International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Maintaining Minimum Spanning Trees in Fully Dynamic Graphs

Muteb Alshammari[1]*

Submitted:11/03/2024 **Revised:** 25/04/2024 **Accepted:** 02/05/2024

Abstract: Graphs are mathematical structures utilized in a variety of contexts. Numerous applications have emerged in recent years that require the processing of large dynamic graphs whose structure and properties are continuously evolving. communication, Social, and transportation networks are examples of such applications. The minimum spanning tree (MST) problem is among the most difficult challenges in dynamic graphs at a large scale. The MST problem is notoriously difficult to solve with traditional (algorithmically static) methods, especially on large dynamic graphs that undergo frequent changes. Accordingly, we propose an efficient and fully dynamic MST algorithm for large dynamic graphs in this paper. In light of this, the purpose of this paper is to present an efficient and fully dynamic MST algorithm designed for use with large dynamic graphs. First, we describe our algorithm, then we evaluate the proposed solution. In addition, we present a comprehensive experimental examination of our solution.

Keywords: *Minimum Spanning Tree, MST, Dynamic Algorithm.*

1. Introduction

Graphs are mathematical structures that are employed in the representation of the connections that exist between various things. Vertices are the "objects" that make up a graph, and edges are the "relationships" that connect the various vertices. Numerous disciplines, including computer science, chemistry, and biology, make use of graphs in their respective practices [2]. The majority of graph literature is focused on static graphs, or graphs which do not alter over time. However, the underlying data structure of many modern applications is in the form of dynamic graphs, which change as the application runs over time [21]. As a result, there has been a renewed focus on developing more effective algorithms that can handle the massive size and rapid evolution of graphs essential to many contemporary applications, such as routing protocols in communication networks and social graphs. Minimum Spanning Tree (MST) is a classic problem in graph theory that has a wide range of practical applications. The MST problem is especially difficult in the setting of dynamic graphs (DMST). The challenge lies in figuring out how to efficiently update and maintain the MST after encountering an update (or series of updates) on the underlying graph.

Dynamic graph algorithms are characterized according to the type of supported operations in which they are fully or partially dynamic. Algorithms that merely allow for the addition (or deletion) of vertices and edges are referred to as partially dynamic algorithms. By contrast, fully dynamic algorithms can support vertices and edge insertion and deletion.

Several scholars have presented solutions to the DMST problem (e.g., [12, 18, 4, 8, 20]). The time complexity has been the focus of many of these methods, while the practical considerations have been neglected. To overcome these limitations, we introduce a novel DMST algorithm in this paper. The proposed algorithm is fully dynamic, efficient, scalable, and less complex than earlier methods.

The remainder of this paper is structured as follows. Section 2 presents related work. The dynamic graph model and related terminologies are introduced in Section 3. Section 4 presents the proposed DMST approach. Section 5 describes our experimental evaluation in detail and discusses our findings. Section 6 brings the paper to a close.

Department of Information Technology, Faculty of Computing and Information Technology, Northern Border University, Arar, Saudi Arabia
ORCHI ID: 0000-0002-3473-5959
**Corresponding Author Email: Muteb.Alshammari@nbu.edu.sa*

2. Related Work

The DMST problem has been investigated for three decades and has received significant attention in recent years. This is due to the growing demand for new dynamic applications that rely on dynamic graph techniques. This need has prompted a reconsideration of the recommended solutions in the literature in order to further improve the state-of-the-art modules. Other reasons for implementing efficient dynamic algorithms include power consumption and restricted resources, such as those found in embedded systems.

Frederickson proposed in [8] a data structure for DMST maintenance (called topology trees) that sup$\sqrt{}$ports fully dynamic graphs. The topology trees, theoretically, reduced the MST updating cost to $O(\sqrt{m})$ per update. In [12], Holm et al. proposed an improvement to the update cost to $O(\sqrt{n})$ by deploying the sparsification technique. Their model supports fully dynamic graphs and only edge operations. It begins with the assumption of starting a fully dynamic graph with no edges and a fixed number of vertices. Ribeiro and Toso in [18] proposed deterministic fully dynamic algorithms for the DMST based on doubly-linked dynamic trees with a worst case updating time of O(|E|), where |E| is the number of edges. The paper included an experimental study that compares their implementations to [12] and [4].

Considering Las Vegas algorithms, Wulff-Nilsen in [20] proposed a fully dynamic algorithm with $O(n^{1/2-c})$worst-case updating time and a probability of at least $1-n^{-d}$. Nanongkai et al.

developed fully dynamic algorithms in [16] that improve the updating cost of [20] and [11].
The proposed fully deterministic algorithms for the connectivity problem can also solve the fully dynamic MST. Several algorithms for the connectivity problem, such as [13], have been proposed in the literature. In [13], Kapron et al. proposed polylogarithmic fully dynamic randomized algorithm for the connectivity problem with a worst case updating time of $O(log^4 n)$ for each edge insertion, $O(log^5 n)$ for each edge deletion, and $O(\log n / \log\log n)$ per query operation.
Giuseppe et al. in [4] implemented several deterministic DMST algorithms such as [12, 3, 9, 7] and reported their analysis and findings in a comprehensive experimental study.

3. Model for Dynamic Graphs

In this section, we present a mathematical model for fully dynamic graphs. We begin by defining an undirected weighted graph G = {V, E, W}. G consists of finite set of edges and vertices |E| and |V|, consecutively. G consists of n vertices and m edges where n = |V| and m = |E|, consecutively. W is a function that returns a real weight of an edge such that W: E → R. The graph G is dynamic in which it is under constant updates on vertices and edges. We assume graph G has no cycles of negative or zero length.

Assume s ∈ V is a source vertex predefined initially with no constrains. Additionally, consider the function out(v) that return the set of edges connected to v, where v ∈ V. T is an MST rooted at vertex s and $\boldsymbol{T_v}$ is a part of T containing vertex v, where v ∈ V.

To maintain the MST T, we assume that G is updated prior to each update operation and then invoke the corresponding update algorithm. Graph G supports update operations on both edges and vertices. The following operations can be performed on the graph G:

- Add_vertex(v), where v ∉ V.
- Remove_vertex(v), where v ∈ V.
- Add_edge (x, y, w), where (x, y) ∉ E, x, y ∈ V, and x ≠ y.
- Remove_edge (x, y), where (x, y) ∈ E and x, y ∈ V.
- Weight_decrease (x, y, w), where (x, y) ∈ E, x, y ∈ V, w < W(x, y).
- Weight_Increase (x, y, w), where (x, y) ∈ E, x, y ∈ V, and w > W (x, y).

Considering the first operation, the insertion of vertex x to G will not have any effect because x is not currently connected to any other vertex. For the removing vertex operation, we will assume that removing a vertex x from G is equivalent to removing one edge from out(x) at a time until the vertex in question no longer has any edges, and only then removing x from the graph. Other operations are explained in the rest of the paper.

4. Approach

This section describes and introduces the deterministic MST algorithm for fully dynamic graphs. Our approach consists of two algorithms: *incremental* and *decremental* dynamic algorithm. The *incremental* dynamic algorithm (MA_INC) in Section 4.1 supports insertion and weight decreasing of edges. The *decremental* dynamic algorithm (MA_DEC) in Section 4.2 supports deletion and weight increasing of edges.

4.1. Incremental Dynamic Algorithm

The incremental dynamic algorithm (MA_INC) in Algorithm 1 performs two operations: (a) the insertion of a new edge, and (b) the increasing of an edge weight. The algorithm begins by determining whether the edge is a tree edge, after which it returns with no changes. This is due to the fact that if the edge is a tree edge, then it will remain a tree edge. Otherwise, regardless of whether (x, y) is a newly inserted edge or an existing edge, we will add the edge (x, y) to the tree, introducing a cycle of the form x, y, . . ., x (note that T is not tree here, but it will be recovered at line 10). The algorithm then invokes the procedure Find_Path(s, x, y) to determine the path of the cycle x, y, . . . , x and stores it in a temporary list called Path. Following that, in lines 6-9, we find the edge with the highest weight in the list Path and delete it in line 10. The procedure Find_Path(x, y) is a recursive function whose main objective is to find the cycle x, y, . . . , x and return a list containing the edges of the cycle. In this function, we use the parameter x for the recursion base case (i.e., when x = y). The function uses a simple technique to avoid revisiting scanned edges by coloring visited edges "red." Initially, the function will be called with the edge (x, y) (i.e., the updated edge). The tree edges connected to the current edge's end point (i.e., y) are scanned and the function is called recursively for each neighbor (of y) who has not been visited (i.e., white edges). Then, the original color of each edge is recovered. Finally, the current edge is appended to other edges only if the function reaches the base condition. We note here that the condition in line 9 is to ensure that we only retain the path that

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

```
1: procedure MA_INC(x,y,w)
2:     if (x,y) ∈ T then
3:         return
4:     color(x,y) = red
5:     insert(T,x,y)
6:     Path = Find path(x,y)
7:     max = (x,y)
8:     for every (u,v) ∈ Path do
9:         if W(u,v) > W(max) then
10:            max = (u,v)
11:    T.delete(max)
12:    color(x,y) = white
```

hits the base case of the recursive function.

Algorithm 2: Find path in the cycle x, y, . . . , x

```
1: procedure Find_Path (x,y)
2:    if x == y then
3:       return
4:    for every v ∈ T.out(y) do
5:       if color(v,y) == white then
6:          color(v,y) = red
7:          found = Find_Path(x,v)
8:          color(v,y) = white
9:          if found then
10:            found.append((v,y))
11:            return found
12:   return
```

4.2. Decremental Dynamic Algorithm

The decremental dynamic algorithm (MA_DEC) in Algorithm 3 performs two operations: (a) edge deletion and (b) edge weight increasing. The algorithm begins by determining whether the edge is not a tree edge, after which it returns with no changes if it is not a tree edge. This is due to the fact that if it is not a tree edge, it will have no effect to the tree T. Then, the updated edge (x, y) is removed from the tree T. This will separate the tree into two sub-trees (or cuts): T_x and T_y). Each of which contains a vertex of the updated edge's endpoints. We employ a coloring strategy in which the vertices of the graph are colored either red or white (vertices originally were white). Therefore, we color vertices in the tree cut that contains x to red color and the other cut (cut y) will remain white. This will help in line 12 to differentiate the two cuts. After that, we iterate over the vertices in cut x and scan all edges connected to these vertices to find the connected edge with the minimum weight to vertices in the cut y. After finding the minimum edge, if it exists, we insert that edge to the tree T. This will reconnect the two cuts (after we delete (x, y in line 4) and the MST is recovered. Finally, we recover the original white color to vertices in the cut y.

Algorithm 3: Deleting the edge (x, y) or increasing its weight

```
1: procedure MA_DEC(x, y, w)
2:    if (x, y) ∈/ T then
3:       Return
4:    T.delete(x, y)
5:    Cut_x = ∅
6:    Min_w = ∞
7:    for z ∈ T_x∞ do
8:       color(z) = red
9:       Cut_x.insert(z)
10:   for every u ∈ Cut_x do
11:      for every v ∈ out(u) do
12:         if color(u) ≠ color(v) & W(u, v) < Min_w then
13:            Min_w = W(u, v)
14:            Min_edge = (u, v)
15:   if Min_edge then
16:      insert(T, Min_edge)
17:   for every v ∈ Cut_x do
18:      color(v) = white
```

4.3. Complexity Analysis

Worst-case time complexity analysis in dynamic graph algorithms is a challenge since many algorithms have no better performance than recomputing from scratch. As a result, many models, such as [17, 19, 14, 5, 10], have been proposed in the literature to provide more accurate time complexity analysis. In this paper, we extend Ramalingam and Reps model in [17] by computing the difference between the input and output of the applied algorithms. This model provides more accurate analysis because it measures the actual changes by the applied algorithms. In the extended model:

- β is a list of edges that form a cycle in the tree after inserting an edge of the form x, y, . . . , x into the tree (at most n edges).
- |β| is the number of edges in β.
- δ is a list of vertices in the cut x caused by deleting the edge (x, y) from the tree.
- |δ| is the length of δ. i.e. the sum of the number of vertices in the cut x (at most n − 1).
- ||δ|| is |δ| + number of edges connected to each of those vertices in δ.

Given the decremental dynamic algorithm, the loops in lines 7-9 and 17-18 will iterate 2 × |δ| times. Lines 10-14 will iterate ||δ||. The rest of the operations take constant time. Therefore, the decremental dynamic algorithm requires O(|δ| + ||δ||)

Considering the incremental dynamic algorithm, the algorithm is primarily depending on Find_Path procedure which recursively scans the tree edges to find the path x, y, . . . , x that takes |β| time. The loop in lines 8-10 iterates over the path edges which takes |β| times. Therefore, the incremental dynamic algorithm requires O(|β|).

Table 1: Random data sets details

n	*m*	*Density*	*n*	*m*	*Density*
200	1990	0.1	**1000**	200,000	0.2
200	5970	0.3	**1000**	400,000	0.4
200	9950	0.5	**1000**	600,000	0.6
200	13930	0.7	**1000**	800,000	0.8
200	17910	0.9	**2000**	800,000	0.2
400	7980	0.1	**2000**	1,600,000	0.4
400	23940	0.3	**2000**	2,400,000	0.6
400	39900	0.5	**2000**	3,200,000	0.8
400	55860	0.7	**3000**	1,800,000	0.2
400	71820	0.9	**3000**	3,600,000	0.4
600	17970	0.1	**3000**	5,400,000	0.6
600	53910	0.3	**3000**	7,200,000	0.8
600	89850	0.5			
600	125790	0.7			
600	161730	0.9			

200 to 3000 vertices and densities ranging from 10% to 90% as shown in Table 1. In each graph size, we consider different graph density to better view the performance of different algorithms. The graph density is defined as follows:

Table 2: Random data sets details

Data Set Type	n	m	Density
AS	500	2406	0.019
	1000	4546	0.009
	1500	6994	0.006
Road network of Pennsylvania (PA)	1,088,092	1,541,898	0.000003
Road network of Texas (TA)	1,379,917	1,921,660	0.000002
Road network of California (CA)	1,965,206	2,766,607	0.0000014

5. Experiments

This section describes the experimental study of our approach in compared to Ribeiro and Toso approach [18] (dynamic version) and Prim's algorithm (static version). We consider the algorithms of [18] since it is lately proved to be faster and more reliable in compared to the literature. All algorithms implemented in this paper are coded in Python and executed on AWS virtual machine [1] with Ubuntu 20.04, 16 vCPUs, and 128 GB of RAM. For the sake of perfection, we have not used any predefined graph library. Next, we provide the data sets used in this study along with edge operations. Then, we give the experimental results on the performance of the proposed incremental algorithm (MA_INC), decremental algorithm (MA_DEC), Ribeiro and Toso incremental algorithm (RIB_INC), Ribeiro and Toso decremental algorithm (RIB_DEC), and Prim's Algorithms. For each data set, we randomly choose 100 edges from the MST. Then, we delete an edge and apply the corresponding algorithm. After that, we reinsert the deleted edge and apply the corresponding algorithms. This will enforce all considered algorithms to update the MST. Our comparison with Prim's algorithm is to investigate the time it takes to only initialize the MST without considering the updates occurred to the graph. Each reported result is an average of five distinct runs. All experiments were validated by comparing the sum of edges on each MST of each algorithm.

5.1 Data Sets

In this experimental study, we take into consideration two different kinds of data sets: random data sets and real data sets. In the random data sets, we consider 24 different graphs of sizes from

$$density = 2m/(n \times (n-1))$$

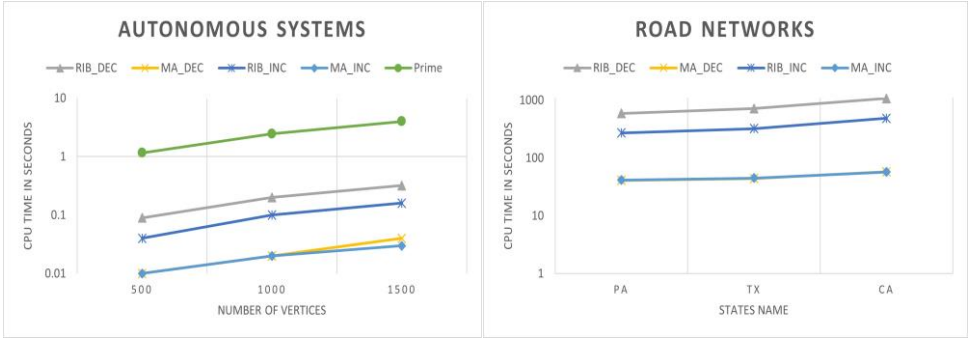
The graph density formula is used to determine the number of edges in the considered graph as follows:

$$m = \left(n \times (n-1)\right) \times density)/2$$

In the real data sets, we consider autonomous systems connection (AS) obtained from [6] and road networks of three states collected from [15]. In AS, we consider 3 graphs of different sizes: 500, 1000, and 1500 vertices as shown in Table 2. Where in road networks, we consider the available data sets for Pennsylvania (PA), Texas (TA), and California (CA). For each graph, we choose 100 edge deletions and 100 edge insertions from those who were chosen as part of the MST.

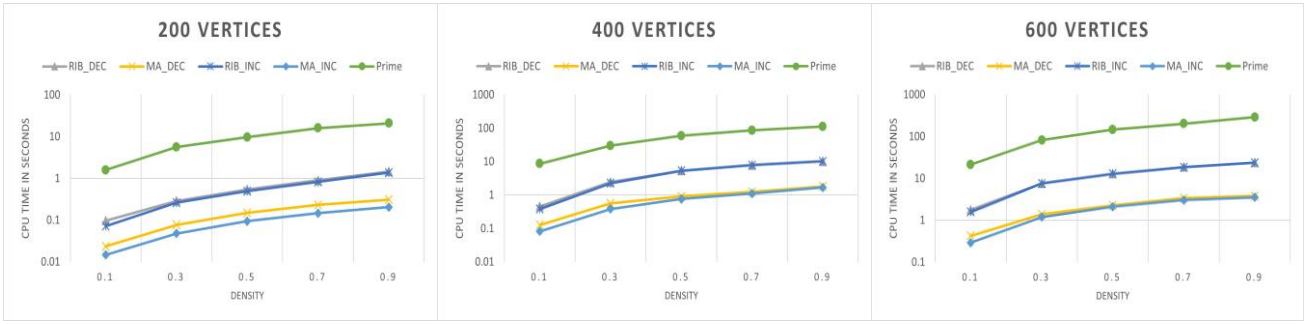
5.2. Performance Evaluation

The CPU consumption time for different algorithms is discussed and provided for synthetic and real data sets with varying numbers of vertices and densities. Across different numbers of vertices (200, 400, 600, 1000, 2000, and 3000) and densities (0.1, 0.3, 0.5, 0.7, and 0.9), the CPU consumption time increases with an increase in the number of vertices and density for all algorithms. Considering graphs with the same number of vertices, as the density of the graph increases, the CPU consumption time generally increases across all algorithms, indicating that denser graphs require more computational resources to process. We have observed that this is due to the need of these algorithms to scan more edges (either partially in the case of dynamic algorithms or



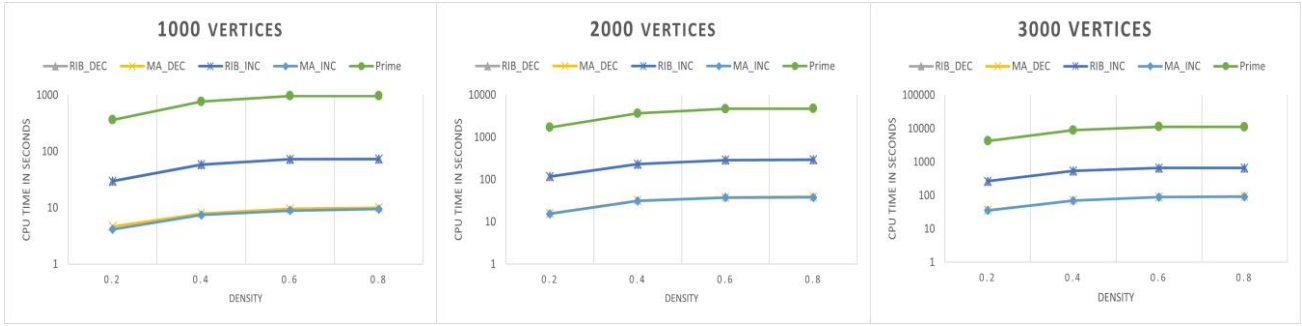
(a) (b)

Figure 1: Experiments on real data sets: (a) Autonomous Systems and (b) Road Networks



(a) (b) (c)

Figure 2: Experiments on small random data sets.



(a) (b) (c)

Figure 3: Experiments on large random data sets

completely in the case of static algorithm).

Our experiments on real data sets (i.e. Autonomous Systems and road networks) with various graph sizes are presented in Figure 1. We can observe the performance of Prim's algorithm (Figure 1) in comparison to dynamic algorithms (i.e. RIB_DEC, RIB_INC, MA_INC and MA_DEC). This is due to the fact that when Prim's algorithm is applied, it scans all edges and vertices. MA_INC and MA_DEC algorithms appear to be 5 to 10 times faster in comparison to RIB_DEC and RIB_INC. We realized that MA_INC is the fastest in all data sets because the algorithm is based primarily on the Find_Path procedure which scans only the tree edges with a maximum of n edges (i.e., $n - 1$ for tree edges plus the inserted edge (x, y)). We observed that RIB_DEC and RIB_INC exhibits more computational cost to update data structures used in their method which consists of a linked list for all edges in the graph. Our experiments show that preparing and updating such data structures would improve the theoretical time complexity rather than the computational complexity on experimental studies.

In addition, to evaluate the cost of different graph densities, we also take into consideration random data sets of varying densities from 10% to 90%. Figure 2 shows our experiments on three small graphs of different vertex sizes: 200, 400, 600, 1000, 2000, and 3000 vertices. Moreover, Figure 3 shows our experiments on three larger graphs with 1000, 2000, and 3000 vertices with edges between 200,000 and 7,200,000 edges. A pattern that is analogous to that of real data sets is repeated.

Among all algorithms, the Prime algorithm consistently exhibits the highest CPU consumption time, followed by RIB_DEC, RIB_INC, MA_DEC, and MA_INC, respectively.

In summary, the choice of algorithm, graph size, and density significantly impact CPU consumption. Optimizing algorithms for specific scenarios is crucial for efficient resource utilization. Understanding these trends can help in selecting appropriate algorithms and optimizing computational resources for graph processing tasks. The provided data allows for a comparative analysis of CPU consumption time across different algorithms, numbers of vertices, densities, and types of data sets.

6. Conclusion

This paper presents a novel, efficient, and less complex approach for maintaining the minimum spanning tree in fully dynamic graphs. In addition, a fully dynamic graph model was addressed. We then presented and analyzed the time complexity of our algorithms. Finally, we conducted a comprehensive experimental study utilizing both synthetic and actual data sets. Experiments conducted on both types of data sets demonstrated that our algorithms are efficient, fast, and less complex.

Declarations

- ***Ethics approval*** *Not Applicable.*
- ***Availability of data and materials*** *Available at https://github.com/muteb-nbu/D_MST.git.*
- ***Authors' contributions*** *This manuscript is the work of the single author.*

Conflicts of interest

The authors declare no conflicts of interest.

References

[1] Cloud computing services - amazon web services (aws). 5

[2] M Alshammari and Abdelmounaam Rezgui. A single-source shortest path algorithm for dynamic graphs. AKCE International Journal of Graphs and Combinatorics, 17(3):1063–1068, 2020. 1

[3] Giuseppe Amato, Giuseppe Cattaneo, and Giuseppe F Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In SODA, volume 97, pages 314–323. Citeseer, 1997. 2

[4] Giuseppe Cattaneo, Pompeo Faruolo, U Ferraro Petrillo, and Giuseppe F Italiano. Maintaining dynamic minimum spanning trees: An experimental study. Discrete Applied Mathematics, 158(5):404–425, 2010. 1, 2

[5] Camil Demetrescu and Giuseppe F Italiano. A new approach to dynamic all pairs shortest paths. Journal of the ACM (JACM), 51(6):968–992, 2004. 4.3

[6] Camil Demetrescu and Giuseppe F Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. ACM Transactions on Algorithms (TALG), 2(4):578–601, 2006. 5.1

[7] David Eppstein, Zvi Galil, Giuseppe F Italiano, and Amnon Nissenzweig. Sparsification—a technique for speeding up dynamic graph algorithms. Journal of the ACM (JACM), 44(5):669–696, 1997. 2

[8] Greg N Frederickson. Data structures for on-line updating of minimum spanning trees. In Proceedings of the fifteenth annual ACM symposium on Theory of computing, pages 252–257, 1983. 1, 2

[9] Greg N Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. SIAM Journal on Computing, 26(2):484–538, 1997. 2

[10] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. Journal of Algorithms, 34(2):251–281, 2000. 4.3

[11] Monika Rauch Henzinger and Valerie King. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation. SRC Technical Note, 4, 1997. 2

[12] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. Journal of the ACM (JACM), 48(4):723–760, 2001. 1, 2

[13] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms, pages 1131–1142. SIAM, 2013. 2

[14] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In 40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039), pages 81–89. IEEE, 1999. 4.3

[15] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014. 5.1

[16] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. Dynamic minimum spanning forest with subpolynomial worst-case update time. In 2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS), pages 950–961. IEEE, 2017. 2

[17] Ganesan Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. Theoretical Computer Science, 158(1–2):233–277, 1996. 4.3

[18] Celso C Ribeiro and Rodrigo F Toso. Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes on edge weights. In Experimental Algorithms: 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007. Proceedings 6, pages 393–405. Springer, 2007. 1, 2, 5

[19] 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, pages 112–119. ACM, 2005. 4.3

[20] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, pages 1130–1143, 2017. 1, 2

[21] Aya Zaki, Mahmoud Attia, Doaa Hegazy, and Safaa Amin. Comprehensive survey on dynamic graph models. International Journal of Advanced Computer Science and Applications, 7(2):573–582, 2016. 1