chapter 3

# Increase-Only Algorithm

In this chapter we present an increase-only algorithm for the APASP problem, where each update in $G$ either deletes or increases the weight of some edges incident on a vertex[1]. Our method is a generalization of the method developed by Demetrescu and Italiano [DI04] (the 'DI' method) for increase-only APSP where only one shortest path is needed. The DI increase-only algorithm [DI04] runs in $O(n^2 \cdot \log n)$ amortized time per update, for a sufficiently long update sequence. This increase-only algorithm is also extended to a fully dynamic algorithm in [DI04] that runs in $O(n^2 \cdot \log^3 n)$ time, and this result was improved to $O(n^2 \cdot \log^2 n)$ amortized time by Thorup [Tho04]; both algorithms have within them essentially the same increase-only algorithm.

In [DI04, Tho04] the goal was to compute all pairs shortest path distances, and hence these algorithms preprocess the graph in order to have a unique shortest path between every pair of vertices. The unique shortest paths assumption, although not restrictive in their case, is crucial to the correctness and time complexity of their algorithms. We are interested in the more general problem of APASP, and this poses several challenges in generalizing the approach in [DI04].

[1]The results presented in this chapter appeared in [NPR14b].

In addition to APASP, our method gives an increase-only algorithm for computing BC (Section 5.1 in Chapter 5 explains how to use the data structures introduced in this chapter to compute BC scores).

**Locally Shortest Paths (LSPs).** For a path $\pi_{xy} \in G$, we define the $\pi_{xy}$ *distance* from $x$ to $y$ as $\mathbf{w}(\pi_{xy}) = \sum_{e \in \pi_{xy}} \mathbf{w}(e)$, and the $\pi_{xy}$ *length* from $x$ to $y$ as the number of edges on $\pi_{xy}$. For any $x, y \in V$, $d(x, y)$ denotes the shortest path distance from $x$ to $y$ in $G$. A path $\pi_{xy}$ in $G$ is a *locally shortest path (LSP)* [DI04] if either $\pi_{xy}$ contains a single vertex, or every proper subpath of $\pi_{xy}$ is a shortest path in $G$. As noted in [DI04], every shortest path (SP) is an LSP, but an LSP need not be an SP (e.g., every single edge is an LSP).

The DI method maintains all LSPs in a graph with unique shortest paths, and these are key to efficiently maintaining shortest paths under increase-only and fully dynamic updates. The increase-only method we present here maintains all LSPs for all (multiple) shortest paths in a graph, using a compact *tuple* representation.

In Chapter 2, we gave a simple decrease-only BC algorithm [NPR14a], that provably improves on Brandes' on sparse graphs, and also typically improves on Brandes' in dense graphs (e.g., in the setting of Theorem 3 below). In this chapter, we complement the results in Chapter 2; however, increase-only updates are considerably more challenging (similar to APSP, as noted in [DI04]).

The key step in the decrease-only BC algorithm (Chapter 2) is the decrease-only maintenance of the APASP dags (achieved there using techniques unrelated to this thesis). After the updated dags are obtained, the BC scores can be computed in time linear in the combined sizes of the APASP dags (plus $O(n^2)$). Thus, if we instead use our increase-only APASP algorithm in the key step in [NPR14a], we obtain an increase-only algorithm for BC with the same bound as APASP.

**Our Results.** Recall the definitions of $m^*$ and $\nu^*$ in Section 1.1.1, Chapter 1. Our main result is the following theorem, where we have assumed that $\nu^* = \Omega(n)$.

**Theorem 2.** *Let $\Sigma$ be a sequence of increase-only updates on $G = (V, E)$. Then, all SP dags, all LSPs, and all BC scores can be maintained in amortized time $O(\nu^{*2} \cdot \log n)$ per update when $|\Sigma| = \Omega(m^*/\nu^*)$.*

In many real graphs $\nu^*$ behaves as discussed in Section 1.1.1, Chapter 1. Thus we have:

**Theorem 3.** *Let $\Sigma$ be a sequence of increase-only updates on graphs where the number of edges on shortest paths through any single vertex is $O(n)$. Then, all SP dags, all LSPs, and all BC scores can be maintained in amortized time $O(n^2 \cdot \log n)$ per update when $|\Sigma| = \Omega(m^*/n)$.*

**Corollary 1.** *If the number of shortest paths for any vertex pair is bounded by a constant, then increase-only APASP, LSPs, and BC have amortized cost $O(n^2 \cdot \log n)$ per update when the update sequence has length $\Omega(m^*/n)$.*

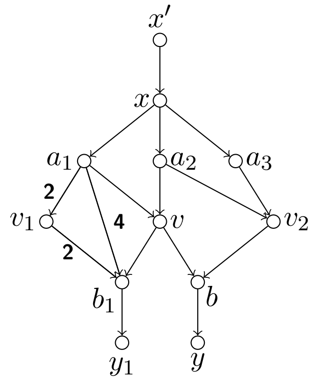


Figure 3.1: Graph $G$

| Set | $G$ (before update on $v$) |
|---|---|
| $P(x,y)$ $= P^*(x,y)$ | $\{((xa_1, by), 4, 1), ((xa_2, by), 4, 2),$ $((xa_3, by), 4, 1)\}$ |
| $P(x, b_1)$ | $\{(xa_1, vb_1), 3, 1), ((xa_2, vb_1), 3, 1)\}$ |
| $P^*(x, b_1)$ | $\{((xa_1, vb_1), 3, 1), ((xa_2, vb_1), 3, 1)\}$ |
| $L^*(v, y_1)$ | $\{a_1, a_2\}$ |
| $L(v, b_1y_1)$ | $\{a_1, a_2\}$ |
| $R^*(x, v)$ | $\{b, b_1\}$ |
| $R(xa_2, v)$ | $\{b, b_1\}$ |

Figure 3.2: A subset of the tuple-system for $G$ in Fig. 3.1

**The DI method.** Here we will use an example to give a quick review of the DI approach [DI04], which forms the basis for our method. Consider the graph $G$ in Fig. 3.1, where all edges have weight 1 except for the ones with explicit weights.

As in DI, let us assume here that $G$ has been pre-processed to identify a unique shortest path between every pair of vertices. In $G$ the shortest path from $a_1$ to $b_1$ is $\langle a_1, v, b_1 \rangle$ and has weight 2, and by definition, the paths $p_1 = \langle a_1, b_1 \rangle$ and $p_2 = \langle a_1, v_1, b_1 \rangle$ of weight 4 are both LSPs. Now consider an increase-only update on $v$ that increases $\mathbf{w}(a_1, v)$ to 10 and $\mathbf{w}(a_2, v)$ to 5, and let $G'$ be the resulting graph (see Fig. 3.3). In $G'$ both $p_1$ and $p_2$ become shortest paths. Furthermore, a *left extension* of the path $p_1$, namely $p_3 = \langle x, a_1, b_1 \rangle$ becomes a shortest path from $x$ to $b_1$ in $G'$. Note that the path $p_3$ is not even an LSP in the graph $G$; however, it is obtained as a left extension of a path that has become shortest after the update.

The elegant method of storing LSPs and creating longer LSPs by left and right extending shortest paths is the basis of the DI approach [DI04]. To achieve this, the DI approach uses a succinct representation of SPs, LSPs and their left and right extensions using suitable data structures. It then uses a procedure *cleanup* to remove from the data structures all the shortest paths and LSPs that contain the updated vertex $v$, and a complementary procedure *fixup* that first adds all the trivial LSPs (corresponding to edges incident on $v$), and then restores the shortest paths and LSPs between all pairs of vertices. The DI approach thus efficiently maintains a single shortest path between all pairs of vertices under increase-only updates.

**Organization.** In Section 3.1 we present a new *tuple system* which succinctly represents all LSPs in a graph with multiple shortest paths and in Section 3.2 we present our increase-only algorithm for maintaining this tuple system, and hence for maintaining APASP.

## 3.1 A System of Tuples

In this section we present an efficient representation of the set of SPs and LSPs for an edge weighted graph $G = (V, E)$. We first define the notions of *tuple* and *triple.*

**Tuple.** A tuple, $\tau = (xa, by)$, represents the set of LSPs in $G$, all of which use the same first edge $(x, a)$ and the same last edge $(b, y)$. The weight of every path represented by $\tau$ is $\mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$. We call $\tau$ a *locally shortest path tuple (LST)*. In addition, if $d(x, y) = \mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$, then $\tau$ is a *shortest path tuple (ST)*. Fig. 3.5(a) shows a tuple $\tau$.

**Triple.** A triple $\gamma = (\tau, wt, count)$, represents the tuple $\tau = (xa, by)$ that contains $count > 0$ number of paths from $x$ to $y$, each with weight $wt$. In Fig. 3.1, the triple $((xa_2, by), 4, 2)$ represents two paths from $x$ to $y$, namely $p_1 = \langle x, a_2, v, b, y\rangle$ and $p_2 = \langle x, a_2, v_2, b, y\rangle$ both having weight 4.

**Storing Locally Shortest Paths.** We use triples to succinctly store all LSPs and SPs for each vertex pair in $G$. For $x, y \in V$, we define:

$$P(x,y) = \{((xa, by), wt, count)\colon (xa, by) \text{ is an LST from } x \text{ to } y \text{ in } G\}$$
$$P^*(x,y) = \{((xa, by), wt, count)\colon (xa, by) \text{ is an ST from } x \text{ to } y \text{ in } G\}.$$

Note that all triples in $P^*(x, y)$ have the same weight. We will use the term LST to denote either a locally shortest tuple or a triple representing a set of LSPs, and it will be clear from the context whether we mean a triple or a tuple.

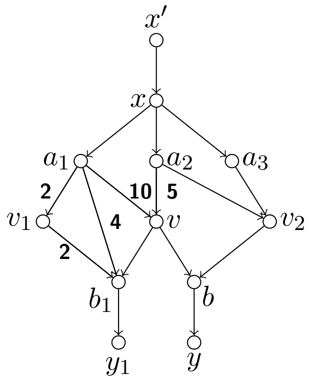
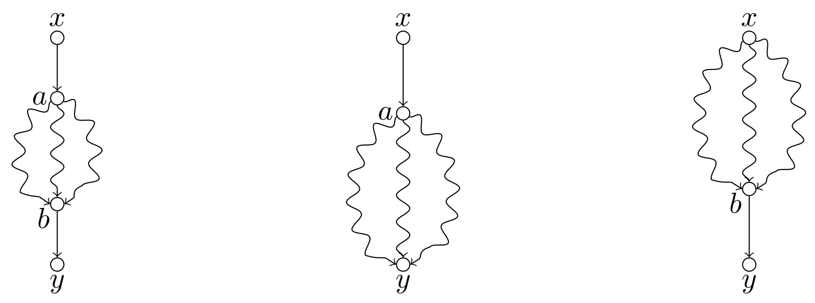


Figure 3.3: Graph $G'$

| Set | $G'$ (with $\mathbf{w}(a_1, v) = 10$, $\mathbf{w}(a_2, v) = 5$) |
|---|---|
| $P(x,y) = P^*(x,y)$ | $\{((xa_2, by), 4, 1), ((xa_3, by), 4, 1)\}$ |
| $P(x, b_1)$ | $\{((xa_1, v_1b_1), 5, 1), ((xa_2, vb_1), 7, 1), ((xa_1, a_1b_1), 5, 1)\}$ |
| $P^*(x, b_1)$ | $\{((xa_1, v_1b_1), 5, 1), ((xa_1, a_1b_1), 5, 1)\}$ |
| $L^*(v, y_1)$ | $\{a_2\}$ |
| $L(v, b_1y_1)$ | $\{a_2\}$ |
| $R^*(x, v)$ | $\emptyset$ |
| $R(xa_2, v)$ | $\{b_1\}$ |

Figure 3.4: A subset of the tuple-system for $G'$



(a) tuple $\tau = (xa, by)$ (b) $\ell$-tuple $\tau_\ell = (xa, y)$ (c) $r$-tuple $\tau_r = (x, by)$

Figure 3.5: Tuples

**Left Tuple and Right Tuple.** A left tuple (or $\ell$-tuple), $\tau_\ell = (xa, y)$, represents the set of LSPs from $x$ to $y$, all of which use the same first edge $(x, a)$. The weight of every path represented by $\tau_\ell$ is $\mathbf{w}(x, a) + d(a, y)$. If $d(x, y) = \mathbf{w}(x, a) + d(a, y)$, then $\tau_\ell$ represents the set of shortest paths from $x$ to $y$, all of which use the first edge $(x, a)$. A right tuple ($r$-tuple) $\tau_r = (x, by)$ is defined analogously. Fig. 3.5(b) and Fig. 3.5(c) show a left tuple and a right tuple respectively. In the following, we will say that a tuple (or $\ell$-tuple or $r$-tuple) *contains* a vertex $v$, if at least one of the paths represented by the tuple contains $v$. For instance, in Fig. 3.1, the tuple $(xa_2, by)$ contains the vertex $v$ as well as the vertex $v_2$.

**ST and LST Extensions.** For a shortest path $r$-tuple $\tau_r = (x, by)$, we define $L(\tau_r)$ to be the set of vertices which can be used as pre-extensions to create LSTs in $G$. Similarly, for a shortest path $\ell$-tuple $\tau_\ell = (xa, y)$, $R(\tau_\ell)$ is the set of vertices which can be used as post-extensions to create LSTs in $G$. We do not define $R(\tau_r)$ and $L(\tau_\ell)$. So we have:

$$\begin{aligned} L(x, by) &= \{x' : (x', x) \in E(G) \text{ and } (x'x, by) \text{ is an LST in } G\} \\ R(xa, y) &= \{y' : (y, y') \in E(G) \text{ and } (xa, yy') \text{ is an LST in } G\}. \end{aligned}$$

For $x, y \in V$, $L^*(x, y)$ denotes the set of vertices which can be used as pre-extensions to create shortest path tuples in $G$; $R^*(x,y)$ is defined symmetrically:

$$\begin{aligned} L^*(x,y) &= \{x' : (x',x) \in E(G) \text{ and } (x'x,y) \text{ is a } \ell\text{-tuple representing SPs in } G\} \\ R^*(x,y) &= \{y' : (y,y') \in E(G) \text{ and } (x,yy') \text{ is an } r\text{-tuple representing SPs in } G\}. \end{aligned}$$

Fig. 3.2 shows a subset of these sets for the graph $G$ in Fig. 3.1.

**Key Deviations from DI [DI04].** The assumption of unique shortest paths in [DI04] ensures that $\tau = (xa, by)$, $\tau_\ell = (xa, y)$, and $\tau_r = (x, by)$ all represent exactly the same (single) locally shortest path. However, in our case, the set of paths represented by $\tau_\ell$ and $\tau_r$ can be different, and $\tau$ is a subset of paths represented by $\tau_\ell$ and $\tau_r$. Our definitions of ST and LST extensions are derived from the analogous definitions in [DI04] for SP and LSP extensions of paths. For a path $\pi = x \to a \rightsquigarrow b \to y$, DI defines sets $L$, $L^*$, $R$ and $R^*$. In our case, the analog of a path $\pi = x \to a \rightsquigarrow b \to y$ is a tuple $\tau = (xa, by)$, but to obtain efficiency, we define the set $L$ only for an $r$-tuple and the set $R$ only for an $\ell$-tuple. Furthermore, we define $L^*$ and $R^*$ for each pair of vertices.

In the following two lemmas we bound the total number of tuples in the graph and the total number of tuples that contain a given vertex $v$. These bounds also apply to the number of triples since there is exactly one triple for each tuple in our tuple-system.

**Lemma 6.** *The number of LSTs in $G = (V, E)$ is bounded by $O(m^* \cdot \nu^*)$.*

*Proof.* For any LST $(\times a, \times\times)$, for some $a \in V$, the first and last edge of any such tuple must lie on a shortest path containing $a$. Let $E^*_a$ denote the set of edges that lie on shortest paths through $a$, and let $I_a$ be the set of incoming edges to $a$. Then, there are at most $\nu^*$ ways of choosing the last edge in $(\times a, \times\times)$ and at most $E^*_a \cap I_a$ ways of choosing the first edge in $(\times a, \times\times)$. Since $\sum_{a \in V} |E^*_a \cap I_a| = m^*$, the number of LSTs in $G$ is at most $\sum_{a\in V} \nu^* \cdot |E^*_a \cap I_a| \leq m^* \cdot \nu^*$. □

**Lemma 7.** *The number of LSTs that contain a vertex $v$ is $O(\nu^{*2})$.*

*Proof.* We distinguish three different cases:

1. Tuples starting with $v$: for a tuple that starts with edge $(v, a)$, the last edge must lie on $a$'s SP dag, so there are at most $\nu^*$ choices for the last edge. Hence, the number of tuples with $v$ as start vertex is at most $\sum_{a \in V \setminus v} \nu^* \leq n \cdot \nu^*$.

2. Similarly, the number of tuples with $v$ as end vertex is at most $n \cdot \nu^*$.

3. For any tuple $\tau = (xa, by)$ that contains $v$ as an internal vertex, both $(x, a)$ and $(b, y)$ lie on a shortest path through $v$, hence the number of such tuples is at most $\nu^{*2}$. □

## 3.2 Increase-Only Algorithm

Here we present our increase-only APASP algorithm. Recall that an increase-only update on a vertex $v$ either deletes or increases the weights of a subset of edges incident on $v$. We begin with the data structures we use.

**Data Structures.** For every $x, y$, $x \neq y$ in $V$, we maintain the following:

1. $P(x, y)$ – a priority queue containing LSTs from $x$ to $y$ with weight as key.

2. $P^*(x, y)$ – a priority queue containing STs from $x$ to $y$ with weight as key.

3. $L^*(x, y)$ – a balanced search tree containing vertices with vertex ID as key.

4. $R^*(x, y)$ – a balanced search tree containing vertices with vertex ID as key.

For every $\ell$-tuple we have its right extension, and for every $r$-tuple its left extension. These sets are stored as balanced search trees (BSTs) with the vertex ID as a key. Additionally, we maintain all tuples in a BST *dict*, with a tuple $\tau = (xa, by)$ having key $[x, y, a, b]$. We also maintain pointers from $\tau$ to $R(xa, y)$ and $L(x, by)$, and to the corresponding triple containing $\tau$ in $P(x, y)$, (and in $P^*(x, y)$ if $(xa, by)$ is an

ST). Finally, we maintain a sub-dictionary of *dict* called Marked-Tuples (explained below). Marked-Tuples, unlike the other data structures, is specific only to one update.

**The Increase-Only Algorithm.** Given the updated vertex $v$ and the updated weight function $\mathbf{w}'$ over all the incoming and outgoing edges of $v$, the increase-only algorithm performs two main steps *cleanup* and *fixup*, as in DI. The cleanup procedure removes from the tuple-system every LSP that contains the updated vertex $v$. The following definition of a *new* LSP is from DI [DI04].

**Definition 1.** *A path that is shortest (locally shortest) after an update to vertex $v$ is* new *if either it was not an SP (LSP) before the update, or it contains $v$.*

The fixup procedure adds to the tuple-system all the *new* shortest and locally shortest paths. In contrast to DI, recall that we store locally shortest paths in $P$ and $P^*$ as triples. Hence removing or adding paths implies decrementing or incrementing the count in the relevant triple; thus a triple is removed or added only if its count goes down to zero or up from zero. Moreover, new tuples may be created through combining several existing tuples. Some of the updated data structures for the graph $G'$ in Fig. 3.3, obtained after an increase-only update on $v$ in the graph $G$ in Fig. 3.1, are schematized in Fig. 3.4.

### 3.2.1 The Cleanup Procedure

Algorithm 8 (cleanup) uses an initially empty heap $H_c$ of triples. It also initializes the empty dictionary Marked-Tuples. The algorithm then creates the trivial triple corresponding to the vertex $v$ and adds it to $H_c$ (Step 2, Algorithm 8). For a triple $((xa, by), wt, count)$ the key in $H_c$ is $[wt, x, y]$. The algorithm repeatedly extracts min-key triples from $H_c$ (Step 4, Algorithm 8) and *processes* them. The processing of triples involves left-extending (Steps 5–17, Algorithm 8) and right-extending triples (Step 18, Algorithm 8) and removing from the tuple system the set of LSPs thus formed. This is similar to cleanup in DI. However, since we deal with a set of paths instead of a single path, we need significant modifications, of which we now highlight two: (i) Accumulation used in Step 4 and (ii) use of Marked-Tuples in Step 7 and Step 11.

**Algorithm 8** cleanup($v$)

```
1: $H_c \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2: $\gamma \leftarrow ((vv, vv), 0, 1)$; add $\gamma$ to $H_c$
3: while $H_c \neq \emptyset$ do
4:    extract in $S$ all the triples with min-key $[wt, x, y]$ from $H_c$
5:    for every $b$ such that $(x\times, by) \in S$ do
6:       let $fcount' = \sum_i ct_i$ such that $((xa_i, by), wt, ct_i) \in S$
7:       for every $x' \in L(x, by)$ such that $(x'x, by) \notin$ Marked-Tuples do
8:          $wt' \leftarrow wt + \mathbf{w}(x', x)$; $\gamma' \leftarrow ((x'x, by), wt', fcount')$; add $\gamma'$ to $H_c$
9:          remove $\gamma'$ in $P(x', y)$ // decrements $count$ by $fcount$
10:         if a triple for $(x'x, by)$ exists in $P(x', y)$ then
11:            insert $(x'x, by)$ in Marked-Tuples
12:         else
13:            delete $x'$ from $L(x, by)$ and delete $y$ from $R(x'x, b)$
14:         if a triple for $(x'x, by)$ exists in $P^*(x', y)$ then
15:            remove $\gamma'$ in $P^*(x', y)$ // decrements $count$ by $fcount$
16:            if $P^*(x, y) = \emptyset$ then delete $x'$ from $L^*(x, y)$
17:            if $P^*(x', b) = \emptyset$ then delete $y$ from $R^*(x', b)$
18:   perform symmetric steps 5 – 17 for right extensions
```

### 3.2.2 Accumulation

In Step 4 we extract a collection $S$ of triples all with key $[wt, x, y]$ from $H_c$ and process them together in that iteration of the while loop. Assume that for a fixed last edge $(b, y)$, $S$ contains triples of the form $(xa_t, by)$, for $t = 1, \ldots, k$. Our algorithm processes and left-extends all these triples with the same last edge together. This ensures that, for any $x' \in L(x, by)$, we generate the triple $(x'x, by)$ exactly once. The accumulation is correct because any valid left extension for a triple $(xa_i, by)$ is also a valid left extension for $(xa_j, by)$ when both triples have the same weight.

**Need for accumulation.** In Step 5 of Algorithm 8, we consider every $b$ such that $(x\times, by)$ belongs to $S$. We also assume that we have the accumulated count of such triples available in Step 6. An efficient method to accumulate these counts is given below. We use this accumulated count to generate a longer LST for each $x' \in L(x, by)$ (Step 8). (For the moment, ignore the check of a tuple being present in Marked-Tuples.) Consider our example in Fig. 3.1 where after the increase-only update on $v$, we intend to remove from the tuple system the following two paths passing through $v$ namely (i) $p_1 = \langle x'x, a_1, v, b, y\rangle$ and (ii) $p_2 = \langle x', x, a_2, v, b, y\rangle$. Note that both these paths represented by triples of the form $(x'x, by)$. We further remark that $p_1$ can be generated by left extending the triple $((xa_1, by), 4, 1)$ whereas $p_2$ can be generated by left extending the triple $((xa_2, by), 4, 1)$. However, instead of left-extending each triple individually, our algorithm accumulates the count to obtain 2 paths represented by the $r$-tuple $(x, by)$ and then generates the triple $((x'x, by), 5, 1)$. We note that such an implementation is correct because any valid left extension of triples of the form $(xa_1, by)$ is also a valid left extension of triples of the form $(xa_2, by)$ when the triples have the same weight. Furthermore, it is efficient since it generates the triple of the form $(x'x, by)$ at most once. This is the precise reason for defining the set $L$ for an $r$-tuple $(x, by)$ instead of defining it for the tuple $(xa_1, by)$.

**Accumulation technique.** An efficient implementation of getting accumulated counts can be achieved in several ways. For the sake of concreteness, we sketch an implementation by maintaining two arrays $A$ and $B$ of size $n$ each and two linked lists $L_a$ and $L_b$. Assume that the arrays are initialized to zero and the linked lists are empty just before any triple with key $[wt, x, y]$ is extracted from the heap. When a triple $\gamma = ((xa_i, b_jy), wt, count_{ij})$ is extracted from $H_c$, we add $count_{ij}$ to $A[a_i]$ and $B[b_j]$. The lists $L_a$ and $L_b$ maintain pointers to non-zero locations in the arrays

$A$ and $B$ respectively. Thus, when all triples of weight $wt$ corresponding to tuples of the form $(x\times, \times y)$ are extracted from $H_c$, the value in $A[a_i]$ denotes the number of locally shortest paths of the form $(xa_i, \times y)$ to be updated. Similarly, the value in $B[b_j]$ denotes number of locally shortest paths of the form $(x\times, b_jy)$ to be updated. Using the lists $L_a$ and $L_b$, we can efficiently access the accumulated counts as well as reinitialize (to zero) all the non-zero values in the two arrays $A$ and $B$.

### 3.2.3 Need for Marked-Tuples

Consider the example in Fig. 3.1 and assume that we have deleted the two paths of the form $(x'x, by)$ which pass through $v$. Furthermore, assume that we have generated them via left extending the two triples of the form $(xa_1, by)$ and $(xa_2, by)$. Now note that since path $\langle x', x, a_2, v_2, b, y\rangle$ continues to exist in the tuple system, $x' \in L(x, by)$ and $y \in R(x'x, b)$. Thus, when we consider the triples of the form $(x'x, b)$ for right extension, it is possible to generate the same paths again. To avoid such a double generation we use the dictionary Marked-Tuples. In Step 7 of Algorithm 8, just before we create a left extension of a set of triples of the form $(x\times, by)$ using the vertex $x' \in L(x, by)$, we check whether $(x'x, by)$ is present in Marked-Tuples. Recall that, Marked-Tuples is empty when the cleanup begins. When a triple for $(x'x, by)$ is generated for the first time (either by a left extension or right extension), and there are additional locally shortest paths in $G$ of the form $(x'x, by)$ which do not pass through $v$, we insert a tuple $(x'x, by)$ in Marked-Tuples (Step 11, Algorithm 8). Thus the data structure Marked-Tuples and the checks in Step 7 of Algorithm 8 ensure that the paths are generated exactly once either as a left extension or as a right extension but not by both. Note that such a marking is not required when there are no additional paths in $G$ which do not pass through $v$. In that case, we immediately delete $x'$ from $L(x, by)$ and $y$ from $R(x'x, b)$ (Step 13, Algorithm 8) ensuring that a triple for $(x'x, by)$ gets generated exactly once. This is the only case that can occur in DI [DI04] due to the assumption of unique shortest paths, and therefore this book-keeping with Marked-Tuples is not required in [DI04].

**Marked-Tuples.** The dictionary of Marked-Tuples is used to ensure that every path through the vertex $v$ is removed from the tuple system exactly once and therefore counts of paths in triples are correctly maintained. Note that a path of the form $(xa, by)$ can be generated either as a left extension of $(a, by)$ or by a right extension of $(xa, b)$. This is true in DI as well. However, due to the assumption of unique shortest paths they do not need to maintain counts of paths, and hence do not require the book-keeping using Marked-Tuples.

**Correctness and Complexity.**

We establish the correctness of cleanup in Lemma 8 and an upper bound on its worst case time in Lemma 9.

**Lemma 8.** *After Algorithm 8 is executed, the counts of triples in $P$ ($P^*$) represent counts of LSPs (SPs) in $G$ that do not pass through $v$. Moreover, the sets $L, L^*, R, R^*$ are correctly maintained.*

*Proof.* To prove the lemma statement we show that the while loop in Step 3 of Algorithm 8 maintains the following invariants.

**Loop Invariant:** At the start of each iteration of the while loop in Step 3 of Algorithm 8, assume that the min-key triple to be extracted and processed from $H_c$ has key $[wt, x, y]$. Then the following properties hold about the tuple system and $H_c$. We assert the invariants about the sets $P$, $L$, and $R$. Similar arguments can be used to establish the correctness of the sets $P^*$, $L^*$, and $R^*$.

$\mathcal{I}_1$ For any $a, b \in V$, if $G$ contains $c_{ab}$ number of locally shortest paths of weight $wt$ of the form $(xa, by)$ passing through $v$, then $H_c$ contains a triple $\gamma =$

$((xa, by), wt, c_{ab})$. Further, $c_{ab}$ has been decremented from the initial count in the triple for $(xa, by)$ in $P(x, y)$.

$\mathcal{I}_2$ Let $[\hat{wt}, \hat{x}, \hat{y}]$ be the key extracted from $H_c$ and processed in the previous iteration. For any key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, let $G$ contain $c > 0$ number of LSPs of weight $wt_1$ of the form $(x_1a_1, b_1y_1)$. Further, let $c_v$ (resp. $c_{\bar{v}}$) denote the number of such LSPs that pass through $v$ (resp. do not pass through $v$). Here $c_v + c_{\bar{v}} = c$. Then,

(a) if $c > c_v$ there is a triple in $P(x_1, y_1)$ of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ representing $c - c_v$ LSPs. If $c = c_v$ there is no such triple in $P(x_1, y_1)$.

(b) $x_1 \in L(a_1, b_1y_1)$, $y_1 \in R(x_1a_1, b_1)$, and $(x_1a_1, b_1y_1) \in$ Marked-Tuples iff $c_{\bar{v}} > 0$.

(c) For every $x' \in L(x_1, b_1y_1)$, a triple corresponding to $(x'x_1, b_1y_1)$ with weight $wt' = wt_1 + \mathbf{w}(x', x_1)$ and the appropriate count is in $H_c$ if $[wt', x', y_1] \geq [wt, x, y]$. A similar claim can be stated for every $y' \in R(x_1a_1, y_1)$.

$\mathcal{I}_3$ For any key $[wt_2, x_2, y_2] \geq [wt, x, y]$, let $G$ contain $c > 0$ LSPs of weight $wt_2$ of the form $(x_2a_2, b_2y_2)$. Further, let $c_v$ (resp. $c_{\bar{v}}$) denote the number of such LSPs that pass through $v$ (resp. do not pass through $v$). Here $c_v + c_{\bar{v}} = c$. Then the tuple $(x_2a_2, b_2y_2) \in$ Marked-Tuples, iff $c_{\bar{v}} > 0$ and a triple for $(x_2a_2, b_2y_2)$ representing $c_v$ LSPs is present in $H_c$.

**Initialization:** We show that the invariants hold at the start of the first iteration of the while loop in Step 3 of Algorithm 8. The min-key triple in $H_c$ has key $[0, v, v]$. Invariant $\mathcal{I}_1$ holds since we inserted into $H_c$ the trivial triple of weight 0 corresponding to the vertex $v$ and that is the only triple of such key. Moreover, since we do not represent trivial paths containing the single vertex, no counts need to be decremented. Since we assume positive edge weights, there are no LSPs in $G$

of weight less than zero. Thus, invariants $\mathcal{I}_2$(a), $\mathcal{I}_2$(b), and $\mathcal{I}_2$(c) hold trivially. Invariant $\mathcal{I}_3$ holds since $H_c$ does not contain any triple of weight $> 0$ and we initialized Marked-Tuples to empty.

**Maintenance:** Assume that the invariants are true at the beginning of the $1 \leq i \leq k$-th iteration of the while loop. We now prove that the claims are true at the beginning of the $(k+1)$-th iteration. Let the min-key triple at the beginning of the $k$-th iteration be $[wt_k, x_k, y_k]$. By invariant $\mathcal{I}_1$, we know that for any $a_i, b_j$, if there exists LSPs in $G$ of the form $(x_k a_i, b_j y_k)$ of weight $wt_k$, they have been inserted into $H_c$ and further their counts have been decremented from appropriate triples in $P(x_k, y_k)$. Now consider the set of triples with key $[wt_k, x_k, y_k]$ which we extract in the set $S$ (Step 4, Algorithm 8). We consider left-extensions of triples in $S$; symmetric arguments apply for right-extensions. Consider for a particular $b$, the set of triples $S_b \subseteq S$ and let $fcount'$ denote the sum of the counts of the paths represented by triples in $S_b$. Let $x' \in L(x_k, by_k)$; our goal is to generate the paths $(x'x_k, by_k)$ with count $= fcount'$ and weight $wt' = wt_k + \mathbf{w}(x', x_k)$. However, we generate such paths only if they have not been generated by a right-extension of another set of paths. We note that the paths of the form $(x'x_k, by_k)$ can be generated by right extending the set of triples of the form $(x'x_k, \times b)$. Without loss of generality assume that the triples of the form $(x'x_k, \times b)$ have a key which is greater than the key $[wt_k, x_k, y_k]$ and they are not in $H_c$. Thus, at the beginning of the $k$-th iteration, by invariant $\mathcal{I}_3$, we know that $(x'x_k, by_k) \notin$ Marked-Tuples. Steps 8–9, Algorithm 8 create a triple of the form $(x'x_k, by_k)$ of weight $wt'$ and decrement $fcount'$ many paths from the appropriate triple in $P(x', y_k)$ and add it to $H_c$. This establishes invariants $\mathcal{I}_2$(a) and $\mathcal{I}_2$(c) at the beginning of the $(k+1)$-th iteration. In addition, if there are no LSPs in $G$ of the form $(x'x_k, by_k)$ which do not pass through $v$, we delete $x'$ from $L(x_k, by_k)$ and delete $y_k$ from $R(x'x_k, b)$ (Step 13, Algorithm 8). On the other hand, if there exist LSPs in $G$ of the form $(x'x_k, by_k)$, then $x'$ (resp. $y_k$) continues to exist in $L(x_k, by_k)$ (resp. in $R(x'x, b)$). Further, we add the tuple $(x'x_k, by_k)$ to Marked-Tuples and note that the corresponding triple is already present in $H_c$ (Step 11, Algorithm 8). Since the invariants $\mathcal{I}_2$(b) and $\mathcal{I}_2$(c) were true for every key $< [wt_i, x_i, y_i]$ and by the above steps we ensure that these invariants hold for every key $= [wt_i, x_i, y_i]$. Thus, invariant $\mathcal{I}_2$(b) is true at the beginning of the $(k+1)$-th iteration. Note that any triple that is generated by a left extension (or symmetrically right extension) is inserted into $H_c$ as well as into Marked-Tuples. This establishes invariant $\mathcal{I}_3$ at the beginning of the $(k+1)$-th iteration.

Finally, to see that invariant $\mathcal{I}_1$ holds at the beginning of the $(k+1)$-th iteration, let the min-key at the $(k+1)$-th iteration be $[wt_{k+1}, x_{k+1}, y_{k+1}]$. Note that triples with weight $wt_{k+1}$ starting with $x_{k+1}$ and ending in $y_{k+1}$ can be created either by left extending or right extending the triples of smaller weight. And since for each of iteration $\leq k$ invariant $\mathcal{I}_2$(c) holds, we conclude that invariant $\mathcal{I}_1$ holds at the beginning of the $(k+1)$-th iteration.

**Termination:** The exit condition of the while loop is when the heap $H_c$ is empty. Because Invariant $\mathcal{I}_1$ maintains in $H_c$ the first triple to be extracted and processed, then $H_c = \emptyset$ implies that there are no more triples containing the vertex $v$ that need to be left or right extended and removed from the tuple system. Moreover, since the invariants hold for the last set of triples of weight $\hat{wt}$ extracted from the heap, by $\mathcal{I}_2$(a), all LSPs having weight less than or equal to $\hat{wt}$ have been decremented from the appropriate sets $P(\cdot)$. Finally, due to $\mathcal{I}_2$(b), the sets $L$ and $R$ are also correctly maintained after the while loop terminates. $\square$

**Lemma 9.** *For an update on a vertex $v$, Algorithm 8 takes $O(\nu^{*2} \cdot \log n)$ time.*

*Proof.* The cleanup procedure examines a triple $\gamma$ only if the tuple in $\gamma$ contains the updated vertex $v$. It removes each such $\gamma$ from a constant number of data structures $(P, P^*, L, L^*, R, R^*)$, each with an $O(\log n)$ cost. In addition, each triple is inserted into $H_c$ and extracted from it exactly once. Since the number of tuples containing $v$ is bounded by Lemma 7, the lemma follows.

□

### 3.2.4 The Fixup Procedure

The goal of the fixup procedure is to add to the tuple-system all *new* shortest and locally shortest paths (recall Definition 1).

The fixup procedure (pseudo-code in Algorithm 9) works with a heap of triples ($H_f$ here), which is initialized with a *candidate* shortest path triple for each pair of vertices. Recall that for a pair $x, y$, there may be several triples of a given weight $wt$ in $P(x, y)$. Instead of inserting all min-weight triples (which are candidates for shortest path triples), our algorithm inserts exactly one triple for every pair of vertices into $H_f$. This ensures that the number of triples examined during fixup is not too large. Once $H_f$ is suitably initialized, the fixup algorithm repeatedly extracts the set of triples with minimum key and processes them. The main invariant for the algorithm (similar to DI [DI04]) is that for a pair $x, y$, the weight of the first set of triples extracted from $H_f$ gives the distance from $x$ to $y$ in the updated graph. Thus, these triples are all identified as shortest path triples, and we need to extend them if in fact they represent *new* shortest paths. To readily identify triples containing paths through $v$ we use some additional book-keeping: for every triple $\gamma$ we store the update number (update-num($\gamma$)) and a count of the number of paths in that triple that pass through $v$ ($paths(\gamma, v)$). Finally, similar to cleanup, the fixup procedure also left and right extends triples to create triples representing new locally shortest paths.

**Algorithm 9** fixup($v, \mathbf{w}'$)

```
1:  $H_f \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2:  **for** each edge incident on $v$ **do**
3:      create a triple $\gamma$; set $paths(\gamma, v) = 1$; set update-num($\gamma$); add $\gamma$ to $H_f$ and to $P()$
4:  **for** each $x, y \in V$ **do**
5:      add a min-weight triple from $P(x, y)$ to $H_f$
6:  **while** $H_f \neq \emptyset$ **do**
7:      extract in $S'$ all triples with min-key $[wt, x, y]$ from $H_f$; $S \leftarrow \emptyset$
8:      **if** $S'$ is the first extracted set from $H_f$ for $x, y$ **then**
9:          {Steps 10–17: add new STs (or increase counts of existing STs) from $x$ to $y$.}
10:         **if** $P^*(x, y)$ is empty **then**
11:             **for** each $\gamma' \in P(x, y)$ with weight $wt$ **do**
12:                 let $\gamma' = ((xa', b'y), wt, count')$
13:                 add $\gamma'$ to $P^*(x, y)$ and $S$; add $x$ to $L^*(a', y)$ and $y$ to $R^*(x, b')$
14:         **else**
15:             **for** each $\gamma' \in S'$ containing a path through $v$ **do**
16:                 let $\gamma' = ((xa', b'y), wt, count')$
17:                 add $\gamma'$ with $paths(\gamma', v)$ in $P^*(x, y)$ and $S$; add $x$ to $L^*(a', y)$ and $y$ to $R^*(x, b')$

18:         {Steps 19–28: add new LSTs (or increase counts of existing LSTs) that extend SPs
            from $x$ to $y$.}
19:         **for** every $b$ such that $(x\times, by) \in S$ **do**
20:             let $fcount' = \sum_i ct_i$ such that $((xa_i, by), wt, ct_i) \in S$
21:             **for** every $x'$ in $L^*(x, b)$ **do**
22:                 **if** $(x'x, by) \notin$ Marked-Tuples **then**
23:                     $wt' \leftarrow wt + \mathbf{w}(x', x)$; $\gamma' \leftarrow ((x'x, by), wt', fcount')$
24:                     set update-num($\gamma'$); $paths(\gamma', v) \leftarrow \sum_{\gamma=(x\times,by)} paths(\gamma, v)$; add $\gamma'$ to $H_f$
25:                     **if** a triple for $(x'x, by)$ exists in $P(x', y)$ **then**
26:                         add $\gamma'$ with $paths(\gamma', v)$ in $P(x', y)$; add $(x'x, by)$ to Marked-Tuples
27:                     **else**
28:                         add $\gamma'$ to $P(x', y)$; add $x'$ to $L(x, by)$ and $y$ to $R(x'x, b)$
29:         perform steps symmetric to Steps 19 – 28 for right extensions.
```

We now describe the steps of the algorithm. Algorithm 9 initializes $H_f$ in Steps 2–5 as follows. (i) For every edge incident on $v$, it creates a trivial triple $\gamma$ which is inserted into $H_f$ and $P$. It also sets update-num($\gamma$) and paths($\gamma, v$) for each such $\gamma$; (ii) For every $x, y \in V$, it adds a candidate min-weight triple from $P(x, y)$ to $H_f$ (even if $P(x, y)$ contains several min-weight triples; this is done for efficiency).

Algorithm 9 executes Steps 10–17 when for a pair $x, y$, the first set of triples $S'$, all of weight $wt$, are extracted from $H_f$. We claim (Invariant 4) that $wt$ denotes the shortest path distance from $x$ to $y$ in the updated graph. The goal of Steps 10–17 is to create a set $S$ of triples that represent *new* shortest paths, and this step is considerably more involved than the corresponding step in DI. In DI [DI04], only a single path $p$ is extracted from $H_f$ possibly resulting in a *new* shortest path from $x$ to $y$. If $p$ is *new* then it is added to $P^*$ and the algorithm extends it to create new LSP. In our case, we extract not just multiple paths but multiple shortest path triples from $x$ to $y$, and some of these triples may not be in $H_f$. We now describe how the algorithm generates the new shortest paths in Steps 10–17.

Steps 10–17, Algorithm 9 – As mentioned above, Steps 10–17 create a set $S$ of triples that represent *new* shortest paths. There are two cases.

- $P^*(x, y)$ is empty: Here, we process the triples in $S'$, but in addition, we may be required to process triples of weight $wt$ from the set $P(x, y)$. To see this, consider the example in Fig. 3.1 and consider the pair $a_1, b_1$. In $G$, there is one shortest path $\langle a_1, v, b_1\rangle$ which is removed from $P(a_1, b_1)$ and $P^*(a_1, b_1)$ during cleanup. In $G'$, $d(a_1, b_1) = 4$ and there are 2 shortest paths, namely $p_1 = \langle a_1, b_1\rangle$ and $p_2 = \langle a_1, v_1, b_1\rangle$. Note that both of these are LSPs in $G$ and therefore are present in $P(a_1, b_1)$. In Step 5, Algorithm 9 we insert exactly one of them into the heap $H_f$. However, both need to be processed and also left and right extended to create new locally shortest paths. Thus, under this condition, we examine all the min-weight triples present in $P(a_1, b_1)$.

- $P^*(x,y)$ is non-empty: After an increase-only update, the distance from $x$ to $y$ can either remain the same or increase, but it cannot decrease. Further, cleanup removed from the tuple-system all paths that contain $v$. Hence, if $P^*(x,y)$ is non-empty at this point, it implies that all paths in $P^*(x,y)$ avoid $v$. In this case, we can show (Invariant 5) that it suffices to only examine the triples present in $H_f$. Furthermore, the only paths that we need to process are the paths that pass through the vertex $v$.

Steps 19–29, Algorithm 9 – These steps left-extend and right-extend the triples in $S$ representing *new* shortest paths from $x$ to $y$.

Fixup maintains the following two invariants. The invariant below (Invariant 4) shows that for any pair $x, y$, the weight of the first set of the triples extracted from $H_f$ determines the shortest path distance from $x$ to $y$. The proof of the invariant is similar to the proof of Invariant 3.1 in [DI04].

**Invariant 4.** *If the set $S'$ in Step 7 of Algorithm 9 is the first extracted set from $H_f$ for $x, y$, then the weight of each triple in $S'$ is the shortest path distance from $x$ to $y$ in the updated graph.*

*Proof.* Assume for the sake of contradiction that the invariant is violated at some extraction. Thus, the first set of triples $S'$ of weight $\hat{wt}$ extracted for some pair $(x,y)$ do not represent the set of shortest paths from $x$ to $y$ in the updated graph. Consider the earliest of these events and let $\gamma = ((xa', b'y), wt, count)$ be a triple in the updated graph that represents a set of shortest paths from $x$ to $y$ with $wt < \hat{wt}$. The triple $\gamma$ cannot be present in $H_f$, else it would have been extracted before any triple of weight $\hat{wt}$ from $H_f$. Moreover, $\gamma$ cannot be in $P(x,y)$ at the beginning of fixup otherwise $\gamma$ (or some other triple of weight $wt$) would have been inserted into $H_f$ during Step 5 of Algorithm 9. Thus $\gamma$ must be a *new* LST generated by the algorithm. Since all edges incident on $v$ are added to $H_f$ during Step 2 of Algorithm 9 and $\gamma$ is not present in $H_f$, implies that $\gamma$ represents paths which have at least two or more edges. We now define left($\gamma$) as the set of LSTs of the form $((xa, c_i b), wt - \mathbf{w}(b, y), count_i)$ that represent all the LSPs in the left tuple $(xa, b)$; similarly we define right($\gamma$) as the set of LSTs of the form $((ad_j, by), wt - \mathbf{w}(x, a), count_j)$ that represent all the LSPs in the right tuple $(a, by)$. Note that since $\gamma$ is a shortest path tuple, all the paths represented by LSTs in left($\gamma$) and right($\gamma$) are also shortest paths. All of the paths in either left($\gamma$) or in right($\gamma$) are *new* shortest paths and therefore are not present in $P^*$ at the beginning of fixup. Since edge weights are positive $(wt - \mathbf{w}(b, y)) < wt < \hat{wt}$ and $(wt - \mathbf{w}(x, a)) < wt < \hat{wt}$. As we extract paths from $H_f$ in increasing order of weight, and all extractions before the wrong extraction were correct, the triples in left($\gamma$) and right($\gamma$) should have been extracted from $H_f$ and added to $P^*$. Thus, the triple corresponding to $(xa, by)$ of weight $wt$ should have been generated during left or right extension and inserted in $H_f$. Hence, some triple of weight $wt$ must be extracted from $H_f$ for the pair $(x, y)$ before any triple of weight $\hat{wt}$ is extracted from $H_f$. This contradicts our assumption that the invariant is violated. □

Using Invariant 5 below we show that fixup indeed considers all of the *new* shortest paths for any pair $x, y$. Recall that all the *new* shortest paths for a pair need not be present in $H_f$ and we may be required to consider min-weight triples present in $P(\cdot)$ as well.

**Invariant 5.** *The set $S$ of triples constructed in Steps 10–17 of Algorithm 9 represents all of the* new *shortest paths from $x$ to $y$.*

*Proof.* Any new SP from $x$ to $y$ is of the following three types:

1. a single edge containing the vertex $v$ (such a path is added to $P(x, y)$ and $H_f$ in Step 2)

2. a path generated via left/right extension of some shortest path (such a path is added to $P(x, y)$ and $H_f$ in Step 24 and an analogous step in right-extend).

3. a path that was an LSP but not SP before the update and is an SP after the update.

In (1) and (2) above any new SP from $x$ to $y$ which is added to $H_f$ is also added to $P(x, y)$. However, amongst the several triples representing paths of the form (3) listed above, only one candidate triple will be present in $H_f$. Thus we conclude that for a given $x, y$ and when we extract from $H_f$ triples of weight $wt$, $P(x, y)$ contains a superset of the triples that are present in $H_f$. We now consider the two cases that the algorithm deals with.

- $P^*(x, y)$ is empty when the first set of triples for $x, y$ is extracted from $H_f$. In this case, we process all the min-weight triples in $P(x, y)$. By the above argument, we know that all new SPs from $x$ to $y$ are present in $P(x, y)$. Therefore it suffices to argue that all of them are *new*. Assume for the sake of contradiction, some path $p$ represented by them is *not new*. By definition, $p$ does not contain $v$ and $p$ was a SP before the update. Therefore, clearly $p$ was in $P^*(x, y)$ before the update. However, since cleanup only removes paths that contain $v$, the path $p$ remains untouched during cleanup and hence continues to exist in $P^*(x, y)$. This contradicts the fact that $P^*(x, y)$ is empty.

- $P^*(x, y)$ is not empty when the first set of triples for $x, y$ is extracted from $H_f$. Let the weight of triples in $P^*(x, y)$ be $wt$. This implies that the shortest path distance from $x$ to $y$ before and after the update is $wt$. Recall that we are dealing with increase-only updates. We first argue that it suffices to consider triples in $H_f$. This is observed from the fact that any *new* SP of the form (1) and (2) listed above is present in $H_f$. Furthermore, note that any path of form (3) above has a weight strictly larger than $wt$ since it was an LSP and not SP before the update. Thus in the presence of paths of weight $wt$, none of the paths of form (3) are candidates for shortest paths from $x$ to $y$. This justifies considering triples only in $H_f$.

Finally, we note that for any triple considered, our algorithm only processes paths through $v$. This again follows from the fact that only paths through $v$ were removed by cleanup and possibly need to be restored if the distance via them remains unchanged after the update.

□

The following lemma establishes the correctness of fixup.

**Lemma 10.** *After execution of Algorithm 9, for any $(x, y) \in V$, the counts of the triples in $P(x, y)$ and $P^*(x, y)$ represent the counts of LSPs and SPs from $x$ to $y$ in the updated graph. Moreover, the sets $L, L^*, R, R^*$ are correctly maintained.*

*Proof.* We prove the lemma statement by showing the invariants are maintained by the while loop in Step 6 of Algorithm 9.

**Loop Invariant:** At the start of each iteration of the while loop in Step 6 of Algorithm 9 let the min-key triple to be extracted and processed from $H_f$ have key $= [wt, x, y]$. We claim the following about the tuple-system and $H_f$.

$\mathcal{I}_1$ For any $a, b \in V$, if $G'$ contains $c_{ab}$ number of LSPs of weight $wt$ of the form $(xa, by)$. Further, a triple $\gamma = ((xa, by), wt, c_{ab})$ is present in $P(x, y)$ (note that $H_f$ can also contain other triples from $x$ to $y$ with weight $wt$).

$\mathcal{I}_2$ Let $[\hat{wt}, \hat{x}, \hat{y}]$ be the last key extracted from $H_f$ and processed before $[wt, x, y]$. For any key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, let $G'$ contain $c > 0$ number of SPs of weight $wt_1$ of the form $(x_1a_1, b_1y_1)$. Further, let $c_{new}$ (resp. $c_{old}$) denote the number of such SPs that are *new* (resp. not *new*). Here $c_{new} + c_{old} = c$. Then,

(a) the triple for $(x_1a_1, b_1y_1)$ with weight $wt_1$ in $P^*(x_1, y_1)$ represents $c$ SPs.

(b) $x_1 \in L(a_1, b_1y_1)$, $x_1 \in L^*(a_1, y_1)$, and $y_1 \in R(x_1a_1, b_1)$, $y_1 \in R^*(x_1, b_1)$. Further, $(x_1a_1, b_1y_1) \in$ Marked-Tuples iff $c_{old} > 0$.

(c) If $c_{new} > 0$, for every $x' \in L(x_1, b_1y_1)$, a triple corresponding to $(x'x_1, b_1y_1)$ with weight $wt' = wt_1 + \mathbf{w}(x'x_1)$ and the appropriate count is in $P(x_1, y_1)$ and in $H_f$ if $[wt', x', y_1] \geq [wt, x, y]$. A similar claim can be stated for every $y' \in R(x_1a_1, y_1)$.

$\mathcal{I}_3$ For any key $[wt_2, x_2, y_2] \geq [wt, x, y]$, let $G'$ contain $c > 0$ number of LSPs of weight $wt_2$ of the form $(x_2a_2, b_2y_2)$. Further, let $c_{new}$ (resp. $c_{old}$) denote the number of such SPs that are *new* (resp. not *new*). Here $c_{new} + c_{old} = c$. Then the tuple $(x_2a_2, b_2y_2) \in$ Marked-Tuples, iff $c_{old} > 0$ and $c_{new}$ paths have been added to $H_f$ by some earlier iteration of the while loop.

The proof that these invariants hold at initialization and termination and are maintained at every iteration of the while loop is similar to the proof of Lemma 8. □

**Complexity of Fixup.**

As in DI, we observe that shortest paths and LSPs are removed only in cleanup and are added only in fixup. In a call to fixup, accessing a triple takes $O(\log n)$ time since it is accessed on a constant number of data structures. So, it suffices to bound the number of triples accessed in a call to fixup, and then multiply that bound by $O(\log n)$.

We will establish an amortized bound. The total number of LSTs at any time, including the end of the update sequence, is $O(m^* \cdot \nu^*)$ (by Lemma 6). Hence, if fixup accessed only *new* triples outside of the $O(n^2)$ triples added initially to $H_f$, the amortized cost of fixup (for a sufficiently long update sequence) would be $O(\nu^{*2} \cdot \log n)$, the cost of a cleanup. This is in fact the analysis in DI, where fixup satisfies this property. However, in our algorithm fixup accesses several triples that are already in the tuple system: In Steps 11–13 we examine triples already in $P$, in Steps 15–17 we could increment the count of an existing triple in $P^*$, and in Steps 19–28 we increment the count of an existing triple in $P$. We bound the costs of these steps in Lemma 11 below by classifying each triple $\gamma$ as one of the following disjoint types:

- **Type-0 (contains-v):** $\gamma$ represents at least one path containing vertex $v$.
- **Type-1 (new-LST):** $\gamma$ was not an LST before the update but is an LST after the update, and no path in $\gamma$ contains $v$.
- **Type-2 (new-ST-old-LST):** $\gamma$ is an ST after the update, and $\gamma$ was an LST but not an ST before the update, and no path in $\gamma$ contains $v$.
- **Type-3 (new-ST-old-ST):** $\gamma$ was an ST before the update and continues to be an ST after the update, and no path in $\gamma$ contains $v$.
- **Type-4 (new-LST-old-LST):** $\gamma$ was an LST before the update and continues to be an LST after the update, and no path in $\gamma$ contains $v$.

The following lemma establishes an amortized bound for fixup which is the same as the worst case bound for cleanup.

**Lemma 11.** *The fixup procedure takes time $O(\nu^{*2} \cdot \log n)$ amortized over a sequence of $\Omega(m^*/\nu^*)$ increase-only updates.*

*Proof.* We bound the number of triples examined; the time taken is $O(\log n)$ times the number of triples examined due to the data structure operations performed on a triple. The initialization in Steps 1–5 takes $O(n^2)$ time. We now consider the triples examined after Step 5. The number of Type-0 triples is $O(\nu^{*2})$ by Lemma 7. The number of Type-1 triples is addressed by amortizing over the entire update sequence as described in the paragraph below. For Type-2 triples we observe that since updates only increase the weights on edges, a shortest path never reverts to being an LSP. Further, each such Type-2 triple is examined only a constant number of times (in Steps 10–13). Hence we charge each access to a Type-2 triple to the step in which it was created as a Type-1 triple. For Type-3 and Type-4, we note that for any $x, y$ we add exactly one candidate min-weight triple from $P(x, y)$ to $H_f$, hence initially there are at most $n^2$ such triples in $H_f$. Moreover, we never process an old LST which is not an ST so no additional Type-4 triples are examined during fixup. Finally, triples in $P^*$ that are not placed initially in $H_f$ are not examined in any step of fixup, so no additional Type-3 triples are examined. Thus the number of triples examined by a call to fixup is $O(\nu^{*2})$ plus $O(X)$, where $X$ is the number of *new* triples fixup adds to the tuple system. (This includes an $O(1)$ credit placed on each new LST for a possible later conversion to an ST.)

Let $\sigma$ be the number of updates in the update sequence. Since triples are removed only in cleanup, at most $O(\sigma \cdot \nu^{*2})$ triples are removed by the cleanups. There can be at most $O(m^* \cdot \nu^*)$ triples remaining at the end of the sequence (by Lemma 1), hence the total number of new triples added by all fixups in the update sequence is $O(\sigma \cdot \nu^{*2} + m^* \cdot \nu^*)$. When $\sigma > m^*/\nu^*$, the first term dominates, and this gives an average of $O(\nu^{*2})$ triples added per fixup, and the desired amortized time bound for fixup. □

### 3.2.5 Complexity of the Increase-Only Algorithm.

Lemma 11 establishes that the amortized cost per update of fixup is $O(\nu^{*2} \cdot \log n)$ when the increase-only update sequence is of length $\Omega(m^*/\nu^*)$. Lemma 9 shows that the worst case cost per update of cleanup is $O(\nu^{*2} \cdot \log n)$. Since an update operation consists of a call to cleanup followed by a call to fixup, this establishes Theorem 2. The space used by our algorithm is $O(m^* \cdot \nu^*)$, the worst case number of triples in our tuple system.

# Chapter 4

# Improved Increase-Only Algorithm

In Chapter 3, we presented an increase-only APASP algorithm which requires $O(m^* \cdot \nu^*)$ space. In this chapter, we show how to reduce the space complexity of the above by designing an enhanced algorithm that uses only $O(m^* \cdot n)$ space. There are two main ideas: the first is to change the data structures used by our algorithms, while the seconds consists in avoid the double generations of tuples using a new structure called Marked-Pairs.

**Organization.** The rest of the Chapter is organized as follows: in Section 4.1, we describe the new data structures used by our refined algorithms (some of these are very similar to the structures already introduced in Chapter 3). In Section 4.2, we present a refined cleanup algorithm which integrates the new tuple-system. In this section, we also explain how to avoid the double generation problem using Marked-Pairs. In Section 4.3, we discuss the main challenges imposed by our new tuple-system and we also explore the limits of our tuples approach. Finally, in Section 4.4, we present a refined fixup algorithm which completes our APASP algorithm.

## 4.1 Data Structures

We start by recalling the notions of a *tuple*, *$\ell$-tuple*, *$r$-tuple* and a *triple*, already defined in Chapter 3. These will be used to represent a set of LSPs and SPs in $G$.

**Tuple:** A tuple, $\tau = (xa, by)$, represents the set of LSPs in $G$, all of which use the same first edge $(x, a)$ and the same last edge $(b, y)$. The weight of every path represented by $\tau$ is $\mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$. We call $\tau$ a *locally shortest path tuple (LST)*. In addition, if $d(x, y) = \mathbf{w}(x, a) + d(a, b) + \mathbf{w}(b, y)$, then $\tau$ is a *shortest path tuple (ST)*. Fig. 3.5(a) shows a tuple $\tau$. We use the notation $x \to a \rightsquigarrow b \to y$ to denote a set of paths represented by $\tau$.

**$\ell$-tuple:** An $\ell$-tuple, $\tau_\ell = (xa, y)$, represents the set of LSPs in $G$, all of which start with the edge $(x, a)$ and end at the vertex $y$. The weight of every path represented by $\tau_\ell$ is $\mathbf{w}(x, a) + d(a, y)$. We call $\tau_\ell$ a *locally shortest path tuple (LST)* since every path in $\tau$ is an LSP. In addition, if $d(x, y) = \mathbf{w}(x, a) + d(a, y)$, then $\tau_\ell$ is a *shortest path tuple (ST)*. Fig. 3.5(b) shows an $\ell$-tuple $\tau_\ell$.

**$r$-tuple:** An $r$-tuple, $\tau_r = (x, by)$, represents the set of LSPs in $G$, all of which start at the vertex $x$ and end with the edge $(b, y)$. The weight of every path represented by $\tau_r$ is $d(x, b) + \mathbf{w}(b, y)$. We call $\tau_r$ a *locally shortest path tuple (LST)*. In addition, if $d(x, y) = d(x, b) + \mathbf{w}(b, y)$, then $\tau_r$ is a *shortest path tuple (ST)*. Fig. 3.5(c) shows a $r$-tuple $\tau_r$.

### 4.1.1 New Data Structures

Here, we describe the new data structures needed to maintain our new tuple-system. These data structures are used by the algorithms we present in the following sections. For clarity, we refer to a tuple as a *full-tuple*. Although we will not use full-tuples to store locally shortest paths, they are used during the running time analysis of our algorithm. For every $x, y$, $x \neq y$ in $V$, we maintain the following:

1. $P_\ell(x, y)$ – a priority queue containing locally shortest $\ell$-tuples from $x$ to $y$ with weight as key.
2. $P_r(x, y)$ – a priority queue containing locally shortest $r$-tuples from $x$ to $y$ with weight as key.
3. $P^*_\ell(x, y)$ – a priority queue containing shortest $\ell$-tuples from $x$ to $y$ with weight as key.
4. $P^*_r(x, y)$ – a priority queue containing shortest $r$-tuples from $x$ to $y$ with weight as key.
5. $L(a, by)$ – a balanced search tree containing vertices with vertex ID as key.
6. $R(xa, b)$ – a balanced search tree containing vertices with vertex ID as key.
7. $L^*(x, y)$ – a balanced search tree containing vertices with vertex ID as key.
8. $R^*(x, y)$ – a balanced search tree containing vertices with vertex ID as key.

We maintain all $\ell$-tuples and $r$-tuples in balanced search trees $dict_\ell$ and $dict_r$. An $\ell$-tuple $\tau_\ell = (xa, y)$ has key $[x, y, a]$; analogous key is defined for an $r$-tuple. We also maintain pointers from $\tau_\ell$ to $R(xa, y)$ and to the corresponding triple containing $\tau_\ell$ in $P_\ell(x, y)$. Finally, we maintain a set of pairs of vertices which we call as Marked-Pairs. The usefulness of this set will be discussed later.

In Chapter 3, we implemented both cleanup and fixup using single heaps $H_c$ and $H_f$ to store full-tuples. In our enhanced cleanup, we will store respectively $\ell$-tuples and $r$-tuples in dedicated heaps called $H_{\ell c}$ and $H_{rc}$. Similar data structures, called $H_{\ell f}$ and $H_{rf}$, will be used in fixup. The use of these new structures, the refined marking scheme maintained in the new structure Marked-Pairs and the enhanced algorithms, are the core elements for saving space during the computation. We will now describe some properties of the new heaps.

**Heaps and their keys.** Our procedures cleanup and fixup both make use of min-heaps of triples. In fact in each of the procedures we use two heaps – the left heap and the right heap. A left heap stores triples containing $\ell$-tuples and a right heap stores triples containing $r$-tuples. For an $\ell$-triple $((xa, y), wt, count)$ the key in a left heap $H_\ell$ is $[wt, x, y]$. For two tuples $\gamma$ and $\gamma'$ with weights $wt$ and $wt'$ respectively, $wt < wt'$ implies $\gamma < \gamma'$. In case $wt = wt'$, lexicographic ordering of the end points of the paths represented by $\gamma$ and $\gamma'$ determine the ordering of the two triples. We remark that we can have multiple triples with the same key in a particular heap. However, our algorithms process all such triples having the same key together – hence our definition of key suffices. The key for a $r$-triple in a right heap is exactly the same as that of an $\ell$-triple.

**Adding and removing paths.** Recall that we store locally shortest paths in $P_\ell, P_r$ and $P^*_\ell, P^*_r$ as triples. During the course of the update algorithm, we will need to *add* and *remove* a set of paths from these data structures. Since we have stored $\ell$-tuples and $r$-tuples with the counts of the paths that they represent, adding or removing paths implies incrementing or decrementing the count in the relevant triple. Say, at some point in the algorithm $P_\ell(x, y)$ contains an $\ell$-tuple $((xa, y), wt, count)$. Furthermore, let us assume that we execute the following statement in our algorithm: "remove $((xa, y), wt, count')$ in $P_\ell(x, y)$". After the execution of the above statement, $P_\ell(x, y)$ contains the $\ell$-triple $((xa, y), wt, count' - count)$. If $count' - count = 0$, then the $\ell$-triple is deleted from $P_\ell(x, y)$. An analogous meaning is attached to the statement "add $\gamma$ in $P_\ell(x, y)$".

### 4.1.2 Key Deviations from Chapter 3([NPR14a])

An important difference between this version and our increase-only `NPRdec` algorithm in Chapter 3 is to store LSPs as $\ell$-tuples and $r$-tuples instead of full-tuples. As we see later, this improves the space complexity of the algorithm (as compared to NPRdec) while maintaining the same time complexity of NPRdec. We will discuss other significant changes while describing our algorithms. In the rest of this section, we estimate the maximum number of full-tuples and $\ell$-tuples, $r$-tuples that can be present in any graph.

**Lemma 12.** *The number of $\ell$-tuples (analogously $r$-tuples) representing locally shortest paths in $G = (V, E)$ is bounded by $O(m^* \cdot n)$. The number of full-tuples representing locally shortest paths in $G = (V, E)$ is bounded by $O(m^* \cdot \nu^*)$.*

*Proof.* Every single edge is an LSP therefore such LSPs account for at most $O(m)$ many $\ell$-tuples. For any non-trivial $\ell$-tuple, the first edge can be chosen in $m^*$ different ways and end-point can be chosen in $n$ different ways. Thus the number of $\ell$-tuples (and by a symmetric argument the number of $r$-tuples) is at most $O(m^* \cdot n)$. For any full-tuple $(\times a, \times\times)$, for some $a \in V$, the first and last edge of any such tuple must lie on a shortest path containing $a$. Let $E_a^*$ denote the set of edges that lie on shortest paths through $a$, and let $I_a$ be the set of incoming edges to $a$. Then, there are at most $\nu^*$ ways of choosing the last edge in $(\times a, \times\times)$ and at most $E_a^* \cap I_a$ ways of choosing the first edge in $(\times a, \times\times)$. Since $\sum_{a \in V} |E_a^* \cap I_a| = m^*$, the number of full-tuples in $G$ is at most $\sum_{a \in V} \nu^* \cdot |E_a^* \cap I_a| \leq m^* \cdot \nu^*$. $\square$

In the next lemma we bound the number of $\ell$-tuples, $r$-tuples and full-tuples that contain a given vertex $v$.

**Lemma 13.** *The number of $\ell$-tuples (analogously $r$-tuples) that contain a vertex $v$ is $O(\nu^* \cdot n)$. The number of full-tuples that contain a vertex $v$ is $O(\nu^{*2})$.*

*Proof.* Number of $\ell$-tuples that start/end with $v$ are bounded by $O(n^2)$. Number of $\ell$-tuples that contain $v$ as an internal vertex are bounded by $O(\nu^* \cdot n)$.

To bound the number of full-tuples, we consider three different cases:

1. Full-tuples starting with $v$: For a full-tuple that starts with edge $(v, a)$, the last edge must lie on $a$'s SP dag, so there are at most $\nu^*$ choices for the last edge.

Hence, the number of full-tuples with $v$ as start vertex is at most $\sum_{a \in V \setminus v} \nu^* \leq n \cdot \nu^*$.

2. Similarly, the number of full-tuples with $v$ as end vertex is at most $n \cdot \nu^*$.

3. For any full-tuple $\tau = (xa, by)$ that contains $v$ as an internal vertex, both $(x, a)$ and $(b, y)$ lie on $v$'s SP dag, hence the number of such full-tuples is at most $\nu^{*2}$. $\square$

## 4.2 The Cleanup Algorithm

This section discusses the cleanup algorithm. The cleanup procedure removes from the tuple-system every LSP that contains the updated vertex $v$. The pseudo-code for our algorithm is described in Section 4.2.1. In Section 4.2.2 we prove useful invariants of cleanup. Finally, in Section 4.2.3 we prove the correctness of cleanup and analyze its running time.

### 4.2.1 Pseudo-code

Alg. 10 (cleanup) uses two min-heaps $H_{\ell c}$ and $H_{rc}$ which are initialized to empty. It also initializes the dictionary Marked-Pairs to empty. The algorithm then creates the trivial left and right triples corresponding to the vertex $v$ and adds it to $H_{\ell c}$ and $H_{rc}$ respectively (Step 4, Alg. 10). The algorithm repeatedly extracts min-key triples from $H_{\ell c}$ (Step 6, Alg. 10) and *processes* them. The processing of triples involves left-extending and right-extending triples, which our algorithm achieves by invoking Algorithm 11 process_left (Step 7, Alg. 10). The triples formed by left and right extensions are removed from the tuple system. Analogous steps are performed for the triples present in $H_{rc}$. These steps are similar in spirit to cleanup in DI. However, since we deal with a set of paths instead of a single path, we need the following two significant modifications:

1. Use of *two* heaps from which we repeatedly extract min-key triples.

2. Use of Marked-Pairs to ensure that the counts of the LSPs are correctly maintained.

**Algorithm 10** cleanup($v$)

```
1: Set $H_{\ell c}$, $H_{rc}$ and Marked-Pairs to $\emptyset$
2: $\gamma_\ell \leftarrow ((vv, v), 0, 1)$
3: $\gamma_r \leftarrow ((v, vv), 0, 1)$
4: add $\gamma_\ell$ to $H_{\ell c}$ and $\gamma_r$ to $H_{rc}$
5: **while** $H_{\ell c} \neq \emptyset$ and $H_{rc} \neq \emptyset$ **do**
6:     extract in $S_\ell$ all triples with same min-key from $H_{\ell c}$
       let $[wt, x, y]$ be the min-key of $H_{\ell c}$
7:     process_left($S_\ell$)
8:     extract in $S_r$ all triples with same min-key from $H_{rc}$
       //the min-key of $H_{rc}$ will also be $[wt, x, y]$
9:     process_right($S_r$)
```

We now describe the Algorithm 11 (process_left). The input to the algorithm is a set $S_\ell$ of left triples all having the same key $[wt, x, y]$. Consider an $\ell$-tuple of the form $(xa, y)$ and let $\gamma = ((xa, y), wt, ct) \in S_\ell$. Let us consider all the right-extensions $y'$ of $(xa, y)$. These extensions are present in $R(xa, y)$. However, since we extract and extend triples from both $H_{\ell c}$ and $H_{rc}$, we need to ensure that a triple of the form $(xa, y')$ gets generated exactly once during our algorithm. In Section 4.2.3 we characterize triples that can potentially get generated twice during our cleanup. For the moment, assume that the set $\mathcal{RE}_c$ (in Step 9, Alg. 11) contains all the extensions of $(xa, y)$ that need to be processed. Our algorithm considers every extension $y' \in \mathcal{RE}_c$ (Step 9) and invokes the right_extend procedure for the triple $\gamma$ and the extension $y'$.

We now turn our attention to Alg. 13 right_extend. The algorithm takes as its input an $\ell$-tuple $\gamma = ((xa, y), wt, count)$ and a vertex $y' \in \mathcal{RE}_c$ with which the paths represented by $\gamma$ are right-extended. That is, we create paths of the form $x \rightarrow a \rightsquigarrow y \rightarrow y'$). Though the algorithm works to right-extend paths, in the process, in Step 2 we create triples of both the types; triple $\gamma_r$ containing $\ell$-tuple

**Algorithm 11** process_left($S_\ell$)

```
1: for every $\gamma$ in $S_\ell$ do
2:   let $\gamma = ((xa, y), wt, ct)$
3:   // Populate $\mathcal{RE}_c$ with the right extensions of $\gamma$ which need to be processed.
4:   if $P^*_\ell(a, y) \neq \emptyset$ and $v \neq x$ then
5:     add $(a, y)$ to Marked-Pairs // marked for fixup
6:     $\mathcal{RE}_c = \{y' \mid y' \in R(xa, y)$ and $\mathbf{w}(y, y') > \mathbf{w}(x, a)\}$
7:   else
8:     $\mathcal{RE}_c \leftarrow R(xa, y)$
9:   for every $y' \in \mathcal{RE}_c$ do
10:     right_extend($\gamma$, $y'$)
```

**Algorithm 12** process_right($S_r$)

```
1: for every $\gamma$ in $S_r$ do
2:   let $\gamma = ((a, yy'), wt, ct)$
3:   // Populate $\mathcal{LE}_c$ with the left extensions of $\gamma$ which need to be processed.
4:   if $P^*_r(a, y) \neq \emptyset$ and $v \neq y'$ then
5:     add $(a, y)$ to Marked-Pairs // marked for fixup
6:     $\mathcal{LE}_c = \{x \mid x \in L(a, yy')$ and $\mathbf{w}(y, y') \leq \mathbf{w}(x, a)\}$
7:   else
8:     $\mathcal{LE}_c \leftarrow L(a, yy')$
9:   for every $x \in \mathcal{LE}_c$ do
10:     left_extend($\gamma$, $y'$)
```

representation of the paths and triple $\gamma_\ell$ containing $r$-tuple representation of the paths. The need to create triples of both types is discussed in Section 4.3.2. Step 14 adds the triples to the appropriate heaps and Step 3 removes these triples from the appropriate data structures. Note that this corresponds to decrementing the count of the triples. In Step 6 we check if the shortest path triples of the form $(xa, y)$ have been removed from $P^*_\ell(x, y)$. If this is the case, it implies that in $G$ all shortest paths of the form $x \rightarrow a \rightsquigarrow y$ are via the updated vertex $v$. Thus in $G$, all LSPs of the form $(xa, yy')$ are also via the vertex $v$. Thus after removal of these paths from the appropriate data structures, it is also required to remove $y'$ from $R(xa, y)$. Our algorithm does the same in Step 6–7 of Alg.13. Symmetric steps are performed for checking if $x$ needs to be removed from $L(a, yy')$. Finally, if the triple generated represents shortest paths from $x$ to $y'$ then the corresponding entries are removed from the data structures $P^*$, $L^*$ and $R^*$ (in steps 8–13).

**Algorithm 13** right_extend($\gamma = ((xa, y), wt, count), y'$)

```
1: $wt' \leftarrow wt + \mathbf{w}(x, y)$;
2: $\gamma_r \leftarrow ((x, yy'), wt', ct)$; $\gamma_\ell = ((xa, y'), wt', ct)$;
3: remove $\gamma_r$ in $P_r(x, y')$; remove $\gamma_\ell$ in $P_\ell(x, y')$
4: if a triple for $(xa, y)$ does not exist in $P_\ell^*(x, y)$ then
5:    delete $y'$ from $R(xa, y)$
6: if a triple for $(a, yy')$ does not exist in $P_r^*(a, y')$ then
7:    delete $x$ from $L(a, yy')$
8: if the shortest path distance from $x$ to $y'$ is $wt'$ then
9:    remove $\gamma_r$ in $P_r^*(x, y')$; remove $\gamma_\ell$ in $P_\ell^*(x, y')$
10:   if a triple for $(x, yy')$ does not exist in $P_r^*(x, y')$ then
11:      delete $y'$ from $R^*(x, y)$
12:   if a triple for $(xa, y')$ does not exist in $P_\ell^*(x, y')$ then
13:      delete $x$ from $L^*(a, y')$
14: add $\gamma_r$ to $H_{rc}$; add $\gamma_\ell$ to $H_{\ell c}$
```

### 4.2.2 Invariants of cleanup

We now state some simple but useful invariants of the cleanup algorithm (Alg. 10).

**Invariant 6.** *If a triple with key $k = (wt, x, y)$ is extracted from either $H_{\ell c}$ or $H_{rc}$ for processing during Step 6 or Step 8 of Alg. 10 then all triples of key strictly smaller than $k$ have been extracted and processed from both $H_{\ell c}$ and $H_{rc}$.*

*Proof.* We note that once we extract a set of triples with key $k = (wt, x, y)$ from $H_{\ell c}$ in Step 6, we extract all triples with the same key from $H_{rc}$ in Step 8 before extracting any higher key triples from $H_{\ell c}$. Furthermore the algorithms process_left (Step 7) and process_right (Step 9) add triples with keys strictly larger than $k$ to $H_{\ell c}$ and $H_{rc}$. Thus the invariant holds throughout the algorithm. □

We have an immediate corollary of the above invariant.

**Corollary 2.** *At Step 5 of Alg. 10 both the heaps $H_{\ell c}$ and $H_{rc}$ have the same min-key.*

Before we move to the analysis, we state and prove the following lemma which characterizes the pairs of vertices which are present in Marked-Pairs at the end of cleanup procedure. The set Marked-Pairs is used in fix-up to avoid generating the same set of paths twice.

**Lemma 14.** *A pair $(a, y)$ belongs to Marked-Pairs at the end of cleanup procedure iff all of the following hold:*

- *In $G$ there exists an SP from $a$ to $y$ which contains the updated vertex $v$.*
- *In $G$ there exists an SP from $a$ to $y$ which avoids the updated vertex $v$.*
- *In $G$ there exists an LSP either of the form $x \to a \rightsquigarrow y$ (here $x \neq v$) or of the form $a \rightsquigarrow y \to y'$ (here $y' \neq v$).*

*Proof.* Let $(a, y)$ be a pair which belongs to Marked-Pairs at the end of cleanup. We show that all the three conditions listed above are met. Observe that a pair gets added to Marked-Pairs at the Step 5 of Alg. 11 while processing a set of locally shortest paths of the form $x \to a \rightsquigarrow y$. (An analogous step in process_right also adds pairs to Marked-Pairs.) In either of the cases, we note that $G$ contains an LSP of the form $x \to a \rightsquigarrow y$ or $a \rightsquigarrow y \to y'$ respectively. Since cleanup procedure considers only LSPs that contain the vertex $v$, it must be the case that there exists a *shortest* path from $a$ to $y$ which contains the updated vertex $v$. To see this, recall that every proper sub-path of a locally shortest path is a shortest path. Finally, in Step 4 of Alg. 11 (just before adding $(a, y)$ to Marked-Pairs) our algorithm checks whether $P^*_\ell(a, y)$ is non-empty. Since we are processing a triple containing $\ell$-tuples $(xa, y)$, all shorter weight LSPs (and SPs) from $a$ to $y$ have been processed. That is, all LSPs from $a$ to $y$ of strictly smaller weight and containing $v$ have been removed from the tuple system. Thus, at this stage if $P^*_\ell(a,y)$ is non-empty, then it implies that there is at least one shortest path from $a$ to $y$ which avoids the updated vertex $v$. This completes the *if* part of the proof.

To see the other direction, assume that a pair $(a,y)$ satisfies all the three conditions of the lemma statement. We claim that $(a,y)$ must belong to Marked-Pairs at the end of cleanup. W.l.o.g. assume that $G$ contains an LSP of the form $x \to a \rightsquigarrow y$. Since cleanup processes all LSPs containing the updated vertex $v$, a triple $\gamma$ containing this path gets extracted at Step 6 of Alg. 10 at some point during cleanup. We note that since $G$ contains an SP from $a$ to $y$ that avoids $v$, therefore $P^*_\ell(a,y)$ will continue to remain non-empty even after completion of cleanup. Thus when $\gamma$ is processed at Step 2–6 of Alg. 11, the if condition in Step 4 is satisfied and the pair $(a,y)$ is inserted into Marked-Pairs. An analogous argument proves it for the case when an LSP of the form $a \rightsquigarrow y \to y'$ is present in $G$. $\square$

### 4.2.3 Analysis of cleanup

In this section we prove the correctness of the cleanup procedure and analyze its running time. We argue that at the end of cleanup procedure, the LSPs that contain the vertex $v$ have been removed from the data structures, while all the others LSPs are maintained in the tuple-system. This is achieved by changing the counts in the respective triples. Furthermore the left and the right extension sets $L, L^*, R, R^*$ are correctly maintained. Before we prove the main lemma in this section (Lemma 17), we discuss the need for additional book-keeping in our case since we deal with a set of paths instead of a single path (as in DI). In fact without the careful book-keeping, it is possible that the same set of paths are generated twice during the cleanup (as well as later during fixup). Below we discuss when a set of paths can potentially get generated twice the checks in our algorithm that avoids the same.

**Tuples Generated Twice**

Consider any set of LSPs of the form $x \to a \rightsquigarrow y \to y'$ in $G$. Any such set of paths contributes to the counts of two different triples in our tuple system – a triple $\gamma_\ell$ containing the left-tuple $(xa, y')$ and a triple $\gamma_r$ containing the right tuple $(x, yy')$. For the moment assume that none of $x, a, y$ and $y'$ is equal to the updated vertex $v$. This set of paths can be generated either by right-extending $(xa, y)$ or by left-extending $(a, yy')$. Note that this is true for every set of paths and is also true in case of DI. Let us consider the case of DI, that is, when there is a unique shortest path between any pair of vertices. Assume that the tuple $(xa, y)$ is considered for right extension during cleanup and the LSP of the form $(xa, yy')$ is generated. Now the path $x \to a \rightsquigarrow y \to y'$ is deleted from the respective $P_\ell, P_r$ and $P^*_\ell, P^*_r$ (if the path is a shortest path). However, note the subtle but important point. Due to the unique shortest path assumption, this is the *only* LSP of the form $(xa, by)$. Therefore once we delete the LSP $x \to a \rightsquigarrow y \to y'$ from the tuple system, $x$ is deleted from $L(a, yy')$ as well as $y'$ is deleted from $R(xa, y)$. Thus, at a later point in the algorithm, when the tuple $(a, yy')$ is considered for left extension, $x$ is no longer present in $L(a, yy')$. Thus, the path $(xa, by)$ gets generated exactly once under the DI assumption.

Now consider the general case when $x \to a \rightsquigarrow y \to y'$ represents a set of paths instead of a single path. Since this set of paths is generated during cleanup, it implies that at least one of these paths contains the updated vertex $v$. We consider two cases that can arise:

1. All the paths represented by $x \to a \rightsquigarrow y \to y'$ use the updated vertex $v$. We note that when any of one $x, a, y, y'$ is equal to the updated vertex $v$ we are in this case. But we may fall in this case despite none of $x, a, y, y'$ being equal to the updated vertex $v$.

2. A subset of paths represented by $x \to a \rightsquigarrow y \to y'$ use the updated vertex $v$ whereas a subset of the paths avoid the updated vertex $v$.

We show below that Case (1) is similar to the unique shortest paths assumption and therefore the set of paths get generated exactly once during cleanup. In Case (2) we can potentially generate the set of paths twice but we show that our algorithm avoids this by making a simple check in Step 4 of Alg. 11.

**Lemma 15.** *Let $\gamma = ((xa, y), wt, ct)$ be a triple in $S_\ell$ that is input to Alg. 11 and let $y' \in R(xa, y)$ before the start of cleanup. Furthermore, let $wt(x \to a \rightsquigarrow y) < wt(a \rightsquigarrow y \to y')$. Then, $y' \in \mathcal{RE}_c$ at Step 9, Alg. 11 (process_left) and $x \notin \mathcal{LE}_c$ at an analgous step in process_right. That is, during cleanup the triple $\gamma' = ((xa, yy'), wt + wt(y, y'), ct)$ is generated as a right extension* **only** *and does not get generated as a left extension.*

*Proof.* We divide the proof into two cases depending on the shortest paths from $a$ to $y$ in $G$.

- **In $G$, all shortest paths from $a$ to $y$ use $v$:** In this case, it is clear that all LSPs of the form $x \to a \rightsquigarrow y \to y'$ use $v$. Note that the assumption that all shortest paths from $a$ to $y$ are via $v$ and that cleanup has processed every triple of strictly smaller weight than $wt$ imply that at Step 4 of Alg. 11 we must have $P^*(a, y) = \emptyset$. Thus the set $\mathcal{RE}_c$ contains the complete set of right extensions of $(xa, y)$ and therefore contains $y'$. Thus, in Step 10 the set of paths $x \to a \rightsquigarrow y \to y'$ get generated as right extension. It remains to argue that these paths do not get generated as a left-extension. For this, we examine steps that get executed due to the call to the function right_extend ($\gamma$, $y'$) in Step 10 of Alg. 11. Steps 1–5 of Alg. 13 (right_extend) generate the appropriate left and right triples for the set of paths $x \to a \rightsquigarrow y \to y'$ and delete them from the relevant $P$ and $P^*$. Since all shortest paths from $a$ to $y$ use the updated vertex $v$, it must be the case that after right extending $(a\times, y)$ all paths of the form $(a \rightsquigarrow y \to y')$ must have been deleted from the tuple system. Thus at Step 7 of Alg. 13 our algorithm will delete $x$ from $L(a, yy')$.

Therefore when the triple of the form $(a, yy')$ is extracted from the heap $H_{rc}$, the vertex $x$ no longer belongs to $L(a, yy')$ and hence the paths do not get generated as a left extension.

- **In $G$, there exists at least one shortest path from $a$ to $y$ via $v$ and at least one shortest path from $a$ to $y$ that avoids $v$:** In this case we note that, when the execution of cleanup reaches Step 4 of Alg. 11, it must be the case that $P^*(a, y) \neq \emptyset$. This is because the shortest paths from $a$ to $y$ that do not use the vertex $v$ continue to exist in $P^*(a, y)$. However, due to the assumption that $wt(x \to a \leadsto y) < wt(a \leadsto y \to y')$, it must be the case that $wt(x, a) < wt(y, y')$. Therefore $y' \in \mathcal{RE}_c$. Thus, it is clear that the set of paths get generated as a right extension. To see that the paths do not get generated as a left extension, we consider Alg. 12 process_right. When invoked with the triple of the form $(a, yy')$ for left extension, in Step 4, the set $P_r^*(a, y) \neq \emptyset$ and $y' \neq v$. Here, the algorithm checks whether $x$ needs to be added to $\mathcal{LE}_c$. However, since $w(x, a) < w(y, y')$, the vertex $x$ does not get added to the set $\mathcal{LE}_c$. Thus, our algorithm does not generate the set of paths as a left-extension.

This completes the proof of the lemma. $\square$

### Correctness

Using Lemma 16 we show that at the end of cleanup procedure the extension sets are correctly maintained. In Lemma 17 we show that at the end of cleanup all paths containing $v$ have been deleted from the tuple system and the counts of triples are correctly maintained.

**Lemma 16.** *The $L$ and $R$ sets as well as the $L^*$ and $R^*$ sets are correctly maintained at the end of cleanup.*

*Proof.* We argue about the sets $L$; symmetric arguments apply for the sets $R$. We show that at the end of cleanup:

- $x \in L(a, yy')$ if and only if there exists an LSP in $G$ of the form $(xa, yy')$ that avoids the updated vertex $v$.

We note that if $G$ contains LSPs of the form $(xa, yy')$, then before the execution of cleanup $x \in L(a, yy')$ and $y' \in R(xa, y)$. Let $wt$ be the weight of LSPs represented by $(xa, yy')$. We break our proof into 3 cases depending on whether $v$ lies on none of the paths, a non-empty subset of the paths or all of the paths represented by $(xa, yy')$.

Case 1: Before cleanup, the vertex $v$ lies on *none* of the paths represented by $(xa, yy')$: In this case, $x$ continues to belong to $L(a, yy')$ at the end of cleanup. This is because cleanup does generate any path of the form $(xa, yy)$. Thus $x \in L(a, yy')$ remains unmodified.

Case 2: Before cleanup, the vertex $v$ lies on *a non-empty proper subset* of the paths represented by $(xa, yy')$: In this case, our goal is to show that $x$ continues to belong to $L(a, yy')$ at the end of cleanup. We remark that the only steps in cleanup that delete vertices from the sets $L$ and $R$ are Step 5 and Step 7 in Alg. 13 (right_extend) (and symmetric steps in left_extend). In particular, $x$ can be deleted from $L(a, yy')$ during the right extension of the triple $(xa, y)$ using the vertex $y'$. We show below that this deletion is not possible.

In this case at least one, *but not all* of the LSPs represented by $(xa, yy')$ use the updated vertex $v$. This implies that in $G$ there exists at least one LSP of the form $(xa, yy')$ which avoids the vertex $v$. This in-turn implies that in $G$ there exists at least one SP of the weight $wt_1 = wt - \mathbf{w}(x, a)$ and of the form $(a, yy')$ which avoids the vertex $v$. The $r$-triple corresponding to $(a, yy')$ continues to exist in $P_r^*(a, y')$ even after cleanup has removed LSPs (and SPs) of weight $wt_1$. Now consider the call of right_extend (Alg. 13) with the triple of the form $(xa, y)$ with the vertex $y'$. The check at Step 6, Alg. 13 fails (since $P_r^*(a, y')$ contains the triple corresponding to $(a, yy')$ ) and therefore $x$ does not be deleted from $L(a, yy')$.

Case 3: Before cleanup, the vertex $v$ lies on all of the paths represented by $(xa, yy')$: In this case, our goal is to show that $x$ does not belong to $L(a, yy')$ at the end of cleanup. Our assumption that all LSPs of the form $(xa, yy')$ use the updated vertex $v$, implies that

- all shortest paths from $a$ to $y$ in $G$ pass through the updated vertex $v$.
- all shortest paths from $x$ to $y$ of the form $(xa, y)$ in $G$ pass through the updated vertex $v$.
- all shortest paths from $a$ to $y'$ of the form $(a, yy')$ in $G$ pass through the updated vertex $v$.

Thus, after cleanup removes all LSPs (in fact SPs) of weight strictly smaller than $wt$, we are guaranteed that :

$(i)$ the sets $P_\ell^*(a, y)$ and $P_r^*(a, y)$ are empty.

$(ii)$ the triple corresponding to $(xa, y)$ has been deleted from $P_\ell^*(x, y)$.

$(iii)$ the triple corresponding to $(a, yy')$ has been deleted from $P_r^*(a, y')$.

Assume for the sake of contradiction, that $x$ is not deleted from the set $L(a, yy')$. Furthermore, assume that this the first set for which our algorithm makes an error. We now observe that the either the call to right_extend $(xa, y)$ with the vertex $y'$ or the call to left_extend$(a, yy')$ with vertex $x$ must happen during the course of the algorithm. To see this, assume w.l.o.g. that $\mathbf{wt}(x, a) < \mathbf{wt}(y, y')$. Since (by assumption), all the $L$ and $R$ sets are maintained correctly upto this point, it is clear that $y' \in R(xa, y)$ (since it was present at the start of the cleanup algorithm). Consider the call to process_left (Alg. 11) which includes the triple corresponding to $(xa, y)$. The check at Step 4 of 11, fails (since $P^*_\ell(a, y)$ is empty by $(i)$ above). Thus, the set $\mathcal{RE}_c$ is set to $R(xa, y)$. This ensures that right_extend $(xa, y)$ with the vertex $y'$ is invoked.

Now consider the call of right_extend (Alg. 13) with the triple of the form $(xa, y)$ with the vertex $y'$. The check in the if condition at Step 6, Alg. 13 passes (since $P^*_r(a, y')$ does not contain the triple corresponding to $(a, yy')$ by $(iii)$ above) and therefore $x$ gets deleted from $L(a, yy')$. This contradicts our assumption that $x$ does is not deleted from $L(a, yy')$. □

**Lemma 17.** *At the end of cleanup, the counts of triples in $P_\ell$, $P_r$ ($P^*_\ell$, $P^*_r$) represent the number of LSPs (SPs) in $G$ that do not pass through $v$.*

*Proof.* We argue about the correctness of $\ell$-triples, a similar argument holds for $r$-triples. Furthermore, since a subset of triples in $P_\ell$ are present in $P^*_\ell$, it suffices to argue correctness of counts for $P_\ell$. We begin by noting that at the end of cleanup, the counts of LSPs containing a single edge are correct. If, for an edge $(x, y)$, neither of $x$ or $y$ is equal to $v$, then the count in the corresponding triple remains unmodified by cleanup (the count is equal to 1 before and after cleanup). If $x$ or $y$ is equal to $v$, the count of the triple (which was orignially 1) is decremented by 1 and hence the triple gets deleted from $P_{xy}$.

Assume for the sake of contradiction that for some LSP containing more than one edge, the corresponding triple has an incorrect count at the end of cleanup. W.l.o.g. let $(x, yy')$ be the first such triple for which cleanup made an error. Assume that before cleanup, the number of LSPs in $G$ of the form $(x, yy')$ is $c$. In addition, assume that in $G$ the number of LSPs of the form $(x, yy')$ which pass through $v$ are $c^v$ (where $c^v \leq c$). Since the count is incorrect after cleanup, let the count after cleanup be $c' \neq c - c^v$. We note that the triple cannot represent a trivial path of zero or one edges. Therefore assume that the paths represented by $(x, yy')$ contain two or more edges. Let $A = \{a_1, a_2, \ldots, a_k\}$ be a set of vertices such that for $a_i \in A$,

$(xa_i, yy')$ are LSPs in $G$. Note that the only way to generate paths of the form $(x, yy')$ is either by left extending $(a_i, yy')$ or right extending $(xa_i, y)$. Assume that before cleanup, $c_{a_i}$ denotes the number of LSPs in $G$ of the form $(xa_i, yy')$ and $c^v_{a_i}$ denotes the number of LSPs in $G$ of the form $(xa_i, yy')$ that pass through $v$. This implies that $c = \sum_{a_i \in A} c_{a_i}$ and $c^v = \sum_{a_i \in A} c^v_{a_i}$. We now note the following:

- $(i)$ For each $a_i \in A$, for the triples corresponding to $(xa_i, y)$ the counts are correctly maintained, since $(x, yy')$ is the first time the algorithm makes an error. Thus, for each $a_i \in A$, $c_{a_i}$ paths are decremented from the triple $(xa_i, y)$ and the triple $((xa_i, y), wt, c^v_{a_i})$ is added to $H_{\ell c}$ for extension.
- $(ii)$ The paths of the form $(xa_i, yy')$ are generated by cleanup either by left-extension or by right extension but not both. This follows from Lemma 3.5.
- $(iii)$ The sets $L$ and $R$ are correctly maintained during cleanup. This follows from Lemma 3.6.

Assume w.l.o.g. for each $a_i$, the paths $(xa_i, yy')$ are generated by right extension. We consider the call to right_extend (Alg. 13) for $(xa_i, y)$ with the vertex $y'$. Step 2 generates the triple of the form $(xa_i, yy')$ with count $c^v_{a_i}$ and this count is decremented from $P_r(x, yy')$. Since this holds for every $a_i$, $\sum_{a_i \in A} c^v_{a_i} = c^v$ gets decremented from $c$ due to these extensions. We finally remark that these are the only triples that can generate paths of the form $(x, yy')$ and hence the count of triple corresponding to $(x, yy')$ is correctly decremented by $c^v$ during cleanup. This contradicts our assumption that at the end of cleanup the count is not equal to $c - c^v$. □

## Time Complexity

In this section we bound the complexity of cleanup.

**Lemma 18.** *For an update on a vertex $v$, Alg. 10 takes $O(\nu^{*2} \cdot \log n)$ time.*

*Proof.* The cleanup algorithm processes $\ell$-triples or $r$-triples each of which contain the updated vertex $v$. However, note that during the execution of cleanup, an $\ell$-triple (symmetrically $r$-triple) of the form $(xa, y)$ may get updated multiple times (counts being modified) due left-extension of $r$-triples of the form $(a, b_j y)$. Thus the overall processing done in the algorithm can be charged to full-tuples that contain the updated vertex $v$. By Lemma 13, the number of full-tuples containing $v$ are bounded by $O(\nu^{*2})$. Accounting for a $O(\log(n))$ for data-structure operations, we conclude that the running time of cleanup is $O(\nu^{*2} \cdot \log n)$ time. $\square$

## 4.3 Challenges for Multiple Paths

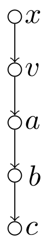
In this section, we will discuss the challenges and limitations posed due to dealing with multiple paths. In order, we show why a set of paths could be generated twice (Section 4.3.1), why our algorithms need to create $\ell$-tuple and $r$-tuple out of a single extension (Section 4.3.2), and finally the limits of our tuple-system approach (Section 4.3.3).

### 4.3.1 Generating a set of paths twice

As seen in cleanup, a set of paths can get generated twice, once via left extension and the second time via right extension. This happens because an $\ell$-tuple $(xa, y)$ representing a set of paths (unlike a single path as in DI) can contain a subset of paths which use the updated vertex $v$ and the remaining subset of paths avoid the vertex $v$. In Section 4.2.3 we have discussed how cleanup ensures that a set of paths gets generated exactly once. We will show later in fixup that the data-structure Marked-Pairs, populated during cleanup, avoids double generation of paths.

### 4.3.2 Need to create $\ell$-tuple and $r$-tuple both during extension

We first remark that the tuple-system used in this version of the paper is different from the one in the conference version [NPR14a]. In the current version, a set of paths of the form $x \to a \rightsquigarrow b \to y$ contribute to the counts of an $\ell$-tuple $(xa, y)$ and an $r$-tuple $(x, by)$. On the other hand, in [NPR14a], the paths contributed to counts of the *full*-tuple $(xa, by)$. The modified representation is used due to the ability to significantly reduce the space used by our algorithms. It also allows us to avoid the *accumulation* of tuples [NPR14a] sharing either the same first or the same last edge.



| Operation | extn. | tuple | extn. | tuple |
|---|---|---|---|---|
| Initialization | | $(vv, v)$ | | $(v, vv)$ |
| 1 extension | $a \in R(vv, v)$ | $(v, va)$ | $x \in L(v, vv)$ | $(xv, v)$ |
| 2 extensions | $x \in L(v, va)$ | $(xv, a)$ | $a \in R(xv, v)$ | $(x, va)$ |
| 3 extensions | $b \in R(xv, a)$ | $(x, ab)$ | $L(x, va) = \emptyset$ | – |
| 4 extensions | $L(x, ab) = \emptyset$ | | | – |

Figure 4.1: Evolution of triples removed during an incorrect 'cleanup' phase performed on $v$, in graph $G$ which is a path on 5 vertices. Note that after 4 extensions, both branches cannot generate additional triples. Thus, the path from $x$ to $c$ is never generated in such an approach although it is a shortest path in the graph.

We now consider extending paths in the $\ell$-tuple, $r$-tuple system. Consider an $\ell$-triple containing $(xa, y)$ which is extended using $y'$ (see for example Step 1–2, Alg. 13). Our algorithm creates two triples one containing the $\ell$-tuple $(xa, y)$ and another containing the $r$-tuple $(x, yy')$. A rather simplified approach is to only right-extend left triples and left-extend right triples. Here, we show that this natural approach does not suffice and some paths will never get generated. Consider a graph which is a simple path $x \to v \to a \to b \to c$. Let $v$ be the vertex on which the increase-only update happens. During cleanup $H_{\ell c}$ and $H_{rc}$ get initialized with the trivial triples corresponding to $v$, namely $(vv, v)$ and $(v, vv)$. Let us assume that we only right-extend $\ell$-tuples and left-extend $r$-tuples, during cleanup. The paths that get generated during such extensions are shown in Table 4.1.

Thus the path $x \rightsquigarrow c$ never gets generated if we only right-extend $\ell$-tuples and left-extend $r$-tuples. This is precisely the reason why we generate both the $\ell$-triple and $r$-triple even when right-extending an $\ell$-tuple (as in Step 2 of Alg. 13) and later in fixup algorithm as well.

### 4.3.3 Limits of the tuple-system

In this section we construct an explicit counterexample and a worst case increase-only update sequence. Observe that all the dynamic algorithms presented in chapters 3,4,5 and 6, generate full tuples in order to maintain shortest paths of the same weight. We will show that there exists a pathological graph where a single increase-only update is able to delete and create $\Theta(n^4)$ distinct SPs where $\Theta(n^4) = \Theta(\nu^{*2})$. Moreover, each one of these SPs must be handled by a unique full tuple. In fact our lower bound applies to any APASP algorithm which generates full tuples during its execution in order to update multiple SPs. With this counter example we have reached the limits of our tuple-system approach, where our $\tilde{O}(\nu^{*2})$ amortized results are near-optimal, and can only be improved by a polylogarithmic factor. We start by defining an explicit graph as showed in figure 4.2.

The graph $G = (V, E)$ (Figure 4.2) consists of $n+1$ vertices. Let $k = \frac{n}{4}$. For convenience, we divide the graph in two parts: the *left portion* which is the subgraph induced on vertices with subscripts from 1 up to $\frac{k}{2}$ (included) together with $v$, and the *right portion* which is the subgraph induced on vertices with subscripts from $\frac{k}{2} + 1$ to $k$ together with $v$. We now describe the edge set with its initial weights (see also the caption of Fig. 4.2). The edge set of $G$ is defined as follows:

- **Edges for which weights do not change:**
  - **Left-Short (LS) edges**: Set of edges where, for all $1 \leq p \leq k/2$ and $1 \leq q \leq k/2$, there is an edge for each pair $(x_p, a_q)$, and an edge for each

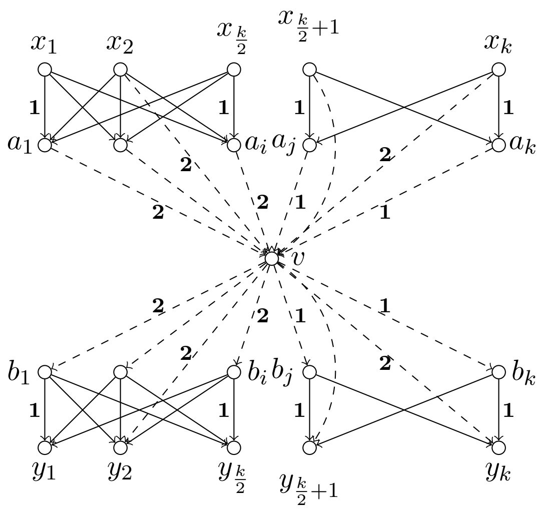


Figure 4.2: A worst case example for the number of tuples generated. Here $|V| = n + 1$ and $k = n/4$. The initial set-up for the edge set is defined as follows: For all $1 \leq p \leq k/2$ and $1 \leq q \leq k/2$ there is an edge for each pair $(x_p, a_q)$ with $\mathbf{w}(x_p, a_q) = 1$, and an edge for each pair $(b_p, y_q)$ with $\mathbf{w}(b_p, y_q) = 1$. For all $k/2 < p \leq k$ and $k/2 < q \leq k$ there is an edge for each pair $(x_p, a_q)$ with $\mathbf{w}(x_p, a_q) = 1$, and an edge for each pair $(b_p, y_q)$ with $\mathbf{w}(b_p, y_q) = 1$. These edges will never change weights. The remaining edges (represented with dashed lines) will change because of the updates on $v$. They are defined as follows: for $1 \leq i \leq \frac{k}{2}$, both the edges $(a_i, v)$ and $(v, b_i)$ have weight 2; while for all $k/2 < j \leq k$, both the edges $(a_j, v)$ and $(v, b_j)$ have both weight 1. Finally, for all $1 \leq p \leq k$, both the edges $(x_p, v)$ and $(v, y_p)$ have weight 2. The graph undergoes a sequence of alternating $\alpha$-updates and $\beta$-updates.

pair $(b_p, y_q)$; all these edges have weight 1.

- **Right-Short (RS) edges**: Set of edges where, for all $k/2 < p \leq k$ and $k/2 < q \leq k$, there is an edge for each pair $(x_p, a_q)$, and an edge for each pair $(b_p, y_q)$; all these edges have weight 1.

These weights on these edges never change throughout the update sequence.

- **Edges for which weights change:** The remaining edges (represented with dashed lines in Fig. 4.2) change weights due to increase-only updates on vertex

$v$. The dashed edges and their initial weights are as follows:

- **Left-Medium (LM) edges**: Set of edges where, for all $1 \leq i \leq k/2$, there is an edge for each pair $(a_i, v)$, and an edge for each pair $(v, b_i)$; these edges start with weight 2.
- **Right-Medium (RM) edges**: Set of edges where, for all $k/2 < j \leq k$, there is an edge for each pair $(a_j, v)$, and an edge for each pair $(v, b_j)$; these edges start with weight 1.
- **Left-Long (LL) edges**: Set of edges where, for all $1 \leq i \leq k/2$, there is an edge for each pair $(x_i, v)$, and an edge for each pair $(v, y_i)$; these edges start with weight 2.
- **Right-Long (RL) edges**: Set of edges where, for all $k/2 < j \leq k$, there is an edge for each pair $(x_j, v)$, and an edge for each pair $(v, y_j)$; these edges start with weight 2.

Let $e_{LS}$ be an edge $e \in E$ that is also in the set $LS$. In a similar way, we can define $e_{RS}, e_{LM}, e_{RM}, e_{LL}$ and $e_{RL}$. To compute the value of $v^*$, we observe that all the edges (except for the set $LS$) are on a shortest path through $v$. Thus, we have:

$$\nu^* = |RS| + |RM| + |RL| + |LM| + |LL| = 2 \cdot \left(\frac{n}{8}\right)^2 + 4 \cdot \left(\frac{n}{4}\right) = \Theta(n^2)$$

The overall idea is to show that a bad update sequence can force $\Theta(n^4)$ SPs to switch from the left portion to the right portion and vice versa after each increase-only update. Moreover, each one of these SPs is maintained in a different full tuple.

Note that, in its initial state, $G$ contains more than $(n/8)^4$ SPs through $v$ in its right portion (that is $\Theta(\nu^{*2})$ in our example), defined by edges in the sets

$\{RS, RM\}$. These SPs are contained in the induced subgraph of $G$ on the set of vertices $\{x_j, a_j, v, b_j, y_j \mid \text{for all } k/2 < j \leq k\}$. Each path has the form $\langle e_{RS}, e_{RM}, e_{RM}, e_{RS}\rangle$ and can be distinguished in $G$ by the first and last edge. This implies that each one of the $(n/8)^4$ SPs will be contained in a separate full tuple. (There are other shortest paths in the right portion, but they are not relevant for our lower bound.) The left portion of $G$ has only $\Theta(n^2)$ SPs through $v$, defined by edges in the set $LL$ and they are contained in the induced subgraph of $G$ on the set of vertices $\{x_i, v, y_i \mid \text{for all } 1 \leq i \leq k/2\}$. Each path is of the form $\langle e_{LL}, e_{LL}\rangle$.

Our update progression consists of an infinite sequence of an $\alpha$-update followed by a $\beta$-update, defined as follows:

1. $\alpha$-update. The vertex $v$ undergoes the following increase-only changes: for all $1 \leq p \leq k$, we increase by 1 the weight of $(x_p, v)$ and $(v, y_p)$ (the $LL$ and $RL$ edges). For all $\frac{k}{2} + 1 \leq j \leq k$, we increase by 2 the weight of $(a_j, v)$ and $(v, b_j)$ (the $RM$ edges).

2. $\beta$-update. The node $v$ undergoes the following increase-only changes: for all $1 \leq p \leq k$, we increase by 1 the weight of $(x_p, v)$ and $(v, y_p)$ (the $LL$ and $RL$ edges). For all $1 \leq i \leq \frac{k}{2}$, we increase by 2 the weight of $(a_i, v)$ and $(v, b_i)$ (the $LM$ edges).

After an $\alpha$ update the set of $(n/8)^4$ SPs in the right portion shifts to the left portion of the graph, and it is constituted by edges in the sets $\{LS, LM\}$. These SPs are contained in the induced subgraph of $G$ on the set of vertices $\{x_i, a_i, v, b_i, y_i \mid \text{for all } 1 \leq i \leq k/2\}$ which contains the $\Theta(n^4)$ SPs. Each path is of the form $\langle e_{LS}, e_{LM}, e_{LM}, e_{LS}\rangle$ and, as in the previous case, can be distinguished in $G$ by the first and last edge. Similarly, a $\beta$ update restores the previous scenario where the set of $\Theta(n^4)$ SPs are in the right portion. The value of $\nu^*$ does not change after an $\alpha$ or $\beta$ update, since the set of $\Theta(n^4)$ SPs moves from one portion of $G$ to the other simply mirroring the previous layout of the paths. In general, after $t$ *combined* updates (where a combined update consists of an $\alpha$-update followed by a $\beta$-update), the weight of each dashed edge will increase by $2t$ over its initial weight.

After each $\alpha$ or $\beta$ update, $\Theta(n^4) = \Theta(\nu^{*2})$ distinct SPs completely change their location and weight. Hence, any APASP algorithm that correctly maintains SPs via full tuples (or needs to generate full tuples to update multiple SPs) must access and modify the memory for at least each distinct SP. This enforces at least $\Theta(\nu^{*2})$ operations per update.

## 4.4 The Fixup Algorithm

The goal of the fixup procedure is to add all *new* SPs and *new* LSPs to the tuple-system (recall Definition 1). The SPs and LSPs which do not contain the updated vertex $v$ continue to remain in the tuple-system with correct weight and count. In Section 4.4.1 we give an overview of the fixup algorithm. We then give the detailed pseudo-code for fixup in Section 4.4.2. Section 4.4.3 discusses the invariants of fixup. In Section 4.4.4 we prove the correctness of fixup and analyze its running time.

### 4.4.1 Overview of Fixup

As in cleanup, the fixup procedure (pseudo-code in Alg. 14) works with two heaps of triples $H_{\ell f}$ and $H_{rf}$, which are initialized with *candidate* shortest path triples for each pair of vertices. The algorithm repeatedly extracts from these heaps the set of triples with minimum key and processes them. The main invariant (similar to DI [DI04]) is that the weight of the first set of triples, for a pair $x, y$, extracted from each heap gives the shortest path distance from $x$ to $y$ in the updated graph. Thus, these triples are all identified as shortest path triples and we need to extend them if in fact they represent *new* shortest paths. Note that, if triples do not contain *new* shortest paths, their extensions are correctly maintained and we do not need to process them.

Since we deal with a set of paths, it is sometimes necessary in fixup to address only the *new* paths going through the updated vertex $v$. This happens when the shortest distance for a given pair of nodes does not increase after the update. To readily identify triples containing paths through $v$ we use additional book-keeping: for every triple $\gamma$ examined during the fixup phase, we store the update number (update-num($\gamma$)) and a count of the number ($paths(\gamma, v)$) of paths in that triple that pass through $v$. These allow us to easily identify shortest triples containing $v$ that will be extended to LSTs, and the corresponding number of shortest paths that pass through $v$. *New* shortest triples are added to the appropriate data structures by Alg. 15; moreover, the same algorithm identifies the exact number of *new* shortest paths within a triple that will be extended in the next step.

Finally, similar to cleanup, the fixup procedure also left and right extends triples to create *new* locally shortest triples. This task is performed by Alg. 16. As in cleanup, we need to avoid generating the same triple twice, in order to maintain the correct count of paths. In Section 4.4.4 we prove that we indeed generate every path (including *new* shortest paths) exactly once. In fact, during fixup, we avoid double generation of paths by using the data-structure Marked-Pairs populated during the cleanup phase. The fixup algorithm (Alg. 14) invokes fixup_process_left (Alg. 15) which in turn invokes fixup_right_extend (Alg. 16). Below, we give a full description of this algorithm.

### 4.4.2 Pseudo-code

Alg. 14 (**fixup**) begins by initializing the heap $H_{\ell f}$ in Steps 3–9 as follows (symmetric steps are performed for $H_{rf}$): For every edge incident on $v$, it creates a trivial $\ell$-triple $\gamma_\ell$ which is inserted into $H_{\ell f}$ and the corresponding $P_\ell(\cdot)$. For example, for edge $(u, v)$ we create $(uv, v)$ and $(u, uv)$ and we insert them in the respective data structures. It also sets update-num($\gamma_\ell$) and paths($\gamma_\ell, v$) for each generated $\gamma_\ell$.

Then, for every $x, y \in V$, it adds a single candidate min-weight triple from $P_\ell(x, y)$ to $H_{\ell f}$ (even though $P_\ell(x, y)$ may contain several min-weight triples; this is done for efficiency).

The main while loop in Step 10 of Alg. 14 extracts all the min weight triples from both $H_{\ell f}$ and $H_{rf}$. These will be processed by fixup_process_left (Alg. 15) which will add *new* shortest triples to the $P^*_\ell$ and $P^*_r$, updating $L^*$ and $R^*$ corresponding data structures, and will create a set $S_\ell$ of *new* shortest triples to be extended.

**Algorithm 14** fixup($v, \mathbf{w}'$)

```
1: $H_{\ell f} \leftarrow \emptyset$; $H_{rf} \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2: let $round \leftarrow$ current update number
3: for each edge incident on $v$ do
4:     create triples $\gamma_\ell$ and $\gamma_r$
5:     set $paths(\gamma_\ell, v) \leftarrow 1$; update-num($\gamma_\ell$) $\leftarrow round$ ; add $\gamma_\ell$ to $H_{\ell f}$ and to $P_\ell(\cdot)$
6:     set $paths(\gamma_r, v) \leftarrow 1$; update-num($\gamma_r$) $\leftarrow round$ ; add $\gamma_r$ to $H_{rf}$ and to $P_r(\cdot)$
7: for each $x, y \in V$ do
8:     add a min-weight triple from $P_\ell(x, y)$ to $H_{\ell f}$
9:     add a min-weight triple from $P_r(x, y)$ to $H_{rf}$
10: while $H_{\ell f} \neq \emptyset$ and $H_{rf} \neq \emptyset$ do
11:     extract in $S'_\ell$ all triples with same min-key from $H_{\ell f}$;
        let $[wt, x, y]$ be the min-key of $H_{\ell f}$; $S_\ell \leftarrow \emptyset$
12:     fixup_process_left($S'_\ell$)
13:     extract in $S'_r$ all triples with same min-key from $H_{rf}$;
        the min-key of $H_{rf}$ will also be $[wt, x, y]$; $S_r \leftarrow \emptyset$
14:     fixup_process_right($S'_r$)
```

**fixup_process_left** – Alg. 15 checks whether for the pair $x, y$, the set $S'_\ell$ is the first set of triples (all of weight $wt$) extracted from $H_{\ell f}$. We claim (Invariant 7) that $wt$ denotes the shortest path distance from $x$ to $y$ in the updated graph. (Note that, if $S'_\ell$ is not the first extraction for the pair $x, y$ we do not process it.) The goal of steps 4–11 in Alg. 15 is to add the *new* paths, collected for a pair of nodes during the first extraction from $H_{rc}$ and $H_{\ell c}$ (Steps 11 and 13, Alg. 14), to the corresponding $P^*_\ell$ and $P^*_r$, to update the sets $L^*$ and $R^*$ accordingly, and to create a set $S_\ell$ of triples that represent *new* shortest paths which will be extended to LSTs. Note that this step is considerably more involved than the corresponding step in DI. In DI, only a single path $p$ is extracted from $H_{\ell f}$ possibly resulting in a *new* shortest path from $x$ to $y$. If $p$ is *new* then it is added to $P_\ell^*$ and $P_r^*$, thus the algorithm extends it to create *new* LSPs. In our case, we extract not just multiple paths but several shortest path triples from $x$ to $y$, and not all of these triples need to be in $H_{\ell f}$. We now describe how our algorithm handles the *new* shortest paths in Steps 4–11. Note that in DI, for each pair of nodes, only a single path is extracted/processed and eventually added as a *new* shortest path to their dataset.

Handling *new* shortest paths – As mentioned above, the challenging task of Alg. 15 is to create a set $S_\ell$ of triples that represent the *new* shortest paths. There are two cases to consider.

- $P_\ell^*(x,y)$ is empty: Here, we process the triples in $S'_\ell$, but in addition, we may be required to process triples of weight $wt$ from the set $P_\ell(x,y)$. To see this, consider the example in Fig. 3.1 and consider the pair $a_1, b_1$. In $G$, there is one shortest path $\langle a_1, v, b_1\rangle$ which is removed from $P_\ell(a_1,b_1)$ and $P_\ell^*(a_1,b_1)$ during cleanup. In the updated graph $G'$, where $\mathbf{w}(a_1,v) = 10$ and $\mathbf{w}(a_2,v) = 5$, the distance $d(a_1,b_1) = 4$ and there are 2 shortest paths, namely $p_1 = \langle a_1, b_1\rangle$ and $p_2 = \langle a_1, v_1, b_1\rangle$. Note that both of these are LSPs in $G$ and therefore are present in $P_\ell(a_1,b_1)$. In Step 8, Alg. 14 we insert exactly one of them into the heap $H_{\ell f}$. However, both need to be processed and also left and right extended to create *new* locally shortest paths. Thus, under this condition, we examine all the min-weight triples present in $P_\ell(a_1,b_1)$. We will show in invariant 8 that the additional triples added from $P_\ell$ are in fact *new* shortest triples.

- $P^*(x,y)$ is non-empty: After an increase-only update, the shortest distance from $x$ to $y$ can either remain the same or increase, but it cannot decrease.

Further, cleanup removed from the tuple-system all paths that contain $v$. Hence, if $P^*_\ell(x,y)$ is non-empty at this point then all the paths in $P^*_\ell(x,y)$ avoid $v$. In this case, we show (Invariant 8) that it suffices to only examine the triples present in $H_{\ell f}$. Furthermore, the only paths that we need to process are the paths that pass through the vertex $v$.

**Algorithm 15** fixup_process_left($S'_\ell$)

```
1: if $S'_\ell$ is the first extracted (shortest) set of tuples from $H_{\ell f}$ for $x, y$ then
2:     $S_\ell \leftarrow \emptyset$
3:     {Steps 4–11: add *new* STs (or increase counts of STs) from $x$ to $y$ to $S_\ell$.}
4:     if $P^*_\ell(x,y)$ is empty then
5:         for each $\gamma'_\ell \in P_\ell(x,y)$ with weight $wt$ do
6:             let $\gamma'_\ell = ((xa', y), wt, count')$
7:             add $\gamma'_\ell$ to $P^*_\ell(x,y)$ and $S_\ell$; add $x$ to $L^*(a', y)$
8:     else
9:         for each $\gamma'_\ell \in S'_\ell$ containing a path through $v$ do
10:            let $\gamma'_\ell = ((xa', y), wt, count')$
11:            add $\gamma'_\ell$ with $paths(\gamma'_\ell, v)$ in $P^*_\ell(x,y)$ and $S_\ell$; add $x$ to $L^*(a', y)$
12:    fixup_right_extend($S_\ell$)
```

**fixup_right_extend** – Alg. 16 gets as its input the set $S_\ell$ of STs all having the same key. The goal is to right extend these triples to generate LSPs of larger weight. The loop at Step 2 (Alg. 16) considers every triple $\gamma_\ell \in S_\ell$ and extends it using *appropriate* extensions. As in cleanup, we need to guard against the possible double generation of paths. Steps 4–8 build the set $\mathcal{RE}^*_f$ which contains the extensions that will be used to right extend the triple under consideration. Here we make use of the data structure Marked-Pairs which was populated during cleanup. In particular, consider a triple $(xa', y), wt, count')$ being right extended during Step 9, Alg. 16. In case, $(a', y) \in$ Marked-Pairs we select in $\mathcal{RE}^*_f$ only those $y' \in R^*(a', y)$ which satisfy the property that $w(y, y') > w(x, a')$. In case, $(x, a')$ does not belong to Marked-Pairs, $\mathcal{RE}^*_f$ is equal to the set $R^*(a', y)$. We prove in Section 4.4.4 that this selection of vertices ensures that every set of paths is generated exactly once. Note that the conference version [NPR14a] follows a different approach, which correctly maintains the tuple-system but requires more space in the worst case. The rest of Alg. 16 proceeds similarly to DI by inserting the newly generated LST into the corresponding data structures $P, L$ and $R$, with the notable difference that now for a generated LST $(xa', yy')$ we populate both $P_r(x, y')$ with a right-tuple $(x, yy')$ and $P_\ell(x, y')$ with a left-tuple $(xa', y')$.

**Algorithm 16** fixup_right_extend($S_\ell$)

```
1: {Steps 2–21: add *new* LSTs (or increase counts of existing LSTs) that extend
   SPs from $x$ to $y$.}
2: **for** every $\gamma_\ell \in S_\ell$ **do**
3:    $\gamma_\ell = ((xa', y), wt, count')$;
4:    **if** $(a', y) \in$ Marked-Pairs **then**
5:       $\mathcal{RE}^*_f = \{\ y' \ \mid\ y' \in R^*(a', y)$ and $\mathbf{w}(y, y') > \mathbf{w}(x, a')\ \}$
6:       {Note that this step in fixup_left_extend($S_r$) is slightly different. We place
         in $\mathcal{LE}^*_f$ all $x' \in L^*(x, b')$ such that $\mathbf{w}(x', x) \geq \mathbf{w}(b', y)$.}
7:    **else**
8:       $\mathcal{RE}^*_f \leftarrow R^*(a', y)$
9:    **for** every $y'$ in $\mathcal{RE}^*_f$ **do**
10:      $wt' \leftarrow wt + \mathbf{w}(y, y')$;
11:      $\gamma'_\ell \leftarrow ((xa', y'), wt', count')$ ; $\gamma'_r \leftarrow ((x, yy'), wt', count')$
12:      set update-num($\gamma'_\ell$); $paths(\gamma'_\ell, v) \leftarrow paths(\gamma, v)$; add $\gamma'$ to $H_{\ell f}$
13:      perform symmetric steps for $\gamma'_r$
14:      **if** a triple for $(xa', y')$ exists in $P_\ell(x, y')$ **then**
15:         add $\gamma'_\ell$ with $paths(\gamma'_\ell, v)$ in $P_\ell(x, y')$
16:      **else**
17:         add $\gamma'_\ell$ to $P_\ell(x, y')$; add $x$ to $L(a', yy')$ and $y'$ to $R(xa', y)$
18:      **if** a triple for $(x, yy')$ exists in $P_r(x, y')$ **then**
19:         add $\gamma'_r$ with $paths(\gamma'_r, v)$ in $P_r(x, y')$
20:      **else**
21:         add $\gamma'_r$ to $P_r(x, y')$; add $x$ to $L(a', yy')$ and $y'$ to $R(xa', y)$
```

### 4.4.3 Invariants

Fixup maintains the following invariants. Invariant 7 is proved similarly to Invariant 3.1 in [DI04]. The proof of Invariant 8 requires a careful analysis of various cases to show that indeed all *new* shortest paths are inserted into the set $S$.

**Invariant 7.** *If the set $S'_\ell$ in Step 11 of Alg. 14 is the first extracted set from $H_{\ell f}$ for $x, y$, then the weight of each triple in $S'_\ell$ is the shortest path distance from $x$ to $y$ in the updated graph. (A symmetric proof holds for $S'_r$ extracted from $H_{rf}$ in Step 13 of Alg. 14)*

*Proof.* Assume for the sake of contradiction that the invariant is violated at some extraction. Thus, the first set of triples $S'_\ell$ of weight $\hat{wt}$ extracted for some pair $(x, y)$ does not contain shortest paths from $x$ to $y$ in the updated graph. Consider the earliest of these events and let $\gamma = ((xa, y), wt, count)$ be a $\ell$-triple in the updated graph that represents a set of shortest paths from $x$ to $y$ with $wt < \hat{wt}$. The triple $\gamma$ cannot be present in $H_{\ell f}$, else it would have been extracted before any triple of weight $\hat{wt}$ from $H_{\ell f}$. Moreover, $\gamma$ cannot be in $P_\ell(x, y)$ at the beginning of fixup otherwise $\gamma$ (or some other $\ell$-triple of weight $wt$) would have been inserted into $H_{\ell c}$ during Step 8 of Alg. 14. Thus $\gamma$ must be a *new* locally shortest $\ell$-triple generated by the algorithm.

Since all edges incident on $v$ are added to $H_{\ell f}$ during Step 5 of Alg. 14 and $\gamma$ is not present in $H_{\ell f}$, implies that $\gamma$ represents paths which have at least two or more edges. For $1 \leq i \leq k$, we define left$(\gamma)$ as the set of locally shortest $\ell$-tuples of the form $((xa, b_i), wt - \mathbf{w}(b_i, y), count_i)$ that, when right extended to $y$, would generate $\gamma$; similarly we define right$(\gamma)$ as the set of locally shortest $r$-tuples of the form $((a, b_i y), wt - \mathbf{w}(x, a), count_i)$ that, when left extended to $x$, would generate $\gamma$. Note that since $\gamma$ is a shortest path tuple, all the paths represented in left$(\gamma)$ and right$(\gamma)$ are necessarily shortest paths. Moreover, since $\gamma$ is a *new* triple generated by the algorithm, all of the paths in either left$(\gamma)$ or right$(\gamma)$ are *new* shortest paths and therefore not present in $P^*_\ell$ or $P^*_r$ at the beginning of fixup. Since edge weights are positive $(wt - \mathbf{w}(b_i, y)) < wt < \hat{wt}$ (for all $i$ in left$(\gamma)$) and $(wt - \mathbf{w}(x, a)) < wt < \hat{wt}$. As we extract triples from $H_{\ell f}$ and $H_{rf}$ in increasing order of weight (Steps 11 and

13, Alg. 14), and all extractions before the wrong extraction were correct, the triples in left($\gamma$) (or right($\gamma$)) should have been extracted from $H_{\ell f}$ (or $H_{rf}$) and added respectively to $P^*_\ell$ (or $P^*_r$) in Step 7, Alg. 15. Thus, the triple corresponding to $(xa, by)$ of weight $wt$ should have been generated during a right (or left) extension of the above triples and inserted in $H_{\ell f}$ as an $\ell$-tuple $(xa, y)$ with weight $wt$ (Step 12, Alg. 16). Hence, some $\ell$-triple of weight $wt$ must be extracted from $H_{\ell f}$ for the pair $(x, y)$ before any triple of weight $\hat{wt}$ is extracted from $H_{\ell f}$. This contradicts our assumption that the invariant is violated. □

Using Invariant 8 below we show that fixup indeed considers all of the *new* shortest paths for any pair $x, y$.

**Invariant 8.** *The set of triples $S_\ell$ constructed in Steps 4–11 of Algorithm 15 represents all of the* new *shortest paths from $x$ to $y$, added to $P^*_\ell$ in the above steps. (A symmetric proof holds for the set of triples $S_r$)*

*Proof.* Any *new* SP from $x$ to $y$ is of the following three types:

1. a single edge incident to $v$ (such a path is added to $P_\ell(x, y)$, $P_r(x, y)$, $H_{\ell f}$ and $H_{rf}$ by the *for* loop at Step 3 of Alg. 14) (these paths trivially contain the updated node $v$),
2. a path that was an LSP but not SP before the update and is an SP after the update (this path does not contain the updated node $v$),
3. a path generated via left/right extension of some shortest path (such a path is added to $P_\ell(x, y)$, $P_r(x, y)$, $H_{\ell f}$ and $H_{rf}$ in Steps 12, 13 and 14–21 Alg. 16 and analogous steps in fixup_left_extend) (this path could be an extension of a shortest path containing the updated node $v$ or not).

In (1) and (3) above any *new* SP from $x$ to $y$ which is added to $H_{\ell f}$ is also added to $P_\ell(x, y)$ (Steps 14–17, Alg. 16) (same for $H_{rf}$ and $P_r(x, y)$). However, amongst the several triples representing paths of the form (2) listed above, only one candidate triple will be present in $H_{\ell f}$ or $H_{rf}$. In this case, for a given $x, y$, when we extract from $H_{\ell f}$ ($H_{rf}$) triples of weight $wt$, $P_\ell(x, y)$ ($P_r(x, y)$) could contain other triples with same weight $wt$ which differ from the ones extracted from $H_{\ell f}$ ($H_{rf}$). We now consider the two cases that the algorithm deals with. For simplicity we only study the case for $\ell$-tuples, noting that the same arguments apply to $r$-tuples.

- $P^*_\ell(x, y)$ is empty when the first set of triples for $x, y$ is extracted from $H_{\ell f}$ (Steps 5–7, Alg. 15). Here, all the shortest paths from $x$ to $y$ were passing through $v$. In this case, we process all the min-weight triples in $P_\ell(x, y)$. By the above argument, we know that all *new* SPs from $x$ to $y$ are present in $P_\ell(x, y)$. Therefore it suffices to argue that all of them are *new*. Assume for the sake of contradiction, some path $p$ represented by them is *not new*. By definition, $p$ does not contain $v$ and $p$ was a SP before the update. Therefore, clearly $p$ was in $P^*_\ell(x, y)$ before the update. However, since cleanup only removes paths that contain $v$, the path $p$ remains untouched during cleanup and hence continues to exist in $P^*_\ell(x, y)$. This contradicts the fact that $P^*_\ell(x, y)$ is empty.

- $P^*_\ell(x, y)$ is not empty when the first set of triples for $x, y$ is extracted from $H_{\ell f}$ (Steps 9–11, Alg. 15). Here, some of the shortest paths from $x$ to $y$ remained in the tuple-system after the cleanup phase. Let the weight of triples in $P^*_\ell(x, y)$ be $wt$. This implies that the shortest path distance from $x$ to $y$ before and after the update remains $wt$. Recall that we are dealing with increase-only updates. We first argue that it suffices to consider triples in $H_{\ell f}$. This is observed from the fact that any *new* SP of the form (1) and (3) listed above is present in $H_{\ell f}$. Furthermore, note that any path of form (2) above has a weight strictly larger than $wt$, since it was an LSP and not an SP before the update. Thus in the presence of paths of weight $wt$, none of the paths of form (3) are *new* candidates for shortest paths from $x$ to $y$. This justifies considering triples only in $H_{\ell f}$.

Finally, we note that for any triple considered, in the case when $P^*_\ell(x, y)$ is not empty, our algorithm only processes paths through $v$. This again follows from the fact that only paths through $v$ were removed by cleanup and possibly need to be restored if the distance via them remains unchanged after the update.

□

### 4.4.4 Analysis of Fixup

In this section we prove the correctness of the fixup procedure and analyze its running time. In the next section, we first give the complete characterization of LSPs that could be generated twice during fixup. Then, we proceed by proving correctness in Section 4.4.5, and the complexity of our algorithm in Section 4.4.6.

#### Tuples generated twice

In this section we give the characterization of LSPs that can potentially get generated twice during fixup.

**Lemma 19.** *Assume that none of $x, a, y, y'$ is equal to the updated vertex $v$. If $G'$ contains* new *LSPs of the form $(xa, yy')$, then during fixup these paths get generated either as a right extension of the triple $(xa, y)$ or as a left extension of the triple $(a, yy')$ but not as both.*

*Proof.* We break the proof into two parts.

**The pair $(a, y) \in$ Marked-Pairs:** As $(a, y) \in$ Marked-Pairs, by Lemma 14, the shortest path distance from $a$ to $y$ remains unchanged from $G$ and $G'$, and further in $G$ some SPs from $a$ to $y$ use the vertex $v$ whereas some SPs from $a$ to $y$ avoid the vertex $v$. In addition, we claim that the graph $G$ (before update on $v$) also contains LSPs of the form $(xa, yy')$. This is because $G'$ contains LSPs of the form

$(xa, yy')$, and the weights of $(x, a)$ and $(y, y')$ have not been altered by the update on $v$. This implies that $x \in L^*(a, y)$ and $y' \in R^*(a, y)$ at the beginning of cleanup. Since $(a, y) \in$ Marked-Pairs, these extensions continue to exist even at the end of cleanup procedure and the beginning of fixup. Now assume w.l.o.g. that $\mathbf{wt}(x, a) < \mathbf{wt}(y, y')$. Then at Step 5 of Alg. 16, we note that $y'$ gets added to $\mathcal{RE}^*_f$. Thus the paths of the form $(xa, yy')$ get generated as right extension. In addition, it is easy to see that in the symmetric procedure fixup_left_extend (of Alg. 16), the corresponding check at Step 5 fails and therefore $x \notin \mathcal{LE}^*_f$. Thus, the set of paths get generated as a right extension and not as a left extension in this case. A symmetric argument proves that when $\mathbf{wt}(x, a) \geq \mathbf{wt}(y, y')$ the paths get generated as a left extension only and not as a right extension.

**The pair $(a, y) \notin$ Marked-Pairs:** We note that we fall in this case when either (i) the shortest path distance from $a$ to $y$ strictly increases between $G$ and $G'$ or (ii) the shortest path from $a$ to $y$ remains unchanged between $G$ and $G'$ but all the shortest paths from $a$ to $y$ use the updated vertex $v$. Thus, just after cleanup $x \notin L^*(a, y)$ and $y' \notin R^*(a, y)$. This case is similar to the unique shortest paths case in DI. Thus, when the smaller amongst the two paths $(xa, y)$ and $(a, yy')$ is extracted from the corresponding heap, the LSPs of the form $(xa, yy')$ do not get generated. Hence, the LSPs of the form $(xa, yy')$ get generated exactly once when the larger of the two paths is identified as a set of shortest paths. $\square$

### 4.4.5 Correctness

**Lemma 20.** *After execution of Alg. 14, for any $(x, y) \in V$, the counts of the triples in $P_\ell$, $P_r$ ($P^*_\ell$, $P^*_r$) represent all the LSPs (SPs) in $G$ from $x$ to $y$ in the updated graph. Moreover, the sets $L, L^*, R, R^*$ are correctly maintained.*

*Proof.* We prove the lemma statement by showing the invariants are maintained by the while loop in Step 10 of Algorithm 14.

**Loop Invariant:** At the start of the $t$-th iteration of the while loop in Step 10 of Algorithm 14 let the min-key triples to be extracted and processed from $H_{\ell f}$ have key $[wt, x, y]$. We claim the following about the tuple-system and $H_{\ell f}$ (a symmetric proof holds for $H_{rf}$ and $r$-triples).

$\mathcal{I}_1$ For any $a \in V$, if $G'$ contains $c_a$ number of LSPs of weight $wt$ of the form $(xa, y)$, then an $\ell$-triple $\gamma = ((xa, y), wt, c_a)$ is present in $P_\ell(x, y)$ (note that $H_{\ell f}$ can also contain other $\ell$-triples from $x$ to $y$ with weight $wt$).

$\mathcal{I}_2$ Let $[\hat{wt}, \hat{x}, \hat{y}]$ be the last key extracted from $H_{\ell f}$ and processed before $[wt, x, y]$. For any key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, let $G'$ contain $c > 0$ number of SPs of weight $wt_1$ of the form $(x_1a_1, y_1)$, and let $B = \{b_1, b_2, \ldots, b_k\}$ be the set of the last nodes, before $y_1$, on these shortest paths. Further, let $c_{new}$ (resp. $c_{old}$) denote the number of such SPs of the form $(x_1a_1, y_1)$ that are *new* (resp. not *new*). Here $c_{new} + c_{old} = c$. Then,

(a) the $\ell$-triple $((x_1a_1, y_1), wt_1, c)$ is in $P^*_\ell(x_1, y_1)$ representing $c$ SPs.

(b) for all $b \in B$, $x_1 \in L(a_1, by_1)$ and $y_1 \in R(x_1a_1, b)$. Moreover $x_1 \in L^*(a_1, y_1)$.

(c) If $c_{new} > 0$, for every right extension $y' \in R(x_1a_1, y_1)$, an $\ell$-triple corresponding to $(x_1a_1, y')$ with weight $wt' = wt_1 + \mathbf{w}(y_1, y')$ and the appropriate count is in $P_\ell(x_1, y')$ and in $H_{\ell f}$ if $[wt', x_1, y'] \geq [wt, x, y]$. A similar claim can be stated for an $r$-tuple $(x_1, y_1y')$ with weight $wt'$ in $P_r(x_1, y')$ and in $H_{rf}$.

$\mathcal{I}_3$ For any key $[wt_2, x_2, y_2] \geq [wt, x, y]$, if any path of the form $(x_2a_2, b_2y_2)$ and weight $wt_2$ is generated either as a left or right extension but not both. Furthermore, a single $\ell$-triple of the form $(x_2a_2, y_2)$ and appropriate count will be inserted in the heap $H_{\ell f}$ (and similarly an $r$-triple of the form $(x_2, b_2y_2)$ in $H_{rf}$).

The proof that these invariants hold at initialization and termination, and are maintained at every iteration of the while loop, are shown below.

**Initialization:** We initialize the heaps $H_{\ell f}$ and $H_{rf}$ with the set of edges incident to $v$ (Steps 3–6, Alg. 14). Moreover, a min-weight triple for each pair $x, y$ is inserted in both $H_{\ell f}$ and $H_{rf}$ (Steps 8 and 9, Alg. 14). The first extraction (with key $[wt, x, y]$) from $H_{\ell f}$ is a min-weight edge in the graph. Thus $\mathcal{I}_1$ is easily verified. Since there is no key smaller than $[wt, x, y]$ already extracted, $\mathcal{I}_2$ is also true at initialization. Moreover, since there is no key yet, bigger than $[wt, x, y]$ and generated by the fixup and inserted in $H_{\ell f}$, $\mathcal{I}_3$ holds at initialization.

**Maintenance:** Given that the min-key triple in $H_{\ell f}$ is $[wt, x, y]$ with weight $wt$, let $(xa, y)$ be any of the $\ell$-triples in $H_{\ell f}$ ready to be extracted. Let $B = \{b_1, b_2, \ldots, b_k\}$ be the set of the last nodes, before $y$ on the paths represented by $(xa, y)$. We consider the following triples:

- For each $j = 1, \ldots, k$, LSPs of the form $(a, b_j y)$ of weight $wt_a = wt - wt(x, a)$. The count of these paths is exactly $c_{ab_j}$.
- For each $j = 1, \ldots, k$, LSPs of the form $(xa, b_j)$ of weight $wt_b = wt - wt(b_j, y)$. The count of these paths is exactly $c_{ab_j}$.

Note that, the above two cases are the only set of triples which are able to generate the $\ell$-tuple $(xa, y)$. Since $wt_a$ and $wt_b$ are strictly smaller than $wt$, the loop invariant holds when both the heaps have min-key $[wt_a, a, y]$ or $[wt_b, x, b_j]$. Consider the iteration where min-key in the heaps is $[wt_a, a, y]$. Since Invariant $\mathcal{I}_3$ holds at the end of that iteration, we are guaranteed that for each $i = 1, \ldots k$, the triple $((a, b_j y), wt, c_{ab_j})$ is left-extended to generate LSPs of the form $(xa, b_j y)$ (having correct counts) each of which contribute to the $\ell$-triple $(xa, y)$. Thus the triple $((xa, y), wt, c_a)$ is *added* in $H_{\ell c}$ (Step 12, Alg.16), as well as added to $P(x, y)$ (Steps 15 or 19, Alg.16). This establishes $\mathcal{I}_1$. If an $\ell$-triple $(xa, y)$ is generated by a right extension of the $\ell$-triple $(xa, b)$, an analogous argument establishes $\mathcal{I}_1$. Note that, since a full triple is only generated once during fixup (Invariant $\mathcal{I}_3$), the count of paths is correct by induction.

To prove that Invariant $\mathcal{I}_2$ holds, we note that Invariant $\mathcal{I}_1$ holds for the $\ell$-tuple $(x_1a_1, y_1)$; hence, there is an $\ell$-triple $\gamma = ((x_1a_1, y), wt_1, c)$ in $P_\ell(x_1, y_1)$. If $\gamma$ was not modified during the fixup (thus it is not in $H_{\ell f}$ and $c = c_{old}$), and since $wt_1$ is a shortest distance from $x_1$ and $y_1$, then $\mathcal{I}_2$(a) immediately holds. Otherwise if $\gamma$ is also present in $H_{\ell f}$ then ($c_{new} > 0$), it has been already extracted from the heap and processed. Since $wt_1$ is a shortest distance, then $\gamma$ is contained within the first set of $\ell$-triples extracted for $x_1$ and $y_1$. Moreover, $\gamma$ contains $c_{new}$ *new* paths and they are added to $P^*_\ell(x_1, y_1)$ (Steps 4–11, Alg. 15). Thus $\mathcal{I}_2$(a) holds also in this case. For $\mathcal{I}_2$(b), recall that in $x_1$ is also added to $L^*(a_1, y_1)$ when $\gamma$ is processed as a shortest $\ell$-triple (Steps 7 or 11, Alg. 15). Moreover, each LSP represented in $\gamma$ was added to $H_{\ell f}$ only because LSP of the form $(x_1a_1, by_1)$, with $b \in B$ were formed during previous fixup iterations (in Alg. 16 and the symmetric). In that case, for all $b \in B$, $x_1$ was placed in $L(a_1, by_1)$ and $y_1$ in $R(x_1a_1, b)$ (Steps 17 and 21, Alg. 16). Finally, for $\mathcal{I}_2$(c), when $\gamma$ contains *new* paths ($c_{new} > 0$), then it is inserted into $S_\ell$ (Steps 7 or 11, Alg. 15) and processed by Alg. 16. During this last phase $\gamma$ is extended to every $y' \in \mathcal{RE}_f$ (which represents the set of valid right extensions to avoid double generations) to generate the tuples $(x_1a_1, y_1y')$. These are finally inserted in $H_{\ell f}$, $P_\ell(x_1, y')$, $H_{rf}$ and $P_r(x_1, y')$ (Steps 9 – 21, Alg. 16). This completes the proof of Invariant $\mathcal{I}_2$.

To prove that Invariant $\mathcal{I}_3$ holds, we note that a tuple is generated only as a left extension of an $r$-tuple or as a right extension of a $\ell$-tuple. In both cases, Lemma 19 shows that a tuple is only generated once during fixup. Thus every tuple generated and inserted into $H_{\ell f}$ and $H_{rc}$, after the current min-key, is the result of a left-extension od a right-extension but not both.

**Termination:** The exit condition of the while loop is when the heap $H_{\ell f}$ (and $H_{rc}$)

is empty. Because Invariant $\mathcal{I}_1$ maintains in $H_{\ell f}$ the first triple to be extracted and processed, then $H_{\ell f} = \emptyset$ implies that there are no more triples that need to be processed and eventually left or right extended. Moreover, since the invariants hold for the last set of triples of weight $\hat{wt}$ extracted from the heap, by $\mathcal{I}_2$(a), all LSPs having weight less than or equal to $\hat{wt}$ have been processed and inserted in the appropriate sets $P(\cdot)$ and $P^*(\cdot)$ if SPs. Finally, due to $\mathcal{I}_2$(b), the sets $L$,$R$, $L^*$,$R^*$ are also correctly maintained after the while loop terminates. □

### 4.4.6 Complexity of Fixup

We bound the number of accesses to $\ell$-triples during the course of the fixup algorithm (a symmetric bound applies for the accesses to $r$-triples). We note that an $\ell$-triple is *accessed* when the corresponding $\ell$-tuple is created (for the first time), deleted or its count is modified. An $\ell$-triple is accessed if and only if a smaller $\ell$-triple or $r$-triple is extended to generate the corresponding full-tuple. For example, the $\ell$-triple $((xa, y), wt, count)$ can only be accessed because of full-tuples of the form $(xa, b_i y)$ and weight $wt$, created as right extesion of $\ell$-tuples of the form $(xa, b_i)$ or as left extensions of $r$-tuples of the form $(a, b_i y)$. Thus, we will charge all the accesses to $\ell$-triples and $r$-triples to the full-tuples that are causing them. Note that fixup generates only *new* $\ell$-triples outside of the $O(n^2)$ $\ell$-triples added initially to $H_{\ell f}$. Lemma 21 below is similar in spirit to the complexity analysis presented in Chapter 3, however new ideas are used for the charging mechanism. Below we classify each $\ell$-triple $\gamma$ (and similarly each $r$-triple) as one of the following disjoint types. The same classification was also applied to full-tuples in Chapter 3.

- **Type-0 (contains-v):** $\gamma$ represents at least one path containing vertex $v$.
- **Type-1 (new-LST):** $\gamma$ was not an LST before the update but is an LST after the update, and no path in $\gamma$ contains $v$.

- **Type-2 (new-ST-old-LST):** $\gamma$ is an ST after the update, and $\gamma$ was an LST but not an ST before the update, and no path in $\gamma$ contains $v$.
- **Type-3 (new-ST-old-ST):** $\gamma$ was an ST before the update and continues to be an ST after the update, and no path in $\gamma$ contains $v$.
- **Type-4 (new-LST-old-LST):** $\gamma$ was an LST before the update and continues to be an LST after the update, and no path in $\gamma$ contains $v$.

The following lemma establishes an amortized bound for fixup which is the same as the worst case bound for cleanup.

**Lemma 21.** *The fixup procedure takes time $O(\nu^{*2} \cdot \log n)$ amortized over a sequence of $\Omega(m^*/\nu^*)$ increase-only updates.*

*Proof.* We bound the total number of accesses to $\ell$-triples (a similar argument applies to $r$-triples); for each access the time taken is $O(\log n)$ due to the data structure operations performed on a triple. The initialization in Steps 1–9 of Alg. 14 access at most $O(n^2)$ $\ell$-triples. We now consider the number of accesses performed on the triples examined after Step 9 of Alg. 14.

As mentioned before, we charge an access to an $\ell$-triple to the corresponding full-tuple. We address the Type-0, Type-3 and Type-4 $\ell$-triples first. The number of accesses performed on Type-0 triples is bounded by the number of full-tuples containing the updated vertex $v$. By Lemma 13, the number of full-tuples containing $v$ is $O(\nu^{*2})$. Thus the number of accesses to Type-0 triples is bounded by $O(\nu^{*2})$. For Type-3 and Type-4, we note that for any $x, y$ we add exactly one candidate min-weight $\ell$-triple from $P_\ell(x, y)$ to $H_{\ell f}$ (Steps 8–9 of Alg. 14), hence initially there are at most $n^2$ such triples in $H_{\ell f}$. Moreover, we never process/access an old LST which is not an ST so no additional Type-4 triples are accessed during fixup. Finally, $\ell$-triples in $P_\ell^*$ that are not placed initially in $H_{\ell f}$ are not accessed in any step of fixup, so no additional Type-3 triples are examined.

We now argue about the Type-1 and Type-2 triples. The number of accesses performed on Type-1 triples is addressed by amortizing over the entire set of Type-1 full-tuples generated during the update sequence as described in the last paragraph below. To analyze the Type-2 triples we recall that our updates are increase-only. Thus an $\ell$-tuple $(xa, y)$ with weight $wt$, which was not an LST, can become an LST after an update, at a later point can become an ST (after another vertex update) and finally cease to be an ST (after yet another vertex update). However, never in future for any update will $(xa, y)$ with weight $wt$ become an LST again. We use this fact crucially to charge the accesses to the Type-2 triple to the corresponding full-tuples that made this an LST for the first time. We also argue, in the following argument, that the full-tuples which are paying for those accesses are indeed Type-1 full-tuples. Consider an $\ell$-triple $\gamma = ((xa, y), wt, count)$ of Type-2 which is moved into $S_\ell$ from $P_\ell(x, y)$ during step 7 of Alg. 15. Here $\gamma$ was never an ST before the current update; thus every access performed to $\gamma$ is paid for by a full-tuple of the form $(xa, b_i y)$ that was generated (as a left or right extension of some triple) for the first time as an LST (Steps 12 and 13, Alg. 16 in a previous update). This is the exact definition of Type-1 full-tuple. Further, each Type-1 full-tuple is generated only once during the entire update sequence.

Thus the number of accesses on $\ell$-triples by a call to fixup is $O(\nu^{*2})$ plus $O(X)$, where $X$ is the number of *new* full-tuples that fixup generates during its execution. Each Type-1 $\ell$-triple is a *new* triple added to the tuple-system and, by Lemma 12, we cannot have more than $O(m^* \cdot n)$ of them. However, to bound the total cost of accessing these triples we need to amortize the number of full-tuples generated by our algorithm. Let $\sigma$ be the number of updates in the update sequence. During cleanup, at most $O(\sigma \cdot \nu^{*2})$ full-tuples are generated and removed from $G$. There can be at most $O(m^* \cdot \nu^*)$ full tuples remaining at the end of the sequence (by Lemma 12), hence the total number of *new* full tuples generated by all fixups in the update sequence is $O(\sigma \cdot {\nu^*}^2 + m^* \cdot \nu^*)$. When $\sigma > m^*/\nu^*$, the first term dominates, and this gives an average of $O({\nu^*}^2)$ full-tuples generated per fixup phase, and the desired amortized time bound for fixup. $\square$

### Space Analysis

Both cleanup and fixup algorithms only store $\ell$-tuples or $r$-tuples in our tuple-system. These are bounded by Lemma 12 and there are no more than $O(m^* \cdot n)$ of them, at any time in $G$. During cleanup and fixup, a set $S_\ell$ of $\ell$-tuples (and $S_r$ for $r$-tuples) is prepared for extension. However, tuples in $S_\ell$ are extended and processed one at a time generating a single full-tuple for each extension: In cleanup, Alg. 13 generates the full-tuple $(xa, yy')$ for each extension $y'$ of $(xa, y)$ (Step 9, Alg. 11); similarly in fixup, Alg. 16 generates the full-tuple $(xa', yy')$ for each extension $y'$ of $(xa', y)$ (Step 9, Alg. 16). In both cases, the full-tuples are immediately removed after the corresponding $\ell$-tuple and $r$-tuple are updated with the correct count. Moreover, Marked-Pairs only marks pairs of nodes and it can be stored as a $O(n^2)$ binary matrix. All the other data structures have only linear size. Thus, the overall space complexity is bounded by $O(m^* \cdot n)$.

# Chapter 5

# Fully Dynamic Algorithm

In this chapter we present a fully dynamic algorithm for the APASP problem, where each update in $G$ is either decrease-only or increase-only[1]. A decrease-only update either inserts a new vertex along with incident edges of finite weight, or decreases the weights of some existing edges incident on a vertex. An increase-only update deletes an existing vertex, or increases the weights of some edges incident on a vertex.

We presented a simple decrease-only APASP algorithm [NPR14a] in Chapter 2, and a more involved increase-only APASP algorithm [NPR14b] in Chapter 3. Neither of these algorithms is correct for the fully dynamic case. The fully dynamic methods that we present in this chapter build on the techniques presented in Chapter 3, and also incorporate a variant of the fully dynamic methods developed by Demetrescu and Italiano [DI04] (the `DI` method) and Thorup [Tho04] (the `Thorup` method) for APSP where only one shortest path is maintained for each pair of vertices. The `DI` algorithm runs in $O(n^2 \cdot \log^3 n)$ amortized time per update, where $n = |V|$. The `Thorup` algorithm is faster by a logarithmic factor but is considerably more complicated, even for the unique shortest paths case. The algorithms in both [DI04] and [Tho04] use the unique shortest paths assumption

[1]The results presented in this chapter appeared in [PR15b].

crucially, and need considerable enhancements even to maintain a small number of multiple shortest paths correctly.

Here we present two fully dynamic algorithms for APASP, which maintain all of the multiple shortest paths for every pair of vertices. Our basic algorithm (presented in this Chapter) is as simple as the `DI` method (though somewhat different) when specialized to unique shortest paths. In fact, it matches the `DI` bound for graphs with a constant number of (or unique) shortest paths, while being applicable to the more general APASP problem. Moreover, it provides a new amortized analysis for the fully dynamic `DI` algorithm if our 'dummy sequence' replaces the one used in `DI`. Our second algorithm improves the amortized bound by a logarithmic factor using data structures and techniques that are considerably more complicated.

The results in this chapter are mainly for fully dynamic APASP, which is an important graph-theoretic property of independent interest. However, we show that our fully dynamic APASP algorithms give fully dynamic BC algorithms with the same bounds.

**Our Results.** Recall the definitions of $\nu^*$ in Section 1.1.1, Chapter 1. Our main results are the following theorems, where we assume for convenience $\nu^* = \Omega(n)$. We present two algorithms: FULLY-DYNAMIC (basic) and FFD (faster fully dynamic) with bounds stated in the following theorem.

**Theorem 9.** *Let $\Sigma$ be a sequence of $\Omega(n)$ fully dynamic APASP updates on an $n$-node graph $G = (V, E)$. Then,*

1. *algorithm* FULLY-DYNAMIC *mantains APASP and all BC scores in amortized time $O(\nu^{*2} \cdot \log^3 n)$ per update,*

2. *algorithm* FFD *mantains APASP and all BC scores in amortized time $O(\nu^{*2} \cdot \log^2 n)$ per update (in Chapter 6),*

*where $\nu^*$ bounds the number of distinct edges that lie on shortest paths through any* *given vertex in any of the updated graphs or their vertex induced subgraphs.*

Our algorithms are provably faster than Brandes on dense graphs with succinct single-source SP dags. Our techniques rely on recomputing BC scores using certain data structures related to shortest paths extensions (see Section 5.1). These are generalizations of structures introduced by Demetrescu and Italiano [DI04] for fully dynamic APSP and `Thorup`, where only one SP is maintained for each pair of vertices. Our generalizations build on the tuple-system introduced in Chapter 3 for increase-only APASP (see Section 3.1, Chapter 3), which is a method to succinctly represent all of the multiple SPs for every pair of vertices. Additionally, in algorithm FFD, one of the main challenges we address is to generalize the 'level graphs' of `Thorup` to the case when different SPs for a given vertex pair can be distributed across multiple levels.

Of independent interest is a new amortized analysis for the fully dynamic `DI` algorithm which we obtain using a new 'dummy updates' sequence in our FULLY-DYNAMIC algorithm.

In many real graphs $\nu^*$ behaves as discussed in Section 1.1.1, Chapter 1. In all such cases our improved algorithm FFD will run in amortized $O(n^2 \log^2 n)$ time per update. Thus we have:

**Theorem 10.** *Let $\Sigma$ be a sequence of $\Omega(n)$ updates on graphs with $O(n)$ distinct edges on shortest paths through any single vertex in any vertex-induced subgraph. Then, APASP and all BC scores can be maintained in amortized time $O(n^2 \cdot \log^2 n)$ per update.*

**Corollary 3.** *If the number of shortest paths through any single vertex is bounded by a constant, then fully dynamic APASP and BC have amortized cost $O(n^2 \cdot \log^2 n)$ per update if the update sequence has length $\Omega(n)$.*

Both our algorithms use $\widetilde{O}(m \cdot \nu^*)$ space, extending the $\widetilde{O}(mn)$ `DI` result for APSP.

Brandes uses only linear space, but all known dynamic algorithms require at least $\Omega(n^2)$ space.

**Overview of the Chapter.** In Section 5.1 we describe a very simple method to obtain a fully dynamic BC algorithm using a fully dynamic APASP algorithm. The rest of the thesis will focus only on obtaining efficient algorithms for APASP. In Section 5.2 we briefly review the `NPRdec` increase-only APASP algorithm and the fully dynamic `DI` algorithm. In Section 5.3 we describe our basic fully dynamic APASP algorithm FULLY-DYNAMIC, its properties and complexity analysis. Pseudocode and correctness are in Section 5.4. Finally, our analysis appears in Section 5.4.3.

## 5.1 Fully Dynamic Betweenness Centrality

The static Brandes algorithm [Bra01] computes BC scores in a two phase process. The first phase computes the SP out-dag for every source through $n$ applications of Dijkstra's algorithm. The second phase uses an 'accumulation' technique that computes all BC scores using these SP dags in $O(n \cdot \nu^*)$ time.

In our fully dynamic algorithm, we will leave the second phase unchanged. In the first phase, we implicitly maintain the SP dags with some of the structures maintained by our algorithms. In contrast, a different approach was previously used in the decrease-only BC algorithm in Chapter 2, where the SP dags were explicitly maintained. In the `NPRdec` algorithm the SP dags are also implicitly maintained.

We now describe a very simple method to construct the SP dags from the following structures. For every vertex pair $x, y$, consider the following sets $R^*(x,y)$, $L^*(x,y)$:

- $R^*(x,y)$ contains all nodes $y'$ such that every shortest path $x \rightsquigarrow y$ in $G$ can be extended with the edge $(y,y')$ to generate another shortest path $x \rightsquigarrow y \rightarrow y'$.

- $L^*(x,y)$ contains all nodes $x'$ such that every shortest path $x \rightsquigarrow y$ in $G$ can be extended with the edge $(x',x)$ to generate another shortest path $x' \rightarrow x \rightsquigarrow y$.

These sets were introduced in `DI` and allow us to construct the SP dag for each source $s$ using the following simple algorithm BUILD-DAG.

**Algorithm 17** BUILD-DAG$(G, s, \mathbf{w}, D)$ ($\mathbf{w}$ is the weight function; $D$ is the distance matrix)

```
1: for each $t \in V$ do
2:     for each $u \in R^*(s,t)$ do
3:         if $D(s,t) + \mathbf{w}(t,u) = D(s,u)$ then add the edge $(t,u)$ to dag($s$)
```

Our fully dynamic algorithms will maintain the $R^*$ and $L^*$ sets. More precisely, in our algorithms $R^*$ and $L^*$ will be supersets of the exact collections of nodes defined above, but the check in Step 3 will ensure that only the correct SP dag edges are included. The combined sizes of these $R^*$ and $L^*$ sets is $\tilde{O}(n \cdot \nu^*)$ in our algorithms, hence the amortized time bound for the overall fully dynamic BC algorithm is dominated by the time bound for computing fully dynamic APASP. In the rest of this chapter, we will present our fully dynamic APASP algorithms.

## 5.2 Background

Our fully dynamic APASP algorithms build on the approach used in the elegant fully dynamic APSP algorithm of Demetresu and Italiano [DI04] for unique shortest paths (the '`DI`' method). Initially, the `DI` method solves the increase-only APSP problem. Then, the authors extend it to a fully dynamic algorithm. For APASP, in Chapter 3 we presented a increase-only algorithm (the '`NPRdec`' method). However, its extension to a fully dynamic algorithm for APASP presents several challenges that we address in our results. We now briefly review these two methods (`DI` and `NPRdec`), and then we give an overview of our methods for fully dynamic APASP.

### 5.2.1 The NPRdec Increase-Only APASP Algorithm

The `NPRdec` algorithm maintains all STs and LSTs in the current graph, and for each tuple, it maintains the $L$, $R$, $L^*$ and $R^*$ sets. To execute a new update to a vertex $v$, `NPRdec` (similar to `DI`) first calls an algorithm <u>cleanup</u> on $v$ which removes all STs and LSTs that contain $v$. This is followed by a call to algorithm <u>fixup</u> on $v$ which computes all STs and LSTs in the updated graph that are not already present in the system. The overall algorithm <u>update</u> consists of cleanup followed by fixup. If the updates are all increase-only then `NPRdec` maintains exactly all the SPs and LSPs in the graph in $O(\nu^{*2} \cdot \log n)$ amortized time per update. Several challenges to adapting the techniques in the `DI` increase-only method to the tuple-system are addressed in Chapter 3. The analysis of the amortized time bound is also more involved since with multiple shortest paths it is possible for the dynamic APASP algorithm to examine a tuple and merely change its count; in such a case, the `DI` proof method of charging the cost of the examination to the new path added to or removed from the system does not apply.

### 5.2.2 The `DI` Fully Dynamic APSP Algorithm

The `DI` method first gives an increase-only APSP algorithm, and shows that this is also a correct, though inefficient, fully dynamic APSP algorithm. The inefficiency arises because under decrease-only updates the method may maintain some old SPs and their combinations that are not currently SPs or LSPs; such paths are called historical shortest paths (HPs) and locally historical paths (LHPs). To obtain an efficient fully dynamic algorithm, the `DI` method introduces 'dummy updates' into the update sequence. A dummy update performs cleanup and fixup on a vertex that was updated in the past. Using a strategically chosen sequence of dummy updates, it is established in [DI04] that the resulting APSP algorithm runs in amortized time $O(n^2 \cdot \log^3 n)$ per real update. The `DI` method continues to use the notation $P^*$, $L^*$, etc., even though these are supersets of the defined sets in a fully dynamic setting. We will do the same in our fully dynamic algorithms.

## 5.3 Basic FULLY-DYNAMIC APASP Algorithm

A natural approach to obtain a fully dynamic APASP algorithm would be to convert the `NPRdec` increase-only APASP algorithm to an efficient fully dynamic APASP algorithm by using dummy updates, similar to `DI`. There are two steps in this process, and each has challenges (the second step is more challenging).

**Step 1:** *Converting `NPRdec` increase-only algorithm to a correct (but inefficient) fully dynamic APASP algorithm.* In [DI04], the authors initially show an increase-only algorithm for APSP which is also correct for fully dynamic updates. However, this initial approach is generally inefficient and requires several enhancements to achieve efficiency.

For APASP, the `NPRdec` method gives an increase-only algorithm which, in this case, is not correct for fully dynamic updates. In Section 5.3.1 we describe algorithm FULLY-UPDATE which is a correct fully dynamic APASP algorithm that maintains a superset of all STs and LSTs in the current graph. This algorithm is very similar to `NPRdec` but contains several changes to ensure correctness under fully dynamic updates. In fact, additional features are required for the `NPRdec` data structures to ensure correctness when we deal with fully dynamic updates (see the beginning of Section 5.3.1). However, FULLY-UPDATE is not very efficient since it may add, remove, and examine a large number of tuples. This is similar to the `DI` increase-only algorithm when used as a fully dynamic algorithm without any additional features.

**Step 2:** *Obtaining a good 'dummy sequence' for efficient fully dynamic APASP.* In Section 5.4.3 we present Algorithm FULLY-DYNAMIC, which calls FULLY-UPDATE not only on the current update, but also on a sequence of vertices updated at suitable previous time steps. This is similar to the dummy updates used in `DI`, however we use a different dummy update sequence. We now describe the `DI` dummy update sequence and its limitations for APASP. Then, we introduce our new method (whose full details are in Section 5.4.3).

The DI method uses 'dummy updates', where a vertex updated at time $t$ is also given a 'dummy' update at steps $t+2^i$, for each $i \geq 0$ (this update is performed along with the real update at step $t+2^i$). The effect of a dummy update on a vertex $v$ is to remove any HP or LHP that contains $v$, thereby streamlining the collection of paths maintained. A useful property when SPs are unique (as in DI) is that each HP in $P^*(x,y)$, for a given pair $x, y$, will have a different weight. An $O(\log n)$ bound on the number of HPs in a $P^*(x,y)$ is established in DI as follows. Let the current time step be $t$, and consider an HP $\tau$ last updated at $t' < t$. Let us denote the smallest $i$ such that $t' + 2^i > t$ as the dummy-index for $\tau$. By observing that different HPs for $x, y$ must have different dummy-indices, it follows that their number is $O(\log t)$, which is $O(\log n)$ since the data structure is reconstructed after $O(n)$ updates.

### 5.3.1 Algorithm FULLY-UPDATE for APASP

We first extend the notions of historical and locally historical paths [DI04] to tuples and triples. In the following definition, we will consider a tuple $\tau$ over an interval of time $[t', t]$. If $\tau$ becomes an ST, after its creation at time $t'$, it will remain an HT until it is completely removed from the tuple-system.

Note that tuples are used instead of triples in the definition below. This distinction is relevant in our algorithm because during cleanup or fixup, a (historical) triple could change its count without losing its property of being an historical tuple. A similar behavior could also affect a locally historical tuple. With our definition, we can immediately refer to it without specifying the count of its associated triple.

**Definition 2** (HT, THT, LHT, and TLHT)**.** *Let $\tau$ be a tuple in the tuple-system at time $t$. Let $t' \leq t$ denote the time at which $\tau$ was originally added for the first time in the tuple-system. Then $\tau$ is a historical tuple (HT) at time $t$ if $\tau$ was an ST at* *least once in the interval $[t', t]$; $\tau$ is a true HT (THT) at time $t$ if it is not an ST in the current graph. A tuple $\tau$ is a locally historical tuple (LHT) at time $t$ if either it only contains a single vertex or every proper sub-path in it is an HT at time $t$; a tuple $\tau$ is a true LHT (TLHT) at time $t$ if it is not an LST in the current graph.*

The above definition is used extensively in our proof of correctness (see Section 5.4, lemmas 22 and 25). In particular, in the cleanup loop inveriant (Lemma 22), we show how each TLHT (and THT) is representing only paths that avoids the updated node $v$. Moreover, in the fixup loop invariant (Lemma 25), we require that each HT (LHT) maintains the correct count of HPs (LHPs) in the graph after the fixup phase.
In order to correctly extend $\ell$-tuple and $r$-tuple under fully-dynamic updates, we extend the following data structures from `NPRdec`.

$$\begin{aligned}
L(x, by) &= \{(x', wt') : (x', x) \in E(G) \text{ and } (x'x, by) \text{ is an LHT of weight } wt' \text{ in } G\}\\
R(xa, y) &= \{(y', wt') : (y, y') \in E(G) \text{ and } (xa, yy') \text{ is an LHT of weight } wt' \text{ in } G\}\\
L^*(x, y) &= \{(x', wt') : (x', x) \in E(G) \text{ and } (x'x, y) \text{ is an } \ell\text{-tuple of weight } wt'\\
&\qquad \text{representing HPs in } G\}\\
R^*(x, y) &= \{(y', wt') : (y, y') \in E(G) \text{ and } (x, yy') \text{ is an } r\text{-tuple of weight } wt'\\
&\qquad \text{representing HPs in } G\}
\end{aligned}$$

Note that $L^*$, $R^*$, $L$ and $R$ are now kept in a stack order, thus giving priority to the nodes that are inserted more recently. Another important data structure introduced for the cleanup phase is the *combined history* $CH(\gamma)$ of a triple $\gamma$. The combined history for $\gamma$ is an array of updates, and each update $t$ contains the number of paths that joined $\gamma$ during the $t$-th update. To efficiently implement $CH$ we also maintain the update number associated to each node. The structure $CH$ is initialized with update-num($v$) that is the last update in which $v$ was updated before this cleanup phase, associated with the value 1 representing the trivial tuple $(vv, vv)$. Additional data structures will be suggested in the description of the algorithm (see Section 5.4), but they are only implementation oriented.

Note that, if we try to apply the `DI` dummy sequence to APASP, we are faced with the issue that a new ST for $x, y$ (with the same weight) could be created at each update in a long sequence of successive updates. Then, a decrease-only update could transform all of these STs into HTs. If this happens, then several HTs for $x, y$, all with the same weight, could have the same dummy-index (in `DI` only one HP can be present for this entire collection due to unique SPs). Thus, the `DI` approach of obtaining an $O(\log n)$ bound for the number of HPs for each vertex pair does not work for HTs in our tuple-system.

Our method for Step 2 is to use a different dummy sequence, and a completely different analysis that obtains an $O(\log n)$ bound for the number of different 'PDGs' (a PDG is a type of derived graph defined in Section 5.4.3) that can contain the HTs. Our new dummy sequence is inspired by the 'level graph' method introduced in `Thorup` [Tho04] (see Chapter 6, where we generalize this approach to multiple shortest paths in our FFD algorithm) to improve the amortized bound for fully dynamic APSP to $O(n^2 \cdot \log^2 n)$, saving a log factor over `DI`. This method is complex because it maintains $O(\log n)$ levels of data structures for suitable 'level graphs'. Our algorithm FULLY-DYNAMIC does not maintain these level graphs. Instead, FULLY-DYNAMIC performs exactly like the fully dynamic algorithm in `DI`, except that it uses this alternate dummy update sequence, and it calls FULLY-UPDATE for APASP instead of the `DI` update algorithm for APSP. Our change in the update sequence requires a completely new proof of the amortized bound which we present in Section 5.4.3. We consider this to be a contribution of independent interest: If we replace FULLY-UPDATE by the `DI` update algorithm in FULLY-DYNAMIC, we get a new fully dynamic APSP algorithm which is as simple as `DI`, with a new analysis.

FULLY-UPDATE is given in the two-line Algorithm 18. It calls FULLY-CLEANUP and FULLY-FIXUP in sequence, on the updated vertex $v$ (see Section 5.4 for the complete pseudocode). This is similar to the update procedure in the increase-only and fully dynamic algorithms for unique paths in `DI` [DI04] and in the increase-only algorithm in Chapter 3.

The cleanup procedure removes from the tuple-system every HPs and LHPs containing $v$ by decrementing the count of triples which represent them. Thus, each TLHT (and THT) containing the updated vertex $v$ is updated in the tuple-system with its correct count even if some of its represented paths avoid $v$. To achieve this property for THTs, we use the new data structure $CH$ during the cleanup phase. FULLY-CLEANUP works by repeatedly extracting triples from a heap $H_c$, generated as extensions of the updated node $v$ using the data structures of the tuple-system.

The fixup phase adds to the tuple-system a superset of LSPs generated in the graph by the update: if a new LSP discovered during the fixup phase is of the form $x \to a \rightsquigarrow b \to y$ and weight $wt$, FULLY-FIXUP will increase the count of the tuple $\tau = ((xa, by), wt)$ by one (creating $\tau$ itself if not in the tuple-system). Also in this case, triples are extracted from a heap $H_f$, added to the system, and finally extended to candidate triples to be processed in future steps.

**Algorithm 18** FULLY-UPDATE($v, \mathbf{w}'$)

```
1: FULLY-CLEANUP($v$)
2: FULLY-FIXUP($v, \mathbf{w}'$)
```

In FULLY-UPDATE, the sets $P^*(x, y)$ and $P(x, y)$ will contain HTs (including all STs) and LHTs (including all LSTs), respectively, from $x$ to $y$. It was observed in [DI04] that the increase-only algorithm they presented for the unique SP case is a correct algorithm when decrease-only updates are interleaved with increase-only ones. However here, in contrast to `DI`, the increase-only APASP algorithm in Chapter 3 (the `NPRdec` algorithm) needs to be refined before it becomes correct for a fully dynamic sequence, and this is due to the presence of multiple shortest paths. Consider, for instance, a THT $\tau = (xa, by)$ with weight $wt$ which is currently an LST. Using the `NPRdec` algorithm, since $\tau$ is a THT it will be present in $P^*(x, y)$ (but not as an ST). Suppose a new set of paths represented by a new triple $\tau' = ((xa, by), wt, count')$, with the same weight $wt$, is added to the count of this tuple $\tau$. We cannot simply add $count'$ to $\tau$ in $P^*(x, y)$ because its extensions were performed using the old count, and if $\tau$ is restored as an ST in $P^*(x, y)$, these extensions will not have the correct count. (With unique SPs this situation can never occur.) Our solution here is to have $\tau$ in $P(x, y)$ with the larger correct count (thus including $count'$), and to leave the corresponding $\tau$ in $P^*(x, y)$ with its original count. Should $\tau$ later be restored as an ST then the difference in counts between $\tau$ in $P^*(x, y)$ and the corresponding $\tau$ in $P(x, y)$ will trigger left and right extensions of $\tau$ with the correct count even though $\tau$ is currently in $P^*(x, y)$ (see points [F.1] and [F.2] in Section 5.4.2).

There are additional subtleties. During FULLY-CLEANUP starting from the current updated vertex $v$, we may reach a triple $\gamma = (\tau, wt, count)$ in $P$ through say, a left extension, while the triple in $P^*$ for $\tau$ with weight $wt$ is $\gamma' = (\tau, wt, count')$, with $count' < count$ (the extreme case being that $count' = 0$, in which case there is no $\gamma'$ in $P^*$). This is an indication that the paths in $\gamma - \gamma'$ were formed after $\gamma'$ became a THT. Moreover, the number of paths going through $v$ in $\gamma' \in P^*$ could be different from the number of paths going through $v$ in $\gamma \in P$, posing an interesting dilemma on the correct number of paths to be removed from $\gamma' \in P^*$.

We address this situation in FULLY-CLEANUP by using the $CH(\gamma)$ data structure for the triple $\gamma$ in cleanup. In fact, we can easily check the last time $t$ that $\gamma'$ was updated, and remove from it only the paths in $CH(\gamma)$ that were created before or during update $t$. This technique guarantees that only the paths going through $v$ but truly represented by $\gamma'$ are removed from $P^*$.

The full pseudocodes for FULLY-CLEANUP and FULLY-FIXUP are given in th next section with the main new features highlighted. Recall that FULLY-UPDATE is simply an execution of FULLY-CLEANUP followed by FULLY-FIXUP. The correctness of this algorithm is argued by noting that our method ensures that when an ST in $P^*$ is processed during FULLY-FIXUP without further extensions, it has the correct weight and count and all of its extensions have been performed with that count; every ST and LST is generated starting with singleton edges, min-weight tuples from the $P$ sets, and correct STs from the $P^*$ sets, hence the counts of the tuples identified as STs and LSTs are maintained correctly.

## 5.4 Pseudocode and Correctness of FULLY-DYNAMIC Algorithm

Here we give the full pseudocode for FULLY-CLEANUP (Algorithm 19) and FULLY-FIXUP (Algorithm 20). They are similar to the corresponding pseudocode cleanup and fixup in Chapter 3, and we have marked the steps changed from these algorithms with a • at the end of the line. A description of the new features of the algorithms is given below, while a detailed description is available in Chapter 3. The other steps in Algorithm 20 are described in Chapter 3, as are the two parameters paths($\gamma, v$), which gives the number of paths containing the node $v$ that are represented by the triple $\gamma$, and update-num($\gamma$), which is a timestamp that indicates the last update in which the triple $\gamma$ is involved. Then, the correctness of the algorithms is established.

The only changes from Chapter 3 in Algorithm 19 are in steps 7, and 15 where we decrement any THT we encounter during cleanup, while extending from the updated node $v$, using the new data structure $CH$. More specifically, in step

**Algorithm 19** FULLY-CLEANUP($v$)

```
1: $H_c \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2: $\gamma \leftarrow ((vv, vv), 0, 1)$; add $\gamma$ to $H_c$
3: while $H_c \neq \emptyset$ do
4:    extract in $S$ all the triples with min-key $[wt, x, y]$ from $H_c$
5:    for every $b$ such that $(x\times, by) \in S$ do
6:       let $fcount'$ be the number of deleted paths of the form $((xa_i, by), wt)$
7:       for every $(x', wt') \in L(x, by)$ such that $((x'x, by), wt')$ is an LHT not in
         Marked-Tuples do
8:          $\gamma' \leftarrow ((x'x, by), wt', fcount')$; add $\gamma'$ to $H_c$
9:          remove $\gamma'$ in $P(x', y)$ // decrements its count in $P$ by $fcount'$
10:         if a triple for $((x'x, by), wt')$ exists in $P(x', y)$ then
11:            insert $((x'x, by), wt')$ in Marked-Tuples
12:         else
13:            delete $(x', wt')$ from $L(x, by)$ and delete $(y, wt')$ from $R(x'x, b)$
14:         if a triple $\gamma''$ for $((x'x, by), wt')$ exists in $P^*(x', y)$ with count paths then
15:            let $fcount''$ be the number of deleted paths of the form $((x'x, by), wt')$
               created during an update smaller (older) than update-num($\gamma''$) •
16:            remove $\gamma'$ in $P^*(x', y)$ // decrements count by $fcount''$
17:            if $P^*(x, y)$ doesn't contain triples of weight $wt$ then delete $(x', wt')$
               from $L^*(x, y)$
18:            if $P^*(x', b)$ doesn't contain triples of weight $wt' - \mathbf{w}(b, y)$ then delete
               $(y, wt')$ from $R^*(x', b)$
19:   perform symmetric steps 5 – 18 for right extensions
```

7, we use the weight associated with the extension to avoid the creation of cycles. An implementation to achieve this is the following: for a left extension $x'$, let $ldc$ be the set of paths of the form $(x'x, \times b)$ and weight $w' - \mathbf{w}(b, y)$ removed from $P^*$ during this cleanup phase (note that these paths can pe maintained in a temporary data structure of size $O(mn)$). Similarly, let $rdc$ be a the set of paths of the form $(x\times, by)$ and weight $w' - \mathbf{w}(x', x)$ removed from $P^*$ during this cleanup. If the updated node $v \not| y$ we set $fcount = \max(ldc, rdc)$, alias we only count all the locally historical paths of $(x'x, by)$ that are formed by a valid $l$-tuple and $r$-tuple both in $P^*$ at the beginning of the cleanup phase and now removed. Thus no cycle path is ever considered because, no cycle is ever created during the fixup phase and placed in $P^*$. In step 7, we deal with the delicate task of correctly decrementing an HT from the tuple-system. As discussed in Section 5.3.1, when we deal with a THT $\gamma''$ in $P^*$ we could only delete a subset of paths identified by the cleanup algorithm for its respecting LHT $\gamma$ in $P$ (note that if $\gamma''$ is a ST then we trivially subtract the same number $fcount$ of paths substracted from $\gamma$, since they have the some count). To address this issue, we use the new $CH(\gamma')$ data structure (in the algorithm $\gamma'$ is the triple to be deleted from the tuple-system): let $t =$ update-num$(\gamma'')$ be the most recent update in which $\gamma''$ was updated in $P^*$; in $CH(\gamma')$ we sum all the paths up to time $t$, we call this value $fcount''$. Finally, we decrement $fcount''$ paths from $\gamma'' \in P^*$. Now, we only need to address how to efficiently build $CH$ for a given triple $\gamma'$. Let us assume that $\gamma'$ is a left extension to node $x'$ for a set of $k$ triples of the form $x\times, by$. Each one of the non-extended triples has its own $CH_j$ data structure built from a previous cleanup iteration, with $j \leq k$. Then, for a given $CH_j$, for each pair $(\mathit{updnum}, \mathit{count})$ in $CH_j$ we add the pair $(\max(\text{update-num}(x'), \mathit{updnum}), \mathit{count})$ to $CH(\gamma')$. Note that, if such pair is already present in $CH(\gamma')$ with $\mathit{count}'$, we increment it by $\mathit{count}$. The complexity analysis for this implementation is given in Corollary 4.

Algorithm 20 introduces new features to achieve correctness and efficiency. We observe that we may revert an HT from, say, $x$ to $y$, back to an ST during an update, and this happens only if the shortest path distance from $x$ to $y$ increases. This condition translates into the new check in Step 9 of Algorithm 20. Here we proceed as in `NPRdec` keeping in mind that an LHT extracted from $P$ as an ST (Step 10, Algorithm 20) may or may not be in $P^*$. If the LHT is not in $P^*$ (Step 13, Algorithm 20) we add the triple to the tuple-system as in `NPRdec`. If the LHT is already in $P^*$ with a different count (Step 16, Algorithm 20), we replace the count of the triple in $P^*$ with $\mathit{count}'$ from the triple in $P$ and we add the triple to $S$ (Step 17, Algorithm 20). In step 10, Algorithm 20 we check the bit $\beta$ associated with the triple $\gamma$. Since $P$ is a priority queue, we will process only the triples in $P$ with min-key $[wt, 0]$, so we avoid examining the triples that are already in $P^*$ with a correct count. We set $\beta$ to 1 for any triple added to or updated in $P^*$ with the correct count (Steps 18 and 23, Algorithm 20). Also, for an LHT updated in $P$ and not $P^*$, we set $\beta = 0$ (Step 34, Algorithm 20). Finally, we use the $L^*$ and $R^*$ stacks to generate only LHT that are not cycles. We pop the extensions from the stack in reversed order of update time. This is because the newest extensions always refers to STs, while older extensions may refer to THTs. Whenever, we encounter an extension that does not satisfy the distance check at step 26, we know that the extensions left in the stack are associated to tuples that are even older than the current one; thus we can skip them. A possible implementation to avoid cycles is the following: for each triple $\gamma' = (x'x, by)$ that FF-FIXUP generates with an extension $(x', wt')$ to the left, we check if $P^*(x', b)$ contains a triple of the form $(x'x, \times b)$ and weight $w < wt'$. If this is the case, we do not extend to $x'$ because it will create a cycle. A symmetric check is applied to the right extensions.

### 5.4.1 Correctness of FULLY-DYNAMIC

In this section, we establish the correctness of Algorithm 19 and 20.

We assume that all the local structures are correct before the update, and we will show the correctness of them after the update. FULLY-CLEANUP works with a heap $H_c$ of triples. The algorithm maintains the loop invariant that any triple inserted into $H_c$ has already been deleted from the tuple-system: only its extensions remain to be processed and deleted. We prove the following lemma:

**Lemma 22.** *After Algorithm 19 (*FULLY-CLEANUP*) is executed, for any $(x, y) \in V$, the STs in $P^*(x, y)$ (LSTs in $P(x, y)$) represent all the SPs (LSPs) from $x$ to $y$ in $G$ that do not pass through $v$. Moreover, every THT (TLHT) present in the tuple-system represents a collection of HPs (LHPs) in $G$ that contains only paths that do*

**Algorithm 20** FULLY-FIXUP$(v, \mathbf{w}')$

```
$H_f \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
for each edge incident on $v$ do
    create a triple $\gamma$; set paths$(\gamma, v) = 1$; set update-num$(\gamma)$; add $\gamma$ to $H_f$ and to $P$
for each $x, y \in V$ do
    add a min-key triple from $P(x,y)$ to $H_f$
while $H_f \neq \emptyset$ do
    extract in $S'$ all triples with min-key $[wt, x, y]$ from $H_f$; $S \leftarrow \emptyset$
    if $S'$ is the first extracted set from $H_f$ for $x, y$ then
        if $P^*(x,y)$ increased min-weight after cleanup • then
            for each $\gamma' \in P(x,y)$ with min-key $[wt, 0]$ • do
                let $\gamma' = ((xa', b'y), wt, count')$
                {Next step check if $\gamma'$ is completely missing from $P^*$}
                if $\gamma'$ is not in $P^*(x,y)$ then
                    add $\gamma'$ in $P^*(x,y)$ and $S$; add $(x, wt)$ to $L^*(a', y)$ and $(y, wt)$ to $R^*(x, b')$
                    {Next step check if $\gamma'$ is in $P^*$ with a different count}
                else if $\gamma'$ is in $P(x,y)$ and $P^*(x,y)$ with different counts • then
                    replace the count of $\gamma'$ in $P^*(x,y)$ with $count'$ and add $\gamma'$ to $S$ •
                set $\beta$ for $\gamma' \in P(x,y)$ to 1 •
        else
            for each $\gamma' \in S'$ containing a path through $v$ do
                let $\gamma' = ((xa', b'y), wt, count')$
                add $\gamma'$ with paths$(\gamma', v)$ in $P^*(x,y)$ and $S$; add $(x, wt)$ to $L^*(a', y)$ and $(y, wt)$
                to $R^*(x, b')$
                set $\beta$ for $\gamma' \in P(x,y)$ to 1 •
        for every $b$ such that $(x\times, by) \in S$ do
            let $fcount' = \sum_i ct_i$ such that $((xa_i, by), wt, ct_i) \in S$
            for every $(x', wt')$ in $L^*(x, b)$ such that $((x'x, by), wt')$ is an LHT do
                if $((x'x, by), wt') \notin$ Marked-Tuples then
                    $wt' \leftarrow wt + \mathbf{w}(x', x)$; $\gamma' \leftarrow ((x'x, by), wt', fcount')$
                    set update-num$(\gamma')$; paths$(\gamma', v) \leftarrow \sum_{\gamma=(x\times, by)}$ paths$(\gamma, v)$; add $\gamma'$ to $H_f$
                    if a triple for $((x'x, by), wt')$ exists in $P(x', y)$ then
                        add $\gamma'$ with paths$(\gamma', v)$ in $P(x', y)$
                    else
                        add $\gamma'$ to $P(x', y)$; add $(x', wt')$ to $L(x, by)$ and $(y, wt')$ to $R(x'x, b)$
                    set $\beta$ for $\gamma' \in P(x', y)$ to 0 •
                    add $((x'x, by), wt')$ to Marked-Tuples
        perform symmetric steps 24 – 35 for right extensions
```

*not pass through $v$. Finally, the sets $L, L^*, R, R^*$ are correctly maintained.*

*Proof.* The lemma is established with the following loop invariant.

**Loop Invariant:** At the start of each iteration of the while loop in Step 3 of

Algorithm 19 the following properties hold about the tuple-system and $H_c$. Assume that the first triple to be extracted from $H_c$ and processed has min-key $= [wt, x, y]$.

1. Any LHP going through the update node $v$, which is contained in a triple $\gamma$ already processed, is removed from the tuple-system. For any $a, b \in V$, if $G$ contains $c_{ab}$ LHPs of weight $wt$ of the form $(xa, by)$ passing through $v$, then $H_c$ contains a triple $\gamma = ((xa, by), wt, c_{ab})$ with key $[wt, x, y]$ already processed: the $c_{ab}$ LHPs through $v$ are cleaned from the system.

2. For each triple $\gamma = ((xa, by), wt, count)$ already processed by the algorithm, $\gamma$ represents the exact set of LHPs of the form $x \to a \leadsto b \to y$ of weight $wt$ which avoid $v$. Moreover if $count = 0$ the triple $\gamma$ is correctly removed from all the data structures in the tuple-system associated to it. Finally, $\gamma$ has been placed in $H_c$ for future extensions.

   Let $[\hat{wt}, \hat{x}, \hat{y}]$ be the last key extracted from $H_c$ and processed before $[wt, x, y]$. For any key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, let $G$ contain $c > 0$ number of LHPs of weight $wt_1$ of the form $(x_1\times, b_1y_1)$. Further, let $c_v$ (resp. $c_{\overline{v}}$) denote the number of such LHPs that pass through $v$ (resp. do not pass through $v$). Here $c_v + c_{\overline{v}} = c$. For every extension $(x', wt') \in L(x_1, b_1y_1)$, let let $wt' = wt_1 + \mathbf{w}(x', x_1)$ be the weight of the extended triple $(x'x_1, b_1y_1)$. Then, (the following assertions are similar for $(y', wt') \in R(x_1a_1, y_1)$)

   (a) if $c > c_v$ there is a triple in $P(x', y_1)$ of the form $(x'x_1, b_1y_1)$ and weight $wt'$ representing $c - c_v$ LHPs. If $c = c_v$ there is no such triple in $P(x', y_1)$.

   (b) If a triple of the form $(x'x_1, b_1y_1)$ and weight $wt'$ is present as an HT in $P^*(x', y_1)$, then it represents the exact same number of LHPs $c - c_v$ of the corresponding triple in $P(x', y_1)$. This is exactly the number of HPs of the form $(x'x_1, b_1y_1)$ and weight $wt'$ in $G - \{v\}$.

(c) $(x', wt') \in L(x_1, b_1y_1)$, $(y_1, wt') \in R(x'x_1, b_1)$ and $(x'x_1, b_1y_1) \in$ Marked-Tuples iff $c_{\bar{v}} > 0$. If the triple $(x'x_1, b_1y_1)$ is an HT, a similar statement holds for $(x', wt') \in L^*(x_1, y_1)$ iff there is a triple of weight $wt_1$ in $P^*(x_1, y_1)$, and $(y_1, wt') \in R^*(x', b_1)$ iff there is a triple of weight $wt' - \mathbf{w}(b_1, y_1)$ in $P^*(x', b_1)$.

(d) A triple corresponding to $(x'x_1, b_1y_1)$ with weight $wt'$ and counts $c_v$ is in $H_c$. A similar assertion holds for $y' \in R(x_1a_1, y_1)$.

3. Any triple $\gamma$ in $H_c$, which is longer than the last triple processed, is also in Marked-Tuples if and only if it contains at least one path not passing through $v$. For any key $[wt_2, x_2, y_2] \geq [wt, x, y]$, let $G$ contain $c > 0$ LHPs of weight $wt_2$ of the form $(x_2a_2, b_2y_2)$. Further, let $c_v$ (resp. $c_{\bar{v}}$) denote the number of such LHPs that pass through $v$ (resp. do not pass through $v$). Here $c_v + c_{\bar{v}} = c$. Then the tuple $(x_2a_2, b_2y_2) \in$ Marked-Tuples, iff $c_{\bar{v}} > 0$ and a triple for $(x_2a_2, b_2y_2)$ is present in $H_c$

**Initialization:** We start by showing that the invariants hold before the first loop iteration. The min-key triple in $H_c$ has key $[0, v, v]$. Invariant assertion 1 holds since we inserted into $H_c$ the trivial triple of weight 0 corresponding to the vertex $v$ and that is the only triple of such key. Moreover, since we do not represent trivial paths containing the single vertex, no counts need to be decremented. Since we assume positive edge weights, there are no LHPs in $G$ of weight less than zero. Thus all the points of invariant assertion 2 hold trivially. Invariant assertion 3 holds since $H_c$ does not contain any triple of weight $> 0$ and we initialized Marked-Tuples to empty.

**Maintenance:** Assume that the invariants are true before an iteration $k$ of the loop. We prove that the invariant assertions remains true before the next iteration $k+1$. Let the min-key triple at the beginning of the $k$-th iteration be $[wt_k, x_k, y_k]$. By invariant assertion 1, we know that for any $a_i, b_j$, if there exists a triple $\gamma$ of the form $(x_k a_i, b_j y_k)$ of weight $wt_k$ representing *count* paths going through $v$, then it is present in $H_c$. Now consider the set of triples with key $[wt_k, x_k, y_k]$ which we extract in the set $S$ (Step 4, Algorithm 19). We consider left-extensions of triples in $S$; symmetric arguments apply for right-extensions. Consider for a particular $b$, the set $S_b \subseteq S$ of triples of the form $(x_k \times, by_k)$ and let $fcount'$ denote the sum of the counts of the paths represented by triples in $S_b$. Let $(x', wt') \in L(x_k, by_k)$ be a left extension; our goal is to generate the triple $(x'x_k, by_k)$ with count $fcount'$ and weight $wt' = wt_k + \mathbf{w}(x', x_k)$. However, we generate such triple only if it has not been generated by a right-extension of another set of paths. We observe that the paths of the form $(x'x_k, by_k)$ can be generated by right extending to $y_k$ the set of triples of the form $(x'x_k, \times b)$. Without loss of generality assume that the triples of the form $(x'x_k, \times b)$ have a key which is greater than the key $[wt_k, x_k, y_k]$. Thus, at the beginning of the $k$-th iteration, by invariant assertion 3, we know that $(x'x_k, by_k) \notin$ Marked-Tuples. Steps 8–9, Algorithm 19 create a triple of the form $(x'x_k, by_k)$ of weight $wt'$. The generated triple can be an LST or a TLHT in $P$. In both cases the condition at step 9, Algorithm 19 holds and we remove $\gamma'$ by decrementing $fcount'$ many paths from the appropriate triple in $P(x', y_k)$. Moreover, if the generated triple is also contained in $P^*(x', y_k)$, we check if it is an ST or a THT using step 15, Algorithm 19. In the case of an ST we normally decrement $fcount'$ paths from the appropriate triple in $P^*(x', y_k)$, otherwise we decrement the THT from $P^*(x', y_k)$ in step 16, Algorithm 19 removing only the paths going through the updated node $v$ that are contained in the THT, by using the new data structure $CH$. This establishes invariant assertions $2a$ and $2b$. In addition, if there are no LSPs in $G$ of the form $(x'x_k, by_k)$ which do not pass through $v$, we delete $(x', wt')$ from $L(x_k, by_k)$ and delete $(y_k, wt')$ from $R(x'x_k, b)$ (Step 13, Algorithm 19). On the other hand, if there exist LSPs in $G$ of the form $(x'x_k, by_k)$, then $x'$ (resp. $y_k$) continues to exist in $L(x_k, by_k)$ (resp. in $R(x'x_k, b)$). Further, we add the tuple $(x'x_k, by_k)$ to Marked-

Tuples and observe that the corresponding triple is already present in $H_c$ (Step 11, Algorithm 19). Similarly, if the generated triple $(x'x_k, by_k)$ is an HT, then we check if $P^*(x_k, y_k)$ does not contain any triple of weight $wt_k$, and similarly $P^*(x', b)$ does not contain any triple of weight $wt' - \mathbf{w}(b, y_k)$, in order to delete $(x', wt')$ from $L^*(x_k, y_k)$ and $(y_k, wt')$ from $R^*(x', b)$. By the loop invariant, invariant assertions $2c$ and $2d$ were true for every key $< [wt_k, x_k, y_k]$ and by the above steps we ensure that these invariant assertions hold for every key $= [wt_k, x_k, y_k]$. Thus, invariant assertions $2c$ and $2d$ are true at the beginning of the $(k + 1)$-th iteration. Note that any triple that is generated by a left extension (or symmetrically right extension) is inserted into $H_c$ as well as into Marked-Tuples. This establishes invariant assertion 3 at the beginning of the $(k + 1)$-th iteration.

Finally, to see that invariant assertion 1 holds at the beginning of the $(k+1)$-th iteration, let the min-key at the $(k+1)$-th iteration be $[wt_{k+1}, x_{k+1}, y_{k+1}]$. Observe that triples with weight $wt_{k+1}$ starting with $x_{k+1}$ and ending in $y_{k+1}$ can be created either by left extending or right extending the triples of smaller weight. And since for each of iteration $\leq k$, invariant assertion 2 holds for any extension, we conclude that invariant assertion 1 holds at the beginning of the $(k + 1)$-th iteration. This finishes our maintenance step.

**Termination:** The condition to exit the loop is $H_c = \emptyset$. Because invariant assertion 1 maintains in $H_c$ all the triples already processed, then $H_c = \emptyset$ implies that there are no other triples to extend in the graph $G$ that contain the updated node $v$. Moreover, because of invariant assertion 1, every triple containing the node $v$ inserted into $H_c = \emptyset$, has been correctly decremented from the tuple-system. Finally, for invariant assertion $2c$, the stacks $L, L^*, R, R^*$ are correctly maintained. This completes the proof. □

For FULLY-FIXUP , we first show that Algorithm 20 computes the correct distances for all the SPs in the updated graph $G'$ (Lemma 23). Moreover, we process all the *new* SPs in $G'$ (Lemma 24). Finally, we show that data structures and counts are correctly maintained after the algorithm (Lemma 25). Here we use the notion of a *fresh* LHT for a triple that represents at least one path that is in $P$ but not in $P^*$. We will consider fresh triples in Lemma 24 and Observation 13.

**Invariant 11.** *During the execution of Algorithm 20, for any pair* $(x, y)$*, consider the first extraction from* $H_f$ *of a set of triples from* $x$ *to* $y$*, and let their weight be* $wt$*. Then* $wt$ *is the shortest path distance from* $x$ *to* $y$ *in the updated graph* $G'$*.*

**Lemma 23.** *Algorithm 20 maintains Invariant 11.*

*Proof.* Suppose for a contradiction that the invariant is violated at some extraction. Consider the earliest event in which the first set of triples $S'$ of weight $\hat{wt}$, extracted for some pair $(x, y)$, does not contain STs in $G'$. Let $\gamma = ((xa, by), wt, count)$ be a triple in $G'$ that represents at least one shortest path from $x$ to $y$ in $G'$, with $wt < \hat{wt}$. The triple cannot be in $P(x, y)$ at the beginning of fixup otherwise it (or another triple with same weight $wt$) would have been inserted in $H_f$ during step 5 of Algorithm 20. Moreover, $\gamma$ cannot be in $H_f$ otherwise it would have been extracted before any triple of weight $\hat{wt}$ in $S'$; hence $\gamma$ must be a *new* LST generated by the algorithm. Since all the edges incident to $v$ are added to $H_f$ during step 3 of Algorithm 20, then $\gamma$ must represent SPs of at least two edges. We define $left(\gamma)$ as the set of LSTs of the form $((xa, c_ib), wt - \mathbf{w}(b, y), count_{c_i})$ that represent all the LSPs in the left tuple $((xa, b), wt - \mathbf{w}(b, y))$; similarly we define $right(\gamma)$ as the set of LSTs of the form $((ad_j, by), wt - \mathbf{w}(x, a), count_{d_j})$ that represent all the LSPs in the right tuple $((a, by), wt - \mathbf{w}(x, a))$.

Observe that since $\gamma$ is an ST, all the LSTs in $left(\gamma)$ and $right(\gamma)$ are also STs. A triple in $left(\gamma)$ and a triple in $right(\gamma)$ cannot be present in $P^*$ together at the beginning of fixup. In fact, if at least one triple from both sets is present in $P^*$ at the beginning of fixup, then the last one inserted during the fixup triggered by the previous update, would have generated an LST of the form

$((xa, by), wt)$ automatically inserted and thus present in $P$ at the beginning of fixup (a contradiction). Thus either there is no triple in $left(\gamma)$ in $P^*$, or there is no triple in $right(\gamma)$ in $P^*$.

Assume w.l.o.g. that no triple in $right(\gamma)$ is in $P^*$. Since edge weights are positive, $wt - \mathbf{w}(x, a) < wt < \hat{wt}$, and because all the extractions before $\gamma$ were correct, then the triples in $right(\gamma)$ were correctly extracted from $H_f$ and placed in $P^*$ before the wrong extractions in $S'$. If at least one triple in $left(\gamma)$ is in $P^*$ then the fixup would generate the tuple $((xa, by), wt)$ and place it in $P$ and $H_f$ (Steps 24–35, Algorithm 20). Otherwise, since $wt - \mathbf{w}(b, y) < wt < \hat{wt}$, the triples in $left(\gamma)$ were discovered by the algorithm before the wrong extractions in $S'$. Moreover the algorithm would generate the tuple $((xa, by), wt)$ (as right extensions) and place it in $P$ and $H_f$ (because at least one triple in $right(\gamma)$ is already in $P^*$). Thus, in both cases, a tuple $((xa, by), wt)$ should have been extracted from $H_f$ before any triple in $S'$. A contradiction. $\square$

**Invariant 12.** *The set $S$ of triples constructed in Steps 9–23 of Algorithm 20 represents all the* new *shortest paths from $x$ to $y$.*

**Lemma 24.** *Algorithm 20 maintains Invariant 12.*

*Proof.* Any new SP from $x$ to $y$ is of the following three types:

1. a single edge containing the vertex $v$ (such a path is added to $P(x, y)$ and $H_f$ in Step 3)

2. a path generated via left/right extension of some shortest path previously extracted from $H_f$ during the execution of Algorithm 20 (this generated path is added to $P(x, y)$ and $H_f$ in Step 29 and an analogous step in right-extend).

3. a path that was an LSP but not an SP before the update and is an SP after the update.

In type (1) and (2) above any new SP from $x$ to $y$ which is added to $H_f$ is also added to $P^*(x, y)$. However, amongst the several triples representing paths of the type (3) listed above, only one candidate triple will be present in $H_f$. Thus we conclude that, for a given $x, y$, when we extract from $H_f$ a type (3) triple of weight $wt$, $P(x, y)$ could contain a superset of triples with the same weight $wt$ that are not present in $H_f$. We now consider the two cases the algorithm deals with.

- $P^*(x, y)$ increased its min-weight, when the first set of triples for $x, y$ is extracted from $H_f$. This is the only case where we could restore historical triples, or process fresh triples from scratch because they are not yet in $P^*$ or they are present in $P^*$ with a lower count than the corresponding triple in $P$ (note that this condition is triggered only by increase-only updates). Note that we process all the min-weight triples in $P(x, y)$, but before we really add a triple in $S$ for further extensions, we check if it is present in $P^*$ with a lower count (Step 16, Algorithm 20), or it is not present in $P^*$ (Step 13, Algorithm 20). By the above argument, we consider all the new STs from $x$ to $y$ present in $P(x, y)$. Therefore it suffices to argue that all of them contains *new* shortest paths to be processed. Suppose for contradiction that some triple $\gamma$ does not contains *new* shortest paths. Thus, $\gamma$ was a ST before the update and already in $P^*$ with at least one path not going through $v$. However, since cleanup only removes paths that contain $v$, the triple $\gamma$ remained in $P^*(x, y)$ after the FULLY-CLEANUP phase. This contradicts the fact that $P^*(x, y)$ increased its min-weight.

- $P^*(x, y)$ didn't change its min-weight when the first set of triples for $x, y$ is extracted from $H_f$. Let the weight of triples in $P^*(x, y)$ be $wt$. This implies that the shortest path distance from $x$ to $y$ before and after the update is $wt$. Both in the case of decrease-only and increase-only updates, all the new paths that we need to consider from $x$ to $y$ are going through the updated node $v$.

By construction of the Algorithm 20, every triple containing the updated node $v$ is always placed into $H_f$. Thus it suffices to consider only triples in $H_f$.

$\square$

**Observation 13.** *During the execution of Algorithm 20, consider a THT $\tau$ that becomes shortest. If $\tau$'s corresponding triple in $P$ is not fresh, then it is simply restored (not processed); otherwise $\tau$'s count is replaced with the updated count from $P$ and it is extended anew.*

*Proof.* When we restore an existing HT $\tau$ from $P^*$, we always check if its corresponding triple in $P$ contains more paths (Step 16, Algorithm 20) or the counts match. In the first case $\tau$ in $P^*$ is carrying an obsolete number of SPs and is therefore replaced with the correct count in $P$ and extended anew (Step 17, Algorithm 20). Otherwise it is still representing the correct number of SPs to be restored and it is not processed. $\square$

**Lemma 25.** *After the execution of Algorithm 20 (*FULLY-FIXUP*), for any $(x, y) \in V$, the STs in $P^*(x, y)$ (LSTs in $P(x, y)$) represent all the SPs (LSPs) from $x$ to $y$ in the updated graph. Also, the sets $L, L^*, R, R^*$ are correctly maintained.*

*Proof.* **Loop Invariant:** At the start of each iteration of the while loop in Step 6 of Algorithm 20, assume that the first triple in $H_f$ to be extracted and processed has min-key $= [wt, x, y]$. Then the following properties hold about the tuple-system and $H_f$.

1. If $G$ contains $c_{ab}$ SPs of form $(xa, by)$ and weight $wt$, then $H_f$ contains a triple of form $(xa, by)$ and weight $wt$ to be extracted and processed. For any $a, b \in V$, if $G'$ contains $c_{ab}$ SPs of form $(xa, by)$ and weight $wt$, then $H_f$ contains a triple of form $(xa, by)$ and weight $wt$ to be extracted and processed. Further, a triple $\gamma = ((xa, by), wt, c_{ab})$ is present in $P(x, y)$.

2. Let $[\hat{wt}, \hat{x}, \hat{y}]$ be the last key extracted from $H_f$ and processed before $[wt, x, y]$. Every SP in $G$, with an associated key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, is represented by a triple $\gamma$ in the tuple-system having the correct count. Moreover, the tuple-system contains all the valid extensions derived by $\gamma$, and any extension of $\gamma$ itself is present in $H_f$ as a candidate triple. For any key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, let $G'$ contain $c > 0$ number of LHPs of weight $wt_1$ of the form $(x_1a_1, b_1y_1)$. Further, let $c_{new}$ (resp. $c_{old}$) denote the number of these LHPs that are *new* (resp. not *new*). Here $c_{new} + c_{old} = c$. If $c_{new} > 0$ then,

   (a) there is an LHT in $P(x_1, y_1)$ of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ that represents $c$ LHPs.

   (b) If a triple of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ is present as an HT in $P^*$, then it represents the exact same count of $c$ HPs of its corresponding triple in $P$. This is exactly the number of HPs of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ in $G'$.

   (c) $(x_1, wt_1) \in L(a_1, b_1y_1)$, $(y_1, wt_1) \in R(x_1a_1, b_1)$, and if the triple of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ is also shortest then $(x_1, wt_1) \in L^*(a_1, y_1)$, $(y_1, wt_1) \in R^*(x_1, b_1)$. Further, $(x_1a_1, b_1y_1) \in$ Marked-Tuples iff $c_{old} > 0$.

   (d) If $c_{new} > 0$, for every $(x', wt') \in L(x_1, b_1y_1)$, an LHT corresponding to $(x'x_1, b_1y_1)$ with weight $wt' = wt_1 + \mathbf{w}(x', x_1) \geq wt$ and counts equal to the sum of new paths represented by its constituents, is in $H_f$ and $P$. A similar assertion holds for $(y', wt') \in R(x_1a_1, y_1)$.

3. Any triple $\gamma$ in $H_f$, which is longer than the last triple processed, is also in Marked-Tuples if and only if it contains at least one old LHP (generated by a previous update) and a new LHP added to $H_f$ during a previous step of the current update. For any key $[wt_2, x_2, y_2] \geq [wt, x, y]$, let $G'$ contain $c > 0$ number of LHPs of weight $wt_2$ of the form $(x_2a_2, b_2y_2)$. Further, let $c_{new}$

(resp. $c_{old}$) denote the number of such LHPs that are *new* (resp. not *new*). Here $c_{new} + c_{old} = c$. Then the tuple $(x_2a_2, b_2y_2) \in$ Marked-Tuples, iff $c_{old} > 0$ and $c_{new}$ paths have been added to $H_f$ by some earlier iteration of the while loop.

Initialization and Maintenance for the 3 invariant assertions are similar to the proof of Lemma 22.

**Termination:** The condition to exit the loop is $H_f = \emptyset$. Because invariant assertion 1 maintains in $H_f$ the first triple to be extracted and processed, then $H_f = \emptyset$ implies that there are no triples, formed by a valid left or right extensions, that contain *new* SPs or LSPs, that need to be added or restored in the graph $G$. Moreover, because of invariant assertions $2a$ and $2b$, every triple containing the node $v$, extracted and processed before $H_f = \emptyset$, has been added or restored with its correct count in the tuple-system. Finally, for invariant assertion $2c$, the stacks $L, L^*, R, R^*$ are correctly maintained. This completes the proof of the loop invariant.

By Lemma 24, all the new SPs in $G'$ are placed in $H_f$ and processed by the algorithm and hence are in $P^*$ after the execution of Algorithm 20. Moreover, for a pair $(x, y)$, the check in Step 9 of Algorithm 20 would fail if the distance from $x$ to $y$ doesn't change after the update. Thus the old SPs from $x$ to $y$ will remain in $P^*(x, y)$. Hence, after Algorithm 20 is executed, every SP in $G'$ is in its corresponding $P^*$.

Since every LST of the form $(xa, by)$ in $G'$ is formed by a left extension of a set of STs of the form $(a\times, by)$ (Steps 24–35, Algorithm 20), or a right extension of a set of the form $(xa, \times b)$ (analogous steps for right extensions), and all the STs are correctly maintained by the algorithm, then all the LSTs are correctly maintained at the end of the fixup algorithm. This completes the proof of the Lemma. □

### 5.4.2 Analysis and Properties of FULLY-UPDATE

As discussed in the previous sections, both FULLY-CLEANUP and FULLY-FIXUP are similar to the corresponding pseudocode cleanup and fixup in Chapter 3, but they require some important changes to ensure correctness and efficiency. In this section, we highlight several new components that will be relevant for proving the properties of FULLY-UPDATE presented in this section.

**New Components in FULLY-CLEANUP relative to `NPRdec`**

C.1 FULLY-CLEANUP decrements THT using the new data structure $CH$. More specifically, a THT $\gamma' = ((xa, by), wt, count')$ is decremented only by *count* paths, where *count* is the number of paths, going through the updated node $v$, in $CH(\gamma)$ and generated during the updates at times $t \leq$ update-num($\gamma'$). The structure $CH$ is initialized with update-num($v$) associated with the value 1 representing the trivial tuple $(vv, vv)$.

**New Components in FULLY-FIXUP relative to `NPRdec`**

F.1 A triple $\gamma$ is now inserted in $P$ with a key $[wt, \beta]$, instead of just $wt$. Here $\beta$ is a control bit that is set during the fixup phase and indicates if $\gamma$ is present in $P^*$ with the correct count ($\beta = 1$), or if $\gamma$ has the correct count only in $P$ ($\beta = 0$). In the latter case $\gamma$ could be also present in $P^*$ but with a wrong count.

F.2 Before processing a triple $\gamma$, we check the bit $\beta$ associated with it. In fact, since $P$ is a priority queue, we will process only the triples in $P$ with min-key $[wt, 0]$, thus avoiding the triples that are already in $P^*$ with a correct count.

F.3 Reverting an HT from, say $x$ to $y$, back to an ST during an update happens only if the shortest path distance from $x$ to $y$ increases.

We now establish some basic properties of algorithm FULLY-UPDATE based on the high-level description we have given above. We start with a general bound on the running time.

**Lemma 26.** *Consider a sequence of $r$ calls to* FULLY-UPDATE *on a graph with $n$ vertices. Let $C$ be the maximum number of tuples in the tuple-system that can contain a path through a given vertex, and let $D$ be the maximum number of tuples that can be in the tuple-system at any time. Then* FULLY-UPDATE *executes the $r$ updates in $O((r \cdot (n^2 + C) + D) \cdot \log n)$ time.*

*Proof.* We bound the cost of FULLY-UPDATE by classifying each triple $\gamma$ as one of the following disjoint types:

- **Type-0 (contains-v):** $\gamma$ represents at least one path containing vertex $v$.
- **Type-1 (new-LHT):** $\gamma$ was not an LHT before the update but is an LHT after the update, and no path in $\gamma$ contains $v$.
- **Type-2 (new-HT-old-LHT):** $\gamma$ is an HT after the update, and $\gamma$ was an LHT but not an HT before the update, and no path in $\gamma$ contains $v$.
- **Type-3 (renew-ST):** $\gamma$ was a THT before the update and it is restored as a ST after the update, and no path in $\gamma$ contains $v$.
- **Type-4 (new-LHT-old-LHT):** $\gamma$ was an LHT before the update and continues to be an LHT after the update, and no path in $\gamma$ contains $v$.

The number of Type-0 triples, processed by FULLY-FIXUP is at most $C$. The number of Type-1 triples, processed by FULLY-FIXUP is addressed by amortizing over the entire update sequence as described in the paragraph below. For a Type-2 triple processed by FULLY-FIXUP, we observe that after such a triple becomes an HT, it is not removed from $P^*$ unless a real or dummy update is triggered on a vertex that lies in it. But in such an update this would be counted as a Type-0 triple. Further, each such Type-2 triple is examined only a constant number of times FULLY-FIXUP, because after they are inserted into $P^*$ the bit $\beta$ associated changes to 1 and they will not be processed again by fixup, unless the number of paths they represent is changed (fact[F.2]). Hence we charge each access to a Type-2 triple to the step in which it was created as a Type-1 triple. For Type-3 triples, we distinguish two cases: if $\gamma$ didn't change its count in $P$ after it became a THT then its flag is $\beta = 1$ and it is present in $P^*$ with the correct count (fact[F.1]). Thus the fixup algorithm will not process it. If $\gamma$ changed its count in $P$ while it was a THT then its flag is $\beta = 0$ and we can charge the processing of $\gamma$ (if extracted from $H_f$) to the sub-triple $\gamma'$ generated from the updated node that increased the count of $\gamma$; in other words there was an ST $\gamma'$, created in a previous update, whose extensions added to the count of $\gamma$. Observe that, triples in $P^*$ that are not placed initially in $H_f$ and have $\beta = 1$ in $P$ (no additional path was added to that triple) are not examined in any step of fixup (fact[F.2]), so no additional Type-3 triples are examined. For Type-4, we note that for any $x, y$ we add exactly one candidate min-key triple from $P(x, y)$ to $H_f$, hence initially there are at most $n^2$ such triples in $H_f$, any of which could be Type-4. Moreover, we never process an old LHT which is not a new HT so no additional Type-4 triples are examined during fixup. Thus the number of triples examined by a call to fixup is $C$ plus $X$, where $X$ is the number of *new* triples fixup adds to the tuple system. (This includes an $O(1)$ credit placed on each new LHT for a possible later conversion to an HT.)

Let $r$ be the number of updates in the update sequence. Since triples are removed only in cleanup, at most $O(r \cdot C)$ triples are removed (or decremented) by the cleanups. There can be at most $D$ triples remaining at the end of the sequence, hence the total number of new triples added by all fixups in the update sequence is $O(r \cdot C + D)$. Since the time taken to access a triple is $O(\log n)$ due to the data structure operations, and we examine at least $n^2$ triples at each round, the total time spent by fixup over $r$ updates is $O((r \cdot (n^2 + C) + D) \cdot \log n)$. □

In Section 5.4.3, we will use the algorithm FULLY-UPDATE within algorithm FULLY-DYNAMIC, that performs a special sequence of 'dummy' updates, to obtain a fully dynamic APASP algorithm with an $O(\nu^{*2} \log^3 n)$ amortized cost per update; we obtain this amortized bound by establishing suitable upper bounds on the parameters $C$ and $D$ in Lemma 26 when algorithm FULLY-DYNAMIC is used. We conclude this section with some lemmas that will be used in the analysis of the amortized time bound of Algorithm 21.

**Lemma 27.** *At each step $t$, the tuple-system for Algorithm* FULLY-UPDATE *maintains a subset of HTs and LHTs that includes all STs and LSTs for step $t$. Further, for every LHT triple $((xa, by), wt, count)$ in step $t$, there are HTs $(a*, by)$ and $(xa, *b)$ with weights $wt - w(x, a)$ and $wt - w(b, y)$ respectively, in that step.*

*Proof.* For the first part see correctness in Section 5.4.1 (Lemmas 22 and 25). For the second part, if a triple $\gamma = ((xa, by), wt, count)$ is present in step $t$, then $\gamma$ was generated during FULLY-FIXUP of some step $t' \leq t$. By the construction of our algorithm, at the end of step $t'$, the tuple-system contains at least one HT of the form $(a*, by)$ and one of the form $(xa, *b)$ with weights $wt - w(x, a)$ and $wt - w(b, y)$ respectively. W.l.o.g. suppose that the set $S$ of all the HTs of the form $(a*, by)$ and weight $wt - w(x, a)$ are removed during FULLY-CLEANUP at some step $t'' \leq t$, then since these HTs are the right constituents of $\gamma$, when $S$ is left extended to $x$ in FULLY-CLEANUP then exactly *count* paths will be removed from $\gamma$ making the triple disappear from the tuple-system. Thus at step $t$, at least one HT of the form $(a*, by)$ with weight $wt - w(x, a)$ must be in $P^*$. Similarly, there must be an HT of the form $(xa, *b)$ with weight $wt - w(b, y)$ in $P^*$ at step $t$. □

**Lemma 28.** *If* FULLY-UPDATE *is called on vertex $v$ at step $t$, then at the end of step $t$ any TLHT $(xa, by)$ in the tuple-system that contains a path through $v$, has the vertex $v$ as one of the endpoints $x$ or $y$.*

*Proof.* By Lemma 27, any LHT in the tuple-system is formed by combining two HTs that are in the tuple-system. Now consider the TLHT $(xa, by)$ that contains $v$. Since by definition of TLHT, $(xa, by)$ is not an LST in the current graph, assume w.l.o.g. that $(xa, b)$ is the subtuple that is not an ST when an end edge is deleted from $(xa, by)$. But this is not possible since any tuple HT $(xa, *b)$ that contains $v$ must be an ST. The lemma follows. □

**Lemma 29.** *Let $G$ be a graph after a sequence of calls to* FULLY-UPDATE, *and suppose every HT in the tuple-system is an ST in one of $z$ different graphs $H_1, \cdots, H_z$, and every LHT is formed from these HTs. If $n$ and $m$ bound the number of vertices and edges, respectively, in any of these graphs, and if $\nu^*$ bounds the maximum number of edges that lie on shortest paths through any given vertex in any of the these graphs, then:*

1. *The number of LHTs in $G$'s tuple-system is at most $O(z \cdot m \cdot \nu^*)$.*
2. *The number of LHTs that contain the newly updated vertex $v$ in $G$ is $O(z \cdot \nu^{*2})$.*
3. *Let $u$ be any vertex in $G$, and suppose that the HTs that contain $u$ lie in $z' \leq z$ of the graphs $H_1, \cdots, H_z$. Then, the number of LHTs that contain the vertex $u$ is $O((z + z'^2) \cdot \nu^{*2})$.*

*Proof.* For part 1, we bound the number of LHTs $(xa, by)$ (across all weights) that can exist in $G$. The edge $(x, a)$ can be chosen in $m$ ways, and once we fix $(x, a)$, the $r$-tuple $(a, by)$ must be an ST in one of the $H_j$. Since $(b, y)$ must lie on a shortest path through $a$ in the graph $H_i$ that contains the $r$-tuple $(a, by)$ of that weight, the number of different choices for $(b, y)$ that will then uniquely determine the tuple

$(xa, by)$, together with its weight, is $z \cdot \nu^*$. Hence the number of LHTs in $G$ 's tuple-system is $O(z \cdot m \cdot \nu^*)$.

For part 2, the number of LHTs that contain $v$ as an internal vertex is simply the number of LSTs in the current graph by Lemma 28, and using Lemma 7 (Chapter 3), this is $O(\nu^{*2})$. We now bound the number of LHTs $(va, by)$. There are $n-1$ choices for the edge $(v, a)$ and $z \cdot \nu^*$ choices for the $r$-tuple $(a, by)$, hence the total number of such tuples is $O(z \cdot n \cdot \nu^*)$. The same bound holds for LHTs of the form $(xa, bv)$. Since $\nu^* = \Omega(n)$, the result in part 2 follows.

For part 3, we observe that each LHT that contains $u$ as an internal vertex must be composed of two HTs, each of which is an ST in one of the $z'$ graphs that contain $v$. Thus, there are $O(z'^2 \cdot \nu^{*2})$ such tuples. For an LHT, say $\tau = (ua, by)$, that contains $u$ as an end vertex, the analysis remains the same as above in part 2: there are $n-1$ choices for the edge $(u, a)$ and $z \cdot \nu^*$ choices for the $r$-tuple $(a, by)$, hence the total number of such tuples is $O(z \cdot n \cdot \nu^*)$. This gives the desired result. □

### 5.4.3 The Overall Algorithm FULLY-DYNAMIC

Algorithm FULLY-UPDATE in Section 5.3.1 is a correct fully dynamic algorithm for APASP, but it is not a very efficient algorithm, since $C$ and $D$ in Lemma 26 could be very large. We now present our overall fully dynamic algorithm for APASP. As in [DI04, Tho04] we build up the tuple-system for the initial $n$-node graph $G = (V, E)$ with $n$ inserts starting with the empty graph (and hence $n$ decrease-only updates), and we then perform the first $n$ updates in the given update sequence $\Sigma$. After these $2n$ updates, we reset all data structures and start afresh.

Consider a graph $G = (V, E)$ with weight function $\mathbf{w}$ in which a decrease-only or increase-only update is applied to a vertex $u$. Let $\mathbf{w}'$ be the weight function after the update, hence the only changes to the edge weights occur on edges incident to $u$. Algorithm 21 gives the overall fully dynamic algorithm for the $t$-th update to a vertex $v$ with the new weight function $\mathbf{w}'$. This algorithm applies FULLY-UPDATE to vertex $v$ with the new weight function, thus it will be correct if we executed only the first step, but not necessarily efficient. To obtain an efficient algorithm we execute 'dummy updates' on a sequence $\mathcal{N}$ of the most recently updated vertices as specified in Steps 2-5. The length of this sequence of vertices is determined by the position $k$ of the least significant bit set to 1 in the bit representation $B = b_{r-1} \cdots b_0$ of $t$. We denote $k$ by *set-bit*$(t)$.

**Algorithm 21** FULLY-DYNAMIC$(G, v, \mathbf{w}', t)$

```
FULLY-UPDATE$(v, \mathbf{w}')$
$k \leftarrow$ set-bit$(t)$ (i.e., if the bit representation of $t$ is $b_{r-1} \cdots b_0$, then $b_k$ is the least
significant bit with value 1)
$\mathcal{N} \leftarrow$ set of vertices updated at steps $t-1, \cdots, t-(2^k-1)$
for each $u \in \mathcal{N}$ in decreasing order of update time do
    FULLY-UPDATE$(u, \mathbf{w}')$     (dummy updates)
```

**Properties of $\mathcal{N}$.** Consider the current update step $t$, with its bit representation $B = b_{r-1} \cdots b_0$ and with *set-bit*$(t) = k$. We say that index $i$ is a *time-stamp* for $t$ if $b_i = 1$ (so $r - 1 \geq i \geq k$), and for each such time-stamp $i$, we let $time_t(i)$ be the earlier update step $t'$ whose bit representation has zeros in positions $b_i - 1, \cdots, 0$, and which matches $B$ in positions $b_{r-1} \cdots b_i$. In other words, $time_t(i) = t'$, where $t'$ has bit representation $b_{r-1} \cdots b_{i+1} b_i 0 \cdots 0$. We do not define $time_t(i)$ if $b_i = 0$. We define *Prior-times*$(t)$ be the set of $time_t(i)$ where $i$ is a time-stamp for $t$, and we also include in *Prior-times*$(t)$ the initial time $t_0 = 0$. Note that $|\textit{Prior-times}(t)| \leq r + 1 = O(\log n)$, since $r \leq \log(2n)$.

Let $G_t$ be the graph after the $t$-th update is applied, $t \geq 1$, with the initial graph being $G_0$ (at time $t_0 = 0$). Thus, the input graph to Algorithm 21 is $G_{t-1}$, and the updated graph is $G_t$.

**Lemma 30.** *For every vertex $v$ in $G_t$, the step $t_v$ of the most recent update to $v$ is in Prior-times$(t)$.*

*Proof.* Let the bit representation of $t$ be $B = b_{r-1} \cdots b_0$, let *set-bit*$(t) = i$, and let $j_1 > j_2 > \cdots > j_s = i$ be the time-stamps for $t$. Let $t_u = time_t(b_{j_u})$, thus the bit representation of $t_u$ is the same as $B$, with all bits in positions less than $j_u$ set to zero; let $t_0 = 1$. Then, we observe that during the execution of Algorithm 21 for the $t_u$-th update, the vertex for update $t_u$ will be updated in Step 1, and the vertices updated in steps $t_{u-1} + 1, \cdots, t_u - 1$ will be updated in Steps 4-5 of Algorithm 21. Hence all vertices updated in $[t_{u-1} + 1, t_u]$ are more recently updated in step $t_u$. Thus the most recent update step for every vertex in $G_t$ is one of the $O(\log n)$ steps in *Prior-times*$(t)$. □

### 5.4.4 Analysis of Algorithm FULLY-DYNAMIC

The analysis in this section incorporates many elements from `Thorup` algorithm [Tho04]. However, these are present only in the analysis, and the only component of that rather complicated algorithm that we use is the form of the dummy update sequence in Step 3 of Algorithm 21. Our Algorithm 21 is about as simple as the `DI` algorithm when specialized to unique shortest paths since in that case it suffices to use the update algorithm in `DI` instead of the more elaborate FULLY-UPDATE we use here.

#### The Prior Deletion Graph (PDG)

Let $t' < t$ be two update steps, and let $W$ be the set of vertices that are updated in the interval of steps $[t'+1, t]$. We define the *prior deletion graph (PDG)* $\Gamma_{t',t}$ as the induced subgraph of $G_{t'}$ on the vertex set $V(G_{t'}) - W$. If $t$ is the current update step, then we simply use $\Gamma_{t'}$ instead of $\Gamma_{t',t}$.

We say that a path $p$ *is present in both* $G_{t'}$ *and* $G_t$ if no call to FULLY-UPDATE is made on any vertex in $p$ during the update steps in the interval $[t'+1, t]$.

**Lemma 31.** *Let tuple $\tau$ represent a collection of paths in $G_{t'}$.Then,*

1. *If $\tau$ is an ST in $G_{t'}$ then $\tau$ continues to be an ST in every PDG $\Gamma_{t',t}$ with $t \geq t'$ in which $\tau$ is present.*

2. *For any $\hat{t} \geq t'$, if $\tau$ is an ST in $G_{\hat{t}}$ then $\tau$ is an ST in every PDG $\Gamma_{t',t''}$, $t'' \geq \hat{t}$, in which $\tau$ is present.*

*Proof.* As observed in `NPRdec` (and in [DI04] for unique shortest paths), an ST in a graph remains an ST after a weight increase on any edge that is not on it. This establishes the first part since an increase-only updates does not affect the weight of existing STs that avoid the updated node. For the second part, we observe that $\Gamma_{t',\hat{t}}$ can be viewed as being obtained from $G_{\hat{t}}$ by deleting the vertices updated in $[t'+1, \hat{t}]$. Since $\tau$ is an ST in $G_{\hat{t}}$, it continues to be an ST in the graph $\Gamma_{t',\hat{t}}$, which can be obtained from $G_{\hat{t}}$ through a sequence of increase-only updates that do not change the weight of any edge on $\tau$. Finally since $\tau$ is an ST in $\Gamma_{t',\hat{t}}$, it must be an ST in any $\Gamma_{t',t''}$ in which it appears, for $t'' > \hat{t}$. □

**PDGs for Update $t$:** We will associate with the current update step $t$, the set of PDGs $\Gamma_{t'}$, for $t' \in$ *Prior-times*$(t)$. These PDGs are similar to the *level graphs* maintained in `Thorup` algorithm [Tho04] (our FFD algorithm in Chapter 6 generalizes this approach to multiple shortest paths), but we choose to give them a different name since we use them here only to analyze the performance of our algorithm.

**Lemma 32.** *Consider a sequence of fully dynamic updates performed using Algorithm 21. Let the current update step be $t$. Then, each HT in the tuple-system for $G_t$ is an ST in at least one of the $\Gamma_{t'}$, where $t' \in$ Prior-times$(t)$. Further $z = O(\log n)$ in Lemma 29 for $G_t$.*

*Proof.* Consider an HT $\tau = (xa, by)$ in $G_t$. Let the most recently updated vertex in $\tau$ be $v$, and let its update step be $t_v \leq t$. By definition of HT, $\tau$ is an ST in some $t' \leq t$. If $t' < t_v$ then trivially $\tau$ is an ST in $\Gamma_{t_v}$. Otherwise if $t'$ in $[t_v, t]$ then, by part 2 of Lemma 31, using $\hat{t} = t' = t_v$ and $t'' = t$, $\tau$ is an ST in $\Gamma_{t_v}$. By Lemma 30,

$t_v \in \textit{Prior-times}(t)$. Finally, since $|\textit{Prior-times}(t)| \leq \log n$ for any $t$, $z = O(\log n)$ in Lemma 29. □

Recall that $CH$ is the combined history of a triple $\gamma$ (as defined in Section 5.3.1). Here, we bound its space complexity.

**Corollary 4.** *The size of $CH$ is $O(\log n)$ for each THT processed during cleanup. Thus the additional processing time for a triple in cleanup is $O(\log n)$ and, by Lemma 29 part 2, the overall space of the $CH$ structures is $\widetilde{O}(\nu^{*2})$.*

*Proof.* From lemma 32, a single THT $\gamma$ can exist in at most $z = O(\log n)$ PDG (with different counts). Moreover, all the paths joining $\gamma$ in a specific PDG $\Gamma_t$ can only be generated by the node updated at time $t$. Thus, the combined history $CH(\gamma)$ contains at most $O(\log n)$ entries. □

We will use the above lemma to obtain our amortized time bound in Lemma 33 in the next section. It is not clear that a similar result can be obtained with the `DI` dummy update sequence.

## Amortized Cost of Algorithm FULLY-DYNAMIC

We will now bound the amortized cost of an update in a sequence $\Sigma$ of $n$ real updates on an initial $n$-node graph $G = (V, E)$. As mentioned earlier, in our method this will translate to a sequence $\Sigma'$ of $2n$ real (as opposed to dummy) updates: there is an initial sequence of $n$ updates, starting with the empty graph, which inserts each of the $n$ vertices in $G$ along with incident edges that have not yet been inserted. Following this is the sequence of $n$ real updates in $\Sigma$. Each of these $2n$ updates in $\Sigma'$ will make a call to Algorithm 21. As before, let $\nu^*$ be a bound on the number of edges that lie on shortest paths through any given vertex in any of $G_t$. The following lemma establishes our main Theorem 9.

**Lemma 33.** *Algorithm 21 executes a sequence $\Sigma$ of $n$ real updates on an $n$-node graph in $O(\nu^{*2} \cdot \log^3 n)$ amortized time per update.*

*Proof.* Let $n' = 2n$, and as described above, let $\Sigma'$ be the sequence of $n'$ calls to Algorithm 21 used to execute the $n$ updates in $\Sigma$. We first observe that Algorithm 21 performs $O(n' \log n')$ dummy updates when executing these $n'$ calls. This is because there are $n'/2^k$ real updates for update steps $t$ with *set-bit*$(t) = k$, and each such update is accompanied by $2^k - 1$ dummy updates. So, across all real updates there are $O(n')$ dummy updates for each position of *set-bit*, adding up to $O(n' \log n')$ in total, across all *set-bit* positions.

We now use Lemma 26 to bound the time needed to execute the $d = n' \log n'$ dummy updates and $n'$ real updates. We need to bound the parameters $C$ and $D$ in Lemma 26. We first consider $D$. For this we will use part 1 of Lemma 29. By Lemma 32, $z = O(\log n)$ for any $t$. Hence by Lemma 29 the maximum number of tuples that can remain at the end of the update sequence is $D = O(m \cdot \nu^* \cdot \log n)$.

Now we bound the parameter $C$, for which we will obtain separate bounds, $C_1$ for the real updates, and $C_2$ for the dummy updates. For each real update we have $z = z' = O(\log n)$ in part 3 of Lemma 29, hence the number of tuples that contain a path through the updated vertex is $O(\nu^{*2} \cdot \log^2 n)$, thus $C_1 = O(\nu^{*2} \cdot \log^2 n)$.

Now consider a dummy update on a vertex $u$ in Step 5 of Algorithm 21, and let $u$ be in some level $i$ (note the $i$ must be less than the current level $k$). At the time this call is made, all vertices that were updated after $u$ was last updated in the graph (i.e., after $time(i)$) are now in the newest level $k$. Thus, any LHT that contains $u$ lies in either $\Gamma_{time(i)}$ or in $G_t = \Gamma_t$. Hence $z' = 2$ in part 3 of Lemma 29 for any vertex $u$ undergoing a dummy update. Thus $C_2 = O(\nu^{*2} \cdot \log n)$.

Hence the total time taken by Algorithm 21 for its $d = n' \log n'$ dummy updates and the $n'$ real updates is, by Lemma 26, $O(\ (n' \cdot (n^2 + C_1) + (n' \log n') \cdot (n^2 + C_2) + D) \cdot \log n) = O(n' \cdot \nu^{*2} \cdot \log^2 n + (n' \cdot \log n') \cdot \nu^{*2} \cdot \log n + D \log n) = O(n \cdot \nu^{*2} \cdot \log^3 n)$

(the $n^2$ and $D$ terms are dropped since $\nu^* = \Omega(n)$, hence $m = O(n \cdot \nu^*)$).

It follows that the amortized cost of each of the $n$ updates in $\Sigma$ is $O(\frac{1}{n} \cdot n' \cdot \nu^{*2} \cdot \log^3 n) = O(\nu^{*2} \cdot \log^3 n)$. $\square$

### 5.4.5 Experimental Results

For evaluation purpose, we also designed an implementable pseudocode for our FULLY-DYNAMIC APASP algorithm. Russell F. McQueeney, an undergraduate student whose research was supervised by Prof. Ramachandran, implemented this pseudocode to obtain a working prototype of the algorithm presented in this chapter.

Russell tested his code on several graphs showing interesting experimental results when compared to an already implemented version of the `DI` algorithm [DI06]. In these experiments, we only used input graphs with unique shortest paths in order to match the requirements of `DI`. Recall that, in the presence of unique shortest paths, our FULLY-DYNAMIC algorithm is just `DI` enhanced with our new `PR` dummy sequence (described in Section 5.4.3). Thus, for our experiments, we decided to use the `DI` algorithm enhanced with our dummy sequence, and compare it with an implementation of the standard `DI` algorithm based on [DI06], which was provided to us by the authors. Here, we present the results obtained by Russell when testing our `PR` and the `DI` algorithms on the Maryland road network (Figure 5.1), and on a pathological graph (Figure 5.2). All the graphs and plots were obtained by Russell for his undergraduate research.

In the first three graphs in Fig.5.1 and 5.2, we study the behavior of the number of LHPs added, deleted and maintained in the data structures ($y$ axis), at each update ($x$ axis). Then, in graphs 4 and 5, we look at the cumulative addition and deletion of LHPs ($y$ axis) over a sequence of 1024 updates ($x$ axis). Finally, in the last graph, we look at the raw running time of the algorithms over a sequence of

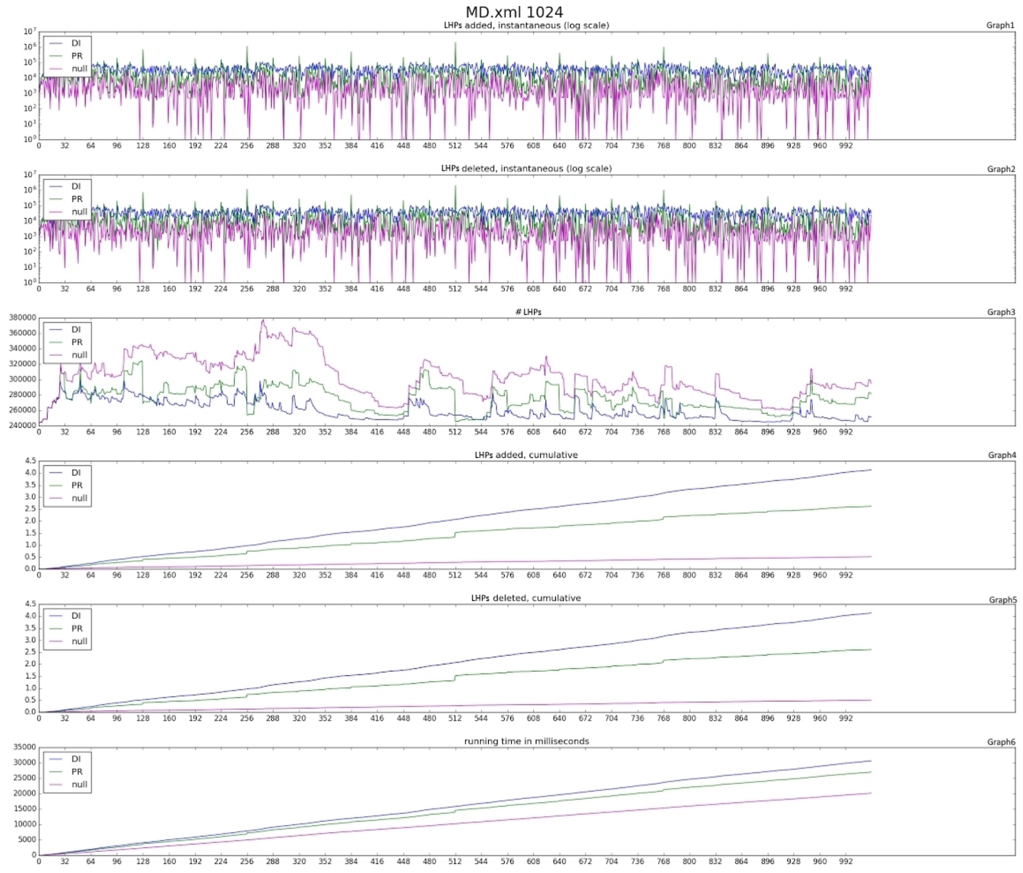


Figure 5.1: Results on the Maryland road network

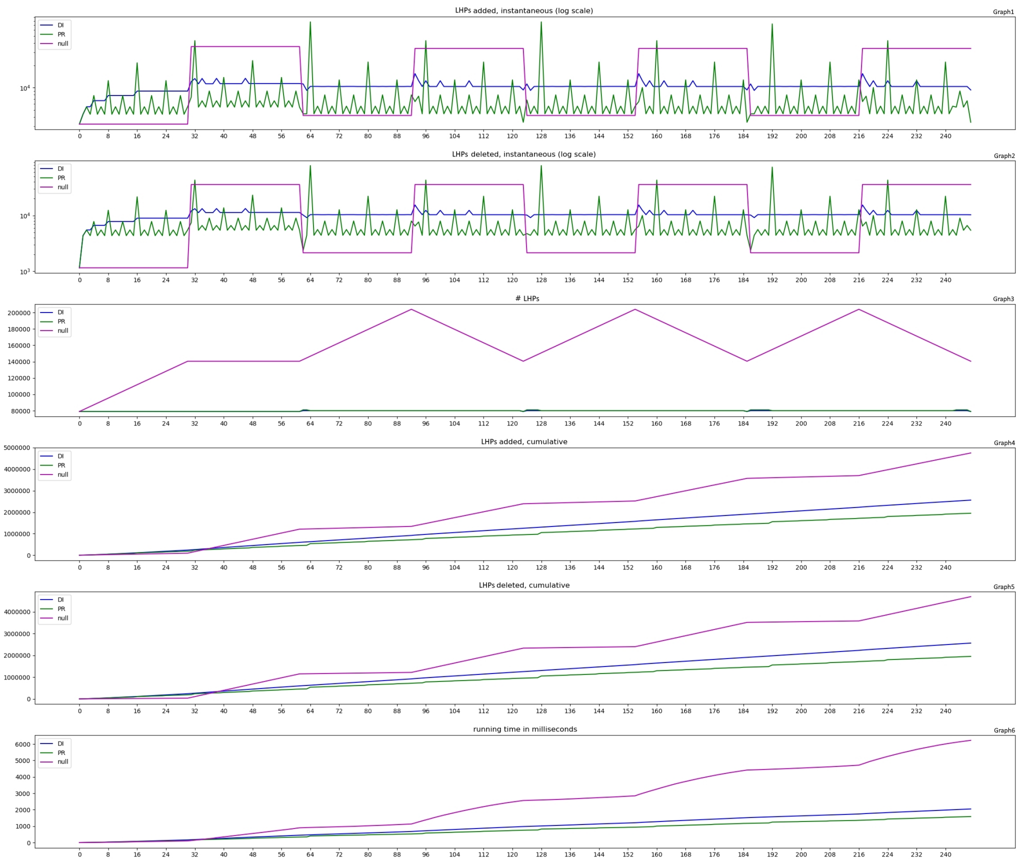


Figure 5.2: Results on a pathological graph

1024 updates. The third line plotted in each graph, named *null* in the figures, is for an implementation of `DI` which does not use the dummy sequence at all. The null algorithm is used in our experiments because it was showed in [DI06] that the best performances, in real-world graphs, are obtained by ignoring the dummy sequence (as shown in Fig.5.1). However, ignoring the dummy sequence when a pathological graph is processed can dramatically slow down the algorithm (see Fig.5.2).

The improvements obtained by our algorithm in Russell's experiments are clearly associated with the `PR` dummy sequence, that is used to keep under control the number of LHPs in the system. When our dummy sequence is used, the average number of LHPs in the data structures, at any one time, is significantly less than the number of LHPs maintained by `DI`. This is accomplished with fewer dummy updates as well. However, this improvement comes at the cost of local space: the `PR` dummy sequence allows many LHPs to accumulate before removing them with a sequence of dummy updates. This is sometime a problem on pathological graphs (see [DI04]), where the number of LHPs and HPs is large. As observed in the experimental results our approach outperforms `DI` although, in real-world graphs, skipping dummy updates remains the fastest solution.

Russell also developed a working implementation of our FULLY-DYNAMIC algorithm for computing APASP, which we could not test against `DI` given its unique shortest path requirement. However, this implementation was extensively tested for correctness and provided important insights for rewriting an improved and more readable pseudocode for our APASP algorithm.

# Chapter 6

# Faster Fully Dynamic Algorithm

In this chapter we present our improved algorithm FFD for computing APASP, which uses new techniques and data structures (see Section 6.1.1)[1]. Our approach is based on the results obtained in [Tho04], for which we give a brief introduction in the following section. We then discuss the new data structures involved in our result (Sections 6.1.1), new features not present in [Tho04], that arise from having multiple SPs (Sections 6.1.2), and an overall description of the FFD algorithm and its new components (Sections 6.1.3). The complexity of FFD is discussed in Section 6.3, while the pseudocode and correctness are available in Section 6.2. The chapter ends with a conclusion in Section 6.4.

**The `Thorup` Fully Dynamic APSP Algorithm:** In [Tho04], Thorup improves by a logarithmic factor over `DI` (for unique shortest paths) by using a *level system* of increase-only graphs. The shortest paths and locally shortest paths are generated level by level leading to a different complexity analysis from `DI`. When a node is removed from the current graph, it is also removed from every older level graph that contains it. The implementation of the `Thorup` APSP algorithm is not fully

[1]The results presented in this chapter appeared in [PR15b].

specified in [Tho04]. For our FFD algorithm, we present generalizations of the data structures sketched in `Thorup` together with new data structures required to achieve efficiency (see Section 6.1 for a summary of these data structures).

## 6.1 Data Structures for Algorithm FFD

Our algorithm FFD requires several data structures. Some of these are already present in `NPRdec` and `Thorup`, while others are newly defined or generalized from earlier ones. We will use components from our basic algorithm such as the abstract representation of the level system using PDGs (see Section 6.1.1) and the flag bit $\beta$ for a triple, the Marked-Tuples scheme introduced in `NPRdec` (see Chapter 3 for more details), and the maintenance of level graphs from `Thorup`.

In the following sections, we describe all data structures used by our algorithm. In Table II we summarize the structures we use, including those inherited from [NPR14b, Tho04]. The new components we introduce in this chapter to achieve efficiency for fully dynamic APASP, are described in section 6.1.1 and listed in Table II, Part D.

### 6.1.1 A Level System for Centered Tuples

Algorithm FFD uses the PDGs defined in 5.4.4 as real data structures similar to `Thorup` for APSP. This is done in order to generate a smaller superset of LSTs than FULLY-DYNAMIC, and it is the key to achieving the improved efficiency. Here we describe the level system and the data structures we use in FFD, with special attention to the new elements we introduce.

As in [DI04, Tho04] we build up the tuple-system for the initial $n$-node graph $G = (V, E)$ with $n$ insert updates (starting with the empty graph), and we then perform $n$ updates according to the update sequence $\Sigma$. After $2n$ updates, we reset all data structures and start afresh.

Our level system is a generalization of `Thorup` to fully dynamic APASP. For an update at step $t$, let $k$ be the position of the least significant bit with value 1 in the binary representation of $t$. Then the $t$-th update activates level $k$, and deactivates all levels $j < k$ by folding these levels into level $k$. These levels are considered implicitly in our basic result, and using the same notation, we will say that $\mathit{time}(k) = t$, and $\mathit{level}(t) = k$; moreover $G_t$ indicates the graph after the $t$-th update. Note that the largest level created before we start afresh is $r = \log 2n$.

**Centering vertices and tuples/triples** As in `Thorup`, each *vertex $v$ is centered* in level $k = \mathit{level}(t)$, where $t$ is the most recent step in which $v$ was updated. A path $p$ in a tuple is centered in level $k' = \mathit{level}(t')$, where $t'$ is the most recent step in which $p$ entered the tuple system (within some tuple) or was modified by a vertex update. Hence, in contrast to `Thorup`, a triple can represent paths centered in different levels. Thus, for a triple $\gamma = ((xa, by), wt, \mathit{count})$ we maintain an array $C_\gamma$ where

$$C_\gamma[i] = \text{number of paths represented by } \gamma \text{ that are all centered in level } i$$

It follows that $\sum_i C_\gamma[i] = \mathit{count}$. The *level center of the triple* $\gamma$ is the smallest (i.e., most recent level) $i$ such that $C_\gamma[i] \neq 0$.

**Level graphs (PDGs)** In our basic result, the PDGs (introduced in 5.4.4) are used only in the analysis, and are not maintained by the algorithm. Here, in FFD, we will maintain a set of local data structures for each PDG that is relevant to the current graph; also, in a small change of notation, we will denote a level graph for time $t' \leq t$ as $\Gamma_{k'}$, where $k' = \mathit{level}(t')$ rather than the our previous notation of $\Gamma_{t'}$. These graphs are similar to the level graphs in `Thorup`. As in `Thorup`, only certain information for $\Gamma_k$ is explicitly maintained in its local data structures: the

STs centered in level $k$ plus all the extensions that can generate STs in $\Gamma_k$. The data structures used by our algorithm to maintain triples are Global and Local, which we now describe.

| **Notation** | **Data Structure** | **Appears** |
|---|---|---|
| **Part A :: Global Data Structures** (for each pair of nodes $(x, y)$) | | |
| $P(x, y)$ | all (centered) LHT s from $x$ to $y$ with weight as key | [DI04] for paths [NPR14b] for LSTs |
| $P^*(x, y)$ | all (centered) HT s from $x$ to $y$ with weight as key | |
| $L(x, by)$ | $\{x' : (x'x, by)$ denotes a (centered) LHT$\}$ | [NPR14b] for LSTs |
| $R(xa, y)$ | $\{y' : (xa, yy')$ denotes a (centered) LHT$\}$ | [NPR14b] for LSTs |
| Marked-Tuples | global dictionary for Marking scheme | [NPR14b] |
| **Part B :: Local Data Structures** (for each active level $i$, for each pair of nodes $(x, y)$) | | |
| $P_i^*(x, y)$ | STs from $x$ to $y$ centered in level $i$ | sketched in [Tho04] for paths |
| $L_i^*(x, y)$ | $\{x' : (x'x, y)$ denotes an $\ell$-tuple for SPs centered in level $i\}$ | |
| $R_i^*(x, y)$ | $\{y' : (x, yy')$ denotes an $r$-tuple for SPs centered in level $i\}$ | |
| $LC_i^*(x, y)$ | the subset $\{x' \in L_i^*(x, y) : x'$ is centered in level $i\}$ | |
| $RC_i^*(x, y)$ | the subset $\{y' \in R_i^*(x, y) : y'$ is centered in level $i\}$ | |
| $dict_i$ | dictionary of pointers from local STs to global $P$ and $P^*$ | new |
| **Part C :: Inherited Data Structures** | | |
| $\beta(\gamma)$ | flag bit for the (centered) triple $\gamma$ | basic algorithm |
| $level(t)$ | level activated during $t$-th update | basic algorithm |
| $time(k)$ | most recent update in which level $k$ is activated | basic algorithm |
| $\mathcal{N}$ | nodes (centered in levels) deactivated in the current step | basic algorithm |
| $\Gamma_k$ | level graph (PDG) created during $time(k)$-th update | [Tho04] for paths |
| **Part D :: New Data Structures** | | |
| $C_\gamma$ | distribution of all paths in triple $\gamma$ among active levels | new |
| $DL(x, y)$ | linked-list containing the history of distances from $x$ to $y$ | new |
| $LN(x, y, wt)$ | the set $\{b : \exists\, (xa, by)$ of weight $wt$ in $P(x, y)\}$ | new |
| $RN(x, y, wt)$ | the set $\{a : \exists\, (xa, by)$ of weight $wt$ in $P(x, y)\}$ | new |

Table 6.1: Notation summary

**Global Structures** The global data structures are $P^*$, $P$, $L$ and $R$ (see Table II, Part A).

- The structures $P^*(x, y)$ and $P(x, y)$ will contain HTs (including all STs) and LHTs (including all LSTs), respectively, from $x$ to $y$. They are priority queues with the weights of the triple and a flag bit $\beta$ as key. For a triple $\gamma$ in $P$, the flag bit $\beta(\gamma) = 0$ if the triple $\gamma$ is in $P$ but not in $P^*$, and $\beta(\gamma) = 1$ if the triple $\gamma$ is in $P$ and $P^*$.

- The structure $L(x, by)$ ($R(xa, y)$) is the set of all left (right) extension vertices that generate a centered LHT in the tuple-system.

**Local Structures** The local data structures we introduce in this chapter are $L_i^*, R_i^*, LC_i^*$ and $RC_i^*$ (see Table II, Part B). These are generalization of the data structures sketched in `Thorup` for unique SPs in the graph. For every pair of nodes $(x, y)$:

- The structure $P_i^*(x, y)$ contains the set of STs from $x$ to $y$ centered in $\Gamma_i$. It is implemented as a set.
- The structure $L_i^*(x, y)$ ($R_i^*(x, y)$) contains all left (right) extensions that generate a shortest $\ell$-tuple ($r$-tuple) centered in level $i$. It is implemented as a balanced search tree.
- The structure $LC_i^*(x, y)$ ($RC_i^*(x, y)$) contains left (right) extensions centered in level $i$ that generate a shortest $\ell$-tuple ($r$-tuple) centered in level $i$. It is implemented as a balanced search tree.
- A dictionary $dict_i$, contains STs in $P_i^*$ using the key $[x, y, a, b]$ and two pointers stored along with each ST. The two pointers refer to the location in $P(x, y)$ and $P^*(x, y)$ of the triple of the form $(x, a, b, y)$ contained in $P_i^*(x, y)$.

In order to recompute BC scores (see Section 5.1) we will consider $R^*(x, y) = \bigcup_i R_i^*(x, y)$ and similarly $L^*(x, y) = \bigcup_i L_i^*(x, y)$.

**New Structures**

We introduce two completely new data structures, not used in previous results [DI04, Tho04, NPR14b], which are essential to achieve efficiency for our FFD algorithm. Both are needed to address the Partial Extension Problem (PEP) which does not occur in `Thorup` for fully dynamic APSP (see section 6.1.2).

**Distance History Matrix** The *distance history matrix* is a matrix $DL$ where each entry is a pointer to a linked list: for each $x, y \in V$, the linked list $DL(x, y)$ contains the sequence of different pairs $(wt, k)$, where each one represents an SP weight $wt$ from $x$ to $y$, along with the most recent level $k$ in which the weight $wt$ was the shortest distance from $x$ to $y$ in the graph $\Gamma_k$. The pair with weight $wt$ in $DL(x, y)$ is double-linked to every triple from $x$ to $y$ with weight $wt$ in the system. Precisely, when a new triple $\gamma$ from $x$ to $y$ of weight $wt$ is inserted in the algorithm, a link is formed between $\gamma$ and the pair $(wt, k)$ in $DL(x, y)$. With this structure, the FFD algorithm can quickly check if there are still triples of a specific weight in the tuple-system, especially for example when we need to remove a given weight from $DL$. Note that the size of each linked list is $O(\log n)$.

**Historical Extension (HE) Sets RN and LN** Another important type of structure we introduce are the sets $RN$ and $LN$. These structures are crucial to select efficiently the set of restored historical tuples that need to be extended (see Section 6.2.2). $RN(x, y, wt)$ ($LN(x, y, wt)$ works symmetrically) contains all nodes $b$ such that there exists at least one tuple of the form $(x\times, by)$ and weight $wt$ in $P(x, y)$. Similarly to $DL$, every time a new triple $\gamma$ of this form is inserted in the tuple-system, a double link is created between $\gamma$ and the occurrence of $b$ in $RN(x, y, wt)$ in order to quickly access the triple when needed.
The total space used by $DL$, $RN$ and $LN$ is $O(n^2 \log n)$. This is dominated by the overall space used by the algorithm to maintain all the triples in the tuple-system across all levels (see Lemma 40, Section 6.3).

### 6.1.2 New Features in Algorithm FFD

In this section, we discuss two challenges that arise when we attempt to generalize the level graph method used in `Thorup` (for APSP with unique SPs) to a fully dynamic APASP algorithm. Both are addressed by the algorithms in Section 6.2.

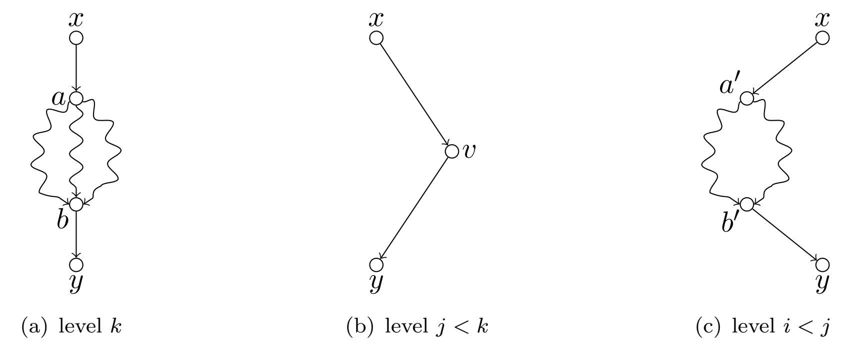
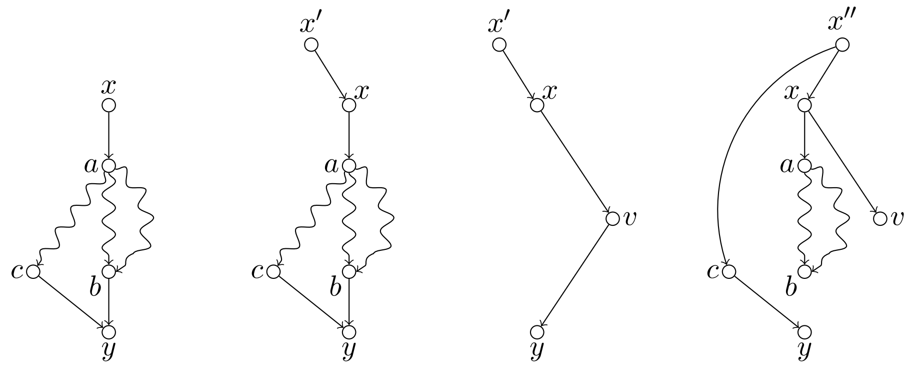


Figure 6.1: The bit $\beta$ feature

**The bit $\beta$ feature** The control-bit $\beta$ is introduced (and only briefly described) in our basic result to avoid the processing of untouched historical triples. Here, we elaborate on this technique in more details and we also describe how it helps in the more complex setting of the level tuple-system.

Consider figure 6.1. The ST $\gamma = ((xa, by), wt, count)$ is created in level $k$ (Fig. 6.1(a)). At $time(k)$, we have $\gamma \in P^*$ and also $\gamma \in P$ with $\beta(\gamma) = 1$. In a more recent level $j < k$, a shorter triple $\gamma' = ((xv, vy), wt', count')$, with weight $wt' < wt$, that goes trough an updated vertex $v$ is generated (Fig. 6.1(b)). Thus at $time(j)$, we have $\gamma' \in P^*$ and also $\gamma' \in P$ with $\beta(\gamma') = 1$; but $\gamma$ still appears in both $P^*$ and $P$ as a historical triple. Finally, a new LST $\gamma'' = ((xa', b'y), wt, count'')$, with the same weight as $\gamma$, is generated in level $i < j$ (Fig. 6.1(c)). Note that $\gamma''$ is only in $P$ with $\beta(\gamma'') = 0$ and not in $P^*$, as is the case of every LST that is not an ST. When an increase-only update removes $v$ and the triple $\gamma'$, the algorithm needs to restore all the triples with shortest weight $wt$. But while $\gamma$ is historical and does not require any additional extension, $\gamma''$ is only present in $P$ and needs to be processed. Our FFD algorithm achieve this by checking the bit $\beta$ associated to each of these triples. The algorithm will extract and process all the triples with $\beta = 0$ from $P(x, y)$. These guarantees that a triple only present in $P$, or present in $P$ and $P^*$ with different counts is never missed by the algorithm.



(a) level $k$ (b) level $j < k$ (c) level $i < j$ (d) level $h < i$

Figure 6.2: PEP instance (only centered STs are kept in each level) – all edge weights are unitary

**The partial extension problem (PEP)** Consider the update sequence described below and illustrated in figure 6.2. Here the STs $\gamma = ((xa, by), wt, count)$ and $\hat{\gamma} = ((xa, cy), wt, count')$ are created in level $k$ (Fig. 6.2(a)). Later, a left-extension to $x'$ generates the STs $\gamma' = ((x'x, by), wt', count)$ and $\hat{\gamma}' = ((x'x, cy), wt', count')$ in level $j < k$ (Fig. 6.2(b)). Note that $\gamma$, $\hat{\gamma}$, $\gamma'$ and $\hat{\gamma}'$ are all present in $P^*$ and $P$ at $time(j)$. In a more recent level $i < j$, a decrease-only update on $v$ generates a shorter triple $\gamma_s = ((xv, vy), wt_s, count_s)$ from $x$ to $y$, with $wt_s < wt$ going through $v$. In the same level, the triple $\gamma_s$ is also extended to $x'$ generating a triple $\gamma'' = ((x'x, vy), wt'', count_s)$ shorter than $\gamma'$ and $\hat{\gamma}'$ (Fig. 6.2(c)). Thus at $time(i)$, $\gamma$, $\hat{\gamma}$, $\gamma'$ and $\hat{\gamma}'$ remain in $P^*$ as historical triples. Then, in level $h < i$, an update on $x''$ inserts the edges $(x'', x)$ and $(x'', c)$. This update generates an ST $\gamma''' = ((x''c, cy))$ (shorter than $(x''x, vy)$) and also inserts $x'' \in LC^*_h(x, b))$ since $(x'', x)$ is on a shortest path from $x''$ to $b$; but it should not generate the triple of the form $(x''x, by)$ because $b$ is not on a shortest path from $x$ to $y$ at $time(h)$ (Fig. 6.2(d)).

When an increase-only update removes $v$ and the shortest triple $\gamma_s$ from $x$ to $y$, the algorithm needs to restore all historical triples with shortest weights from $x$ to $y$. When $\gamma$ and $\hat{\gamma}$ are restored, we need to perform suitable left extensions as follows. An extension to $x''$ is needed only for $\gamma$: in fact $\hat{\gamma}$ should not be extended to $x''$ because the $\ell$-tuple of the newly generated tuple is not an ST in the graph. On the other hand, no extension to $x'$ is needed since both $\gamma'$ and $\hat{\gamma}'$ will be restored (from HT to ST). Our algorithm needs to distinguish all of these cases correctly and efficiently.

In order to maintain both correctness and efficiency in this scenario for APASP, we use two new data structures (described in Section 6.1.1): (1) the historical distance matrix $DL$ that allow us to efficiently determine the most recent level graph in which an HT was an ST, and (2) the HE sets $LN$ and $RN$ that allow us to efficiently identify exactly those new extensions that need to be performed. The methodology of these data structures is fully discussed in the description of FF-FIXUP (Section 6.2.2). Note that, the PEP doesn't arise in `Thorup` because of the unique SP assumption: in fact when only a single SP of a given length is present in the graph for each pair of nodes, the algorithm can check for all the $O(n^2)$ paths maintained in each level and decide which one should be extended. Given the presence of multiple SPs in our setting, we cannot afford to look at each tuple in the tuple-system .

### 6.1.3 An Overview of the FFD Algorithm

We now give an overall summary of the FFD algorithm. A detailed description of its sub-procedures can be found in the next section.

Algorithm FFD is similar to our basic fully dynamic algorithm and its overall description is given in Algorithm 22. The main difference is the introduction of the notion of levels as described in Section 6.1, and their activation/deactivation as in `Thorup`. At the beginning of the $t$-th update (with $k = level(t)$), we first activate the new level $k$ and we perform FF-UPDATE (Alg. 23) on the updated node $v$. As in our basic algorithm and shown in Table II - Part C, the set $\mathcal{N}$ consists of all vertices centered at these lower deactivated levels. All vertices in $\mathcal{N}$ are re-centered at level $k$ during the $t$-th update (Alg. 22, Step 5), and 'dummy' update operations are performed on each of these vertices. Note that $\mathcal{N}$ contains the $2^k - 1$ most recently updated vertices in reverse order of update time (from the most recent to the oldest). Procedure FF-UPDATE is invoked with the parameter $k$ representing the newly activated level. Finally, all levels $j < k$ are deactivated (Alg. 22, Step 6).

**Algorithm 22** FFD$(G, v, \mathbf{w}', k)$

```
1: activate the new level $k$
2: FF-UPDATE$(v, \mathbf{w}', k)$
3: generate the set $\mathcal{N}$
4: for each $u \in \mathcal{N}$ in decreasing order of update time do
5:     FF-UPDATE$(u, \mathbf{w}', k)$ {dummy updates}
6: deactivate all levels lower than $k$
```

FF-UPDATE As in `DI`, `NPRdec` and our basic algorithm, the update of a node occurs in a sequence of two steps: a *cleanup phase* and a *fixup phase*. Both FF-CLEANUP and FF-FIXUP are more involved algorithms than their counterparts in our basic result, and the resulting algorithm will save a $O(\log n)$ factor over the amortized cost. Note that FF-FIXUP is called with the additional argument $k$ which indicates the current active level.

**Algorithm 23** FF-UPDATE$(v, \mathbf{w}', k)$

```
1: FF-CLEANUP$(v)$
2: FF-FIXUP$(v, \mathbf{w}', k)$
```

We now highlight some new components of our cleanup and fixup algorithms that will be helpful to prove the lemmas in the next section.

**New Components in FF-CLEANUP relative to FULLY-CLEANUP**

FF–C.1 For each triple $\gamma$ processed by FF-CLEANUP, the array $C_\gamma$ can be updated with the new centers in time $O(z')$, where $z'$ is the number of active levels that contains the updated vertex $v$.

FF–C.2 Each triple requires constant time to be linked and unlinked from structures $DL$, $RN$ and $LN$.

**New Components in FF-FIXUP relative to FULLY-FIXUP**

FF–F.1 For each triple $\gamma$ processed by FF-FIXUP, the array $C_\gamma$ can be updated with the new centers in time $O(z')$, where $z'$ is the number of active levels that contain the triple $\gamma$.

FF–F.2 Each triple requires constant time to be linked and unlinked from structures $DL$, $RN$ and $LN$.

FF–F.3 FF-FIXUP only processes a triple $\gamma$ with $\beta = 1$ if it has a centered extensions in some active level younger than then the level in which $\gamma$ was shortest for the last time.

We will establish in Section 6.3 that FFD correctly updates the data structures with the amortized bound given in Theorem 9.

## 6.2 The FFD Algorithm for APASP

### 6.2.1 Description of FF-CLEANUP

FF-CLEANUP removes all the LHPs going through the updated vertex $v$ from all the global structures $P$, $P^*$, $L$ and $R$, and from all local structures in any active level graph $\Gamma_j$ that contains these triples. This involves decrementing the count of some triples or removing them completely (when all the paths in the triple go through $v$). The algorithm also updates local dictionaries and the $DL$, $RN$ and $LN$ structures.

Algorithm FF-CLEANUP is a natural extension of the `NPRdec` cleanup. An extension of the `NPRdec` cleanup is used also in our basic algorithm but in a different way.

**Algorithm 24** FF-CLEANUP($v$)
```
1: $H_c \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2: $\gamma \leftarrow [(v,v),0,1]$; $C_\gamma[center(v)] = 1$; add $[\gamma, C_\gamma]$ to $H_c$
3: while $H_c \neq \emptyset$ do
4:     extract in $S$ all the triples with the same min-key $[wt,x,y]$ from $H_c$
5:     FF-CLEANUP-$\ell$-extend($S$,$[wt,x,y]$) (see Algorithm 25)
6:     FF-CLEANUP-$r$-extend($S$,$[wt,x,y]$)
```

FF-CLEANUP starts as in the `NPRdec` algorithm. We add the updated node $v$ to $H_c$ (Step 2 – Alg. 24) and we start extracting all the triples with same min-key (Step 4 – Alg. 24). The main differences from `NPRdec` start after we call Algorithm 25. As in Chapter 3, we start by forming a new triple $\gamma'$ to be deleted (Steps 5 – Alg. 24). A new feature in Algorithm 25 is to accumulate the paths that we need to remove level by level using the array $C'$. This is inspired by `Thorup` where unique SPs are maintained in each level. However, our algorithm maintains multiples paths spread across different levels using the $C_\gamma$ arrays associated to LSTs, and the technique used to update the $C_\gamma$ arrays is significantly different and more involved than the one described in `Thorup`. Step 6 - Alg. 25 calls FF-CLEANUP-CENTERS (Alg. 26) that will perform this task.

FF-CLEANUP-CENTERS takes as input the generated triple $\gamma'$ of the form $(x'x, by)$ which contains all the paths going through the updated node $v$ to be removed, and the set of triples $S_b$ of the form $(x\times, by)$ that are extended to $x'$ to generate $\gamma'$. This procedure has two tasks: (1) generating the $C_{\gamma'}$ vector for the triple $\gamma'$ that will be reinserted in $H_c$ for further extensions, and (2) updating the $C_{\gamma''}$ vector for the tuple $\gamma''$ in $P(x', y)$ (note that $\gamma''$ is the corresponding triple in $P$ of $\gamma'$, before we subtract all the paths represented by $\gamma'$ level by level).

(1) - This task, which is more complex than the second task (which is a single step

**Algorithm 25** FF-CLEANUP-$\ell$-extend$(S, [wt, x, y])$

```
1:  for every $b$ such that $(x\times, by) \in S$ do
2:      let $S_b \subseteq S$ be the set of all triples of the form $(x\times, by)$
3:      let $fcount'$ be the sum of all the $counts$ of all triples in $S_b$
4:      for every $x'$ in $L(x, by)$ s.t. $(x'x, by) \notin$ Marked-Tuples do
5:          $wt' \leftarrow wt + \mathbf{w}(x', x)$; $\gamma' \leftarrow ((x'x, by), wt', fcount')$
6:          $C_{\gamma'} \leftarrow$ FF-CLEANUP-CENTERS$(\gamma', S_b)$
7:          add $[\gamma', C_{\gamma'}]$ to $H_c$
8:          remove $\gamma'$ in $P(x', y)$ // decrements $count$ by $fcount'$
9:          set new center for $\gamma'' = ((x'x, by), wt')$ in $P(x', y)$ as $\mathrm{argmin}_i(C_{\gamma''}[i] \neq 0)$
10:         if a triple for $(x'x, by)$ exists in $P(x', y)$ then
11:             insert $(x'x, by)$ in Marked-Tuples
12:         else
13:             delete $x'$ from $L(x, by)$ and delete $y$ from $R(x'x, b)$
14:         if no triple for $((x'-, by), wt')$ exists in $P(x', y)$ then
15:             remove $b$ from $RN(x', y, wt')$
16:         if no triple for $((x'x, -y), wt')$ exists in $P(x', y)$ then
17:             remove $x$ from $LN(x', y, wt')$
18:         if a triple for $((x'x, by), wt')$ exists in $P^*(x', y)$ then
19:             remove $\gamma'$ in $P^*(x', y)$ // decrements $count$ by $fcount'$
20:             if $\gamma' \notin P^*(x', y)$ then
21:                 remove the element with weight $wt'$ from $DL(x', y)$ if not linked to
                    other tuples in $P^*(x', y)$
22:             for each $i$ do
23:                 decrement $C_{\gamma'}[i]$ paths from $\gamma' \in P^*_i(x', y)$
24:                 if $\gamma'$ is removed from $P^*_i(x', y)$ then
25:                     if $x'$ is centered in level $i$ then
26:                         if $\forall j \geq i, P^*_j(x, y) = \emptyset$ then
27:                             remove $x'$ from $L^*_i(x, y)$ and remove $x'$ from $LC^*_i(x, y)$
28:                     else if $P^*_i(x, y) = \emptyset$ then
29:                         remove $x'$ from $L^*_i(x, y)$
30:                     if $y$ is centered in level $i$ then
31:                         if $\forall j \geq i, P^*_j(x', b) = \emptyset$ then
32:                             remove $y$ from $R^*_i(x', b)$ and remove $y$ from $RC^*_i(x', b)$
33:                     else if $P^*_i(x', b) = \emptyset$ then
34:                         remove $y$ from $R^*_i(x', b)$
```

**Algorithm 26** FF-CLEANUP-CENTERS($\gamma'$, $S_b$)

```
1: let $\gamma' = ((x'x, by), wt', fcount')$ (the triple of the form $((x'x, by), wt')$ that contains all the paths through $v$ to be removed)
2: let $\gamma'' = ((x'x, by), wt', fcount'')$ (the triple of the form $((x'x, by), wt')$ in $P(x', y)$. Note that $\gamma'$ represents a subset of $\gamma''$)
3: $j \leftarrow \text{argmax}_j(C_{\gamma''}[j] \neq 0)$ // This is the oldest level in which a path in $\gamma''$ appeared for the first time
4: $C' \leftarrow \sum_{\gamma \in S_b} C_\gamma[r-1, \ldots, 0]$ // This is the sum (level by level) of the triples of the form $(xa_i, by)$ that go through $v$ and are extending to $\gamma'$ during this stage
5: create a new center vector $C_{\gamma'}$ for the triple $\gamma'$ as follows
6:     for all the levels $m > j$ we set $C_{\gamma'}[m] = 0$
7:     for the level $j$ we set $C_{\gamma'}[j] = \sum_{k=j}^{r-1} C'[k]$
8:     for all the levels $i < j$ we set $C_{\gamma'}[i] = C'[i]$
9: $C_{\gamma''}[r-1, \ldots, 0] \leftarrow C_{\gamma''}[r-1, \ldots, 0] - C_{\gamma'}[r-1, \ldots, 0]$ // We update the $C$ vector for $\gamma'' \in P(x', y)$
10: return $C_{\gamma'}$ // We return the correct vector for the generated $\gamma'$ triples
```

in the algorithm, see point (2) below), is accomplished in steps 4 to 8, Alg. 26 and uses the following technique. In step 4 – Alg. 26, we store into the $\log n$-size array $C'$ the distribution over the active levels for the set of triples in $S_b$ that generates $\gamma'$ using the left extension to $x'$. In order to generate the correct vector $C_{\gamma'}$ (to associate with the triple $\gamma'$), we need to reshape the distribution in $C'$ according to the corresponding distribution of the triple $\gamma'' \in P$. The reshaping procedure works as follows: we first identify the oldest level $j$ in which the triple $\gamma''$ appeared in $P$ for the first time (Step 3 – Alg. 26). Recall that we want to remove $\gamma'$ paths containing $v$ from $\gamma''$, and $\gamma''$ does not exist in any level older than $j$. Vector $C'$ is the sum of $C_\gamma$ for all $\gamma \in S_b$ (Step 4 – Alg. 26). Those triples are of the form $(xa_i, by)$ and they could exist in levels older, equal or more recent than $j$. But the triples in $S_b$ that were present in a level older than $j$, were extended to $\gamma''$ in $P$ for the first time in level $j$. For this reason, step 7 - Alg. 26 aggregates all the counts in $C'$ in levels older or equal $j$ in $C_{\gamma'}[j]$. Moreover, for each level $i < j$, if a triple $\gamma \in S_b$ is present in the level graph $\Gamma_i$ with *count* paths centered in level $i$, then $\Gamma_i$

also contains its extension to $x'$ that is a subtriple of $\gamma''$ located in level $i$ with at least *count* paths. Thus for each level $i < j$ step 8 - Alg. 26, copies the number of paths level-wise. This procedure allows us to precisely remove the LHPs only from the level graphs where they exist. After $C'$ is reshaped into $C_{\gamma'}$ (steps 5 to 8 - Alg. 26), the algorithm returns this correct array for $\gamma'$ to Alg. 25.
(2) - This task is performed by the simple step 9, Alg. 26, which is a subtraction level by level of LHPs.

After adding the new triple $\gamma'$ to $H_c$ (Step 7 - Alg. 25), the algorithm continues as the `NPRdec` (Steps 7 to 13 – Alg. 25) with some differences: we need to update centers, local data structures, $DL$, $RN$ and $LN$. We update the center of $\gamma'$ using $C_{\gamma'}$ (Step 9 - Alg. 25). If $\gamma'$ is a shortest triple, we decrement the count of $\gamma' \in P^*(x', y)$ (Step 19 - Alg. 25). If $\gamma'$ is completely removed from $P^*(x', y)$ and $DL(x', y, wt')$ is not linked to any other tuple, we remove the entry with weight $wt'$ from $DL(x', y)$ (Step 21 - Alg. 25). Moreover, we subtract the correct number of paths from each level using the (previously built) array $C_{\gamma'}$ (Step 23 - Alg. 25). Finally for each active level $i$, if $\gamma'$ is removed from $P_i^*(x', y)$, we take care of the sets $L_i^*$ and $R_i^*$ (Steps 24 to 34 - Alg. 25). In the process, we also update $LC_i^*$ and $RC_i^*$ in case the endpoints of $\gamma'$ are centered in level $i$. If $\gamma'$ is completely removed from $P(x', y)$, using the double links to the node $b$ in $RN(x', y, wt')$, we check if there are other triples that use $b$ in $P(x', y)$ (Step 14 - Alg. 25): if not we remove $b$ from $RN(x', y, wt')$. A similar step handles $LN(x', y, wt')$.

### 6.2.2 Description of FF-FIXUP

FF-FIXUP is an extension of FULLY-FIXUP rather than `NPRdec`. This is because of the presence of the control bit $\beta$ (defined in Section 6.1), and the need to process historical triples (that are not present in `NPRdec`). Algorithm FF-FIXUP will efficiently maintain exactly the LSTs and STs for each level graph in the tuple-system.

This is in contrast to FULLY-FIXUP, which can maintain LHTs that are not LSTs in any level graph (PDG). FF-FIXUP maintains a heap $H_f$ of candidate LHTs to be processed in min-weight order. The main phase (Alg. 27) is very similar to the fixup in our basic algorithm. The differences are again related to levels, centers and the new data structures.

We start describing Algorithm 27. We initialize $H_f$ by inserting the edges incident on the updated vertex $v$ with their updated weights (Steps 2 to 7 – Alg. 28), as well as a candidate min-weight triple from $P$ for each pair of nodes $(x, y)$ (Step 10 – Alg. 28). Then we process $H_f$ by repeatedly extracting collections of triples of the same min-weight for a given pair of nodes, until $H_f$ is empty (Steps 3 to 10 – Alg. 27). We will establish that the first set of triples for each pair $(x, y)$ always represents the shortest path distance from $x$ to $y$ (see Lemma 35), and the triple extracted are added to the tuple-system if not already there (see Alg. 29 and Lemma 36). For efficiency, among all the triples present in the tuple-system for a pair of nodes, we select only the ones that need to be extended: this task is performed by Algorithm 29 (this step is explained later in the description). After the triples in $S$ are left and right extended by Algorithm 30, we set the bit $\beta(\gamma') = 1$ for each triple $\gamma'$ that is identified as shortest in $S$, since $\gamma'$ is correctly updated both in $P^*(x, y)$ and $P(x, y)$ (Step 9 – Alg. 27). Finally, we update the $DL(x, y)$ structure by inserting (or updating if an element with weight $wt$ is already present) the element with weight $wt$ and the current level at the end of the list (Step 10 – Alg. 27). This concludes the description of Algorithm 27.

We now describe Algorithm 29 which is responsible to select only the triples that have valid extensions that will generate LHTs in the current graph. In Algorithm 29, we distinguish two cases. When the set of extracted triples from $x$ to $y$ contains at least one path not containing $v$ (Step 2 – Alg. 29), then we process all the triples from $P(x, y)$ of the same weight. Otherwise, if all the paths extracted go through $v$ (Step 18 – Alg. 29), we only use the triples extracted from $H_f$.

**Algorithm 27** FF-FIXUP$(v, \mathbf{w}', k)$

```
1: $H_f \leftarrow \emptyset$; Marked-Tuples $\leftarrow \emptyset$
2: FF-POPULATE-HEAP$(v, \mathbf{w}', k)$
3: while $H_f \neq \emptyset$ do
4:     extract in $S'$ all the triples with min-key $[wt, x, y]$ from $H_f$
5:     if $S'$ is the first extracted set from $H_f$ for $x, y$ then
6:         $S \leftarrow$ FF-NEW-PATHS$(S', P(x, y))$
7:         FF-FIXUP-$\ell$-extend$(S,[wt, x, y])$ (see Algorithm 30)
8:         FF-FIXUP-$r$-extend$(S,[wt, x, y])$
9:         for every $\gamma \in S$ set $\beta(\gamma) = 1$
10:        add an element with weight $wt$ and level $k$ to $DL(x, y)$ or update the level
           in the existing one
```

**Algorithm 28** FF-POPULATE-HEAP$(v, \mathbf{w}', k)$

```
1: for each $(u, v)$ do
2:     $\mathbf{w}(u, v) = \mathbf{w}'(u, v)$
3:     if $\mathbf{w}(u, v) < \infty$ then
4:         $\gamma = ((uv, uv), \mathbf{w}(u, v), 1)$; $C_\gamma[k] \leftarrow 1$
5:         update-num$(\gamma) \leftarrow$ curr-update-num; num-v-paths$(\gamma) \leftarrow 1$
6:         add $[\gamma, C_\gamma]$ to $H_f$ and $P(u, v)$
7:         add $u$ to $L(-, vv)$ and $v$ to $R(uu, -)$
8: for each $(v, u)$ do
9:     symmetric processing as Steps 2–7 above
10: for each $x, y \in V$ do
11:     add a min-key triple $[\gamma, C_\gamma] \in P(x, y)$ to $H_f$
```

Both cases have a similar approach but here we focus on the former which is more involved than the latter. As soon as we identify a new triple $\gamma'$ we compute its center $j$ by using its associated array $C_{\gamma'}$ (Step 4 – Alg. 29). This is straightforward if compared to FF-CLEANUP where we first need to update the center arrays. We add this triple to $P^*(x, y)$ and to $S$, which contains the set of triples that need to be extended. We also add $\gamma'$ to $P_j^*(x, y)$ (Steps 10 and 21 – Alg. 29). We update $dict_j$ to keep track of the locations of the triple in the global structures. A similar sequence of steps takes place when all the extracted paths go through $v$ (Steps 18

**Algorithm 29** FF-NEW-PATHS($S', P_{xy}$)

```
1: $S \leftarrow \emptyset$; let $i$ be the min-weight level associated with $DL(x, y)$
2: **if** $P^*(x, y)$ increased min-weight after cleanup **then**
3:    **for** each $\gamma' \in S$ with-key $[wt, 0]$ **do**
4:       let $\gamma' = ((xa', b'y), wt, count')$ and $j = \text{argmin}_j(C_{\gamma'}[j] \neq 0)$
5:       **if** $\gamma'$ is not in $P^*(x, y)$ **then**
6:          add $\gamma'$ in $P^*(x, y)$ and $S$; add $x$ to $L^*(a', y)$ and $y$ to $R^*(x, b')$
7:          add $b'$ to $RN(x, y, wt)$; place a double link between $\gamma'$ and $DL(x, y, wt)$
8:       **else if** $\gamma'$ is in $P(x, y)$ and $P^*(x, y)$ with different counts **then**
9:          replace the count of $\gamma'$ in $P^*(x, y)$ with $count'$ and add $\gamma'$ to $S$
10:      add $\gamma'$ to $P^*_j(x, y)$ and $dict_j$
11:      add $x$ to $L^*_j(a', y)$ and $y$ to $R^*_j(x, b')$
12:      add $x$ to $LC^*_j(a', y)$ ($y$ to $RC^*_j(x, b')$) if $x$ ($y$) is a level $i$ center
13:      add $\gamma'$ in $S$
14:   **for** each $b' \in RN(x, y, wt)$ **do**
15:      **if** $\exists\, h < i : L^*_h(x, b') \neq \emptyset$ **then**
16:         add any $\gamma'$ of the form $(x\times, b'y)$ and weight $wt$ in $P^*(x, y)$ with $\beta(\gamma') = 1$
            to $S$
17: **else**
18:   **for** each $\gamma' \in S'$ containing a path through $v$ **do**
19:      let $\gamma' = ((xa', b'y), wt, count')$ and $k$ the current level
20:      add $\gamma'$ with paths($\gamma', v$) to $P^*(x, y)$, and $[\gamma', C_{\gamma'}]$ to $S$
21:      add $\gamma'$ to $P^*_k(x, y)$ and $dict_k$, $x$ to $L^*_k(a', y)$ and $y$ to $R^*_k(x, b')$
22:      add $x$ to $LC^*_k(a', y)$ ($y$ to $RC^*_k(x, b')$) if $x$ ($y$) is a level $k$ center
23: return $S$
```

to 22 – Alg. 29). The only difference is that the local data structures to be updated are only the $\Gamma_k$ data stuctures (Steps 21 and 22 – Alg. 29).

A crucial difference from FULLY-FIXUP and this algorithm is the way we collect the set $S$ of triples to be extended. Here we require the new HE data structures $RN$ and $LN$ (see Section 6.1.1) because of PEP instances (see Section 6.1.2). Let $i$ be the min-weight level associated with $DL(x, y)$. For each node $b \in RN(x, y, wt)$ we check if $L^*_h(x, b)$ contains at least one extension, for every $h < i$ (Steps 14 to 16 – Alg. 29). In fact we need to discover all tuples with $\beta = 1$ that are inside a PEP instance. In this instance, the triples restored as STs may or may not be extended. We cannot afford to look at all of them, thus our solution should check only the triples with an available extension. Moreover, all the extendable triples with with $\beta = 1$ have extension only in levels younger than the level where they last appear as STs. Thus, we check for extensions only in the levels $h < i$.

Using the HE sets, is the key to avoid an otherwise long search of all the valid extensions for the set of examined triples with $\beta = 1$. In particular, without the HE sets, the algorithm could waste time by searching for extensions that are not even in the tuple-system. Correctness of this method is proven in section 6.3. After the algorithm collects the set $S$ of triples that can be extended, FF-FIXUP calls FF-FIXUP-$\ell$-extend (Alg. 30).

Here we describe the details of algorithm 30. Its goal is to generate LHTs for the current graph $G$ by extending HTs. Let $h$ be the *center of* $S_b$ defined as the most recent center among all the triples in $S_b$, and let $j$ be the level associated to the first weight $wt'$ larger than $wt$ in $DL(x, y)$. The extension phase for triples is different from FULLY-FIXUP: in fact, the set of triples $S_b$ could contain only triples with $\beta(\gamma) = 1$. In FULLY-FIXUP, the corresponding set $S_b$ contains only triples with $\beta(\gamma) = 0$. We address two cases:

**(a)** – If $S_b$ contains at least one triple $\gamma$ with $\beta(\gamma) = 0$, we extend $S_b$ using the sets

**Algorithm 30** FF-FIXUP-$\ell$-extend($S$,$[wt, x, y]$)

```
1: for every $b$ such that $(x\times, by) \in S$ do
2:    let $S_b \subseteq S$ be the set of all triples of the form $(x\times, by)$
3:    let $fcount'$ be the sum of all the $counts$ of all triples in $S_b$; let $h$ be the $center(S_b)$
4:    if $\exists\, \gamma \in S_b : \beta(\gamma) = 0$ then
5:       let $j$ be the level associated to the minweight $wt' > wt$ in $DL(x, y)$
6:       for every active level $h \le i < j$ do
7:          for every $x'$ in $L_i^*(x, b)$ do
8:             if $(x'x, by) \notin$ Marked-Tuples then
9:                $wt' \leftarrow wt + \mathbf{w}(x', x)$; $\gamma' \leftarrow ((x'x, by), wt', fcount')$
10:               $C_{\gamma'} \leftarrow$ FF-FIXUP-CENTERS($S_b$); add $\gamma'$ to $H_f$
11:               if a triple $\gamma''$ for $((x'x, by), wt')$ exists in $P(x', y)$ then
12:                  update the count of $\gamma''$ in $P(x', y)$ and $C_{\gamma''} = C_{\gamma''} + C_{\gamma'}$
13:                  add $(x'x, by)$ to Marked-Tuples
14:               else
15:                  add $[\gamma', C_{\gamma'}]$ to $P(x', y)$; add $x'$ to $L(x, by)$ and $y$ to $R(x'x, b)$
16:               set $\beta(\gamma') = 0$; set update-num($\gamma'$)
17:      for every level $i < h$ do
18:         for every $x'$ in $LC_i^*(x, b)$ do
19:            execute steps 8 to 16
20:   else
21:      let $j$ be the level associated to the minweight $wt$ in $DL(x, y)$
22:      for every level $i < j$ do
23:         for every $x'$ in $LC_i^*(x, b)$ do
24:            execute steps 8 to 16
```

**Algorithm 31** FF-FIXUP-CENTERS($S_b$)

```
1: let $C' = \sum_{\gamma \in S_b} C_\gamma$ be the sum (level by level) of the new paths that are found shortest
2: let $j$ be $\text{argmax}_j(C'[j] \neq 0)$, and $k = center(x')$
3: if $k < j$ then
4:    for all the levels $i < k$ we set $C_{\gamma'}[i] = C'[i]$
5:    for the level $k$ we set $C_{\gamma'}[k] = \sum_{q=k}^{r-1} C'[q]$
6:    for all the levels $m > k$ we set $C_{\gamma'}[m] = 0$
7: else
8:    $C_{\gamma'} = C'$
9: return $C_{\gamma'}$
```

$L_i^*$ and $R_i^*$ with $h \leq i < j$ (Steps 7 to 16 – Alg.30). In fact, the set $S_b$ contains at least one new path that was not extended in the previous iterations when $wt$ was the shortest distance from $x$ to $y$ (because of the $\beta(\gamma) = 0$ triple). The LST generated in this way remains centered in level $h$. Moreover we extend $S_b$ also using the sets $LC_i^*$ and $RC_i^*$ with $i < h$ (Steps 17 to 19 – Alg.30). This ensures that every LST generated in a level $i$ lower than $h$ is centered in $i$ thanks to the extension node itself. This technique guarantees that each LHT generated by Algorithm 30 is an LST centered in a unique level.

(**b**) – In the case when there is no triple $\gamma$ in $S_b$ with $\beta(\gamma) = 0$, then there is at least one extension to perform for $S_b$ and it must be in some level younger than the level where $wt$ stopped to be the shortest distance from $x$ to $y$ (this follows from the use of the HE sets in Alg. 29). To perform these extensions we set $j$ as the level associated with the min-weight element in $DL(x, y)$, and we extend $S_b$ using the sets $LC_i^*$ and $RC_i^*$ with $i < j$ (Steps 21 to 24 – Alg.30). Again, every LHT generated is an LST centered in a unique level. Finally, every generated LHT is added to $P$ and $H_f$ and we update global $L$ and $R$ structures.

**Observation 14.** *Every LHT generated by algorithm* FF-FIXUP *is an LST centered in a unique level graph.*

*Proof.* As described in (a) and (b) above, every LHT is generated using two triples which are shortest in the same level graph $\Gamma_i$. Moreover, since at least one of them must be centered in level $i$, the resulting LHT is an LST centered in level $i$. □

The last novelty in the algorithm is updating center arrays (Alg. 31 called at step 10 – Alg. 30) in a similar way of FF-CLEANUP: Algorithm 31 identifies the oldest level $j$ related to the triples contained in $S_b$ (Step 2 – Alg. 31). If $j > k$ then we reshape the distribution for $\gamma'$ similarly to FF-CLEANUP (Steps 4 to 6 – Alg. 31). Otherwise $\gamma'$ is completely contained in level $k$ and no reshaping is required (Step 8 – Alg. 31).

### 6.2.3 Correctness of FFD

For the correctness, we assume that all the global and local data structures are correct before the update, and we will show the correctness of them after the update.

**Correctness of Cleanup** The correctness of FF-CLEANUP is established in Lemma 34. We will prove that all paths containing the updated vertex $v$ are removed from the tuple-system. Moreover, the center of each triple is restored, if necessary, to the level containing the most recently updated node on any path in this triple. Note that (as in [DI04, NPR14b]) at the end of the cleanup phase, the global structures $P$ and $P^*$ may not have all the LHTs in $G \setminus \{v\}$.

**Lemma 34.** *At the end of the cleanup phase triggered by an update on a vertex $v$, every LHP that goes through $v$ is removed from the global structures. Moreover, in each level graph $\Gamma_i$, each SP that goes through $v$ is removed from $P_i^*$. For each level $i$, the local structures $L_i^*$, $R_i^*$, $RC_i^*$ and $LC_i^*$ contain the correct extensions; the global structures $L$ and $R$ contain the correct extensions, for each $r$-tuple and $\ell$-tuple respectively, and the structures $RN$ and $LN$ contain only nodes associated with tuples in $P$. The $DL$ structure only contains historical distances represented by at least one path in the updated graph. Finally, every triple in $P$ and $P^*$ has the correct updated center for the graph $G \setminus \{v\}$.*

*Proof.* The lemma is established with the following loop invariant.
**Loop Invariant:** At the start of each iteration of the while loop in Step 3 of Algorithm 24, assume that the first triple to be extracted from $H_c$ and processed has min-key $= [wt, x, y]$. Then the following properties hold about the tuple-system and $H_c$.

1. For any $a, b \in V$, if $G$ contains $c_{ab}$ LHPs of weight $wt$ of the form $(xa, by)$ passing through $v$, then $H_c$ contains a triple $\gamma = ((xa, by), wt, c_{ab})$ with key

$[wt, x, y]$ already processed: the $c_{ab}$ LHPs through $v$ are not present in the tuple-system.

2. Let $[\hat{wt}, \hat{x}, \hat{y}]$ be the last key extracted from $H_c$ and processed before $[wt, x, y]$. For any key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, let $G$ contain $c > 0$ number of LHPs of weight $wt_1$ of the form $(x_1\times, b_1y_1)$. Further, let $c_v$ (resp. $c_{\bar{v}}$) denote the number of such LHPs that pass through $v$ (resp. do not pass through $v$). Here $c_v + c_{\bar{v}} = c$. For every extension $x' \in L(x_1, b_1y_1)$, let $wt' = wt_1 + \mathbf{w}(x', x_1)$ be the weight of the extended triple $(x'x_1, b_1y_1)$. Then, (the following assertions are similar for $y' \in R(x_1a_1, y_1)$)

   **Global Data Structures:**

   (a) if $c > c_v$ there is a triple in $P(x', y_1)$ of the form $(x'x_1, b_1y_1)$ and weight $wt'$ representing $c - c_v$ LHPs. Moreover, its center is updated according to the last update on any path represented by the triple. If $c = c_v$ there is no such triple in $P(x', y_1)$.

   (b) If a triple of the form $(x'x_1, b_1y_1)$ and weight $wt'$ is present as an HT in $P^*(x', y_1)$, then it represents the exact same number of LHPs $c - c_v$ of the corresponding triple in $P(x', y_1)$. This is exactly the number of HPs of the form $(x'x_1, b_1y_1)$ and weight $wt'$ in $G \setminus \{v\}$.

   (c) $x' \in L(x_1, b_1y_1)$, $y_1 \in R(x'x_1, b_1)$, and $(x'x_1, b_1y_1) \in$ Marked-Tuples iff $c_{\bar{v}} > 0$.

   (d) A triple corresponding to $(x'x_1, b_1y_1)$ with weight $wt'$ and counts $c_v$ is in $H_c$. A similar assertion holds for $y' \in R(x_1a_1, y_1)$.

   (e) The structure $RN(x', y_1, wt')$ contains a node $b$ iff at least one path of the form $(x'\times, by_1)$ and weight $wt'$ is still represented by a triple in $P(x', y_1)$. A similar assertion holds for a node $a$ in $LN(x', y_1, wt')$.

(f) If there is no HT of the form $(x'x, b_1y_1)$ and weight $wt'$ in $P^*(x', y_1)$ then the entry $DL(x', y_1)$ with weight $wt'$ does not exists.

**Local Data Structures:** for each level $j$, let $c_j$ be the number of LSPs of the form $(x'x_1, b_1y_1)$ and weight $wt'$ centered in $\Gamma_j$ and let $c_j(v)$ be the ones that go through $v$. Thus $c = \sum_j c_j$ and $c_v = \sum_j c_j(v)$. Then,

(g) the value of $C_\gamma[j]$, where $\gamma$ is the triple of the form $(x'x_1, b_1y_1)$ and weight $wt'$ in $P(x', y_1)$, is $c_j - c_j(v)$.

(h) If a triple $\gamma$ of the form $(x'x_1, b_1y_1)$ and weight $wt'$ is present as an HT in $P^*$, then $P^*_j(x', y_1)$ represents only $c_j - c_j(v)$ paths. If $c_j - c_j(v) = 0$ then the link to $\gamma$ is removed from $dict_j$. Moreover, $x' \in L^*_j(x_1, y_1)$ (respectively $LC^*_j(x_1, y_1)$ if $x'$ is centered in $\Gamma_j$) iff $x'$ is part of a shortest path of the form $(x'x_1, \times y_1)$ centered in $\Gamma_j$. A similar statement holds for $y_1 \in R^*_j(x', b_1)$ (respectively $RC^*_j(x', b_1)$ if $y_1$ is centered in $\Gamma_j$).

3. For any key $[wt_2, x_2, y_2] \geq [wt, x, y]$, let $G$ contain $c > 0$ LHPs of weight $wt_2$ of the form $(x_2a_2, b_2y_2)$. Further, let $c_v$ (resp. $c_{\bar{v}}$) denote the number of such LHPs that pass through $v$ (resp. do not pass through $v$). Here $c_v + c_{\bar{v}} = c$. Then the tuple $(x_2a_2, b_2y_2) \in$ Marked-Tuples, iff $c_{\bar{v}} > 0$ and a triple for $(x_2a_2, b_2y_2)$ is present in $H_c$

**Initialization:** We start by showing that the invariants hold before the first loop iteration. The min-key triple in $H_c$ has key $[0, v, v]$. Invariant assertion 1 holds since we inserted into $H_c$ the trivial triple of weight 0 corresponding to the vertex $v$ and that is the only triple of such key. Moreover, since we do not represent trivial paths containing the single vertex, no counts need to be decremented. Since we assume positive edge weights, there are no LHPs in $G$ of weight less than zero. Thus all the points of invariant assertion 2 hold trivially. Invariant assertion 3 holds since

$H_c$ does not contain any triple of weight $> 0$ and we initialized Marked-Tuples to empty.

**Maintenance:** Assume that the invariants are true before an iteration $k$ of the loop. We prove that the invariant assertions remain true before the next iteration $k+1$. Let the min-key triple at the beginning of the $k$-th iteration be $[wt_k, x_k, y_k]$. By invariant assertion 1, we know that for any $a_i, b_j$, if there exists a triple $\gamma$ of the form $(x_k a_i, b_j y_k)$ of weight $wt_k$ representing *count* paths containing $v$, then it is present in $H_c$. Now consider the set of triples with key $[wt_k, x_k, y_k]$ which we extract in the set $S$ (Step 4, Algorithm 24). We consider left-extensions of triples in $S$; symmetric arguments apply for right-extensions. Consider for a particular $b$ the set $S_b \subseteq S$ of triples of the form $(x_k -, b y_k)$, and let $fcount'$ denote the sum of the counts of the paths represented by triples in $S_b$. Let $x' \in L(x_k, b y_k)$ be a left extension; our goal is to generate the triple $\gamma'$ of the form $(x' x_k, b y_k)$ with count $fcount'$ and weight $wt' = wt_k + \mathbf{w}(x', x_k)$, and an associated vector $C(\gamma')$ that specifies the distribution of paths represented by $\gamma'$ level by level. These paths will be then removed by the algorithm. However, we generate such triple only if it has not been generated by a right-extension of another set of paths by checking the Marked-Tuples structure: we observe that the paths of the form $(x' x_k, b y_k)$ can be generated by right extending to $y_k$ the set of triples of the form $(x' x_k, \times b)$. Without loss of generality assume that the triples of the form $(x' x_k, \times b)$ have a key which is greater than the key $[wt_k, x_k, y_k]$. Thus, at the beginning of the $k$-th iteration, by invariant assertion 3, we know that $(x' x_k, b y_k) \notin$ Marked-Tuples. Step 5, Alg. 24 creates a triple $\gamma'$ of the form $(x' x_k, b y_k)$ of weight $wt'$ and $fcount'$.

The set of triples in $S_b$ can have different centers and we are going to remove (level by level) paths represented by $\gamma'$. To perform this task we consider the vector $C_{\gamma''}$: it contains the full distribution of the triple $\gamma'' \in P(x' y_k)$ of the form $((x' x_k, b y_k), wt')$ and indicates the oldest level $j$ in which $\gamma''$ was generated for the first time. This level is exactly $\text{argmax}_j(C_{\gamma''}[j] \neq 0)$ and it is identified in Step 3 - Alg. 25. All the paths represented in $S_b$ that are centered in some level $m$ older or equal to $j$ were extended for the first time in level $j$ to generate $\gamma''$. Moreover, each path centered in a level $i$ younger than $j$ was extended in level $i$ itself. Thus, we can compute a new center vector $C_{\gamma'}$ (according to the distribution in $C_{\gamma''}$) of the paths containing $v$ that we want to delete at each active level, as in steps 5 to 8 - Alg. 26. In step 9 - Alg. 26 the vector $C_{\gamma''}$ is updated: the paths are removed level by level according to the new distribution. This establishes invariant assertion $2g$.

The triple $\gamma'$ is immediately added to $H_c$ with $C_{\gamma'}$ for further extensions (Step. 7 - Alg. 25). This establishes invariant assertions $2d$. Thus we reduce the counts of $\gamma'$ in $P(x', y_k)$ by $fcount$ (Step. 8 - Alg.25) and we set the new center for the remaining tuple $\gamma''$ in $P(x', y_k)$ establishing invariant assertion $2a$. Steps 10 to 13 - Alg. 25 check if there is any path of the form $(x-, by)$ that can use $x'$ as an extension. In this case we add $\gamma'$ to the Marked-Tuples. If not, we safely remove the left and right extension ($x'$ and $y$) from the tuple-system. This establishes invariant assertion $2c$. If $\gamma'$ is an HT in $P^*(x', y_k)$, we decrement its count (Step. 19 - Alg.25) establishing invariant assertion $2b$. In steps 14 to 17 - Alg. 25, we use the double links between $b \in RN(x', y_k, wt')$ and tuples to efficiently check if there are other triples linked to $b$; if not we remove $b$ from $RN(x', y_k, wt')$ establishing invariant assertion $2e$. Using a similar double link method with the structure $DL(x', y_k)$, we establish invariant assertion $2f$ after step 21 - Alg. 25.

To operate in the local data structures we require $\gamma'$ to be an HT in $P^*(x', y_k)$. Using the previously created vector $C_{\gamma'}$, we reduce the count associated with $\gamma' \in 1P^*_i(x', y_k)$ for each level $i$ (Step 23 - Alg. 25). After the above step, if there are no paths left in $P^*_i(x', y_1)$ then there are no STs of the form $(x'x_1, by_1)$ centered in level $i$. In this case we remove the extension $x'$ and $y_k$ from the local structures of level $i$. This is done in steps 24 to 34 -Alg. 25: in case $x'$ is not centered in level $i$, then any path in $\gamma'$ centered in level $i$ is generated by a node centered in level $i$ located between $x_k$ and $y_k$. Thus if any SP from $x'$ to $y_k$ (that uses $(x', x_k)$ as a first edge) remains in in $\Gamma_i$, it must be also counted in $P_i^*(x_k, y_k)$. Thus, we remove $x'$ from $L_i^*(x_k, y_k)$ only if $P_i^*(x_k, y_k)$ is empty. In the case $x'$ is centered in level $i$ and $P_i^*(x_k, y_k)$ is empty, $x'$ could still be the extension of other paths from $x_k$ to $y_k$ centered in levels older than $i$. The algorithm checks them all and if they do not exist in older levels we can safely remove $x'$ from $LC_i^*(x_k, y_k)$ (Step 32, Alg. 25). A similar argument holds for the right extension $y_1$. This establishes invariant assertion $2h$ and completes claim 2.

When any triple is generated by a left extension (or symmetrically right extension), it is inserted into $H_c$ as well as into Marked-Tuples. This establishes invariant assertion 3 at the beginning of the $(k+1)$-th iteration.

Finally, to see that invariant assertion 1 holds at the beginning of the $(k+1)$-th iteration, let the min-key at the $(k+1)$-th iteration be $[wt_{k+1}, x_{k+1}, y_{k+1}]$. Observe that triples with weight $wt_{k+1}$ starting with $x_{k+1}$ and ending in $y_{k+1}$ can be created either by left extending or right extending the triples of smaller weight. And since for each of iteration $\leq k$, invariant assertion 2 holds for any extension, we conclude that invariant assertion 1 holds at the beginning of the $(k+1)$-th iteration. This concludes our maintenance step.

**Termination:** The condition to exit the loop is $H_c = \emptyset$. Because invariant assertion 1 maintains in $H_c$ all the triples already processed, then $H_c = \emptyset$ implies that there are no other triples to extend in the graph $G$ that contain the updated node $v$. Moreover, because of invariant assertion 1, every triple containing the node $v$ inserted into $H_c$ has been correctly decremented from the tuple-system. Remaining triples have the correct update center because of invariant $2a$. Finally, for invariant assertions $2g$ and $2h$, the structures $L_i^*, LC_i^*, R_i^*, RC_i^*$ are correctly maintained for every active level $i$ and the paths are surgically removed only from the levels in which they are centered. This completes the proof. □ □

**Correctness of Fixup** For the fixup phase, we need to show that the triples generated by our algorithm are sufficient to maintain all the ST and LST in the current graph $G$. As in our basic algorithm, we first show in the following lemma that FF-FIXUP computes all the correct distances for each pair of nodes in the updated graph. Finally, we show that data structures and counts are correctly maintained at the end of the algorithm (Lemma 36).

**Lemma 35.** *For every pair of nodes $(x, y)$, let $\gamma = ((xa, by), wt, count)$ be one of the min-weight triples from $x$ to $y$ extracted from $H_f$ during* FF-FIXUP*. Then $wt$ is the shortest path distance from $x$ to $y$ in $G$ after the update.*

*Proof.* Suppose that the lemma is violated. Thus, there will be an extraction from $H_f$ during FF-FIXUP such that the set of extracted triples $S'$, of weight $\hat{wt}$ is not shortest in $G$ after the update. Consider the earliest of these events when $S'$ is extracted from $H_f$. Since $S'$ is not a set of STs from $x$ to $y$, there is at least one shorter tuple from $x$ and $y$ in the updated graph. Let $\gamma' = ((xa', b'y), wt, count)$ be this triple that represents at least one shortest path from $x$ to $y$, with $wt < \hat{wt}$. Since $S'$ is extracted from $H_f$ before any other triple from $x$ to $y$, $\gamma'$ cannot be in $H_f$ at any time during FF-FIXUP. Hence, it is also not present in $P(x, y)$ as an LST at the beginning of the algorithm, otherwise it (or another triple with the same weight) would be placed in $H_f$ by step 2 - Alg. 27. Moreover, if $\gamma'$ is a single edge (trivial triple), then it was already an LST in $G$ present in $P(x, y)$ before the update, and it is added to $H_f$ by step 10 - Alg. 28; moreover since all the edges incident to $v$ are added to $H_f$ during steps 2 to 7 of Alg. 28, then $\gamma'$ must represent SPs of at least two edges. We define $left(\gamma')$ as the set of LSTs of the form $((xa', c_ib'), wt - \mathbf{w}(b', y), count_{c_i})$ that represent all the LSPs in the left tuple $((xa', b'), wt - \mathbf{w}(b', y))$; similarly we define $right(\gamma')$ as the set of LSTs of the form

$((a'd_j, b'y), wt - \mathbf{w}(x, a'), count_{d_j})$ that represent all the LSPs in the right tuple $((a', b'y), wt - \mathbf{w}(x, a'))$.

Observe that since $\gamma'$ is an ST, all the LSTs in $left(\gamma')$ and $right(\gamma')$ are also STs. A triple in $left(\gamma')$ and a triple in $right(\gamma')$ cannot be present in $P^*$ together at the beginning of FF-FIXUP. In fact, if at least one triple from both sets is present in $P^*$ at the beginning of FF-FIXUP, then the last one inserted during the fixup phase triggered during the previous update, would have generated an LST of the form $((xa', b'y), wt)$ automatically inserted, and thus present, in $P$ at the beginning of the current fixup phase (a contradiction). Thus either there is no triple represented by $left(\gamma')$ in $P^*$, or there is no triple represented by $right(\gamma')$ in $P^*$.

Assume w.l.o.g. that the set of triples in $right(\gamma')$ is placed into $P^*$ after $left(\gamma')$ by FF-FIXUP. Since edge weights are positive, $wt - \mathbf{w}(x, a') < wt < \hat{wt}$, and because all the extractions before $\gamma$ were correct, then the triples in $right(\gamma')$ were correctly extracted from $H_f$ and placed in $P^*$ before the wrong extraction of $S'$. Let $i$ be the level in which $left(\gamma')$ is centered, and let $j$ be the level in which $right(\gamma')$ is centered. By the assumptions, all the triples in $left(\gamma')$ are in $P^*$ and we need to distinguish 3 cases:

1. if $j = i$, then FF-FIXUP generates the tuple $((xa', b'y), wt)$ in the same level and place it in $P$ and $H_f$.

2. if $i > j$, the algorithms FF-FIXUP extends the set $right(\gamma')$ to all nodes in $L_i^*(a', b')$ for every $i \geq j$ (see Steps 7 to 16 - Alg. 30). Thus, since $left(\gamma')$ is centered in some level $i > j$, the node $x$ is a valid extension in $L_i^*(a', b')$, making the generated $\gamma'$ an LST in $\Gamma_j$ that will be placed in $P(x, y)$ and also into $H_f$ (during Step 10 - Alg. 30).

3. if $j > i$, then $x$ was inserted in a level younger than $i$. In fact, all the paths from $a'$ to $b'$ must be the same in $right(\gamma')$ and $left(\gamma')$ otherwise the center of

$right(\gamma')$ should be $i$. Hence, the only case when $j > i$ is when the last update on $left(\gamma')$ is on the node $x$ in a level $i$ younger than $j$. Thus $x \in LC_i^*(a', b')$. But FF-FIXUP extends $right(\gamma')$ to all nodes in $LC_i^*(a', b')$ for every $i < j$, placing the generated LST $\gamma'$ in $P(x, y)$ and also into $H_f$ (see Steps 17 to 19 - Alg. 30).

Thus the algorithm would generate the tuple $((xa', b'y), wt)$ (as a left extension) and place it in $P$ and $H_f$ (because all the triples in $left(\gamma')$ are already in $P^*$). Therefore, in all cases, a tuple $((xa', b'y), wt)$ should have been extracted from $H_f$ before any triple in $S'$. A contradiction. □

**Lemma 36.** *After the execution of* FF-FIXUP*, for any* $(x, y) \in V$*, the sets* $P^*(x, y)$ *(*$P(x, y)$*) contains all the SPs (LSPs) from* $x$ *to* $y$ *in the updated graph. Also, the global structures* $L, R$ *and the local structures* $P_i^*, L_i^*, R_i^*, LC_i^*, RC_i^*$ *and* $dict_i$ *for each level* $i$ *are correctly maintained. The structures* $RN$ *and* $LN$ *are updated according to the newly identified tuples. The* $DL$ *structure contains the updated distance for each pair of nodes in the current graph. Finally, the center of each new triple is updated.*

*Proof.* We prove the lemma statement by showing the following loop invariant. Let $G'$ be the graph after the update.

**Loop Invariant:** At the start of each iteration of the while loop in Step 3 of FF-FIXUP, assume that the first triple in $H_f$ to be extracted and processed has min-key $= [wt, x, y]$. Then the following properties hold about the tuple-system and $H_f$.

1. For any $a, b \in V$, if $G'$ contains $c_{ab}$ SPs of form $(xa, by)$ and weight $wt$, then $H_f$ contains a triple of form $(xa, by)$ and weight $wt$ to be extracted and processed. Further, a triple $\gamma = ((xa, by), wt, c_{ab})$ is present in $P(x, y)$.

2. Let $[\hat{wt}, \hat{x}, \hat{y}]$ be the last key extracted from $H_f$ and processed before $[wt, x, y]$. For any key $[wt_1, x_1, y_1] \leq [\hat{wt}, \hat{x}, \hat{y}]$, let $G'$ contain $c > 0$ number of LHPs of

weight $wt_1$ of the form $(x_1a_1, b_1y_1)$. Further, let $c_{new}$ (resp. $c_{old}$) denote the number of these LHPs that are *new* (resp. not *new*). Here $c_{new} + c_{old} = c$. If $c_{new} > 0$ then,

**Global Data Structures:**

(a) there is an LHT $\gamma$ in $P(x_1, y_1)$ of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ that represents $c$ LHPs, with an updated center defined by the last update on any of the paths represented by the LHT.

(b) If a triple of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ is present as an HT in $P^*$, then it represents the exact same count of $c$ HPs of its corresponding triple in $P$. This is exactly the number of HPs of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ in $G'$. Its control bit $\beta$ is set to 1.

(c) $x_1 \in L(a_1, b_1y_1)$, $y_1 \in R(x_1a_1, b_1)$. Further, $(x_1a_1, b_1y_1) \in$ Marked-Tuples iff $c_{old} > 0$.

(d) If $\beta(\gamma) = 0$ or $\beta(\gamma) = 1$ and there is an extension $x' \in L^*_j(x_1, y_1)$ that generates a centered LST in a level $j$, an LHT corresponding to $(x'x_1, b_1y_1)$ with weight $wt' = wt_1 + \mathbf{w}(x', x_1) \geq wt$ and counts equal to the sum of new paths represented by its constituents, is in $H_f$ and $P$. A similar assertion holds for an extension $y' \in R^*_j(x_1, y_1)$.

(e) The structure $RN(x_1, y_1, wt_1)$ contains a node $b$ iff at least one path of the form $(x_1\times, by_1)$ and weight $wt_1$ is represented by a triple in $P(x_1, y_1)$. A similar assertion holds for a node $a$ in $LN(x_1, y_1, wt_1)$.

(f) The entry $DL(x_1, y_1)$ with weight $wt_1$ is updated to the current level.

**Local Data Structures:** for each level $j$, let $c_j$ be the number of SPs of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ centered in $\Gamma_j$ and let $c_j(n)$ be the new ones discovered bythe algorithm. Thus $c = \sum_j c_j$ and $c_{new} = \sum_j c_j(n)$. Then,

(g) the value of $C_\gamma[j]$, where $\gamma$ is the triple of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ in $P(x_1, y_1)$, is $c_j$.

(h) If a triple $\gamma$ of the form $(x_1a_1, b_1y_1)$ and weight $wt_1$ is present as an HT in $P^*$, then $P_j^*(x1, y1)$ represents $c_j$ paths. A link to $\gamma$ in $P$ is present in $dict_j$. Moreover, $x_1 \in L_j^*(a_1, y_1)$ (respectively $LC_j^*(a_1, y_1)$ if $x_1$ is centered in $\Gamma_j$). A similar statement holds for $y_1 \in R_j^*(x_1, b_1)$ (respectively $RC_j^*(x_1, b_1)$ if $y_1$ is centered in $\Gamma_j$).

3. For any key $[wt_2, x_2, y_2] \geq [wt, x, y]$, let $G'$ contain $c > 0$ number of LHPs of weight $wt_2$ of the form $(x_2a_2, b_2y_2)$. Further, let $c_{new}$ (resp. $c_{old}$) denote the number of such LHPs that are *new* (resp. not *new*). Here $c_{new} + c_{old} = c$. Then the tuple $(x_2a_2, b_2y_2) \in$ Marked-Tuples, iff $c_{old} > 0$ and $c_{new}$ paths have been added to $H_f$ by some earlier iteration of the while loop.

Initialization and Maintenance for the invariant assertions above are similar to the proof of Lemma 34.

**Termination:** The condition to exit the loop is $H_f = \emptyset$. Because invariant assertion 1 maintains in $H_f$ the first triple to be extracted and processed, then $H_f = \emptyset$ implies that there are no triples, formed by a valid left or right extension, that contain *new* SPs or LSPs, that need to be added or restored in the graph $G$. Moreover, because of invariant assertions $2a$ and $2b$, every triple containing the node $v$, extracted and processed before $H_f = \emptyset$, has been added or restored with its correct count in the tuple-system. Finally, for invariant assertions $2c$ and $2h$, the sets $L, R$ and $L^*, LC^*, R^*, RC^*$ for each level, are correctly maintained. This completes the proof of the loop invariant.

By Lemma 35, all the SP distances in $G'$ are placed in $H_f$ and processed by the algorithm. Hence, after Algorithm 27 is executed, every SP in $G'$ is in its corresponding $P^*$ by the invariant of Lemma 36. Since every LST of the form $(xa, by)$ in $G'$ is formed by a left extension of a set of STs of the form $(a\times, by)$ (Step 7 -

Algorithm 27), or a right extension of a set of the form $(xa, \times b)$ (analogous steps for right extensions), and all the STs are correctly maintained and extendend (by the invariant of Lemma 36), then all the LSTs are correctly maintained at the end of FF-FIXUP. This completes the proof of the Lemma. □

## 6.3 Complexity of FFD Algorithm

In this section we will prove the complexity bounds of our FFD algorithm. The correctness is addressed in Section 6.2.3. The complexity analysis is similar to that for our basic algorithm. We highlight the following new elements:

1. Every triple created by FF-FIXUP is an LST in the level graph (PDG) in which is centered (see Observation 14), and by the increase-only properties of level graphs, it will continue to be an LST in that level graph until it is removed. In contrast, our basic algorithm can create LHTs by combining HTs not centered in any PDG. This results in an additional $\Theta(\log n)$ factor in the amortized bound there.

2. We can bound the number of LHTs that contain a given vertex $u$ as $O(z' \cdot \nu^{*2})$, where $z'$ is the number of active level graphs that contain vertex $u$ and tuples passing through $u$ (by Corollary 5). Given our level tuple-system, $z'$ is clearly $O(\log n)$. In our basic result, this bound is $(z + z'^2)$ where $z$ is the number of active PDGs, and $z'$ is the number of PDGs that contain $v$.

3. We can show that the number of accesses to $RN$ and $LN$, outside of the newly created tuples, is worst-case $O(n \cdot \nu^*)$ per call to FF-UPDATE. The overhead given by the level data structures is $O(\log n)$ for each access (see Lemma 41). These structures are not used in our basic algorithm.

All the algorithms referenced in the following lemmas are described in Section 6.2.

**Lemma 37.** *Let $G$ be a graph after a sequence of calls to* FF-UPDATE. *Let $z$ be the number of active level graphs (PDGs), and let $z' \leq z$ be the number of level graphs that contain a given vertex $v$. Suppose that every HT in the tuple-system is an ST in some level graphs, and every LHT is an LST in some level graph. If $n$ and $m$ bound the number of vertices and edges, respectively, in any of these graphs, and if $\nu^*$ bounds the maximum number of distinct edges that lie on shortest paths through any given vertex in any of the these graphs, then:*

1. *The number of LHTs in $G$'s tuple-system is at most $O(z \cdot m \cdot \nu^*)$.*
2. *The number of LHTs that contain a vertex $v$ in $G$ is $O(z' \cdot \nu^{*2})$.*

*Proof.* For part 1, we bound the number of LHTs $(xa, by)$ (across all weights) that can exist in $G$. The edge $(x, a)$ can be chosen in $m$ ways, and once we fix $(x, a)$, the $r$-tuple $(a, by)$ must be an ST in one of the $\Gamma_j$. Since $(b, y)$ must lie on a shortest path through $a$ centered in a graph $\Gamma_i$, that contains the $r$-tuple $(a, by)$ of shortest weight in $\Gamma_i$, the number of different choices for $(b, y)$ that will then uniquely determine the tuple $(xa, by)$, together with its weight, is $z \cdot \nu^*$. Hence the number of LHTs in $G$'s tuple-system is $O(z \cdot m \cdot \nu^*)$.

For part 2, the number of LHTs that contain $v$ as an internal vertex is simply the number of LSTs across the $z'$ graphs that contains $v$, and this is $O(z' \cdot \nu^{*2})$. We now bound the number of LHTs $(va, by)$. There are $n-1$ choices for the edge $(v, a)$ and $z' \cdot \nu^*$ choices for the $r$-tuple $(a, by)$, hence the total number of such tuples is $O(z' \cdot n \cdot \nu^*)$. The same bound holds for LHTs of the form $(xa, bv)$. Since $\nu^* = \Omega(n)$, the result in part 2 follows. □

**Corollary 5.** *At a given time step, let $B$ be the maximum number of tuples in the tuple-system containing a path through a given vertex in a given level graph. Then, $B = O(\nu^{*2})$.*

**Lemma 38.** *(a) - The cost for an* FF-CLEANUP *call on a node $v$ when $z'$ active levels contain triples through $v$ is $O(z' \cdot \nu^{*2} \cdot \log n)$.*
*(b) - The cost for a real* FF-CLEANUP *call is $O(\nu^{*2} \cdot \log^2 n)$*
*(c) - The cost for a dummy* FF-CLEANUP *call is $O(\nu^{*2} \cdot \log n)$.*

*Proof.* (a) - Since the number of LHTs containing the updated vertex $v$, processed by FF-CLEANUP, is bounded by $B$ at each level (by Corollary 5), the total cost is $O(z' \cdot B \log n)$ where $z'$ is the number of active levels that contain triples through $v$. The worst-case cost for update an array $C_\gamma$ within an FF-CLEANUP phase is $O(z')$ (point [FF–C.1]). Note that a triple can be processed by a constant number of priority queues among $z'$ different active levels. Moreover, for the structures $DL$, $RN$ and $LN$ each triple spends a constant time to be unlinked and eventually to update the structures (point [FF–C.2]). Since, priority queue operations have a $O(\log n)$ cost and the number of triples examined is bounded by $O(z' \cdot \nu^{*2})$, the complexity of FF-CLEANUP that operates on $z'$ active levels requires at most $O(z' \cdot \nu^{*2} \cdot \log n)$.
(b) - Since the active levels are bounded by $z \leq \log 2n$, the cost for a real FF-CLEANUP call is $O(\nu^{*2} \cdot \log^2 n)$ (by part (a)).
(c) - For a dummy cleanup on a vertex $w$, FF-CLEANUP only needs to clean the local data structures in level *center*$(w)$, where $w$ is centered, and in the current level graph. In fact, let $t$ be the current update step; in the dummy cleanup phase, we start with the node $u$ that was updated at time $t-1$ (the most recent update before the current one). The node $u$ received an update in the previous phase, thus it disappeared from all the levels older than *level*$(t-1)$ and, with it, all the LSTs containing $u$ in these levels. Hence, all the triples containing $u$ in the tuple-system must be LSTs in *level*$(t-1)$. We have at most $B$ of them and FF-CLEANUP spends $O(B \cdot \log n)$ (considering the access to the data structures) to remove them. Then, the dummy update reinserts $u$ only in the current graph. The next phase moves on

the node $u'$ updated at time $t-2$. Again, all the tuples containing $u'$ must be LSTs in $level(t-2)$ and eventually the current graph if they were inserted because of the previous dummy update on $u$.

Suppose in fact that there is a tuple $\gamma$ that contains $u'$ in another level (except the current graph). The tuple $\gamma$ cannot be in a level older than $level(t-2)$ because when $u'$ was updated at time $(t-2)$, the cleanup algorithm removed all the tuples containing $u'$ from any level older than $t-2$. Moreover, a tuple containing $u'$ present in a level younger than $level(t-2)$ could appear if and only if it was generated by any update more recent of $t-2$ (in this case only the dummy update on $u$ performed in the current graph). Thus a contradiction.

This argument can be recursively applied to every other node in the sequence: in fact for the node $u''$ updated at time $(t-i)$ all the nodes updated in the interval $[t-i+1, t-1]$ will be already processed by FF-CLEANUP, leaving all the tuples containing $u''$ only in $level(t-i)$ and $t$. It follows that, for a dummy update, $z' = 2$. Thus the cost for a dummy FF-CLEANUP call is $O(\nu^{*2} \cdot \log n)$ (by part (a)). □

**Lemma 39.** *The cost for a dummy* FF-FIXUP *call on a node $v$ is $O(\nu^{*2} \cdot \log n)$.*

*Proof.* Consider a dummy FF-FIXUP applied to a vertex $v$ in $\mathcal{N}$. We only need to bound the cost for accessing the entries in the $P^*(x,y)$ and the cost of re-adding LSTs containing $v$, previously removed by the dummy FF-CLEANUP but still in the current graph after the dummy update. In fact the vertex $v$ is removed by an earlier dummy FF-CLEANUP, and while this removes all the HPs containing the vertex $v$, it does not change any LST centered in any $\Gamma_j$ that does not contain $v$. Hence these other LSTs will be present in the tuple-system with unchanged weight and count, when dummy FF-FIXUP is applied to $v$. Since for any pair $x, y$, the SP distance will not change after the dummy update, the dummy FF-FIXUP will only insert in the set $S$ triples containing the node $v$ for additional extension. Hence, only the LSTs containing $v$ in the current $level(t)$ graph will be processed and added to the tuple-system, and there are at most $B$ of them (by Corollary 5). Thus a dummy FF-FIXUP for any $v$ needs to access $P^*$ for each pair of nodes, and reinsert at most $B$ tuples (containing $v$) in the current graph. Hence the overall complexity for a dummy FF-FIXUP is $O((n^2 + B) \cdot \log n) = O(\nu^{*2} \cdot \log n)$. $\square$

We now address the complexity of a real FF-FIXUP call. We first define the concept of a *triple pair* that will be used in lemma 41 to establish the bound for a real FF-FIXUP call. Finally, we complete our analysis by presenting a proof of Theorem 9.

**Definition 3.** *If $C_\gamma[i] \geq 1$ then $(\gamma, i)$ is a* triple pair *in the tuple-system. If $(\gamma, i)$ is not a triple pair in the tuple-system at the start of step $t$ but is a triple pair after the update at time step $t$, then $(\gamma, i)$ is a* newly created triple pair *at time step $t$.*

**Lemma 40.** *At a given time step, let $D$ be the number of triple pairs in the level tuple-system. Then,*

1. *The value of $D$ is at most $O(m \cdot \nu^* \cdot \log n)$.*

2. *The space used is $O(m \cdot \nu^* \cdot \log n)$.*

*Proof.* 1. Every $C_\gamma[i] \geq 1$ represents a distinct LST in $\Gamma_i$, hence the result follows since the number of levels is $O(\log n)$ and the number of LSTs in a graph is $O(\nu^* \cdot m^*)$.
2. Since every triple is of size $O(1)$, the memory used by our FFD algorithm is dominated by $D$, and result follows from 1. $\square$

**Lemma 41.** *The cost for a real* FF-FIXUP *call is $O(\nu^{*2} \cdot \log^2 n + X \cdot \log n)$ , where $X$ is the number of newly created triple pairs after the update step.*

*Proof.* Recall that, for the structures $DL$, $RN$ and $LN$ each triple spends a constant time to be unlinked and eventually to update the structures (fact[FF–F.2]); moreover, updating an array $C_\gamma$ with the new centers requires only additional

$O(z' \leq 2\log n)$ time (fact[FF–F.1]). Thus, a triple is accessed only a constant number of time during FF-FIXUP with a total cost of $O(\log n)$, and it suffices to establish that the number of existing triples accessed during the call is $O(\nu^{*2} \cdot \log n)$.

There are only $O(n^2)$ accesses to triples to initialize FF-FIXUP since $O(n^2)$ entries in the global $P^*(x, y)$ structures are accessed to populate $H_f$ (a shortest triple for each pair $(x, y)$). This takes $O(n^2 \cdot \log n)$ time after considering the $O(\log n)$ cost per data structure operation. We now address the accesses made in the main loop. We will distinguish two cases and they will be charged to $X$ as follows.

1: $\underline{\beta(\gamma) = 0}$ – Any triple $\gamma$ that is accessed with $\beta(\gamma) = 0$ is an LST at some level $i$ where it is not identified as an ST in $\Gamma_i$. In this case, if the distance for the endpoints of $\gamma$ did not change, $\gamma$ is added as an ST in level $i$, and will never be removed as an ST for level $i$ until it is removed from the tuple system (due to the fact that $\Gamma_i$ is a purely increase-only graph). Since $\gamma$ with $\beta(\gamma) = 0$ is a newly added triple to level $i$, then the pair $(\gamma, i)$ is a newly created triple pair at step $t$. Hence, we can charge $(\gamma, i)$ to $X$ in this call of FF-FIXUP.

2: $\underline{\beta(\gamma) = 1}$ – We now consider triples accessed that have $\beta = 1$. This is the most nontrivial part of our analysis since even though any such triple $\gamma$ must exist with the same count in every level in both $P$ and $P^*$, we may still need to form some extensions since the triple may have been an HT when extension vertices were updated, and hence these extension may not have been performed. Here is where the $LN$ and $RN$ sets are accessed, and we now analyze the cost of these accesses.

Let $j$ be the most recent level in which $\gamma$ was an ST in $G$ and assume we are dealing with left extensions (right extensions are symmetrical). Now that $\gamma$ is restored, the only case in which we need to process it is when there exists a left extension for the $\ell$-tuple of $\gamma$ to a node $x'$ centered in a level $i$ more recent than $j$. In fact, the LST generated by this extension will appear for the first time centered in level $i$, hence the pair $(\gamma, i)$ is a newly created triple pair at step $t$ and we can charge its creation to $X$. We now show how our HE sets efficiently handle this case. FF-FIXUP only processes a restored triple $\gamma$ with $\beta(\gamma) = 1$ when it has at least one centered extension in some active level younger than the level in which $\gamma$ was shortest for the last time (fact[FF–F.3]). We can bound the total computation for these steps as follows: for a given $x$, $RN(x, y, wt)$ contains a node $b$ for every incoming edge to $y$ in one of the SSSP dags (historical and shortest) rooted at $x$. Since we can extend in at most $O(\log n)$ active levels during any update and the size of a single dag is at most $\nu^*$, these steps take time $O(\nu^* \cdot n \log n)$ throughout the entire update computation. □

We can now establish the proof of our main theorem.

**Proof of Theorem 9.** Consider a sequence $\Sigma$ of $r = \Omega(n)$ calls to algorithm FFD. Recall that the data structure is reconstructed after every $2n$ steps, so we can assume $r = \Theta(n)$. These $r$ calls to FFD make $r$ real calls to FF-UPDATE, and also make additional dummy updates. As in our basic algorithm, across the $r$ real updates in $\Sigma$, the algorithm performs $O(r \log n)$ dummy updates. This is because $r/2^k$ real updates are performed at level $k$ during the entire computation, and each such update is accompanied by $2^k - 1$ dummy updates. So, across all real updates there are $O(r)$ dummy updates per level, adding up to $O(r \log n)$ in total, across the $O(\log n)$ levels.

When FF-CLEANUP is called on a vertex $v$ for a dummy update, $z' = 2$ since $v$ can be present only in the most recent current level and the level at which it is centered. (This is because every vertex that was centered at a more recent level than $v$ has already been subjected to a dummy update, and hence all of these vertices are now centered in the current level.) Thus, by Lemma 38, each FF-CLEANUP for a dummy update has cost $O(B \cdot \log n)$. By Lemma 39, a call to FF-FIXUP for a dummy update has cost $O(\nu^{*2} \cdot \log n)$. Thus the total cost is $O((\nu^{*2} \cdot \log n) \cdot r \log n)$ across all dummy updates. Also, the number of tuples accessed by all of the dummy update calls to FF-CLEANUP, and hence the number of tuples removed by all dummy updates, is $O(r \cdot \nu^{*2} \cdot \log n)$.

For the real calls to FF-FIXUP, let $X_i$ be the number of newly added triple pairs in the $i$th real call to FF-FIXUP. Then by Lemma 41, the cost of this $i$th call is $O(\nu^{*2} \cdot \log^2 n + X_i \cdot \log n)$. Let $X = \sum_{i=1}^{r} X_i$. Hence the total cost for the $r$ real calls to FF-FIXUP is $O(r \cdot \nu^{*2} \cdot \log^2 n + X \cdot \log n)$. We now bound $X$ as follows: $X$ is no more than the maximum number of triples that can remain in the system after $\Sigma$ is executed, plus the number of tuples $Y$ removed from the tuple-system. Tuples are removed only in calls to FF-CLEANUP. The total number removed by $r \log n$ dummy calls is $O(r \cdot \log n \cdot \nu^{*2})$ (by Lemma 38). The total number removed by the $r$ real calls is $O(r \cdot \nu^{*2} \cdot \log n)$ (by Lemma 38). Hence $Y = O(r \cdot \nu^{*2} \cdot \log n)$. Clearly the maximum number of triples in the tuple-system is no more than $D$, which counts the number of triple pairs, and we have $D = O(m \cdot \nu^* \cdot \log n) = O(n^2 \cdot \nu^* \cdot \log n)$ (by Lemma 40). Since $r = \Theta(n)$, we have $D = O(r \cdot n \cdot \nu^* \cdot \log n)$, and this is dominated by $Y$ since $\nu^* = \Omega(n)$. Hence the cost of the $r$ calls to FFD is $O(r \cdot \nu^{*2} \cdot \log^2 n)$ (after factoring in the $O(\log n)$ cost per tuple access), and hence the amortized cost of each call to FF-UPDATE is $O(\nu^{*2} \cdot \log^2 n)$.

## 6.4 Discussion

We have presented efficient fully dynamic algorithms for APASP (Chapters 5 and 6). Our algorithms store a superset of the STs and LSTs in the current graph in two priority queues $P^*(x, y)$ and $P(x, y)$ for each vertex pair $x, y$. To generate all shortest paths from all other vertices to $x$, we construct the shortest path in-dag rooted at $x$ in $O(\nu^*)$ time. Then, the shortest paths ending in $x$ can be enumerated in a traversal of this dag, starting from $x$. This query time is output-optimal, and takes time proportional to the number of edges on these paths.

Our basic algorithm (Chapter 5), when specialized to fully dynamic APSP (i.e., for unique SPs), is a variant of the `DI` method [DI04], and it uses a different 'dummy update sequence' from the one in `DI`, with different properties. Our dummy update sequence is inspired by the updates performed on 'level graphs' in [Tho04], though our algorithm is considerably simpler (but is also slower by a logarithmic factor). As noted in Section 5.4.4, our analysis is tailored to the dummy update sequence we use, and a different analysis would be needed if the `DI` update sequence is to be used.

Our faster fully dynamic algorithm (Chapter 6) for APASP runs in amortized $O(\nu^{*2} \cdot \log^2 n)$ time, which is a log factor faster than the basic result. This algorithm is considerably more involved and adapts the `Thorup` method [Tho04] to the APASP problem by maintaining the PDGs explicitly. An additional complexity in this fully dynamic APASP algorithm (beyond that present in `Thorup`) is the need to maintain several different time-stamps for each tuple in the data structures, since the component paths in a tuple may have been updated at different time steps.

### 6.4.1 Open Problems

It would be interesting to investigate if one could stay with our method here of only performing dummy updates and not maintaining the PDGs explicitly, but still obtain the improved bound in our FFD algorithm (and in `Thorup` for unique shortest paths). This would give a reasonably simple fully dynamic APASP algorithm and it would lead to a simpler fully dynamic APSP algorithm with $O(n^2 \cdot \log^2 n)$ amortized time than `Thorup` algorithm. One appealing approach is to only form LSTs in the current graph during the fixup phase (instead of forming all LHTs). It is not difficult to see that this would reduce the cost of the cleanup phase by a logarithmic factor. However, if an HT $\tau$ becomes an ST at a later step, we would then have no guarantee that all of its extensions have been generated. Hence, if this approach is to succeed, a modified algorithm is needed.

Another avenue would be to understand if it is possible to reduce the space complexity by applying a similar data representation as in Chapter 4. Given the fully dynamic nature of the updates, different techniques should by developed to maintain paths as $r$-tuples and $\ell$-tuples in our algorithms. Some non-trivial obstacles to achieve the space reduction are (but not limited to):

- adjusting the marking scheme to handle historical tuples,
- correctly identify which tuples could be generated twice,
- adjust the new tuple-system to handle historical tuples.

# Chapter 7

# Distributed Algorithms

Many real-world networks are very large and unlikely to be represented and processed on a single machine. Computing BC, APASP and even the classic APSP solutions becomes prohibitive in such scenarios. Distributed models of computations aim to solve this problem by considering each node in the network as a standalone machine, that can communicate only with its neighbors. In this case, the distributed network will compute BC, APASP or APSP on the graph represented by the network itself, and not on an arbitrary graph.

In this chapter, we consider problems over distributed networks reviewed in the next section. We adopt the widely used CONGEST model (reviewed in Section 7.1.1) to study our problems. Path problems in graphs have received considerable attention in the CONGEST model, and there has been substantial work on distributed algorithms for undirected graphs in this model. However, as we note in Section 7.2.1, to the best of our knowledge, only few distributed algorithms are known for path problems in directed graphs. There has been a considerable amount of research on designing distributed algorithms on networks, for various properties of the graph represented by the network [Lyn96b, GKP98, PR00, Elk06, Pel00, LPS13a, Nan14], with the goal to minimize the number of rounds used by the distributed algorithm.

In this chapter, we present several distributed algorithms for computing BC in unweighted graphs in near-optimal round complexity (see Section 7.3). These improve over the current state of art algorithms for undirected graphs, while they provide new techniques for directed graphs. Moreover, we present the first $n+O(L)$-round algorithm for APSP in weighted directed acyclic graphs (dags). This result implies an $2n + O(L)$-round algorithm for computing BC in weighted dags (see Section 7.5). Here $L$ is the length of a longest hop path in the dag. Finally, we show a simplified $2n$-round algorithms for transitive closure and strongly connected components, and an interesting property of unilateral graphs (see Sections 7.6).

## 7.1 Distributed Networks

In the basic model, a network of processors is generally modeled by an undirected unweighted graph $G = (V, E)$, with $|V| = n$ nodes and $|E| = m$ edges. When required by the problem, a positive integer weight function $\mathbf{w} : E \rightarrow \{1, \ldots, \text{poly}(n)\}$ is associated to the set of edges; moreover, when specified, the graph can be directed instead of undirected. In contrast to the real edge weights used in the dynamic settings, we use integer weights for distributed network. Each node (processor) in the network has a unique ID in the range $\{1, \ldots, \text{poly}(n)\}$ and infinite computational power. The topological knowledge of each node is limited: a node $v \in V$ only knows the set of its neighbors $\Gamma(v)$. The network activity is determined by *rounds*. In a single round each node can act in different ways:

- A node $v \in V$ can remain silent.
- A node $v \in V$ can send a message of $B$ bits along a subset of its incident edges.
- A node $v \in V$ can send a message of $B$ bits along every incident edge (*broad-*

*cast*).

We will consider only *synchronous*[AFL83] models in this proposal, where all the processors run using the same clock (messages are exchanged in a lockstep in each round). The common measure used to analyze the running time in distributed networks is defined as the *number of rounds* used by the distributed algorithm to complete its task. An important measure used in the analysis of distributed systems is the diameter $D$ of the network. Note that is always possible to compute $n$ in $O(D)$ rounds.

### 7.1.1 The Models

Here, we discuss the main models adopted for distributed networks and we summarize the state of the art regarding shortest paths (SPs) related algorithms and betweenness centrality (BC, see Section 1.1.4) in each model.

**The LOCAL Model:** In the *LOCAL model*, each node in the network can send/receive messages of unbounded size (i.e. $B = \infty$) during a round. For any problem, a simple algorithm can collect the entire topology of the network in a single node (*aggregate the network or aggregation technique*) in $O(D)$ rounds, and then compute the result using the infinite computational power of the node. For a survey on local algorithms refer to [Suo13].

**The CONGEST Model:** The (synchronous) *CONGEST model* reflects a more realistic scenario where the size of the messages is bounded: each node can send on each channel at most $B$ bits, where usually $B = O(\log n)$. If $B \neq O(\log n)$, we call the model CONGEST($B$). Given the limit on the data that can be transferred on each channel, this model takes into account *congestion issues*: when a long queue of different messages is scheduled to be sent by the same node on the same edge. The round complexity in this model has been studied extensively in distributed computing [Pel00]. All the exact results for shortest paths problem in the general CONGEST model are trivial application of the aggregation technique (see Section 7.1.1) or derive from the Bellman-Ford algorithm [Bel58, For56]. If each pair of the $n$ nodes in the network can communicate in each round, the model is called *CONGEST clique* [CHKK$^+$15a]: a pair of nodes can communicate even if is not adjacent in the graph $G$. The CONGEST clique model provides more bandwidth for the communication focusing less on the distance of the nodes in $G$.

### 7.1.2 Directed Graphs

In this chapter, we consider algorithms in the CONGEST model for a directed graph $G = (V, E)$. There has been very little work in this area, and a recent paper [GU15] considers it "an interesting subarea of distributed graph algorithms which deserves more attention". For the directed case we assume that the communication channels (edges in $G$) are bidirectional, even though the graph $G$ represented in the model is directed. Thus, the communication network is represented by the undirected graph $U_G$. For convenience we will also assume that $G$ is weakly connected.

## 7.2 Our Results

We present the following results for path problems on directed graphs in the CONGEST model (see Table 1).

- Unweighted Directed Graphs:

  **Theorem 15.** *Let $D$ be the directed diameter in a directed graph and let $D_u$ be the diameter of an undirected graph. Given an unweighted directed (undirected) graph $G$ on $n$ vertices,*

  1. *Algorithm 34 computes BC scores of all nodes in* $\min\{2n + O(D), 4n\}$

| Graph Problem | *Number of Rounds* | |
|---|---|---|
| | Previous Results | **Our Results** |
| *APSP and BC for unweighted directed graph* | APSP: $2n$ [HW12]<br>BC: $O(m)$ (trivial) | APSP: $\min\{n + O(D), 2n\}$<br>BC: $\min\{2n + O(D), 4n\}$ |
| *BC for unweighted undirected graph* | $O(n)$ ( $\geq 6n$) [HFA$^+$16] | $\min\{2n + O(D_u), 4n\}$ |
| *APSP and BC for weighted dag* | $O(m)$ (trivial) | APSP: $n + O(L)$<br>BC: $2n + O(L)$ |

Table 7.1: A summary of our results in the CONGEST model. Here $D$ ($D_u$) is the directed (undirected) diameter of a directed (undirected) graph (if it is finite), and $L$ is the longest length of a path in a dag. All the algorithms are deterministic and compute exact values. (Note: The $2n$ round APSP algorithm in [HW12] was presented for undirected graphs, but it also works for directed graphs; their improvement to $n + O(D_u)$ does not work for directed graphs.)

*rounds if $G$ is directed ($2n + O(D_u)$ rounds if $G$ is undirected) in the CONGEST model.*

2. *Algorithm 32 computes directed APSP in $\min\{n + O(D), 2n\}$ rounds in the CONGEST model (improved from $2n$ rounds in [LP13]).*

- Weighted Directed Acyclic Graphs:

**Theorem 16.** *Let $L$ be the number of edges in a longest finite path in a directed acyclic graph (dag). Given a weighted dag on $n$ vertices,*

1. *Algorithm 36 computes APSP in $n + O(L)$ rounds in the CONGEST model.*
2. *Algorithm 37 computes BC scores of all nodes in $2n + O(L)$ rounds in the CONGEST model.*

### 7.2.1 Related Work

Path problems on *undirected* graphs have been studied extensively in distributed computing ([Pel00]). All of the results described below are for the CONGEST model.

We recall that for a weighted undirected (or directed) graph, the exact APSP problem can be trivially solved in the CONGEST model in $O(m)$ rounds using the *aggregation* technique, where the entire network is aggregated at a single node. However, no faster algorithms are known. For undirected graphs, if we allow approximations, an $\tilde{O}(n)$-round APSP algorithm was proposed in [LPS13b] for an $O(1)$ approximation factor, later improved to a $1 + o(1)$ factor in [Nan14]. For exact SSSP, the classic Bellman-Ford [Bel58] which runs in $O(n)$ rounds is still the fastest, but an improved $O(n^{3/4} + D_u)$ algorithm with $1 + o(1)$ approximation factor is given in [Nan14] for undirected graphs, further improved to $O(n^{1/2+o(1)} + D_u^{1+o(1)})$ rounds in [HKN16]. Here $D_u$ is the diameter of the undirected graph. Nanongkai results [Nan14] include tools such as *h-hop SSSP* and using *shortcuts* to reduce shortest paths diameter. The results presented by Lenzen and al. [CHKK$^+$15a] work in the CONGEST clique model: they show a fast exact algorithm and a better approximation algorithm for APSP using algebraic methods such as distributed matrix multiplication. Table 7.2 contains a summary of the current results for weighted undirected graphs in the CONGEST model.

For an unweighted undirected graph, a lower bound of $\Omega\left(\frac{n}{\log n}\right)$ for computing diameter was established in [FHW12], which implies a lower bound for solving APSP. Two nearly optimal algorithms for this problem, running in $O(n)$ rounds, were given in [HW12] and [PRT12]. Then, the $\Omega(n/\log n)$ lower bound was recently matched in [HFQ$^+$16] with a $O(n/\log n)$-round algorithm for APSP, which uses a multiplexing technique over the communication channels.

For unweighted directed graphs, we are not aware of published results that

| Problem | Topology | Rounds | Approximation | References | Year |
|---|---|---|---|---|---|
| SSSP | General | $O(n)$ | exact | [Bel58, For56] | 1956-58 |
| | | $\tilde{O}(n^{1/2+1/2k}+D)$ | $8k\lceil\log(k+1)\rceil-1$ | [LPS13b] | 2013 |
| | | $\tilde{O}(n^{3/4}+D)$ | $1+o(1)$ | [Nan14] | 2014 |
| | Clique | $\tilde{O}(n^{1/k})$ | $2k-1$ | [BS07] | 2007 |
| | | $\tilde{O}(n^{1/2})$ | exact | [Nan14] | 2014 |
| APSP | General | $O(m)$ | exact | Trivial | - |
| | | $\tilde{O}(n)$ | $O(1)$ | [LPS13b] | 2013 |
| | | $\tilde{O}(n)$ | $1+o(1)$ | [Nan14] | 2014 |
| | Clique | $\tilde{O}(n^{1/k})$ | $2k-1$ | [BS07] | 2007 |
| | | $\tilde{O}(n^{1/2})$ | $2+o(1)$ | [Nan14] | 2014 |
| | | $O(n^{1/3}\log n)$ | exact | [CHKK+15a] | 2015 |
| | | $O(n^{1-2/w})$ | $1+o(1)$ | [CHKK+15a] | 2015 |
| APASP/BC | General | $O(m)$ | exact | Trivial | - |

Table 7.2: CONGEST models results summary for weighted undirected graphs, where $w < 2.3728639$ is the matrix multiplication exponent [LG14]. $\tilde{O}$ notation hides polylogarithmic factors. $k$ is a positive integer.

claim exact APSP in $O(n)$ rounds, but we were recently aware [Pel] that the APSP algorithm claimed for undirected graphs in [LP13] in fact works for directed graphs. We observe that the bound on the number of rounds is $2n$, and the improved $n + O(D)$ bound obtained there for undirected graphs does not hold for directed graphs. (This $O(n)$ round result is not widely known, and even one of the authors of [LP13] was unaware of its applicability to directed graphs until informed of this by us recently [Pel].) In other works on directed graphs, a randomized algorithm for single source reachability with $\tilde{O}(D_u + \sqrt{n}D_u^{1/2})$ complexity was given in [Nan14], and recently improved in [GU15] to $\tilde{O}(D_u + \sqrt{n}D_u^{1/4})$ rounds with an algorithm that also works for directed graphs; here $D_u$ is the diameter of the underlying undirected graph. This last result gets closer to the lower bound of $\tilde{\Omega}(D_u + \sqrt{n})$ established in [DSHK+11]. Except for the APSP and reachability results in [LP13, Nan14, GU15] and [CHKK+15b] (for the CONGEST clique model), we are unaware of additional literature targeting directed graphs. A randomized algorithm for computing APSP in weighted graphs in $O(n^{5/4})$ rounds has recently appeared in FOCS'17.

Betweenness centrality has recently received attention in the distributed setting [WT13, YTQ17, HFA$^+$16]. While the problem is approached from a practical prospective in [WT13] and [YTQ17], very recently in [HFA$^+$16] the authors give an $O(n)$-round algorithm for computing BC for unweighted undirected graphs in the CONGEST model. They also show an $\Omega(\frac{n}{\log n}+D)$-round lower bound for computing BC and give a method to handle an exponential number of shortest paths.

**Organization of the Chapter.** The rest of the chapter is organized as follows. In Section 7.3 we review the distributed BC algorithm in [HFA$^+$16] and we highlight our new techniques. In Section 7.4 we describe our distributed BC algorithm for unweighted graphs; this section assumes some familiarity with Brandes' algorithm [Bra01] (see Chapter 1). Section 7.5 presents our APSP algorithm for weighted dags and a distributed BC algorithm with the same round complexity. Section 7.6 presents a simple algorithm for reachability, transitive closure, and scc.

## 7.3 Betweenness Centrality

Betweenness centrality has been already experimentally studied in distributed models, but recently a first theoretical approach for undirected unweighted graphs was published in [HFA$^+$16]. In this section, we quickly review the BC algorithm from [Bra01] (note that a full description can be found in Chapter 1); then we provide an overview of the results in [HFA$^+$16].

Brandes' algorithm consists in the following steps: for each source $s$ compute the SSSP dag $DAG(s)$ rooted at $s$ (Alg. 1), for each $DAG(s)$ compute $\sigma_{sv}$ for each $v \in DAG(s)$ (Alg. 1) and, for each $DAG(s)$ starting from the leaves, apply equation 1.4 up to the root (Alg. 2).

The structure of the above algorithm can be naturally adapted into a distributed algorithm. However, some challenges must be addressed to allow us to efficiently compute BC in a directed graph. We describe them section 7.4.

### 7.3.1 BC in Undirected Unweighted Graphs [HFA$^+$16]

Recently, a distributed BC algorithm for unweighted undirected graphs which terminates in $O(n)$ rounds in the CONGEST model was presented in [HFA$^+$16], together with a lower bound of $\Omega(n/\log n)$ rounds for computing BC. This algorithm computes the predecessor lists and the number of shortest paths (Step 3 in Alg. 1) by a natural extension of the unweighted undirected APSP algorithm in [HW12] (see also [PRT12]). The undirected APSP algorithm in [HW12] starts concurrent BFS computations from different sources scheduled by a pebble that performs a DFS traversal of a spanning tree for $G$. Each time the pebble reaches a new node $v$, it pauses for one round before activating $BFS(v)$ and then proceeds to the next unexplored node. At each node $v$, all messages for a given BFS (say started at source $s$) reach $v$ at the same round, and the updated distance is sent out from $v$ in the next round. Hence, before $v$ broadcasts its distance from $s$ to adjacent nodes, it can readily compute and store $P_s(v)$ and $\sigma_{sv}$ using the incoming messages related to $BFS(s)$ in this round. It is well known that this approach does not work in directed graphs, since the APSP algorithm in [HW12] could create congestion (see Figure 7.1).

Since the pebble pauses at each node and a DFS traversal backtracks over $\Theta(n)$ nodes before activating the last BFS, this distributed algorithm for step 3 in Algorithm 1 completes in $3n + O(D)$ rounds.

The distributed algorithm in [HFA$^+$16] for Algorithm 2 is described in the next section. It uses the triangle inequality for its proof of correctness, which does not apply to the directed case (since the pebble backtracks along DFS edges in this algorithm). The algorithm in [HFA$^+$16] also handles the issue that a graph could have an exponential number of shortest paths which would cause the $\sigma_{st}$ values to have a linear number of bits. Since the CONGEST model allows only messages of size $O(\log n)$, they use a floating point representation with $O(\log n)$ bits

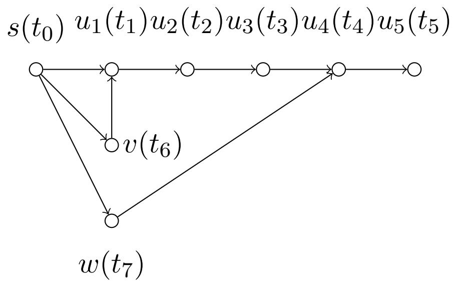


Figure 7.1: Counterexample for the APSP algorithm in [HW12] for directed graphs. Here $BFS(v)$ and $BFS(w)$ will congest at node $u_4$. Value $t_j$ represents the round when the pebble $P$ starts the BFS from the corresponding node, with $t_i < t_j$ iff $i < j$. In this example $BFS(v)$ will start at round $t_6 = 21$, while $BFS(w)$ will start at round $t_7 = 24$. They will both reach $u_4$ at the beginning of round 25 creating a congestion for the next round.

to approximate the $\sigma_{st}$ values. We review this method in Section 7.3.1. We will use this same method in our algorithms since it works without change for directed graphs and for weighted graphs.

### Accumulation Phase for Undirected BC [HFA$^+$16]

The distributed method for Algorithm 2 in [HFA$^+$16] first computes and broadcasts the diameter $D_u$ of the network during the APSP algorithm. Then, each node $v$ sets its accumulation broadcast time for each source $s$ to $T_s(v) = T_s + D_u - d(s, v)$, where $T_s$ is the absolute time when $BFS(s)$ started in the APSP algorithm. The global clock is reset to 0 and each node $v$ sends its accumulation value for $s$ at time $T_s(v)$. Since $D_u - d(s, v) \geq 0$, this approach completes in at most $3n$ rounds. Thus, overall BC algorithm in [HFA$^+$16] runs in $6n + O(D_u)$ rounds.

In Section 7.4 we present another simple method which works for our algorithm and can also replace the above algorithm in [HFA$^+$16].

**Handling Exponential Values [HFA$^+$16]**

Given the $O(\log n)$ bits restriction in the CONGEST model, [HFA$^+$16] maintains approximate values of the $\sigma_{st}$ values using a floating point representation, and guarantee a relative error for the computed BC which is only $O(n^{-c})$ (where $c$ is a constant). In other words the approximated BC score $\hat{BC}(v)$ will be bounded by (see [HFA$^+$16], Theorem 1)

$$\left(\frac{1}{1+n^{-c}}\right) BC(v) \leq \hat{BC}(v) \leq (1+n^{-c})BC(v) \tag{7.1}$$

Since this technique in [HFA$^+$16] works for both undirected and directed graphs (weighted or unweighted), we will use the same method in our algorithms in order to handle exponential counts of paths.

For completeness, we briefly describe this techniques. To express large (exponential) values with a linear number of bits, an integer value $a$ can be represented as $a = y \cdot 2^x$, where $y \in [0,1]$ and $x \in \mathbb{Z}$. If we allow $2L$ bits to store the values of $x$ and $y$, it is possible to bound any integer value $b \in [2^{-2^L+1+L}, 2^{2^L-1}]$ with an estimated ceiling value $a$ such that the relative error between $b$ and $a$ is (see [HFA$^+$16], Lemma 1)

$$\left|\frac{a}{b} - 1\right| \leq 2^{-L+1} = \eta$$

Thus, when we encounter an exponential value for $\sigma_{sv}$, we will instead use an approximated value $\hat{\sigma}_{sv}$ (using the floating point formulation above) where

$$\left(\frac{1}{1+\eta}\right) \sigma_{sv} \leq \hat{\sigma}_{sv} \leq (1+\eta)\sigma_{sv}$$

In order to bound the error in the accumulation phase, another clever technique is used in [HFA$^+$16]. A new parameter $\psi_s(v) = \delta_{s\bullet}(v)$ is used to redefine Equation 1.4

as

$$\psi_s(v) = \sum_{w:v \in P_s(w)} \frac{1}{\sigma_{sw}} + \psi_s(w)$$

The advantage is that $\psi_s(v) = \sum_{q:q \in R_s(v)} 1/\sigma_{sq}$ where $R_s(v)$ is the set of all descendants of $v$ in the SSSP rooted at $s$ (see [HFA$^+$16], Lemma 2). This allows to compute an approximate $\hat{\psi}_s(v) = \sum_{q:q \in R_s(v)} 1/\hat{\sigma}_{sq}$ which, once multiplied for $\sigma_{sv}$, will provide an approximate accumulation value for BC($v$) within the bounds in Equation 7.1.

### 7.3.2 Our Techniques

Our algorithm for betweenness centrality (BC) for directed unweighted graphs (Section 7.4) is again a distributed implementation of Brandes' sequential algorithm for BC (Algorithm 1). But it differs from the distributed algorithm in [HFA$^+$16] due to the fact that we work with a directed graph. In particular, in the algorithm in [HFA$^+$16] for the undirected case it is straightforward to propagate the number of shortest paths (as reviewed in Section 7.3.1), but in the directed case propagating the number of shortest paths (see Step 3, Alg. 1) is not immediate, since the directed APSP algorithm we use does not wait to hear from all predecessors of a node $v$ before broadcasting the shortest distances stored at $v$. We describe our method in Section 7.4.2. For the accumulation phase for BC (Step 5, Alg. 1 which calls Alg. 2), the method in [HFA$^+$16] for undirected graphs was tailored to the undirected APSP algorithm they use, and it does not work for our method. We give a simple time reversing technique in Section 7.4.3 which works for our algorithm, and can also replace the accumulation technique used in [HFA$^+$16], to improve the round complexity by a constant factor. However, an even faster solution for the unweighted undirected case is to simply use our unweighted directed algorithm.

Our weighted APSP algorithm for a dag (Section 7.5) initially constructs a longest length tree (LLT) using a 'delayed' BFS. The level of a node in this tree guides the global delay that the node will introduce for its outgoing messages. These global delays guarantee that a node has received its shortest path from a given source before it sends out shortest path information along outgoing edges of the graph. This allows to efficiently compute BC in $2n + O(L)$ rounds in weighted dags by using techniques which are already known for undirected graphs and our accumulation procedure.

## 7.4 BC in Unweighted Directed Graphs

In this section, we present our algorithm for computing betweenness centrality in unweighted directed graphs in the CONGEST model. In Section 7.4.1 we give a short review of the Lenzen-Peleg distributed unweighted APSP algorithm [LP13]. In Section 7.4.2 we present our enhanced distributed APSP algorithm for unweighted directed graphs. Section 7.4.3 gives our simple distributed algorithm for the accumulation phase (Alg. 2) in Brandes' algorithm, and our overall BC algorithm.

### 7.4.1 Directed APSP Algorithm in [LP13]

The Lenzen-Peleg algorithm [LP13] consists of lines 2–3, 5–8, 13–15 and 17 in Alg. 32. (The overall Alg, 32 is our enhanced algorithm and is described in Section 7.4.2; the new lines introduced by us are marked by a • to indicate they are not part of the algorithm in [LP13].)

Using the notation in [LP13], $L_v$ is an ordered list at node $v$ which stores pairs $(d_s, s)$, where $s$ is a source and $d_s$ is the current best upper bound received at $v$ for $\delta(s, v)$ (the shortest distance from $s$ to $v$). These pairs are stored on $L_v$ in lexicographically sorted order, with $(d_q, q) < (d_s, s)$ if either $d_q < d_s$, or $d_q = d_s$ and $q < s$. Initially each node $v$ has just the pair $(0, v)$ on $L_v$ and this pair has *status* set to *ready* (Step 3, Alg. 32). Let $L_v^r$ be the state of $L_v$ at the beginning of round $r$, and let $\ell_v^{(r)}(d_s, s)$ be the index of the pair $(d_s, s)$ in $L_v^r$. In round $r$, each node $v$ sends along its outgoing edges the pair with smallest index in $L_v^r$ which has its *status* still set to *ready*, and then sets the *status* of this pair to *sent* (Steps 6–8, Alg. 32). If $v$ receives $(d_x, x)$ in the current round for possible placement on $L_v$ (Step 13, Alg. 32), it places this pair on $L_v$ in sorted order if no entry for $x$ is already present (Steps 14–15 and 17, Alg. 32). If there is an entry $(d'_x, x)$ on $L_v$ then if $d'_x \leq d_x$ the new pair is discarded while if $d'_x > d_x$, the old pair is discarded, and $(d_x, x)$ is placed on $L_v$ in sorted order (Steps 14–15 and 17, Alg. 32).

It is shown in [LP13] that this computation completes in $2n$ rounds and correctly computes shortest path distances to $v$, from each node $s$ that has a path to $v$. Although this is claimed in [LP13] only for undirected APSP the result holds for directed APSP as well. The above algorithm is enhanced to an $n + O(D_u)$ round algorithm in [LP13] by using the fact that in undirected graphs the height of any BFS tree is a 2-approximation to the diameter. However, this result does not apply to directed graphs since a directed BFS tree height can be much smaller than the directed diameter.

In Section 7.4.2 we present a method to improve the number of rounds from $2n$ to $n + O(D)$ that works for both directed and undirected unweighted graphs. Further, since we are interested in computing BC, we enhance the APSP algorithm to also compute for each node $v$ the set $P_s(v)$ of predecessors of $v$ in the shortest path dag rooted at each source $s$, and the number of shortest paths $\sigma_{sv}$ from $s$ to $v$. In the undirected BC algorithm in [HFA$^+$16], the computation of shortest paths (using the associated APSP algorithm) proceeds level by level in each BFS, so $P_s(v)$ and $\sigma_s(v)$ are readily computed along with $\delta(s, v)$, but in the algorithm we consider here for directed graphs, a node $v$ could receive the messages from its predecessors at different time steps. In [LP13], since only APSP is of interest, a node forwards only the first shortest path message it receives from a predecessor in its shortest path dag. But here we need to monitor messages from all incoming edges to identify the shortest path predecessors and to compute the number of shortest paths for each source. These enhancements are made in Algorithm 32, together with a call to our new Algorithm 33 to reduce the number of rounds to $n + O(D)$ (when $D$ is finite); this is described in the next section. We will use the output of Algorithm 32 to compute directed BC in Algorithm 34 in Section 7.4.3.

### 7.4.2 Improved Directed APSP Algorithm

In our improved algorithm (Alg. 32) we enhance the APSP algorithm in [LP13] to compute the number of shortest paths and the set of predecessors through the steps marked with a • in Algorithm 32, which we discuss next in Section 7.4.2. We also reduce the number of rounds to $n + O(D)$ when the directed diameter $D$ is finite using Algorithm 33, which will be described in Section 7.4.2.

#### Predecessors and Number of Shortest Paths

As noted in Section 7.4.1, the shortest path messages from the predecessors of a vertex $v$ in the shortest path dag for a source $s$ can arrive at different rounds in the algorithm in [LP13], and this is the case in Algorithm 32 as well. At the same time, the algorithm cannot afford to wait to receive all of the shortest path values before propagating the distance since it is possible that a source may not even have a path to some of the predecessors. It was established in [LP13] that the shortest path distance from $s$ to $v$ is finalized in $L_v$ at round no later than round $r = \delta(s, v) + \ell_v(s)$ by their algorithm. We enhance this result to show that in Algorithm 32 the correct $\sigma_{sv}$ value and the predecessor list $P_s(v)$ are finalized by round $r$, and this $\sigma_{sv}$ value is sent out by $v$ (Steps 9 – 11, Alg. 32), even though it might have sent out the $\delta(s, v)$ value earlier. If the estimated distance $d_{sv}$ (from $s$ to $v$) is updated at $v$ to a smaller value before round $\delta(s, v) + \ell_v(s)$, then Algorithm 32 starts afresh by resetting the predecessors list and the number of shortest paths (Step 16, Alg. 32).

This is necessary because our algorithm sends the current best estimate for the distance from a source in Step 8, similar to the algorithm in [LP13], even if this value is not known be the correct shortest path distance and could be updated later to a smaller value. At the same time, we also accumulate predecessors and number of paths from all neighbors that reach $v$ with the current value computed for the shortest distance from source $s$ (Steps 18 – 20, Alg. 32). (Steps 1 and 12 in Alg. 32 will be discussed in Section 7.4.2 and Step 10 will be discussed in Section 7.4.3.)

Algorithm 32 may need to send more than one message from a vertex $v$ in a round (e.g., in both Steps 8 and 11, or because the parallel computation of Step 1), but it never sends more than a constant number of messages. In such cases, $v$ will combine all these messages into a single $O(\log n)$-bit message in that round.

We now establish the correctness of Algorithm 32. For convenience, in Lemma 42 we will use $(d_{sv}, s)$ to denote the pair $(d_s, s)$ in $L_v$. We recall that $\ell_v^{(r)}(d_s, s)$ is the index of the pair $(d_s, s)$ in $L_v^r$.

**Lemma 42.** *For any source $s$ from which node $v$ is reachable, Algorithm 32 sends the correct shortest distance message $(\delta(s, v), s)$ to all nodes in $\Gamma_{out}(v)$ no later than round $r = \ell_v^{(r)}(d_{sv}, s) + d_s$ and the correct $\sigma_{sv}$ value in a message in round $r$. Also, $P_s(v)$ contains the correct predecessors of $v$ in $s$'s shortest path dag at round $r$.*

*Proof.* The fact that $v$ sends the correct shortest distance message $(\delta(s, v), s)$ to all nodes in $\Gamma_{\text{out}}(v)$ no later than round $r = \ell_v^{(r)}(d_{sv}, s) + d_s$ is shown in [LP13], so here we establish correctness of the $\sigma_{sv}$ value in the message $(d_{s,s}, \sigma_{sv})$ sent by $v$ in round $r$ (Step 11 in Alg. 32). The $\sigma_{sv}$ values for all $s, v$ such that $(s, v)$ is an edge are correctly set as 1 at $v$ by the initialization in Step 4 and the update in Step 20 in round 1 (and never updated further), and hence the $\sigma_{sv}$ values sent out in Step 11 are correct when $\delta_{sv} = 1$. Consider round $r = \ell_v^{(r)}(d_{sv}, s) + d_s$ and assume that the messages sent out in Step 11 by all nodes have the correct values up to round $r - 1$. Let $u$ be any predecessor of $v$ in $s$'s shortest path dag, and for $L_u$ consider the round

**Algorithm 32** Directed-APSP($G$)

```
1: compute (in parallel with Step 5) a BFS tree $B$ rooted node $v_1$; each node $u$ computes its set of children $C_u$ and its parent $p_u$ in $B$ $\bullet$ {This will be used in Alg. 33}
2: for each node $v$ in $G$ do
3:   $L_v \leftarrow ((0, v))$; set $status((0, v))$ = ready; set flag $f_v \leftarrow 0$ {Initialize}
4:   for each source $s$ in $G$ do if $s = v$ then $\sigma_{vv} \leftarrow 1$ else $\sigma_{sv} \leftarrow 0$; $P_s(v) \leftarrow \emptyset$ $\bullet$
5:   for rounds $1 \le r \le 2n$ do {Step 12 could cause termination before round $2n$}
6:     if $\exists\,(d_s, s) \in L_v$ with $status((d_s, s)) = ready$ then
7:       let $(d_s, s)$ the lexic. smallest message in $L_v$ with $status((d_s, s)) = ready$
8:       send $(d_s, s)$ to all outgoing edges of $v$; set $status((d_s, s)) = sent$
9:     if $r = d_s + \ell_v^r(d_s, s)$ $\bullet$ then
10:      $T_{sv} \leftarrow r$ $\bullet$ {This will be used in Alg. 34}
11:      send $(d_s, s, \sigma_{sv})$ to all nodes in $\Gamma_{\text{out}}(v)$ $\bullet$
12:    run APSP-Finalizer($v, p_v, C_v$) $\bullet$ {See Alg. 33}
13:    for a received $(d_s, s)$ from an incoming neighbor $(u, v)$ do
14:      if $\nexists\,(d'_s, s) \in L_v$ with $d'_s \le d_s + 1$ then
15:        remove the message of the form $(\cdot, s)$ from $L_v$
16:        set $P_s(v) \leftarrow \emptyset$ and $\sigma_{sv} \leftarrow 0$ $\bullet$
17:        add $(d_s, s)$ to $L_v$; set $status((d_s, s)) = ready$
18:    for a received $(d_s, s, \sigma_{su})$ from an incoming neighbor $(u, v)$ $\bullet$ do
19:      if $\exists\,(d'_s, s) \in L_v^r$ with $d'_s = d_s + 1$ $\bullet$ then
20:        node $v$ updates $\sigma_{sv} \leftarrow \sigma_{sv} + \sigma_{su}$; $P_s(v) \leftarrow P_s(v) \cup \{u\}$ $\bullet$
```

$r' = \delta(s, u) + \ell_u^{r'}(\delta(s, u), s)$ when $u$ sends $\sigma_{su}$ in the message sent by it in Step 11. For convenience, let $i = \ell_u^{r'}(\delta(s, u), s)$. If $r' < r$, then by our assumption (that all messages sent out before round $r$ have the correct value) $u$ will have sent the correct value of $\sigma_{su}$ to $v$ before round $r$. But $r'$ must be less than $r$ since $\delta(s, u) = \delta(s, v) - 1$, and further, the $i - 1$ entries in $L_u$ ahead of $(\delta(s, u), s)$ (at the start of round $r'$) were sent by $u$ before round $r'$, since they must all have had status *sent* at start of round $r'$. All of these $i - 1$ messages from $L_u$ will be in $L_v$ with distance value at most one greater than the value in $L_u$ (the distance value could be smaller if it were decreased by a message received at $v$ from a predecessor other than $u$). But since $(d_{sv}, s)$ is added to $L_v$ with value $d_{sv} = d_{su} + 1$ in round $r'$, all of these $i - 1$ messages from $u$ will be ahead of it in $L_v$ and hence $(d_{sv}, s)$ is placed at index $i$ or greater in $L_v$ in round $r'$. But this implies $\delta(s,v) + \ell_v^r(\delta(s,), s) \geq \delta(s,u) + 1 + i = r' + 1$. Hence $r > r'$, i.e, $v$ receives the correct $\sigma_{su}$ value from $u$ before round $r$. This holds for every predecessor of $v$ in $s$'s shortest path dag. Further all of these $\sigma_{su}$ values are added to (an initially zero-valued) $\sigma_{sv}$ in Step 20, so $v$ sends out the correct $\sigma_{sv}$ value in round $r$. Finally, since the predecessor list $P_s(v)$ is updated each time $\sigma_{sv}$ is updated, this list is also correct by the above argument. □

**Improving the Round Complexity**

When the diameter $D \leq n/3$, Algorithm 33 guarantees that all nodes stop their activity after at most $n + O(D)$ rounds. Moreover, it broadcasts $D$ to all nodes in $G$. In the case where $D > n/3$, Algorithm 32 will terminate no later than $2n$ rounds, because of its *for* loop in step 5.

We now describe Algorithm 33. Let $B$ be the tree created in Step 1, Alg. 32. Note that $B$ will be completely defined after $D$ rounds, and the activity of Alg. 33 for a node $v$ becomes relevant only after $n$ rounds. In the first step, the algorithm checks if $v$ has received the diameter $D$ from its parent $p_v$ in $B$. In this case, $v$ broadcasts $D$ to all its children in $C_v$ and it stops. Otherwise, the algorithm checks if $v$ has received one estimate finite distance from every node in $G$ (Step 2, Alg. 33). (The flag $f_v$ is used to avoid the repetition of steps 3–12 after they are performed once.) These distances will be correct at rounds $r \geq \max_s(d_s + \ell_v^{(r)}(d_s, s))$ (see Lemma 42), and Algorithm 33 proceeds by distinguishing two cases: if a node $v$ is a leaf in the tree $B$ (Step 3, Alg. 33), it computes the maximum shortest distance $d_v^*$ from any other node $s$ and broadcasts $d_v^*$ to its parent $p_v$ in $B$ (Step 4, Alg. 33). Then, $v$ will wait up to round $2n$ for the diameter $D$ from its parent $p_v$ in $B$ (because of the check in step 1, Alg. 33).

In the second case, when $v$ is not a leaf (and not $v_1$), if it has collected (for the first time) the distances $d_c^*$ from all its children in $C_v$ (Step 6, Alg. 33), it will execute the following steps only once (thanks to the flag $f_v$ initialized to 0 in Alg. 32, and updated to 1 in Step 10, Alg. 33): $v$ computes the maximum shortest distance $d_v^*$ from any other node $s$ (Step 7, Alg. 33), then it computes the largest distance value $d_{C_v}^*$ received from its children in $C_v$ (Step 8, Alg. 33). Thus, $v$ sends the largest value among $d_v^*$ and $d_{C_v}^*$ to its parent $p_v$ (Step 10, Alg. 33), and it waits for $D$ from $p_v$ as in the first case. Finally, when $v$ is in fact $v_1$, after receiving the distances from all its children, it broadcasts the diameter $D$ to its children in $C_{v_1}$ (Step 12, Alg. 33).

**Algorithm 33** APSP-Finalizer($v, p_v, C_v$) $\triangleright$ $p_v, C_v$ computed in Step 1, Alg. 32

```
Ensure: Compute and broadcast the network directed diameter $D$, if $D \leq n/3$
 1: if $v$ receives diameter $D$ from parent $p_v$ in round $r < 2n$, it broadcasts $D$ to all
    nodes in $C_v$ and stops
 2: if $|L_v^r| = n$ and $f_v = 0$ then
 3:    if $r = \max_s(d_s + \ell_v^{(r)}(d_s, s))$ and $C_v = \emptyset$ then {$v$ is a leaf in the BFS tree $B$}
 4:       $d_v^* \leftarrow \max_s(d_s)$; send $d_v^*$ to parent $p_v$; $f_v \leftarrow 1$
 5:    if $r \geq \max_s(d_s + \ell_v^{(r)}(d_s, s))$ then {completed only once}
 6:       if $v$ has collected distances $d_x^*$ from all children $x \in C_v$ then
 7:          $d_v^* \leftarrow \max_s(d_s)$
 8:          $d_{C_v}^* \leftarrow \max_{x \in C_v}(d_x^*)$
 9:          if $v \neq v_1$ then
10:             send $\max(d_v^*, d_{C_v}^*)$ to parent $p_v$; $f_v \leftarrow 1$
11:          else {when $v = v_1$}
12:             broadcast $D = \max(d_{v_1}^*, d_{C_{v_1}}^*)$ to $C_{v_1}$; stop
```

It is readily seen that Algorithm 33 broadcasts the correct diameter to all nodes in $G$. In fact, after round $r = \max_s(d_s + \ell_v^{(r)}(d_s, s))$ the value $d_s$ cannot decrease at $v$ for any source $s$ since this would violate Lemma 42. Hence the $d_s$ values at $v$ after this round are the correct shortest path lengths to $v$. Moreover, since $\max_s(d_s + \ell_v^{(r)}(d_s, s)) > n$ when $|L_v^r| = n$, Step 1 of Alg. 32 is completed and each node $v$ knows its parent and its children in $B$. Thus, the value sent by $v$ to its parent in Step 10 of Alg. 33 is the largest shortest path length to any descendant of $v$ in $B$, including $v$ itself. Thus, node $v_1$ computes the correct diameter of $G$ in Step 12, Alg. 33.

**Lemma 43.** *The execution of Algorithm 32 requires at most* $\min\{2n, n+3D\}$ *rounds.*

*Proof.* Step 1 of Alg. 32 can be completed in $D$ rounds using standard techniques, and it is executed in parallel with the loop in step 5, Alg. 32. Moreover, when $D = \infty$ each node stops after $2n$ rounds because of step 5 of Alg. 32.

When $D$ is bounded, each $v \in V$ will have $|L_v^r| = n$ at some round $r$. In Alg. 33 (called in Step 12, Alg. 32), the longest shortest path value reaches $v_1$ within $D$ rounds after the last node computes its local maximum value. At this point $v_1$ computes the diameter $D$ and broadcasts it to all nodes $v$ in at most $D$ steps. In addition to these $2D$ steps, the last extraction from a set $L_v$ (for any $v$) is performed no later than step $\max_v \max_s\{d_s + \ell_v^{(r)}(d_s, s)\} \leq n + D$. Thus, the total number of rounds is at most $n + 3D$. The lemma is proved. □

### 7.4.3 Accumulation Technique and BC Computation

In Algorithm 34 we present a simple distributed algorithm to implement the accumulation phase in the Brandes algorithm (Alg. 2). Recall that in Algorithm 32, in the round when node $v$ broadcasts its finalized message $(d_{sv}, s, \sigma_{sv})$ on its outgoing edges in step 11, it also notes the absolute time of this round in $T_{sv}$. Also, by Lemma 43, Alg. 32 completes in round $R = \min\{n + 3D, 2n\}$. Alg. 34 sets the global clock to 0 in Step 3 after these $R$ rounds complete in Alg. 32. In Step 5 each node $v$ computes its accumulation round $A_{sv}$ as $R - T_{sv}$. Then, $v$ computes $\delta_{s\bullet}(v)$ and broadcasts $\frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$ to its predecessors in $P_s(v)$ in round $A_{sv}$ (Steps 6–8, Alg. 34).

Although we have described the Alg. 34 specifically as a follow-up to Algorithm 32, it is a general method that works for any distributed BC algorithm where

**Algorithm 34** BC($G$)

```
1: run Algorithm 32 on $G$; let $R$ be the termination round for Alg 32
2: {Recall that $T_{sv}$ is the round when $v$ broadcasts $(d_s, \sigma_{sv})$ to $\Gamma_{\text{out}}(v)$ in Step 11,
   Alg. 32}
3: set absolute time to 0
4: for each node $v$ in $G$ do
5:    for all $s$ do $A_{sv} = R - T_{sv}$
6:    for round $0 \le r \le R$ do {Each iteration of the for loop is a round}
7:       if $r = A_{sv}$ then send $m = \frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$ to $v$'s predecessors in $P_s(v)$
8:       for a received $m$ from an outgoing neighbor in $\Gamma_{\text{out}}(v)$ do $\delta_{s\bullet}(v) \leftarrow \delta_{s\bullet}(v) +$
         $m$
```

each node can keep track of the round in which step 3 in Algorithm 1 (Brandes' algorithm) is finalized for each source. This is the case not only for Algorithm 32 for both directed and undirected unweighted graphs, but also for our BC algorithm for weighted dags in the next section, and for the BC algorithm in [HFA$^+$16] for undirected unweighted graphs (though our Algorithm 32 uses a smaller number of rounds). On the other hand, the distributed accumulation phase in [HFA$^+$16] is tied to the start times of the shortest path computations at each node in the first phase of the undirected APSP algorithm they use, and hence is specific to their method.

**Lemma 44.** *In Algorithm 34 each node $v$ computes the correct value of $\delta_{s\bullet}(v)$ at round $A_{sv} = R - T_{sv}$, and the only message it sends in round $A_{sv}$ is $m = \frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$, which it sends to its predecessors in the SSSP dag for $s$.*

*Proof.* We first show that at time $A_{sv}$, node $v$ has received all accumulation values from its successors in DAG($s$). This follows from the fact that, in the forward phase, each successor $w$ of $v$ will send its message for source $s$ to nodes in $\Gamma_{\text{out}}(w)$ in round $T_{sw}$, which is guaranteed to be strictly greater than $T_{sv}$. Thus, since $A_{sw} < A_{sv}$, node $v$ will receive the accumulation value from every successor in the dag for $s$ before time $A_{sv}$, and hence computes the correct values of $\delta_{s\bullet}(v)$ and $\frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$. Further, since the timestamps $A_{sv}$ are different for different sources $s$, only the message for source $s$ is sent out by $v$ in round $A_{sv}$. □

## 7.5 APSP and BC in Weighted DAGs

We now consider the case when the input graph $G = (V, E)$ is a directed acyclic graph (dag), where each edge $(x, y)$ has an $O(\log n)$ bit weight $\mathbf{w}(x, y)$. For simplicity, we will assume that the dag has a single source $s$. If the dag contains multiple sources $(s_1, \ldots, s_k)$, we will assume a virtual source $\hat{s}$ which is connected with a direct edge to the real sources. The procedures we present can be readily adapted to the multiple sources using such a virtual source.

We start with some definitions. Given a path $\pi$ in $G$, the *length* $\ell(\pi)$ will denote the number of edges on $\pi$ and the *weight* $w(\pi)$ will denote the sum of the weights on the edges in $\pi$. The shortest path weight from $x$ to $y$ will be denoted by $\delta(x, y)$.

**Definition 4.** *A* longest length tree (LLT) $T_s$ *for a dag* $G$ *is a directed spanning tree rooted at its source* $s$ *where, for each node* $v$*, the path in* $T_s$ *from* $s$ *to* $v$ *has the maximum length (number of edges) of any path from* $s$ *to* $v$ *in* $G$*. The level* $\ell(v)$ *of a node* $v$ *is the length of the path from* $s$ *to* $v$ *in* $T_s$*. An edge* $(u, v) \in E$ *is a* crossing *edge if* $u$ *is not an ancestor of* $v$ *in* $T_s$*.*

We can similarly define an LLdag, and compute it with a slight extension to Algorithm 35. However, for our results an LLT suffices.

### 7.5.1 APSP in a Weighted DAG

Our distributed algorithm LLT$(G)$ (Alg. 35) uses a *delayed-BFS algorithm* to compute an LLT for dag $G$. It starts a BFS from the (virtual) source $s$ (Step 2, Alg. 35), and it delays the BFS extension from each node $v$ until each incoming node $u \in \Gamma_{\text{in}}(v)$ has propagated its longest length $\ell(u)$ from $s$ to $v$. Then node $v$ will finalize the longest length received, $\max_{u \in \Gamma_{\text{in}}(v)}(\ell(u)+1)$, as its level $\ell(v)$ (Steps 4 – 6, Alg. 35) and it will broadcast $\ell(v)$ to all outgoing nodes $x \in \Gamma_{\text{out}}(v)$ (Step 8, Alg. 35).

**Algorithm 35** LLT($G$)

```
1: set $\ell(v) \leftarrow 0$, $\pi(v) \leftarrow NIL$ for all $v \in G$
2: start a BFS from the (virtual) source $s$, broadcasting $\ell(s) = 0$ to all nodes in
   $\Gamma_{\text{out}}(s)$
3: for each node $v \in V$ do {actions of each node during the BFS}
4:   for each message $\ell(u)$ received from node $u \in \Gamma_{\text{in}}(v)$ do
5:     if $\ell(u) + 1 > \ell(v)$ then
6:       $\ell(v) \leftarrow \ell(u) + 1$; $\pi(v) \leftarrow u$
7:   if $v$ has just received a message from the last node in $\Gamma_{\text{in}}(v)$ then
8:     $v$ broadcasts $\ell(v)$ to all nodes in $\Gamma_{\text{out}}(v)$
```

The proof of the following lemma is straightforward.

**Lemma 45.** *Algorithm LLT computes the parent pointers $\pi(\cdot)$ for an LLT tree $T_s$ of dag $G$ in $L$ rounds, where $L$ is the length of a longest finite directed path in $G$.*

**Observation 17.** *For an LLT $T_s$, every edge $(u, v)$ has $\ell(u) < \ell(v)$.*

The above observation readily follows from the fact that, given the edge $(u, v)$, $\ell(v)$ can be made at least one larger than $\ell(u)$ by taking a longest path from $s$ to $u$ and following it with edge $(u, v)$.

To compute APSP in the weighted dag $G$, we assume the vertices are numbered 1 to $n$, and we construct the SSSP dags for all sources by using a predetermined schedule based on nodes IDs and levels in the *LLT* of $G$. Algorithm 36 presents the overall weighted dag APSP algorithm. For each node $v$, the SSSP at node $v$ starts at absolute time $ID_v + \ell(v)$ (Step 3, Alg. 36), where a message containing the source $v$, the distance (0) and the number of shortest paths (1), is sent to each outgoing node $w$ of $v$. In general, messages for SSSP($x$) will leave node $y$ at absolute time $t_{xy} = ID_x + \ell(y)$ (Steps 5 – 11, Alg. 36), where $\ell(y)$ is the level of $y$ in the

*LLT* of $G$ (see Fig. 7.2). After $y$ receives all the distances (from a source $x$) from its incoming nodes, it updates its shortest distance $\delta(x, y)$ as the minimum value received (Step 7, Alg. 36). Moreover, for a node $y$ we also compute in the set $P_x(y)$, the predecessors of $y$ in the SSSP dag rooted at $x$ (Step 9, Alg. 36), and the number of shortest paths from $x$ to $y$ in $\sigma_{xy}$ (Step 10, Alg. 36). These values are computed for each source $x$ from which $y$ is reachable, and they are crucial since we will use this algorithm in the next section to compute BC.

**Algorithm 36** Weighted dag-APSP($G$)

```
1: compute a directed LLT $T_s$ rooted at source $s$ of $G$ and the level $\ell(v)$ of each
   vertex $v$ using algorithm LLT (Alg.35)
2: set absolute time to 0
3: **for** each node $v$ **do** start SSSP($v$) at absolute time $ID_v + \ell(v)$ by sending
   $(v, 0, \sigma_{vw} = 1)$ to each node $w \in \Gamma_{\text{out}}(v)$
4: **for** each pair of vertices $x, y$ with $\ell(x) < \ell(y)$ **do**
5:    schedule the following at node $y$ at absolute time $t_{xy} = ID_x + \ell(y)$ for SSSP($x$)
      :
6:    for each $u \in \Gamma_{\text{in}}(y)$ let $(x, \delta(x, u), \sigma_{su})$ be the message received by $y$ for
      SSSP($x$) from $u$
7:    compute $\delta(x, y)$ as $\min_{u \in \Gamma_{\text{in}}(y)} \delta(x, u) + \mathbf{w}(u, y)$ (if at least one value $\delta(x, u)$
      is received), otherwise set $\delta(x, y) \leftarrow \infty$
8:    **if** $\delta(x, y) \neq \infty$ **then**
9:       $P_x(y) \leftarrow \{u \in \Gamma_{\text{in}}(y)$ such that $\delta(x, u) + \mathbf{w}(u, y) = \delta(x, y)\}$
10:      $\sigma_{xy} \leftarrow \sum_{u \in P_x(y)} \sigma_{xu}$
11:      send message $(x, \delta(x, y), \sigma_{xy})$ to $z$ for each $z \in \Gamma_{\text{out}}(y)$
```

*Complexity.* We establish correctness and round complexity of Algorithm 36 with the following two lemmas.

**Lemma 46.** *The value $\delta(x, y)$ computed in Step 7 is the correct shortest path distance from $x$ to $y$.*

*Proof.* Since every edge $(u, v)$ in $G$ has $\ell(u) < \ell(v)$ (see Observation 17), any path from $x$ to $y$ in $G$ has length at most $\ell(y) - \ell(x)$. The SSSP($x$) starts at $x$ at absolute time $ID_x + \ell(x)$, and hence the value sent on every path from $x$ to $y$ in $G$ arrives at $y$

at absolute time $ID_x + \ell(y)$ or less. Since $\delta(x, y)$ is computed as the minimum of the values received at time $ID_x + \ell(y)$, this is the correct $x$–$y$ shortest path weight. $\square$

**Lemma 47.** *Each node transmits a message for at most one SSSP in each round.*

*Proof.* Consider a node $x$ and let $u$ and $v$ be any two nodes from which $x$ is reachable. Node $x$ will transmit the message for SSSP($u$) in round $ID_u + \ell(x)$, and the message for SSSP($v$) in round $ID_v + \ell(x)$. Hence, these messages will be transmitted on different rounds. Hence the message for at most one SSSP dag will be sent out by $x$ in each round. $\square$

Finally, since $ID_x + \ell(y) \leq n + L$ for all $x, y \in V$, the round complexity of computing APSP in a weighted dag is $n + O(L)$.

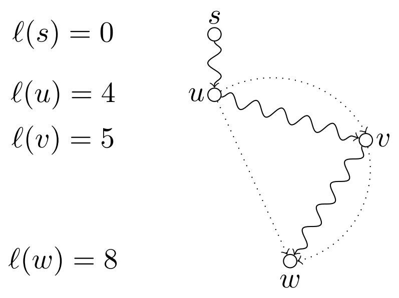


Figure 7.2: Example of SSSP distributed execution from $u$ and $v$. Snake lines represent the LLT $T_s$ path. Dotted lines are shortest paths. SSSP($u$) and SSSP($v$) will leave $w$ at different times $ID_u + 8$ and $ID_v + 8$. Moreover the two SSSP($u$) paths $u \rightsquigarrow w$ and $u \rightsquigarrow v \rightsquigarrow w$ could reach $w$ at different time steps, but they will be processed (only the shortest path will be propagated) at the same absolute time $ID_u + 8$.

### 7.5.2 BC in a Weighted DAG

We can now use Algorithm 34 to compute betweenness centrality in a weighted dag $G$ using the round numbers $t_{xy}$ computed in Algorithm 36 to schedule the accumulation step at node $y$ for source $x$. As seen from step 5 of Algorithm 36, $t_{xy}$ is the round when node $y$ broadcasts $\delta(x,y)$ and $\sigma_{xy}$ to all its outgoing nodes (Steps 5–11, Alg. 36). Thus, similarly to Alg. 34, in the accumulation round $A_{xy} = 2n - T_{xy}$ where $T_{xy} = t_{xy}$ (Step. 5, Alg. 34), node $y$ will receive all the accumulation values from every successor in the dag for $x$, and it will compute the correct value of $\delta_{x\bullet}(y)$. The overall dag BC algorithm is in Algorithm 37. It uses double the number of rounds as the dag APSP algorithm, and hence runs in $2n + O(L)$ rounds.

**Algorithm 37** Weighted dag-BC($G$)

```
1: run Algorithm 36 on $G$
2: **for** all $s, v$, $T_{sv} \leftarrow t_{sv}$, where $t_{sv}$ is from Step 5 in Algorithm 36
3: set absolute time to 0
4: run steps 4–8 of Algorithm 34 on $G$
```

## 7.6 A Simple Transitive Closure Algorithm

Algorithm Reachability (Algorithm 38) maintains for each node $v$ in $G$ an array $A_v[1..n]$, initially with all positions unmarked, which at termination will have marked every $A[y]$ where $v$ has a path from $v$ to $y$. In this algorithm each vertex $v$ initializes a queue $Q_v$ with $v$ and marks $v$ in $A_v$. In each round every node $v$ in the graph dequeues an element from its queue and broadcasts it along its out-edges, i.e., to the nodes in $\Gamma_{\text{out}}(v)$. For each node ID $y$ that $v$ obtains along an in-edge, it checks if $A_v[y]$ is unmarked, and if so, it marks $A_v[y]$ and then enqueues $y$. This computation proceeds for $2n$ rounds and then terminates. Clearly the algorithm runs in $O(n)$ rounds.

*Correctness.* We now show that this algorithm correctly marks in $A_v$ exactly those positions $y$ such that $v$ is reachable from $y$. Since any node $x$ is eventually broadcasted over all the outgoing edges, and is blocked only if the current node already

**Algorithm 38** Reachability($G$)

```
1: for each $v \in V$ do initialize $Q_v \leftarrow \langle v \rangle$ and mark $A_v[v]$
2: for each node $v$ in parallel for $2n$ rounds do
3:     $v$ dequeues an element $x$ from $Q_v$ (if nonempty)
4:     $v$ broadcasts $x$ to its outgoing nodes $\Gamma_{\text{out}}(v)$
5:     for each $y$ received by $v$ from a node in $\Gamma_{\text{in}}(v)$ do
6:         if $A_v[y]$ is unmarked then
7:             $v$ marks $A_v[y]$ and enqueues $y$ in $Q_v$
```

processed it, it will eventually reach every other node $y$ for which there is a path $x \rightsquigarrow y$ in $G$.

So, we only need to show that $2n$ rounds suffice to correctly identify all nodes $y$ from which $v$ is reachable, for each $v \in V$. For this we first define the notion of a *clean path.* A directed path $\pi_{xy} = (x, v_1, v_2, \ldots, v_j, y)$ is a *clean path* from $x$ to $y$ if for each vertex $v$ in $\pi_{xy}$ other than $x$, $A_v[x]$ is unmarked when $x$ arrives at $v$ along an edge in $\pi_{xy}$.

**Lemma 48.** *For any pair of nodes $x, y$ such that $y$ is reachable from $x$, there exists a clean path $\hat{\pi}_{xy}$ from $x$ to $y$.*

*Proof.* We show the existence of $\hat{\pi}_{xy}$ by constructing it, starting from the last node $y$. Let $v$ be the node in $\Gamma_{\text{in}}(y)$ that sent $x$ to $y$ at the earliest time step (breaking ties arbitrarily). If $v = x$ we are done. Otherwise, we repeat this process by checking in $\Gamma_{\text{in}}(v)$ the node that sent $x$ to $v$ in the earliest round. At any time, let $\pi_t$ be the current path built so far that ends in $y$. We claim that every time we extend $\pi_t$ to the left we either reach $x$ or we add a node that does not already appear in the path. To see this, we observe that if we extend to a node $u$ already in $\pi_t$, we form a cycle of the form $C = (u, \ldots, u)$ where each node pushed $x$ to its successor for the first time. But since all the nodes in $C$ must be different from $x$, the node $x$ was sent into $C$ for the first time by a node outside $C$. A contradiction. □

The clean path $\hat{\pi}_{xy}$ is used in the proof of the next lemma.

**Lemma 49.** *Let $\hat{\pi}_{xy}$ be the clean path from $x$ to $y$ in $G$. Then the message containing node $x$, propagated by Algorithm 38, will reach $y$ in at most $n - 1 + |\hat{\pi}_{xy}| < 2n$ rounds.*

*Proof.* Let us consider the traversal $\tau$ of $x$ along the clean path $\hat{\pi}_{xy}$. If the queue at every vertex $v$ in $\hat{\pi}_{xy}$ is empty when $x$ reaches $v$ along $\tau$, then $x$ will reach $y$ in at most $|\hat{\pi}_{xy}|$ rounds since there is no delay in $\tau$. In general, some of the queues for vertices in $\hat{\pi}_{xy}$ will be nonempty when $x$ arrives. We now claim that if $R$ is the set of distinct vertices that are sent along some edge in the path $\hat{\pi}_{xy}$ ahead of $x$'s traveral $\tau$, then the delay that $\tau$ can incur in the worst case is $|R|$. To see this, consider any vertex $u$ in $R$ and let $v$ be the earliest vertex on $\hat{\pi}_{xy}$ where $u$ is added to the queue before $x$. If $u$ is dequeued from $Q_v$ before $x$ is enqueued at $Q_v$ then $u$ will not delay $x$ at $v$, but let us assume the worst case situation where every vertex in $R$ that was placed on a queue for a vertex in $\hat{\pi}_{xy}$ is in fact present in that queue when $x$ is added to the queue. In this case $u$ will delay $x$ by one round at $v$. Further, $u$ will proceed along each succeeding vertex on $\hat{\pi}_{xy}$, but if no other instance of $u$ arrives at these succeeding vertices, then $u$ does not delay $x$ further, since both $u$ and $x$ will proceed in a pipelined manner along these vertices.

In the case when other instances of $u$ arrive at some of the succeeding vertices (these arrivals will occur along edges not on $\hat{\pi}_{xy}$), we claim that there is no further delay to $x$: If another copy of $u$ arrives at a vertex $w$ on $\hat{\pi}_{xy}$ later than this first copy of $u$ that $x$ encountered at $v$ then $A_w[u]$ will be marked when this other copy arrives, and it will not be added to the queue and will not impact $x$; if this other copy of $u$ arrives at $w$ earlier than the first copy of $u$, then the first copy of $u$ will not queued, but this other copy will be enqueued even earlier than the latter's arrival, which will cause it to exit ahead of $x$ with the same pipelined property as the first copy.

Thus each distinct vertex that is enqueued at a vertex on $\hat{\pi}_{xy}$ ahead of $x$

can delay $x$ in the traversal $\tau$ by at most one round. Further $x$ will not encounter another copy of $x$ along $\hat{\pi}_{xy}$ since it is a clean path. Hence the message containing $x$ sent along the clean path $\hat{\pi}_{xy}$ reaches $y$ in at most $|R| + |\hat{\pi}_{xy}|$ rounds, giving the desired result. □

**Transitive Closure and SCC.** To obtain at $v$ an array $A_v^+[1..n]$ where all nodes $x$ that are reachable from $v$ are marked, we can repeat this algorithm treating each directed edge as reversed (run the algorithm on the transpose graph of $G$). Then, each vertex knows all vertices in its strongly connected component (scc), and can label it consistently with the minimum ID of the vertices in the scc.

## 7.7 Conclusion

We have presented several distributed algorithms in the CONGEST model for computing BC and path problems in directed graphs. The sub-area of distributed algorithms for directed graphs is still in early development, and our work has presented several new results and techniques. A useful observation highlighted by our research is that global delay techniques can in fact cooperate to improve the efficiency of distributed algorithms for directed graphs.

An important avenue for further research in the CONGEST model is to find $O(n)$-round algorithms for computing APSP and BC in general weighted graphs. We have made a first step in this direction with our $O(n)$ round algorithms for weighted dags, but for general graphs these problems are open for both undirected and directed graphs.

# Bibliography

[ACK17] Ittai Abraham, Shiri Chechik, and Sebastian Krinninger. Fully dynamic all-pairs shortest paths with worst-case update-time revisited. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 440–452, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

[AFL83] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *J. ACM*, 30(3):449–456, July 1983.

[AGvW13] L. Arge, M. T. Goodrich, and F. van Walderveen. Computing betweenness centrality in external memory. In *2013 IEEE International Conference on Big Data*, pages 368–375, Oct 2013.

[AISN91] Giorgio Ausiello, Giuseppe F. Italiano, Alberto Marchetti Spaccamela, and Umberto Nanni. Incremental algorithms for minimal length paths. *J. Algorithms*, 12(4):615–638, December 1991.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BHS07] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms*, 62(2):74 – 92, 2007.

[BKMM07] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *Proc. of the 5th WAW*, pages 124–137, 2007.

[BM84] Hans-Jrgen Bandelt and Henry Martyn Mulder. Interval-regular graphs of diameter two. *Discrete Mathematics*, 50(0):117 – 134, 1984.

[BM15] Elisabetta Bergamini and Henning Meyerhenke. Fully-dynamic approximation of betweenness centrality. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms ESA 2015*, volume 9294 of *Lecture Notes in Computer Science*, pages 155–166. Springer Berlin Heidelberg, 2015.

[BMS15] Elisabetta Bergamini, Henning Meyerhenke, and Christian L. Staudt. Approximating betweenness centrality in large evolving networks. In *Proc. of ALENEX 2015*, chapter 11, pages 133–146. SIAM, 2015.

[Bra01] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[BS07] Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, July 2007.

[CFF13] Salvatore Catanese, Emilio Ferrara, and Giacomo Fiumara. Forensic analysis of phone call networks. *Social Network Analysis and Mining*, 3(1):15–33, 2013.

[CGM04] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, 2004.

[CHKK$^+$15a] Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 143–152, New York, NY, USA, 2015. ACM.

[CHKK$^+$15b] Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic methods in the congested clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 143–152, New York, NY, USA, 2015. ACM.

[CK01] Anne Condon and Richard M. Karp. Algorithms for graph partitioning on the planted partition model. *Random Struct. Algorithms*, 18(2):116–140, March 2001.

[DI04] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[DI06] Camil Demetrescu and Giuseppe F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms*, 2(4):578–601, 2006.

[DSHK$^+$11] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proceedings of the Forty-third Annual ACM Symposium* *on Theory of Computing*, STOC '11, pages 363–372, New York, NY, USA, 2011. ACM.

[Elk06] Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM Journal on Computing*, 36(2):433–456, 2006.

[FG85] A.M. Frieze and G.R. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10(1):57 – 77, 1985.

[FHW12] Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1150–1162, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[FMSN00] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251 – 281, 2000.

[For56] Lester R. Ford. Network Flow Theory. Report P-923, The Rand Corporation, 1956.

[Fre77] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[Gar02] Vijay K. Garg, Ph.D. *Elements of Distributed Computing.* John Wiley & Sons, Inc., New York, NY, USA, 2002.

[GKP98] Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, February 1998.

[GMB12] Oded Green, Robert McColl, and David A. Bader. A fast algorithm for streaming betweenness centrality. In *Proc. of 4th PASSAT*, pages 11–20, 2012.

[GN02] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[GOKK03] K.-I. Goh, E. Oh, B. Kahng, and D. Kim. Betweenness centrality correlation in social networks. *Phys. Rev. E*, 67:017101, 2003.

[GSS08a] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *Proc. of 10th ALENEX*, pages 90–100, 2008.

[GSS08b] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *Proc. of ALENEX 2008*, chapter 8, pages 90–100. SIAM, 2008.

[GU15] Mohsen Ghaffari and Rajan Udwani. Brief announcement: Distributed single-source reachability. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 163–165, New York, NY, USA, 2015. ACM.

[Hag00] Torben Hagerup. Improved shortest paths on the word ram. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, ICALP '00, pages 61–72, London, UK, UK, 2000. Springer-Verlag.

[HFA$^+$16] Q. S. Hua, H. Fan, M. Ai, L. Qian, Y. Li, X. Shi, and H. Jin. Nearly optimal distributed algorithm for computing betweenness centrality.

In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 271–280, June 2016.

[HFQ$^+$16] Qiang-Sheng Hua, Haoqiang Fan, Lixiang Qian, Ming Ai, Yangyang Li, Xuanhua Shi, and Hai Jin. Brief announcement: A tight distributed algorithm for all pairs shortest paths and applications. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 439–441, New York, NY, USA, 2016. ACM.

[HKN16] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 489–498, New York, NY, USA, 2016. ACM.

[HKYH02] Petter Holme, Beom Jun Kim, Chang No Yoon, and Seung Kee Han. Attack vulnerability of complex networks. *Phys. Rev. E*, 65:056109, 2002.

[HW12] Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 355–364, New York, NY, USA, 2012. ACM.

[HZ85] Refael Hassin and Eitan Zemel. On shortest paths in graphs with random weights. *Mathematics of Operations Research*, 10(4):557 – 564, 1985.

[KAS$^+$12] Nicolas Kourtellis, Tharaka Alahakoon, Ramanuja Simha, Adriana

Iamnitchi, and Rahul Tripathi. Identifying high betweenness centrality nodes in large social networks. *SNAM*, pages 1–16, 2012.

[KHP+07] Maksim Kitsak, Shlomo Havlin, Gerald Paul, Massimo Riccaboni, Fabio Pammolli, and H. Eugene Stanley. Betweenness centrality of fractal and nonfractal scale-free model networks and tests on real networks. *Phys. Rev. E*, 75:056115, 2007.

[KI12] Nicolas Kourtellis and Adriana Iamnitchi. Leveraging peer centrality in the design of socially-informed peer-to-peer systems. *CoRR*, abs/1210.6052, 2012.

[Kin99] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 81–, Washington, DC, USA, 1999. IEEE Computer Society.

[KKP93] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993.

[KMB14] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi. Scalable online betweenness centrality in evolving graphs. *CoRR*, abs/1401.6981, 2014.

[Kre02] Valdis Krebs. Mapping networks of terrorist cells. *CONNECTIONS*, 24(3):43–52, 2002.

[KWCC13] Miray Kas, Matthew Wachs, Kathleen M. Carley, and L. Richard Carley. Incremental algorithm for updating betweenness centrality in dynamically growing networks. In *Proc. of ASONAM*, 2013.

[Ley07] Loet Leydesdorff. Betweenness centrality as an indicator of the interdisciplinarity of scientific journals. *J. Am. Soc. Inf. Sci. Technol.*, 58(9):1303–1319, July 2007.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY, USA, 2014. ACM.

[LLP$^+$12] Min-Joong Lee, Jungmin Lee, Jaimie Yejean Park, Ryan Hyun Choi, and Chin-Wan Chung. Qube: a quick algorithm for updating betweenness centrality. In *Proc. 21st WWW Conference*, pages 351–360, 2012.

[LP13] Christoph Lenzen and David Peleg. Efficient distributed source detection with limited bandwidth. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 375–382, New York, NY, USA, 2013. ACM.

[LPS13a] Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 381–390, New York, NY, USA, 2013. ACM.

[LPS13b] Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 381–390, New York, NY, USA, 2013. ACM.

[LR89] Michael Luby and Prabhakar Ragde. A bidirectional shortest-path algorithm with good average-case behavior. *Algorithmica*, 4(1-4):551–567, 1989.

[Lyn96a] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Lyn96b] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[McG95] Catherine C. McGeoch. All-pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995.

[MEJ$^+$09] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS*, pages 1–8. IEEE, 2009.

[MK12] LeandrosA. Maglaras and Dimitrios Katsaros. New measures for characterizing the significance of nodes in wireless ad hoc networks via localized path-based neighborhood analysis. *Social Network Analysis and Mining*, 2(2):97–106, 2012.

[Mul82] Henry Martyn Mulder. Interval-regular graphs. *Discrete Mathematics*, 41(3):253 – 269, 1982.

[MV13] Fragkiskos D. Malliaros and Michalis Vazirgiannis. Clustering and community detection in directed networks: A survey. *Physics Reports*, 533(4):95 – 142, 2013. Clustering and Community Detection in Directed Networks: A Survey.

[Nan14] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the 46th Annual ACM*

*Symposium on Theory of Computing*, STOC '14, pages 565–573, New York, NY, USA, 2014. ACM.

[NPR14a] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Betweenness centrality  incremental and faster. In *MFCS 2014*, volume 8635 of *LNCS*, pages 577–588. Springer, 2014.

[NPR14b] Meghana Nasre, Matteo Pontecorvi, and Vijaya Ramachandran. Decremental all-pairs all shortest paths and betweenness centrality. In *ISAAC 2014*, volume 8889 of *LNCS*, pages 766–778. Springer, 2014.

[PAE$^{+}$13] Rami Puzis, Yaniv Altshuler, Yuval Elovici, Shlomo Bekhor, Yoram Shiftan, and Alex (Sandy) Pentland. Augmented betweenness centrality for environmentally aware traffic monitoring in transportation networks. *J. of Intell. Transpor. Syst.*, 17(1):91–105, 2013.

[Pel] David Peleg. Private communication, april 26, 2017. .

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[Pet04] Seth Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47 – 74, 2004.

[PMW05] John W. Pinney, Gleen A. McConkey, and David R. Westhead. Decomposition of biological networks using betweenness centrality. In *Proc. of 9th RECOMB*, 2005.

[PR00] David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM J. Comput.*, 30(5):1427–1442, May 2000.

[PR05] Seth Pettie and Vijaya Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, 34(6):1398–1431, 2005.

[PR15a] Matteo Pontecorvi and Vijaya Ramachandran. A faster algorithm for fully dynamic betweenness centrality. http://arxiv.org/abs/1506.05783, 2015.

[PR15b] Matteo Pontecorvi and Vijaya Ramachandran. Fully dynamic betweenness centrality. In *Algorithms and Computation - ISAAC 2015*, volume 9472 of *LNCS*, page to appear. Springer, 2015.

[PRT12] David Peleg, Liam Roditty, and Elad Tal. *Distributed Algorithms for Network Diameter and Girth*, pages 660–672. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[QH10] Daniele Quercia and Stephen Hailes. Sybil attacks against mobile users: Friends and foes to the rescue. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 336–340, Piscataway, NJ, USA, 2010. IEEE Press.

[Ram04] Ramírez. The social networks of academic performance in a student context of poverty in Mexico. *Social Networks*, 26(2):175–188, 2004.

[RK14] Matteo Riondato and Evgenios M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. In *Proc. of the 7th ACM WSDM*, pages 413–422. ACM, 2014.

[RMRR98] J. Sicilia R. M. Ramos and M. T. Ramos. A generalization of geodetic graphs: K-geodetic graphs. *Inverstigacin Operativa*, 1:85–101, 1998.

[Sch07] Satu Elisa Schaeffer. Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, August 2007.

[SG05] Brajendra K. Singh and Neelima Gupte. Congestion and decongestion in a communication network. *Phys. Rev. E*, 71:055103, 2005.

[SGIS13] Rishi Ranjan Singh, Keshav Goel, Sudarshan Iyengar, and Sukrit. A faster algorithm to update betweenness centrality after node alteration. In *Proc. of 10th WAW*, 2013.

[Shi53] Alfonso Shimbel. Structural parameters of communication networks. *The bulletin of mathematical biophysics*, 15(4):501–507, 1953.

[SOA88] N. Srinivasan, J. Opatrny, and V.S. Alagar. Bigeodetic graphs. *Graphs and Combinatorics*, 4(1):379–392, 1988.

[Suo13] Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24:1–24:40, March 2013.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[Tho99] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, May 1999.

[Tho04] Mikkel Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT*, pages 384–396, 2004.

[Tho05] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 112–119, New York, NY, USA, 2005. ACM.

[WT13] W. Wang and C. Y. Tang. Distributed computation of node and edge betweenness on tree graphs. In *52nd IEEE Conference on Decision and Control*, pages 43–48, Dec 2013.

[YTQ17] K. You, R. Tempo, and L. Qiu. Distributed algorithms for computation of centrality measures in complex networks. *IEEE Transactions on Automatic Control*, 62(5):2080–2094, 2017.

[ZWZC11] Yuanyuan Zhang, Xuesong Wang, Peng Zeng, and Xiaohong Chen. Centrality characteristics of road network patterns of traffic analysis zones. *Transportation Research Record: Journal of the Transportation Research Board*, 2256:16–24, 2011.

# Vita

Matteo Pontecorvi was born in Velletri, Italia. After completing his high school in Velletri, he attended the University of Rome 'La Sapienza'. He received a Master Degree in Computer Science from La Sapienza in 2010. He entered the graduate school at the University of Texas at Austin in September 2011.

Permanent Address: cavia@cs.utexas.edu

This dissertation was typeset with $\LaTeX\,2_\varepsilon$[1] by the author.

[1] $\LaTeX\,2_\varepsilon$ is an extension of $\LaTeX$. $\LaTeX$ is a collection of macros for $\TeX$. $\TeX$ is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.