International Journal of Mathematics and
Computer Science, **15**(2020), no. 1, 347–365

An All Pairs Shortest Path Algorithm for Dynamic Graphs

Muteb Alshammari[1], **Abdelmounaam Rezgui**[2]

[1]Department of Computer Science & Engineering
New Mexico Tech
Socorro, NM, 87801, USA

[2]School of Information Technology
Illinois State University
Normal, IL 61790-5150, USA

email: mutebalshammari@cs.nmt.edu

(Received October 12, 2019, Accepted November 23, 2019)

Abstract

Graphs are mathematical structures used in many applications. In recent years, many applications emerged that require the processing of large *dynamic* graphs where the graph's structure and properties change constantly over time. Examples include social networks, communication networks, transportation networks, etc. One of the most challenging problems in large scale dynamic graphs is the all-pairs, shortest path (APSP) problem. Traditional solutions (based on Dijkstra's algorithms) to the APSP problem do not scale to large dynamic graphs with a high change frequency. In this paper, we propose an efficient APSP algorithm for large sparse dynamic graphs. We first present our algorithm and then we give an analytical evaluation of the proposed solution. We also present a thorough experimental study of our solution and compare it to two of the best known algorithms in the literature.

Key words and phrases: Dynamic graphs, shortest paths, APSP.
AMS (MOS) Subject Classifications: 05C85, 68R10, 68R05.
ISSN 1814-0432, 2020, http://ijmcs.future-in-tech.net

1 Introduction

Many real life problems can be solved using graphs as their underlying data structure. For example, finding the shortest distance between two points is interpreted as a shortest path problem in graphs. Graphs are used in many applications such as transportation, protein interaction networks, and data mining [20, 18, 19].

Traditionally, graphs have been viewed as static data structures. A static graph is a graph that does not change over time. However, recent applications have led to the emergence of problems that can only be solved using graphs that change over time, i.e., *dynamic* graphs. Dynamic graphs are subject to changes in their underlying data structures. Dynamic graphs can be either partially or fully dynamic. Partially dynamic graphs support either (i) insertion of vertices and edges and decreasing edge weights (i.e., incremental) or (ii) deletion of vertices and edges and increasing edge weights (i.e., decremental). Fully dynamic graphs support both incremental and decremental operations.

One of the most challenging problems in dynamic graphs is the *all pairs shortest path* (APSP) problem. In this problem, we need to maintain the shortest paths and distances between each pair of vertices in a fully dynamic graph subject to update operations. In this paper, we propose an efficient APSP algorithm for large sparse dynamic graphs. We evaluate our solutions through an analytical analysis and a thorough experimental study where we compare the proposed solution to two well-known algorithms in the literature related to dynamic graphs.

This paper is organized as follows. In Section 2, we describe our model of dynamic graphs and the used data structures. In Section 3, we provide an overview of related work. In Sections 4 and 5, we present our approach including algorithms and complexity analysis. In Section 6, we give details about our experimental evaluation and discuss our results. Section 7 concludes the paper.

2 Model for Dynamic Graphs

Let $G = \{V, E, W\}$ be a dynamic directed weighted graph (digraph) where: V is a finite set of vertices of size $n = |V|$, $E \subseteq V \times V$ is a finite set of weighted edges of cardinality $m = |E|$, and W is a weight function such that $W(x, y)$ returns a real weight of edge (x, y) (where $x, y \in V$, $(x, y) \in E$, and $x \neq y$). We assume that the graph G has no loops and no negative cycles.

For each vertex $x \in V$, $in(x)$ is the set of edges ending at vertex x and $out(x)$ is a set of edges starting at vertex x. The graph G supports the following operations:

- $Insert\_vertex(x)$ where $x \notin V$.
- $Delete\_vertex(x)$ where $x \in V$.
- $Insert\_edge(x, y, w)$ where $x, y \in V$, $(x, y) \notin E$, and $x \neq y$.
- $Delete\_edge(x, y)$ where $x, y \in V$ and $(x, y) \in E$.
- $Increase\_weight(x, y, w)$ where $x, y \in V$, $(x, y) \in E$, and $W(x, y) < w < \infty$.
- $Decrease\_weight(x, y, w)$ where $x, y \in V$, $(x, y) \in E$, $w < W(x, y)$.

Inserting a vertex x in the graph has no effect on existing shortest paths since x is not connected to any vertex yet. For the second operation, we interpret deleting a vertex x as deleting one edge at a time (from $in(x) \cup out(x)$) until x has no more edges at which point x is removed from the graph.

Let T be a forest of trees in which each tree t represents the single source shortest path tree of a vertex. For any vertex $s \in V$, $T(s)$ is a shortest path tree (SPT) of vertex s where s is the source vertex of $T(s)$ (see Figure 1). We denote by $t.source$ the source vertex of the SPT t where $t \in T$.

$\mathbf{T} =$

$\mathbf{V_1}$	$\mathbf{V_2}$	. . . .	$\mathbf{V_n}$
$\mathbf{t(V_1)}$	$\mathbf{t(V_2)}$	. . . .	$\mathbf{t(V_n)}$

Figure 1: The forest T. $t(v_i)$ is a SPT rooted at vertex v_i for $i = 1, 2, \ldots, n$.

$t.d(x)$ is the shortest distance from $t.source$ to x before the update operation. We denote the new distance of x after an update operation as $t.d'(x)$. Similarly, $t.P(x)$ is the parent of x in t. Table 1 is a summary of the main notations that we will use throughout this paper.

For each update operation, we first update the graph G. Then, for each tree t in the forest T, we call the corresponding algorithm of that operation to maintain the trees in T.

Table 1: Summary of notations

$W(x, y)$	The weight of the edge (x, y).
T	A forest of trees.
t or $T(s)$	A shortest path tree (SPT).
$t.source$	The source vertex of the SPT t where $t \in T$.
$t.d(x)$	The shortest distance from $t.source$ to x before the update operation.
$t.P(x)$	The parent of x in t.
$in(x)$	A set of incoming edges to x.
$out(x)$	A set of outgoing edges from x.

3 Related Work

Several algorithms have been proposed in the literature to maintain shortest paths in dynamic graphs (e.g., [8, 6, 7, 5, 1, 3]). In [6], Ramalingam and Reps proposed the first fully dynamic algorithm to maintain shortest paths in dynamic graphs. Their algorithm runs in $O(||\delta|| + |\delta| \log |\delta|)$ in the case of single-source shortest paths (SSSP) where $|\delta|$ is the number of vertices affected by the update operation[1] and $||\delta||$ is the number of edges connected to at least one affected vertex. The authors in [2] used the results of [6] to develop an APSP algorithm that runs in $O(mn + n^2 \log n)$ in the worst case. The proposed algorithm performs a high number of edge scans in which it re-scans incoming edges of every affected vertex in the case of decremental operations. In [7], the same authors proposed another set of algorithms with a similar time complexity. In their later work, they use a directed-cyclic graph (called SP) to maintain the shortest paths and their algorithms requires a high number of edge scans in the case of incremental operations leading to a high computational cost.

In [8], Ausiello et al. proposed an incremental algorithm (insertion and decreasing the weight of edges) for the dynamic APSP in digraphs with positive integer edge weights less than C. The algorithm runs in $O(Cn^3 \log(Cn))$ for any sequence of at most $O(n^2)$ (resp. $O(Cn^2)$) of edge insertions (resp. weights decreases). Note that the proposed algorithm does not support decremental operations (i.e., deleting or increasing the weight of edges). Moreover, the authors proposed a technique to detect the affected trees that need to be updated. The technique consists of maintaining two shortest path trees (backward and forward trees) for each vertex which results in more compu-

[1] An affected vertex is a vertex that changes its distance, parent, or both as a result of a given update.

tations.

In [5], King proposed a set of fully dynamic algorithms for digraphs with positive integer edge weights less than b. The algorithms run in $O(n^{2.5}\sqrt{b \log n})$ amortized time and maintain a tree of depth b for each vertex in a forest of depth nb. The proposed algorithms maintain two shortest path trees (IN and OUT) for every vertex up to depth d. Practically, the proposed algorithms are limited to a small range of weights [2].

In [1], Demetrescu and Italiano proposed a fully dynamic algorithm for digraphs with positive real edge weights. Their algorithm runs in $O(n^2 \log^3 n)$ amortized time per operation for any sequence of operations. In [3], the same authors proposed another algorithm for the dynamic APSP in digraphs with positive and negative real edge weights and S different values of each edge weight. The algorithm runs in $O(n^{2.5}\sqrt{S \log^3 n})$. In [22], Thorup extended the work of Demetrescu and Italiano in [1] to include graphs with negative edge weights in $O(n^2)$ amortized time. In a later work, Thorup presented a technique in [23] and show how to maintain APSP in $O(n^{2.75})$ by de-amortization of [1]. Both algorithms of Thorup are designed for theoretical interest.

Recently, another version of the APSP problem has emerged that consists of maintaining the *approximate* shortest path (ASP). In the ASP, the reported distances have an error factor of f. A main objective of ASP is to process distributed large graphs. Several algorithms have been proposed to solve this new version of the problem (e.g., see [12, 13, 10, 9, 11, 24]). In this paper, we are interested in the *exact* APSP problem in which we report the exact distances and paths. We propose a set of algorithms that overcome the limitations of the previous algorithms.

4 Our Approach

In this section, we show our approach to maintain the APSP in a fully dynamic graph. The proposed approach consists of two algorithms. The first algorithm (Section 4.1) is an *incremental* dynamic algorithm that maintains the APSP after inserting an edge or decreasing an edge weight. The second algorithm (Section 4.2) is a *decremental* dynamic algorithm that maintains the APSP after deleting an edge or increasing an edge weight.

For simplicity, we do not consider trivial operations such as deleting a non-existing edge and updating an edge weight with the same previous weight. Also, we interpret the *Delete_edge*() operation as increasing the edge weight

to $+\infty$.

To maintain the APSP in a fully dynamic graph, we maintain a forest of shortest path trees (SPTs) in which we have an SPT for each vertex in the graph G. Each SPT (say $T(s)$ for any s where $s \in V$) is rooted at vertex s where s is the source vertex of the corresponding tree $T(s)$. For each SPT, our algorithms maintain the shortest paths and distances from the source vertex in this tree to every other vertex independently from other trees.

Algorithm 1 Updating edge (x, y)

```
1: procedure UPDATE(x, y, w, T )                ▷ w is the new edge weight
2:     if w = ∞ then
3:         Delete_edge(x, y)
4:         for every t ∈ T do
5:             Decremental_Algorithm(x, y, ∞, t)
6:         end for
7:     else if W(x, y) < w then
8:         Δ = w − W(x, y)
9:         W(x, y) = w
10:        for every t ∈ T do
11:            Decremental_Algorithm(x, y, Δ, t)
12:        end for
13:    else
14:        if (x, y) ∉ E then
15:            Insert_edge(x, y, w)
16:        else
17:            W(x, y) = w
18:        end if
19:        for every t ∈ T do
20:            Incremental_Algorithm(x, y, t)
21:        end for
22:    end if
23: end procedure
```

After an edge operation in G, we apply the corresponding algorithm (either *incremental* or *decremental* dynamic algorithm) for every SPT in T as shown in procedure UPDATE*(x,y,w,T)* (see Algorithm 1). If the operation affects any shortest path in any SPT, the corresponding algorithm updates the shortest paths and distances of that SPT. In our algorithms, we use a min-heap, denoted H, to store and lookup vertices based on their distances

from the source of the current tree.

In the following two sections, we present our algorithms. Then, we conduct a complexity analysis of the proposed algorithms.

4.1 Incremental Dynamic Algorithm

We now present the first algorithm that is the *incremental* dynamic algorithm (see Algorithm 2). The algorithm maintains the shortest paths and distances from a source vertex (which is the root vertex of the SPT t) to every other vertex. The algorithm supports both inserting an edge and decreasing an edge weight. The algorithm reconstructs the sub-tree $t(y)$ after an edge operation by using an adaption of Dijkstra's algorithm. The algorithm starts by fixing the first affected vertex y (i.e. the endpoint of the modified edge (x, y)). Then, the algorithm fixes the other affected vertices by walking down the sub-tree $t(y)$. Note that $t(y) \subseteq t(s)$ where s is the source of the current tree t.

The algorithm receives three parameters:

1. x: the source vertex of the affected edge.
2. y: the target vertex of the affected edge.
3. t: a SPT rooted at a vertex (any vertex).

The algorithm is divided into three phases. The first phase is to find the effect of the edge (x, y) on the SPT t. If (x, y) does not affect t, i.e., does not improve the distance of y in t (Line 2) then the algorithm will stop. Otherwise, the algorithm proceeds to phases 2 and 3 and updates t.

In the second phase, the algorithm updates the distance and parent of y since the edge (x, y) can improve the distance of y in t. Then, the algorithm inserts y into H.

The third phase is the essential phase in which the algorithm updates affected vertices to maintain SPT t. This phase is an adaptation of Dijkstra's algorithm. It uses a loop over H. Initially, H contains the endpoint of the updated edge (i.e., y). Affected vertices (other than y, if any) are inserted into H and extracted sequentially during the updates of affected vertices. At each iteration, an affected vertex (say u) is extracted from H and its successors (i.e., the neighbors in $out(u)$) are inspected for possible improvements. If a successor of u is affected (distance improvement), then this successor is updated (its distance and parent pointer) and inserted into H. Finally, the

Algorithm 2 Inserting or Decreasing the weight of edge (x, y)

```
1: procedure INCREMENTAL_ALGORITHM(x, y, t)
Phase 1:
2:     if t.d(y) < t.d(x) + W(x, y) then
3:         Stop                          ▷ no improvement can be accomplished
4:     end if
Phase 2:
5:     t.P(y) ← x
6:     t.d(y) ← t.d(x) + W(x, y)
7:     insert(H, y, d(y))                ▷ H is a min-heap
Phase 3:
8:     while H ≠ ∅ do
9:         u ← extract_min(H)
10:        for every v ∈ out(u) in G do
11:            if t.d(u) + W(u, v) < t.d(v) then
12:                t.P(v) ← u
13:                t.d(v) ← t.d(u) + W(u, v)
14:                insert(H, v, t.d(v))
15:            end if
16:        end for
17:    end while
18: end procedure
```

algorithm terminates only when H is empty and when the shortest paths and distances in t are updated correctly.

4.2 Decremental Dynamic Algorithm

We now present our *decremental* dynamic algorithm (see Algorithm 3). The algorithm maintains the shortest paths and distances from a source vertex (i.e., $t.source$) to every other vertex. The algorithm supports both deleting an edge and increasing edge weight. In contrast to several other algorithms that use different techniques to detect affected vertices in the sub-tree $t(y)$ (for example, coloring technique in [16]), this algorithm increases the distances of vertices in the sub-tree $t(y)$ assuming that they do not have alternative paths with the same old distances. This technique is used to guarantee that a vertex (after a decremental edge operation) will not have a parent that was in its sub-tree. As we will see in our experimental study, this technique shows its effects in sparse graphs where more likely there are view alternative paths.

We use a function called $pred\_min(u)$ that returns the best parent of u or *null* if there is no such parent. The $pred\_min(u)$ function is defined as follows:

$$pred\_min(u) = \begin{cases} \min_{v \in in(u)}(d(v) + W(v,u)) & if\ (v,u) \in E\ and\ d(v) \neq \infty \\ null & otherwise \end{cases}$$

The algorithm receives four parameters:

1. x: the source vertex of the affected edge.
2. y: the target vertex of the affected edge.
3. Δ: the difference between the old and new weight of the edge (x, y) or ∞ if the operation is to delete (x, y).
4. t: a SPT rooted at a vertex (any vertex).

The algorithm is divided into three phases. In the first phase, the algorithm terminates if the edge (x, y) does not affect shortest paths and distances of t (Line 2). If the edge (x, y) is not a tree edge in t, then the current SPT (i.e., t) will not be affected. Otherwise, the algorithm proceeds to the second phase.

In the second phase and based on the type of the update operation, the algorithm increases the distances of vertices in the sub-tree rooted at y (i.e.,

Algorithm 3 Deleting or Increasing the weight of edge (x, y)

1: **procedure** DECREMENTAL_ALGORITHM (x, y, Δ, t)

Phase 1:

2: **if** $(x, y) \notin t$ **then**
3: Stop ▷ this edge has no effect on t
4: **end if**

Phase 2:

5: **if** $\Delta \neq \infty$ **then** ▷ for increasing edge weights operations
6: **for** every v *s.t.* $v \in t(y)$ **do**
7: $t.d(v) \leftarrow t.d(v) + \Delta$
8: **end for**
9: **else** ▷ for deleting edges operations
10: **for** every v *s.t.* $v \in t(y)$ **do**
11: $t.d(v) \leftarrow \infty$
12: **end for**
13: **end if**
14: $insert(H, y, 0)$ ▷ H is a min-heap

Phase 3:

15: **while** $(H \neq \emptyset)$ **do**
16: $u \leftarrow extract\_min(H)$
17: $p \leftarrow pred\_min(u)$
18: **if** (p) **then**
19: $t.P(u) \leftarrow p$
20: $t.d(u) = W(p, u) + t.d(p)$
21: **end if**
22: **for** every $v \in out(u)$ **do**
23: **if** ($t.d(u) + W(u, v) <= t.d(v)$) **or** ($(u, v) \in t$) **then**
24: $k \leftarrow t.d(u) + W(u, v)$
25: $insert(H, v, k)$
26: **end if**
27: **end for**
28: **end while**
29: **end procedure**

$t(y)$) either (a) by Δ if the operation is increasing the weight of edge (x, y) or (b) by ∞ if the operation is deleting (x, y). Finally, y is inserted in H and is processed in the third phase.

The third phase is the essential phase in which the algorithm updates affected vertices to maintain SPT t. This phase is an adaptation of Dijkstra's algorithm. It uses a loop over H. Initially, H contains the endpoint of the updated edge (i.e., y). Affected vertices (other than y, if any) are inserted into H and extracted sequentially during the updates of affected vertices. This phase starts by extracting a vertex u from H with the minimum key and then finds the best parent (p) with the minimum distance (Lines 16 and 17). The distance ($t.d(u)$) and parent pointer ($P(u)$) are updated using p (Lines 18-20). Finally, every successor of u (say v) is evaluated and inserted into H if either: (i) the distance of v can be improved, or (ii) u was the parent to v before the operation or u is a possible parent to v (Lines 23-25). This will ensure that all vertices in the sub-tree $t(y)$ are scanned and updated and possibly have new shortest paths that do not involve x or y. Note that $t(y)$ is a sub-tree of t (i.e., $t(y) \subseteq t$).

5 Complexity Analysis

Time complexity analysis is a challenge in dynamic graphs where modifications could change only a subset of the previously computed results. When considering the classical worst-time analysis, many dynamic graphs algorithms that have been proposed in the literature have no better time than recomputing from scratch [7, 23]. Therefore, many papers used different ways to analyze the time complexity of dynamic graphs more accurately. For example, [5, 1] used the *amortized analysis* of a sequence of operations, [6, 7, 15] used the notion of *output complexity* (explained next), and [14] used the notion of *k-bounded accounting function*.

In this paper, we extend the model introduced by Ramalingam and Reps in [7]. In contrast to the classical worst-time complexity analysis in which the complexity is analyzed in terms of the input size, this model computes the difference between the input and the output (i.e. the changes of input and output). In this model:

- δ is a list of affected vertices in all trees (with duplicates).
- $|\delta|$ is the number of vertices in δ. i.e. the sum of the number of affected vertices in all trees (at most n^2).

- $||\delta||$ is $|\delta|$ + number of edges connected to each of those affected vertices in δ (in each tree). For example, if vertex u is the only affected vertex in all trees and u has 3 edges, then $|\delta|$ = n (because we have n trees), and $||\delta||$ =n + (3*n) = 4n.

We assume the use of a Fibonacci heap where the cost of inserting n elements is $O(\log(n))$ amortized time (per operation) and, for the rest of the heap's operations, the cost is $O(1)$ amortized time.

For the incremental algorithm, it is easy to see that the total cost over all trees is $O(||\delta|| + |\delta| \times \log|\delta|)$. The cost comes from the loop in lines (8 - 17) which iterates over affected vertices in δ. The cost of this loop including the heap operations is $O(|\delta| \times \log|\delta|)$. For each affected vertex in the current passed tree t (passed in line 1), the loop in lines (10 - 16) iterates over the edges that are connected to this affected vertex. The total cost of this loop over all trees is $O(||\delta||)$

The same cost holds also for the decremental algorithm with the consideration that the decremental algorithm scans incoming (using $pred\_min(u)$ in line 17) and outgoing edges for every affected vertex while the incremental algorithm scans only the outgoing edges for affected vertices.

The total cost of maintaining the APSP in a fully dynamic graph after a single update operation is $O(||\delta|| + |\delta| \times \log|\delta|)$.

Considering the space complexity, our approach requires $O(n^2)$ as follows:

- The graph G requires $O(n)$ for vertices and at most $O(n^2)$ for edges.
- The forest T requires $O(n^2)$ since each tree requires $O(n)$.

Therefore, the overall space required is $O(n^2)$.

6 Experiments

In this section, we evaluate our approach in contrast to other well-known approaches in the literature. We conducted a thorough experimental study based on a previous study by Demetrescu and Italiano in [2]. The original experiment was a comparison of three dynamic algorithms and two static algorithms. The dynamic algorithms are: (i) the algorithm of Ramalingam and Reps (RR) [6], (ii) the algorithm of Demetrescu and Italiano (DI) [1], and (iii) the algorithm of King [5].

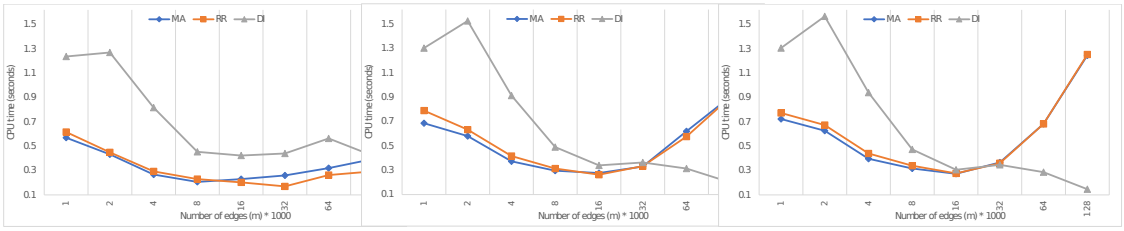
In this work, we are interested in comparing the dynamic algorithms implemented in [2] to our algorithm (MA). Since the algorithm of King was

Table 2: Data sets details

Data set name	Number of vertices (n)	Number of edges (m)	density
Synthetic	500	1000 - 128000	0.004 - 0.51
Synthetic	10,000	100,000 - 500,000	0.001 - 0.005
Road networks	342 - 998	944 - 3240	0.0031 - 0.0088
Internet Autonomous Systems	2000 - 4500	9264 - 20094	0.001 - 0.002

shown to be the slowest among the three algorithms (as we also noticed in our own evaluation), we will no longer discuss King's algorithm in the remainder of this paper. We also did not consider the algorithm of Thorup in both [22, 23] because the objective of both algorithms was for theoretical interest and not for practical uses.

We extend the experiment in [2] by implementing our algorithms using the same criteria and standard used in the original work. Every reported result in this paper is an average of three separate (independent) runs.



(a) Weights in the range of $(1-10)$

(b) Weights in the range of $(1-100)$

(c) Weights in the range $(1-1000)$

Figure 2: Experiments on synthetic data of size $n = 500$ and different edge weights range.

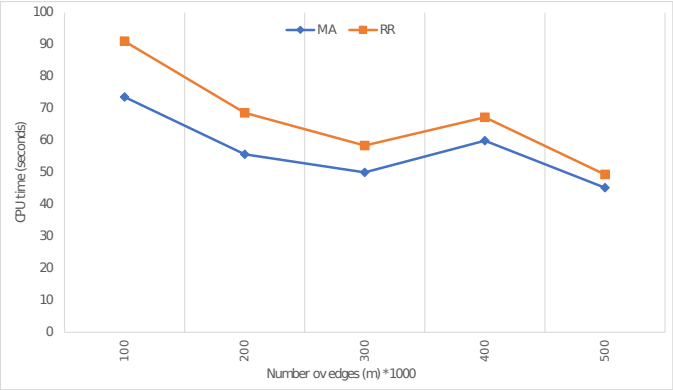


Figure 3: Experiments on synthetic data of size $n = 10,000$ and edges in the range (100,000 - 500,000)

6.1 Data Sets

We consider two types of data sets: (i) synthetic and (ii) semi-real data. The first data set is the *synthetic* data set in which both the graph and its operations are generated randomly based on the user's request of the graph size (number of nodes), number of edges, range of edge weights, and number of operations.For this data set, we conduct four experiments. The first three are on graphs of size 500 vertices with increasing edge density and different edge weight range $((1-10)$, $(1-100)$, and $(1-1000))$. The reason behind this is to test the effect of different edge weights on the running time in a practical environment. The fourth experiment is on a larger graph of size 10,000 vertices with increasing edge density. The main objective of the later experiment is to test our approach on a larger size of graphs. We apply a sequence of 1000 inter-mixed operations (increasing and decreasing weight operations). We generate the graphs and their operations using the generator in the original experiment [2].

The second data set is the *semi-real* data set in which the graphs are from the real world but the operations are randomly generated. The data set includes (i) graphs of autonomous systems connections (AS), and (ii) US road networks. This data set was obtained from [2]. For each graph, we apply 1000 inter-mixed operations. Details of both data sets are given in Table 2.

6.1.1 Synthetic Data Set

Our experiments on synthetic data showed that, compared to RR and DI, MA has the best performance on sparse graphs of all considered weight ranges whereas DI has the best performance on denser graphs with higher weight ranges as shown in Figure 2(b and c). One possible explanation is that smaller weight ranges may result in more alternative paths to a subset of nodes. We found that the range of edge weights has a considerable effect on DI and less impact on MA and RR as shown in Figure 2 (a,b, and c).

Furthermore, we carried another experiments on larger graphs of 10,000 vertices and 100,000 to 500,000 edges. We found that DI algorithm was not applicable in large graphs since it requires very large space complexity. In the experiments of large graphs as shown in Figure 3, MA performance improves over RR compared to smaller graphs and show the applicability over other tested algorithms when considering larger sparse graphs.

6.1.2 Semi-Real Data Set

Our experiments on road networks showed that MA achieves the best performance compared to RR and DI as shown in Figure 4. On average, MA is 10% faster than RR and 68% faster than DI. RR is 53% faster than DI. MA had the best performance on road networks because they are known to be sparse graphs.

We note that the authors in [2, 1] proposed an optional technique, called smoothing, used in DI to reduce the space complexity after an update operation (see [1] for details). However, smoothing can affect the running time. We did not notice any major effect of smoothing in the synthetic and road network data sets, but the effect was clear when we used the AS data set.

Considering AS networks, Figure 5.a shows the performance of DI when we apply full smoothing and Figure 5.b shows its performance without smoothing. In case of no smoothing, MA is 4.2% and 5.4% faster than RR and DI, respectively. In the case of full smoothing, MA is 4% and 109% faster than RR and DI, respectively.

6.2 Final Discussion

In this section, we discuss some final thoughts regarding the applicability of the considered algorithms. We found that DI is a very efficient algorithm that deals with graphs of high density since the main objective of DI is to reduce the number of scanned edges after an operation in a dynamic graph.

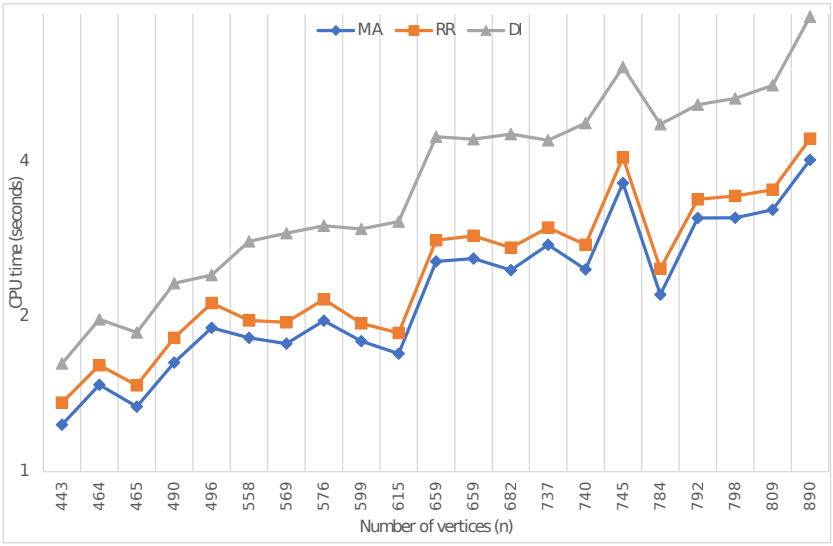


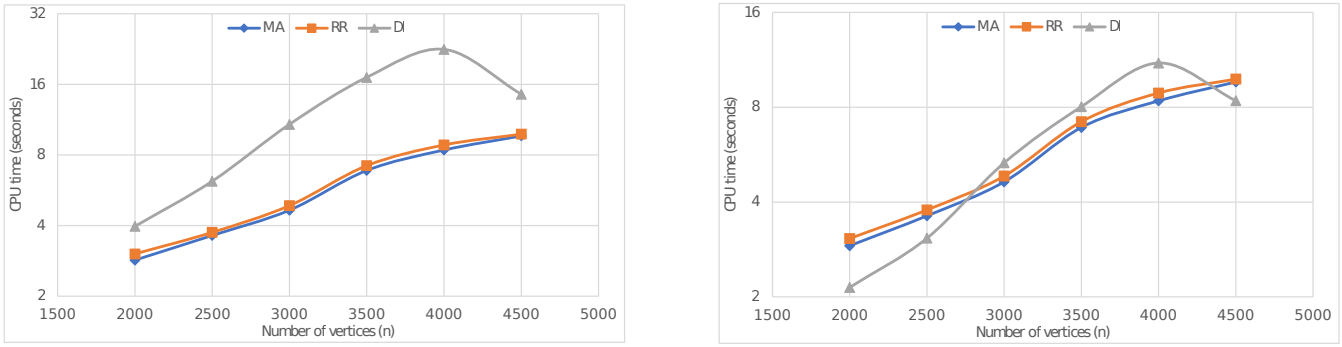
Figure 4: Experiments on road networks

For this reason, DI was very slow (in average) when tested in sparse graphs (e.g., the left half of Figures 2.a, 2.b, and 2.c).

The RR algorithm is another variant of Dijkstra's (different than ours) that uses the notion of consistency status of vertices (see [6] for details). Our study of RR shows that the algorithm re-scans affected vertices multiple times when it encounters decremental operations. Moreover, RR scans the incoming edges to affected vertices more often: (i) first to determine the consistency status and (ii) second to update final distances. This adds up to affect the total running time.

7 Conclusion

Dynamic graphs are important data structures in modeling and solving challenging problems in modern life. Examples of applications where dynamic graphs may be used include ground transportation, aviation, and communication networks. A challenging problem in dynamic graphs is maintaining the set of all-pairs shortest paths (APSP). In this paper, we present a novel, efficient approach to solve the APSP problem in fully dynamic graphs. We presented our algorithms and analyzed their time complexity. We also conducted an extensive experimental study on synthetic and real data sets. Our experiments on both types of data showed that our algorithms are more



(a) Full smoothing

(b) No smoothing

Figure 5: Experiments on AS: (a) with full smoothing and (b) with no smoothing.

efficient than existing well-known algorithms in the case of sparse graphs.

References

[1] Camil Demetrescu, Giuseppe Italiano, FA new approach to dynamic all pairs shortest paths, Journal of the ACM, **51**, no. 6, (2004), 968–992.

[2] Camil Demetrescu, Giuseppe Italiano, Experimental analysis of dynamic all pairs shortest path algorithms, ACM Transactions on Algorithms (TALG), **2**, no. 4, (2006), 578–601.

[3] Camil Demetrescu, Giuseppe Italiano, Fully dynamic all pairs shortest paths with real edge weights, Journal of Computer and System Sciences, **72**, no. 5, (2006), 813–837.

[4] Camil Demetrescu, Daniele Frigioni, Alberto Marchetti-Spaccamela, Umberto Nanni, Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study, International Workshop on Algorithm Engineering, (2000), 218–229.

[5] Valerie King, Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs, 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039), (1999), 81–89.

[6] Ganesan Ramalingam, Thomas Reps, An incremental algorithm for a generalization of the shortest-path problem, Journal of Algorithms, **21**, no. 2, (1996), 267–305.

[7] Ganesan Ramalingam, Thomas Reps, On the computational complexity of dynamic graph problems, Theoretical Computer Science, **158**, nos. 1–2, (1996), 233–277.

[8] Giorgio Ausiello ,Giuseppe Italiano, Alberto Marchetti Spaccamela, Umberto Nanni, Incremental algorithms for minimal length paths, Journal of Algorithms, **12**, no. 4, (1991), 615–638.

[9] Aaron Bernstein, Shiri Chechik, Deterministic partially dynamic single source shortest paths for sparse graphs, Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, (2017), 453–469.

[10] Aaron Bernstein, Liam Roditty, Improved dynamic algorithms for maintaining approximate shortest paths under deletions, Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms, (2011), 1355–1365.

[11] Julia Chuzhoy, Sanjeev Khanna, A new algorithm for decremental single-source shortest paths with applications to vertex-capacitated flow and cut problems, arXiv preprint arXiv:1905.11512,2019.

[12] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, A subquadratic-time algorithm for decremental single-source shortest paths, Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithm,(2014), 1053–1072.

[13] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs, Proceedings of the forty-sixth annual ACM symposium on Theory of computing, (2014), 674–683.

[14] Daniele Frigioni, Alberto Marchetti-Spaccamela, Nanni Umberto, Fully dynamic algorithms for maintaining shortest paths trees, Journal of Algorithms, **34,** no. 2, (2000), 251–281.

[15] Daniele Frigioni, Alberto Marchetti-Spaccamela, Nanni Umberto, Semi-dynamic algorithms for maintaining single-source shortest path trees, Algorithmica, **22**, no. 3, (1998), 250–274.

[16] Daniele Frigioni, Alberto Marchetti-Spaccamela, Nanni Umberto, Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights, European Symposium on Algorithms, (1998), 320–331.

[17] Daniele Frigioni, Mario Ioffreda, Nanni Umberto, Giulio Pasqualone, Experimental analysis of dynamic algorithms for the single source shortest paths problem, Journal of Experimental Algorithmics, **3,** no. 5, (1998).

[18] Franco Manessi, Alssandro Rozza, Mario Manzo, Dynamic graph convolutional networks, (2017), arXiv preprint arXiv:1704.06199.

[19] Ferozuddin Riaz, Khidir M. Ali, Applications of graph theory in computer science, Computational Intelligence, Communication Systems and Networks (CICSyN), 2011 Third International Conference, 142–145.

[20] Harith A. Dawood, Graph Theory and Cyber Security, Advanced Computer Science Applications and Technologies (ACSAT), 2014 3rd International Conference, (2014), 90–96.

[21] Aya Zaki, Mahmoud Attia, Doaa Hegazy, Safaa Ain, Comprehensive survey on dynamic graph models, International Journal of Advanced Computer Science and Applications, 7, no. 2, (2016), 573–582.

[22] Mikkel Thorup, Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles, Scandinavian Workshop on Algorithm Theory, (2004), 384–396.

[23] Mikkel Thorup, Worst-case update times for fully-dynamic all-pairs shortest paths, Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, (2005), 112–119,

[24] Ittai Abraham, Shiri Chechuk, Sebastian Krinninger, Fully dynamic all-pairs shortest paths with worst-case update-time revisited, Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, (2017), 440–452.