AKCE International Journal of Graphs and Combinatorics

ISSN: 0972-8600 (Print) 2543-3474 (Online) Journal homepage: https://www.tandfonline.com/loi/uakc20

A single-source shortest path algorithm for dynamic graphs

Muteb Alshammari & Abdelmounaam Rezgui

To cite this article: Muteb Alshammari & Abdelmounaam Rezgui (2020): A single-source shortest path algorithm for dynamic graphs, AKCE International Journal of Graphs and Combinatorics, DOI: 10.1016/j.akcej.2020.01.002

To link to this article: https://doi.org/10.1016/j.akcej.2020.01.002

Published online: 06 May 2020.

AKCE INTERNATIONAL JOURNAL OF GRAPHS AND COMBINATORICS
https://doi.org/10.1016/j.akcej.2020.01.002

A single-source shortest path algorithm for dynamic graphs

Muteb Alshammari[a] and Abdelmounaam Rezgui[b]

[a]Department of Computer Science & Engineering, New Mexico Tech, Socorro, NM, USA; [b]School of Information Technology, Illinois State University, Normal, IL, USA

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 single-source shortest path (SSSP) problem. Traditional solutions (based on Dijkstra's algorithms) to the SSSP problem do not scale to large dynamic graphs with a high change frequency. In this paper, we propose an efficient SSSP algorithm for large dynamic graphs. We first present our algorithm and give a formal proof of its correctness. Then, we give an analytical evaluation of the proposed solution.

KEYWORDS

Dynamic graphs; shortest paths; SSSP

1. Introduction

Graphs are mathematical structures used to model relationships between objects. A graph consists of a collection of vertices (i.e., objects) and edges (i.e., relationships) that connect vertices. Graphs are used in many areas including computer science (such as data mining, clustering, routing, and networks), biology and chemistry [3, 13, 14]. Most of the literature on graphs is concerned with *static* graphs, i.e., graphs that do not change over time. However, many of today's applications use *dynamic* graphs (that change over time) as their underlying data structure [15]. This has led to a surge in interest in developing new efficient algorithms that can scale to the large size and high frequency of changes in the graphs used in many modern applications, e.g., routing systems in communication networks, social networks. A classic problem in graph theory that is crucial to many applications is the Single-Source Shortest Path (SSSP). The SSSP problem is particularly challenging in the context of dynamic graphs. The difficulty is to efficiently update and maintain the shortest path from a source vertex to every other vertex while the graph is changing but without recomputing the entire path from scratch. In the remainder of this paper, we will use the acronym DSSSP (for Dynamic SSSP) to refer to the problem of computing and maintaining a single-source shortest path in a dynamic graph.

A number of researchers (e.g., [7, 11, 12]) have proposed solutions to the DSSSP problem. However, as we will show later in this paper, those solutions have significant performance limitations. In this paper, we propose a new DSSSP algorithm that addresses those limitations. Our solution is fully dynamic (defined later in this paper), efficient, more scalable, and less complex than previous solutions.

This paper is organized as follows: Section 2 provides an overview of related work. In Section 3, we describe our model of dynamic graphs. In Section 4, we present our approach including algorithms, proofs, and time complexity analysis. Section 5 concludes the paper.

2. Related work

Dynamic graphs can be classified based on the types of operations they support. Dynamic graphs are either *fully or partially dynamic* graphs. In fully dynamic graphs, all types of operations are allowed on the graph including the insertion and deletion of vertices and edges as well as the modification of weights of edges. In partially dynamic graphs, we are only allowed to have either insertion and weight decrease (i.e., *incremental dynamic graphs*) or deletion and weight increase (i.e., *decremental dynamic graphs*).

Depending on the application, one or the other of these classes of dynamic graphs may be used. Special algorithms are developed in the literature for each of these classes. In this paper, we will focus on one particular problem of dynamic graphs, namely, the Single-Source Shortest Path (SSSP) problem in dynamic graphs.

Several algorithms were proposed in the literature for the SSSP problem in dynamic graphs (e.g., [1, 6–12]). In [11], Ramalingam and Reps proposed the first fully dynamic algorithms for the dynamic SSSP problem as a sequence of updates [5]. In [12], they proposed another algorithm for

CONTACT Muteb Alshammari mutebalshammari@cs.nmt.edu Department of Computer Science & Engineering, New Mexico Tech, 801 Leroy Pl., Socorro, NM 87801, USA.

directed graphs that supports single updates. In the latter paper (RR for short), they used a DAG (called *SP*) to hold the paths that belong to the shortest paths of the graph. The use of the DAG forces the algorithm to perform more edge scans (in the case of incremental operations) which lead to a higher computational cost. The use of the DAG can help the algorithm in one specific scenario when it makes more deletions (or increases of weights of) edges than insertions (or decreases of weights of) edges and when those deleted edges can be overcome by using alternative paths. However, this can be eliminated by not scanning incoming edges for affected vertices as we will explain later in our approach.

In [8], Frigioni et al. proposed fully dynamic algorithms for undirected graphs (FMN for short) with an optimal time complexity. This is because they used specifically designed data structures to reduce the number of scanned edges. However, many experimental studies (e.g., [4, 5]) showed that FMN is very slow compared to other algorithms.

In [9], Narváez et al. established a framework for a set of algorithms for directed graphs that support single operations. The same authors proposed another algorithm that supports a sequence of updates in [10]. In [2], Chan and Yang found that the latter one was not correct in a particular scenario and a correction was proposed in the same paper. They also suggested optimizations for [8] and [11].

3. Model for dynamic graphs

Let $G = \{V, E, W\}$ be a dynamic directed weighted graph 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 size $m = |E|$, and $W : E \rightarrow \mathbb{R}$ 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 (including zero-length cycle).

Let s be a predefined source vertex for G, where $s \in V$. For each vertex $x \in V$, $in(x)$ is a set of incoming edges to x, $out(x)$ is a set of outgoing edges from x, and $d(x)$ is the shortest distance from s to x before the update operation. We denote the new distance of x after an update operation as $d'(x)$. $T(s)$ is a tree of the shortest paths rooted at source vertex s and $P(x)$ is the parent of vertex x in $T(s)$, where $x \in T(s) - \{s\}$.

For each update operation, we assume that G is updated first then we call the corresponding algorithm of that operation to maintain the SSSP. 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 $x, y \in V, (x, y) \notin E, x \neq y$, and $w > 0$.
- *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 $w > W(x, y)$.
- *Decrease_weight*(x, y, w) where $x, y \in V, (x, y) \in E, 0 < w < W(x, y)$.

Inserting a vertex x in the graph will have no effect since x is not connected to any vertex yet. For the second operation, we assume that deleting a vertex x is interpreted 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.

4. Our approach

In this section, we introduce and discuss the single-source shortest path (SSSP) algorithm for fully dynamic graphs. The algorithm consists of two partially dynamic algorithms. The first algorithm (Section 4.1) is the *incremental* dynamic algorithm (IDA) for incremental dynamic graphs, where only inserting edges and decreasing the weight of edges are allowed. The second algorithm (Section 4.2) is the *decremental* dynamic algorithm (DDA) for decremental dynamic graphs, where only deleting edges and increasing the weight of edges are allowed. Combining both algorithms will result in a fully dynamic algorithm for the SSSP problem in dynamic graphs.

Both algorithms use two data structures to maintain the SSSP for dynamic graphs. The first data structure is the shortest path tree $(T(s))$ rooted at s. The tree holds the shortest distances and paths from s to every x, $x \in V$ and x is reachable from s.

The second data structure is a minimal heap H that is keyed by the old distance of the inserted vertex (every x that gets extracted from H is an affected vertex). The new distance of a vertex x will be computed after x is extracted from H. Note that any vertex that gets inserted into H will be updated either by modifying its parent and/or assigning a new distance.

4.1. Incremental dynamic algorithm

The algorithm (Algorithm 1) handles two types of operations: (i) inserting a new edge and (ii) decreasing the weight of an existing edge. The algorithm can be divided into three phases. The first phase starts with effect assessment which tests the impact of the updated edge (either new edge inserted or edge weight decreased). If the edge (say (x, y)) does not affect y (i.e., does not improve its distance), then it will not have any effects on any other vertex and we shall stop. In this case, no further actions are needed. If (x, y) affects vertex y, then we can proceed to the second phase by updating the distance and parent of y using x. Then, y is inserted into the heap H with $key = d(y)$.

In the third phase, we loop over all affected vertices and extract them one by one according to their minimal distance from s. Vertices are inserted in the heap H (by line 13) only if they are affected. Note that, in the first iteration, we always extract y. In particular, in every iteration, the algorithm extracts a vertex (say u) from H whose value is minimum (line 8). Finally, lines 9–13 loop over the neighbors of u for possible improvement. If the distance of a neighbor of u (say v) can be improved (using u), then the distance and parent of v is fixed and v is inserted into the heap H to examine its neighbors when it is extracted later in the execution of the algorithm.

Algorithm 1. Inserting or decreasing the weight of edge (x, y)

```
1: procedure INCREMENTAL-ALGORITHM(x, y)
2:   if d(y) < d(x) + W(x,y) then
3:     Stop                ▷ no improvement can be accomplished
4:   P(y) ← x
5:   d(y) ← d(x) + W(x,y)
6:   insert(H, y, d(y))
7:   while H ≠ ∅ do
8:     u ← extract_min(H)
9:     for every v ∈ out(u) in G do
10:      if d(v) > d(u) + W(u,v) then
11:        P(v) ← u
12:        d(v) ← d(u) + W(u,v)
13:        insert(H, v, d(v))
```

4.1.1. Proof

We now prove the correctness of the algorithm by using loop invariant. We will show that the loop invariant holds at initialization, maintenance, and when the algorithm terminates. We assume that distances and paths are correct before the update operation and then we prove that they remain correct afterwards.

Recall that $d'(v)$ is the new distance of v after a graph update, whereas $d(v)$ is the distance of v before the update operation. For the sake of simplicity, we will use the following facts:

1. A vertex is affected if it changes its distance or parent.
2. At termination, every vertex either improves its distance from the source or keeps its old distance from s. In other words, $d'(v) \leq d(v)$, for every v, $s.t.$ $v \in V$.
3. Distances and paths are the shortest distances and paths, respectively, if the following is correct for every vertex v $s.t.$ $v \in V - \{s\}$ and v is reachable from s:
$$d(v) = min_{u \in in(v)}(d(u) + W(u, v))$$
4. For any vertex v $s.t.$ $v \in V$, if v is an affected vertex then it must be affected because of the edge (x, y) and v must be in the new $T(y)$. Therefore, all affected vertices will be in the sub-tree $T(y)$.
5. After an incremental operation on the edge (x, y), vertex y is either not affected and the algorithm will stop in line 3, or affected because the edge (x, y) improves its distance from the source s.

Invariant 1. *Every vertex v $(v \in V)$ that is extracted from the heap (i.e., $v \in H$) satisfies the following:*
$$d(v) = min_{u \in in(v)}(d(u) + W(u, v))$$

Theorem 1. INCREMENTAL-ALGORITHM(x, y) *correctly maintains a single-source shortest path tree in the graph G (using the model given in Section 3) after inserting (or decreasing the weight of) edge (x, y).*

4.1.2. Proof

i. **Initialization.** Before the first iteration, the heap has only one vertex which is y. From fact 5, y has improved its distance, was inserted into H, and has not yet been extracted. Note that the successors of y have not yet been scanned and they are not yet in H. As a result, H has not yet experienced any extraction. Therefore, the invariant holds at initialization.

ii. **Maintenance.** Suppose by contradiction that at some iteration the invariant is violated and there was a vertex u, u $\in V$, that is extracted from H and $d(u) \neq min_{x \in in(u)}(d(x) + W(x, u))$. This means that x was extracted earlier and it changed its distance. But if this is true then lines [1, 10] must have checked u at that time and u was either:
   - an affected vertex. Then, u must be updated (in lines 11 and 12) and inserted into H (in line 13) and either:
     (a) extracted and fixed. But if u was extracted then $d(u) = min_{x \in in(u)}(d(x) + W(x, u))$ which is a contradiction.
     (b) or still in the heap. But we assume that u is extracted from H which is a contradiction.
   - not an affected vertex. But if u is not affected then at least this must hold:
     $d(u) = min_{x \in in(u)}(d(x) + W(x, u))$ which is a contradiction.

   As all three possible scenarios lead to a contradiction, the invariant holds.

iii. **Termination.** The loop terminates after it constructs the sub-tree $T(y)$ (from fact 4) and it has to stop because there are no negative cycles. When the algorithm terminates, the heap is empty and every vertex v $(v \in V)$ has $d(v) = min_{u \in in(v)}(d(u) + W(u, v))$. Therefore, and from fact 3, the algorithm maintains the single-source shortest path tree in G. □

4.1.3. Time complexity

In this section, we analyze the time complexity of the DECREMENTAL-ALGORITHM. In this paper, we will use the model introduced by Ramalingam and Reps in their paper [12]. We recall that $|\delta|$ is the number of affected vertices, and $||\delta||$ is the sum of $|\delta|$ plus the number of edges that have at least one affected endpoint. We assume the use of a Fibonacci heap where the cost of inserting n elements is $O(\log(n))$ amortized time and, for the rest of the heap's operations, the cost is $O(1)$ amortized time.

The cost of lines 2 to 6 is constant. Lines 8 to 13 will repeat $|\delta|$ times (once for every affected vertex). Heap operations cost $O(\log|\delta|)$ (at each iteration) and $O(|\delta|.\log|\delta|)$ in totally. Lines 9 to 13 cost $O(||\delta||)$ in total (actually less than that since we scan only the outgoing edges of affected vertices.) Thus, the algorithm runs in time $O(||\delta|| + |\delta|.\log|\delta|)$.

4.2. Decremental dynamic algorithm

In this section, we present the decremental dynamic algorithm (DDA) for decremental dynamic graphs. In this algorithm, we use a coloring technique to color vertices as red or white. Affected vertices will be colored as red at the beginning and colored as white when they are either fixed

or at the end of the procedure. This will help avoid assigning an affected vertex (i.e., red vertex) as a parent to another vertex before the parent is fixed. Moreover, function *color*(*x*, *red*) (resp. *color*(*x*, *white*)) will color vertex x as red (resp. white). Furthermore, we will use a function called *pred_min*() which returns the best predecessor y with the minimal distance to u or *null* if there is no such path.

The DDA algorithm can be divided into three phases. The first phase is the effect assessment phase in which we check the effect of the edge (x, y) in the tree $T(s)$. If (x, y) is not an edge in $T(s)$, then it will have no effect and we shall stop. Otherwise, (x, y) will affect the tree $T(s)$ and we then need to move to the second phase of the algorithm.

In the second phase, the affected vertices are colored red and then the end point of the edge (x, y), i.e., y, is inserted into H.

The third phase is the main part which updates the tree $T(s)$ properly to maintain the SSSP result. This phase is a loop over all vertices in H. At each iteration, a vertex (u) is extracted from H with the minimal key. Then, its old value is temporarily saved, its distance is raised to ∞ and its parent is removed.

Now, *pred_min*() will find the best path to u with the minimal distance if it exists or *null* otherwise (y is either *null* or the returned parent). If y does not exist, then u is not reachable from s. If y exists and is not a red vertex, then we color u as white (since it has its correct distance) and fix its distance and parent by using y. Otherwise, if y exists but is red, then we need to reinsert u into H (and raise its distance to ∞) and extract it only after y is fixed.

Further, if u changes its distance, then the algorithm scans each neighbor of u. The neighbor of u, say v, is inserted into H if either it is a child of u in $T(s)$ or its distance can be improved (only if v was inserted by line 16). Otherwise, if u does not change its distance from s, then we need to reset the color of every vertex in $T(u)$ to white because they will not find better distances.

Finally, just before the termination of the procedure and after updating the result, we need to reset all remaining red vertices to white as a final step.

4.2.1. Proof

In this section, we prove the correctness of the DECREMENTAL-ALGORITHM by using the same method of loop invariant. To do so, we assume that G and $T(s)$ are correct before the update operation and all distances and paths are correct. We prove that they remain correct after the update operation. For simplicity, we use the following facts:

1. After encountering a decremental operation and updating the solution, this inequality must be satisfied, $d'(x) >= d(x)$, for every x, $s.t.\ x \in V$.
2. If a vertex x $(x \in V)$ changes only its parent, then its children in $T(x)$ will not be affected since they cannot find better paths (from fact 1).
3. If a vertex x $(x \in V)$ changes its distance from s, then its children in $T(x)$ will be affected (by changing their parent, distance or both).
4. Distances and paths are the shortest distances and paths, respectively, if the following is correct for every $v s.t. v \in V - \{s\}$:

$$d(v) = min_{u \in in(v)}(d(u) + W(u, v))$$

Algorithm 2. Deleting or increasing the weight of edge (x, y)

```
1: procedure DECREMENTAL-ALGORITHM (x, y)
2:   if (x,y) ∉ T(s) then
3:     Stop                                  ▷ this edge will have no effect
4:   for every w s.t. w ∈ T(y) do            ▷ color affected vertices as red
5:     color(w, red)
6:   insert(H, y, d(y))                      ▷ H is a min − heap
7:     while (H ≠ ∅) do
8:     u ← extract_min(H)
9:     old_value ← d(u)
10:    d(u) ← ∞
11:    P(v) ← null
12:    y ← pred_min(u)
13:    if y then
14:      if y is red then
15:        d(u) ← ∞
16:        insert(H, u, d(u))
17:      else
18:        color(u, white)
19:        d(u) ← d(y) + W(y, u)
20:        P(u) ← y
21:    if d(u) ≠ old_value then              ▷ if there is no alter path with same weight
22:      for every v ∈ out(u) in G do
23:        if (u, v) ∈ T(s) then             ▷ (u, v) is a tree edge
24:          insert(H, v, d(v))
25:        else
26:          if d(u) + W(u, v) < d(v) then
27:            insert(H, v, d(u) + W(u, v))
28:    else
29:      for every x s.t. x ∈ T(u) do
30:        color(x, white)
31:  for every w s.t. w is red do
32:    color(w, white)
```

Invariant 2. *Every vertex v that is: (i) neither in the heap (i.e., $v \notin H$) nor in sub-trees of vertices in the heap (i.e., $v \notin T(x)$, for every x s.t. $x \in H$), or (ii) neither has a neighbor u $(u \in in(v))$ that is in the heap (i.e., $u \notin H$) nor in sub-trees of vertices in the heap (i.e., $u \notin T(x)$, for every x s.t. $x \in H$), has its correct value (i.e., shortest distance and path from s).*

4.2.2. Proof

Suppose we have an update operation DECREMENTAL-ALGORITHM(x, y). Further, suppose (x, y) is a preferred path for y (i.e., x is the parent of y in $T(s)$), so (x, y) is a tree edge.

i. **Initialization.** Before the first iteration, the heap has only one vertex which is y. Vertex y is affected because:

(i) the edge (x, y) is a preferred edge for y in $T(s)$ and (ii) this edge was modified (otherwise the algorithm would stop in line 3). So, if the operation is to decrease the weight of edge (x, y), then y must either choose a different parent or change its value (or both). Otherwise, if the operation is to delete the edge (x, y), then y may be disconnected from s. In either case, y is affected.

Moreover, it is clear that (x, y) will only affect vertices that chose it as a preferred edge in their paths from s (i.e., y and vertices in sub-tree $T(y)$). This is because if (x, y) is not in their path then they must have better paths, and since the edge is either deleted or increased, it will never make any improvement for them (from fact 1). As a result, any vertex that does not choose (x, y) as a preferred edge in its path will not be affected and y is inserted into H earlier. Therefore, the loop invariant holds.

ii. **Maintenance.** Line 8 extracted a vertex u from H with the minimum key. Line 12 finds a parent with the best path that u can get if it exists and either:
   i. there is no such parent because either: (i) u is no longer reachable from s and, in this case, lines 10 and 11 had already raised u's distance to ∞ and removed u's parent, or (ii) the proper parent of u is in H (with distance raised to ∞) and is not fixed yet and, in this case, u will be reinserted into H after the proper parent gets fixed (using line 27).
   ii. there is such a parent (i.e., y) but it is red and not yet fixed. In this case, the distance of u is raised to ∞ and u is reinserted into H. It will be extracted and fixed after this parent is fixed.
   iii. there is such parent (i.e., y) and it is not red. In this case, u will get its final distance and parent. Also, its color will be set to white (so other vertices can use them as proper parent since u has its correct distance).

   It is easy to see that the invariant holds in all these cases.

   In either case, u either has changed its distance or not:
      (a) u has changed its distance. In this case, the algorithm will scan all neighbors of u. If a neighbor is a child of u in $T(s)$, then it will be inserted into H because it is affected (from fact 3). If the neighbor is not a child of u in $T(s)$, then it will be scanned for possible improvement and, if so, inserted into H. This step will ensure that the vertices that were scanned earlier than u and the distances of both u and those neighbors were raised to ∞.
      (b) u has not changed its distance. In this case, we reset the white color of every vertex in the sub-tree $T(u)$.

   Therefore, the loop invariant holds.

iii. **Termination.** It is clear that the loop will terminate because the loop is, basically, iterating over the sub-tree $T(y)$. The loop will terminate when the heap H becomes empty. This implies that all affected vertices were inserted into the heap (at some particular iteration). Then, they were extracted and fixed (by getting their correct distance, parent or both). Now, as the heap is empty, the vertices have their correct values. □

4.2.3. Time complexity

We now analyze the time complexity of the DECREMENTAL-ALGORITHM. We use the same model as in the previous algorithm.

Lines 2, 3, and 6 have a constant time whereas lines 4 and 5 take $O(|\delta|)$ since it is a loop over affected vertices (i.e., vertices in the sub-tree $T(y)$). Consequently, phases 1 and 2 take time $O(|\delta|)$.

Vertices are inserted into H at most 3 times (first time by line 6 or 24, second time by line 16, and last time by line 27). As a result, the loop in lines 7 to 30 will repeat at most $3 \times |\delta|$ times. Heap operations (in lines 8, 16, 24, and 27) cost $O(|\delta| \times \log(|\delta|))$ in total. Line 12 takes $O(||\delta||)$. The overall cost of the two loops in lines 29 to 32 is $O(|\delta|)$. The loop in lines 22 to 27 will repeat $||\delta||$ times and takes $O(||\delta|| + |\delta| \times \log(|\delta|))$ Therefore, the algorithm runs in time $O(||\delta|| + |\delta| \times \log|\delta|))$.

5. Conclusion

In this paper, we presented a novel, efficient approach to solve the problem of the Single-Source Shortest Path problem in dynamic graphs. We discussed the limitations of some of the most popular algorithms. We then presented our algorithms and established their correctness proofs, and analyzed their time complexity.

References

[1] Alshammari, M., Rezgui, A. (2020). An all pairs shortest path algorithm for dynamic graphs. *Int. J. Math. Comput. Sci.* 15(1): 347–365.

[2] Chan, E. P. F, Yang, Y. (2009). Shortest path tree computation in dynamic graphs. *IEEE Trans. Comput.* 58(4):541–557.

[3] Dawood, H. A. (2014). Graph theory and cyber security. In Advanced Computer Science Applications and Technologies (ACSAT), Amman, Jordan: IEEE, pp. 90–96.

[4] Demetrescu, C., Frigioni, D., Marchetti-Spaccamela, A, Nanni, U. (2000). Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In International Workshop on Algorithm Engineering, Saarbrücken, Germany: Springer, pp. 218–229.

[5] Frigioni, D., Ioffreda, M., Nanni, U, Pasquale, G. (1998). Experimental analysis of dynamic algorithms for the single source shortest paths problem. *J. Exp. Algorithmics* 3:5–es.

[6] Frigioni, D., Marchetti-Spaccamela, A, Nanni, U. (1996). Fully dynamic output bounded single source shortest path problem. In: *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '96*. Philadelphia, PA: Society for Industrial and Applied Mathematics, pp. 212–221.

[7] Frigioni, D., Marchetti-Spaccamela, A, Nanni, U. (1998). Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In European Symposium on Algorithms, Venice, Italy: Springer, 320–331.

[8] Frigioni, D., Marchetti-Spaccamela, A, Nanni, U. (2000). Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms* 34(2):251–281.

[9] Narváez, P., Siu, K.-Y, Tzeng, H.-Y. (2000). New dynamic algorithms for shortest path tree computation. *IEEE/ACM Trans. Netw.* 8(6):734–746.

[10] Narváez, P., Siu, K.-Y, Tzeng, H.-Y. (2001). New dynamic spt algorithm based on a ball-and-string model. *IEEE/ACM Trans. Netw.* 9(6):706–718.

[11] Ramalingam, G, Reps, T. (1996). An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms* 21(2):267–305.

[12] Ramalingam, G, Reps, T. (1996). On the computational complexity of dynamic graph problems. *Theoret. Comput. Sci.* 158(1-2):233–277.

[13] Riaz, F, Ali, K. M. Applications of graph theory in computer science. In 2011 Third International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN), Bali, Indonesia: IEEE, 2011, pp. 142–145.

[14] Shirinivas, S., Vetrivel, S, Elango, N. (2010). Applications of graph theory in computer science an overview. *Int. J. Eng. Sci. Technol.* 2(9):4610–4621.

[15] Zaki, A., Attia, M., Hegazy, D, Amin, S. (2016). Comprehensive survey on dynamic graph models. *Int. J. Adv. Comput. Sci. Appl.* 7(2):573–582.