# Maintaining an Approximate Steiner Tree in Dynamic Graphs

***Thesis submitted to the***
***Indian Institute of Technology Guwahati***
***for the award of the degree***

***of***

**Doctor of Philosophy**

in

**Computer Science and Engineering**

Submitted by

**Hemraj Raikwar**

*Under the supervision of*

**Prof. Sushanta Karmakar**

**Department of Computer Science and Engineering**

**Indian Institute of Technology Guwahati**

**Guwahati - 781039 Assam India**

**October, 2025**

Copyright © Hemraj Raikwar 2025. All Rights Reserved.

*Dedicated to*

*My family*

# Declaration

I certify that

- The work contained in this thesis is original and has been done by myself and under the general supervision of my supervisor.
- The work reported herein has not been submitted to any other Institute for any degree or diploma.
- Whenever I have used materials (concepts, ideas, text, expressions, data, graphs, diagrams, theoretical analysis, results, etc.) from other sources, I have given due credit by citing them in the text of the thesis and giving their details in the references. Apart from the text taken from the work published by me, elaborate sentences used verbatim from other published works have been clearly identified and quoted.
- I also affirm that no part of this thesis can be considered plagiarism to the best of my knowledge and understanding and take complete and sole responsibility if any complaint arises in the future.
- I am fully aware that my thesis supervisor is not in a position to check for any possible instance of plagiarism within this thesis.

Date: October 2025

Place: Guwahati — Hemraj Raikwar

# Acknowledgements

I want to express my gratitude to my thesis supervisor, Prof. Sushanta Karmakar. He allowed me the opportunity to venture into the exciting and intellectually fulfilling research landscape covered under this thesis. I would also like to thank him for the offshoot discussions and brainstorming sessions done during the initial phase of my research which I still cherish and value. I thank him for introducing me to the evolutionary side of theoretical computer science which is still untapped in many ways and intrigues me with its fascinating mechanisms.

I am thankful to Prof. Purandar Bhaduri for chairing my doctoral committee and guiding me with his valuable suggestions. I cherish our discussions and also the extra-curricular engagements that I could be a part of. I am also grateful to Prof. Pinaki Mitra and Prof. Partha Sarathi Mandal for presiding over my doctoral committee and scrutinizing my research trajectory with their valuable assessments. I am also thankful to Mr. Apuroop, Ms Mikki, and Mr. Harshil for collaborating as a mentee on interesting research engagements.

I am and will always be grateful to my parents, Shri Shyamlal Raikwar and Smt. Guddi Raikwar, for nurturing within me the interest and aptitude that led to this research endeavor and allowing me the freedom and due support to focus on this pursuit. Their foresight, planning, patience, and teachings not only enabled me to land on this coveted platform but also helped me steer through the challenges faced along this journey. I shall forever be indebted to them for their contribution. I am also thankful to my elder brother, Krishna Gopal, and elder sisters, Parvati, Kavita, Ranjana, and Malti, who are always there for me, right from childhood. I am thankful to my wife, Kiran, for being a constant support. Her presence made it easier to steer through the demanding pursuit of this research engagement.

I am thankful to the heads of the department of computer science and engineering, Indian Institute of Technology (IIT) Guwahati, who presided over the course of my PhD, Prof. S V Rao, Prof. Jatindra Kumar Deka and Prof. T. Venkatesh, for allowing me the facilities of the department and for supporting my conference registrations. I am also thankful to them and Prof. Sushanta Karmakar for helping me with the finances to arrange for the international workshops and conferences I attended during this research work.

I am thankful to the Ministry of Education of India and IIT Guwahati for supporting my PhD with a scholarship. I am also thankful to the organizers of the Indo-German Spring School on Algorithms for Big Data for providing me with a generous travel grant.

I am thankful to Mr. Hemanta Kumar Nath, Mr. Pranjit Talukdar, Mr. Raktajit Pathak, Mr. Nanu Alan Kachari, Mr. Nava Kumar Boro, and Mr. Bhriguraj Borah for always being there in case of any technical issues or requirements. I would also like to thank Mrs. Gauri Khuttiya Deori, Mr. Gourish Mazumder and Mr. Prabin Bharali for helping with the administrative procedures during my PhD. I enjoyed taking the coursework at IIT Guwahati and would thank the faculty members and the associated teaching assistants for the conduct of the course. I am also thankful to the department's security staff, janitors, and our college's canteen staff for their round-the-clock support.

I would also like to thank my friends at IIT Guwahati, including Yashdeep, Suraj, Nilotpal, Debanjan, Maithili, and Nidhi who made my stay at the institute pleasant. I am happy to mention my childhood friend Priyamwad for his constant encouragement and support. I am also thankful to my seniors, Dr. Parikshit, Dr. Mirza, Dr. Swarup, Dr. Pradeepkumar, Dr. Pawan, Dr. Ujjwal, and Dr. Deepankar, for their company and helpful conversations.

Lastly, I would like to thank the IIT Guwahati administration for maintaining a picturesque and pristine ambiance throughout the campus along with the top-notch facilities, including the sports facilities, the swimming pool, the lush green grounds, and the scenic lakes.

October 2025 Hemraj Raikwar

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati - 781039 Assam India

---

# Certificate

This is to certify that this thesis entitled "**Maintaining an Approximate Steiner Tree in Dynamic Graphs**" submitted by **Hemraj Raikwar**, in partial fulfillment of the requirements for the award of the degree of Doctor of Philosophy to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Assam, India, is a record of the bonafide research work carried out by him under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, Assam, India. To the best of my knowledge, no part of the work reported in this thesis has been presented for the award of any degree at any other institution.

Date: October 2025
Place: Guwahati

**Prof. Sushanta Karmakar**
(Thesis Supervisor)
Professor
Department of Computer Science and Engineering
IIT Guwahati
Email : sushantak@iitg.ac.in
Phone : +91-361-258-2368

# Abstract

Computing a Steiner tree of a given graph is a well-studied problem. Given an undirected connected graph $G = (V, E)$, a weight function $w : E \rightarrow R^+$, and a set of vertices $S \subseteq V$, known as the set of terminals, the goal of the Steiner tree problem is to find a tree $T = (V', E') \subseteq G$ such that $\Sigma_{e \epsilon E'} w(e)$ is minimized subject to the conditions: $S \subseteq V' \subseteq V$ and $E' \subseteq E$. We study the Steiner tree problem in dynamic graphs. A dynamic graph is a graph where the edge set, the vertex set or both change with time. The graph is subjected to a sequence of updates that may insert or delete some edges or vertices. The algorithms for dynamic graphs are based on maintaining data structures to update the existing solution after each update to minimize the update time rather than computing the solution from scratch. Our objective is to efficiently update and maintain a good approximate Steiner tree under the graph updates. We surveyed the existing algorithms and techniques for computing a Steiner tree in static and dynamic graphs. The dynamic Steiner tree has immense applications in wireless networks, cyber-physical systems, and multicast trees for information sharing. The Steiner tree formation also acts as a building block in designing *P2P* networks, ad-hoc networks, and solving more complex problems such as prize-collecting Steiner tree.

The dynamic Steiner tree problem with edge insertion and deletion is a new problem, whereas the dynamic Steiner tree problem with conversion of terminal to non-terminal and vice versa is an existing problem. We propose five dynamic algorithms for maintaining an approximate Steiner tree under different kinds of updates and graphs. We consider an edge-weighted undirected graph with positive real edge weights. We propose a fully dynamic algorithm in planar graphs where updates can insert and delete edges, and an incremental algorithm for the Steiner tree problem in general graphs where updates can insert edges in the graph. In the fully dynamic case, our analysis demonstrates that the presented algorithm maintains an approximation factor of $(2 + \epsilon)$ in $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$ worst case update time for processing a series of $k$ number of updates where $n$ is the number of vertices in the input graph, $D$ denotes the unweighted diameter of the updated graph, and $k \in Z^+ : 1 \leq k \leq (\sqrt{n + m})$. We show that the update time improves to $O(n(\epsilon')^{-2})$ in a special case for $\epsilon' = \epsilon/2$. On the other hand, the incremental

algorithm maintains a $(2-\epsilon)$-approximate Steiner tree in general graphs in $O(nD_s)$ worst case update time. Here, $D_s$ is the shortest path diameter of the modified graph. Given a graph $G=(V,E)$ with edge weights defined by a function $w: E \rightarrow \mathbb{R}^+$, the shortest path distance between two vertices $u, v \in V$ is denoted by $d(u,v)$, which is the length of the shortest path connecting $u$ and $v$ in $G$. The *shortest path diameter* of the graph $G$ is defined as the maximum shortest path distance over all pairs of vertices: $D_s = \max_{u,v \in V} d(u,v)$. The fully dynamic algorithm leverages tools from a dynamic distance oracle, whereas the incremental algorithm maintains a partition of the input graph in the form of a shortest path forest, which aids in efficiently updating a Steiner tree.

Most of the existing algorithms for the Steiner tree problem are based on an MST heuristic establishing a $(2-\epsilon)$-approximate Steiner tree. We establish a lower bound on the update time required to maintain an MST heuristic based $(2-\epsilon)$-approximate Steiner tree (where $\epsilon$ is a small fraction) in a general graph undergoing edge insertions or deletions. Also, we propose a decremental algorithm for the Steiner tree problem in weighted planar graphs. The graph undergoes a sequence of edge deletion updates. The proposed algorithm maintains a $(2+\epsilon)$-approximate Steiner tree under each edge deletion in $\tilde{O}(\ell\sqrt{n})$ worst case update time. Here, $\ell$ is the maximum hop length of a $(1+\epsilon'')$-approximate shortest path between any two nodes in a graph, and $\epsilon$ and $\epsilon''$ are small fractions. The relations between $\epsilon$ and $\epsilon''$ are based on other factors, and are discussed during their derivation.

Moreover, we propose a novel approach to maintain an approximate Steiner tree in fully dynamic planar graphs where the updates can be the insertion or deletion of a vertex from the graph, the insertion or deletion of an edge from the graph, terminal to non-terminal conversion, and non-terminal to terminal conversion. The algorithm maintains an approximate MST and a $(|S|-1)$-approximate Steiner tree. We also prove that the weight of any spanning tree of the complete distance graph on terminals is at most $(|S|-1)$ times the weight of an optimal Steiner tree. To the best of our knowledge, the proposed algorithm is the first algorithm that allows these six types of graph updates and maintains a $(|S|-1)$-approximate Steiner tree in planar graphs in $\tilde{O}((\delta p \log U)/\epsilon + |S| \cdot \sqrt{n} + n)$ update time for $p$ number of updates. Here, $\delta$ is the maximum degree of a vertex, and $U$ is the maximum edge weight. We propose a fully dynamic algorithm in general graphs supporting the same six types of updates. By integrating dynamic clustering, spanner-based connectivity, and a hybrid distance oracle, the algorithm achieves an approximation factor of $(2+\epsilon)$ for some tunable $0 < \epsilon < 1$. Furthermore, it achieves an expected update time complexity of $O(m^{1/2} + n^{2/3})$ per update.

These algorithms significantly improve over existing dynamic Steiner tree algorithms in terms of flexibility, approximation guarantees, and update time complexity. The proposed algorithms are based on several existing techniques, including maintaining approximate shortest paths, a shortest path forest, *ET*-trees, spanners, and several techniques designed by us, including efficient computation of shortest paths between terminals, efficient computation of shortest paths between trees and augmenting existing distance oracles.

The main results of the thesis are the following:

1. ***Fully dynamic algorithms for planar graphs:*** An algorithm that maintains a $(2+\epsilon)$-approximate Steiner tree with worst-case update time $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$, where $n$ is the number of vertices and $D$ is the (unweighted) diameter. This bound holds for up to $k$ updates, with $1 \leq k \leq \sqrt{n+m}$. In special cases, the update time improves to $O(n\epsilon'^{-2})$.

2. ***An incremental algorithm for general graphs:*** An algorithm that maintains a $(2-\epsilon)$-approximate Steiner tree with worst-case update time $O(nD_s)$, where $D_s$ is the shortest-path diameter of the updated graph.

3. ***A lower bound on the update time:*** An $\Omega(n)$ lower bound is established on the update time required to maintain an MST-heuristic-based $(2-\epsilon)$-approximate Steiner tree in general graphs under edge updates.

4. ***A decremental algorithm for planar graphs:*** An algorithm that maintains a $(2+\epsilon)$-approximate Steiner tree under edge deletions with worst-case update time $\tilde{O}(l\sqrt{n})$, where $l$ is the maximum hop length of a $(1+\epsilon'')$-approximate shortest path.

5. ***Fully dynamic algorithms for six update types in planar and general graphs:***

   - *Planar graphs:* An algorithm maintaining a $(|S|-1)$-approximate Steiner tree with update time $\tilde{O}\left(\frac{\delta p \log U}{\epsilon} + |S|\sqrt{n} + n\right)$, where $\delta$ is the maximum degree, $U$ the maximum edge weight, and $p$ the number of updates.
   - *General graphs:* An algorithm integrating clustering, spanners, and a hybrid distance oracle to achieve a $(2+\epsilon)$-approximation with expected update time $O\left(m^{1/2} + n^{2/3}\right)$, where $m$ is the number of edges.

# Contents

**Abstract** xi

**List of Figures** xx

**List of Algorithms** xxi

**List of Tables** xxiii

**1 Introduction** 1

1.1 Steiner Tree . . . 1
1.2 Approximation Algorithms and the Steiner Tree Problem . . . 2
1.3 The Steiner Tree Problem in Static Graphs . . . 3
1.4 The Steiner Tree Problem in Dynamic Graphs . . . 5
1.4.1 Dynamic Graphs and Dynamic Algorithms . . . 5
1.4.2 Dynamic Steiner Tree Problem . . . 5
1.5 Applications of the Steiner Tree . . . 7
1.6 Motivation . . . 8
1.7 Objectives . . . 11
1.8 Contribution of the Thesis . . . 12
1.8.1 A Fully Dynamic Algorithm in Planar Graphs . . . 12
1.8.2 An Incremental Algorithm in General Graphs . . . 13
1.8.3 A Lower Bound . . . 13
1.8.4 A Decremental Algorithm in Planar Graphs . . . 14
1.8.5 A Fully Dynamic Algorithm Handling Edge and Vertex Insertions and Deletions and Terminal Conversions in Planar Graphs . . . 14
1.8.6 A Fully Dynamic Algorithm Handling Edge and Vertex Insertions and Deletions and Terminal Conversions in General Graphs . . . 15
1.9 Outline of the Thesis . . . 16

**2 Related Work** **19**
2.1 Dynamic Algorithms . . . 19
2.2 The Static Steiner Tree Problem . . . 20
2.3 The Dynamic Steiner Tree Problem . . . 24
2.4 The Steiner Tree Problem in Distributed Settings . . . 27
2.5 Important Results in the Steiner Tree Problem . . . 31

**3 A Fully Dynamic and an Incremental Algorithm** **35**
3.1 A Fully Dynamic Algorithm in Planar Graphs . . . 36
3.1.1 Preliminaries . . . 36
3.1.2 Algorithm . . . 38
3.1.3 Analysis . . . 43
3.2 An Incremental Algorithm for $(2-\epsilon)$-Approximate Steiner Tree 50
3.2.1 Preliminaries . . . 50
3.2.2 A $(2-\epsilon)$-Approximate Steiner Tree . . . 51
3.2.3 Shortest Path Forest . . . 51
3.2.4 The Incremental Algorithm . . . 52
3.2.5 Analysis . . . 60
3.3 Summary . . . 64

**4 Decremental $(2+\epsilon)$-Approximate Steiner Tree in Planar Graphs** **67**
4.1 Introduction . . . 67
4.2 Preliminaries . . . 68
4.2.1 $(2-\epsilon')$-Approximate Steiner Tree: . . . 68
4.2.2 Dynamic Distance Oracle: . . . 69
4.2.3 ET-Trees: . . . 69
4.3 Lower Bound for MST Heuristic Based Algorithms . . . 73
4.4 The Decremental Algorithm . . . 77
4.5 Analysis and Proof of Correctness . . . 84
4.5.1 Approximation Factor . . . 85
4.5.2 Update Time . . . 87
4.6 Summary . . . 88

**5 Fully Dynamic Algorithms Handling Edge and Vertex Insertions and Deletions and Terminal Conversions** **91**
5.1 Fully Dynamic Approximate Steiner Tree Algorithm for Planar Graphs . . . 91
5.1.1 Preliminaries . . . 91
5.1.2 Overview . . . 92
5.1.3 Updating Approximate Steiner Tree . . . 93
5.1.4 Proof of Correctness . . . 96

5.2 Fully Dynamic Approximate Steiner Tree Algorithm in General Graphs . . . 100
5.2.1 Techniques Used . . . 101
5.2.2 Algorithm Design . . . 102
5.2.3 Analysis . . . 114
5.3 Summary . . . 123

**6 Conclusions and Future Work** **125**
6.1 Conclusions . . . 125
6.2 Future Work . . . 128

**Bibliography** **131**

**Disseminations from the Thesis** **137**

# List of Figures


1.1 An example graph and a corresponding optimal Steiner tree. . 2

1.2 An example directed graph and a corresponding optimal directed Steiner network. . . . . . . . . . . . . . . . . . . . . . . . . 4

1.3 An example directed graph and a corresponding optimal directed Steiner tree. . . . . . . . . . . . . . . . . . . . . . . . . . . 4

3.1 Inserting edges $(v_2, v_6, 1)$ and $(v_3, v_5, 3)$ to the graph $G$. . . . . 40

3.2 The graph $G_1$(left) and the tree $T_1$(right) . . . . . . . . . . . . 41

3.3 The tree $T_2$(left) and the final Steiner tree $T_3$(right) for $G'$ . . . 43

3.4 The shortest path between the terminals $s_1$ and $s_2$ through $r$ as $p_2$ and $s_2$ as $s$ . . . . . . . . . . . . . . . . . . . . . . . . . 53

3.5 Both endpoints of the inserted edges lie inside a tree. Insertion of the edge $(u_2, u_7)$ does not change the shortest path to $u_2$ or $u_7$. Insertion of the edge $(u_3, u_9)$ changes the shortest path to $u_9$. 54

3.6 Endpoints of the inserted edges lie in different trees. Insertion of the edge $(u_4, u_{10})$ does not change the shortest path between $s_1$ and $s_2$. Insertion of the edge $(u_4, u_{11})$ changes the shortest path between $s_1$ and $s_2$. . . . . . . . . . . . . . . . . . . . . . . . 55

4.1 Example trees to show *link* and *cut* operations . . . . . . . . . 70

4.2 $ET$-tree operations on dynamic trees . . . . . . . . . . . . . . . . 70

4.3 Deletion of an edge from Euler tour of a tree . . . . . . . . . . 71

4.4 Changing the root of a tree from $u$ to $v$ . . . . . . . . . . . . . 72

4.5 Merging Euler tours of two trees $T_1$ and $T_2$ by the edge $(u, v)$ . 73

4.6 A graph with three terminals (left) and a graph with four terminals (right). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

4.7 The complete distance graph over the terminals (left) and the MST heuristic based $(2 - \epsilon')$-approximate Steiner tree (right) for the graph in Fig. 4.6a before insertion of the edge $(a, b)$. . 75

4.8 The updated complete distance graph over the terminals (left) and the MST heuristic-based $(2-\epsilon')$-approximate Steiner tree (right) for the graph in Fig. 4.6a after insertion of the edge $(a, b)$. . . . . 75
4.9 The complete distance graph over the terminals (left) and the MST heuristic based $(2-\epsilon')$-approximate Steiner tree (right) for the graph in Fig. 4.6b before insertion of the edge $(a, b)$. . 76
4.10 The updated complete distance graph over the terminals (left) and the MST heuristic-based $(2-\epsilon')$-approximate Steiner (right) for the graph in Fig. 4.6b after insertion of the edge $(a, b)$. . . 76
4.11 Before deletion of the edge $(v_2, v_4)$ . . . . . 79
4.12 After deletion of the edge $(v_2, v_4)$ . . . . . 80
4.13 Updated $G'$, $T_{S_1}$ and $T_{S_2}$ . . . . . 81
4.14 Updated Steiner tree $T_S$ . . . . . 82

5.1 An example graph (a) and Clusters (b) . . . . . 103
5.2 The spanner for the updated graph, showing inter-cluster edges (dashed) and intra-cluster edges (solid). Clusters are enclosed with dashed boundaries. . . . . 105
5.3 Initial 2-approximate Steiner tree (left) and the updated 2-approximate Steiner tree (right) after insertion of $(D, F)$ with weight 1. . . . . 108
5.4 Initial 2-approximate Steiner tree (left) and the updated 2-approximate Steiner tree (right) after deleting $(C, F)$ with weight 6, updated using the spanner edge $(B, D)$. . . . . 109

# List of Algorithms


1 Construction of the complete graph (metric closure) on the terminals . . . 40
2 Path replacement scheme . . . 42
3 An Incremental algorithm for $(2-\epsilon)$-approximate Steiner tree 56
4 FIND_MAX_WEIGHT_PATH(Steiner tree $T$, Terminal $u$, Terminal $v$) . . . 58
5 DFS(Steiner tree $T$, vertex $x$, terminal $p$, weight $c$, max_weight $w_{max}$, $w_{max}$_endpoint $a$, $w_{max}$_endpoint $b$, PARENT, terminal_endpoint $v$) . . . 59
6 A $(2+\epsilon)$-Approximate Steiner Tree . . . 83
7 $(|S|-1)-$approximate Steiner tree . . . 94
8 Neighbor Terminal Pairs . . . 96
9 Fully Dynamic Steiner Tree Algorithm . . . 112

# List of Tables

2.1 Results in the static Steiner tree problem . . . . . . . . . . . . . 31
2.2 Results in the dynamic Steiner tree problem . . . . . . . . . . . . 32
2.3 Results in the Steiner tree problem in distributed settings . . . 33

5.1 Update time to maintain $(1+\epsilon)-$approximate MST . . . . . . 100

6.1 Results in the dynamic Steiner tree problem . . . . . . . . . . . . 127

# 1

# Introduction

This thesis delves into the Steiner tree problem within the realm of dynamic graphs. The Steiner tree problem is a widely studied combinatorial optimization problem that aims to minimize the tree cost while adhering to specific constraints [1]. It is a network optimization problem where the underlying graph topology changes due to updates, and the objective is to update and maintain a tree that connects a set of designated vertices.

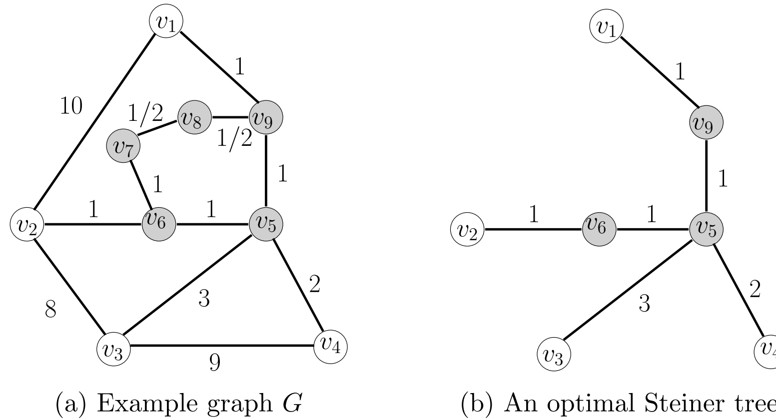
## 1.1 Steiner Tree

Formally, the classic Steiner tree problem is defined as follows:

**Definition 1.1.** *(Steiner Tree Problem (Chlebík and Chlebíková [2])) Given an undirected connected graph $G = (V, E)$, a weight function $w : E \to R^+$, and a set of vertices $S \subseteq V$, known as terminals, the goal of the Steiner tree problem is to find a tree $T = (V', E')$, a sub-graph of $G$, such that $\Sigma_{e \epsilon E'} w(e)$ is minimized subject to the conditions:*

- $S \subseteq V' \subseteq V$
- $E' \subseteq E$

The tree may contain vertices other than terminals to minimize the overall weight. The vertices in $V \setminus S$ are called non-terminals. Non-terminals that are included within the Steiner tree are termed as Steiner vertices. Steiner vertices help connect the terminals efficiently. Figure 1.1 shows an example Steiner tree where the white vertices are terminals and the gray vertices are non-terminals.




(a) Example graph $G$ (b) An optimal Steiner tree

Figure 1.1: An example graph and a corresponding optimal Steiner tree.

The presence of a non-terminal leaf in a Steiner tree implies an unnecessary edge that does not contribute to connecting the terminals. Removing such leaves and their connecting edges would lead to a Steiner tree having a lower cost that still connects all the terminals. Hence in a Steiner tree, all leaves must be terminals.

The Steiner tree problem is a more generalized version of the minimum spanning tree (MST) problem. When the set of terminals $S = V$, the Steiner tree problem becomes the minimum spanning tree problem. When $|S| = 2$, the Steiner tree problem simplifies to the $s - t$ shortest path problem. There exist efficient (polynomial time) algorithms for the $s-t$ shortest path problem and the minimum spanning tree problem, but the Steiner tree problem is an NP-complete problem [3, 4]. Hence, the Steiner tree problem can not be solved optimally in polynomial time unless $P = NP$. This leads to the development of approximation algorithms for the Steiner tree problem.

## 1.2 Approximation Algorithms and the Steiner Tree Problem

In computational complexity, many important optimization problems are known to be *NP-hard*, meaning that finding exact solutions efficiently (in polynomial time) is unlikely unless $P = NP$. To address this, **approximation algorithms** are designed to compute solutions that are *close* to optimal in a reasonable amount of time. These algorithms offer *provable guarantees* on how close the returned solution is to the optimal one.

The quality of an approximation algorithm is measured by its **approx-**

**imation factor** (or *approximation ratio*). For a minimization problem, an algorithm is said to have an approximation factor of $\alpha \geq 1$ if, for every instance, the cost of the solution it returns is at most $\alpha$ times the cost of an optimal solution. A 2 approximation algorithm, for instance, ensures that the output is no worse than a factor of 2 times the optimal.

The **Steiner tree problem** is a classic NP-hard problem in graph theory. Since finding an exact solution is computationally expensive, researchers have developed various *approximate Steiner tree algorithms*. These algorithms efficiently produce trees whose total cost is guaranteed to be within a known approximation factor of the minimum possible.

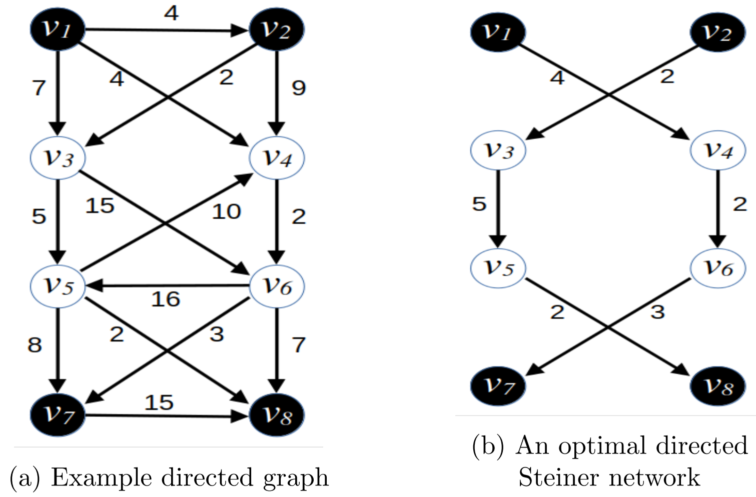
The Steiner tree problem can be addressed in two broad classes of graphs: static and dynamic.

## 1.3 The Steiner Tree Problem in Static Graphs

A graph is said to be a static graph when changes in the vertex set, the edge set, as well as changes in the weight function are not allowed. Once the graph is defined, neither the graph nor any property of the graph changes. The Steiner tree problem is an NP-complete problem [3, 4], with the best approximation factor known to be 1.39 [5]. It is important to note the inherent difficulty of solving the Steiner tree problem optimally. A well-known result shows that achieving an approximation factor better than $\frac{96}{95}$ for the Steiner tree problem in polynomial time is impossible unless $P = NP$ [2]. This implies that there is a fundamental limitation on how close we can get to the optimal Steiner tree in polynomial time. The Steiner tree problem can also be addressed in a directed weighted graph. The directed version of the problem, called the *Directed Steiner Network* problem, is defined as follows:

**Definition 1.2.** *(Directed Steiner Network Problem) Given a weighted directed graph $G = (V, E)$ with weight function $w : E \rightarrow R^+$ and $k$ demand pairs $(u_1, v_1), \ldots, (u_k, v_k) \in V \times V$, the objective is to find a minimum-weight sub-graph of $G$ such that each $u_i$ has a path to $v_i$ where $(u_i, v_i)$ is a demand pair.*

An example instance of the *Directed Steiner Network* problem is shown in Figure 1.2. The black vertices are terminals, and the white vertices are non-terminals. The directed edges are represented by arrows. The demand pairs are $(v_1, v_7)$ and $(v_2, v_8)$. The optimal solution is shown in Figure 1.2b. Another variant of the *Directed Steiner Network* problem is the *Directed Steiner Tree*, in which all demand pairs are of the form $(r, v_i)$ for some root node $r$ of the tree. An example instance of the *Directed Steiner Tree* problem is shown in

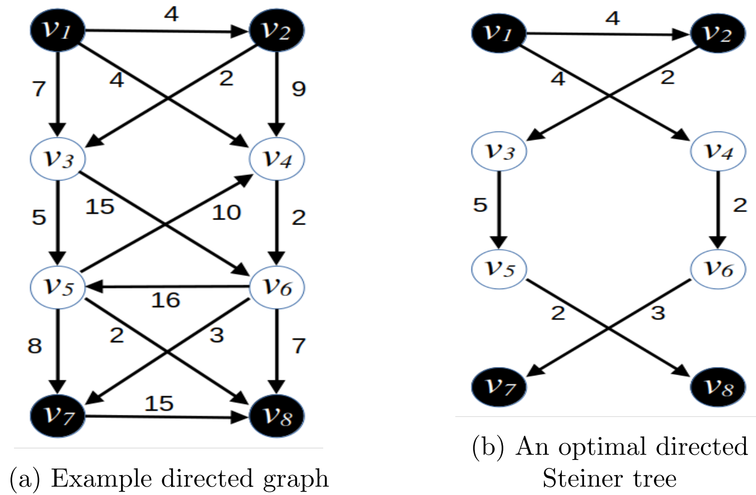



(a) Example directed graph

(b) An optimal directed Steiner network

Figure 1.2: An example directed graph and a corresponding optimal directed Steiner network.

Figure 1.3. The demand pairs are $(v_1, v_2)$, $(v_1, v_7)$ and $(v_1, v_8)$. The optimal solution is shown in Figure 1.3b.




(a) Example directed graph

(b) An optimal directed Steiner tree

Figure 1.3: An example directed graph and a corresponding optimal directed Steiner tree.

## 1.4 The Steiner Tree Problem in Dynamic Graphs

### 1.4.1 Dynamic Graphs and Dynamic Algorithms

Dynamic graphs permit changes in the edge set, the vertex set, or the weight function over time. Such graphs undergo a series of updates. An update may involve vertex or edge insertion (or deletion) in the graph. Updates to the graph might occur at any time. Harary and Gupta [6] present some dynamic graph models. The properties of the graph, like the shortest paths between vertices, routes or connectivity between the vertices, or the graph topology, may change due to updates.

Recently, significant attention has been devoted to the design of dynamic algorithms for certain graph problems. Dynamic algorithms aim to update and maintain an existing solution to the problem under consideration by updating the underlying data structure, avoiding the need for recomputation of the solution from scratch for the updated instance.

A dynamic algorithm needs to maintain the solution after each update. A dynamic algorithm is said to be incremental if it only handles insertion requests ( edge or vertex insertions in the case of graph algorithms). Similarly, a dynamic algorithm is said to be decremental if it only handles deletion requests. A dynamic algorithm is said to be fully dynamic if it handles insertion requests as well as deletion requests.

### 1.4.2 Dynamic Steiner Tree Problem

A Steiner tree in a dynamic graph, also called the *dynamic Steiner tree(DST)* problem, deals with efficiently maintaining an approximate Steiner tree within a dynamic graph. Formally, the definition of the dynamic Steiner tree is as follows:

**Definition 1.3.** *(Dynamic Steiner Tree Problem) Given a connected graph $G = (V, E)$ with positive real edge weights, a set of vertices $S \subseteq V$ called terminals, an existing approximate Steiner tree $T = (V', E')$ on $S$, along with a series of updates, the objective of the dynamic Steiner tree problem is to update and maintain $T$ to an approximate Steiner tree $T' = (V'', E'')$ such that the weight $\Sigma_{e \epsilon E''} w(e)$ of $T'$ is as minimum as possible and $T'$ spans all the terminals, that is, $S' \subseteq V'' \subseteq V$ and $E'' \subseteq E$. Here, $S'$ is the terminal set after updates.*

In general, an update may comprise the insertion into or deletion of an edge $e$ or vertex $v$ from the graph $G$. $S'$ can be the same as $S$ if there are no updates influencing the terminal set. Maintaining a Steiner tree involves

updating it to preserve its approximation factor and properties. Hence, $T'$ may or may not be identical to $T$. While considering the *dynamic Steiner tree* problem, adding or removing vertices in the graph can create new or eliminate existing connections. Inserting new edges may provide more cost-efficient routes, while deleting edges may force the Steiner tree to take alternative paths to maintain quality or retain connectivity. Changing the status of a vertex from non-terminal to terminal or vice versa can modify the set of vertices the Steiner tree needs to span. This can potentially lead to a change in the optimal Steiner tree, impacting the optimal cost.

Since some applications might use a Steiner tree while updates appear in the graph, a Steiner tree needs to be continuously updated as the underlying graph changes. Maintaining the tree ensures that it reflects the current state of the graph and provides a result having desirable quality. Recomputing the Steiner tree from scratch after every update can be computationally expensive. Maintaining an approximate Steiner tree allows for efficient updates, enabling the applications to quickly adapt to the changing graph and continue utilizing the Steiner tree for routing or other purposes. Dynamic graphs are crucial for scenarios where vertices or edges evolve, requiring effective capture and analysis of these changes.

Imase and Waxman [7] and Łacki et al. [8] studied the *dynamic Steiner tree* problem, where the dynamicity refers to the conversion of a terminal to a non-terminal and vice versa. The *dynamic Steiner tree* problem can further be divided into two variants based on the allowance of re-arrangement of edges in the previously computed tree [7]. In the rearrangeable scenario, rearrangement of existing routes in the tree is allowed as the existing nodes are either added to or removed from the terminal set. Conversely, in the non-rearrangeable scenario, existing routes within the tree cannot be modified or dynamically adjusted as the nodes are added to or removed from the terminal set. In the case where re-arrangement of edges is allowed, the updated Steiner tree may have a cost less than the cost of the current Steiner tree. Whereas in the non re-arrangement case, the cost of the updated tree after the addition of a vertex or edge can not be less than the cost of the tree computed after the previous update, i.e., the current Steiner tree. In this variant, the addition of a vertex can not result in the deletion of any existing edge, and the deletion of any vertex can not result in the addition of a new edge. As a consequence, when a vertex is added to the graph $G$, the previous tree turns out to be a sub-tree of the updated tree, and when a vertex is deleted, the updated tree turns out to be a sub-tree of the previous tree. It is *NP-Hard* to compute the exact solution of both of these variants.

## 1.5 Applications of the Steiner Tree

The dynamic Steiner tree problem is a re-optimization problem where we are given an instance of the problem, an approximate solution, and some updates to the graph. The objective is to update the existing solution to a good approximate Steiner tree for the updated instance. The dynamic algorithm saves us over the time required to recompute the solution from scratch. Steiner tree has many applications. One important application is the formation of a multicast tree for information sharing. The multicast is the problem of creating, maintaining, and updating trees that are rooted at source nodes and span a set of destination nodes, which is essentially a Steiner tree with the terminal set consisting of the source nodes and destination nodes [9]. The topology of a mobile network keeps changing as nodes join and leave a multicast network, and links may be added or deleted. It is, therefore, challenging to recompute a valid multicast tree as quickly as possible.

The Dynamic Steiner Tree (DST) problem finds applications in various scenarios. For instance, it can be used to model the multi-point routing problem within communication networks, as explored by Ding and Ishii [10], where the Steiner vertices or the vertices of the graph may change. Incrementally updating and maintaining an approximate Steiner tree is significantly more efficient than recomputing it from scratch after every update, particularly for a few updates, as the optimal and approximate Steiner trees may not change significantly. The Steiner trees are good candidates for the computation of a network that efficiently connects a set of nodes or processors that may not be residing in the nearby geographical area, like servers receiving data from sensors, where the server may be residing far from sensors, or robots that need to communicate but require coordination remotely. These networks are prone to continuous link creation or deletion as well as node addition and deletion. The number of nodes in the network with useful information also needs to be decreased to avoid security and privacy threats. The Steiner forest problem is a strict generalization of the Steiner tree problem. The Steiner tree problem is also driven by a range of applications in both physical and virtual network design, spanning from railway infrastructure planning to virtual private networks (*VPNs*) and multicast streaming scenarios [11].

Dynamic Steiner trees have broad applications, particularly in providing connectivity and services in *CPSs* (cyber-physical systems) and *WSNs* (wireless sensor networks). These networks undergo topology changes due to element movements or configuration changes in real-time, necessitating an efficient dynamic Steiner tree management. The *DST* under edge insertions or deletions acts as a building block in *CPSs* and *WSNs*.

The facilities needed for adapting the changes taking place in some specific nodes to support video broadcasts or conferences can be provided by an underlying Steiner tree designed or computed over the graph generated by the nodes and available links [7]. Applications may vary from forming multicast trees for information sharing in dynamic networks, designing peer-to-peer (*P2P*) networks, sensor networks, ad-hoc networks [12], etc., to solving complex problems such as the prize collecting Steiner tree. It is challenging to update and maintain an efficient solution in such cases.

## 1.6 Motivation

While classical static algorithms for computing exact or approximate Steiner trees have been extensively studied, the growing need for handling dynamically changing graphs, where edges and vertices may be inserted or deleted, necessitates the development of efficient dynamic algorithms for the Steiner tree problem. One may execute a static algorithm for the Steiner tree computation from scratch after each update. However, this approach has serious limitations in terms of the overall update time. Also, for a small number of updates, it is desirable to incrementally change the Steiner tree due to the updates rather than computing it from scratch since it might be used by some applications while updates are appearing in an asynchronous manner. Despite substantial progress in dynamic graph algorithms over the past decades, we observed a significant gap: there is a lack of dynamic algorithms that efficiently maintain a good approximation of a Steiner tree under edge and vertex insertions and deletions. Dynamic algorithms alleviate the need for recomputation, offering efficiency in the face of graph updates.

While prior work has addressed the *DST* problem for vertex deletions [13], and the conversion of a non-terminal to a terminal and vice versa by Imase and Waxman [7], and Lacki et al. [8], limited research has been conducted on the impact of edge insertions and deletions in this context. Therefore, proposing better dynamic algorithms for handling edge insertions and deletions remains a valuable area for future exploration. Specifically, existing literature primarily focuses on static or partially dynamic settings, often limiting updates to specific graph classes or particular types of changes, such as only terminal updates. This gap motivated our research. The best result we found for maintaining a Steiner tree under the conversion of terminals to non-terminals and vice versa is due to Lacki et al. [8] which is a $(4+\epsilon)$ approximation in planar graph in $\tilde{O}(\epsilon^{-1}\log^6 n)$ time and a $(6+\epsilon)$ approximation in $\tilde{O}(\sqrt{n}\log D)$ time in general graphs. The topology of the graph does not change under these updates.

However, the dynamic Steiner tree problem with edge insertions and deletions and vertex insertions and deletions has not yet been studied extensively. The topology of the graph changes under these kinds of updates. It is interesting as well as challenging to incrementally or dynamically maintain a good approximate Steiner tree when the topology of the underlying graph changes.

In this work, we address the *dynamic Steiner tree* problem with dynamic insertions or deletions of edges and vertices in different dynamic graph environments. This leads us to the following question:

**Question 1.** *Is it possible to design a fully dynamic algorithm to efficiently update and maintain a good approximate Steiner tree under edge insertion and deletion updates?*

We found that this problem has not been addressed and explored yet. We approach the problem by attempting to design a fully dynamic algorithm that handles edge insertions and deletions while maintaining a good approximate Steiner tree efficiently. Recognizing the inherent structural properties of planar graphs, we first targeted the planar setting. This led to the development of a fully dynamic algorithm that supports edge insertions and deletions in planar graphs, maintaining a good approximation of the Steiner tree while exploiting planarity to achieve better efficiency.

Encouraged by the success in the planar case, we next turned our attention to more general graphs. However, the complexity of maintaining dynamic Steiner trees in general graphs posed greater challenges. As the first question remains open and difficult to solve, this led us to our second question:

**Question 2.** *Is it possible to design an incremental algorithm to maintain a good approximate Steiner tree in general graphs with edge insertion updates?*

We addressed this by designing an incremental algorithm for general graphs, capable of handling edge insertions while preserving a good approximate Steiner tree.

While striving to improve the update time complexity further, we recognized a key bottleneck: the time required to update the widely used *MST* heuristic based 2-approximate Steiner tree. The simplicity of the *MST* heuristic based algorithms makes them attractive, but it became increasingly clear that reducing the update time beyond a certain threshold would require overcoming the inherent limitations of this heuristic. It led us to the following question:

**Question 3.** *What is the lower bound on the update time complexity to maintain an* MST *heuristic based 2-approximate Steiner tree when the topology of*

*the underlying graph changes?*

Many of the existing algorithms for computing an approximate Steiner tree are based on the MST heuristic based 2-approximate Steiner tree, and require maintaining a complete distance graph over terminals. Therefore, handling edge insertions or deletions becomes challenging as it can alter many edges (all edges in the worst case) in the complete distance graph. Therefore, we shifted focus to formally establish a lower bound on the update time achievable by MST heuristic based dynamic algorithms, providing theoretical justification for the observed complexity barriers.

Having explored an incremental solution in general graphs and a fully dynamic solution in planar graphs, we observed the need for decremental solutions.

**Question** 4**.** *Is it possible to design a decremental algorithm to maintain a good approximate Steiner tree with edge deletion updates?*

We set out to design decremental algorithms that efficiently handle edge deletions in planar graphs. We also realized that a comprehensive solution demands algorithms that can handle not only edge insertion and deletion updates but also vertex insertion and deletion updates and terminal to non-terminal conversions and vice versa, making them more flexible and practically applicable.

**Question** 5**.** *Is it possible to design a fully dynamic algorithm to efficiently update and maintain a good approximate Steiner tree under edge insertions, edge deletions, vertex insertions, vertex deletions, terminal to non-terminal conversion, and non-terminal to terminal conversion?*

Consequently, the final phase of our research focused on developing two dynamic algorithms, one for planar graphs and another for general graphs, capable of efficiently maintaining approximate Steiner trees under all types of updates: edge insertions and deletions, vertex insertions and deletions, and terminal updates. These algorithms achieve both flexibility and efficiency, addressing the practical needs of dynamic network environments.

In summary, our research systematically bridges the existing gap by progressively designing dynamic algorithms that balance approximation guarantees and update efficiency, culminating in flexible solutions that handle comprehensive update operations across both planar and general graphs.

## 1.7 Objectives

With the identified research gaps, we aim to design dynamic algorithms handling edge insertion and deletion updates in planar and general graphs, and then incorporate vertex insertion and deletion updates. The Steiner tree in our problem differs from the Steiner tree in [7], where the previous Steiner tree is a sub-tree of the updated Steiner tree or vice versa (depending on insert/delete request). Our work does not require these conditions to be satisfied.

**Objective I (*A fully dynamic algorithm in planar graphs*):** As the *dynamic Steiner tree* problem is not studied in dynamic graphs undergoing edge insertion and deletion updates, our first target is to design an algorithm in planar graphs to maintain a Steiner tree under these updates. We aim to propose a fully dynamic algorithm for handling edge insertions and deletions in planar graphs. The objective is to maintain a 2-approximate Steiner tree in planar graphs under edge insertions and deletions.

**Objective II (*An incremental algorithm in general graphs*):** As we realize the challenges in developing a fully dynamic algorithm in general graphs, we address the problem in an incremental setting in an attempt to give a dynamic algorithm for general graphs. We target to propose an incremental algorithm for maintaining a 2-approximate Steiner tree in general graphs. To update the Steiner tree efficiently, we propose an incremental rather than a fully dynamic algorithm.

While trying to improve the update time complexity of our algorithms, we observed the inherent complexity of the problem and the existing techniques. Hence, we aim to propose a lower bound as our next objective as follows:

**Objective III (*A lower bound on update time*):** We identified that various algorithms for computing or maintaining a Steiner tree are based on an MST heuristic. We planned to introduce a lower bound on the update time complexity for maintaining a Steiner tree for MST heuristic based algorithms.

**Objective IV (*An efficient dynamic algorithm in planar graphs*):** In an urge to design a dynamic algorithm in planar graphs with a better update time, we propose a decremental algorithm in planar graphs to improve the update time complexity as compared to the proposed fully dynamic algorithm.

While striving to achieve a fully dynamic algorithm for the *dynamic Steiner tree* problem in general graphs, we aim to incorporate vertex insertion

and deletion updates as well as terminal to non-terminal conversion and vice versa. Hence, our next objective is as follows:

**Objective V (*Handling vertex insertion and deletion updates*):** We aim to propose two fully dynamic algorithms, one in planar graphs and the other one in general graphs. The objective is to handle six different types of updates together: edge insertions and deletions, vertex insertions and deletions, and conversion of terminal to non-terminal and vice versa efficiently.

## 1.8 Contribution of the Thesis

We investigate the dynamic Seiner tree problem for edge insertion and deletion updates in planar and general graphs. We propose a lower bound on the MST heuristic based algorithms to maintain a 2-approximate Steiner tree under edge insertions and deletions. Together with edge insertion and deletion updates, we also propose two algorithms to handle vertex insertions and deletions and terminal to non-terminal conversion and vice versa.

### 1.8.1 A Fully Dynamic Algorithm in Planar Graphs

We present a fully dynamic algorithm that maintains a $(2+\epsilon)$-approximate Steiner tree under a sequence of edge insertions and deletions in a planar graph for $0 \leq \epsilon \leq 1$. The algorithm takes the help of a fully dynamic approximate distance oracle which also works in a planar graph. The update time required by the dynamic algorithm is $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$ in the worst case. Here $S$ is the set of terminals, $n$ is the number of vertices and $D$ is the unweighted diameter of the graph. This is significantly better than executing the static algorithm from scratch (requiring $O(|S|V^2)$ time) as proposed by L. Kou et al. [14].

Our algorithm is fully dynamic in the sense that it can handle an arbitrary sequence of edge insertions as well as edge deletions. This is the first work in this direction. We maintain a fully dynamic planar distance oracle on the graph under consideration to efficiently compute approximate shortest paths between terminals. The algorithm works in four steps. In the first step, following a series of updates, the algorithm utilizes the distance oracle to maintain a complete distance graph, denoted as $G_1$, on the terminal set $S$. In $G_1$, the distances between any pair of terminals $u$ and $v$ are approximate representations of their actual distances in the original graph $G$. In the second step, the algorithm constructs a minimum spanning tree, denoted as $T_1$, on the complete distance graph $G_1$. The third step substitutes the edges of

$T_1$ with the approximate shortest paths obtained from the dynamic distance oracle. In the fourth step, the algorithm removes any cycles and non-terminal leaves from the resulting graph to obtain the updated $(2+\epsilon)$-approximate Steiner tree.

Later, we show that the update time of the presented algorithm becomes $O(n(\epsilon')^{-2})$ for some $0 \leq \epsilon' \leq 1$ when $|S|$ and $D$ are logarithmic in $n$. The update time for this case is linear in the number of vertices in the graph.

### 1.8.2 An Incremental Algorithm in General Graphs

We propose an incremental algorithm for maintaining an approximate Steiner tree in general graphs. This algorithm specifically addresses dynamic updates where only edge insertions are allowed. The key objective of the algorithm is to efficiently maintain a $(2-\epsilon)$-approximate Steiner tree. The update time for processing an edge insertion is $O(nD_s)$. The parameter $D_s$ is defined in Chapter 3.

The algorithm creates and maintains a shortest path forest such that each terminal is a root of a shortest path tree in the shortest path forest. The shortest path forest is updated after each edge insertion update and it helps in identifying the updated shortest paths among the terminals. We take the help of the data structures used by Dial [15] to maintain the shortest path forest efficiently. The updated shortest paths among the terminals are utilized from the shortest path forest to update the Steiner tree. The algorithm also handles the cases where the inclusion of a shortest path in the Steiner tree may lead to the creation of a cycle. This issue is addressed by identifying the heaviest path connecting any two terminals in the cycle and removing it without affecting the connectivity of other parts of the tree.

### 1.8.3 A Lower Bound

Kou et al. [14] show that computing MST of the complete distance graph (metric closure) over terminals followed by replacement of the edges of MST with the original shortest path between terminals in the graph avoiding cycles and pruning non-terminal leaves gives a $(2-\epsilon)$-approximate Steiner tree where $\epsilon$ is $2/|S|$. This algorithm is also known as the MST heuristic based $(2-\epsilon)$-approximate Steiner tree. We establish a lower bound of $\Omega(n)$ on the update time required by MST heuristic based algorithms to maintain a $(2-\epsilon)$-approximate Steiner tree in a general graph undergoing edge insertion or edge deletion updates.

### 1.8.4 A Decremental Algorithm in Planar Graphs

We present a decremental algorithm for maintaining a $(2+\epsilon)$-approximate Steiner tree in weighted planar dynamic graphs with edge deletion updates. We initially take a $(2-\epsilon')$-approximate Steiner tree. After each update in the graph, we update the Steiner tree to maintain a $(2+\epsilon)$-approximate Steiner tree. The presented algorithm maintains the $(2+\epsilon)$-approximate Steiner tree in $\tilde{O}(\ell\sqrt{n})$ update time. Here, $\ell$ is the maximum number of hops in a $(1+\epsilon'')$-approximate shortest path between any two vertices in a graph. This is significantly better than executing the static algorithm from scratch as proposed by Kou et al. (requiring $O(|S||V|^2)$ time) [14] and Wu et al. (requiring $O(|E|\log|V|)$ time) [16].

We use a dynamic distance oracle to compute approximate shortest distances efficiently and an $ET$-tree data structure to avoid cycle formation in the approximate Steiner tree. We maintain an auxiliary graph to efficiently use the distance oracle. The auxiliary graph is designed such that a single query of the distance oracle for the distance between any two arbitrary vertices belonging to two different trees gives a path that is an approximation of the minimum of the shortest paths among all the shortest paths between any two vertices in the two trees.

### 1.8.5 A Fully Dynamic Algorithm Handling Edge and Vertex Insertions and Deletions and Terminal Conversions in Planar Graphs

We propose a novel approach to maintain an approximate Steiner tree in fully dynamic undirected weighted planar graphs. The proposed algorithm works for dynamic graphs where the updates can be the insertion or deletion of a vertex from the graph, the insertion or deletion of an edge from the graph, or the conversion of a non-terminal to a terminal and vice-versa. The conversion of a non-terminal to a terminal and a terminal to a non-terminal is equivalent to the insertion and deletion of a vertex from the terminal set, respectively. After each update, the algorithm maintains a $(1+\epsilon)$-approximate MST and a $(|S|-1)$-approximate Steiner tree, where $S$ is the set of terminals.

We also prove that the total weight of any spanning tree of the complete distance graph on terminals is at most $|S|-1$ times the total weight of the optimal Steiner tree for the terminal set $S$. To the best of our knowledge, the proposed algorithm is the first algorithm that allows these six types of graph updates and maintains a $(|S|-1)$-approximate Steiner tree in planar graphs. The expected update time of the proposed algorithm is $\tilde{O}((\delta p\log U)/\epsilon + |S|\cdot\sqrt{n}+n)$ for $p$ number of updates. For $|S|=o(\sqrt{n})$, the update time of the

presented algorithm becomes linear in $n$.

The proposed algorithm employs a fully dynamic distance oracle, the $ET$-trees data structure, and a $(1+\epsilon)$-approximate MST algorithm given by Henzinger and King [17] that maintains an approximate MST under edge insertion and deletion updates. The distance oracle is maintained on the graph under consideration. The algorithm works in four steps. After an update, the algorithm first performs the update operation on the distance oracle. In the second step, the algorithm updates the approximate MST under these updates based on the cut property of MST and the algorithm of Henzinger and King [17]. In the third step, the algorithm finds $(|S|-1)$ Neighbor terminal pairs in the approximate MST with the help of a DFS procedure on the approximate MST. A neighbor terminal pair is defined as a pair of terminals such that the path between the two terminals does not contain any other terminal. In the fourth step, the algorithm updates the $(|S|-1)$-approximate Steiner tree by connecting $(|S|-1)$ neighbor terminal pairs by approximate shortest paths.

### 1.8.6 A Fully Dynamic Algorithm Handling Edge and Vertex Insertions and Deletions and Terminal Conversions in General Graphs

We propose a fully dynamic algorithm for maintaining an approximate Steiner tree in general graphs, supporting six types of updates: edge insertion, edge deletion, vertex insertion, vertex deletion, terminal to non-terminal conversion, and non-terminal to terminal conversion. By integrating dynamic clustering, spanner-based connectivity, and a hybrid distance oracle, the algorithm achieves an approximation factor of $2+\epsilon$ for some tunable $\epsilon > 0$. Furthermore, it achieves efficient updates with an expected time complexity of $O(m^{1/2}+n^{2/3})$ per update. The algorithm offers a significant improvement over existing dynamic Steiner tree algorithms in terms of flexibility, approximation guarantees, and update efficiency.

The graph is decomposed into smaller clusters using low-diameter decomposition. This decomposition divides the graph into subgraphs (clusters) with bounded diameters, ensuring that distances between any two nodes in a cluster are small. This property allows Steiner trees to be efficiently computed locally within clusters. We use the spanner construction technique proposed by Thorup and Zwick [18], which provides a $(1+\epsilon)$ approximation for distances while ensuring sparsity. This spanner construction is applied to the clusters obtained through low-diameter decomposition, ensuring that terminal-to-terminal distances across clusters are well-approximated. We augment the decremental distance oracle proposed by Bernstein and Roditty [19]

to handle additional types of updates, including edge insertions. The hybrid oracle is maintained over the inter-cluster spanner, while edge updates are applied to the original dynamic graph. This ensures that the oracle efficiently tracks terminal-to-terminal distances during dynamic updates.

The total complexity for processing a single update combines the contributions from hybrid distance oracle updates, spanner updates, and Steiner tree updates. Hybrid distance oracle updates dominate with $O(m^{1/2})$ complexity for edge modifications, while vertex updates contribute $O(\log n)$. Spanner updates are bounded by $O(\log n)$ due to sparsity, and intra-cluster Steiner tree updates using Mehlhorn's algorithm scale efficiently with cluster size. Combining these contributions, the overall expected time complexity for processing any dynamic update is: $O(m^{1/2} + n^{2/3})$.

## 1.9 Outline of the Thesis

The thesis is organized as follows.

**Chapter 1:** This chapter gives an introductory overview of the Steiner tree problem and the dynamic graphs. The different configurations and versions of the Steiner tree problem are discussed in different environments, including planar graphs, general graphs, graphs with bounded treewidth, and dynamic graphs with terminal conversions. Several existing results and algorithms for the Steiner tree problem in different static graphs and dynamic graphs are discussed.

**Chapter 2:** This chapter explores recent works and techniques used to solve the Steiner tree problem in various static and dynamic graph configurations. This chapter also explores some dynamic graph algorithms applied to other dynamic graph problems. It gives an understanding of the challenges in maintaining a solution due to updates in the underlying graph, some of the techniques used in dynamic graph algorithms, and summarizes the research gap, the motivation and the research contributions made to the thesis.

**Chapter 3:** This chapter provides a detailed description of the first contribution to the thesis. It includes a detailed description of the fully dynamic algorithm maintaining a $(2 + \epsilon)$-approximate Steiner tree under edge insertions and deletions in planar graphs and the incremental algorithm to maintain a $(2 - \epsilon)$-approximate Steiner tree under edge insertions in general graphs. It also explores a scenario where the fully dynamic algorithm is more efficient in terms of the update time complexity.

**Chapter 4:** In this chapter, first, we prove and establish a lower bound on the time required to maintain an MST heuristic based $(2-\epsilon)$-approximate Steiner tree followed by a detailed discussion of the decremental algorithm maintaining a $(2+\epsilon)$-approximate Steiner tree in planar graphs that has an improved update time complexity as compared to the update time complexity of the fully dynamic algorithm given in Chapter 1.

**Chapter 5:** This chapter revolves around the fully dynamic algorithms handling six types of updates: edge insertion, edge deletion, vertex insertion, vertex deletion, non-terminal to terminal conversion, and terminal to non-terminal conversion. This chapter consists of four contributions. The discussion starts with the multiple techniques used to maintain the solution. The first algorithm in this chapter handles the updates in planar graphs and maintains an $(|S|-1)$-approximate Steiner tree. Later, we show that any spanning tree of the complete distance graph over the terminals is an $(|S|-1)$-approximate Steiner tree. To produce the second algorithm, we augment an existing decremental distance oracle to create a fully dynamic distance oracle. We use this fully dynamic distance oracle together with clustering and spanner techniques to design a fully dynamic algorithm for general graphs that maintains a $(2+\epsilon)$-approximate Steiner tree.

**Chapter 6:** We conclude the thesis with a conclusive discussion of the research done under the thesis, where the achievements, pros, and cons of the algorithms and techniques developed in the research are summarized. Several possibilities and directions for further research are discussed based on the cons of the techniques developed and the existing research gap.

# 2

# Related Work

The field of dynamic algorithms has witnessed significant research attention over the past few decades due to its profound theoretical depth and wide-ranging practical applications. Dynamic algorithms focus on efficiently updating the solution to a problem as the underlying input undergoes modifications, such as insertions or deletions, without recomputing from scratch. This area is particularly fascinating because it blends classic algorithmic techniques with innovative approaches to handle ever-changing data efficiently. At the same time, designing dynamic algorithms poses unique challenges, as it often requires balancing update time, query efficiency, and approximation guarantees under complex update sequences. As real-world systems increasingly demand real-time responsiveness, dynamic algorithms continue to be both an intellectually stimulating and practically essential research area.

This chapter presents an overview of existing research and recent advancements in the field of dynamic algorithms and both static and dynamic versions of the Steiner tree problem, highlighting key developments and techniques across various settings and emphasizing the progress made and the challenges that remain. By examining these works, the chapter sets the foundation for understanding the context and motivation behind the contributions.

## 2.1 Dynamic Algorithms

Hanauer et al. [20] surveyed recent developments in fully dynamic Algorithms for some graph problems and presented a spectrum of dynamic graph algorithms proposed for various graph problems. Holm et al. [21] made significant contributions to the field of dynamic graph algorithms by introducing efficient, fully dynamic algorithms for several key graph problems. These

algorithms achieve a poly-logarithmic deterministic update time complexity. These problems include the connectivity problem, the biconnectivity problem, the 2-edge connectivity problem, and the minimum spanning tree problem. These algorithms are designed to handle graphs that initially do not contain any edges and undergo continuous updates, where edges are added and removed over time. The authors designed an algorithm capable of maintaining a spanning forest. The amortized update time of the presented algorithm is $O(\log^2 n)$ per update. This algorithm enables answering connectivity queries in $O(\log n / \log \log n)$ time. Additionally, an algorithm is presented to maintain a minimum spanning forest. The amortized update time of the algorithm is $O(\log^4 n)$ per operation. This work presents a fully dynamic algorithm for maintaining 2-edge connectivity within a graph. This algorithm boasts an amortized update time complexity of $O(\log^4 n)$ per operation. Similarly, for the fully dynamic biconnectivity problem, they proposed an algorithm with operations supported in $O(\log^5 n)$ amortized time per operation. Here, $n$ is the number of vertices in the graph.

Henzinger and King [17] proposed a dynamic algorithm to maintain an MST for $k$ weighted graphs under edge insertions and deletions in $O(pk \log^3 n)$ expected update time for $p$ number of updates. The proposed algorithm is extended to maintain a $(1 + \epsilon')$-approximate MST in $O((p \log^3 n \log U)/\epsilon')$ expected time for $p$ number of updates where edge weights are in $[1, U]$ and $\epsilon'$ is a small fraction.

## 2.2 The Static Steiner Tree Problem

Karp [3] and Garey and Johnson [22] independently demonstrated that the Steiner tree problem is an *NP-complete* problem. Since then, the hunt for good approximation algorithms for the Steiner tree problem has continued. Some authors worked to find an exact algorithm for the Steiner tree problem before it was known to be *NP-complete*. Winter [23] surveyed the Steiner tree problem, where the author discussed various Steiner tree algorithms. Melzak [24] gave an exact algorithm for the Steiner tree problem in the Euclidean plane. Gilbert and Pollak [25] surveyed the Steiner problem in the Euclidean plane, presenting work done on the Steiner tree problem in the Euclidean plane up to 1968. Chang [26] provided some heuristics on the generation of minimal trees with a Steiner topology. The most efficient heuristic of $O(n \log n)$ time complexity is due to Smith et al. [27], which produces good results.

Chlebík and Chlebíková [2] showed that it is *NP-hard* to approximate the Steiner tree problem within a factor of $\frac{96}{95}$. It is also mentioned that the best approximation till 2018 was achieved by Robins and Zelikovsky [28] with

an approximation factor of 1.550, and 1.279 for *quasi-bipartite* instances. The Steiner tree instance is said to be *quasi-bipartite* if there is no edge among the non terminals [2].

**Theorem 2.1.** *(Chlebík and Chlebíková [2]) Given an integer* $d \geq 3$*, let* $q(d) = min\{\frac{c(d)-1}{2c(d)}, \frac{1}{4}\}, r(d) = 1 + \frac{q(d)}{3(d+1-q(d))}$*, where* $c(d)$ *is the constant defined in the following way:*

$$c(d) = \sup\{c : there\, are\, infinitely\, many\, d\text{-}regular\, c\text{-}good\, expanders\}.$$

*Then, for any constant* $r, 1 < r < r(d)$ *, it is NP-hard to approximate the optimal solution of the Steiner tree problem within a factor* $r$*.* □

In particular, since $c(6) > 1.76222$ implies $r(6) > 1.01063$, inapproximability within a factor of 1.01063 ($> \frac{96}{95}$) follows for the Steiner tree problem unless $P = NP$.

An LP based solution for the $k$-restricted Steiner tree problem is given by Byrka et al. [5]. It is a Steiner tree problem such that each component can have at most $k$ terminals. Byrka et al. applied LP relaxation followed by randomization to obtain a $(\ln(4) + \epsilon)$ approximation for the $k$-restricted Steiner tree problem and later extended the work by applying derandomization to the previous work. As a result, there is a deterministic algorithm with an approximation factor of $(\ln(4) + \epsilon)$ for the Steiner tree problem.

**Theorem 2.2.** *(Byrka et al. [5]) For any* $k = O(1)$ *and any constant* $\epsilon > 0$*, there is a polynomial time deterministic* $(\ln(4) + \epsilon)$ *approximation algorithm for the* $k$*-restricted Steiner tree.* □

**Theorem 2.3.** *(Byrka et al. [5]) For any constant* $\epsilon > 0$*, there is a polynomial-time deterministic* $(\ln(4) + \epsilon)$ *approximation algorithm for the Steiner tree problem.* □

Chimani et al. [29] came up with a Steiner tree algorithm for bounded treewidth that solves the exact Steiner tree problem in $O(B^2_{tw+2}.tw.|V|)$ time where $tw$ is treewidth, bell number $B_k$ is the number of partitions of a set with $k$ elements. The authors applied a dynamic programming paradigm with some numbering and coloring schemes.

**Theorem 2.4.** *(Chimani et al. [29]) Given a graph with vertex set* $V$ *and a tree decomposition with treewidth* $tw$*, the Steiner tree problem can be solved to optimality in* $O(B^2_{tw+2}.tw.|V|)$ *time.* □

**Theorem 2.5.** *(Chimani et al. [29]) Given a graph with vertex set* $V$ *and a tree decomposition with treewidth* $tw$*, the prize-collecting Steiner tree problem can be solved to optimality in* $O(B^2_{tw+2}.tw.|V|)$ *time.* □

The bell number and treewidth are not directly dependent on the input size and may have large values.

Kou et al. [14] presented an algorithm to compute a $2\big(1-\frac{1}{|S|}\big)$-approximate Steiner in general graphs. The proposed algorithm computes a metric closure on the terminals followed by MST construction. The authors show that the MST heuristic used by the algorithm computes a $2\big(1-\frac{1}{|S|}\big)$-approximate Steiner in $O(|S||V|^2)$ time, where $S$ is the set of terminals and $V$ is the set of vertices in the graph. Wu et al. [16] presented a similar algorithm to compute a $2\big(1-\frac{1}{\ell}\big)$-approximate Steiner in a general graph in $O(|E|\log|V|)$ time. Here, $\ell$ is the minimum number of leaves in any optimal Steiner tree. The algorithm computes shortest path trees from terminals as roots using a minimum priority queue, which holds tuples containing information about the distance of vertices from terminals and uses the shortest paths among the terminals to get the final Steiner tree.

Borradaile et al. [30] proposed a polynomial time approximation scheme *(PTAS)* for the Steiner tree problem in a planar graph. The proposed algorithm computes a $(1+\epsilon)$-approximate Steiner tree in $O(2^{poly(1/\epsilon)}n + n\log n)$ time. The technique proposed works on static planar graphs rather than dynamic graphs, and the procedure used works on the assumption that the input graph has a degree at most 3. The algorithm is a remarkable advancement in this domain.

The Steiner tree problem in rectangular grid graphs is known as the rectilinear Steiner tree problem. Rectangular grid graphs form a subclass of planar graphs. Given a rectangular grid graph and a set of terminals, the objective of the rectilinear Steiner tree problem is to find a rectilinear tree connecting all terminals. Hwang [31] presented an $O(n\log n)$ time algorithm for rectilinear minimal spanning trees. The algorithm is based on computing Voronoi diagrams for rectilinear distances. The author also showed that the given time complexity is the lower bound to compute Voronoi diagrams and rectilinear minimal spanning trees. Zelikovsky [32] gave an algorithm for the Steiner problem with rectilinear distances. The presented algorithm improves an existing MST algorithm and produces an (11/8)-approximate Steiner tree in $O(|S|^3)$ time.

Recent progress on the Steiner tree problem has focused on improving approximation guarantees, tightening lower bounds, and expanding tractability through structural parameterization.

Bernardelli et al. [33] revisited the metric Steiner tree problem and analyzed the integrality gap of its linear programming relaxation. They proposed the Complete Metric (CM) formulation, which generalizes the bi-directed cut formulation and leads to stronger dual bounds. Their computational exper-

iments show that the CM model yields tighter LP relaxations in practice, thereby reducing the optimality gap between fractional and integral solutions on benchmark instances.

Ahmadi et al. [34] addressed the Prize-Collecting Steiner Tree (PCST) problem, where terminals may be omitted at the cost of incurring penalties. They presented a 1.7994 approximation algorithm using iterative rounding and a primal-dual framework, improving the best-known approximation factor for PCST. The algorithm is based on a novel LP formulation and carefully designed rounding schemes. While the exact time complexity is not explicitly stated, the focus is on closing the gap between algorithmic and hardness lower bounds under standard complexity assumptions.

Bojikian and Kratsch [35] considered the classical Steiner tree problem parameterized by clique-width $k$. They developed a randomized Monte-Carlo algorithm with a running time of $3^k \cdot n^{O(1)}$. This algorithm matches known lower bounds under the Strong Exponential Time Hypothesis (SETH), making it tight. The primary challenge addressed is that clique-width generalizes many hard graph classes, and dynamic programming approaches must carefully handle label expressions. Their work closes the algorithmic gap between upper and lower bounds in this parameterized regime.

Jansen and Swennenhuis [36] studied parameterized variants of the Steiner tree problem using structural parameters smaller than the number of terminals. In particular, they focused on multiway cut size and the newly introduced $K$-free treewidth parameter. They designed fixed-parameter tractable (FPT) algorithms with running times $2^{O(k)} \cdot n^{O(1)}$, where $k$ is the parameter (e.g., multiway cut size). A key challenge lies in expressing connectivity and penalty constraints within reduced-width tree decompositions, which they overcome by careful encoding of terminal separation. Their results expand the range of practical instances that can be solved efficiently beyond terminal-count-based methods.

There remains a notable lack of dedicated studies on the problem within grid graphs. Grid graphs are discrete, with vertices constrained to lattice points, and are typically studied under combinatorial or metric conditions (e.g., Manhattan distance). The problem of finding Steiner trees on grid graphs is more structured but also different in nature—discretized, combinatorial, and often with different algorithmic approaches compared to the rectilinear Steiner tree. Algorithms explicitly designed for grid graphs with theoretical guarantees appear to be an open or underexplored problem. Beyond planar grids, grids in higher dimensions or irregular sparse grids present additional complexities: the combinatorial complexity might increase, embeddings become less structured, and existing planar graph algorithms may not

extend easily. Hence, fully dynamic approximate Steiner algorithms for general grids remain an open and interesting research direction, with promising areas for exploration inspired by planar cases.

Together, these contributions advance the field by tightening approximability, matching lower bounds, and extending algorithmic reach to more expressive structural settings. While the algorithms presented in this section compute the solution from scratch, our algorithms are dynamic in nature, i.e., they update an existing solution using some dynamically updated data structures after each update in the underlying graph, and hence, the update time is better than recomputing the solution from scratch.

## 2.3 The Dynamic Steiner Tree Problem

The dynamic Steiner tree problem, as initially proposed by Imase and Waxman [7], addresses the challenge of an evolving terminal set within a static graph topology. In this context, dynamicity refers to the transformation of terminals into non-terminals and vice versa. The authors delineate two variants of this problem: the rearrangeable Steiner tree *(DST-R)* and the non-rearrangeable Steiner tree *(DST-N)*. In the rearrangeable scenario *(DST-R)*, rearrangement of existing routes in the tree is allowed as the existing nodes are either added to or removed from the terminal set. Imase and Waxman devise a polynomial time algorithm that produces a $4\delta$-approximate Steiner tree, where the solution is a $\delta$ edge-bounded extension tree. A tree $T$ is said to be $\delta$ edge-bounded if for every pair of vertices $u, v \in T$, $\forall_{e \in p(u,v,T)} cost(e) \leq \delta \cdot dist(u, v)$ where $p(u, v, T)$ denotes the set of edges in the path between $u$ and $v$ in $T$ and $dist(u, v)$ denotes the weight of a shortest path between $u$ and $v$. A tree $T(V, E)$ is said to be an extension tree for a node set $S$ if $S \subseteq V$ and the degree of all nodes in $V \setminus S$ is greater than 2 in $T$. This algorithm offers a reasonable approximation while allowing for some flexibility in route adjustment. Conversely, in the non-rearrangeable scenario, existing routes within the tree cannot be modified or dynamically adjusted as the nodes are added to or removed from the terminal set. For the non-rearrangeable version *(DST-N)*, no edge can be added to the tree when a node is removed, and no edge can be removed when a node is added. The authors demonstrate the impossibility of the existence of an algorithm that can compute a solution costing at most $\frac{1}{2} \log n$ times the cost of an optimal solution, assuming complete rearrangement. However, the authors show the existence of a polynomial time algorithm for the non-rearrangeable Steiner tree. The presented algorithm achieves a performance within twice this bound in the worst case, providing a practical solution for scenarios where rearrangement is not allowed. Our algo-

rithms work on the rearrangeable version of the dynamic Steiner tree problem, without a restriction on the solution being an edge-bounded tree. Furthermore, our algorithms handle multiple types of updates that alter the topology of the underlying graph, and thus, they can handle more complex updates while maintaining a better approximation factor.

Lacki et al. [8] propose an algorithm for maintaining a Steiner tree in planar graphs. This algorithm accommodates the conversion of terminals into non-terminals and vice versa. They introduced a $(2+\epsilon)$-approximate and a $(4+\epsilon)$-approximate Steiner tree algorithm in planar graphs in $\tilde{O}(\sqrt{n}\log D)$ update time and $\tilde{O}(\epsilon^{-1}\log^6 n)$ amortized update time, respectively. The technique used was vertex-to-label distance oracle construction, with the help of which multiple distance calculating operations can be supported. These operations include computing an approximate distance of the nearest active vertex from any vertex $v$, an approximate distance of the nearest active vertex not colored with some specific color from any vertex $v$, activating vertices of a particular color, etc., followed by some edge replacement technique to select cheaper edges after updating the terminal set.

**Theorem 2.6.** *(Lacki et al. [8]) For any constant $\epsilon > 0$, there exists a fully dynamic algorithm for the Steiner tree problem in planar graphs that uses $O(\epsilon^{-5}n^{1.5}\log^{1.5} n\log^2(\epsilon^{-1}n)\log D)$ space and $O(\epsilon^{-5}n^{1.5}\log^{1.5} n\log^2(\epsilon^{-1}n)\log D)$ preprocessing time, and maintains a $(4+\epsilon)$ approximation of the minimum-cost Steiner tree on the terminal set, performing updates in amortized time $\tilde{O}(\epsilon^{-1}\log^6 n)$.* □

Here, $n$ is the total number of vertices in the graph and $D$ is the stretch of the metric induced by $G$. Lacki et al. extended their approach to general graphs, offering a $(6+\epsilon)$-approximate Steiner tree solution in $\tilde{O}(\sqrt{n}\log D)$ update time under the same kinds of updates. The presented algorithm relies on an approximate distance oracle, originally proposed by Thorup and Zwick [37]. The space and time complexity of the oracle construction depends on the graph size and a parameter $k (k \geq 1)$. Specifically, the space requirement scales as $O(kn^{1+1/k})$. The time complexity for constructing the oracle scales as $O(kmn^{1/k})$. The distance oracle can respond to approximate distance queries in $O(k)$ time for some $k \geq 1$, where the approximate distance returned has a stretch of at most $2k-1$. This stretch factor ensures that the approximate distance is within a certain factor of the actual distance, providing a reliable estimation for the Steiner tree problem. Our algorithms improve upon these results by providing a better approximation factor and handling more complex updates that change the graph topology. The fully dynamic $(2+\epsilon)$-approximate algorithm of Lacki et al. [8] for planar graphs

handles terminal insertions and deletions in the terminal set and outperforms our algorithms in terms of the update time complexity. Whereas our fully dynamic algorithms for the planar graph outperform their $(2+\epsilon)$-approximate algorithm in handling multiple types of updates, including edge and vertex insertions and deletions that alter the graph's topology. Most of our algorithms maintain a $(2-\epsilon)$-approximate or $(2+\epsilon)$-approximate Steiner tree, which is significantly better than the approximation factors of the algorithms presented by Lacki et al. [8]. Moreover, two of our algorithms handle six kinds of updates in the underlying graph, which is more complex than the updates handled by Lacki et al. [8].

Gupta and Kumar [13] delve into the online Steiner tree problem with vertex deletions, introducing innovative solutions to address this dynamic scenario. They propose an online decremental algorithm designed to handle changes in the vertex set of a Steiner tree with a constant number of modifications in the edge set of a Steiner tree in the worst case. This algorithm efficiently adjusts the tree structure to accommodate vertex deletions while maintaining a near-optimal solution. Additionally, Gupta and Kumar present a fully dynamic constant competitive algorithm capable of handling both vertex insertions and deletions in an amortized constant number of changes. This algorithm ensures that the quality of a Steiner tree is maintained despite the evolving nature of the graph. Furthermore, they demonstrate that including higher-degree deleted vertices in the tree does not significantly increase the cost of a tree, and the number of higher-degree vertices is small, affirming the practicality and effectiveness of their algorithm in real-world scenarios. While this work addresses vertex insertions and deletions, our algorithms handle a broader range of updates, including edge insertions and deletions, providing a more comprehensive solution to the dynamic Steiner tree problem. Also, this work considers the number of changes in the edge set of the Steiner tree as a measure of the complexity, while our work considers the standard notion of time complexity (number of elementary steps in the algorithm).

Balev et al. [38] extend the classical Steiner tree problem to dynamic graphs, framing two main connectivity requirements: (1) *instantaneous connectivity*—where all terminals must be connected in every single snapshot, and (2) *journey-based connectivity*—where only the existence of a temporal path between terminals during the graph's evolution is needed. To address realistic scenarios where continual full connectivity may be too restrictive, they introduce the *partially connected model*, a relaxation in which the solution set is required only to connect each terminal to at least one other terminal at every time step. This means the terminals may be split into several connected components in some snapshots, as long as every component contains at least

one terminal and there are no isolated non-terminals.

Within this dynamic, partially connected model, they prove even with two terminals the *dynamic minimum Steiner set problem (DMSS)* remains NP-hard—a stark contrast to the static case. Their proposed exact algorithm for *DMSS* operates in time $\Theta(T \times (n-s)^{k-s} \times k^2/(k-s)!)$ for $n$ nodes, $s$ terminals, Steiner set size $k$, and $T$ time steps. Their experiments demonstrate feasible computation for up to 100 nodes and moderate parameter values. These results illustrate not only the added complexity under persistent connectivity demands but also the necessity of new heuristics and approximation approaches for scalable temporal network design. While this work focuses on the partially connected model, our algorithms operate under the instantaneous connectivity model with various dynamic updates, providing stronger connectivity guarantees.

Chan et al.[39] present the first fully dynamic algorithm for Euclidean Steiner trees, allowing arbitrary point insertions and deletions in $\mathbb{R}^d$. The algorithm maintains a $(1+\epsilon)$-approximate minimum-cost solution after each update by dynamically extending Arora's PTAS-style quadtree-based geometric dynamic programming framework. Updates affect only a sublinear number of subproblems in the dynamic program, and the tree structure is maintained implicitly, enabling efficient membership and traversal queries.

For any $0 < \epsilon, \delta < 1$ and any update sequence, the amortized update time per insertion or deletion is $O\big(\log(t/\delta)\big) \cdot (\log n)^{O(d^{2d}\epsilon^{-d})} \cdot 2^{O(4^d d^{4d}\epsilon^{-2d})}$, where $t$ is the update number, $n$ is the current number of terminals, $d$ is the Euclidean dimension, and $\epsilon, \delta$ are approximation and failure parameters chosen by the user, respectively. The membership queries and neighbor queries run in deterministic time $O(2^d \log n)$ and $(\log n)^{O(d^{2d}\epsilon^{-d})} \cdot 2^{O(4^d d^{4d}\epsilon^{-2d})}$, respectively. With probability at least $1-\delta$, the algorithm maintains a $(1+\epsilon)$-approximate Steiner tree at all times. This fully dynamic, high-accuracy, sublinear-time algorithm represents a substantial advancement in geometric network optimization. While this work focuses on Euclidean spaces, our algorithms are designed for general and planar graphs, broadening their applicability to a wider range of network structures. Additionally, two of our algorithms support six types of updates.

## 2.4 The Steiner Tree Problem in Distributed Settings

The Steiner tree problem has been extensively studied in distributed computing models, particularly in the CONGEST and CONGESTED CLIQUE

models. Saikia and Karmakar [40] introduced a deterministic distributed approximation algorithm in the CONGEST model, achieving a $2(1 - 1/\ell)$ approximation factor, where $\ell$ is the number of leaf nodes in the optimal Steiner tree. Their method constructs a Shortest Path Forest (SPF) by adapting the Bellman-Ford algorithm in a distributed setting. This approach effectively partitions the network into shortest-path trees rooted at terminals, subsequently applying MST heuristics based on the well-known GKP algorithm proposed by Garay et al. [41]. The algorithm operates with a round complexity of $O(S + \sqrt{n} \log^* n)$ and message complexity of $O(mS + n^{3/2})$. Here, $S$ denotes the *shortest path diameter* of the input graph $G = (V, E)$, defined formally as

$$S = \max_{u,v \in V} \rho(u, v),$$

where $\rho(u, v)$ represents the number of edges on the shortest weighted path between vertices $u$ and $v$. Additionally, $m = |E|$ denotes the number of edges in the graph, and $\log^* n$ (pronounced "log-star $n$") is the iterated logarithm, defined as the number of times the logarithm function must be applied to $n$ before the result becomes less than or equal to 1.

In subsequent work, Saikia and Karmakar [42] improved upon these results, proposing two deterministic distributed algorithms in the CONGEST model, maintaining the same approximation factor of $2(1 - 1/\ell)$. Their first algorithm operates in $O(S + \sqrt{n} \log^* n)$ rounds with $O(mS + n^{3/2})$ messages, whereas the second reduces message complexity significantly to $\tilde{O}(Sm)$ at the cost of a slightly higher round complexity of $\tilde{O}(S + \sqrt{n})$. Moreover, for graphs with a small shortest-path diameter, specifically when $S = O(\log n)$, the algorithm achieves complexities of $\tilde{O}(\sqrt{n})$ rounds and $\tilde{O}(m)$ messages, effectively matching known singularly-optimal algorithms for MST problems.

The Steiner tree problem has also been explored within the CONGESTED CLIQUE model. Saikia and Karmakar [43] developed a deterministic distributed algorithm achieving a $2(1 - 1/\ell)$ approximation factor within $\tilde{O}(n^{1/3})$ rounds and a message complexity of $\tilde{O}(n^{7/3})$. Their algorithm utilizes iterated squaring for computing all-pairs shortest paths, constructs an SPF, categorizes edges into inter-tree and intra-tree, and employs a generalized MST construction, ensuring all leaves are terminals.

The round-message trade-off for distributed Steiner tree construction in the CONGEST model is explored by Saikia and Karmakar [44]. They presented a collection of deterministic algorithms offering a tunable balance between round and message complexity. For instance, one variant achieves round complexity $\tilde{O}(S + \sqrt{n})$ and message complexity $\tilde{O}(Sm)$, while another focuses on reducing rounds to $\tilde{O}(\sqrt{n})$ when $S = O(\log n)$, keeping message complexity

within $\tilde{O}(m)$. These trade-offs provide flexibility based on network diameter and density.

Further extending their contributions to the CONGESTED CLIQUE model, Saikia and Karmakar [45] proposed deterministic distributed approximation algorithms that retain the approximation factor $2(1-1/\ell)$, with round complexity $\tilde{O}(n^{1/3})$ and message complexity $\tilde{O}(n^{7/3})$. Their approach integrates all-pairs shortest path computation through iterated matrix squaring, SPF construction, and generalized MST techniques. Additionally, a more refined algorithm presented in [45] achieves a round complexity of $O(S + \log\log n)$ and message complexity $O(Sm + n^2)$ by employing deterministic distributed SPF computation and MST heuristics.

Saikia et al. [46] addressed the Prize-Collecting Steiner Tree (PCST) problem in the CONGEST model and proposed a deterministic distributed approximation algorithm based on the primal-dual framework. Unlike the classical Steiner Tree problem, the PCST formulation allows for terminal nodes to remain unconnected by incurring a penalty cost, thereby generalizing the connectivity requirement. The authors designed the D-PCST algorithm, which guarantees a $(2 - \frac{1}{n-1})$ approximation. The algorithm runs in $O(n^2)$ rounds and incurs $O(mn)$ message complexity, where $n$ and $m$ denote the number of vertices and edges, respectively. This result demonstrates that primal-dual methods can be effectively extended to distributed settings even under bandwidth constraints, and offers a foundation for further exploration of generalized Steiner-type problems in the CONGEST model.

Collectively, these contributions have significantly advanced distributed algorithmic frameworks for the Steiner tree problem, narrowing the complexity gaps between Steiner and Minimum Spanning Tree constructions in distributed network optimization.

A recent work by Lenzen and Patt-Shamir [11] presents the existence of a deterministic distributed algorithm in the congest model for computing the Steiner forest in static graphs to $(2 + \epsilon)$ approximation, an $O(\log n)$ approximation randomized algorithm and a lower bound on the running time for the Steiner forest problem in static graphs.

**Corollary 2.1.** *(Lenzen and Patt-Shamir [11]) For any constant $\epsilon > 0$, a deterministic distributed algorithm can compute a $(2+\epsilon)$-optimal solution for the DSF-IC problem in $\tilde{O}(s \times min\{k_0, WD\} + \sqrt{(min\{st, n\})} + k + D)$ rounds, where $k_0$ is the number of input components with at least two terminals.* □

Here, $s$ is the shortest path diameter, $WD$ is the weighted diameter, $D$ is the unweighted diameter, $k$ is the number of input components, $n$ is the total number of nodes, and $t$ is the number of terminal nodes, of the input graph

*G. DSF-IC(Distributed Steiner Forest-Input Component)* is defined as follows:

*Input:* At each node $v$, a label $\lambda(v) \in \Lambda \cup \{\perp\}$, where $\Lambda$ is the set of component identifiers. The set of terminals is $T = \{v \in V | \lambda(v) \neq \perp\}$. An input component $C_\lambda$ for $\lambda \neq \perp$ is the set of terminals with label $\lambda$.

*Output:* An edge set $F \subseteq E$ such that all terminals in each input component are connected by $F$.

*Goal:* Minimize $w(F) = \Sigma_{e \in F} w(e)$.

**Theorem 2.7.** *(Lenzen and Patt-Shamir [11]) A $O(\log n)$-optimal solution to the DSF-IC problem can be computed in $\tilde{O}(s+k)$ rounds with high probability.* □

Aharoni and Cohen [9] present a distributed algorithm for computing dynamic Steiner trees for multicast, addressing the problem of minimizing the number of nodes that keep the routing information in a minimum possible cost multicast tree in datagram networks. An update request can insert or delete a graph node from the set of destination nodes, and hence, the underlying graph does not change. The burden of routing is restricted to the source node and destination nodes only.

**Theorem 2.8.** *(Aharoni and Cohen [9]) An execution of Restricted-Dynamic Greedy Algorithm (R-DGA) provided in the paper on a sequence $\alpha$ of $i$ add/remove requests has performance ratio*

$$\frac{\textit{R-DGA}(\alpha)}{OPT} \leq \lceil \log(|Z_i|) \rceil$$

*where $Z_i$ is the set of destination nodes in the multicast tree after the add/remove request $i$.* □

The value of $|Z_i|$ may turn out to be large(close to $n$) in some cases.

The *performance ratio*, also known as the competitive ratio of an algorithm or solution, is a measure that compares the quality of the obtained solution to the quality of an optimal solution. Formally, for a minimization problem, if $A(I)$ denotes the cost of the solution produced by the algorithm on instance $I$ and $\mathrm{OPT}(I)$ denotes the cost of an optimal solution for the same instance, then the performance ratio $R$ is defined as:

$$R = \max_I \frac{A(I)}{\mathrm{OPT}(I)}$$

where the maximum is taken over all instances $I$. A performance ratio close to 1 indicates that the solution is near optimal, while larger ratios indicate poorer approximation. For maximization problems, the ratio is typically defined as $R = \min_I \frac{A(I)}{\mathrm{OPT}(I)}$. It refers to how well an algorithm performs on a specific instance or in practice, including empirical or expected behavior. It is used to denote the quality of an online algorithm.

## 2.5 Important Results in the Steiner Tree Problem

Table 2.1: Results in the static Steiner tree problem

| Year | Algorithm | Result | Author |
|---|---|---|---|
| 1981 | A fast algorithm for Steiner trees | $2(1 - \frac{1}{\lvert S\rvert})$ approximation in $O(\lvert S\rvert\lvert V\rvert^2)$ time | Kou et al. [14] |
| 1986 | A faster algorithm for the Steiner problem | $2(1 - \frac{1}{\lvert S\rvert})$ approximation in $O(E \log V)$ time | Wu et al. [16] |
| 1988 | Faster approximation algorithm for the Steiner problem in graphs | $O(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$ time | Mehlhorn [47] |
| 2000 | Steiner tree in quasi-bipartite graph | 1.279 approximation in $O(\lvert V \setminus S\rvert\lvert S\rvert^2)$ time | Zelikovsky [28] |
| 2008 | Inapproximability for Steiner tree | $\beta > \frac{96}{95}$ for all algorithms | Chlebik [2] |
| 2010 | An LP based algorithm for Steiner tree | 1.39 approximation in $O(n^5)$ time | Byrka [5] |
| 2012 | Steiner tree algorithm for bounded treewidth | Optimal solution in $O(B_{tw+2}^2 \times tw \times \lvert V\rvert)$ time | Chimani [29] |

$\beta$: Approximation factor
$tw$: Treewidth
$B_{tw+2}$ (Bell number): The number of partitions of a set with $tw + 2$ elements

Table 2.2: Results in the dynamic Steiner tree problem

| Algorithm | $\beta$ | Update time | Preprocessing time | model of update | Model of graph |
|---|---|---|---|---|---|
| Imase and Waxman Algorithm 1 (DST non-rearrangeable) | $P.R. \leq \lceil \log(n_i) \rceil$ | not mentioned | NA | terminal conversion | general graph |
| Imase and Waxman Algorithm 2 (DST rearrangeable) | $(4\delta)$, $\delta \geq 2$ | at most $\frac{1}{2}K_a(\sqrt{4K_a - 3} - 3) + K_r$ number of changes, $K_a$ and $K_r$ are the number of add and remove requests | NA | terminal conversion | general graph |
| Lacki et al. Algorithm 1 | $(2+\epsilon)$ | $\tilde{O}(\sqrt{n}\log D)$ | $\tilde{O}(n \log D)$ | terminal conversion | planar graph |
| Lacki et al. Algorithm 2 | $(4+\epsilon)$ | $\tilde{O}(\epsilon^{-1}\log^6 n)$ amortized update time | $O(\epsilon^{-1} n \log^2 n)$ | terminal conversion | planar graph |
| Lacki et al. Algorithm 3 | $(6+\epsilon)$ | $\tilde{O}(\sqrt{n}\log D)$ | $\tilde{O}(\sqrt{n}(m + n\log D))$ | terminal conversion | general graph |
| Raikwar and Karmakar Algorithm 1 | $(2+\epsilon)$ | $\tilde{O}(\lvert S\rvert^2\sqrt{n} + \lvert S\rvert D + n)$<br>Avg update time= $(\lvert S\rvert^2\sqrt{n} + \lvert S\rvert D + n)/k$ : $1 \leq k \leq \sqrt{n+m}$ for $k$ number of updates<br>Special case: $O(n^{0.5+\epsilon''}(\epsilon')^{-2}) + O(n)$ when $\lvert S\rvert, D = O(\log n)$ | $O(\epsilon^{-1} n \log^2 n)$ | edge insertion and deletion | planar graph |
| Raikwar and Karmakar Algorithm 2 | $(2-\epsilon)$ | $O(nD_s)$ | $O(\lvert E\rvert \log \lvert V\rvert)$ | edge insertion | general graph |
| Raikwar, Siglipalli, and Karmakar Algorithm 3 | $(2+\epsilon)$ | $\tilde{O}(\ell\sqrt{n})$ | $O(m + \epsilon^{-1} n \log^2 n)$ | edge deletions | planar graph |
| Raikwar, Patel, and Karmakar Algorithm 4 | $(\lvert S\rvert - 1)$ | $\tilde{O}((\delta p \log U)/\epsilon + \lvert S\rvert \cdot \sqrt{n} + n)$<br>$O(n)$ when $\lvert S\rvert = o(\sqrt{n})$ | $O(m)$ | edge insertion and deletion, vertex insertion and deletion, terminal conversion | planar graph |
| Raikwar and Karmakar Algorithm 5 | $(2+\epsilon)$ | $O(m^{1/2} + n^{2/3})$ | $O(kmn^{1/k})$, $k \geq 2$ | edge insertion and deletion, vertex insertion and deletion, terminal conversion | general graph |

$\beta$: Approximation factor; $P.R.$: Performance ratio (competitive ratio); $D$: Stretch of the matrix induced by the graph under consideration

Table 2.3: Results in the Steiner tree problem in distributed settings

| Year | Algorithm | Result | Author |
|---|---|---|---|
| 1998 | Dynamic restricted multicast tree | $P.R. \leq \lceil \log(z_i) \rceil$ | Aharoni [9] |
| 2013 | Distributed Dynamic Steiner tree | $P.R. = \log(\lvert S \rvert)$ in $O(\lvert S \rvert N)$ | Blin [12] |
| 2014 | Distributed static Steiner tree | $O(\log(n))$ approximation | Lenzen [11] |
| 2019 | Deterministic approximation in CONGEST | $2(1 - 1/\ell)$ approximation in $O(\lvert S \rvert + \sqrt{n} \log^* n)$ rounds and $O(m\lvert S \rvert + n^{3/2})$ messages | Saikia and Karmakar [40] |
| 2021 | Improved trade-offs in CONGEST | $2(1-1/\ell)$ approximation; $\tilde{O}(\lvert S \rvert m)$ messages in $\tilde{O}(\lvert S \rvert + \sqrt{n})$ rounds | Saikia and Karmakar [42] |
| 2020 | Round-message trade-off in CONGEST | Deterministic tunable trade-off: e.g. $\tilde{O}(\lvert S \rvert + \sqrt{n})$ rounds and $\tilde{O}(\lvert S \rvert m)$ messages | Saikia and Karmakar [44] |
| 2019 | Steiner tree in CONGESTED CLIQUE | $2(1 - 1/\ell)$ approximation in $\tilde{O}(n^{1/3})$ rounds and $\tilde{O}(n^{7/3})$ messages | Saikia and Karmakar [43] |
| 2020 | Improved CONGESTED CLIQUE algorithm | $2(1 - 1/\ell)$ approximation in $O(\lvert S \rvert + \log \log n)$ rounds and $O(\lvert S \rvert m + n^2)$ messages | Saikia and Karmakar [45] |
| 2021 | Prize-Collecting Steiner Tree in CONGEST | Deterministic $(2 - \frac{1}{n-1})$ approximation in $O(n^2)$ rounds and $O(mn)$ messages | Saikia et al. [46] |
| 1998 | Sublinear-time distributed algorithms for Steiner trees | MST-based approximation in CONGEST | Garay et al. [41] |

*P.R.*: Performance ratio (competitive ratio)
$z_i$: Set of destination nodes in the multicast tree after the add/remove request $i$

# 3

# A Fully Dynamic and an Incremental Algorithm

In this chapter, we propose two dynamic algorithms: A fully dynamic algorithm to maintain an approximate Steiner tree in planar graphs and an incremental algorithm to maintain an approximate Steiner tree in general graphs. We focus on edge-weighted connected graphs. The graph undergoes dynamic updates where edges with specific weights can be either inserted or deleted. The goal is to efficiently compute a Steiner tree of the updated graph, guaranteeing a solution quality (Steiner tree cost) within a good factor of the optimal Steiner tree.

This work explores the Steiner tree problem within two dynamic graph frameworks. Our contributions are as follows:

1. We introduce a fully dynamic algorithm capable of preserving a $(2+\epsilon)$-approximate Steiner tree in planar graphs while updates can insert and delete edges in a graph. This algorithm works in planar graphs because it leverages a dynamic distance oracle that operates in planar graphs, which restricts its applicability to general graphs. The proposed dynamic algorithm exhibits a worst-case update time complexity of $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$ for processing a series of $k$ number of updates where $k \in Z^+ : 1 \leq k \leq (\sqrt{n+m})$. Here $|S|$ represents the number of terminals, $D$ denotes the unweighted diameter of the initial graph, and $\epsilon$ is a small fraction $(0 \leq \epsilon \leq 1)$. This update time complexity arises from the need to update the distance oracle and the complete distance graph (metric closure) over terminals and maintain a Steiner tree structure efficiently in a dynamic planar graph. In a special case, the update time improves to $O(n(\epsilon')^{-2})$, where $\epsilon' = \epsilon/2$. This algorithm is fully dynamic

and capable of handling any sequence of edge insertion and edge deletion updates.

2. Additionally, we propose an incremental algorithm for maintaining an approximate Steiner tree in general graphs. This algorithm specifically addresses dynamic updates where only edge insertions are allowed. Each inserted edge has a positive real weight. The key objective of the algorithm is to efficiently maintain a $(2-\epsilon)$-approximate Steiner tree. The update time for processing an edge insertion is guaranteed to be $O(nD_s)$ where $D_s$ is known as the *shortest path diameter* of a graph and is defined in Section 3.2.1. The incremental algorithm maintains a partition of the input graph in the form of a shortest path forest, which aids in efficiently updating a Steiner tree.

Our proposed algorithms exhibit significant performance improvements over executing the static algorithm from scratch as given by Kou et al. [14] and Wu et al. [16], which require $O(|S|n^2)$ and $O(m \log n)$ time respectively. The dynamic Steiner tree problem under edge insertions and deletions is a relatively unexplored area, and this research attempts to fill this gap.

## 3.1 A Fully Dynamic Algorithm in Planar Graphs

### 3.1.1 Preliminaries

This work considers a dynamic graph denoted by $G = (V, E)$. The graph is planar, undirected, and edge-weighted. The set of edges, $E$, undergoes dynamic changes through insertions and deletions. Each edge $e$ has a positive real weight $w(e)$ within a range of 1 to $M$. This restricted weight assignment is necessary for the distance oracle that is used later. An update involves either the insertion or deletion of an edge $e$ from the graph $G$, with the inserted edge carrying a positive real weight $w(e)$. The weight of the shortest path between any two vertices $u$ and $v$ in a graph $\mathcal{G}$ is denoted by $dist(u, v, \mathcal{G})$. We assume that $G$ maintains both planarity and connectivity even after edges are inserted or deleted. Our Steiner tree problem differs from the one proposed by Imase and Waxman [7], where depending on the insert or delete request, the previous Steiner tree becomes either a subtree of the updated Steiner tree or vice versa. This constraint is not applicable to our problem. The notation $\tilde{O}$ hides polylogarithmic factors.

### Dynamic Distance Oracle

A distance oracle is a data structure, along with associated routines, designed to compute distances within a given graph efficiently. In the context of our research, a dynamic distance oracle refers to a data structure that, after some initial preprocessing, can manage a sequence of graph updates while minimizing both update time and data structure space. Specifically, a fully dynamic distance oracle can handle the insertion and deletion of edges, vertices, or both in a graph. This work leverages a dynamic approximate distance oracle. This oracle efficiently computes an approximate shortest path (and its weight) between any two vertices within the graph, even as the graph undergoes edge insertions and deletions.

A forbidden-set distance oracle for a graph $G$ is a preprocessed data structure that can answer distance queries of the form $(s, t, F)$, where $F$ is a set of faulty vertices (or edges), and $s$ and $t$ are vertices for which the distance needs to be computed in the graph $G \setminus F$ (the graph $G$ with the vertices or edges in $F$ removed).

Abraham et al. [48] proposed a forbidden-set distance labeling scheme. This scheme efficiently computes an approximate shortest path (and its weight) between two designated points $s$ and $t$ within the graph, even when specific vertices or edges represented by the set $F$ are excluded from the path calculation. This scheme, with some modifications (shown in [48]), can be adapted to construct a fully dynamic distance oracle with a stretch factor of $(1 + \epsilon')$, where $\epsilon'$ is a freely chosen stretch parameter such that $0 < \epsilon' < 1$. This dynamic distance oracle supports various operations, including edge insertion, edge deletion, vertex insertion, and vertex deletion, while maintaining worst case update and query times of $\tilde{O}(n^{1/2})$.

**Theorem 3.1.** *(Abraham et al. [48]) Given an $n$-vertex planar graph $G$ with edge weights in $[1, M]$ and a parameter $\epsilon' > 0$, a fully dynamic $(1 + \epsilon')$-approximate distance oracle of size $O(n \log n \cdot ((\epsilon')^{-1} + \log n))$ can be constructed in $O((\epsilon')^{-1} n \log^2 n)$ time. Each query operation and each update operation takes $O((\epsilon')^{-1} n^{1/2} \log^2 n \log(nM) \cdot ((\epsilon')^{-1} + \log n))$ time.*

The distance oracle is constructed based on a tree decomposition $\mathscr{T}$. Given a graph $G(V, E)$, a tree decomposition consists of a tree $T$ where each node (known as a "bag") of $T$ is associated with a subset of vertices from $V$ such that every vertex in $V$ appears in at least one bag. For every edge $(u, v)$ in $E$, there must be a bag in $T$ that contains both $u$ and $v$. If $v \in V$ appears in two different bags of $T$, it must also appear in all bags along the path between these two bags in the tree. A tree decomposition helps decompose a complex graph into simpler, more "tree-like" parts, where the relationship between

nodes is easier to analyze. The tree decomposition $\mathscr{T}$ divides the graph into multiple regions. These regions contain local information, reducing the overall complexity of query operations in the dynamic graph with updates. For every vertex $v$ in a region $R$, the oracle constructs an edge set $\tilde{E}(v, R)$. These sets are called vertex-region pairs. These vertex-region pairs allow quick retrieval of routing information within regions, enabling alternative path-finding when certain vertices or edges are no more available. This structure is essential for maintaining low computational complexity while ensuring robustness against updates. Additionally, a sketch graph $H$ is maintained by the distance oracle. A sketch graph is a data structure used to efficiently summarize and analyze large, dynamic graphs in a streaming environment. It is a compact data structure that approximates the properties of a graph without storing the entire graph in memory. The sketch graph $H$ plays a crucial role by approximating the original graph's structure in a smaller, more manageable form. It simplifies and optimizes distance queries by retaining key vertices and paths that are important for computation in a dynamic graph. Each vertex $u$ has a label $L(u)$, which helps identify safe edges forming a $(u \rightsquigarrow v)$ path, and a failed vertex label $FL(u)$, which stores the labels of other failure-free vertex-region pairs.

After updates that insert or delete edges or vertices in the graph, the distance oracle updates its data structure ($L$ and $FL$ labels and $\tilde{E}(v, R)$ vertex-region pairs). When queried for the distance between $u$ and $v$ in the updated graph, the oracle uses these labels and vertex-region pairs to find an alternative path between $u$ and $v$. This path is guaranteed to be within $(1+\epsilon')$ times the length of the shortest path between $u$ and $v$ in the updated graph.

### 3.1.2 Algorithm

The proposed algorithm utilizes a dynamic distance oracle to efficiently compute an approximate shortest path (and its weight) between a pair of terminals. This oracle's key feature is its ability to provide a $(1+\epsilon')$ approximation of distances between any pair of vertices quickly. Our algorithm leverages these approximate distances to construct a complete distance graph, where each edge represents an approximate distance between two terminals. Subsequently, it uses paths corresponding to these distances to construct a Steiner tree by applying an MST heuristic based $(2-\epsilon)$-approximate Steiner tree algorithm.

The main steps of our approach are outlined as follows:

- Construct and maintain a fully dynamic $(1+\epsilon')$-approximate distance oracle for the input planar graph $G$. This construction process takes

$O((\epsilon')^{-1} n \log^2 n)$ time and utilizes $O(n \log n \cdot ((\epsilon')^{-1} + \log n))$ space. The distance oracle has $\tilde{O}(\sqrt{n})$ worst case query and update times, as demonstrated in Theorem 3.1.

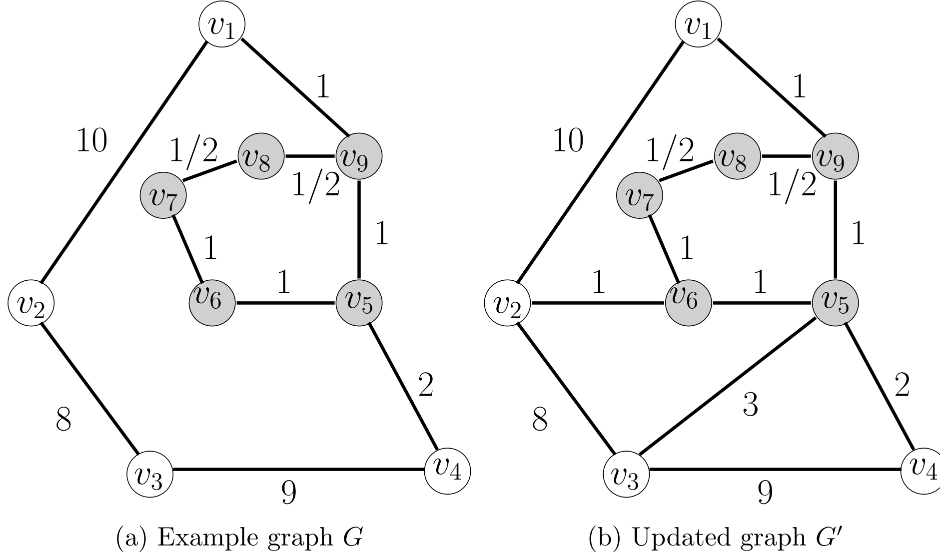
- Following a series of updates, the algorithm utilizes the distance oracle to generate a complete distance graph, denoted as $G_1$, on the terminal set $S$. In $G_1$, the distances between any pair of terminals $u$ and $v$ are approximate representations of their actual distances in the original graph $G$.
- Construct a minimum spanning tree, denoted as $T_1$, on the complete distance graph $G_1$.
- Substitute the edges of $T_1$ with the approximate shortest paths acquired from the dynamic distance oracle.
- Remove any cycles and non-terminal leaves from the resulting graph to obtain the updated $(2+\epsilon)$-approximate Steiner tree.

We begin with an input graph $G$ with vertex set $V$ and edge set $E$. A weight function $w : E \to \mathbb{R}^+$ assigns a positive real weight to each edge. Additionally, a subset $S$ of $V$ defines the terminal set. The update sequence $U$ modifies $G$ to $G'$ through edge insertions and deletions. (Details of update visualization can be found in FIGURE 3.1).

Terminals are represented by white vertices, while non-terminals are represented by grey vertices. The number of updates in $U$ is constrained to $O(\sqrt{n+m})$ to comply with the distance oracle requirements. If the update count exceeds this limit, $U$ can be divided into batches of this size for efficient processing. Each inserted edge has a positive real weight. The proposed algorithm operates in four stages. During updates, the distance oracle modifies its internal data structures, including labels and the sketch graph $H$. As given by Theorem 3.1, updates to the distance oracle can be performed in $\tilde{O}(\sqrt{n})$ time. This updated sketch graph $H$ guarantees the existence of $(1+\epsilon')$-approximate paths between any two vertices in the modified graph $G'$. These paths can then be used to estimate the approximate distance and corresponding path between any pair of terminals $u$ and $v$ within $G'$.

**Lemma 3.2.** *(Abraham et al. [48])* $dist(u, v, H) \leq (1+\epsilon')dist(u, v, G)$.

The distance oracle plays a crucial role in estimating distances between vertices. Querying the distance oracle for any two vertices $u$ and $v$, the oracle returns an approximate distance denoted by $dist(u, v, H)$. This approximation is based on the internal sketch graph $H$ maintained by the oracle. We utilize




(a) Example graph $G$

(b) Updated graph $G'$

Figure 3.1: Inserting edges $(v_2, v_6, 1)$ and $(v_3, v_5, 3)$ to the graph $G$.

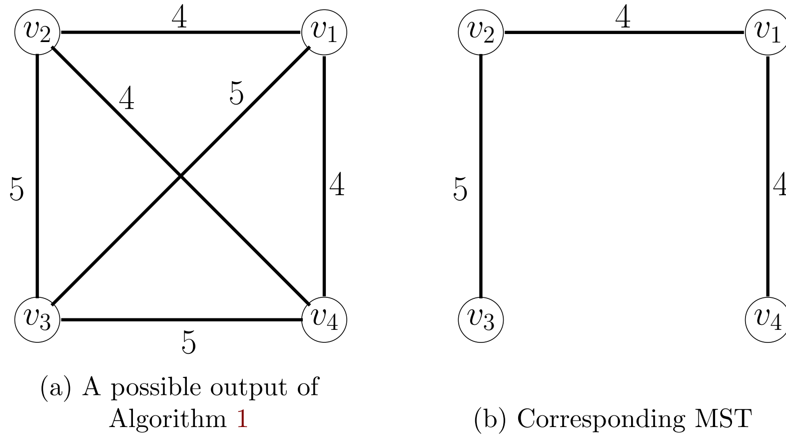
these approximate distances to construct a complete graph $G_1 = (S, E_1)$ on the terminal set $S$. The details of this construction can be found in Algorithm 1. Within $G_1$, each edge $(u, v)$ is assigned a weight of an approximate shortest path between $u$ and $v$ in the modified graph $G'$. This weight is directly obtained from the distance oracle's output. This approximation incorporates a factor of $(1+\epsilon')$, guaranteeing a close estimate of the weight of the shortest path.

**Algorithm 1** Construction of the complete graph (metric closure) on the terminals

$E_1 = \phi$
**for** all $u, v \in S : u \neq v$ **do**
  $E_1 = E_1 \cup \{(u, v)\}$
  query the distance oracle for $dist(u, v, H)$
  $w(u, v) \leftarrow dist(u, v, H)$
**end for**

Adding a small number of edges to the graph can significantly alter the shortest paths between most terminal pairs. This, in turn, affects the corresponding weights of the shortest path. The MST based heuristic necessitates recomputing all pairwise distances between the terminals.

The second step involves finding an MST $T_1 = (S, E_1')$ (FIGURE 3.2b)




(a) A possible output of Algorithm 1

(b) Corresponding MST

Figure 3.2: The graph $G_1$(left) and the tree $T_1$(right)

of the graph $G_1$. Kruskal's algorithm proves to be an efficient approach for this task.

**Lemma 3.3.** *(Kruskal [49]) Given a weighted undirected graph $G = (V, E)$, Kruskal's minimum spanning tree algorithm gives a minimum spanning tree of $G$ in $O(m \log n)$ time.*

The third step leverages the sketch graph $H$ to construct a new graph $T_2 = (V_2, E_2)$. This process, called the path replacement scheme, starts with an empty set of vertices ($V_2$) and edges ($E_2$). We then iterate over each edge $e(u, v)$ in the MST $T_1$ obtained in the previous step. For each edge $e(u, v)$, we add the corresponding $\Pi_H(u, v)$ within the sketch graph $H$ to $T_2$, obtained by querying the distance oracle. Equation 3.1 provides the details for incorporating these replacement paths ($\Pi_H(u, v)$) into $T_2$.

$$\forall\, e'(x, y) \in \Pi_H(u, v) : (u, v) \in E'_1 \tag{3.1}$$

$$E_2 \leftarrow E_2 \cup \{e'(x, y)\}$$

$$V_2 \leftarrow V_2 \cup \{x, y\}$$

While constructing $T_2$ from $T_1$, our approach avoids redundant vertices and edges. Set operations inherently prevent duplicates, making them unnecessary. However, directly incorporating the shortest paths $\Pi_H(u, v)$ might introduce cycles in the resulting graph $T_2$. To address this and ensure $T_2$ remains a tree structure, the path replacement scheme utilizes a $Union - Find$ data structure. Sets are named after one of the elements inside them. Initially,

each vertex $v$ in $V$ has a separate record, essentially its own set. This is similar to initializing each set in the $Make\text{-}Union\text{-}Find(V)$ routine. The $find(u)$ operation retrieves the set name containing vertex $u$, while $union(u, v)$ merges the sets with names $u$ and $v$. These set names essentially represent connected components within the graph $T_2$ being built. Algorithm 2 outlines the application of this path replacement scheme with the $Union - Find$ data structure.

**Algorithm 2** Path replacement scheme

```
for all x ∈ V do
    find(x) ← x
end for
for all (u, v) ∈ E'_1 do
    for all e'(x, y) ∈ Π_H(u, v) do
        if find(x) ≠ find(y) then
            apply union(find(x), find(y))
            E_2 ← E_2 ∪ {e'(x, y)}            ▷ Adding edge e'(x, y) to T_2
            V_2 ← V_2 ∪ {x, y}
        else
            Do not add edge e'(x, y) to T_2
        end if
    end for
end for
```

Our analysis demonstrates that the graph $T_2$ constructed by Algorithm 2 (Path Replacement Scheme) possesses two key properties:

- **Tree Structure:** $T_2$ is guaranteed to be a tree. This is achieved by the careful application of the $Union - Find$ data structure, which prevents cycle formation during path incorporation.
- **Vertex Set Coverage:** The vertex set of $T_2$, denoted by $V_2$, encompasses all terminals in $S$. Additionally, $V_2$ may include some non-terminals.

After constructing the initial approximate Steiner tree $T_2$, we can reduce its cost further by eliminating edges that have a non-terminal leaf as an endpoint. These non-terminal leaves are essentially dead ends, providing no connection between terminals. By recursively removing such edges and their corresponding non-terminal leaves, we obtain the final approximate Steiner tree, denoted by $T_3$. Formally, $T_3 = (V_2', E_2')$. It is shown in FIGURE 3.3. The Steiner vertices, which are non-terminals in $T_3$, are typically depicted in gray to distinguish them from terminals.

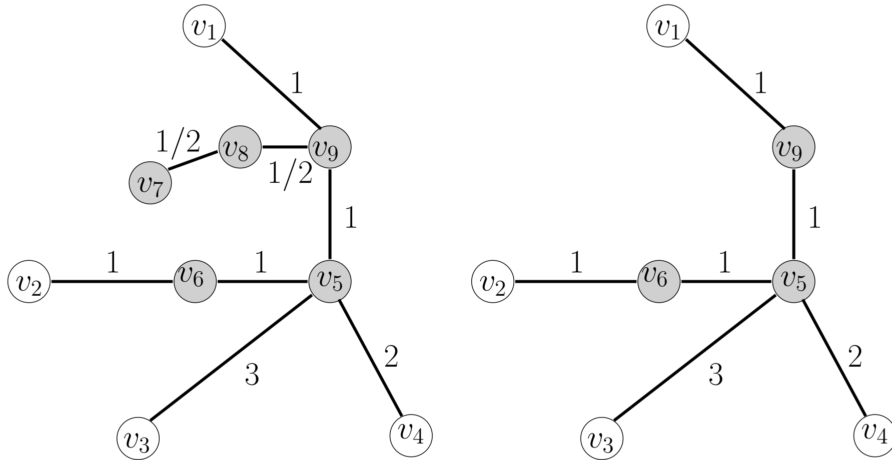



Figure 3.3: The tree $T_2$(left) and the final Steiner tree $T_3$(right) for $G'$

### 3.1.3 Analysis

#### Approximation Factor

**Lemma 3.4.** *The graph $T_2 = (V_2, E_2)$ constructed by Algorithm 2 is a tree spanning a vertex set $V_2$ such that $V_2$ contains all the terminals.*

*Proof.* The construction of $T_2$ leverages the $Union - Find$ data structure similar to Kruskal's algorithm. However, unlike Kruskal's algorithm, we do not require sorting the edges beforehand. Instead, we select edges based on the path replacement scheme outlined in Algorithm 2. An edge $e'(x, y)$ is only added to the set of edges $E_2$ if the $find(x)$ and $find(y)$ operations within the $Union-Find$ data structure return different sets. This essentially ensures that $e'(x, y)$ connects two separate parts (or components) of the graph being built. Additionally, $x$ and $y$ are incorporated into $V_2$ only if the corresponding edge $e'(x, y)$ is included in $E_2$. Following these criteria, it is guaranteed that there is always a unique path between any two vertices within $V_2$. Consequently, the constructed graph $T_2 = (V_2, E_2)$ forms a valid tree structure.

Further, the paths retrieved from the distance oracle (denoted by $\Pi_H(u, v)$) for all edges $(u, v)$ in $T_1(E_1)$ are incorporated into the graph $T_2$. Here's how this process ensures connectivity:

- **Existing Edges:** While adding a path, some edges within this path might already be present in $T_2$ (added due to the inclusion of some earlier paths). These edges contribute to the existing connections between vertices.

- **New Edges and Vertices:** If $u$ and $v$ belong to separate components in $T_2$, the missing edges from $\Pi_H(u, v)$ are added to $T_2$. This process also incorporates $u$ and $v$ in $V_2$.

It is guaranteed that throughout the construction of $T_2$, there is always at least one path between any two terminals in $G'$ and consequently in the sketch graph $H$, which captures approximate paths. This is because $H$ inherently preserves connectivity information. As a result, all terminals are guaranteed to be part of $V_2$ in $T_2$. Additionally, the paths (introduced during tree construction) connecting these terminals may include Steiner vertices. This analysis confirms that $S$ is a subset of $V_2$ ($S \subseteq V_2$) and that $T_2$ is a tree structure spanning the vertex set $V_2$. □

**Lemma 3.5.** *$T_3$ is a Steiner tree of the updated graph.*

*Proof.* Lemma 3.4 established that $T_2$ spans vertices in $V_2$ and incorporates all terminals ($S \subseteq V_2$). The construction of $T_3$ involves removing non-terminal leaves from $T_2$. These non-terminal leaves, by definition, are not part of any path connecting any two terminals ($u, v : u, v \in S$). Therefore, eliminating them does not affect the overall connectivity of the tree. This guarantees that $T_3$ also maintains a valid tree structure. The pruning process during $T_3$'s construction only targets non-terminal leaves. Consequently, no terminals are eliminated from $V_2$ during this transformation. As a result, all terminals remain present in the final vertex set $V_2'$ of $T_3$ ($S \subseteq V_2' \subseteq V_2$). Hence, $T_3 = (V_2', E_2')$ satisfies all the requirements of a Steiner tree within the updated graph $G'$. □

**Theorem 3.6.** *$T_3$ is a $(2 + \epsilon)$-approximate Steiner tree of the updated graph.*

*Proof.* The key to achieving an approximate Steiner tree lies in the graph $G_1 = (S, E_1)$ as $G_1$ incorporates $(1 + \epsilon')$ approximations of the weights of the shortest paths between all terminal pairs. The MST $T_1$ constructed on $G_1$ inherits these approximate distances. By Lemma 3.2, the distance between any two terminals $u$ and $v$ in $T_1$ (denoted by $dist(u, v, T_1)$) is at most $(1 + \epsilon')$ times the weight of the actual shortest path between them in $G'$:

$$dist(u, v, T_1) = dist(u, v, H)$$

$$\Rightarrow dist(u, v, T_1) \leq (1 + \epsilon')\ dist(u, v, G') \tag{3.2}$$

The notation $dist(\mathcal{T})$ denotes the total cost (weight) of a tree $\mathcal{T}$, which is given by the sum of the weights of edges in the tree. Applying (3.2) to all the edges of $T_1$ and adding the inequalities gives an upper bound on the weight

of $T_1$ ( $dist(T_1)$ ). Equation 3.3 expresses this upper bound equal to $(1+\epsilon')$ times the cost of some minimum spanning tree (denoted by $T$) on $S$ in $\overline{G'}[S]$.

$$dist(T_1) = \sum_{(u,v)\in E_1'} dist(u,v,T_1) \leq (1+\epsilon') \sum_{(u,v)\in E_1'} dist(u,v,G') = (1+\epsilon')\, dist(T) \tag{3.3}$$

Applying step 3 (replacing edges of $T$ with actual shortest paths in $G'$ using $Union-Find$ data structure) and step 4 (removing non-terminal leaves) of our algorithm on the tree $T$ produces a tree, say $T_3'$. Let the optimal Steiner tree cost in $G'$ be denoted by $\mathcal{OPT}$. Prior research by Kou et al. [14] established a crucial property. The authors showed that the cost of a Steiner tree $T_3'$ (constructed on the metric closure with exact distances between terminals) is upper bounded by a factor of $2(1-1/|S|)$ times the cost of the optimal Steiner tree $\mathcal{OPT}$ (Equation 3.4).

$$dist(T_3') \leq 2\left(1-\frac{1}{|S|}\right)\mathcal{OPT} \tag{3.4}$$

This inequality essentially states that the cost of $T_3'$ is at most two times the cost of the optimal Steiner tree, adjusted by a factor based on the number of terminals. This lays the groundwork for demonstrating that the final tree $T_3$ remains an approximate Steiner tree. An important observation is that removing non-terminal leaves from the initial tree $T_2$ (resulting in the final tree $T_3$) cannot increase the overall cost of the tree. This is intuitive because non-terminal leaves are essentially dead ends, not contributing to connections between terminals. Eliminating them does not create any new connections. Thus, the cost remains the same or lesser than the cost of $T_2$. Since removing non-terminal leaves does not inflate the cost, the inequalities derived earlier (Equation 3.3 and Equation 3.4) can be directly applied to the final tree $T_3$ as well. These inequalities provide an upper bound on the cost of $T_3$. We denote the optimal Steiner tree cost on the terminal set $S$ within the sketch graph $H$ by $OPT'$. The sketch graph $H$ guarantees $(1+\epsilon')$-approximate paths between terminals. This property allows us to establish a connection between the cost of $T_3$ and the optimal Steiner tree cost in the sketch graph. By considering an adjusted epsilon value ($\epsilon = 2\epsilon'$), we can derive new inequalities that relate the cost of $T_3$ to the optimal Steiner tree cost. This adjusted epsilon incorporates the inherent approximation introduced by the sketch graph.

$$dist(T_3) \leq 2\left(1-\frac{1}{|S|}\right)\mathcal{OPT}'$$

$$\Rightarrow dist(T_3) \leq 2\left(1-\frac{1}{|S|}\right)(1+\epsilon')\mathcal{OPT} \qquad \text{From (3.3)}$$

$$\Rightarrow dist(T_3) \leq 2(1+\epsilon')\mathcal{OPT} = (2+\epsilon)\mathcal{OPT}$$

□

As $0 < \epsilon' < 1$, we have $0 < \epsilon < 2$. Thus, the approximation factor of the final Steiner tree $T_3$ is $(2+\epsilon)$, where $0 < \epsilon < 2$.

### Update Time Complexity

This section addresses the impact of updates on the update time complexity of the algorithm. With a sequence of updates potentially containing $O(\sqrt{n+m})$ number of updates, the number of edges in $G'$ is still $O(m)$. In essence, the update process does not drastically alter the overall size of the graph, preserving the original complexity characteristics ($O(n)$ for vertices and $O(m)$ for edges).

**Lemma 3.7.** *The complete distance graph $G_1$ can be built in $\tilde{O}(|S|^2\sqrt{n})$ time.*

*Proof.* Algorithm 1 calculates the exact distance between each unique pair of terminals $(u, v)$ in the sketch graph $H$ (denoted by $dist(u, v, H)$), which turns out to be $(1+\epsilon')$-approximate as shown in Lemma 3.2. This calculation is performed only once for each distinct pair. All other steps within the algorithm require a constant amount of time, meaning their execution time does not significantly impact the overall complexity. The number of unique pairs of terminals is precisely the number of combinations of choosing two elements from the set $S$ (denoted by ${}^{|S|}C_2$). Therefore, the number of distance calculations scales quadratically with the number of terminals ($O(|S|^2)$). Each distance calculation ($dist(u, v, H)$) relies on a query to the distance oracle. In the worst case scenario, a query takes $\tilde{O}(\sqrt{n})$ time. By combining the factors discussed above, the total time required to construct $G_1$ can be expressed as:

$$O({}^{|S|}C_2) \cdot \tilde{O}(\sqrt{n}) = O(|S|^2) \cdot \tilde{O}(\sqrt{n}) = \tilde{O}(|S|^2\sqrt{n}) \tag{3.5}$$

□

In essence, the proof highlights that the construction time of $G_1$ is dominated by the number of terminal pairs and the time complexity of the distance oracle queries. For scenarios where the number of terminals is relatively small compared to the overall graph size, this construction remains efficient.

**Lemma 3.8.** *The update time of the final Steiner tree $T_3 = (V_2', E_2')$ is $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$.*

*Proof.* The tree $T_3$ is the solution constructed by the presented algorithm in different steps. We start by constructing $G_1$ and then compute the trees $T_1$, $T_2$ and $T_3$. Here is a breakdown of the time complexities for each step:

- **$G_1$ Construction (Lemma 3.7):** As shown earlier, constructing $G_1$ takes $\tilde{O}(|S|^2\sqrt{n})$ time.
- **Minimum Spanning Tree $T_1$:** We employ Kruskal's algorithm [49] to compute the minimum spanning tree $T_1$ on $G_1$. Since $G_1$ has $|S|$ number of vertices and $O(|S|^2)$ number of edges, this step takes $O(|S|^2 \log |S|)$ time.
- **$T_2$ Construction (Algorithm 2):** This involves building $T_2$ using the path replacement scheme. The key data structure employed here is a $Union - Find$ data structure implemented with pointers and path compression techniques (similar to Kruskal's algorithm). Kleinberg and Tardos [50] showed that with a pointers-based approach:
  - Initialization takes $O(n)$ time.
  - *union* operations (performed when merging components) take constant time each. There can be, at most, $n$ such operations, leading to a total time of $O(n)$.
  - $find$ operations (performed to check for existing connections) are more intricate. The combined time complexity of all the $find$ operations is $O(|S|D)$. It is elaborated below.
- **$T_3$ Construction:** Removing non-terminal leaves from $T_2$ to obtain $T_3$ can be done in $O(n)$ time as demonstrated by Kou et al. [14].

For constructing $T_2$ using the path replacement scheme, the process begins by initializing the $Union - Find$ data structure. This involves creating singleton sets, each containing a single vertex from $G'$. This initialization step can be accomplished efficiently in $O(n)$ time.

The *union* operation within the $Union - Find$ data structure combines two previously separate sets (or trees) into a single connected component. In the context of $T_2$ construction, *union* operations are only performed when two distinct components (representing vertices not yet connected in $T_2$) are encountered while adding edges from paths specified by $\Pi_H(u, v)$ (which uses information from the sketch graph $H$). Due to this specific usage, there can be, at most, $n$ number of *union* operations throughout the entire construction process. This is because each operation merges two separate components, and there are at most $n$ vertices (and thus $n$ potential components) in $G'$.

Each individual *union* operation is very fast, requiring constant time ($O(1)$) because it typically involves manipulating a single pointer within the data structure. Consequently, the total time consumed by all *union* operations becomes $O(n) \cdot O(1) = O(n)$. The $find$ operation within the $Union - Find$ data structure helps determine if two vertices belong to the same connected component. Here's how it applies to $T_2$ construction:

There are a maximum of $|S| - 1$ edges in $T_1$ (which serves as a starting point for $T_2$). For each edge $e(u, v) \in T_1$, a path $\Pi_H(u, v)$ is obtained by querying the distance oracle. Therefore, there can be at most $O(|S|)$ such paths. The $find$ operation is performed twice for each edge present within these paths as shown in Algorithm 2. This is because we need to check if the two endpoints of each edge belong to the same component before potentially adding the edge to $T_2$. Considering the path lengths, each path has a maximum length of $D$. Hence, the total number of $find$ operations is $O(|S|D)$. The time required for a series of $find$ operations can be expressed as $O(k\alpha(k))$, where $k$ represents the number of operations and $\alpha(k)$ is a very slow-growing function called the Inverse Ackermann function. While the technical details of $\alpha(k)$ might seem complex, the key takeaway is that this function has an incredibly slow growth rate. In practical scenarios, even for a large number of $find$ operations ($k$), the impact on the overall time complexity remains minimal. The function's value is guaranteed to be less than four for any value of $k$ encountered in real-world applications.

By combining the time complexities of each step, we arrive at the total update time required to construct $T_3$. This is expressed in Equation 3.6 as:

$$\begin{aligned}
&G_1 \textit{ update time } + T_1 \textit{ update time } + T_2 \textit{ update time } + T_3 \textit{ update time} \\
&= \tilde{O}(|S|^2\sqrt{n}) + O(|S|^2 \log |S|) + O(n) + T_2 \textit{ update time} \\
&= \tilde{O}(|S|^2\sqrt{n}) + O(|S|^2 \log |S|) + O(n) + \textit{ (Time to initialize Union-Find} \\
&\textit{data structure) } + O(n) \textit{ unions } + O(|S|D) \textit{ find operations} \\
&= \tilde{O}(|S|^2\sqrt{n}) + O(|S|^2 \log |S|) + O(n) + O(n) + O(|S|D \cdot \alpha(|S|D)) \\
&= \tilde{O}(|S|^2\sqrt{n} + |S|D + n) \qquad (3.6)
\end{aligned}$$

In essence, this section explains the role of *union* and $find$ operations in building $T_2$ and analyzes their time complexity. It highlights that *union* operations are relatively fast and limited in number. In contrast, $find$ operations are more frequent but have a small impact due to the slow-growing nature of the Inverse Ackermann function. □

**Theorem 3.9.** *There exists a fully dynamic algorithm to maintain a $(2+\epsilon)$-approximate Steiner tree in undirected connected weighted planar graphs under a sequence of $O(\sqrt{n+m})$ edge insertion and deletion updates in $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$ worst case update time, where $n$ is the number of vertices, $m$ is the number of edges, $S$ is the set of terminals and $D$ is the unweighted diameter of the graph after updates.*

*Proof.* Combining the results of Theorem 3.6 together with Lemma 3.7 and Lemma 3.8 implies Theorem 3.9. □

The presented algorithm significantly improves the update time compared to the static Steiner tree algorithm by Kou et al. [14], which achieves a $(2-\epsilon)$ approximation for static Steiner trees in $O(|S|n^2)$ time.

### Special Case

We previously established that the update time for constructing the initial graph $G_1$ is dominated by the distance query complexity, as shown in (3.5). Lemma 3.7 provided an initial estimate using the tilde notation $\tilde{O}$, ignoring logarithmic factors. Now, we consider the exact worst case time complexity of the distance query as shown in Theorem 3.1. This exact time complexity is $O((\epsilon')^{-1}n^{1/2}\log^2 n\log(nM)\cdot((\epsilon')^{-1}+\log n))$. By substituting this exact distance query time complexity into (3.5), we can determine the revised update time for constructing $G_1$. Using exact query time complexity, the update time shown in (3.6) becomes as follows:

$$
\begin{gathered}
O(|S|^2)O((\epsilon')^{-1}n^{1/2}\log^2 n\log(nM)\cdot((\epsilon')^{-1}+\log n)) \\
+O(|S|^2\log|S|)+O(n)+O(n)+O(|S|D\cdot\alpha(|S|D)) \\
=O(|S|^2)O((\epsilon')^{-1}n^{1/2}\log^2 n\log(nM)\cdot((\epsilon')^{-1}+\log n)) \\
+O(|S|^2\log|S|)+O(n)+O(|S|D)
\end{gathered}
$$

When size of $S$ and $D$ are logarithmic in $n$, the update time of the presented algorithm reduces to $O(\log^2 n)O((\epsilon')^{-1}n^{1/2}\log^2 n\log(nM)\cdot((\epsilon')^{-1}+\log n)) + O(\log^3 n)+O(n)+O(\log^2 n)$.

Using $\forall\, k, \epsilon''>0, log^k(n)=o(n^{\epsilon''})$:

$$
\begin{gathered}
O(\log^2 n)O((\epsilon')^{-1}n^{1/2}\log^2 n\log(nM)\cdot((\epsilon')^{-1}+\log n))+O(\log^3 n) \\
+O(n)+O(\log^2 n)=O(n^{0.5+\epsilon''}(\epsilon')^{-2})+O(n)
\end{gathered}
$$

Hence, the proposed algorithm has a linear update time in terms of $n$ when the size of $S$ and $D$ are $O(\log n)$.

## 3.2 An Incremental Algorithm for $(2-\epsilon)$-Approximate Steiner Tree

### 3.2.1 Preliminaries

The graph under consideration is a connected general graph $G = (V, E)$. Each edge has a positive real weight. Each update to the graph involves insertion of an edge with some positive real weight. The weight of an edge $e$ is denoted by $w(e)$. To facilitate a clear understanding of the algorithm's behavior, we adopt the following standard notation: $\Pi(u, v)$ denotes the shortest path between two vertices $u$ and $v$, $dist(u, v)$ represents the weight of the shortest path between vertices $u$ and $v$ in a Steiner tree. The weight of a path $P$ (sum of weights of the edges in the path) is denoted by $w(P)$. The symbol $\circ$ denotes the concatenation of two paths. We define $D_s$ as the *shortest path diameter* of the graph. It represents the maximum number of hops between any two vertices based on the shortest path: $D_s = \max_{u,v \in V} \rho(u, v)$. Here, $\rho(u, v)$ represents the number of edges in a shortest path between any two vertices $u$ and $v$. The graph maintains connectivity since an update does not remove an edge or a vertex.

We focus only on edge insertion updates. The Steiner tree in this scenario contrasts with the Steiner tree proposed by Imase and Waxman [7] as discussed in Section 3.1.

In a connected, undirected graph, a shortest path tree rooted at a vertex $u$ is a special kind of spanning tree. A key property of this tree is that the distance along the path within the tree between the root $u$ and any other vertex $v$ is guaranteed to be identical to the weight of the shortest path between $u$ and $v$ in the original graph. A shortest path forest of a graph $G = (V, E)$ is a collection of shortest path trees, each of which is rooted at a special vertex (say $u_i$). Each tree within the forest ensures the property of a shortest path tree. For clarity, the root of a vertex $u$ in a specific tree within the forest is denoted as $root(u)$. The proposed algorithm maintains a shortest path forest of the given graph under edge insertions.

#### Key Idea

The construction of a shortest path forest is similar to Dijkstra's single-source shortest path algorithm. However there is a key difference. Instead of focusing on a single source vertex, we consider all designated terminals(roots) simultaneously during the forest building process. If there is an edge $(u, v)$ that links two shortest path trees and establishes the shortest path between $root(u)$ and

$root(v)$, we include the path $\Pi(root(u), root(v)) = \Pi(root(u), u) + (u, v) + \Pi(v, root(v))$ in the Steiner tree. This approach efficiently handles updates that modify the shortest paths between terminals. When a new edge is inserted, the data structure captures the change and updates the shortest paths accordingly. The maintained Steiner tree is also adjusted to reflect these modifications by considering the revised shortest paths. The key challenge lies in maintaining a shortest path forest efficiently. The proposed algorithm maintains a shortest path forest in $O(n)$ time under a single edge insertion.

### 3.2.2 A $(2 - \epsilon)$-Approximate Steiner Tree

Wu et al. [16] propose an algorithm to find an approximate Steiner tree with an approximation factor of $2(1 - 1/\ell)$ as discussed in the related work chapter. Here, $\ell$ is the minimum number of leaves in any optimal Steiner tree. Their method leverages a concept similar to Dijkstra's shortest path algorithm but builds shortest path trees simultaneously from all terminals. The algorithm utilizes a minimum priority queue. Each element in the queue is a tuple $(r, d, s)$, where $r$ represents a vertex and $d$ signifies the weight of the current shortest path from $r$ to its closest terminal $s$. To track the path itself, the tuples are later augmented to $(r, d, s, p_1, p_2)$. Here, $p_1$ denotes the predecessor of $r$ in the shortest path, and $p_2$ is set to $NULL$ when $r$ does not directly connect two terminals. When $r$ connects two terminals, $p_2$ is set to $r$. The algorithm efficiently maintains predecessor information for each vertex. When a tuple extracted from the priority queue is such that $r$ is a terminal, the corresponding path information stored in the predecessors $p_1$ and $p_2$ is crucial for constructing the connection between $r$ and $s$ in the Steiner tree. This is because the extracted tuple represents the shortest path between these very terminals. Subsequently, this path is incorporated into the ongoing computation of the Steiner tree (FIGURE 3.4).

### 3.2.3 Shortest Path Forest

A shortest path forest refers to a collection of disjoint shortest path trees. Each tree in the forest has a designated root vertex, and the trees collectively encompass all other vertices in the graph. Dial's algorithm [15] efficiently computes such a shortest path forest for a general graph $G = (V, E)$. It achieves this in $O(m)$ time. The algorithm operates by simultaneously expanding these shortest path trees outward from their root vertices, and it maintains a topological ordering of the vertices throughout the process. The algorithm presents a way to perform minimum priority queue operations in constant time with the help of some data structures. We use the shortest path forest given by

Dial to maintain a partition of vertices into shortest path trees as it computes the shortest path forest in $O(m)$ time and facilitates the selection of a vertex in the forest to be explored in constant time with the help of priority queue operations. To the best of our knowledge, this is the best time complexity for computing a shortest path forest and performing minimum priority queue operations.

### 3.2.4 The Incremental Algorithm

We consider a pre-computed $2(1-1/\ell)$-approximate Steiner tree on the input graph denoted by $T_s$, obtained using the algorithm of Wu et al. [16]. Our goal is to process edge insertion updates to the input graph efficiently. We store and use some information about $T_s$ to update $T_s$ when some edge insertions are done to the graph. Wu et al.'s algorithm [16] utilizes a priority queue to manage the $(r, d, s, p1, p2)$ tuples containing information about vertices. The algorithm extracts tuples from the queue to construct $T_s$. When processing edges leaving the vertex $r$ identified in an extracted tuple $(r, d, s, p1, p2)$, new tuples are generated and inserted back into the queue. Importantly, the algorithm stores and updates the vertex $r$ associated with each tuple upon its insertion into the queue. If $root(r)$ and $s$ belong to different shortest path trees and $r \in V \setminus S$, the tuple corresponds to a path connecting the two terminals $root(r)$ and $s$. In such a case $p_2$ is set to $r$ ($p_2 = r$) so that the path connecting $root(r)$ and $s$ can be obtained by iterating predecessors from $p_1$ and $p_2$ as shown in case 3.2 of Wu et al. [16]. It is shown in FIGURE 3.4. When $r \in S$, the tuple $(r, d, s, p_1, p_2)$ indicates a shortest path between the terminals $r$ and $s$. This path comprises the paths $\Pi(p_1, s)$ and $\Pi(p_2, r)$, along with the edge $(p_1, p_2)$. It can be observed in FIGURE 3.4 with $r = s_1$ and $s = s_2$. The dashed arcs denote paths that do not belong to the forest. The two directed edges denote the shortest path between terminals (roots) $s_1$ and $s_2$. The algorithm of Wu et al. [16] terminates queue extraction operations when all the terminals are connected. For our algorithm, we need the pairwise shortest paths among all the terminals rather than the $(r, d, s, p_1, p_2)$ tuples. Therefore, we keep extracting tuples from the queue until all the pairwise shortest paths among terminals are known. For the tuples where $r \in S$, the $(p_1, p_2)$ pair corresponding to each $(r, s)$ pair are stored to obtain $\Pi(r, s)$ using the computed and stored predecessors provided by the algorithm. We also keep track of whether or not the shortest path between any two terminals is used in $T_s$ when the tuples are extracted from the queue.

Upon termination, the algorithm successfully identifies all the shortest paths among the designated terminals. We maintain a set of lists, denoted by

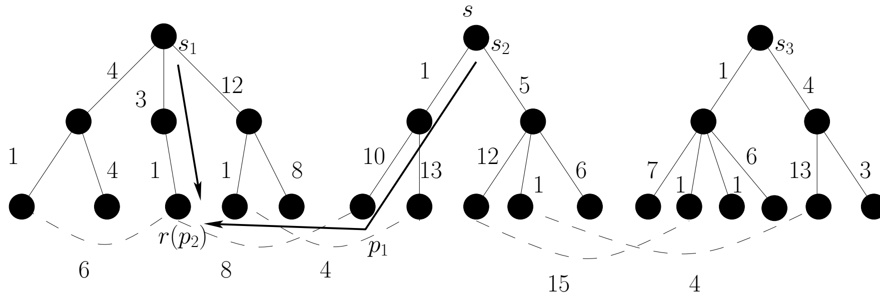



Figure 3.4: The shortest path between the terminals $s_1$ and $s_2$ through $r$ as $p_2$ and $s_2$ as $s$

$L_1, L_2, \ldots, L_{|S|}$. Each list $L_i$ corresponds to a specific terminal $t_i$ within the set $S$, where $i$ ranges from 1 to $|S|$. Importantly, list $L_i$ stores the distances from all other terminals in the graph to the terminal $t_i$ it represents. This allows for efficient retrieval of shortest path information.

Our algorithm benefits from a precomputed shortest path forest $F$ for the underlying graph. Notably, we employ Dial's algorithm [15] for precomputing $F$, which itself builds upon the foundation laid by Moore's work [51]. Dial's algorithm utilizes specialized data structures to perform minimum priority queue operations in $O(1)$ time and to perform shortest path computations efficiently. The maximum edge weight and the maximum path weight are denoted by $maxd$ and $maxdist$, respectively. As the proposed algorithm works for real edge weights, we redefine the mod function of Dial [15] as required by our algorithm. For a given real number $w$, the modified mod function is: $mod(w, maxd) = w - maxd \times \lfloor w/maxd \rfloor$. We desire to grow $F$ from terminals. Hence, each terminal is initialized as a root vertex by setting its distance to zero. The rest of the vertices are initialized as non-root vertices by setting their distance to $maxdist$ in $F$.

**Data Structures Used:** The algorithm utilizes several lists ($NEXT$, $LAST$, $INDEX$) and arrays ($HEAD$, $TAIL$, $D$, $I$, $J$, $DIST$) to manage the forest's state during computation. $NEXT$ is a circular list that provides the forward topological order of nodes. The forward topological order is a sequence of vertices in which every vertex is listed after every other vertex that lies on the path from its root to it. $NEXT$ helps determine the next vertex to be examined while building $F$. $LAST$ maintains the backward topological order. The backward topological order has the vertices arranged in decreasing distance from their nearest root node. $LAST$ helps in updating the existing paths when a new shortest path is found. $HEAD$ and $TAIL$ track the first and the last nodes in a sublist based on distance properties. For each

vertex $u$, $J$ holds $v$ for all edges $(u,v)$ continuously in increasing order of edge lengths, and $D$ contains corresponding edge lengths. $INDEX(u)$ points to the first entry in $J$ corresponding to $u$. The roots are added to $NEXT$ and $LAST$. It chains the terminals into the list of vertices to be scanned. The algorithm iteratively examines outgoing edges from root vertices (terminals) and adds the edges to $F$. This process continues by selecting the next vertex with the minimal distance for exploration. As the algorithm progresses, it checks if examining an edge $(u,v)$ would lead to a shorter path for $v$. If so, the root of $v$ is updated to match the root of $u$. The root of a terminal is set to itself. The arrays $DIST$ and $I$ store the weights $(DIST(u))$ of shortest paths from root nodes and predecessor information, respectively, for efficient retrieval of shortest paths. The distance $DIST(u)$ is the distance of $u$ from the nearest terminal in $F$. $I(v)$ contains $u$ for an edge $(u,v)$ in $F$, allowing path reconstruction. The $HEAD$ and $TAIL$ function as a minimum priority queue, enabling the selection of the next vertex for exploration in constant time. Upon completion, the algorithm determines the shortest distance from each non-terminal to its closest terminal, along with the identification of that closest terminal.

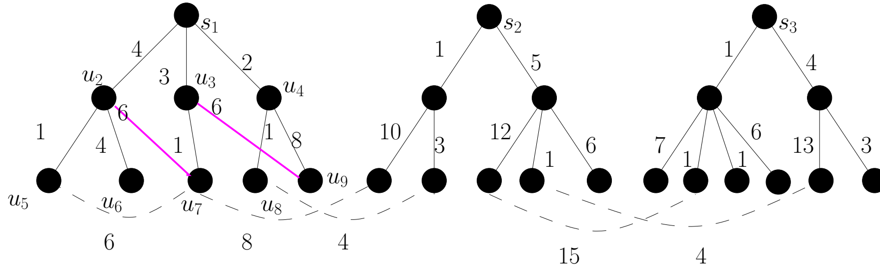



Figure 3.5: Both endpoints of the inserted edges lie inside a tree. Insertion of the edge $(u_2, u_7)$ does not change the shortest path to $u_2$ or $u_7$. Insertion of the edge $(u_3, u_9)$ changes the shortest path to $u_9$.

Upon inserting an edge $(u,v)$ with weight $w$ into the graph, it may decrease the path length from the closest terminal to either $u$ or $v$, but not both. If $u$ is closer to the root than $v$ from its corresponding root, then $u$ can update the path to $v$ using the inserted edge $(u,v)$ but not the other way. Similarly, If $v$ is closer to the root than $u$ from its corresponding root, then $v$ can update the path of $u$ using the inserted edge $(u,v)$ but not the other way. The formal proof is given in Lemma 3.10. The dashed edges in FIGURE 3.5 and FIGURE 3.6 denote the edges inserted as updates. It can be observed in FIGURE 3.5 that the inserted edge may not change the shortest paths at all.

The proposed algorithm is shown in Algorithm 3. The algorithm effi-

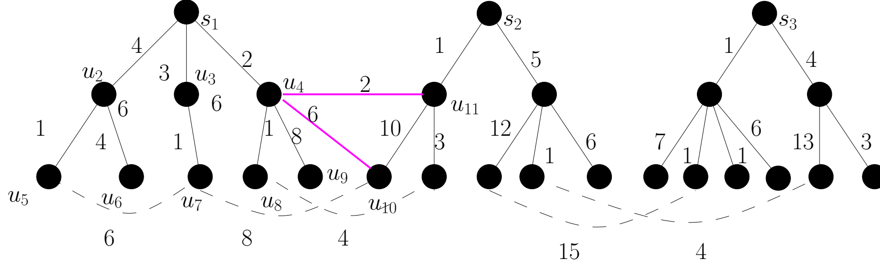


Figure 3.6: Endpoints of the inserted edges lie in different trees. Insertion of the edge $(u_4, u_{10})$ does not change the shortest path between $s_1$ and $s_2$. Insertion of the edge $(u_4, u_{11})$ changes the shortest path between $s_1$ and $s_2$.

ciently incorporates the inserted edge into the maintained data structures. This is achieved by adding a new entry to both arrays $J$ and $D$ (Algorithm 3, line 2). This entry stores details relevant to the new edge, including its weight. Following the insertion of a new edge $(u, v)$, the algorithm strategically selects the next vertex to explore to update $F$ and $T_s$ (lines 3-7 and line 31 of Algorithm 3). Dial [15] shows that the lists $HEAD$, $TAIL$, $NEXT$, and $LAST$ help in the efficient selection of the next vertex to explore. This decision ensures efficient updates to $F$ while incorporating the new edge information. If the newly inserted edge $(u, v)$ alters the shortest path to $v$, then $u$ becomes the chosen vertex for further exploration (denoted as *selected_vertex*). This is depicted in FIGURE 3.5. Otherwise, we choose $v$ as the *selected_vertex* because even if the shortest paths to $u$ and $v$ remain unchanged due to the edge insertion, the shortest path between the terminals (the roots of $u$ and $v$) might be affected. This scenario is illustrated in FIGURE 3.6. To account for this possibility, $v$ is chosen as the *selected_vertex* for further exploration. This ensures that potential changes to the shortest paths among terminals are identified and addressed. By following these selection criteria, the algorithm efficiently determines the vertex that holds the most potential for influencing the shortest path structure after the edge insertion. It then focuses its exploration efforts on the descendants (connected vertices) of the *selected_vertex* to update and maintain $F$ accurately and update $T_s$ accordingly.

While updating $F$, if the algorithm encounters an edge $(u', v')$ such that the root of $u'$ differs from the root of $v'$, as illustrated in FIGURE 3.6, we get a new path $root(u') \rightsquigarrow u' \circ (u', v') \circ v' \rightsquigarrow root(v')$ connecting the terminals $root(u')$ and $root(v')$. We examine if the weight $DIST(u') + w(u', v') + DIST(v')$ of this new path is less than the weight $dist(root(u'), root(v'))$ of the previous shortest path between the terminals $root(u')$ and $root(v')$ (from the list $L_{root(u')}$). If the weight of the new path is not smaller than the cur-

**Algorithm 3** An Incremental algorithm for $(2-\epsilon)$-approximate Steiner tree

1: An update inserts an edge $e = (u, v)$ with weight $w(e)$ in the graph
2: Incorporate $(u, v)$ by adding to data structure $J$ and data structure $D$
3: **if** $DIST(v) > DIST(u) + w(u, v)$ **then** ▷ Choose the vertex required to be explored
4: $i \leftarrow u$
5: **else**
6: $i \leftarrow v$
7: **end if**
8: $r$ : ▷ Examine outgoing edges of the *selected_vertex*
9: **for** each edge $(i, j)$ **do**
10: **if** $DIST[j] > DIST[i] + w(i, j)$ **then**
11: ▷ Found a new and shorter path to $j$ via $i$
12: Add $(i, j)$ to $F$
13: **end if**
14: **if** $root(i) \neq root(j)$ **then** ▷ The edge $(i, j)$ being examined creates a new path between the terminals $root(i)$ and $root(j)$.
15: **if** $dist(root(i), root(j)) > DIST(i) + w(i, j) + DIST(j)$ **then**
16: ▷ New shortest path between terminals is found
17: $dist(root(i), root(j)) \leftarrow DIST(i) + w(i, j) + DIST(j)$
18: ▷ Update $T_s$, $L_i$ and $L_j$
19: **if** $root(i) \rightsquigarrow root(j)$ shortest path $\in T_s$ **then**
20: Select the pair $(p_1, p_2)$ corresponding to the pair $(root(i), root(j))$
21: Remove the $p_1 \rightsquigarrow root(i) \circ (p_1, p_2) \circ p_2 \rightsquigarrow root(j)$ path from $T_s$
22: Add the $i \rightsquigarrow root(i) \circ (i, j) \circ j \rightsquigarrow root(j)$ path present in $F$ to $T_s$
23: $p_1 \leftarrow j$, $p_2 \leftarrow I[j]$ (previous predecessor of $j$), $root[j] \leftarrow root[i]$
24: **else**
25: $(P', w_{max}) \leftarrow$ FIND_MAX_WEIGHT_PATH $(T_s, root(i), root(j))$
26: ▷ Path $P'$ is defined in *Case* 2 of Section 3.2.4
27: **if** $w_{max} > DIST(i) + w(i, j) + DIST(j)$ **then** ▷ $w_{max}$ is weight of the path $P'$
28: Delete $P'$
29: Copy the path $i \rightsquigarrow root(i) \circ (i, j) \circ j \rightsquigarrow root(j)$ from $F$ to $T_s$

30: $p_1 \leftarrow j$, $p_2 \leftarrow I[j]$ (previous predecessor of $j$), $root[j] \leftarrow root[i]$
31: **end if**
32: **end if**
33: **end if**
34: **end if**
35: Update the data structure to allow for faster selection of the next vertex for processing.
36: **end for**
37: **if** there is a vertex that needs to be explored **then**
38: Select the next vertex in $F$ for which its outgoing edges require exploration, **go to** $r$
39: **end if**

rent shortest path between $root(u')$ and $root(v')$, then we show that $T_s$ is not required to be modified to maintain the approximation factor. $T_s$ needs to be updated if the newly created path has a weight smaller than the current shortest path. There are two scenarios when the new shortest path has a smaller weight than the current shortest path:

- **Case 1:** $root(u') \rightsquigarrow root(v')$ **shortest path** $\in T_s$**:** The algorithm updates the shortest path between $root(u')$ and $root(v')$ within $T_s$ by leveraging the information stored in the $(p_1, p_2)$ pair associated with $(root(u'), root(v'))$ pair in $T_s$. This is done by deleting $p1 \rightsquigarrow root(u') \circ (p_1, p_2) \circ p_2 \rightsquigarrow root(v')$ path from $T_s$ and adding the updated $u' \rightsquigarrow root(u') \circ (u', v') \circ v' \rightsquigarrow root(v')$ path from the updated shortest path forest in $T_s$. Update $(p_1, p_2) = (u', v')$ for $(root(u'), root(v'))$.

- **Case 2 :** $root(u') \rightsquigarrow root(v')$ **shortest path** $\notin T_s$**:** In this scenario, the addition of this new path into $T_s$ creates a cycle. Hence, before the inclusion of this new path in $T_s$, for some $z : v_z \in S$, the algorithm finds a sub-path $P' = u_1, u_2, \ldots, u_z$ in the $u' \rightsquigarrow v'$ path in $T_s$ such that the endpoints of the path are terminals ($u_1, u_z \in S$), rest of the vertices in the path are non-terminals, and the weight of this sub-path is maximum among all such sub-paths of the $u' \rightsquigarrow v'$ path in the tree. This can be computed by a DFS procedure on the tree as shown in Algorithm 4 and Algorithm 5. If $DIST(u') + w(u', v') + DIST(v') < w(P')$, exclude $P'$ from $T_s$ and include the updated shortest path found by the algorithm. Otherwise, $T_s$ needs not to be modified to maintain the $(2 - \epsilon)$ approximation.

Presence of such a $(u', v')$ edge may also result in changing the root of either $u'$ or $v'$ as the edge $(u', v')$ may shorten path of $u'$ (or $v'$) from $root(v')$ (or $root(u')$) via $v'$ (or $u'$). The algorithm updates the trees of $F$ in such cases. If the weight $DIST(u') + w(u', v') + DIST(v')$ of the new path found by the algorithm exceeds the weight $dist(root(u'), root(v'))$ of the existing shortest path in $T_s$, there is no requirement to take any action to maintain the approximation factor of $T_s$.

**Algorithm 4** FIND_MAX_WEIGHT_PATH(Steiner tree $T$, Terminal $u$, Terminal $v$)

1: $PARENT$ - To store the parent of a vertex in $T$ rooted at $u$
2: **for** all vertices $i$ **do**
3:     **if** $i = u$ **then**
4:         $PARENT[u] \leftarrow u$ ▷ Parent of $u$ is $u$ itself, as $DFS$ starts at $u$
5:     **else**
6:         $PARENT[i] \leftarrow NIL$ ▷ Initial parent of other vertices is $NIL$
7:     **end if**
8: **end for**
9: $t_1, t_2$ - To store terminals such that $\Pi(t_1, t_2)$ is a sub-path of $u \rightsquigarrow v$ path in $T$ and $w(\Pi(t_1, t_2))$ is maximum
10: $w_{max}$ - To store $w(\Pi(t_1, t_2))$
11: $(t_1, t_2, w_{max}) \leftarrow DFS\ (T, u, u, 0, 0, NIL, NIL, PARENT, v)$
12: ▷ From Algorithm 5
13: $PATH$ - To store the maximum weighted sub-path of $u \rightsquigarrow v$ path
14: **while** $t_2 \neq t_1$ **do**
15:     $PATH \Leftarrow t_2$ ▷ $\Leftarrow$ appends vertex $t_2$ to $PATH$ array
16:     $t_2 \leftarrow PARENT[t_2]$
17: **end while**
18: $PATH \Leftarrow t_1$
19: return $PATH, w_{max}$

$FIND_MAX_WEIGHT_PATH$ procedure takes a Steiner tree $T$ and terminals $u$ and $v$ as inputs. The procedure finds the maximum weighted sub-path of $u \rightsquigarrow v$ path in $T$, which has terminals as endpoints and does not include any other terminal. The procedure constructs the described path as follows. It calls the $DFS$ procedure (Algorithm 5), which calculates the weights of the sub-paths between subsequent terminals in the $u \rightsquigarrow v$ path present in $T$. The $DFS$ procedure also maintains parents of vertices in the DFS traversal initiated at $u$ as root. The $DFS$ procedure returns the endpoints of the maximum weighted sub-path and the weight of that path. The

$FIND_MAX_WEIGHT_PATH$ procedure stores the endpoints returned by the $DFS$ procedure in $t_1$ and $t_2$ and the maximum weight in $w_{max}$. It computes the actual sub-path using $PARENT$ array, $t_1$, and $t_2$ by backtracking parents. The actual sub-path is stored in $PATH$ array. The procedure returns the sub-path $PATH$ and the maximum weight $w_{max}$.

**Algorithm 5** DFS(Steiner tree $T$, vertex $x$, terminal $p$, weight $c$, max_weight $w_{max}$, $w_{max}$_endpoint $a$, $w_{max}$_endpoint $b$, PARENT, terminal_endpoint $v$)

```
1:  for all neighbors y of x in T do
2:      if PARENT[y] = NIL then                ▷ Explore y if it is not visited
3:          PARENT[y] ← x
4:          if y ∈ S then
5:              if w_max < c + w(x, y) then
6:                  w_max ← c + w(x, y)
7:                  a ← p, b ← y
8:              end if
9:              if y = v then
10:                 return a, b, w_max
11:             end if
12:             return DFS (T, y, y, 0, w_max, a, b, PARENT, v)
13:         else
14:             return DFS (T, y, p, c + w(x, y), w_max, a, b, PARENT, v)
15:         end if
16:     end if
17: end for
```

The $DFS$ procedure (Algorithm 5) takes as input a Steiner tree $T$, a vertex $x$, a terminal $p$, the current weight $c$ of the sub-path being explored, the weight $w_{max}$ of the maximum weight sub-path found so far, endpoints $a$ and $b$ of the maximum weight sub-path, the $PARENT$ array, and the terminal endpoint $v$ such that sub-paths between $x$ and $v$ are to be explored. The initial value of $x$ is $u$ for exploring the $u \rightsquigarrow v$ path in $T$. The procedure computes weights of terminal-to-terminal sub-paths in the $u \rightsquigarrow v$ path and maintains the weight of the maximum weight sub-path with its terminal endpoints. When the $DFS$ procedure reaches the terminal $v$, it returns the endpoints of the maximum weight sub-path and the weight of the maximum weight sub-path. The $DFS$ procedure is recursively called for the unexplored vertex $x$, starting from the terminal $u$. The terminal $p$ stores one of the terminal endpoints (the last terminal ancestor in the $DFS$ traversal) of the current sub-path being explored. The initial value of $p$ is $u$. Current weight $c$ stores the sub-path

weight from the terminal $p$ to the vertex $x$. The weight $w_{max}$ stores the weight of the terminal-to-terminal maximum sub-path in $u \rightsquigarrow x$ path, and $a$ and $b$ store the two terminal endpoints of this sub-path.

The procedure call starts with $x = p = u$ and $c = w_{max} = 0$. The vertex $x$ explores its neighbors $y$ except for its parent and assigns $x$ as the parent for each neighbor $y$. If a neighbor $y \notin S$, then the procedure explores neighbor $y$ with the updated current weight $c$ as $c + w(x, y)$. If a neighbor $y \in S$, then the weight of this terminal-to-terminal sub-path $p \rightsquigarrow y$ is given by $w(p \rightsquigarrow y) = c + w(x, y)$. If $w(p \rightsquigarrow y)$ is greater than the maximum weight $w_{max}$, then $w(p \rightsquigarrow y)$ is assigned to $w_{max}$, the endpoint $a$ stores the terminal $p$, and the endpoint $b$ stores the terminal $y$. Otherwise, the maximum weight remains unchanged. The Procedure recursively explores the terminal $y$ by resetting the current weight $c = 0$ and $p = y$. If a neighbor $y = v$, then the procedure returns the endpoints $a$, $b$, the maximum weight $w_{max}$, and ends the exploration.

### 3.2.5 Analysis

This section formally analyzes the performance of our proposed algorithm, Algorithm 3. We prove that it effectively maintains a $(2-\epsilon)$-approximate Steiner tree in the dynamic graph under a series of edge insertions. Furthermore, we demonstrate that the algorithm achieves an efficient worst case update time complexity of $O(nD_s)$. The proofs leverage the specific constructs and techniques employed by the algorithm. We say an edge $(u, v)$ is a *crossing edge* if the roots of $u$ and $v$ differ (i.e., $root(u) \neq root(v)$) in $F$. In simpler terms, a crossing edge connects terminals belonging to separate trees within $F$.

**Lemma 3.10.** *The insertion of a new edge can only decrease the distance (shortest path) of at most one of its two endpoints from their corresponding roots in $F$.*

*Proof.* Let the notation $G$ and $G'$ represent the graph prior to and later the insertion of an edge $(u, v)$, respectively. Let $F'$ be a shortest path forest of $G'$. For a vertex $x$, $dist_F(x)$ and $dist_{F'}(x)$ signify the distances from the nearest root in $F$ and $F'$, respectively. For the sake of contradiction, let us assume that the inserted edge decreases the distance of both of its endpoints from their corresponding roots (which might be the same as the previous roots), that is, $dist_{F'}(u) < dist_F(u)$ and $dist_{F'}(v) < dist_F(v)$.This implies that the shortest path for $u$ must utilize the newly inserted edge $(u, v)$ (otherwise, $u$ would utilize the same path, and $dist_{F'}(u) = dist_F(u)$). Hence $dist_{F'}(u) = dist_F(v) + w(u, v)$. Similarly, $dist_{F'}(v) = dist_F(u) + w(u, v)$. Summing up the

two inequalities yields $dist_{F'}(u) + dist_{F'}(v) = dist_F(u) + dist_F(v) + 2w(u, v)$. It is a contradiction because $dist_{F'}(u) + dist_{F'}(v) < dist_F(u) + dist_F(v)$ and $w(u, v) > 0$ according to our assumption statement. Based on the contradiction, the assumption that both $u$ and $v$ have shorter distances is proven to be wrong. Therefore, the lemma concludes that the edge inserted as an update can only shorten the distance of at most one of its endpoints from the roots in $F$. □

**Theorem 3.11.** *Following each update (edge insertion) in the dynamic graph, Algorithm 3 successfully maintains a valid Steiner tree structure.*

*Proof.* The algorithm starts with a precomputed $2(1-1/\ell)$-approximate Steiner tree $T_s$ for the given graph, built using a method from Wu et al. [16]. This initial tree efficiently connects the designated terminals. The shortest path forest $F$ is constructed within the graph, with terminals acting as the root vertices of each tree. When a new edge $(u, v)$ is inserted, the algorithm checks if it affects the weight of the shortest path of either endpoint within $F$. If so, $F$ undergoes an update process. This involves examining all descendant vertices (connected vertices) of the affected endpoint and adjusting their weights of the shortest paths accordingly.

In $F$, all vertices, except for the roots, are non-terminals. It is important to note that modifications to the shortest paths between terminals (which are also the roots in $F$) only occur under specific conditions. This can happen only when a new edge or an edge being inspected by the algorithm, denoted by $(i, j)$, connects vertices from different trees in $F$. In such a case, if the edge $(i, j)$ creates a shorter path between the root vertices of the two trees (i.e., $DIST(i) + w(i, j) + DIST(j) < dist(root(i), root(j))$), the algorithm replaces the existing shortest path between these roots in $T_s$ with this newly discovered shorter path. If the inclusion of the updated shortest $root(i) \rightsquigarrow root(j)$ path creates a cycle, a sub-path of maximum weight (with no terminal inside) connecting two terminals in the previous shortest $root(i) \rightsquigarrow root(j)$ path is deleted to avoid the cycle formation and minimize the weight of $T_s$.

This process effectively disjoins and rejoins the affected terminals within $T_s$ using the newly discovered shorter path. It is crucial to note that there remains only one path connecting these two terminals after the update. Since all other components of the Steiner tree are preserved, the updated structure continues to satisfy the core properties of a Steiner tree: it connects all terminals and avoids unnecessary loops, adhering to the fundamental definition of a Steiner tree. □

### Approximation Factor

Kou et al. [14] present a method for obtaining a $(2-\epsilon)$-approximate Steiner tree. Their approach leverages the concept of a metric closure on the terminals of the graph. This involves creating a complete graph where edge weights represent the distances between terminals in the original graph. A minimum spanning tree is then generated on this complete graph. Subsequently, each edge in the minimum spanning tree is replaced with the corresponding shortest path between the connected terminals within the original graph. Finally, the resulting structure is processed to eliminate cycles and remove non-terminal leaves, yielding the approximate Steiner tree. These shortest paths can be efficiently computed using the algorithm proposed by Wu et al. [16]

**Theorem 3.12.** *(Wu et al. [16]) For a connected undirected weighted graph $G = (V, E, d)$, and a set of vertices $S \subseteq V$, a Steiner tree $G_s$ can be computed in $O(m \log n)$ time having total weight at most $2(1 - 1/\ell)$ times that of an optimal Steiner tree.* □

Here, $\ell$ is the minimum number of leaves in any optimal Steiner tree.

**Theorem 3.13.** *The solution maintained by the Algorithm 3 is a $(2-\epsilon)$-approximate Steiner tree.*

*Proof.* Our algorithm maintains a collection of shortest path trees, denoted by $T_1, T_2, \ldots, T_{|S|}$, each rooted at a designated terminal. When an edge, denoted by $e(u, v)$, is inserted into the graph, it's possible that the shortest path between some terminals may change. Let's consider two trees, $T_i$ and $T_j$, where $1 \le i, j \le |S|$ and $i \ne j$. Depending on the specific edge insertion and tree structure, we can categorize the potential impact into two distinct cases:

- **Case 1.** $\mathbf{u \in T_i, v \in T_j}$**:** As the vertices $u$ and $v$ belong to different trees in $F$, the edge $(u, v)$ is a crossing edge, and it connects the trees $T_i$ and $T_j$. The edges $(u_4, u_{10})$ and $(u_4, u_{11})$ in FIGURE 3.6 constitute examples of such edges. Algorithm 3 efficiently identifies such edges by verifying if $root(u) \ne root(v)$. The edge $(u, v)$ creates a new path solely between the root vertices, $root(u)$ and $root(v)$, without impacting any other shortest paths between terminals. The newly formed path might be shorter than the previous shortest path (e.g. $(u_4, u_{11})$ in FIGURE 3.6 creates a shorter path connecting $s_1$ and $s_2$) or it might not be shorter, (e.g., in the case of $(u_4, u_{10})$ in FIGURE 3.6). Algorithm 3 identifies it by comparing whether $DIST(u) + w(u, v) + DIST(v) < dist(root(u), root(v))$. In this case, either the existing path $(p1 \rightsquigarrow root(u) \circ (p_1, p_2) \circ p_2 \rightsquigarrow root(v))$

connecting the corresponding root vertices within $T_s$ is replaced by the newly discovered shorter path ($u \rightsquigarrow root(u) \circ (u, v) \circ v \rightsquigarrow root(v)$) (*Case* 1 *of Section 3.2.4*), or the new shortest path replaces the maximum weighted sub-path of the previous shortest $root(u) \rightsquigarrow root(v)$ path (*Case* 2 *of Section 3.2.4*).

- **Case 2.** $\mathbf{u}, \mathbf{v} \in \mathbf{T_i}$**:** As the edge $(u, v)$ connects the vertices within the same shortest path tree, it does not affect the shortest paths between any of the designated root vertices This is illustrated by the example edges $(u_2, u_7)$ and $(u_3, u_9)$ in FIGURE 3.5. Furthermore, the addition of the edge $(u, v)$ may not alter the weight of the shortest path for any non-terminal within the graph (e.g., edge $(u_2, u_7)$ in FIGURE 3.5). Otherwise, the edge $(u, v)$ changes the shortest distance of one of its endpoints (e.g., edge $(u_3, u_9)$ in FIGURE 3.5), potentially leading to changes in the shortest paths of all its descendants. As Algorithm 3 updates these distances and paths for the descendants to sustain $F$, it may encounter an edge crossing the tree $T_i$, thus triggering *Case* 1. While updating $F$, the algorithm evaluates this new path against the previous shortest path between the roots of the two trees, making path adjustments if required.

In both scenarios described above, the algorithm guarantees that the shortest paths between terminals are maintained and efficiently updates $F$ to handle subsequent updates. Consequently, the solution maintained by Algorithm 3 remains a $2(1 - 1/\ell)$-approximate Steiner tree. Hence, the Steiner tree maintained by Algorithm 3 is a $(2 - \epsilon)$-approximate Steiner tree for $\epsilon = 2/\ell$. As discussed in Chapter 1, all the leaves in an optimal Steiner tree must be terminals; otherwise, one could simply delete the non-terminal leaves to obtain a Steiner tree of lesser cost, as the non-terminal leaves do not contribute to the connectivity of terminals. Hence, $\ell \leq |S|$. A tree, in general, has at least two leaves, except in the trivial case where the tree consists of only one vertex. Hence, $2 \leq \ell \leq |S|$. Therefore, $\epsilon = 2/\ell \in [2/|S|, 1]$. We can use the cut property of MST to ensure that the collection of paths used in $T_s$ resembles an MST of terminals. The cut property for MST states that the minimum weight edge in any cut $C$ of a graph belongs to all MSTs of the graph. If there are multiple edges in the cut $C$ that have minimum weight, then each such edge belongs to some MST of the graph. It can be observed that if each path among the terminals in the tree computed by the algorithm is replaced by edges of equivalent weights, then the algorithm maintains the cut property in the MST over terminals. Hence, the shortest paths among the terminals in the Steiner tree $T_s$ computed by Algorithm 3 represents an MST

over terminals. □

### Update Time Complexity

**Theorem 3.14.** *The update time complexity of the Algorithm 3 is $O(nD_s)$ in the worst case.*

*Proof.* Lemma 3.10 implies that when an edge is inserted, it can affect the weight of the shortest path of only one of its endpoints. When an inserted edge $(u, v)$ impacts the weight of the shortest path of any of its endpoints, let's say $v$, Algorithm 3 proceeds to update the shortest paths for all descendants of $v$. The if condition in line 3 of Algorithm 3 selects $v$. The update process involves a forward traversal of the edges in $F$ starting from $v$, as executed by the for loop in line 9 of Algorithm 3. Each iteration of this loop typically takes constant time, except in cases where an edge crosses the tree and consequently modifies the weight of the shortest path between terminals. In such a case, the shortest path in the tree is replaced by another shortest path. Each individual distance update within this process can still be performed in constant time. The shortest path to be replaced can be found in $O(D_s)$ time by following predecessors from $p_1$ and $p_2$ if it is present in the tree. Otherwise, the shortest path to be replaced is computed using Algorithm 4 and Algorithm 5 in $O(n)$ time, which is a $DFS$ procedure on a tree, and hence, requires $O(n)$ time in the worst case. The number of edges in each shortest path is $O(D_s)$. Hence, each shortest path update (replacement) requires $O(D_s)$ time. The subsequent steps of the loop are dedicated to maintaining the data structure, which aids in efficiently selecting the next vertices for subsequent iterations. With these operations, all the data structures, and $F$ and $T_s$ are updated. The total update time can be expressed as $O(n)O(1) + O(nD_s) = O(nD_s)$. Consequently, Algorithm 3 ensures that the $(2 - \epsilon)$-approximate Steiner tree in the given graph is updated and sustained within $O(nD_s)$ update time in the worst case scenario for each update. □

## 3.3 Summary

We present a fully dynamic algorithm to maintain a $(2 + \epsilon)$-approximate Steiner tree in planar graphs. The worst case update time complexity of the presented algorithm is $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$. The Steiner tree is updated after a sequence $U$ of updates where $1 \leq |U| \leq \sqrt{n+m}$. Hence, for $k$ number of updates, the average update time of the algorithm is $(|S|^2\sqrt{n} + |S|D + n)/k$ where $1 \leq k \leq \sqrt{n+m}$. The proposed algorithm works nicely in dynamic

scenarios with frequent updates. Its update time scales more favorably, particularly with a moderate number of terminals ($|S|$). This makes it more suitable for graphs that undergo frequent changes. The update time complexity of the proposed algorithm becomes more favorable when the number of terminals $|S|$ and the parameter $D$ are both logarithmic in the number of vertices, specifically $O(\log n)$. In this scenario, the worst-case update time reduces to $O(n^{0.5+\epsilon''}(\epsilon')^{-2}) + O(n)$. This offers a significant improvement compared to the previously proposed $PTAS$ by Borradaile et al. [30]. Their approach has a time complexity of $O(2^{poly(1/\epsilon)}n + n \log n)$.

We propose an incremental algorithm that offers an efficient solution for maintaining a $(2-\epsilon)$-approximate Steiner tree in weighted general graphs that undergo edge insertions. The algorithm achieves this by utilizing a shortest path forest and efficiently processing each update in the worst case scenario with an update time complexity of $O(nD_s)$. This significantly outperforms the approach by Kou et al. [14], which requires recomputing the entire $(2-\epsilon)$-approximate Steiner tree from scratch for each update.

Despite our fully dynamic algorithm handling edge insertion and deletion updates in planar graphs, a fully dynamic algorithm for a Steiner tree in general graphs remains a challenge. Additionally, it would be desirable to improve the update time complexity of the proposed fully dynamic algorithm to update the Steiner tree more efficiently.

It is valuable to discuss extending these results or addressing challenges in specialized cases, such as planar grids and general grids. Planar grids, where vertices are arranged in a grid pattern with edges only between grid neighbors, are indeed a restricted subclass of planar graphs. There remains a notable lack of dedicated studies on the problem within grid graphs. Algorithms explicitly designed for grid graphs with theoretical guarantees appear to be an open or underexplored problem. Compared to arbitrary planar graphs, grids exhibit significantly more regularity and geometric structure. The maximum degree of each vertex is bounded by a small constant (four neighbors in a square grid), and the geometric embedding is fixed and simple, allowing for spatial locality and potential geometric shortcut techniques. This regularity can potentially simplify distance computing techniques used in dynamic Steiner tree algorithms. Given these properties, one can hypothesize that specialized data structures and dynamic algorithms might achieve improved update times or tighter approximation bounds on planar grids compared to planar graphs.

**Challenges and considerations for grids:**

1. **Grid Metrics and Distance Measures:** Grids induce specific shortest path metrics (e.g., Manhattan distance), which can be exploited for faster routing or approximation. Dynamic algorithms might leverage

these metrics to maintain Steiner trees efficiently.

2. **Dynamic Update Patterns:** In grids, edge insertions/deletions occur only between neighboring vertices, limiting update complexities spatially. This locality could be exploited for incremental tree update algorithms.

3. **Algorithmic Opportunities:**

    - Using geometric data structures (quadtrees, range trees) for fast neighborhood queries.
    - Exploiting distance properties in grids for efficient Steiner tree updates.

Although planar grids exhibit beneficial structure, the problem remains challenging. Extending full-fledged dynamic approximations with update guarantees as in planar graphs may require new ideas. Restricted dynamic models or heuristic approaches could provide intermediate progress.

Beyond planar grids, grids in higher dimensions or irregular sparse grids present additional complexities: the combinatorial complexity might increase, embeddings become less structured, and existing planar graph algorithms may not extend easily. Hence, fully dynamic approximate Steiner tree algorithms for general grids remain an open and interesting research direction, with promising areas for exploration inspired by planar cases.

# 4

# A Decremental Algorithm for $(2+\epsilon)$-Approximate Steiner Tree in Planar Graphs

## 4.1 Introduction

In this chapter, we study the problem of maintaining an approximate Steiner tree in planar dynamic graphs under edge deletions. Most of the existing algorithms for the Steiner tree problem are MST heuristic based algorithms, which require computing or maintaining a complete distance graph (metric closure) over terminals. Maintaining a Steiner tree under edge insertion and deletion becomes difficult as the insertion or deletion of edges in the graph can change the whole complete distance graph.

In this chapter, we give a lower bound of $\Omega(n)$ on the worst case update time for maintaining an MST heuristic based $(2-\epsilon')$-approximate Steiner tree in general dynamic graphs undergoing edge insertions and deletions.

We propose a decremental algorithm for the Steiner tree problem in dynamic weighted planar graphs. The graph undergoes a sequence of updates where each update deletes an edge from the graph. The goal is to maintain an approximate Steiner tree of the terminals. The proposed algorithm is based on an initial $(2-\epsilon')$-approximate Steiner tree of the terminals and it maintains a $(2+\epsilon)$-approximate Steiner tree under each edge deletion in $\tilde{O}(\ell\sqrt{n})$ worst case update time. Here, $\ell$ is the maximum number of hops in a $(1+\epsilon'')$-approximate shortest path between any two vertices in a graph, and $\tilde{O}()$ notation hides polylogarithmic factors. The $\epsilon$, $\epsilon'$ and $\epsilon''$ are small fractions. To the best of our knowledge, this is the first decremental algo-

rithm for maintaining a Steiner tree under edge deletions having the specified performance guarantees.

We use a dynamic distance oracle to compute approximate shortest distances efficiently and an $ET$-tree data structure to avoid cycle formation in the approximate Steiner tree. This is significantly better than executing the static algorithm from scratch as proposed by Kou et al. (requiring $O(|S||V|^2)$ time) [14] and Wu et al. (requiring $O(|E|\log|V|)$ time) [16]. Our algorithm is dynamic in the sense that it can handle an arbitrary sequence of edge deletions.

## 4.2 Preliminaries

We consider a planar undirected edge weighted connected dynamic graph $G = (V, E)$ where the edge set $E$ is dynamically changing due to the deletion of edges. The updated graph is represented by $G_{updated} = (V, E_{updated})$, $n = |V|$ and $m = |E|$. Each edge is assigned a positive real weight. The weight of an edge $e$ in a graph $I$ is represented by $w_I(e)$. By $D$, we represent the maximum number of hops in a shortest path between any two vertices in $G$. It differs significantly from the weighted shortest path diameter as it accounts for the number of hops in the paths rather than the weights of the paths. Similarly, $\ell$ denotes the maximum number of hops in a $(1+\epsilon'')$-approximate shortest path between any two vertices in an auxiliary graph $G'$ that we maintain such that the vertex set and the edge set of $G$ and $G'$ are same irrespective of the edge weights. $\epsilon, \epsilon', \epsilon''$ and $\epsilon'''$ are small fractions $(0 < \epsilon, \epsilon', \epsilon'', \epsilon''' < 1)$.

A path and a shortest path between two vertices $u$ and $v$ in a graph $I$ are denoted by $P_I(u, v)$ and $\Pi_I(u, v)$ respectively. The weight of a path $P$ (sum of weights of the edges in the path) is denoted by $w(P)$. The concatenation of paths is denoted by $\circ$. The weight $w(T)$ of a tree $T$ is given by the sum of the weights of the edges in $T$. The complete distance graph (metric closure) of a graph $G(V, E)$ is denoted by $\overline{G}[V] = (V, E')$ where $\forall_{u,v \in V, u \neq v}$, $e(u, v) \in E'$ and $w(e(u, v)) = w(\Pi_G(u, v))$. We assume that the graph remains connected throughout the updates. The Steiner tree in our problem differs from the Steiner tree in [7], where the previous Steiner tree is a sub-tree of the updated Steiner tree or vice versa (depending on insertion/deletion request). This constraint is relaxed in this work.

### 4.2.1 $(2-\epsilon')$-Approximate Steiner Tree:

Kou et al. [14] and Wu et al. [16] independently showed that a $2\,(1 - 1/|S|)$-approximate Steiner tree can be computed in $O(|S|n^2)$ and $O(m \log n)$ time

respectively. Let $\overline{G}[S]$ represent the complete distance graph on $S$. Kou et al. [14] show that computing MST of $\overline{G}[S]$ followed by replacement of the edges of MST with the original shortest path between terminals in $G$ avoiding cycles and pruning non-terminal leaves gives a $(2-\epsilon')$-approximate Steiner tree where $\epsilon'$ is $2/|S|$.

**Lemma 4.1** (Kou et al. [14])**.** *Given a graph* $G(V,E)$ *and a terminal set* $S$*, the minimum spanning tree of* $\overline{G}[S]$ *is a* $2\,(1-1/|S|)$*-approximate Steiner tree.*

It is also known as MST heuristic based $(2-\epsilon')$-approximate Steiner tree. This lemma establishes the basis for most algorithms computing or maintaining an approximate Steiner tree. In our algorithm, we initially take a $(2-\epsilon')$-approximate Steiner tree and maintain a $(2+\epsilon)$-approximate Steiner tree under edge deletion updates.

### 4.2.2 Dynamic Distance Oracle:

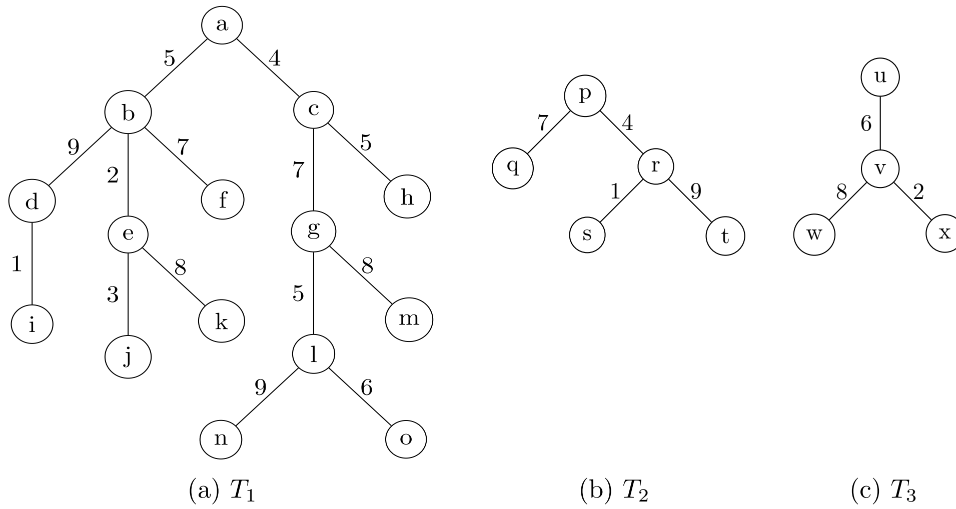
We use the dynamic distance oracle given by Abraham et al. [48] as discussed in Section 3.1.1. It maintains approximate distances in a sketch graph $H$.

**Lemma 4.2** (Abraham et al. [48])**.** $w(\Pi_H(u,v)) \leq (1+\epsilon'')\; w(\Pi_{G_{updated}}(u,v))$.

The approximate Steiner tree splits into two sub-trees due to the deletion of an edge belonging to the tree. We use this distance oracle to find an approximate shortest path between the two sub-trees to connect and maintain an approximate Steiner tree.

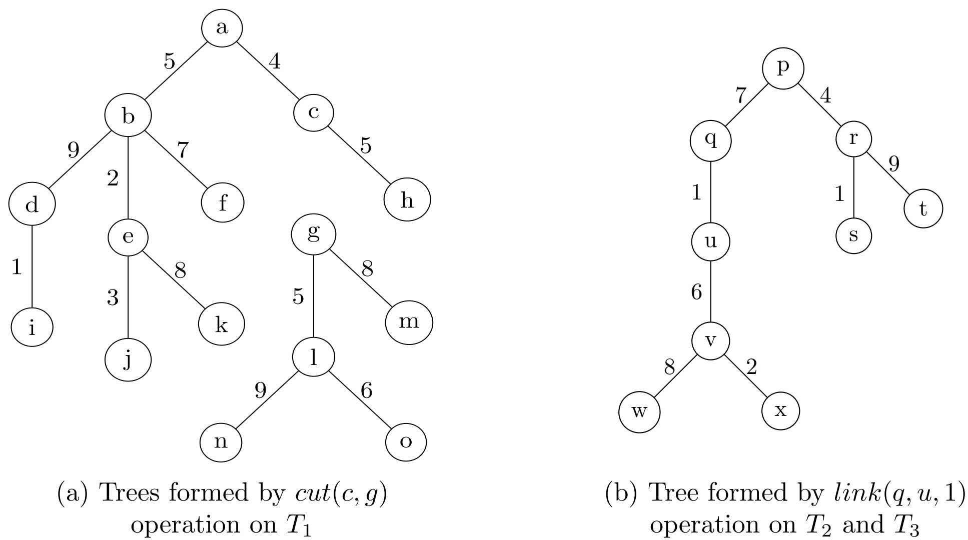
### 4.2.3 ET-Trees:

Henzinger et al. [17] and Sleator and Tarjan [52, 53] presented a dynamic data structure called $ET$-tree. An $ET$-tree is a dynamic balanced search tree over the Euler tours of the trees in a forest. An Euler tour of a tree is a maximal closed walk over the graph obtained from each tree by replacing each edge with a directed edge in each direction. The data structure uses the Euler tours of trees because a tree can be represented as a path by its Euler tour, and balanced search trees can be maintained efficiently over paths rather than trees under the insertions and deletions of edges which require splitting and merging of trees. Each tree in the forest has a root. Let $ET$-*root*$(v)$ be the root of tree containing $v$. Since $ET$-trees are balanced, the tree's root can be found in $O(\log n)$ time.




(a) $T_1$ (b) $T_2$ (c) $T_3$

Figure 4.1: Example trees to show *link* and *cut* operations




(a) Trees formed by $cut(c, g)$ operation on $T_1$

(b) Tree formed by $link(q, u, 1)$ operation on $T_2$ and $T_3$

Figure 4.2: $ET$-tree operations on dynamic trees

$ET$-trees help us to *insert* edges to *join* two trees in the forest by using a *link* operation and *delete* edges to *cut* a tree into two trees by using a *cut* operation. The *link* and *cut* operations are shown in Fig. 4.1 and Fig. 4.2 with three example trees.

The Euler tour is a data structure representing a tree as a sequence of vertices. Each vertex can have one or more occurrences in the tour, depending

on the degree of the vertex. Each edge is visited exactly twice in the Euler tour of the tree. The procedure to find an Euler tour of a tree is shown below.

---

**procedure** ET($x$)
  Visit($x$)
  **for** each child $c$ of $x$ **do**
    ET($c$)
    Visit($x$)
  **end for**
**end procedure**

---

Each vertex, except the leaf vertex, is visited more than once using the above procedure. Each visit of a vertex is called an occurrence of that vertex. The occurrences of a vertex $u$ are denoted as $u_1, u_2, u_3, ..u_i$, where $i = degree\ of\ u$, except for the root. For the root, $i = degree\ of\ u + 1$ . The Euler tour of a tree $T$ can be stored as a $b-$ary search tree, denoted by $ET(T)$-tree, to search an occurrence of any vertex in $O(\log n)$ time. The algorithm proposed by Henzinger and King [17] uses $ET$-tree data structure to maintain an approximate MST. The resulting approximate MST is stored as an Euler tour. This Euler tour is used for subsequent updates on the graph to maintain approximate MST.

## ET-tree operations

1. **Deleting an edge from a tree T:** Deletion of an edge $(u, v)$ from a tree $T$ splits $T$ into two subtrees, say $T_1$ and $T_2$, such that $u \in T_1$ and $v \in T_2$.

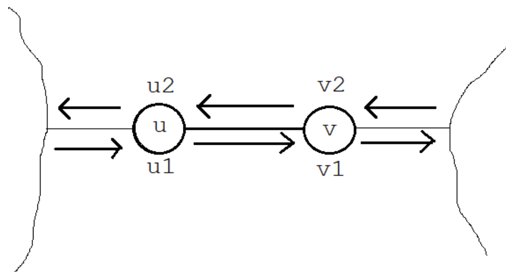



Figure 4.3: Deletion of an edge from Euler tour of a tree

Let $O_{u1}, O_{v1}, O_{v2}$ and $O_{u2}$ be the four occurrences encountered in the traversal of the edge $(u, v)$. The order of occurrences of $u$ and $v$ is

$O_{u1} < O_{v1} < O_{v2} < O_{u2}$ to visit the edge $(u, v)$, where $O_{u1} < O_{v1}$ signifies that $O_{u1}$ occurs before $O_{v1}$ in the traversal. The tree $T_2$ is spliced out from $T$ using the occurrence of $v$. The interval $O_{v1}, \ldots, O_{v2}$ in $ET(T)$ represents the Euler tour for $T_2$. Searching occurrences related to any edge can be done in $O(\log n)$ time using $ET$-trees. Hence, any edge from the tree represented by an Euler tour can be deleted in $O(\log n)$ time.

2. **Changing the root of a tree $T$ from $u$ to $v$:** Let $O_v$ denote the first occurrence of $v$. Remove the part of the sequence before $O_v$, (say $s_1$). Remove the first occurrence of $u$, $(O_u)$ from $s_1$. Append $s_1$ on the end of the sequence, which now begins with $O_v$. Add one new occurrence of $v$ at the end of the sequence.

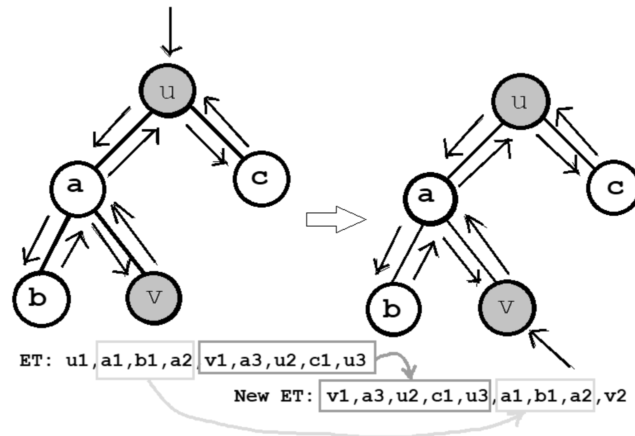



Figure 4.4: Changing the root of a tree from $u$ to $v$

3. **Joining two rooted trees $T_1$ and $T_2$ by an edge:** Let $u \in T_1$ and $v \in T_2$ and the trees $T_1$ and $T_2$ are to be joined by the edge $(u, v)$. The Euler tours for these two trees can be joined using edge $(u, v)$. Re-root the tree $T_2$ at $v$ using the above procedure. Create a new occurrence $O_u$ of $u$. Join the sequence $ET(T_2)O_u$ into $ET(T_1)$ immediately after an occurrence of $u$.

The query *connected*$(u, v)$ of the data structure is used to know whether $u$ and $v$ are connected in an $ET$-tree or not. If *connected*$(u, v)$ returns *true* then $ET$-*root*$(u)$ = $ET$-*root*$(v)$ and vice-versa. The *join, cut* and *connected* operations take $O(\log n)$ time. $ET$-trees support some other useful operations as well [17].

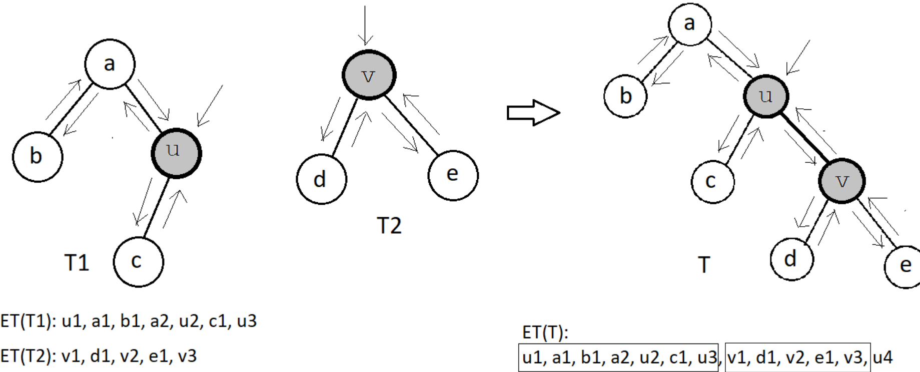



Figure 4.5: Merging Euler tours of two trees $T_1$ and $T_2$ by the edge $(u, v)$

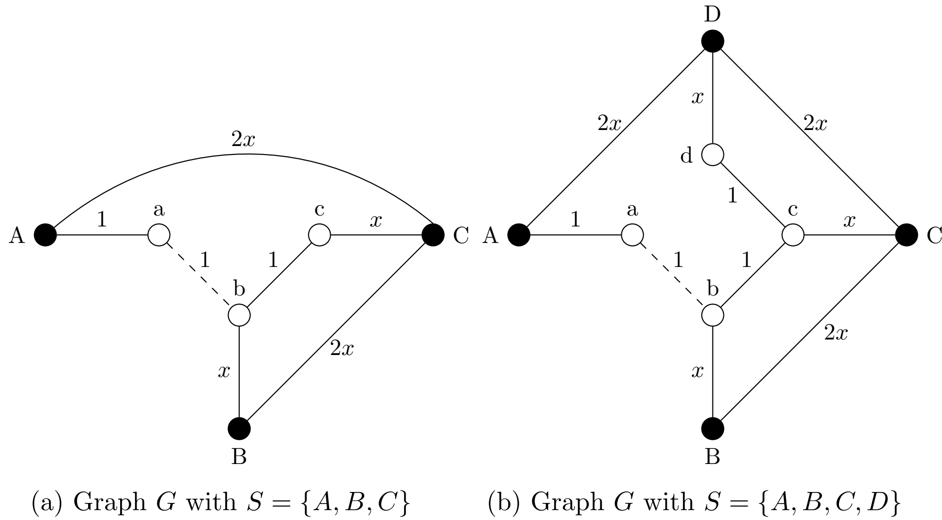
We maintain $ET$-trees on a forest $F$. The Steiner tree that we intend to maintain is contained in $F$. Due to the deletion of an edge belonging to the Steiner tree, the tree splits into two sub-trees. The presented algorithm uses these $ET$-trees to avoid cycle formation with the help of *connected*$(u, v)$ query while connecting the two sub-trees of the Steiner tree by an approximate shortest path. The *join* and *cut* operations are used to efficiently maintain the forest so that the *connected*$(u, v)$ query can be performed efficiently. More details about this data structure can be found in [52, 53].

## 4.3 Lower Bound for MST Heuristic Based Algorithms

The MST heuristic is a fundamental approach for approximating the Steiner tree problem.

**Theorem 4.3** (Lower bound on the update time)**.** *In dynamic graphs with arbitrary edge updates, there exists an instance where MST based heuristic incurs $\Omega(n)$ update time for maintaining a $(2-\epsilon')$-approximate Steiner tree.*

*Proof.* Consider two instances of a dynamic graph $G$ as shown in Fig. 4.6, one with three terminals (Fig. 4.6a) and another with four terminals (Fig. 4.6b). The dashed edge $(a, b)$ initially does not belong to the graph $G$. Fig. 4.7a and Fig. 4.9a show the complete distance graphs on the terminals for the graphs in Fig. 4.6a and Fig. 4.6b respectively. Additionally, Fig. 4.7b and Fig. 4.9b depict MST heuristic based Steiner trees for these instances.




(a) Graph $G$ with $S = \{A, B, C\}$ (b) Graph $G$ with $S = \{A, B, C, D\}$

Figure 4.6: A graph with three terminals (left) and a graph with four terminals (right).

- **Case 1: Update inserts an edge:** Consider an update inserts an edge $(a, b)$ in $G$. The updated complete distance graphs over terminals and the MST heuristic based $(2-\epsilon')$-approximate Steiner trees are shown in Fig. 4.8 and Fig. 4.10. For Fig. 4.8, our assumption is $2x > x + 3$ and $4x > x+2$, which implies $x > 3$, and for Fig. 4.10, $2x > x+4$, $4x > x+3$ and $4x + 2 > x + 2$, which implies $x > 4$. It can be observed that the sets of edges in the updated Steiner trees (Fig. 4.8 and Fig. 4.10) are completely disjoint from the sets of edges in the earlier Steiner trees (Fig. 4.7 and Fig. 4.9). The earlier Steiner tree and the updated Steiner tree together span all the vertices of $G$. This observation indicates that $\Omega(n)$ changes are required to construct a new approximate Steiner tree from an earlier Steiner tree. Hence, the time required to update an MST heuristic based $(2-\epsilon')$-approximate Steiner tree is $\Omega(n)$.

- **Case 2: Update deletes an edge:** Consider instances of dynamic graphs shown in Fig. 4.6 such that the edge $(a, b)$ belongs to the graphs. Fig. 4.8 and Fig. 4.10 show the complete distance graphs over terminals and the MST heuristic based $(2-\epsilon')$-approximate Steiner trees of $G$. An update deletes the edge $(a, b)$ from the graph $G$. The complete distance graphs over terminals and the approximate Steiner trees are updated as shown in Fig. 4.7 and Fig. 4.9. Similar to the insertion case, in this case as well, $\Omega(n)$ changes are required to maintain an MST heuristic based

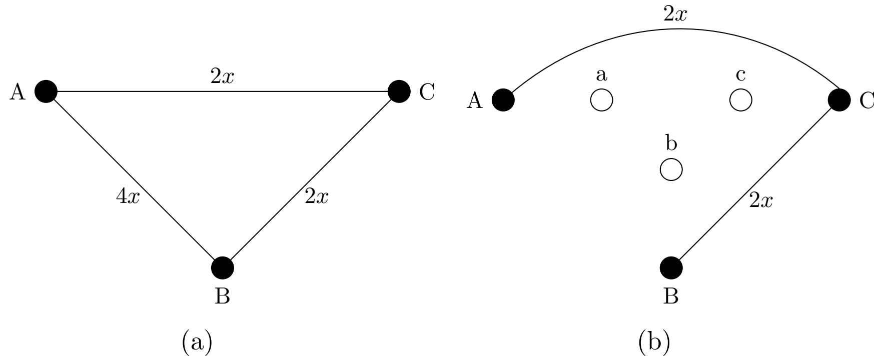



Figure 4.7: The complete distance graph over the terminals (left) and the MST heuristic based $(2-\epsilon')$-approximate Steiner tree (right) for the graph in Fig. 4.6a before insertion of the edge $(a, b)$.

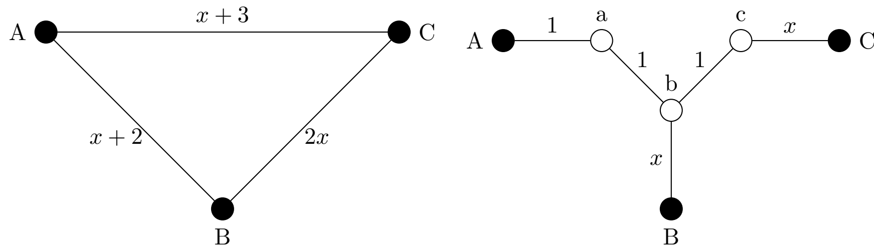



Figure 4.8: The updated complete distance graph over the terminals (left) and the MST heuristic-based $(2-\epsilon')$-approximate Steiner tree (right) for the graph in Fig. 4.6a after insertion of the edge $(a, b)$.

$(2-\epsilon')$-approximate Steiner tree.

We saw instances for three and four terminals. Using a similar structure, we can create instances for any number of terminals. Also, based on the value of $x$, the given graph can be scaled to any number of vertices and edges by replacing the edges of the graph with paths consisting of multiple edges and non-terminals such that the sum of the weights of the edges in a path is equal to the weight of the edge replaced by the path. For example, the graph given in Fig. 4.6b consists of eight vertices. It can be scaled to any $n$ number of vertices (say 1000) by replacing an edge (say $(b, B)$) of the graph with a path consisting of multiple $(1000-8+1)$ edges such that the sum of the weights of the edges in the new path is equal to the weight ($x$) of the edge ($(b, B)$) replaced by the path. Hence, for an arbitrary graph size, there exists an instance such that any MST heuristic based algorithm requires $\Omega(n)$ changes

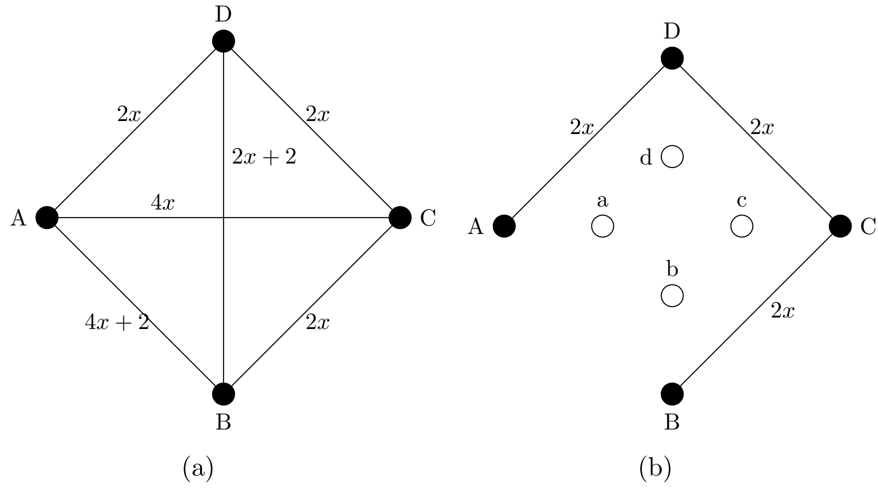



Figure 4.9: The complete distance graph over the terminals (left) and the MST heuristic based $(2-\epsilon')$-approximate Steiner tree (right) for the graph in Fig. 4.6b before insertion of the edge $(a, b)$.

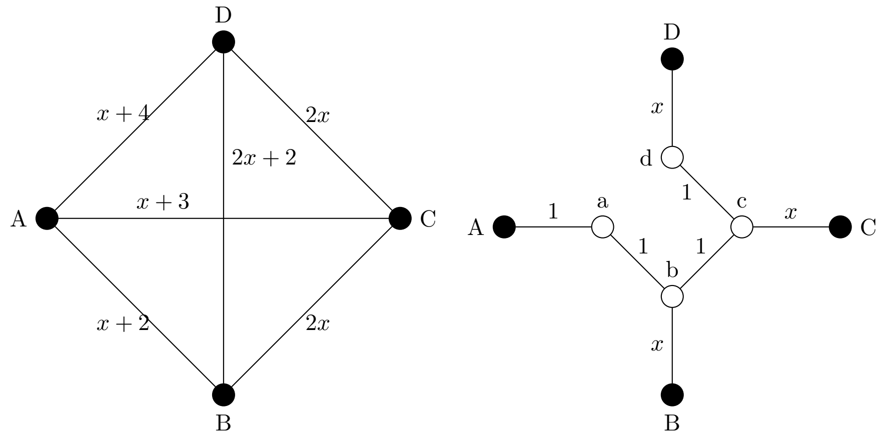



Figure 4.10: The updated complete distance graph over the terminals (left) and the MST heuristic-based $(2-\epsilon')$-approximate Steiner (right) for the graph in Fig. 4.6b after insertion of the edge $(a, b)$.

to update a $(2-\epsilon)$-approximate Steiner tree. □

## 4.4 The Decremental Algorithm

We consider a planar graph instance $G = (V, E)$, a terminal set $S$ and a $(2-\epsilon')$-approximate Steiner tree $T_s = (V_s, E_s)$ on the given graph instance as input to the algorithm followed by some edge deletion updates. We maintain the dynamic distance oracle of Abraham et al. [48] on an auxiliary graph. The tree $T_s$ splits into two disjoint sub-trees due to the deletion of an edge belonging to $T_s$. Let the two sub-trees are $T_{s_1} = (V_{s_1}, E_{s_1})$ and $T_{s_2} = (V_{s_2}, E_{s_2})$. For each pair $(u, v) : u \in V_{s_1}, v \in V_{s_2}$, there is a $(1+\epsilon'')$-approximate $u \rightsquigarrow v$ shortest path in the updated graph $G_{updated}$. Hence there are many $(1+\epsilon'')$-approximate $u \rightsquigarrow v$ shortest paths such that $u \in V_{s_1}$ and $v \in V_{s_2}$. We define the $(1+\epsilon'')$-approximate shortest path between $T_{s_1}$ and $T_{s_2}$ in $G_{updated}$, denoted by $\Pi_{G_{updated}\epsilon''}(T_{s_1}, T_{s_2})$ as min{ $(1+\epsilon'')$-approximate $u \rightsquigarrow v$ shortest path : $u \in V_{s_1}, v \in V_{s_2}$}. Similarly, the shortest path between $T_{s_1}$ and $T_{s_2}$ in $G_{updated}$, denoted by $\Pi_{G_{updated}}(T_{s_1}, T_{s_2})$, is defined as min{$u \rightsquigarrow v$ shortest path : $u \in V_{s_1}, v \in V_{s_2}$}. Let $G_{aux}$ be an auxiliary graph obtained from $G_{updated}$ such that $w_{G_{aux}}(e) = 0$ for $e \in T_{s_1}$ or $e \in T_{s_2}$ and rest of the edges and vertices are the same as $G_{updated}$. A single $(u, v)$ distance query on the dynamic distance oracle on $G_{aux}$ such that $u \in T_{s_1}$ and $v \in T_{s_2}$ can return $\Pi_{G\epsilon''}(T_{s_1}, T_{s_2})$ if the distance oracle supports zero edge weights (Lemma 4.4).

**Lemma 4.4.** *Let $G_{aux}$ be an auxiliary graph such that it differs from $G_{updated}$ only for edges belonging to $T_{s_1}$ and $T_{s_2}$ where the weight of such edges is zero in $G_{aux}$. A single $(u, v)$ distance query on the dynamic distance oracle on $G_{aux}$ such that $u \in T_{s_1}$ and $v \in T_{s_2}$ can return a $(1+\epsilon'')$-approximate shortest path between the two trees $T_{s_1}$ and $T_{s_2}$ given that the distance oracle supports zero edge weights.*

*Proof.* Let $P_1, P_2, P_3, \ldots, P_l$ be the list of all the shortest paths in $G_{updated}$ such that one end point of the path lies in $V_{s_1}$ and the other end point of the path lies in $V_{s_2}$. Let $P_z$ be the shortest path between the trees $T_{s_1}$ and $T_{s_2}$ as defined previously where $1 \leq z \leq l$. Let the end points of $P_z$ are $x$ and $y$ such that $x \in V_{s_1}$ and $y \in V_{s_2}$ without loss of generality. The path $P_z$ cannot contain $T_{s_1}$ or $T_{s_2}$ tree edges, otherwise one could find a sub-path of $P_z$ by deleting $T_{s_1}, T_{s_2}$ tree edges from $P_z$ which is of lesser weight and connects $T_{s_1}$ and $T_{s_2}$, which contradicts that $P_z$ is the shortest path between the trees $T_{s_1}$ and $T_{s_2}$.

For each edge $e \in E_s$, $w_{G_{aux}}(e)$ is zero. As the edge set of $G_{updated}$ and $G_{aux}$ are same except the weight of edges belonging to trees $T_{s_1}$ and $T_{s_2}$,

$$w(P_z) = w(\Pi_{G_{updated}}(T_{s_1}, T_{s_2})) = w(\Pi_{G_{aux}}(T_{s_1}, T_{s_2}))$$

$$\forall(x', y')\ \exists \Pi_{G_{aux}}(x', y') = \Pi_{G_{aux}}(x', x) \circ \Pi_{G_{aux}}(x, y) \circ \Pi_{G_{aux}}(y, y')$$
$$: x' \in T_{s_1}, y' \in T_{s_2}$$

Due to definition of $G_{aux}$,

$$w(\Pi_{G_{aux}}(x', x)) = w(\Pi_{G_{aux}}(y, y')) = 0$$

$$\implies \forall(x', y')\ w(\Pi_{G_{aux}}(x', y')) = w(\Pi_G(x, y)) : x' \in V_{s_1}, y' \in V_{s_2}$$
$$\implies \forall(x', y')\ w(\Pi_{G_{aux}\epsilon''}(x', y')) \leq w(\Pi_{G\epsilon''}(x, y)) : x' \in V_{s_1}, y' \in V_{s_2}$$

Due to this, while finding the shortest path between the two sub-trees in the graph $G_{aux}$ with the help of distance oracle, it acts as all the vertices of $T_{s_1}$ are merged as a single vertex. Similarly, all the vertices of $T_{s_2}$ are considered another single vertex as edges of zero weight connect all the vertices within one sub-tree. Querying the distance oracle for $\Pi_{G_{aux}\epsilon''}(T_{s_1}, T_{s_2})$ by choosing one arbitrary vertex from $T_{s_1}$ and other from $T_{s_2}$, the distance oracle return a $(1+\epsilon'')$-approximate shortest path between $T_{s_1}$ and $T_{s_2}$ in $G_{aux}$ which is same as the $(1+\epsilon'')$-approximate shortest path between the trees in $G_{updated}$. □

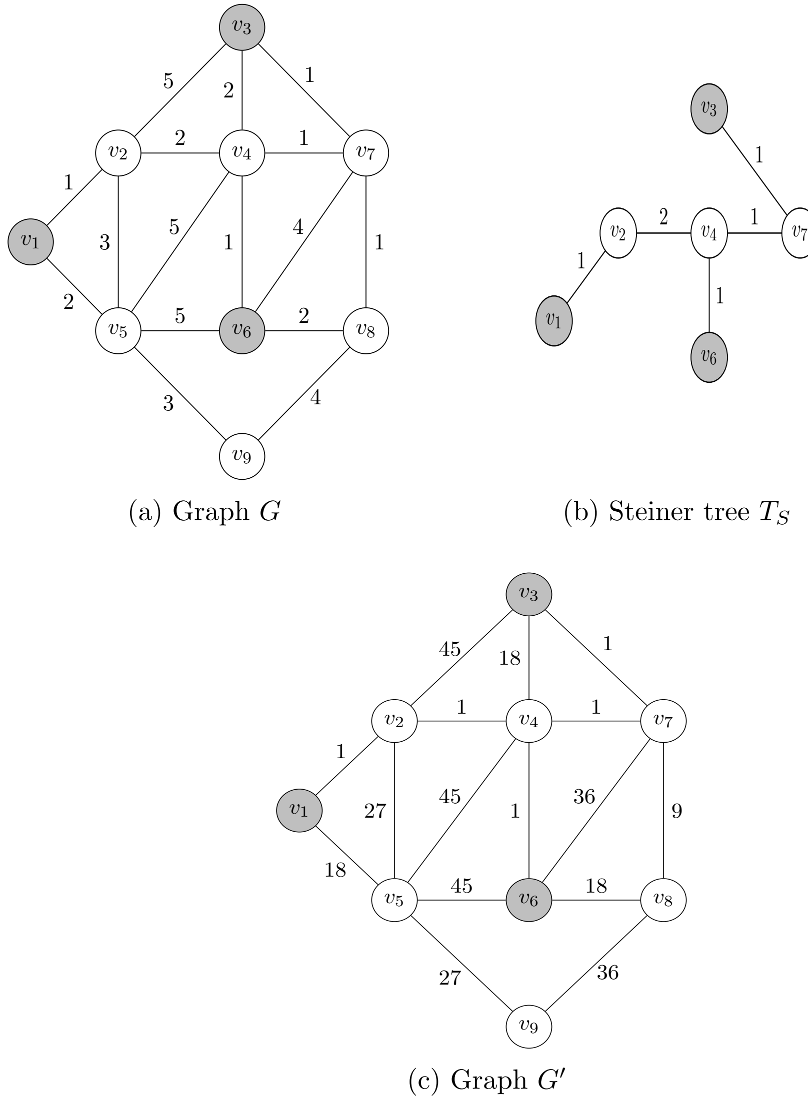
However, the dynamic distance oracle supports only edge weights in $[1, M]$. To get rid of this problem, we keep $w_{G_{aux}}(e) = 1/n$ for edges belonging to $T_{s_1}$ and $T_{s_2}$ and multiply the weight of each edge of $G_{aux}$ by $n$. We call the resulting auxiliary graph $G' = (V, E')$ as shown below and maintain a distance oracle on this graph where edge weights lie in $[1, M'] : M' = nM$.

$$G' = (V, E') : E' = E$$

$$\forall e \in E', w_{G'}(e) = \begin{cases} n \times w_{G_{updated}}(e) : e \notin E_s \\ 1 : e \in E_s \end{cases}$$

Now the distance oracle gives a $(1+\epsilon''')$-approximate shortest path between any given pair of vertices in $G$ rather than $(1+\epsilon'')$-approximate shortest path where $\epsilon'''$ is as shown in Section 4.5.1. We define the $(1+\epsilon''')$-approximate shortest path between the two trees $T_{s_1}$ and $T_{s_2}$ in $G_{updated}$, denoted by $\Pi_{G_{updated}\epsilon'''}(T_{s_1}, T_{s_2})$ as min$\{$ $(1+\epsilon''')$-approximate $u \rightsquigarrow v$ shortest path : $u \in V_{s_1}, v \in V_{s_2}\}$. A forest $F = (V, E_s)$ is maintained using $ET$-trees. It contains all the vertices of $G$ and the edges of $T_s$.

While deleting an edge $(u, v)$ from the graph, the following two cases may occur: The first case is $(\mathbf{u}, \mathbf{v}) \notin \mathbf{E_s}$. In this case, the algorithm deletes $(u, v)$ from $G$ and $G'$. We claim in Lemma 4.7 that deletion of such an edge does not require any modification in the tree $T_s$ to maintain the $(2+\epsilon)$ approximation factor. In the second case, the edge $(\mathbf{u}, \mathbf{v}) \in \mathbf{E_s}$. To handle this case, The



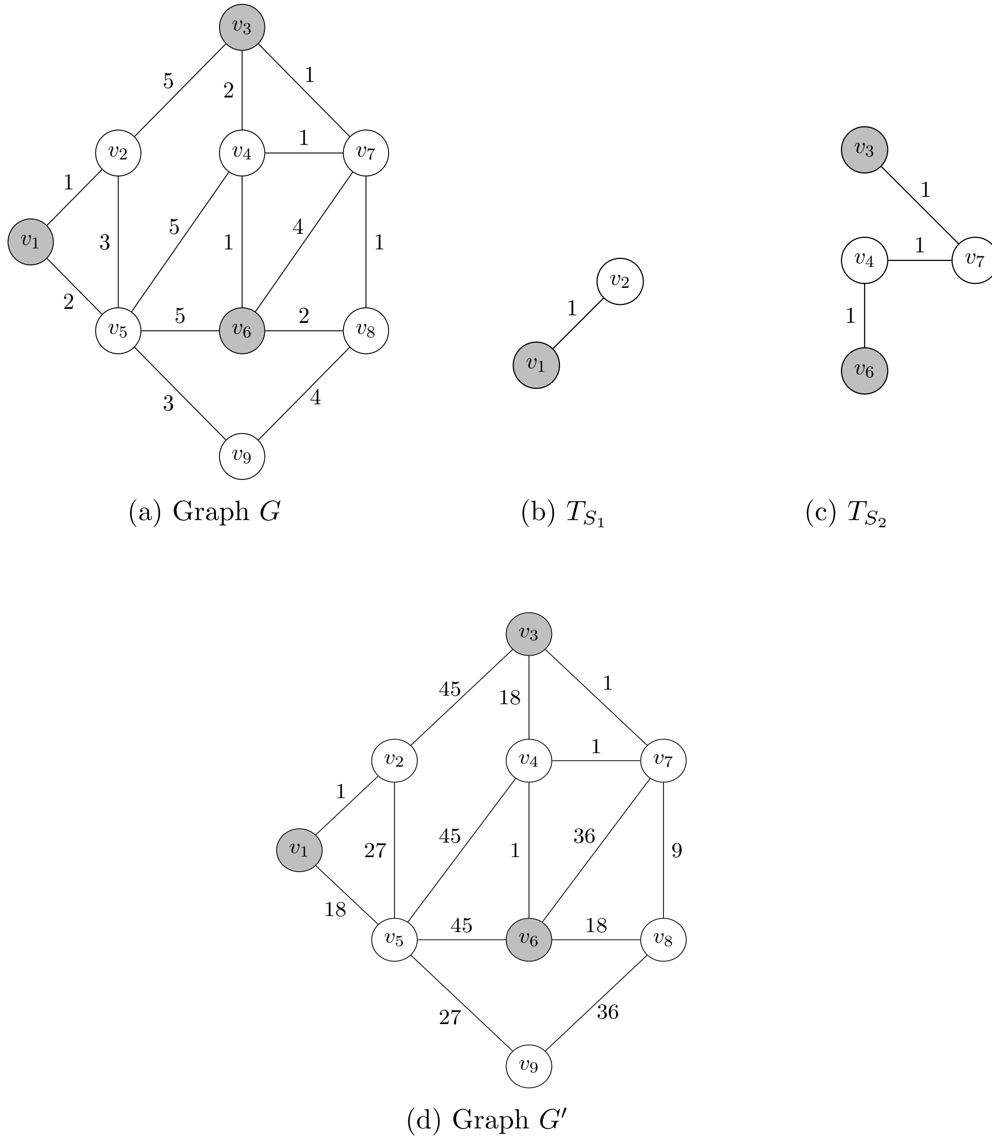
(a) Graph $G$

(b) Steiner tree $T_S$

(c) Graph $G'$

Figure 4.11: Before deletion of the edge $(v_2, v_4)$

algorithm performs the following operations. Delete $(u, v)$ from $G$, $G'$, $T_s$ and $F$. The tree $T_s$ splits into two disjoint sub-trees $T_{s_1}$ and $T_{s_2}$ as shown in Fig. 4.11 and Fig. 4.12. These sub-trees may contain non-terminal leaves. Remove these non-terminal leaves and the corresponding edges from $T_{s_1}$ and $T_{s_2}$. It is shown in Fig. 4.13. Remove the edges corresponding to non-terminal leaves from $F$. For each edge $e$ removed from $T_{s_1}$ and $T_{s_2}$, update the weight




(a) Graph $G$

(b) $T_{S_1}$

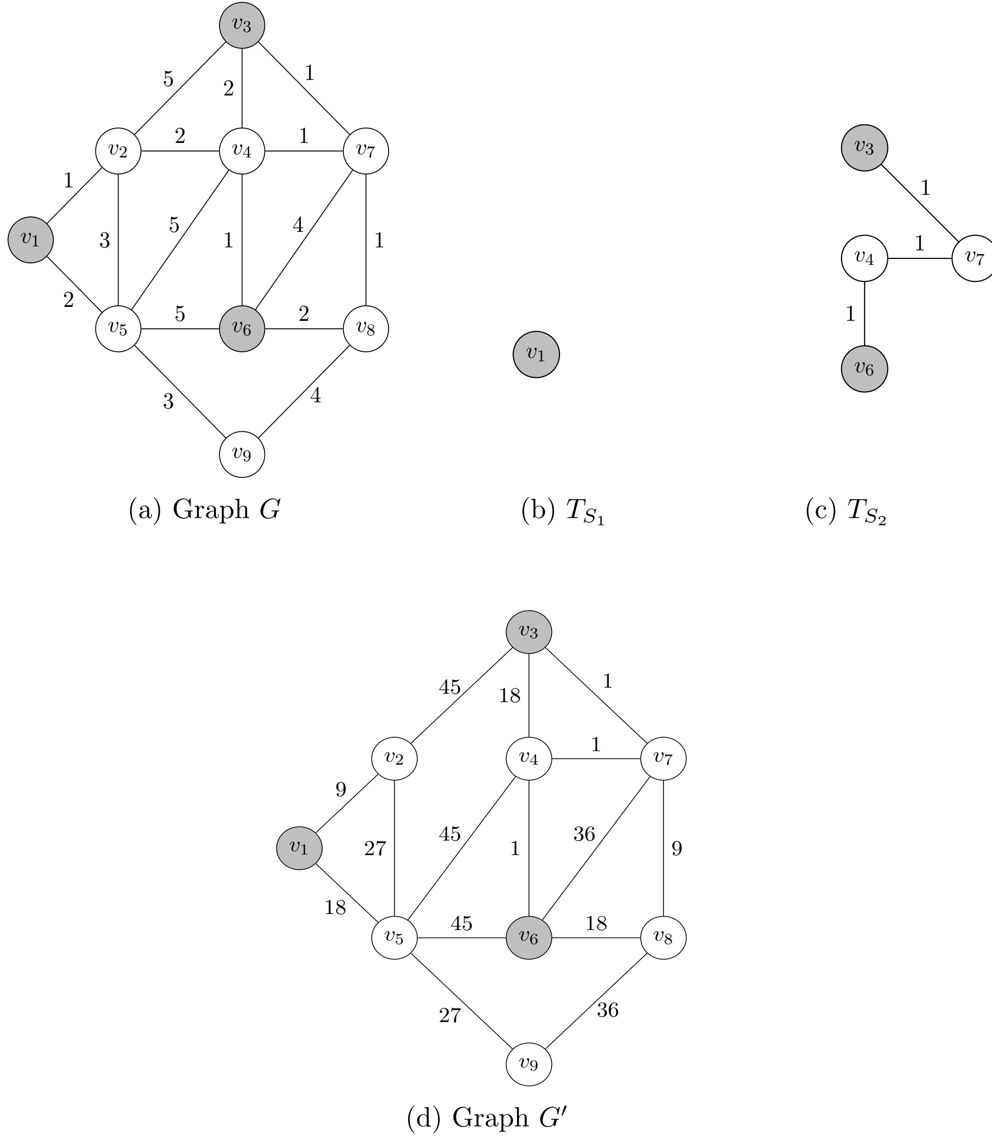
(c) $T_{S_2}$

(d) Graph $G'$

Figure 4.12: After deletion of the edge $(v_2, v_4)$

of $e$ in $G'$ to $n \times w_{G_{updated}}(e) = n \times w_G(e)$. Choose two arbitrary vertices $x \in V_{s_1}$ and $y \in V_{s_2}$. Query the distance oracle for a $x \rightsquigarrow y$ path, which returns a $P_{G'}(x, y) = \Pi_{G'\epsilon''}(x, y) = \Pi_{G\epsilon'''}(x, y)$ path. $P_{G'}(x, y)$ may form a cycle with $T_{s_1}$ and $T_{s_2}$.

**Lemma 4.5.** *Connecting $T_{s_1}$ and $T_{s_2}$ with a $\Pi_{G_{updated}}(T_{s_1}, T_{s_2})$ cannot create*




(a) Graph $G$

(b) $T_{S_1}$

(c) $T_{S_2}$

(d) Graph $G'$

Figure 4.13: Updated $G'$, $T_{S_1}$ and $T_{S_2}$

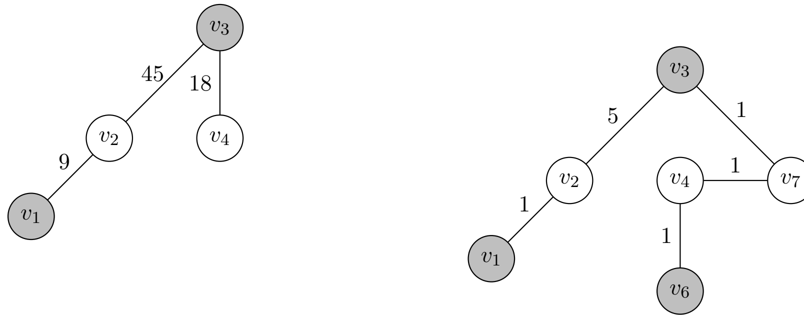
*a cycle.*

*Proof.* Let $P$ be a $u \rightsquigarrow v$ shortest path between $T_{s_1}$ and $T_{s_2}$ in $G_{updated}$, $u \in T_{s_1}$ and $v \in T_{s_2}$. Connecting the two trees with $P$ can only form a cycle if it contains at least two vertices belonging to one of the two trees and edges in $P$ between these two vertices do not belong to the trees. Consider such a

vertex $x \neq u : x \in T_{s_1}$ and $x \in P$. Hence, $P = Q + R$ where $Q$ is a $u \rightsquigarrow x$ path containing non-tree edges and $R$ is a $x \rightsquigarrow v$ path. $w_{G_{updated}}(Q) > 0$. Hence,

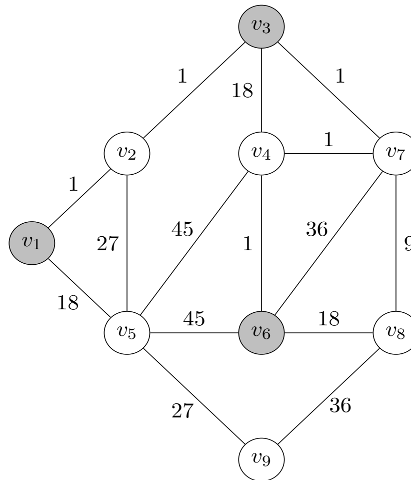
$$w_{G_{updated}}(P) = w_{G_{updated}}(Q) + w_{G_{updated}}(R) > w_{G_{updated}}(R)$$

As $x \in T_{s_1}$, $R$ connects $T_{s_1}$ and $T_{s_2}$. This contradicts that $P$ is the shortest path between $T_{s_1}$ and $T_{s_2}$ in $G_{updated}$. Hence, connecting the disjoint sub-trees $T_{s_1}$ and $T_{s_2}$ with a shortest path in $G_{updated}$ cannot form a cycle. □




(a) Path between vertices 1 and 4 given by the dynamic distance oracle in $G'$

(b) Updated Steiner tree $T_S$




(c) Updated $G'$

Figure 4.14: Updated Steiner tree $T_S$

**Algorithm 6** A $(2+\epsilon)$-Approximate Steiner Tree

1: Take a $2(1-1/|S|)$-approximate Steiner tree $T_s(V_s, E_s)$ ▷ Preprocessing
2: $G'(V, E') \leftarrow G(V, E)$: ▷ Construct $G'$
3: **if** $e \in E_s$ **then**
4: $w_{G'}(e) \leftarrow 1$
5: **else**
6: $w_{G'}(e) \leftarrow n \cdot w_G(e)$
7: **end if**
8: Initialize the distance oracle on $G'$
9: Create the forest $F = (V, E_s)$ using $ET$-trees ▷ Preprocessing ends here
10: An update deletes an edge $e = (u, u')$ from the graph $G$
11: Delete the edge $e$ from $G'$
12: **if** $e \in E_s$ **then**
13: $T_s$ splits into $T_{s_1}$ and $T_{s_2}$ and the tree in $F$ splits into two trees
14: Remove non-terminal leaves and corresponding edges from $T_{s_1}$ and $T_{s_2}$ and remove corresponding edges from $F$
15: $\forall e'$ deleted from $T_{s_1}$ and $T_{s_2}$ and the edge $(u, u')$
16: $w_{G'}(e') \leftarrow n \cdot w_{G_{updated}}(e')$ ▷ Update $G'$
17: Choose arbitrary $x, y : x \in V_{s_1}, y \in V_{s_2}$
18: *Query_Distance_Oracle*$(x, y)$ ▷ Returns $\Pi_{G'\epsilon''}(x, y)$ path
19: **for** $v \in x \rightsquigarrow y$ path (in order from $x$ to $y$) **do**
20: **if** *connected*$(x, v)$ **then**
21: $x' \leftarrow v$
22: **end if**
23: **end for**
24: **for** $v \in x' \rightsquigarrow y$ path (in order from $x'$ to $y$) **do**
25: **if** *connected*$(v, y)$ **then**
26: $y' \leftarrow v$
27: **Break**
28: **end if**
29: **end for**
30: $T_{s_1}$, $T_{s_2}$ and the edges of $x' \rightsquigarrow y'$ path with the edge weights as in $G_{updated}$ form the updated Steiner tree $T_{s_{updated}}$
31: Add the edges of $x' \rightsquigarrow y'$ path to $F$
32: $\forall\ e' \in (x' \rightsquigarrow y') : w_{G'}(e') \leftarrow 1$ ▷ Update $G'$
33: **else**
34: $T_s$ remains a $(2+\epsilon)$-approximate Steiner tree
35: **end if**
36: For a new deletion repeat the process from line 10

**Lemma 4.6.** *Connecting $T_{s_1}$ and $T_{s_2}$ with a $\Pi_{G_{updated}\epsilon'''}(T_{s_1}, T_{s_2})$ may create a cycle.*

*Proof.* Let $P_{G'}(i,j)$ be a $\Pi_{G_{updated}\epsilon'''}(T_{s_1}, T_{s_2})$ where $P_{G'}(i,j) = (i \rightsquigarrow i') \circ (i' \rightsquigarrow j) : i', i \in V_{s_1}, j \in V_{s_2}$ . The $i \rightsquigarrow i'$ path may contain edges that do not belong to $E_{s_1}$ as $P_{G'}(i,j)$ is an approximate shortest path with respect to $G_{updated}$. Since $i, i' \in V_{s_1}$, there is a $i \rightsquigarrow i'$ path in $T_{s_1}$. Adding $P_{G'}(i,j)$ to $T_{s_1}$ may add another $i \rightsquigarrow i'$ path to $T_{s_1}$. Hence, it may lead to cycle creation.

In the given example graph, let $i = v_1$ and $j = v_4$. The distance oracle may return the $(v_1, v_2, v_3, v_4)$ path. Vertices $v_3$ and $v_4$ are already connected in $T_{s_2}$. Hence, combining this path with $T_{s_2}$ creates a cycle. Hence, connecting $T_{s_1}$ and $T_{s_2}$ with this path leads to a cycle creation. □

We use the $ET$-tree data structure to avoid cycle formation while Connecting $T_{s_1}$ and $T_{s_2}$ with a $\Pi_{G_{updated}\epsilon'''}(T_{s_1}, T_{s_2})$. The goal is to find a sub-path $x' \rightsquigarrow y'$ of $P_{G'}(x,y)$ obtained from the distance oracle such that it connects $T_{s_1}$ and $T_{s_2}$ and does not create a cycle. Query the $ET$-tree data structure in the order of vertices present in the $x \rightsquigarrow y$ path $P_{G'}(x,y)$. The vertex $x'$ is found by querying *connected*$(x, v')$ in the $ET$-tree where $v'$ is a vertex in $P_{G'}(x,y)$. The last $v'$ connected to $x$ is $x'$. In the given example, $x' = x = v_1$ because $v_1$ is not connected to any other vertex of $(v_1, v_2, v_3, v_4)$ path in $T_{s_1}$ (Fig. 4.13). Similarly, $y'$ is obtained by querying *connected*$(v', y)$ in the $ET$-tree where $v'$ is a vertex in the $x' \rightsquigarrow y$ sub-path. The first $v'$ connected to $y$ is $y'$. In the given example, $y = v_4$ and $y' = v_3$. It we interchange $T_{s_1}$ and $T_{s_2}$ in the given example, then $v_4$ becomes $x$ and $v_1$ becomes $y$. In this case, $x' = v_3$ and $y' = y = v_1$. The trees $T_{s_1}$ and $T_{s_2}$ together with the edges of this $x' \rightsquigarrow y'$ sub-path with weights the same as in $G_{updated}$ form the updated Steiner tree $T_{s_{updated}}$. As the edges of $x' \rightsquigarrow y'$ belong to $T_{s_{updated}}$, update the weights of these edges to 1 in $G'$. It is shown in Fig. 4.14. Add the edges of $x' \rightsquigarrow y'$ path with weights the same as in $G_{updated}$ to $F$. The tree $T_{s_{updated}}$ is a $(2+\epsilon)$-approximate Steiner tree. There can be $O(n^2)$ paths between $T_{s_1}$ and $T_{s_2}$, but the distance oracle is able to return the required approximate path between the trees in a single query because of the construction of the graph $G'$.

## 4.5 Analysis and Proof of Correctness

**Lemma 4.7.** *Deletion of an edge $(u,v) \notin E_s$ cannot degrade the approximation factor of $T_s$.*

*Proof.* Let $T_{OPT} = (V_{OPT}, E_{OPT})$ be an optimal Steiner tree of the graph $G = (V,E)$ and the terminal set $S \subseteq V$. An update deletes an edge $(u,v) \notin E_s$

from the graph. The updated graph is given by $G_{updated} = (V, E \setminus \{(u,v)\})$. Let $T'_{OPT} = (V'_{OPT}, E'_{OPT})$ be an optimal Steiner tree of $G_{updated}$ such that

$$w(T'_{OPT}) = \sum_{e \in E'_{OPT}} w(e) < \sum_{e \in E_{OPT}} w(e) = w(T_{OPT})$$

As $E'_{OPT} \subseteq E \setminus \{(u,v)\} \subseteq E$ and the terminal set remains the same, $T'_{OPT}$ is also a Steiner tree of $G$. $w(T'_{OPT}) < w(T_{OPT})$ contradicts that $T_{OPT}$ is an optimal Steiner tree of $G$. Hence $w(T'_{OPT}) \geq w(T_{OPT})$. The approximation factor(without modifying $T_s$) in the updated graph is

$$\frac{w(T_s)}{w(T'_{OPT})} \leq \frac{w(T_s)}{w(T_{OPT})}$$

□

### 4.5.1 Approximation Factor

**Theorem 4.8.** *The presented algorithm joins the trees $T_{s_1}$ and $T_{s_2}$ by a $(1+\epsilon''')$-approximate shortest path between the trees.*

*Proof.* Our algorithm maintains the dynamic distance oracle on $G'$. Let $P_1, P_2, P_3, \ldots, P_l$ be the list of all $\Pi_{G_{updated}}(i,j) : i \in V_{s_1}, j \in V_{s_2}$. Let $P_z$ be a $\Pi_{G_{updated}}(T_{s_1}, T_{s_2})$ where $1 \leq z \leq l$ and the end points of $P_z$ are $x$ and $y$, $x \in V_{s_1}$ and $y \in V_{s_2}$. Let $P_{z'G'}$ be the $x' \rightsquigarrow y'$ path returned by the distance oracle for $x' \in V_{s_1}$ and $y' \in V_{s_2}$. For an edge $e \in T_{s_1}$ or $e \in T_{s_2}$, $w_{G'}(e)$ is 1. For all other edges, $w_{G'}(e)$ is $n \cdot w_G(e)$. As the edges of $T_{s_1}$ and $T_{s_2}$ cannot be present in the shortest path between $T_{s_1}$ and $T_{s_2}$,

$$w(\Pi_{G'}(T_{s_1}, T_{s_2})) = n \cdot w(\Pi_{G_{updated}}(T_{s_1}, T_{s_2}))$$

$$\forall (x', y')\ \exists \Pi_{G'}(x', y') : \Pi_{G'}(x', y') = \Pi_{G'}(x', x) \circ \Pi_{G'}(x, y) \circ \Pi_{G'}(y, y'),$$
$$x' \in T_{s_1}, y' \in T_{s_2}$$

Due to definition of $G'$,

$$w(\Pi_{G'}(x', x)) + w(\Pi_{G'}(y, y')) \leq n - 1$$

$$\implies \begin{pmatrix} \forall (x', y')\ w\big((1+\epsilon'')\text{-approximate} \\ x' \rightsquigarrow y' \text{ shortest path in } G'\big) \end{pmatrix} \leq \begin{pmatrix} (1+\epsilon'')\big(n - 1 + \\ n \cdot w(\Pi_{G_{updated}}(x, y))\big) \\ : x' \in V_{s_1}, y' \in V_{s_2} \end{pmatrix}$$

$$\implies w(P_{z'G'}) \leq (1+\epsilon'')(n - 1 + n \cdot w(\Pi_{G_{updated}}(x, y)))$$

$$\implies w(P_{z'G'}) \leq \left(1+\epsilon'' + \frac{1+\epsilon''}{w(\Pi_{G_{updated}}(x,y))}\left(\frac{n-1}{n}\right)\right)\cdot n \cdot w(\Pi_{G_{updated}}(x,y))$$

$$\implies w(P_{z'G'}) < \left(1+\epsilon'' + \frac{1+\epsilon''}{w(\Pi_{G_{updated}}(x,y))}\right)\cdot n \cdot w(\Pi_{G_{updated}}(x,y))$$

$$\implies w(P_{z'G'}) < (1+\epsilon''')\cdot n \cdot w(\Pi_{G_{updated}}(x,y))$$

For $\epsilon''' = \epsilon'' + \frac{1+\epsilon''}{w(\Pi_{G_{updated}}(x,y))}$.

Let $P_{z'G_{updated}}$ be the path in $G_{updated}$ corresponding to the path $P_{z'G'}$ in $G'$: edge $e \in P_{z'G'} \implies e \in P_{z'G_{updated}}, w_{P_{z'G_{updated}}}(e) = w_{G_{updated}}(e)$.

$$\implies w(P_{z'G_{updated}}) < (1+\epsilon''')\cdot n \cdot w(\Pi_{G_{updated}}(x,y))/n)$$

$$\implies w(P_{z'G_{updated}}) < (1+\epsilon''')\cdot w(\Pi_{G_{updated}}(x,y))$$

Hence, the distance oracle returns a $(1+\epsilon'')$-approximate shortest path between $T_{s_1}$ and $T_{s_2}$ in $G'$ which is the same as the $(1+\epsilon''')$-approximate shortest path between $T_{s_1}$ and $T_{s_2}$ in $G_{updated}$. The stretch factor $\epsilon''$ of the distance oracle lies in $(0,1)$. In an arbitrarily large graph, $w(\Pi_{G_{updated}}(x,y))$ can have a large value. Assuming that $w(\Pi_{G_{updated}}(x,y)) > 2$ for large graphs, $\epsilon''' \in (0,2)$. Path $P_{z'G_{updated}}$ is finally trimmed using $ET$-trees to avoid cycle formation before connecting $T_{s_1}$ and $T_{s_2}$ with $P_{z'G_{updated}}$, which cannot increase the cost of the path. □

**Theorem 4.9.** *The presented algorithm maintains a* $(2+\epsilon)$*-approximate Steiner tree.*

*Proof.* Initially we have a $2\,(1-1/|S|)$-approximate Steiner tree $T_s$ given by Lemma 4.1.

$$w(T_s) \leq 2\,(1-1/|S|)\,\mathcal{OPT}$$

Here, $\mathcal{OPT}$ is the weight of an optimal Steiner tree of the given graph $G$. Let $\mathcal{OPT}_{updated}$ denote the cost of an optimal Steiner tree of $G_{updated}$. If the deleted edge belongs to $T_s$, the tree $T_s$ gets divided into two disjoint sub-trees. We remove non-terminal leaves and the corresponding edges in the two sub-trees. These sub-trees are $T_{s_1}$ and $T_{s_2}$. The algorithm connects these two trees with a $\Pi_{G\epsilon'''}(T_{s_1}, T_{s_2})$ (by Lemma 4.2 and Theorem 4.8). By Lemma 4.1,

$$w(MST(\overline{G}[S])) = \sum_{e \in MST(\overline{G}[S])} w(e) \leq 2(1-1/|S|)\mathcal{OPT} \tag{4.1}$$

Where each edge in the $MST(\overline{G}[S])$ has a weight equal to the weight of a shortest path between some two terminals $s_1$ and $s_2$ in the graph $G$, which is

equivalent to the shortest path connecting the two parts of the MST formed by removing the edge $(s_1, s_2)$. The cut property for MST states that the minimum weight edge in any cut $C$ of a graph belongs to all MSTs of the graph. If there are multiple edges in a cut that have minimum weight, then each such edge belongs to some MST of the graph. According to the cut property, the shortest path connecting $T_{s_1}$ and $T_{s_2}$ must belong to MST of $\overline{G}[S]$. Whenever $T_s$, which is initially a $2(1-1/|S|)$-approximate Steiner tree, gets disconnected due to the deletion of an edge, we connect $T_{s_1}$ and $T_{s_2}$ with a sub-path of $\Pi_{G\epsilon'''}(T_{s_1}, T_{s_2})$ as shown in Section 5. Applying the presented algorithm after each update may result in the replacement of a $\Pi_G(T_{s_1}, T_{s_2})$ with a sub-path of $\Pi_{G_{updated}\epsilon'''}(T_{s_1}, T_{s_2})$. It is equivalent to the replacement of each edge of the updated MST over $\overline{G_{updated}}[S]$ by a $(1+\epsilon''')$-approximate edge of updated MST on $\overline{G_{updated}}[S]$, Hence by (4.1)

$$w(T_{s_{updated}}) = \sum_{e \in T_{s_{updated}}} w(e)$$

$$\implies w(T_{s_{updated}}) \leq \sum_{e \in MST(\overline{G_{updated}}[S])} (1+\epsilon''')w(e)$$

$$\implies w(T_{s_{updated}}) \leq 2(1-1/|S|)(1+\epsilon''')\mathcal{OPT}_{updated}$$

$$\implies w(T_{s_{updated}}) \leq 2((1+\epsilon''') - (1/|S|)(1+\epsilon'''))\mathcal{OPT}_{updated}$$

$$\implies w(T_{s_{updated}}) < 2(1+\epsilon''')\mathcal{OPT}_{updated}$$

$$\implies w(T_{s_{updated}}) < (2+2\epsilon''')\mathcal{OPT}_{updated}$$

For $\epsilon = 2\epsilon'''$

$$\implies w(T_{s_{updated}}) < (2+\epsilon)\mathcal{OPT}_{updated}$$

By Lemma 4.7, due to the deletion of an edge that does not belong to $T_s$, $T_s$ remains $(2+\epsilon)$-approximate Steiner tree.

As $\epsilon''' \in (0, 2)$, $\epsilon = 2\epsilon''' \in (0, 4)$. □

### 4.5.2 Update Time

In the preprocessing phase, we have an initial Steiner tree $T_s$. $G'$ can be constructed from $G$ in $O(m)$ time as it differs from $G$ only in edge weights. The distance oracle on $G'$ can be initialized in $O((\epsilon'')^{-1} n \log^2 n)$ time. The $ET$-trees are maintained on a dynamic forest $F$. Initially, $F$ contains the edges of $T_s$ and singleton vertices of $G$. An edge of $T_s$ can be added to $F$ in

$O(\log n)$ time by *join* operation. As $T_s$ may contain at most $n-1$ edges, $F$ can be constructed in $O(n \log n)$ time in the worst case.

**Theorem 4.10.** *The presented algorithm maintains the* $(2+\epsilon)$*-approximate Steiner tree in* $\tilde{O}(\ell\sqrt{n})$ *worst case update time.*

*Proof.* Initially, the terminals in the Steiner tree $T_s$ are connected by shortest paths. After some updates, some or all of these paths may be replaced by $(1+\epsilon''')$-approximate shortest paths. The length of each $(1+\epsilon''')$-approximate shortest path is $O(\ell)$ as it is a sub-path of a $(1+\epsilon'')$-approximate shortest path in $G'$. When an edge $(u,v)$ is deleted from the graph $G$, it is also removed from the graph $G'$. As the distance oracle is maintained on $G'$, each update on $G'$ takes an update time of $\tilde{O}(\sqrt{n})$ to update the distance oracle. Deletion of an edge $e \in T_s$ results in formation of $T_{s_1}$ and $T_{s_2}$. The algorithm removes non-terminal leaves from $T_{s_1}$ and $T_{s_2}$, and the corresponding edges from $T_{s_1}$, $T_{s_2}$ and the forest $F$. Removing an edge from $F$ (a cut on an $ET$-tree) takes $O(\log n)$ time in the worst case. Removing the non-terminal leaves may lead to the removal of $O(\ell)$ number of edges from $F$, which takes $O(\ell \log n)$ time in the worst case. For each edge $e$ removed from $T_{s_1}$ and $T_{s_2}$ due to the removal of non-terminal leaves, we need to update $w_{G'}(e) = n \cdot w_{G_{updated}}(e)$. Each weight update takes $\tilde{O}(\sqrt{n})$ time because it is equivalent to the deletion of an existing edge and the insertion of a new edge that has the updated weight and the endpoints the same as that of the deleted edge. So, the time required to perform this operation is $\tilde{O}(\ell\sqrt{n})$ time. After removing non-terminal leaves, we need to query the distance oracle to obtain the $(1+\epsilon''')$-approximate shortest path connecting the two parts of the tree in $G_{updated}$. It takes $\tilde{O}(\sqrt{n})$ time. We need to find a sub-path of this path if it forms a cycle with the tree, which requires to query $ET$-trees. It takes $O(\ell \log n)$ time. Finally, the algorithm uses this sub-path to connect $T_{s_1}$ and $T_{s_2}$ to form $T_{s_{updated}}$ and update the weights of the edges belonging to this sub-path to 1 in $G'$. This can be done in $\tilde{O}(\ell\sqrt{n})$ time. Overall worst case update time complexity of the algorithm is $\tilde{O}(\ell\sqrt{n} + \ell \log n) = \tilde{O}(\ell\sqrt{n})$. When the deleted edge $e \notin T_s$, then $T_s$ is not required to be modified. Only the deleted edge is removed from $G$ and $G'$, which takes $\tilde{O}(\sqrt{n})$ time to update the distance oracle. □

## 4.6 Summary

This work demonstrates a lower bound of $\Omega(n)$ on the update time for maintaining an MST heuristic based $(2-\epsilon')$-approximate Steiner tree in general dynamic graphs. The updates considered are edge insertions and edge deletions.

We present a decremental Steiner tree algorithm that maintains a $(2+\epsilon)$-approximate Steiner tree with edge deletions in weighted planar graphs in $\tilde{O}(\ell\sqrt{n})$ worst case update time. This is significantly better than computing a $(2-\epsilon')$-approximate Steiner tree from scratch as given by Kou et al.(requires $O(|S|n^2)$ time) and Wu et al.(requires $O(m\log n)$ time). The presented algorithm is also better than our fully dynamic algorithm presented in Chapter 3 in terms of the update time complexity, which requires $\tilde{O}(|S|^2\sqrt{n}+|S|D+n)$ time to update the Steiner tree in the worst case.

The dynamic Steiner tree problem with edge insertions and deletions has not gotten much attention. Maintaining a dynamic Steiner tree with edge insertion as well as edge deletion updates in general graphs is still an open problem. We wish to extend our algorithm for the Steiner tree problem with edge insertions and deletions to general graphs. Our goal is to design an algorithm for maintaining some constant factor approximate Steiner tree in a fully dynamic general graph that can handle edge insertions as well as edge deletions efficiently.

# 5

# Fully Dynamic Algorithms Handling Edge and Vertex Insertions and Deletions and Terminal Conversions

In this chapter, we propose two fully dynamic algorithms for maintaining an approximate Steiner tree supporting six types of updates: edge insertion, edge deletion, vertex insertion, vertex deletion, terminal to non-terminal conversion, and non-terminal to terminal conversion. After each update, the first algorithm maintains a $(|S| - 1)$-approximate Steiner tree in planar graphs. The second algorithm maintains a $(2 + \epsilon)$-approximate Steiner tree in general graphs under the same six types of updates.

## 5.1 Generating Approximate Steiner Tree from a Fully Dynamic Approximate MST of a Planar Graph

### 5.1.1 Preliminaries

We consider a planar weighted dynamic graph $G(V, E)$ and a terminal set $S \subseteq V$. Each edge $e \in E$ is assigned a positive real weight denoted by $w(e)$. The proposed algorithm uses the concept of a complete distance graph on the terminals, the cut property of MST (minimum spanning tree), Euler tour, and Euler tour trees. A path and a shortest path between two vertices $u$ and $v$ in a graph $G$ are denoted by $P_G(u, v)$ and $\Pi_G(u, v)$ respectively. The weight of

a path $P$( sum of weights of the edges in the path) is denoted by $w(P)$.

The approximate MST maintained by the algorithm is represented using the Euler tour data structure known as $ET$-trees. The $ET$-trees data structure is discussed in Section 4.2.3.

#### Complete distance graph on set of terminals

The Complete distance graph (CDG) of the graph $G(V, E)$ on set of terminal $S$ is represented as $CDG(G, S, V', E')$, where $V' = S$ is set of vertices of CDG and $E'$ is the set of edges of CDG. The edge set $E'$ includes an undirected edge between all pairs of terminals with edge weight equal to the shortest distance between them in the graph $G(V, E)$.

**Definition 5.1.** *(*Complete distance graph on the terminals*) The complete distance graph on the terminals for a graph $G(V, E)$ and the terminal set $S$ is defined as a graph $CDG(S, E') : \forall_{u,v \in S, u \neq v} e(u, v) \in E'$, $w(e) = w(\Pi_G(u, v))$.*

#### Cut property of MST

A cut in the graph $G(V, E)$ is the partition of vertex set $V$ into two parts $V_1$ and $V_2$. All the edges $(u, v)$, such that $u \in V_1$ and $v \in V_2$, create the cut set for a cut between $V_1$ and $V_2$. The cut property of the Minimum Spanning Tree states that every minimal weighted edge in the cut set belongs to some Minimum Spanning Tree of the graph.

### 5.1.2 Overview

We use an approximate MST of the graph to maintain an approximate Steiner tree. The algorithm supports six types of updates: insertion of an edge in the graph, deletion of an edge from the graph, insertion of a vertex in the graph, deletion of a vertex from the graph, conversion of a non-terminal to terminal, and conversion of a terminal to a non-terminal. The resulting approximate Steiner tree is maintained by connecting *neighbor terminals* in the approximate MST by the approximate shortest path in the graph. The dynamic distance oracle of Abraham et al. [48] is used to obtain approximate shortest paths among the terminals. The dynamic distance oracle of Abraham et al. [48] is discussed in Section 3.1.1.

**Definition 5.2.** *(Neighbor terminal pair) For a tree $T(V', E')$ rooted at a vertex $v \in V'$ and a terminal set $S \subseteq V'$, two terminals $t_1$ and $t_2$ are said to form a neighbor terminal pair iff:*

- *$t_1$ is an ancestor of $t_2$ or $t_2$ is an ancestor of $t_1$ in $T$*

- *The $t_1 \rightsquigarrow t_2$ path in $T$ does not contain any other terminal*

The algorithm proposed here is modular in nature and requires maintaining an approximate MST and approximate shortest paths between all terminals under the six types of updates mentioned. The approximate MST and approximate shortest paths among terminals do not change due to the conversion of a non-terminal to a terminal and the conversion of a terminal to a non-terminal, as these updates do not change the topology of the graph. Henzinger and King [17] proposed an algorithm to maintain an approximate MST under edge insertions and deletions for fully dynamic graphs in polylogarithmic time. This algorithm is used here to maintain an approximate MST under updates.

The proposed algorithm uses neighbor terminals in the approximate MST to know the relative sequence of terminals. The resulting approximate Steiner tree is computed by connecting neighboring terminal pairs of approximate MST by their approximate shortest paths in the graph. The approximate shortest paths are maintained using the distance oracle proposed by Abraham et al. [48]. The resulting approximate Steiner tree is proved to be $(|S|-1)$-approximate.

The time taken by the proposed algorithm can be represented as $O(t_1 + t_2 + n + |S| t_3)$, where $t_1$ represents the update time to maintain the approximate MST under edge insertion, edge deletion, vertex insertion, and vertex deletion, $t_2$ represents the update time of the distance oracle used to maintain the shortest paths between terminals, $t_3$ represents the query time to find the shortest path between any two terminals in the graph, $n$ is the number of vertices in the graph and $|S|$ is the number of terminals in the terminal set. The algorithm of [17] has an update time of $O((p \log^3 n \log U)/\epsilon)$ for $p$ number of updates to maintain the $(1+\epsilon)$-approximate MST for edge insertions and deletions, where weights of the edges are between 1 and $U$, and $n$ is the number of vertices. The distance oracle proposed by Abraham et al. [48] has an update time and a query time of $\tilde{O}(\sqrt{n})$ for planar graphs.

### 5.1.3 Updating Approximate Steiner Tree

The proposed algorithm maintains an approximate MST after each update. The approximate MST is then used to connect the terminals in the approximate Steiner tree by their approximate shortest path in the graph. The approximate MST is maintained with an approximation factor of $(1+\epsilon)$, and the approximate Steiner tree is maintained with an approximation factor of $|S|-1$. The procedure to maintain the approximate Steiner tree is given below.

**Algorithm 7** $(|S|-1)-$approximate Steiner tree

- STEP 1: Perform the update on the graph or the terminal set.
- STEP 2: Maintain the $(1+\epsilon)$-approximate MST and perform an update query to update the distance oracle.
- STEP 3: Find $(|S|-1)$ Neighbor terminal pairs in the approximate MST.
- STEP 4: Update $(|S|-1)$-approximate Steiner tree by connecting $(|S|-1)$ neighbor terminal pairs by their approximate shortest path.

#### Maintaining approximate MST

The conversion of non-terminal to terminal and terminal to non-terminal does not affect the approximate MST, as the topology of the graph remains unchanged. Hence, there is no need to update the approximate MST after the insertion or deletion of a vertex in the terminal set.

**Maintaining the approximate MST with edge insertions and edge deletions:** After edge insertion and edge deletion, the $(1+\epsilon)$-approximate MST is maintained using the randomized algorithm of Henzinger and King [17]. The algorithm maintains the $(1+\epsilon)$-approximate MST in weighted general graphs. The algorithm maintains the Euler tour of the approximate MST after each edge insertion/deletion in the graph.

**Maintaining approximate MST with vertex insertions:** When a vertex $v$ with degree $d$ is inserted in the graph, $v$ has $d$ number of edges that can connect $v$ to the existing approximate MST. The cut between $\{v\}$ and $V \setminus \{v\}$ in the graph has these $d$ number of edges in its cut set. According to the cut property of MST, the edge having minimum weight in the cut set belongs to MST. Hence we add the minimum weight edge to the approximate MST. The Euler tour of the existing approximate MST is changed by performing a merge operation between the Euler tour of the approximate MST and the new edge. The remaining $d-1$ edges may further reduce the cost of approximate MST. Hence, we perform $d-1$ edge insertion operations for the remaining $d-1$ edges and maintain the approximate MST. Handling edge insertions is discussed in the previous paragraph.

**Maintaining approximate MST under vertex deletion:** When a vertex with degree $d$ is deleted from the graph, it also needs to be deleted from the approximate MST. Due to the deletion of a vertex from the approximate MST, the tree may divide into at most $d$ subtrees, forming a forest. These subtrees need to be connected again to make it a valid approximate MST. This can be done by performing $d$ edge-deletion operations. The edge deletion operations are handled by using the algorithm of Henzinger and King [17].

The deletion of a vertex $v$ from the approximate MST changes the Euler tour of the approximate MST. We perform the deletion procedure on the Euler tour of the approximate MST to remove all $d$ edges. The vertex deletion is modeled as the deletion of edges associated with the deleted vertex. The deletion of an edge belonging to the approximate MST will divide the approximate MST into two subtrees. The replacement edge to connect these subtrees is found using the algorithm of [17]. Then, we perform a merge operation using the replacement edge to merge the Euler tours of the two subtrees.

The algorithm of Henzinger and King is used to maintain the $(1+\epsilon)$-approximate MST. Hence, after $d$ number of edge insertion and deletion operations, the resulting tree remains a $(1+\epsilon)$-approximate MST.

### Maintaining approximate Steiner tree

The approximate Steiner tree is computed by connecting neighbor terminal pairs of the approximate MST by their approximate shortest paths in the graph. According to the definition of the neighbor terminal pairs, one of the terminals in a neighbor terminal pair is an immediate terminal ancestor of the other terminal in the pair. The neighbor terminal pairs are computed by a depth first search *(DFS)* traversal on the approximate MST with an arbitrary terminal as the root. The DFS procedure to find all the neighbor terminal pairs in a tree is shown in Algorithm 8.

There are exactly $(|S|-1)$ unique neighbor terminal pairs possible in any tree. After computing the neighbor terminal pairs, for each $(u, v) \in$ *neighbor terminal pairs*, the algorithm queries the dynamic distance oracle to obtain an approximate shortest path between $u$ and $v$. The dynamic distance oracle returns $(1+\epsilon')$-approximate shortest paths among the terminals in the neighbor terminal pairs ($\epsilon' \in (0,1)$). The neighbor terminal pairs are connected by these approximate paths. The graph so formed may contain cycles. The cycles are removed by deleting the largest weighted edge from each cycle. It results in a tree. The resulting approximate Steiner tree $T$ is obtained by removing non-terminal leaves from the tree so formed. If any leaf node of the resulting tree is not terminal, then we keep removing edges

**Algorithm 8** Neighbor Terminal Pairs

```
Require: T(V, E), S, root ∈ S
Ensure: neighbor_pairs
 1: Initialize:
 2:     Stack ss = [ ]                              ▷ Stack to store terminal ancestors
 3:     neighbor_pairs = [ ]                        ▷ List of neighbor terminal pairs
 4:     Call DFS(root)
 5: procedure DFS(v)
 6:     if v ∈ S then
 7:         if v is not root then
 8:             neighbor_pairs.add({v, ss.top()})
 9:         end if
10:         ss.push(v)
11:     end if
12:     for each child c of v do
13:         DFS(c)
14:     end for
15:     if v ∈ S then
16:         ss.pop()
17:     end if
18: end procedure
```

until all leaf nodes of the tree are terminal nodes. The resulting tree is a $(|S|-1)$-approximate Steiner tree maintained by our algorithm.

### 5.1.4 Proof of Correctness

#### Approximation factor of approximate MST

The conversion of a terminal to a non-terminal and non-terminal to a terminal does not change the graph. Hence, the approximate MST and the approximation factor of the approximate MST do not change due to these two types of updates. For the edge insertion and the edge deletion updates in the graph, the approximate MST is maintained by the algorithm of [17]. The approximate MST under a vertex deletion (from the graph) is maintained by deleting the edges connected to that vertex by the algorithm of [17]. Hence the approximation factor of the approximate MST does not change under the updates- edge insertion to the graph, edge deletion from the graph, and vertex deletion from the graph. For a vertex insertion (to the graph), the proposed algorithm adds the edge associated with the inserted vertex having minimum

weight to the approximate MST. The other edges associated with the inserted vertex having an end vertex in the approximate MST may further reduce the cost of the approximate MST. Hence, the algorithm of [17] is used to insert the remaining edges to maintain the approximate MST and approximation factor of the approximate MST. Hence, after all these six types of updates, the $(1+\epsilon)$-approximate MST is maintained.

#### Approximate Steiner tree

The approximate Steiner tree is obtained by connecting the neighbor terminal pairs by approximate shortest paths in the graph followed by cycle removal. Hence, the resulting tree contains all the terminals, and all leaves are terminals. Hence, the tree maintained by the algorithm is a valid Steiner tree.

#### Approximation Factor of the Steiner tree

The resulting approximate Steiner tree $T$ has neighbor terminals connected by $(1+\epsilon')$-approximate shortest paths in the graph. Let $P_T(t_i, t_j)$ denote the path between $t_i$ and $t_j$ in the approximate Steiner tree $T$ such that $(t_i, t_j)$ is a terminal pair. Hence, For each neighbor terminal pair $(t_i, t_j)$ computed by the DFS procedure:

$$w(P_T(t_i, t_j)) \leq (1+\epsilon')w(\Pi_G(t_i, t_j))$$

Let $T'$ be the minimum Steiner tree for the given graph $G$ and terminal set $S$, then:

$$w(\Pi_G(t_i, t_j)) \leq w(P_{T'}(t_i, t_j))$$

Hence, for each neighbor terminal pair $(t_i, t_j)$:

$$w(P_T(t_i, t_j)) \leq (1+\epsilon')w(P_{T'}(t_i, t_j))$$

The resulting approximate Steiner tree is obtained using the neighbor terminal pairs computed by the *DFS* procedure. The *DFS* procedure starts with a terminal as a root. The number of neighbor terminal pairs remains the same irrespective of the root terminal.

**Theorem 5.1.** *Any tree $T(V', E')$ rooted at any arbitrary terminal node $r \in S$ with terminal set $S \subseteq V'$ has $(|S|-1)$ number of unique neighbor terminal pairs.*

*Proof.* When the DFS traversal is performed on the tree $T(V', E')$ with any arbitrary root $r \in S$, each terminal $s \in S$ has a set of ancestors. The set of

ancestors of a node $s$ includes all the nodes that are in the stack when $s$ is visited for the first time. In the set of ancestors of node $s \in S$, there is at most one terminal node $s' \in S$, such that there is no other terminal node in the path between $s$ and $s'$. The root terminal does not have an ancestor. All terminals, except the root, have exactly one terminal in their set of ancestors such that it follows the conditions of neighbor terminal pairs. Hence there are exactly $(|S|-1)$ number of unique neighbor terminal pairs. □

Hence for all the $(|S|-1)$ neighbor terminal pairs $(t_i, t_j)$, the distance between $t_i$ and $t_j$ in $T$ is at most $(1+\epsilon')$ times the distance between $t_i$ and $t_j$ in an optimal Steiner tree $T'$. Hence, for any neighbor terminal pair $(t_i, t_j)$:

$$w(P_T(t_i, t_j)) \leq (1+\epsilon')w(P_{T'}(t_i, t_j))$$

$$w(P_{T'}(t_i, t_j) \leq \sum_{e \in E'(T')} w_e = OPT$$

$$\therefore d_T(t_i, t_j) \leq (1+\epsilon')OPT$$

The cost of the tree $T$ is given by:

$$Cost(T) \leq \sum_{(t_i,t_j) \text{ is a terminal pair}} w(P_T(t_i, t_j))$$

$$\therefore Cost(T) \leq (|S|-1)(1+\epsilon')OPT$$

$$\therefore \frac{Cost(T)}{Cost(T')} \leq (|S|-1)(1+\epsilon')$$

Hence the approximation factor of the resulting approximate Steiner tree is $(|S|-1)(1+\epsilon')$. Here, $\epsilon' \in (0,1)$ is the tunable parameter used in the distance oracle to maintain the approximate shortest paths among the terminals.

### Complete distance graph and its Spanning tree

The algorithm proposed by Kou et al. [14] computes the $(2-\epsilon)-$approximate Steiner tree using a complete distance graph on the terminals. The approximate Steiner tree is computed by finding an MST of the complete distance graph on the terminals. We prove the approximation ratio of the approximate Steiner tree computed by using a spanning tree of the complete distance graph on the terminals.

**Theorem 5.2.** *For a given graph $G(V, E)$ and a terminal set $S \subseteq V$, any spanning tree of the complete distance graph on $S$ gives an approximate Steiner tree with an approximation factor of $(|S|-1)$.*

*Proof.* The complete distance graph $CDG$ on $S$ has the vertex set as the set of terminals and each edge $(t_i, t_j)$ has a weight equal to the shortest path distance between $t_i$ and $t_j$ in the graph $G$. Let $ST$ be a spanning tree of $CDG$. The weight of an edge $(t_i, t_j)$ in $ST$ is given by $w(t_i, t_j) = w(\Pi_G(t_i, t_j))$

A tree $T$ is computed by replacing each edge $e(t_i, t_j)$ in $ST$ by the corresponding shortest path between $t_i$ and $t_j$ in the graph $G$. The cycles and the non-terminal leaves are removed. Hence,

$$w(P_T(t_i, t_j)) = w(t_i, t_j) = w(\Pi_G(t_i, t_j))$$

Let $T'$ be the minimum Steiner tree for the given graph $G$ and terminal set $S$. The cost of $T'$ is represented by $OPT$.

$$w(P_T(t_i, t_j)) = w(\Pi_G(t_i, t_j)) \leq w(P_{T'}(t_i, t_j))$$

There are $|S| - 1$ edges in $ST$, and sum of all edges of $ST$ is $\sum_{i=1}^{|S|-1} w_{ST}(t_i, t_{i+1})$ and $w_{ST}(t_i, t_{i+1}) \leq Cost(T')$. There are $|S| - 1$ paths in the tree $T$ corresponding to the $|S| - 1$ edges in a spanning tree of CDF on $|S|$ number of terminals.

$$\therefore\ Cost(T) \leq \sum_{i=1}^{|S|-1} w_{ST}(t_i, t_{i+1})$$

$$\therefore\ Cost(T) \leq (|S| - 1) \cdot Cost(T')$$

$$\therefore\ \frac{Cost(T)}{Cost(T')} \leq (|S| - 1)$$

Hence, the cost of the Steiner tree computed using an arbitrary spanning tree of the complete distance graph on the set of terminals has an approximation ratio of $(|S| - 1)$. □

**Update time complexity**

The expected update time of the proposed algorithm for $p$ number of updates is $O($ Approximate MST update time + distance oracle update time + neighbor terminal pairs finding time + time to connect neighbor terminal pairs by approximate shortest paths + time to remove cycles and non-terminal leaves). The $(1+\epsilon)-$approximate MST is maintained using the algorithm of Henzinger and King [17]. The expected update time to maintain the approximate MST for $p$ number of edge insertions and edge deletions is $O((p \log^3 n \log U)/\epsilon)$

[17]. Insertion or deletion of a vertex with degree $d$ results in insertion or deletion of $d$ number of edges in the graph. Hence, the expected update time to maintain the approximate MST for $p$ number of vertex insertions and vertex deletions is $O((dp\log^3 n \log U)/\epsilon)$. The expected update time complexity to maintain the approximate MST for $p$ number of updates is given in Table 5.1. Here, $\delta$ is the maximum degree of a vertex in the updated graph. The

Table 5.1: Update time to maintain $(1+\epsilon)-$approximate MST

| **Operation** | **Expected update time** |
|---|---|
| Insertion and deletion of $p$ number of edges from the graph | $O((p\log^3 n \log U)/\epsilon)$ |
| Insertion and deletion of $p$ number of vertices | $O((\delta p\log^3 n \log U)/\epsilon)$ |

worst-case update time to maintain the $(1+\epsilon)-$approximate MST under terminal conversions is $O(1)$. Hence, the approximate MST can be maintained in $O((\delta p\log^3 n \log U)/\epsilon)$ expected time for $p$ number of updates.

The dynamic distance oracle is updated in $\tilde{O}(\sqrt{n})$ time in worst case. The $|S|-1$ neighbor terminal pairs are computed by Algorithm 8 by a DFS procedure on the approximate MST, which requires $O(n)$ time in worst case. The approximate shortest paths between the neighbor terminal pairs can be found in $\tilde{O}((|S|-1)\sqrt{n})$ time in worst case by querying the dynamic distance oracle $|S|-1$ times, once for each neighbor terminal pair. The cycles (if any) and the non-terminal leaves can be removed in $O(n)$ time in worst case [14].

Hence, the expected update time of the proposed algorithm for $p$ number of updates is $\tilde{O}((\delta p \log U)/\epsilon + \sqrt{n} + n + |S|\cdot\sqrt{n} + n) = \tilde{O}((\delta p \log U)/\epsilon + |S|\cdot\sqrt{n}+n)$. For $|S| = o(\sqrt{n})$, the update time of the presented algorithm becomes linear in $n$.

## 5.2 A Fully Dynamic Approximate Steiner Tree Algorithm for General Graphs

In this section, we explore a fully dynamic algorithm for maintaining an approximate Steiner tree in general graphs, supporting six types of updates: edge insertion, edge deletion, vertex insertion, vertex deletion, terminal to non-terminal conversion, and non-terminal to terminal conversion. By integrating dynamic clustering, spanner-based connectivity, and a hybrid distance oracle, the algorithm achieves an approximation factor of $2+\epsilon$ for some tunable $\epsilon > 0$. Furthermore, it achieves efficient updates with an expected time

complexity of $O(m^{1/2} + n^{2/3})$ per update. The algorithm offers a significant improvement over existing dynamic Steiner tree algorithms in terms of flexibility, approximation guarantees, and update efficiency.

### 5.2.1 Techniques Used

#### Clustering and Spanners

**Clustering:** The graph is decomposed into smaller clusters using low-diameter decomposition [50]. This decomposition divides the graph into subgraphs (clusters) with bounded diameters, ensuring that distances between any two nodes in a cluster are small. This property allows Steiner trees to be efficiently computed locally within clusters, minimizing the scope of updates and improving scalability. Clustering reduces the complexity of maintaining global connectivity by enabling localized computations for dynamic updates.

**Spanners:** Spanners are sparse subgraphs that approximate the distances of the original graph while using significantly fewer edges [18]. A $(1+\epsilon)$-spanner ensures that the distance between any two vertices in the spanner is at most $(1+\epsilon)$ times their distance in the original graph. Spanners are used to maintain efficient inter-cluster connectivity in the Steiner tree. Their sparsity minimizes the computational overhead during updates, while their distance-preserving property ensures the approximation factor of the Steiner tree remains bounded.

#### Dynamic Distance Oracle

**Bernstein-Roditty Oracle:** The Bernstein-Roditty distance oracle efficiently handles edge deletions in dynamic graphs by maintaining approximate shortest paths [19]. It achieves an update complexity of $O(\sqrt{m})$ per deletion and constant-time queries. This oracle is particularly suited for decremental updates, ensuring that terminal-to-terminal distances are efficiently maintained as edges are removed.

**Incremental Updates:** For edge insertions, incremental shortest path computations are performed using Dijkstra-like propagation. The updates are restricted to regions affected by the inserted edges, ensuring efficiency. The combination of incremental and decremental techniques provides a hybrid dynamic distance oracle that supports both types of updates seamlessly.

**Localized Updates**

By restricting updates to affected clusters and inter-cluster spanners, the algorithm avoids global recomputation. This localized approach leverages clustering and spanners to minimize the impact of changes, enabling efficient handling of edge and vertex updates as well as terminal conversions.

### 5.2.2 Algorithm Design

**Initialization**

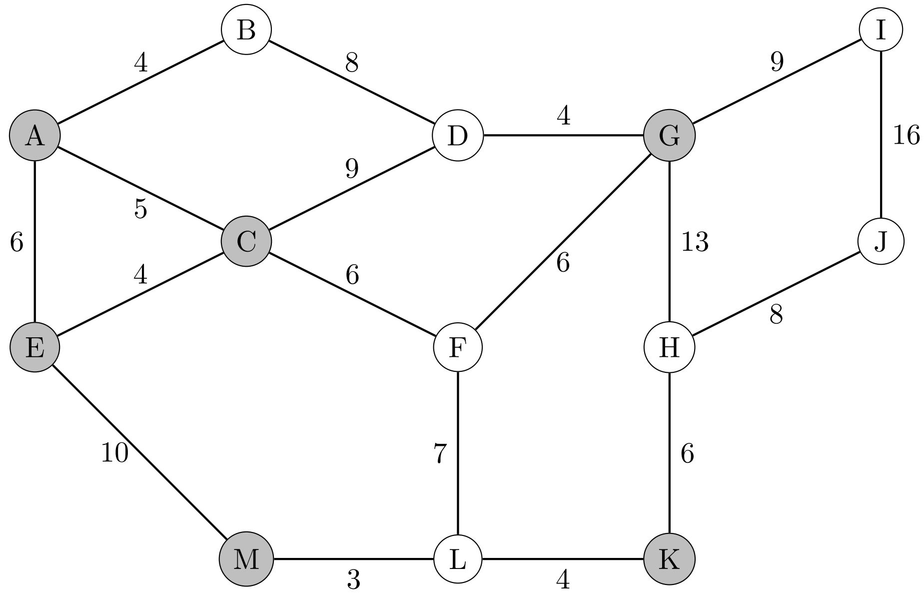
- **Low-Diameter Decomposition:** A low-diameter decomposition divides the graph $G(V, E)$ into disjoint subgraphs (clusters) such that the diameter of each cluster is bounded by a predefined value $\Delta$. The decomposition ensures that vertices within each cluster are close to one another. This step is crucial as it allows the algorithm to limit computations to localized regions of the graph, reducing the overall complexity. Formally, for the given graph $G(V, E)$, the clustering process involves partitioning $V$ into disjoint subsets $C_1, C_2, \ldots, C_k$ such that:

  - Each cluster $C_i$ forms a connected subgraph in $G$.
  - The diameter of each cluster is bounded by $\Delta$, i.e., $\text{diam}(C_i) \leq \Delta$.

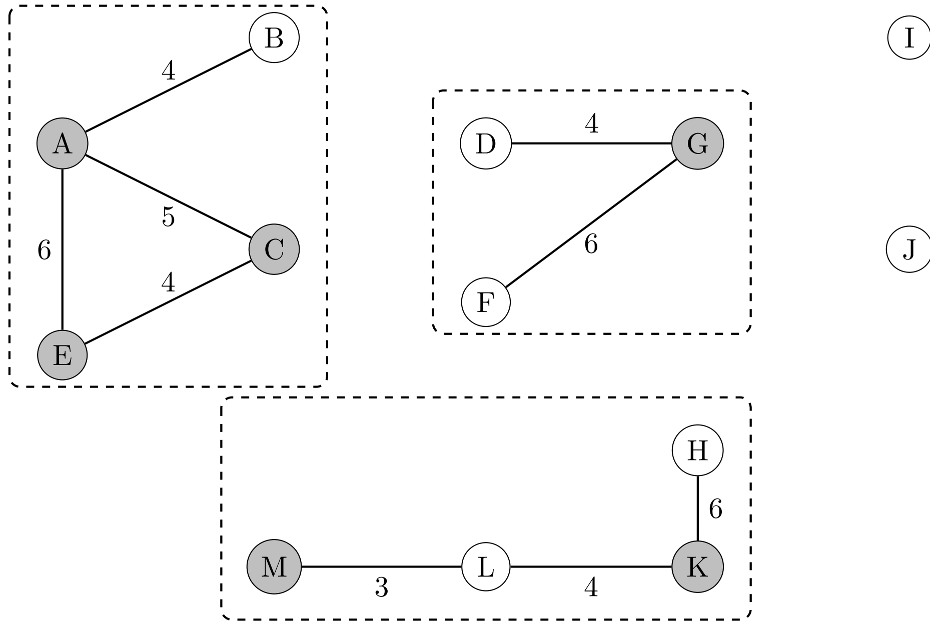
  The process starts by selecting a random vertex $v \in V$ as the cluster center and performing a breadth-first search (BFS) to include all vertices within a radius $\Delta/2$. This ensures that the distance between any two vertices in a cluster is at most $\Delta$. The process is repeated for the remaining unclustered vertices until all vertices are assigned to clusters. Let $d(u, v)$ represent the shortest path distance between vertices $u, v \in V$. The clustering algorithm can be described as:

$$
\begin{aligned}
&\text{Initialize } C = \emptyset, \text{ the set of all clusters.}\\
&\text{While } V \text{ is not empty:}\\
&\quad \text{Select } v \in V \text{ as a cluster center.}\\
&\quad \text{Let } C_i = \{u \in V : d(u, v) \leq \Delta/2\}.\\
&\quad \text{Add } C_i \text{ to } C \text{ and remove all vertices in } C_i \text{ from } V.\\
&\text{Output } C = \{C_1, C_2, \ldots, C_k\}.
\end{aligned}
$$

  Such clustering facilitates efficient computation and updates of local Steiner trees within clusters.




(a) Graph




(b) Clusters

Figure 5.1: An example graph (a) and Clusters (b)

- **Local Steiner Tree Approximation:** An initial Steiner tree is computed for each cluster using the 2 approximation algorithm of Mehlhorn [47]. Formally, for each cluster $C_i$, the algorithm computes a tree $T_i$ that connects all terminals within $C_i$. This ensures that the initial solution within each cluster is a 2-approximate Steiner tree, serving as the foundation for dynamic updates.

- **Construction of Inter-Cluster Spanner:** We use the spanner construction technique proposed by Thorup and Zwick [18], which provides a $(1+\epsilon)$ approximation for distances while ensuring sparsity. This spanner construction is applied to the clusters obtained through low-diameter decomposition, ensuring that terminal-to-terminal distances across clusters are well-approximated.

  The spanner is constructed as follows:

  - **Initialization:** Start with an empty spanner $\mathcal{H} = (V_{\mathcal{H}}, E_{\mathcal{H}})$ where $V_{\mathcal{H}}$ includes all terminals in the graph, and $E_{\mathcal{H}} = \emptyset$.
  - **Path-Based Selection:** For each pair of clusters $(C_i, C_j)$:
    * Identify all terminal-to-terminal paths connecting $t \in C_i$ and $t' \in C_j$. These paths may consist of multiple edges, not just a single edge directly connecting $t$ and $t'$.
    * Sort these paths by total path weight in non-decreasing order.
    * Add a path $P$ to the spanner $\mathcal{H}$ if it reduces the shortest path distance $d_{\mathcal{H}}(t, t')$ in the spanner, ensuring:

    $$d_{\mathcal{H}}(t, t') \leq (1 + \epsilon) \cdot d_G(t, t'),$$

  - **Termination:** Continue this process until all terminal-to-terminal distances across clusters satisfy the $(1+\epsilon)$ approximation guarantee:

  $$d_G(t, t') \leq d_{\mathcal{H}}(t, t') \leq (1 + \epsilon) \cdot d_G(t, t').$$

  **Correctness and Sparsity Guarantee:** The spanner $\mathcal{H}$ ensures that the shortest path distances between terminals in different clusters satisfy the $(1+\epsilon)$ approximation property. By processing paths in ascending order of weight, the construction minimizes the number of edges required in the spanner. The sparsity of the spanner reduces computational overhead while maintaining distance guarantees.

  This spanner construction approach ensures that all terminal-to-terminal distances are approximated efficiently, even when no direct edges exist between the terminals in different clusters.

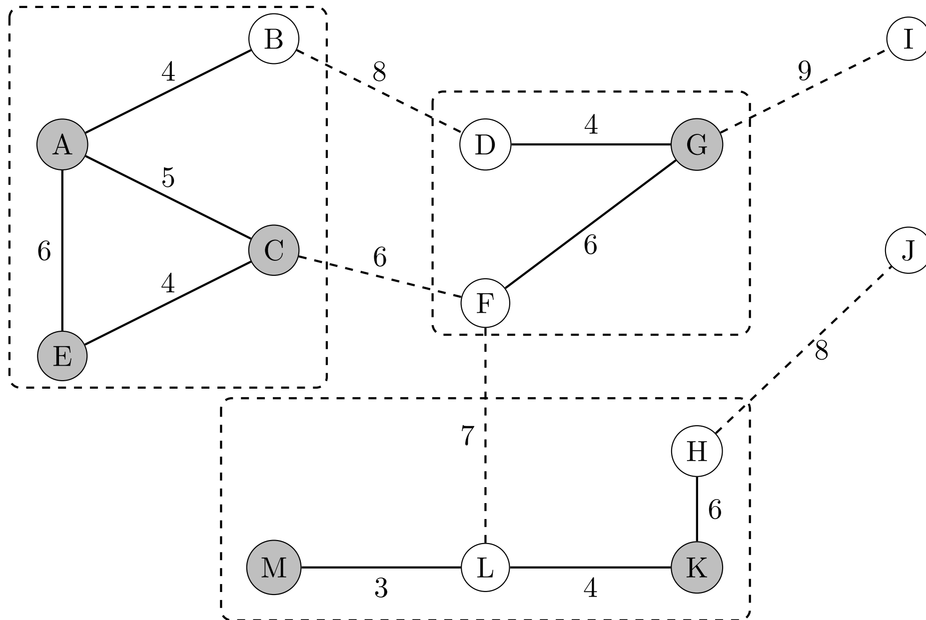



Figure 5.2: The spanner for the updated graph, showing inter-cluster edges (dashed) and intra-cluster edges (solid). Clusters are enclosed with dashed boundaries.

- **Inter-Cluster Connectivity via Spanner:** Clusters are treated as *super-nodes* in a higher-level abstraction of the graph. To connect clusters efficiently, we employ the precomputed spanner $H \subseteq G$. We compute a minimum spanning tree of the graph formed by spanner edges with clusters as *super-nodes*.

  The global Steiner structure is then formed by the union of:

  - intra-cluster Steiner trees (computed independently within each $C_i$), and
  - edges of the minimum spanning tree over the clusters.

  This construction ensures global connectivity while maintaining an approximation guarantee.

- **Hybrid Distance Oracle Initialization:** We augment the decremental distance oracle proposed by Bernstein and Roditty [19] to handle additional types of updates, including edge insertions. The hybrid oracle is maintained over the inter-cluster spanner $\mathcal{H}$, while edge updates are applied to the original dynamic graph $G(V, E)$. This ensures that the

oracle efficiently tracks terminal-to-terminal distances during dynamic updates.

**Existing Functionality:** Bernstein's decremental distance oracle maintains $(1+\epsilon)$-approximate shortest paths for all terminal pairs $(t, t') \in S \times S$. For any terminal pair $(t, t')$, the distance maintained by the oracle satisfies:

$$d_{\mathcal{H}}(t, t') \leq d_{\text{oracle}}(t, t') \leq (1+\epsilon) \cdot d_{\mathcal{H}}(t, t'),$$

The distance oracle handles edge deletions with an amortized update time complexity of $O(\sqrt{m})$ per edge deletion and supports distance queries in $O(1)$ time.

**Augmentation to Handle Edge Insertions in $G(V, E)$:** To handle edge insertions in the original graph $G(V, E)$, we extend the oracle as follows: When an edge $(u, v)$ with a weight $w$ is inserted into $G$, the spanner $\mathcal{H}$ is updated to reflect any changes to inter-cluster distances. If $(u, v)$ connects clusters $C_i$ and $C_j$, the spanner may incorporate the edge if it improves the shortest path distance between affected terminals. The hybrid oracle performs a localized Dijkstra-like propagation in $\mathcal{H}$, restricted to regions where the new edge might improve distances.

**Localized Dijkstra-Like Propagation in $\mathcal{H}$:** Propagation begins from the endpoints of the inserted edge $(u, v)$ in $G$. The propagation is restricted to clusters $C_i$ and $C_j$ connected by $(u, v)$, and the inter-cluster spanner edges in $\mathcal{H}$. For every affected terminal pair $(t, t')$, the oracle updates $d_{\text{oracle}}(t, t')$ by considering:

$$d_{\text{oracle}}(t, t') = \min\left(d_{\text{oracle}}(t, t'), d_{\mathcal{H}}(t, u) + w(u, v) + d_{\mathcal{H}}(v, t')\right)$$

The propagation halts once distances for all relevant terminal pairs stabilize.

**Complete Functionality of the Hybrid Oracle:** After augmentation, the hybrid oracle provides the following capabilities:

- **Edge Deletions in $G(V, E)$:** Handles decremental updates using Bernstein's method, reflected in $\mathcal{H}$.
- **Edge Insertions in $G(V, E)$:** Localized propagation updates terminal to terminal distances in $\mathcal{H}$, reflecting changes caused by the inserted edge.
- **Query Efficiency:** The oracle continues to support $(1+\epsilon)$ approximate distance queries in $O(1)$ time.

**Accuracy and Efficiency Guarantees:**

- The distances maintained by the oracle remain $(1+\epsilon)$-approximate because all updates are applied to the spanner $\mathcal{H}$, which itself preserves $(1+\epsilon)$ approximation guarantees for inter-cluster distances.
- The oracle's query time remains $O(1)$, as the augmented propagation does not increase the complexity of accessing precomputed distances.

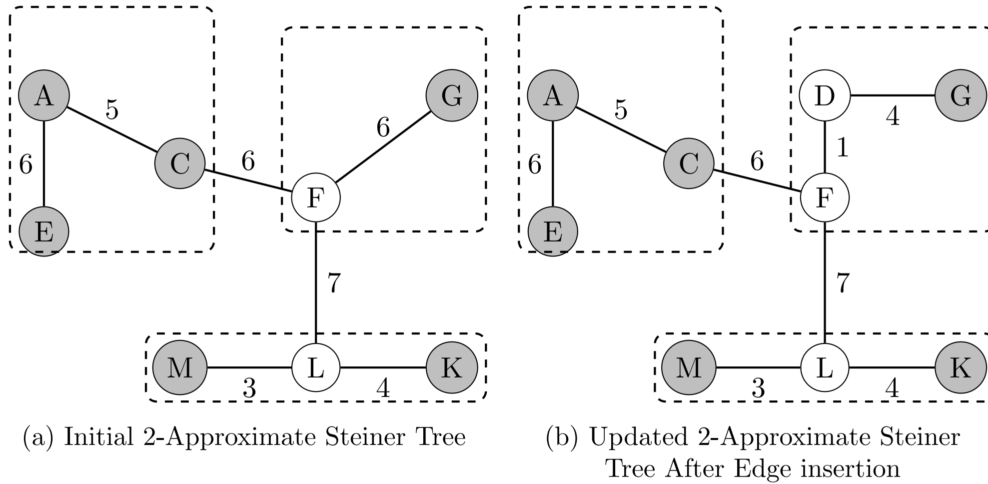
This hybrid distance oracle combines the benefits of Bernstein's efficient decremental method with the flexibility of localized incremental updates, enabling robust and scalable maintenance of terminal-to-terminal distances in dynamic settings.

### Handling Updates

- **Edge Insertion:** When a new edge $(u, v)$ is inserted in the graph $G(V, E)$, the algorithm updates the hybrid distance oracle to reflect the new edge by performing the localized Dijkstra-like propagation. If the inserted edge improves connectivity between the clusters in the spanner $\mathcal{H}$, the algorithm adds it to $\mathcal{H}$. The Steiner tree is updated by considering the affected clusters. The local Steiner tree within each affected cluster is recomputed using the 2 approximation algorithm of Mehlhorn [47]. The inter-cluster connections in the Steiner tree are updated using the spanner $\mathcal{H}$.

  **Updating Inter-Cluster Connections in the Steiner Tree Using the Spanner $\mathcal{H}$:** When a new edge is inserted into $G(V, E)$, or a local Steiner tree is recomputed within a cluster, it is necessary to update the inter-cluster connections in the Steiner tree. This involves determining the affected terminal vertices in clusters that require updated inter-cluster connections. The spanner $\mathcal{H}$ is then leveraged to efficiently recompute shortest paths between terminals across clusters. The spanner $\mathcal{H}$, being sparse, provides $(1+\epsilon)$-approximate distances, ensuring both accuracy and computational efficiency.

  To update the Steiner tree, the algorithm queries the hybrid distance oracle for approximate shortest paths between terminals in different clusters. For each terminal $t$ in an affected cluster $C_i$, the shortest path to terminals in other clusters $C_j$ is found by traversing the spanner $\mathcal{H}$. These inter-cluster paths are then constructed by combining the




(a) Initial 2-Approximate Steiner Tree

(b) Updated 2-Approximate Steiner Tree After Edge insertion

Figure 5.3: Initial 2-approximate Steiner tree (left) and the updated 2-approximate Steiner tree (right) after insertion of $(D, F)$ with weight 1.

intra-cluster Steiner tree connections with the paths traced through $\mathcal{H}$. The updated paths from $\mathcal{H}$ are incorporated into the global Steiner tree structure, replacing outdated or invalid connections, ensuring that the resulting structure spans all terminals in $S$.

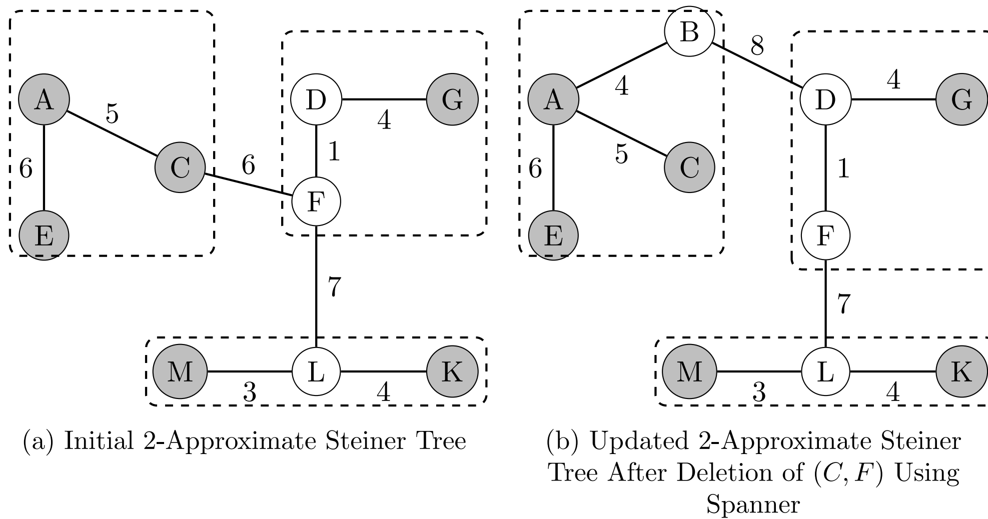
The process preserves the approximation guarantee because $\mathcal{H}$ ensures $(1+\epsilon)$-approximate distances for inter-cluster connectivity. By replacing inter-cluster connections with paths reconstructed from $\mathcal{H}$, the global Steiner tree maintains its approximation factor. Additionally, the sparsity of $\mathcal{H}$ minimizes computational overhead, allowing efficient updates even during frequent edge insertions or local recomputations. This integration ensures that the Steiner tree remains connected, updated, and efficient in dynamic graph scenarios.

- **Edge Deletion:**

  When an edge $e = (u, v)$ is removed from the graph $G(V, E)$, it must also be removed from the spanner $\mathcal{H}$, as it no longer contributes to the connectivity of the graph. The removal of $e$ from $\mathcal{H}$ may break paths that rely on $e$ to maintain approximate distances. To address this, the following steps are performed:

  First, $e$ is directly removed from $\mathcal{H}$, ensuring that the spanner accurately reflects the updated graph. Next, the spanner $\mathcal{H}$ is checked for paths that relied on $e$ to maintain the $(1+\epsilon)$ approximation property for

inter-cluster distances. For each terminal pair $(t, t')$ affected by the removal of $e$, the spanner attempts to restore connectivity by finding an alternative path in the updated graph $G(V, E)$. This is done using the hybrid distance oracle, which provides approximate shortest paths in $G$ efficiently. The spanner $\mathcal{H}$ then adds edges from these newly identified paths, ensuring that the $(1+\epsilon)$ approximation guarantees remain intact. This approach minimizes computational overhead by avoiding global recomputation of the spanner and instead focuses on localized updates for terminal pairs directly affected by the deletion of $e$.




(a) Initial 2-Approximate Steiner Tree

(b) Updated 2-Approximate Steiner Tree After Deletion of $(C, F)$ Using Spanner

Figure 5.4: Initial 2-approximate Steiner tree (left) and the updated 2-approximate Steiner tree (right) after deleting $(C, F)$ with weight 6, updated using the spanner edge $(B, D)$.

By systematically removing $e$ from $\mathcal{H}$ and dynamically rebuilding affected paths, the spanner maintains its accuracy and efficiency, even in fully dynamic scenarios where edge deletions are frequent. This ensures that the Steiner tree relying on $\mathcal{H}$ remains robust and connected, with its approximation guarantees preserved.

- **Vertex Insertion:** Let $N(u)$ denote the neighborhood (adjacent vertices) of $u$. When a vertex $v$ is inserted into the graph, it is assigned to an existing cluster as follows. Since the clustering is a disjoint partition of the vertex set and every existing vertex belongs to some cluster, the assignment of $v$ can be based on its local connectivity. Specifically, the algorithm scans all neighbors $u \in N(v)$ and selects the neighbor

connected by the edge $(v, u)$ of minimum weight. The vertex $v$ is then assigned to the cluster of $u$. This local greedy assignment avoids the need for computing global shortest paths and ensures that $v$ is assigned to the cluster it is most closely connected to.

Next, the Steiner tree for the cluster $C_i$ is updated to incorporate $v$. If $v$ is a terminal, the Steiner tree for $C_i$ is recomputed using the 2-approximation algorithm of Mehlhorn [47], ensuring that all terminals in $C_i$, including $v$, are connected efficiently. If $v$ is not a terminal, its inclusion in the Steiner tree is evaluated based on its potential to reduce the total weight of the Steiner tree in $C_i$. Specifically, the algorithm examines whether $v$ provides shorter paths between existing terminals in the cluster. This is achieved by temporarily introducing $v$ as a candidate Steiner node and recomputing paths between affected terminals. Let $S_i$ be the set of terminals belonging to $C_i$. For each pair of terminals $(t_1, t_2) \in S_i \times S_i$, the algorithm computes the shortest path weight $d_G(t_1, v) + d_G(v, t_2)$ between $t_1$ and $t_2$ via $v$ in the graph $G[C_i \cup \{v\}]$. If this weight is less than $w(P_{T_i}(t_1, t_2))$, the path $P_{T_i}(t_1, t_2)$ is replaced with a path passing through $v$, and the corresponding edges are updated in $T_i$. The updated Steiner tree $T_i'$ is accepted if and only if $w(T_i') < w(T_i)$. In this case, $v$ is incorporated into the cluster's Steiner tree; otherwise, $T_i$ is retained, and $v$ remains disconnected from the Steiner tree.

This process ensures that $v$ is added only if it acts as a useful Steiner node that reduces the overall weight of the tree. Unlike simply attaching $v$ as a leaf, this process ensures that $v$ contributes meaningfully to terminal connectivity. All distance evaluations are performed directly on the subgraph $G[C_i \cup \{v\}]$, as intra-cluster edges are not part of the spanner $\mathcal{H}$. The hybrid distance oracle is not used in this step.

Finally, the spanner $\mathcal{H}$ is updated to reflect the addition of $v$. The algorithm identifies all inter-cluster edges involving $v$, such as edges between $v$ and terminals in other clusters, and evaluates their inclusion in $\mathcal{H}$. These edges are added only if they improve the shortest path distances between terminal pairs across clusters, ensuring that the spanner maintains its $(1 + \epsilon)$-approximation guarantee. Additionally, shortest paths in $\mathcal{H}$ involving $v$ are recomputed, allowing the spanner to remain consistent and efficient. By updating both the Steiner tree and the spanner, the algorithm ensures that the overall structure adapts effectively to the addition of $v$, preserving connectivity and approximation guarantees.

- **Vertex Deletion:** When a vertex $v$ is deleted from $G(V, E)$, it must first be removed from its assigned cluster $C_i$. If $v$ is a terminal, the

Steiner tree for $C_i$ is recomputed using the 2-approximation algorithm of Mehlhorn [47], ensuring that the remaining terminals in $C_i$ are connected efficiently. If $v$ is not a terminal but was part of the Steiner tree, the algorithm evaluates how its removal affects the tree's connectivity. Specifically, paths that previously depended on $v$ are identified, and alternative routes are constructed within the cluster. This is achieved by temporarily treating the nodes adjacent to $v$ as endpoints and recomputing local paths within $C_i$ to restore connectivity, similar to the process of evaluating $v$ as a candidate Steiner node in the Vertex Insertion case.

For inter-cluster connectivity, if $v$ was part of an inter-cluster connection in the Steiner tree, the spanner $\mathcal{H}$ is updated to reflect the deletion. Edges involving $v$ are removed from $\mathcal{H}$, and paths containing $v$ are recomputed to maintain the spanner's $(1 + \epsilon)$-approximation guarantee, as described in the Edge Deletion case. The hybrid distance oracle is updated accordingly to reflect changes in $\mathcal{H}$. Finally, the Steiner tree is updated to replace any disrupted inter-cluster connections by constructing new paths using $\mathcal{H}$, following the same approach as stated in the Edge Deletion case. This ensures that the Steiner tree continues to span all terminals efficiently and preserves its approximation guarantees despite the removal of $v$.

- **Terminal Conversion:** When a vertex $v$ is converted between terminal and non-terminal status, the Steiner tree and spanner $\mathcal{H}$ are updated accordingly. If $v$ is converted into a terminal and already belongs to the Steiner tree, no update is required for the Steiner tree, as it maintains its approximation guarantee by virtue of connecting all terminals using shortest paths, a property ensured by the 2-approximation algorithm of Mehlhorn [47]. If $v$ does not belong to the Steiner tree, it is connected to the existing Steiner tree of its cluster by computing the shortest path from $v$ to the closest vertex in the tree within the subgraph $G[C_i]$. This path is then added to the tree, preserving connectivity. Because this augmentation is done using a shortest path within a bounded-diameter cluster, and only one terminal is being added, the approximation factor remains valid.

  If $v$ is converted from a terminal to a non-terminal, the algorithm examines its role in the Steiner tree. If $v$ is a leaf in the Steiner tree, it is recursively removed along with the edge connecting it, and the process continues until a non-terminal leaf is reached. If $v$ is not a leaf, its removal may disconnect other terminals from the Steiner tree.

**Algorithm 9** Fully Dynamic Steiner Tree Algorithm

**Require:** Graph $G(V, E)$, Terminal Set $S \subseteq V$

**Ensure:** A $(2+\epsilon)$-approximate Steiner tree spanning $S$

1: Initialize the spanner $\mathcal{H}$ using a $(1+\epsilon)$-approximation algorithm [18] for inter-cluster paths.
2: Partition $G$ into clusters $C_1, C_2, \ldots, C_k$ using low-diameter decomposition [50].
3: Compute the local Steiner tree $T_i$ for each cluster $C_i$ using Mehlhorn's algorithm [47].
4: Construct the initial Steiner tree $T$ by combining $T_1, T_2, \ldots, T_k$ and inter-cluster paths from $\mathcal{H}$.
5: **for** each update (edge/vertex/terminal) **do**
6:     **if** Edge Insertion $(u, v, w)$ **then**
7:         Add $(u, v, w)$ to $G$.
8:         Update the hybrid distance oracle to reflect the new edge using localized Dijkstra-like propagation, with complexity $O(m^{1/2})$ in expectation.
9:         Evaluate whether $(u, v, w)$ improves inter-cluster paths in $\mathcal{H}$. If beneficial, add $(u, v, w)$ to $\mathcal{H}$.
10:         Update the Steiner tree $T$ by querying the spanner for affected terminal pairs and adjusting inter-cluster connections.
11:     **end if**
12:     **if** Edge Deletion $(u, v)$ **then**
13:         Remove $(u, v)$ from $G$.
14:         Update the hybrid distance oracle using the decremental algorithm of Bernstein and Roditty [19] in $O(m^{1/2})$.
15:         Remove $(u, v)$ from $\mathcal{H}$, and recompute shortest paths for affected terminal pairs using the hybrid distance oracle.
16:         Update the Steiner tree $T$ by recomputing local Steiner trees for affected clusters and reconnecting inter-cluster paths using $\mathcal{H}$.
17:     **end if**
18:     **if** Vertex Insertion $(v)$ **then**
19:         Add $v$ to $G$ and assign it to the nearest cluster $C_i$ using the hybrid distance oracle.
20:         **if** $v \in S$ (Terminal Vertex) **then**
21:             Recompute the Steiner tree $T_i$ for cluster $C_i$ using Mehlhorn's algorithm.

```
22:         else
23:             Connect v to the Steiner tree T_i if it reduces overall weight,
    using approximate paths from the hybrid distance oracle.
24:         end if
25:         Update inter-cluster connections in H if v introduces new
    inter-cluster edges.
26:     end if
27:     if Vertex Deletion (v) then
28:         Remove v from G and its assigned cluster C_i.
29:         if v ∈ T (Part of the Steiner Tree) then
30:             if v is a leaf then
31:                 Recursively remove v and its connecting path until no
    non-terminal leaves remain.
32:             else
33:                 Identify terminals directly connected to v, and reconnect
    them using approximate paths from the hybrid distance oracle.
34:             end if
35:         end if
36:         Recompute inter-cluster connections in H if v affects inter-cluster
    paths.
37:     end if
38:     if Terminal Conversion (v) then
39:         if v becomes a Terminal then
40:             If v ∉ T, connect it to the Steiner tree using approximate paths
    from the hybrid distance oracle.
41:         end if
42:         if v becomes a Non-Terminal then
43:             if v is a leaf then
44:                 Recursively remove v and its connecting path until no
    non-terminal leaves remain.
45:             else
46:                 Identify terminals directly connected to v, and reconnect
    them using approximate paths from the hybrid distance oracle.
47:             end if
48:         end if
49:         Update H to reflect changes to inter-cluster paths involving v.
50:     end if
51: end for
        return T
```

In this case, the algorithm removes $v$ from the terminal set and recomputes the Steiner tree over the updated terminal set $S_i' = S_i \setminus \{v\}$ within the cluster $C_i$, using the 2-approximation algorithm of Mehlhorn [47]. This guarantees that the updated Steiner tree maintains the required approximation factor after the terminal status of $v$ changes.

For inter-cluster connectivity, if $v$'s terminal conversion affects inter-cluster paths, the spanner $\mathcal{H}$ is updated accordingly. Any inter-cluster paths involving $v$ are recomputed to ensure the $(1+\epsilon)$-approximation guarantee is maintained. These updated paths are incorporated into the Steiner tree to replace any disrupted inter-cluster connections, following the same approach as outlined in the Edge Deletion case. This process ensures that the Steiner tree and spanner $\mathcal{H}$ maintain their efficiency and approximation guarantees after terminal conversion.

### 5.2.3 Analysis

This section provides formal proofs of the correctness, approximation factor, and time complexity of the proposed algorithm under all six types of dynamic updates. The proofs are organized into lemmas and theorems for clarity.

#### Optimal Choice of $\Delta$ in Clustering

The parameter $\Delta$ represents the maximum diameter of a cluster in the low-diameter decomposition used in our algorithm. While its role is well-defined in the clustering process, its optimal value must be carefully chosen to balance **update time complexity, query efficiency, and approximation guarantees**. This section derives the optimal $\Delta$ mathematically by analyzing its impact on **Steiner tree maintenance, inter-cluster spanner connectivity, and distance oracle efficiency**.

The low-diameter decomposition partitions the graph $G(V,E)$ into $k$ disjoint clusters, each with a **diameter at most** $\Delta$. Since each vertex belongs to exactly one cluster, the total number of clusters satisfies $k = O(n/n_c)$, where $n_c$ denotes the number of vertices per cluster. It follows that the number of vertices per cluster is at most:

$$n_c = O(n/k).$$

The expected number of vertices per cluster, $n_c$, is derived based on the assumption that clusters partition the graph while keeping distances between nodes within each cluster at most $\Delta$. The total number of clusters satisfies

$k = O(n/\Delta)$, which follows from graph partitioning results of Fakcharoenphol et al.'s hierarchical decompositions [54] in low-diameter decompositions. Substituting $k = O(n/\Delta)$, into the equation for $n_c$ ensures that each cluster contains at most $O(\Delta)$ vertices. As established by Abraham et al. [55], in graphs with bounded edge density and under the assumption that the number of edges per vertex is proportional to the cluster diameter (as in graphs with low doubling dimension or spatial locality), the number of edges in each cluster satisfies:

$$m_c = O(n_c\Delta).$$

Substituting $n_c = O(n/k)$, we get:

$$m_c = O\left(\frac{n}{k} \cdot \Delta\right).$$

Substituting $k = O(n/\Delta)$, into the equation for $m_c$, we derive:

$$m_c = O(\Delta^2).$$

This confirms that the intra-cluster edge count scales quadratically with $\Delta$.

For efficient Steiner tree updates, the intra-cluster Steiner tree computation must be bounded. The Steiner tree update within each cluster follows Mehlhorn's 2-approximation algorithm, which runs in $O(m + n\log n)$. Since the number of intra-cluster edges is $O(\Delta^2)$, the time complexity of **local Steiner tree recomputation per cluster** becomes:

$$O(\Delta^2 + \Delta\log\Delta).$$

To ensure efficient updates, $\Delta$ must be chosen such that local updates do not dominate the overall complexity.

The impact of $\Delta$ on the inter-cluster spanner connectivity is also critical. The spanner construction guarantees that inter-cluster connections preserve a $(1+\epsilon)$-approximation. The number of inter-cluster edges in the spanner is given by $O(k)$. This does not refer to the total number of edges in the original graph that cross between clusters, which could be as large as $O(m)$, but rather to the number of edges we selectively include in the inter-cluster spanner $\mathcal{H}$. Specifically, the spanner construction adds a constant number of edges between pairs of clusters that contain terminals, ensuring that the number of such inter-cluster representative edges remains small. This sparsification leverages known spanner construction techniques of the Thorup-Zwick framework [18], which guarantee that the total number of edges in the spanner is linear in the number of vertices for fixed stretch $(1+\epsilon)$. Since the number of clusters is $k = O(n/\Delta)$, and each cluster pair is connected by, at most,

a constant number of edges, the total number of inter-cluster spanner edges satisfies:

$$|E(\mathcal{H})| = O(k) = O(n/\Delta).$$

This means that increasing $\Delta$ reduces the number of inter-cluster edges, improving query efficiency but increasing intra-cluster Steiner tree update time.

The decremental distance oracle used in our work (Bernstein-Roditty) supports $(1+\epsilon)$-approximate updates in $O(m^{1/2})$ time per edge deletion. Since updates within clusters dominate the number of affected edges, the amortized cost of updating the distance oracle is:

$$O(m_c^{1/2}) = O(\Delta).$$

Minimizing $\Delta$ improves the oracle update time but at the cost of increasing the number of inter-cluster spanner edges.

To balance the **local Steiner tree update cost** $O(\Delta^2)$ and **distance oracle update time** $O(\Delta)$, the optimal tradeoff is achieved when:

$$O(\Delta^2) = O(n^{2/3}).$$

This ensures that neither update operation dominates the total complexity. Solving for $\Delta$, we obtain:

$$\Delta = O(n^{1/3}).$$

This choice ensures:

- **Efficient intra-cluster Steiner tree updates**, with complexity $O(n^{2/3} + n^{1/3} \log n)$.
- **Optimal inter-cluster connectivity**, maintaining $O(n^{2/3})$ spanner edges.
- **Efficient distance oracle updates**, running in $O(n^{1/3})$ time per update.

Thus, setting $\Delta = O(n^{1/3})$ provides the best balance between intra-cluster Steiner tree maintenance, inter-cluster connectivity, and distance oracle efficiency. This derivation confirms that $\Delta = O(n^{1/3})$ is the optimal choice for balancing update time complexity and ensuring efficient dynamic operations.

**Correctness of the Algorithm** The correctness of the algorithm is established by showing that the updated Steiner tree spans all terminal vertices and remains a tree after each type of dynamic update.

**Lemma 5.3.** *After an edge insertion or deletion update, the maintained structure continues to span all terminal vertices and remains a connected tree.*

*Proof.* For an edge insertion, the hybrid distance oracle is updated to reflect the new shortest paths. If the edge improves inter-cluster connectivity, it is added to the spanner $\mathcal{H}$. The local Steiner trees for affected clusters are recomputed using the 2-approximation algorithm of Mehlhorn [47], while inter-cluster connections are updated using $\mathcal{H}$. These steps ensure that all terminals remain connected and that the resulting structure is a tree.

For an edge deletion, the removed edge is also removed from $\mathcal{H}$ if present, and paths in $\mathcal{H}$ that relied on this edge are updated using the hybrid distance oracle to find alternative shortest paths. As the local Steiner trees are updated using the algorithm of Mehlhorn [47], and the inter-cluster connections are updated using $\mathcal{H}$, the structure remains a tree spanning all the terminals. Hence, the updated Steiner tree spans all terminal vertices and remains a tree after edge insertion or deletion. □

**Lemma 5.4.** *After a vertex insertion or deletion update, the maintained structure continues to span all terminal vertices and remains a connected tree.*

*Proof.* When a vertex $v$ is inserted into the graph, it is assigned to an existing cluster. The algorithm scans all neighbors $u \in N(v)$ and selects the neighbor connected by the edge $(v, u)$ of minimum weight. The vertex $v$ is then assigned to the cluster of $u$. Once assigned, the Steiner tree within that cluster is updated. If $v$ is a terminal, the 2-approximate Steiner tree is recomputed for the cluster using Mehlhorn's algorithm to ensure all terminals remain connected. If $v$ is a non-terminal, it is temporarily introduced as a candidate Steiner node. It is included in the cluster's Steiner tree only if its presence reduces the overall cost of the tree; otherwise, it is excluded. In such a case, an existing path between two terminals is replaced by another path between them via $v$. These operations maintain the tree structure and the properties of a Steiner tree.

In the case of vertex deletion, $v$ is removed from its cluster and from the Steiner tree if present. If $v$ was a leaf or non-terminal, it is simply pruned. If it was internal to the tree or a terminal, the Steiner tree is recomputed locally to ensure connectivity among the remaining terminals. For inter-cluster connectivity, any disruption caused by the removal of $v$ is repaired using approximate paths computed from the hybrid distance oracle.

Therefore, in both vertex insertion and deletion cases, the algorithm ensures that all terminals remain connected through a tree structure, preserving both connectivity and acyclicity. Hence, the maintained structure spans all terminals and remains a valid tree after these updates. □

**Lemma 5.5.** *After a terminal conversion update, the maintained structure continues to span all terminal vertices and remains a connected tree.*

*Proof.* We consider the effect of a terminal conversion update on the maintained structure. Let $T$ denote the current Steiner tree, and let $v$ be the vertex undergoing a change in terminal status.

**Case 1: $v$ is converted into a terminal.** If $v \in T$, no modification to $T$ is required. Since $T$ already spans all existing terminals and includes $v$, it remains a connected tree spanning the updated terminal set.

If $v \notin T$, it is incorporated into the local cluster Steiner tree via a shortest path in the subgraph $G[C_i]$, where $C_i$ is the cluster containing $v$. By construction, the cluster diameter is bounded and the update uses the 2-approximation algorithm of Mehlhorn [47], preserving the tree structure and approximation guarantee. The resulting structure within $C_i$ remains connected.

If $v$ also affects inter-cluster connectivity (due to forming part of an inter-cluster path), the spanner $\mathcal{H}$ is updated accordingly. Affected paths are recomputed and integrated into the global structure, maintaining the spanner's $(1+\epsilon)$-approximation guarantee and preserving global connectivity.

**Case 2: $v$ is converted into a non-terminal.** The algorithm examines the role of $v$ in $T$. If $v$ is a leaf in the Steiner tree, it is recursively removed along with its connecting edge, and pruning continues until a non-terminal leaf is reached. This process maintains connectivity and tree structure, as only redundant vertices are removed.

If $v$ is an internal node of $T$, its removal may disconnect the tree. In this case, the Steiner tree is recomputed over the updated terminal set $S_i' = S_i \setminus \{v\}$ within its cluster using the 2-approximation algorithm, preserving intra-cluster connectivity and approximation bounds. Similarly, any affected inter-cluster paths involving $v$ are recomputed via the spanner $\mathcal{H}$, and updated paths are substituted into the global tree.

In both cases, the resulting structure spans all current terminals and remains connected. Since all insertions and deletions are performed via acyclic augmentation and pruning, the maintained structure continues to be a tree. □

**Theorem 5.6** (Correctness of the Algorithm)**.** *The proposed algorithm maintains a Steiner tree spanning all terminal vertices and preserves the tree structure after any dynamic update.*

*Proof.* Follows directly from Lemmas 1–3. □

**Approximation Factor**

**Theorem 5.7** (Approximation Factor)**.** *The updated Steiner tree maintains a* $(2+\epsilon)$*-approximation after any dynamic update.*

*Proof.* The Steiner tree is composed of two components: (1) the intra-cluster Steiner trees, which span terminals within each cluster, and (2) the inter-cluster connections, which connect terminals across clusters via the spanner $\mathcal{H}$.

**Intra-Cluster Approximation:** For any cluster $C_i$, let the set of terminals in $C_i$ be $S_i$. The Steiner tree within $C_i$ is computed using the 2-approximation algorithm of Mehlhorn [47], which guarantees that the weight of the local Steiner tree $T_i$ satisfies:

$$w(T_i) \leq 2 \cdot \text{OPT}_{C_i}$$

where $\text{OPT}_{C_i}$ is the weight of the optimal Steiner tree within $C_i$.

**Inter-Cluster Approximation:** The inter-cluster connections are computed using the spanner $\mathcal{H}$, which approximates the shortest paths between terminals in different clusters. The set of clusters forms a graph $G_{\text{clusters}}$, where each cluster $C_i$ is represented as a node and inter-cluster edges represent the shortest paths between clusters. The optimal inter-cluster Steiner tree $\text{OPT}_{T_{\text{inter}}}$ is equivalent to the minimum spanning tree (MST) over $G_{\text{clusters}}$. For terminals $t \in C_i$ and $t' \in C_j$ (with $i \neq j$), the spanner guarantees:

$$d_{\mathcal{H}}(t, t') \leq (1+\epsilon) \cdot d_G(t, t')$$

Therefore, the weight of the inter-cluster connections $T_{\text{inter}}$ satisfies:

$$w(T_{\text{inter}}) \leq (1+\epsilon) \cdot \text{OPT}_{T_{\text{inter}}}$$

where $\text{OPT}_{T_{\text{inter}}}$ is the weight of the MST over $G_{\text{clusters}}$.

**Overall Approximation:** The total weight of the Steiner tree $T$ is the sum of the weights of the intra-cluster Steiner trees and the inter-cluster connections:

$$w(T) = \sum_i w(T_i) + w(T_{\text{inter}}).$$

Using the bounds for $w(T_i)$ and $w(T_{\text{inter}})$, we have:

$$w(T) \leq \sum_i 2 \cdot \text{OPT}_{C_i} + (1+\epsilon) \cdot \text{OPT}_{T_{\text{inter}}}.$$

The optimal Steiner tree OPT satisfies:

$$\text{OPT} = \sum_i \text{OPT}_{C_i} + \text{OPT}_{T_{\text{inter}}}.$$

Dividing $w(T)$ by OPT, the approximation factor is:

$$\frac{w(T)}{\text{OPT}} \leq \frac{\sum_i 2 \cdot \text{OPT}_{C_i} + (1+\epsilon) \cdot \text{OPT}_{T_{\text{inter}}}}{\sum_i \text{OPT}_{C_i} + \text{OPT}_{T_{\text{inter}}}}$$

To simplify, let $x = \sum_i \text{OPT}_{C_i}$ and $y = \text{OPT}_{T_{\text{inter}}}$. Then:

$$\frac{w(T)}{\text{OPT}} \leq \frac{2x + (1+\epsilon)y}{x+y}.$$

Factorize $2x + (1+\epsilon)y$:

$$\frac{2x + (1+\epsilon)y}{x+y} = \frac{2(x+y) - y + \epsilon y}{x+y}.$$

Split the numerator:

$$\frac{2(x+y) - y + \epsilon y}{x+y} = 2 - \frac{y}{x+y} + \frac{\epsilon y}{x+y}.$$

Combine terms:

$$\frac{2x + (1+\epsilon)y}{x+y} = 2 - \frac{(1-\epsilon)y}{x+y} = 2 - \frac{y}{x+y} + \frac{\epsilon y}{x+y}.$$

$$\frac{w(T)}{\text{OPT}} \leq 2 + \frac{\epsilon y}{x+y}.$$

**Conclusion:** The approximation factor is bounded by $2 + \epsilon'$, where $\epsilon' = \frac{\epsilon y}{x+y} > 0$, since $y > 0$ (the inter-cluster weight is nonzero in any valid Steiner tree). Thus, the Steiner tree maintains a $(2+\epsilon')$-approximation after any dynamic update. As $\epsilon \in (0,1)$, and $\frac{y}{x+y} < 1$, we can conclude that $\epsilon' = \frac{\epsilon y}{x+y} \in (0,1)$. □

### Update Time Complexity

**Theorem 5.8** (Update Time Complexity)**.** *The expected update time complexity for processing any dynamic update is $O(m^{1/2} + n^{2/3})$.*

*Proof.* **Hybrid Distance Oracle Updates:** The hybrid distance oracle is the core structure for maintaining approximate shortest paths during edge and vertex updates. For **edge deletions**, the decremental oracle of Bernstein and Roditty [19] processes updates in $O(m^{1/2})$ amortized time. This complexity arises because the oracle efficiently restricts updates to a subset of affected edges and vertices. For **edge insertions**, the oracle uses a localized Dijkstra-like propagation mechanism, with expected complexity $O(m^{1/2})$. This expectation is derived as follows: edge insertions affect only the neighborhoods of the edge's endpoints, which are proportional to the average vertex degree $d = O(m/n)$. In a balanced graph (where edges are evenly distributed), the expected size of the affected subgraph is $O(m/n)$. Over multiple updates, this results in an amortized $O(m^{1/2})$ complexity. For **vertex deletions and insertions**, updates are localized to the adjacency list of the affected vertex $v$, contributing $O(d(v))$, where $d(v) \leq O(\log n)$ in sparse graphs. Thus, vertex updates have a complexity of $O(\log n)$.

**Spanner Updates:** The spanner $\mathcal{H}$, maintained as a sparse subgraph with $O(n)$ edges, ensures efficient updates for edge and vertex modifications. When an edge is deleted, the paths in $\mathcal{H}$ that relied on this edge are updated. The update process involves identifying the terminal pairs connected by the deleted edge and recomputing the shortest paths between these pairs using the hybrid distance oracle. Since the spanner has $O(n)$ edges and is sparse, the traversal of adjacency lists to locate affected paths is efficient, contributing $O(\log n)$ complexity. Additionally, querying the hybrid distance oracle to recompute paths adds an $O(\log n)$ overhead, making the total cost for handling edge deletions logarithmic in $n$.

For edge insertions, the spanner determines whether to incorporate the newly added edge $e$ into $\mathcal{H}$. This evaluation involves comparing the weight of $e$ with the shortest paths between clusters that $e$ potentially connects. To perform this comparison, the hybrid distance oracle is queried to obtain the current shortest paths between the affected terminals. The sparse structure of $\mathcal{H}$ ensures that only a small subset of edges or paths needs to be evaluated, and the adjacency list traversal and oracle queries together contribute $O(\log n)$ complexity for edge insertions.

When a new vertex $v$ is inserted into the graph, it may introduce new inter-cluster edges. The spanner $\mathcal{H}$ incorporates these edges if they improve

inter-cluster connectivity. This requires identifying the clusters affected by the addition of $v$ and querying the hybrid distance oracle to determine whether the newly introduced edges reduce terminal-to-terminal distances. The sparsity of $\mathcal{H}$ ensures that the number of affected edges and terminal pairs is limited, resulting in $O(\log n)$ complexity for vertex insertions.

The logarithmic complexity of spanner updates arises from two primary factors. First, the sparsity of $\mathcal{H}$ ensures that updates involve only a small subset of the total edges or paths, limiting the scope of operations. Second, the hybrid distance oracle efficiently handles approximate shortest paths, contributing a logarithmic cost for each query. Together, these factors ensure that spanner updates remain efficient and scale logarithmically with the size of the graph.

**Steiner Tree Updates:** The Steiner tree is updated locally within clusters and globally across clusters. For **intra-cluster updates**, the Steiner tree is recomputed using Mehlhorn's 2 approximation algorithm [47]. Mehlhorn's algorithm has a time complexity of $O(m_{C_i} + n_{C_i} \log n_{C_i})$, where $m_{C_i}$ and $n_{C_i}$ refer to the edges and vertices of a cluster $C_i$. Since low-diameter decomposition ensures small cluster sizes, this complexity is efficient in practice. As shown in the analysis of the *optimal choice of* $\Delta$ *in clustering*, $m_{C_i} = O(\Delta^2)$, $n_{C_i} = O(\Delta)$, and $\Delta = O(n^{1/3})$. Hence, the update time complexity for updating the intra-cluster Steiner trees is $O(n^{2/3} + n^{1/3} \log n) = O(n^{2/3})$.

For **inter-cluster updates**, new paths are constructed using the spanner $\mathcal{H}$. Approximate shortest paths are queried from $\mathcal{H}$, contributing $O(\log n)$ complexity for each query. The sparsity of $\mathcal{H}$ ensures that the total cost of inter-cluster updates is manageable.

**Combined Analysis:** The total complexity for processing a single update combines the contributions from hybrid distance oracle updates, spanner updates, and the Steiner tree updates. Hybrid distance oracle updates dominate with $O(m^{1/2})$ complexity for edge modifications, while vertex updates contribute $O(\log n)$. Spanner updates are bounded by $O(\log n)$ due to sparsity, and intra-cluster Steiner tree updates using Mehlhorn's algorithm are bounded by $O(n^{2/3})$. Combining these contributions, the overall expected time complexity for processing any dynamic update is:

$$O(m^{1/2} + n^{2/3}).$$

□

## 5.3 Summary

The existing algorithms for the dynamic Steiner tree problem consider updates in the terminal set, which consists of declaring a terminal as non-terminal or vice versa. These kinds of updates do not change the topology of the graph. It is more challenging to maintain an approximate Steiner tree under topology changes. The algorithms we proposed in Chapter 3 and 4 maintain approximate Steiner trees for edge insertion and edge deletion updates in the graph. The algorithms proposed in this work are the first of their kind, as they handle multiple kinds of updates altogether, and hence, we can not directly compare them with existing results in the dynamic Steiner tree problem.

The presented algorithm maintains a $(|S|-1)-$approximate Steiner tree in $\tilde{O}((\delta p \log U)/\epsilon + |S| \cdot \sqrt{n} + n)$ update time for dynamic planar graphs. The proposed algorithm supports six types of updates: insertion and deletion of edges in the graph, insertion and deletion of vertices in the graph, and conversion of terminals to non-terminals and vice versa. We also proved that any spanning tree of the complete distance graph on the terminal set $S$ gives a $(|S|-1)$-approximate Steiner tree on the terminal set $S$. The update time of the proposed algorithm becomes linear in $n$ in a case where $|S| = o(\sqrt{n})$.

We propose a fully dynamic algorithm for approximate Steiner trees in general graphs handling the above mentioned six types of updates. We augmented a decremental distance oracle to make it fully dynamic. By combining dynamic clustering, spanners, and hybrid distance oracles, the algorithm achieves a tunable approximation factor of $2+\epsilon'$ in an efficient update time of $O(m^{1/2}+n^{2/3})$. This is a significant improvement over our previous algorithms as it offers a wide range of update operations and better update time and approximation factor.

The dynamic Steiner tree problem has not gotten much attention in general dynamic graphs. The existing algorithms focus on updates in the terminal set. We wish to extend the algorithm proposed in Section 5.1 to maintain an approximate Steiner tree with a better approximation factor or better time complexity for general graphs. To maintain the approximate MST of the graph, any other algorithm with better update time can be used. The presented algorithm works on planar graphs because of the distance oracle used. The algorithm can be extended to general graphs by using or designing a dynamic distance oracle that works efficiently in general graphs. It is an interesting direction to improve the update time complexity of the algorithm given in Section 5.2 by playing with the size of clusters and techniques to maintain the clusters and spanners.

# 6

# Conclusions and Future Work

## 6.1 Conclusions

In this thesis, we propose several algorithms for maintaining an approximate Steiner tree under various dynamic graph models.

In the first contribution, we present a fully dynamic algorithm to maintain a $(2+\epsilon)$-approximate Steiner tree in planar graphs. The worst case update time complexity of the presented algorithm is $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)$. The Steiner tree is updated after a sequence $U$ of updates where $1 \leq |U| \leq \sqrt{n+m}$. Hence, for $k$ number of updates, the average update time of the algorithm is $\tilde{O}(|S|^2\sqrt{n} + |S|D + n)/k$ where $1 \leq k \leq \sqrt{n+m}$. The proposed algorithm works nicely in dynamic scenarios with edge updates. Its update time scales more favorably, particularly with a moderate number of terminals $(|S|)$. The update time complexity of the proposed algorithm becomes more favorable when the number of terminals $|S|$ and the unweighted diameter $D$ are both logarithmic in the number of vertices $(O(\log n))$. In this case, the worst-case update time reduces to $O(n^{0.5+\epsilon''}(\epsilon')^{-2}) + O(n)$. This is a significant improvement compared to the previously proposed $PTAS$ by Borradaile et al. [30]. Their approach has a time complexity of $O(2^{poly(1/\epsilon)}n + n\log n)$; moreover, it is not a dynamic algorithm.

In the second contribution, we proposed an incremental algorithm that offers an efficient solution for maintaining a $(2-\epsilon)$-approximate Steiner tree in general weighted graphs that undergo edge insertions. The algorithm achieves this by utilizing a shortest path forest and efficiently processing each update in the worst case scenario with an update time complexity of $O(nD_s)$. This significantly outperforms the approach by Kou et al. [14], which requires recomputing the entire $(2-\epsilon)$-approximate Steiner tree from scratch for each

update.

In the third contribution, we demonstrate a lower bound of $\Omega(n)$ on the update time for maintaining an MST heuristic based $(2-\epsilon')$-approximate Steiner tree in general dynamic graphs under both edge insertion and edge deletion updates.

In the fourth contribution, we present a decremental Steiner tree algorithm that maintains a $(2+\epsilon)$-approximate Steiner tree with edge deletions in weighted planar graphs in $\tilde{O}(\ell\sqrt{n})$ worst case update time. This is significantly better than computing a $(2-\epsilon')$-approximate Steiner tree from scratch as given by Kou et al.(requires $O(|S|n^2)$ time) and Wu et al.(requires $O(m\log n)$ time). Assuming only deletion updates, this algorithm is also better than our fully dynamic algorithm presented in Chapter 3 (the first contribution) in terms of the update time.

Chapter 5 contains the fifth contribution where we present a fully dynamic algorithm for maintaining a $(|S|-1)$-approximate Steiner tree in planar dynamic graphs with a worst case update time of $\tilde{O}((\delta p\log U)/\epsilon+|S|\cdot\sqrt{n}+n)$. Notably, this algorithm supports a comprehensive set of six update operations: insertion and deletion of edges, insertion and deletion of vertices, and conversion of nodes from terminal to non-terminal and vice versa. These types of updates significantly alter the structure of the graph, making the problem substantially more challenging than terminal-only updates as considered by Lacki et al. [8] and Imase et al. [7]. Our proposal is the first to address this full spectrum of dynamic operations. Furthermore, we prove that any spanning tree of the complete distance graph on the terminal set $S$ yields a $(|S|-1)$-approximate Steiner tree. The update time of our algorithm becomes linear in $n$ under the restriction $|S|=o(\sqrt{n})$, making it attractive for graphs with a small number of terminals.

Finally, in the last contribution, we present a fully dynamic algorithm maintaining a $(2+\epsilon')$-approximate Steiner tree in general graphs that can manage the six kinds of updates as mentioned earlier. This result is an improvement over the previous algorithm in terms of approximation factor and the graph model. This requires incorporating a carefully augmented *decremental distance oracle* into a fully dynamic framework. The proposed approach combines various techniques like *clustering*, *spanner construction*, and *hybrid distance oracle* to achieve a tunable approximation guarantee (in terms of $\epsilon'$) and an update time of $O(m^{1/2}+n^{2/3})$. This result marks a substantial leap forward in the design of dynamic approximation algorithms for the Steiner tree problem, especially considering the generality of the graph class, the number and types of update operations supported, and the low update time complexity.

Table 6.1: Results in the dynamic Steiner tree problem

| **Work** | $\beta$ | **Update time** | **Preprocessing time** | **model of update** | **Model of graph** |
|---|---|---|---|---|---|
| Imase and Waxman Algorithm 1 (DST non-rearrangeable) | $P.R. \leq \lceil \log(n_i) \rceil$ | not mentioned | NA | terminal conversion | general graph |
| Imase and Waxman Algorithm 2 (DST rearrangeable) | $(4\delta)$, $\delta \geq 2$ | at most $\frac{1}{2}K_a(\sqrt{4K_a - 3} - 3) + K_r$ number of changes, $K_a$ and $K_r$ are the number of add and remove requests | NA | terminal conversion | general graph |
| Lacki et al. Algorithm 1 | $(2+\epsilon)$ | $\tilde{O}(\sqrt{n} \log D)$ | $\tilde{O}(n \log D)$ | terminal conversion | planar graph |
| Lacki et al. Algorithm 2 | $(4+\epsilon)$ | $\tilde{O}(\epsilon^{-1} \log^6 n)$ amortized update time | $O(\epsilon^{-1} n \log^2 n)$ | terminal conversion | planar graph |
| Lacki et al. Algorithm 3 | $(6+\epsilon)$ | $\tilde{O}(\sqrt{n} \log D)$ | $\tilde{O}(\sqrt{n}(m + n \log D))$ | terminal conversion | general graph |
| Raikwar and Karmakar Algorithm 1 | $(2+\epsilon)$ | $\tilde{O}(\lvert S\rvert^2 \sqrt{n} + \lvert S\rvert D + n)$ | $O(\epsilon^{-1} n \log^2 n)$ | edge insertion and deletion | planar graph |
| | | Avg update time= $(\lvert S\rvert^2 \sqrt{n} + \lvert S\rvert D + n)/k$ : $1 \leq k \leq \sqrt{n+m}$ for $k$ number of updates | | | |
| | | Special case: $O(n^{0.5+\epsilon''}(\epsilon')^{-2}) + O(n)$ when $\lvert S\rvert, D = O(\log n)$ | | | |
| Raikwar and Karmakar Algorithm 2 | $(2-\epsilon)$ | $O(nD_s)$ | $O(\lvert E\rvert \log \lvert V\rvert)$ | edge insertion | general graph |
| Raikwar, Siglipalli, and Karmakar Algorithm 3 | $(2+\epsilon)$ | $\tilde{O}(\ell \sqrt{n})$ | $O(m + \epsilon^{-1} n \log^2 n)$ | edge deletions | planar graph |
| Raikwar, Patel, and Karmakar Algorithm 4 | $(\lvert S\rvert - 1)$ | $\tilde{O}((\delta p \log U)/\epsilon + \lvert S\rvert \cdot \sqrt{n} + n)$ | $O(m)$ | edge insertion and deletion, vertex insertion and deletion, terminal conversion | planar graph |
| | | $O(n)$ when $\lvert S\rvert = o(\sqrt{n})$ | | | |
| Raikwar and Karmakar Algorithm 5 | $(2+\epsilon)$ | $O(m^{1/2} + n^{2/3})$ | $O(kmn^{1/k})$, $k \geq 2$ | edge insertion and deletion, vertex insertion and deletion, terminal conversion | general graph |

$\beta$: Approximation factor; $P.R.$: Performance ratio (competitive ratio); $D$: Stretch of the matrix induced by the graph under consideration

Byrka et al. [5] present an algorithm that computes a 1.39-approximate Steiner tree using LP based primal-dual approach. We did not find any result mentioning hardness to maintain a better than $2 + \epsilon$ or $2 - \epsilon$ approximation. However, we found it difficult to build a dynamic algorithm based on the complex LP techniques used by Byrka. Further, the time complexity of their algorithm is $O(n^5)$, which makes it difficult to use it in designing efficient dynamic algorithms.

## 6.2 Future Work

An exciting and challenging direction for future research lies in further improving the update time complexity of our fully dynamic algorithm for the Steiner tree problem in planar graphs handling edge insertions and deletions (Chapter 3). Given the inherent structural complexity of dynamically maintaining near-optimal connectivity under arbitrary updates, any advancement in this area would represent a significant theoretical breakthrough and contribute greatly to the broader landscape of dynamic graph algorithms.

It is also an interesting direction to improve the update time complexity of the algorithm given in Section 5.2 by playing with the size of clusters and techniques to maintain the clusters and spanners.

While this thesis has addressed several challenges in the dynamic maintenance of approximate Steiner trees, particularly in centralized settings, the problem remains largely unexplored in *distributed computing models*, where information and updates are scattered across nodes in a network. In dynamic distributed environments, such as sensor networks, ad-hoc systems, and data center routing, nodes must coordinate locally to adapt to frequent edge insertions and deletions without centralized control. To date, there exists **no efficient distributed dynamic algorithm** for maintaining a constant approximation of Steiner trees under a distributed setting.

Future work can explore the design of *distributed fully dynamic algorithms* that maintain low-cost Steiner trees having a constant approximation factor while preserving **locality**, **scalability**, and **low communication overhead**. Interesting directions include developing algorithms in the CONGEST or LOCAL models, analyzing trade-offs between *update time*, *approximation quality*, and *message complexity*, and extending techniques like clustering, spanners, or local reconstructions to distributed settings. Addressing these challenges could significantly impact applications in *resilient communication*, *network optimization*, and *distributed infrastructure planning*, where dynamic connectivity must be preserved with minimal global coordination.

The *Prize-Collecting Steiner Tree (PCST)* problem has been extensively

studied in a static setting, leading to several elegant approximation algorithms with provable guarantees. However designing a PCST algorithm for *dynamic environments* where the underlying graph, prizes, and penalties may change over time due to edge insertions or deletions remains a challenge. Currently there exists no dynamic algorithm that efficiently maintains a good approximation of *PCST* under such updates. Future work may explore the adaptation of our dynamic Steiner tree techniques and primal-dual frameworks, dynamic spanner constructions, or incremental LP-based techniques to this setting.

# Bibliography

[1] Ivana Ljubić. Solving Steiner trees: Recent advances, challenges, and perspectives. *Networks*, 77(2):177–204, 2021.

[2] Miroslav Chlebík and Janka Chlebíková. The Steiner tree problem on graphs: Inapproximability results. *Theoretical Computer Science*, 406(3):207 – 214, 2008.

[3] Richard M. Karp. Reducibility among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[4] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[5] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. An Improved LP-Based Approximation for Steiner Tree. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC '10, page 583–592, New York, NY, USA, 2010.

[6] Frank Harary and Gopal Gupta. Dynamic graph models. *Mathematical and Computer Modelling*, 25(7):79–87, 1997.

[7] Makoto Imase and Bernard M. Waxman. Dynamic Steiner Tree Problem. *SIAM Journal on Discrete Mathematics*, 4(3):369–384, 1991.

[8] Jakub Lacki, Jakub Oćwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych. The Power of Dynamic Distance Oracles: Efficient Dynamic Algorithms for the Steiner Tree. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 11–20, New York, NY, USA, 2015.

[9] Ehud Aharoni and Reuven Cohen. Restricted dynamic Steiner trees for scalable multicast in datagram networks. *IEEE/ACM Transactions on Networking*, 6(3):286–297, 1998.

[10] Shan Ding and Naohiro Ishii. An online genetic algorithm for dynamic Steiner tree problem. In *Proceedings of the 26th Annual Conference of the IEEE Industrial Electronics Society*, volume 2 of *IECON '00*, pages 812–817, Nagoya, Japan, 2000.

[11] Christoph Lenzen and Boaz Patt-Shamir. Improved Distributed Steiner Forest Construction. *CoRR*, abs/1405.2011, 2014.

[12] Lélia Blin, Maria Potop-Butucaru, and Stephane Rovedakis. A super-stabilizing $\log(n)$-approximation algorithm for dynamic Steiner trees. *Theoretical Computer Science*, 500:90–112, 2013.

[13] Anupam Gupta and Amit Kumar. Online Steiner Tree with Deletions. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms,*, SODA '14, pages 455–467, Portland, Oregon, USA, 2014.

[14] Lawrence T. Kou, George Markowsky, and Leonard C. Berman. A Fast Algorithm for Steiner Trees. *Acta Informatica*, 15:141–145, 1981.

[15] Robert B Dial. Algorithm 360: Shortest-path forest with topological ordering [H]. *Communications of the ACM*, 12(11):632–633, 1969.

[16] Ying Fung Wu, Peter Widmayer, and Chak-Kuen Wong. A faster approximation algorithm for the Steiner problem in graphs. *Acta informatica*, 23(2):223–229, 1986.

[17] Monika R. Henzinger and Valerie King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation. *Journal of the ACM*, 46(4):502–516, 1999.

[18] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[19] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, page 1355–1365, San Francisco, California, USA, 2011.

[20] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent Advances in Fully Dynamic Graph Algorithms. In *1st Symposium on Algorithmic Foundations of Dynamic Networks*, volume 221 of *SAND '22*, pages 1:1–1:47, Dagstuhl, Germany, 2022.

[21] 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*, 48(4):723–760, 2001.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, USA, 1990.

[23] Pawel Winter. Steiner problem in networks: A survey. *Networks*, 17(2):129–167, 1987.

[24] Zbigniew A. Melzak. On the Problem of Steiner. *Canadian Mathematical Bulletin*, 4(2):143–148, 1961.

[25] Edward N. Gilbert and Henry O. Pollak. Steiner Minimal Trees. *SIAM Journal on Applied Mathematics*, 16(1):1–29, 1968.

[26] Shi-Kuo Chang. The Generation of Minimal Trees with a Steiner Topology. *Journal of the ACM*, 19(4):699–711, 1972.

[27] James M. Smith, Der-Tsai Lee, and Judith S. Liebman. An $O(n \log n)$ heuristic for Steiner minimal tree problems on the euclidean metric. *Networks*, 11(1):23–39, 1981.

[28] Gabriel Robins and Alexander Zelikovsky. Improved Steiner tree approximation in graphs. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, page 770–779, San Francisco, California, USA, 2000. Society for Industrial and Applied Mathematics.

[29] Markus Chimani, Petra Mutzel, and Bernd Zey. Improved steiner tree algorithms for bounded treewidth. In *Combinatorial Algorithms*, pages 374–386, Berlin, Heidelberg, 2011.

[30] Glencora Borradaile, Philip Klein, and Claire Mathieu. An O $(n \log n)$ approximation scheme for Steiner tree in planar graphs. *ACM Transactions on Algorithms*, 5(3):1–31, 2009.

[31] Frank K. Hwang. An $O(n \log n)$ Algorithm for Rectilinear Minimal Spanning Trees. *Journal of the ACM*, 26(2):177–182, 1979.

[32] Alexander Z. Zelikovsky. An 11/8-approximation algorithm for the Steiner problem on networks with rectilinear distance. In *Colloquia Mathematica Societatis János Bolyai, Conference on Sets, Graphs and Numbers*, volume 60, pages 733–745. Amsterdam, Netherlands, 1992.

[33] Ambrogio Maria Bernardelli, Fabrizio D'Andreagiovanni, and Paolo Detti. On the Integrality Gap of the Complete Metric Steiner Tree Problem via a Novel Formulation. *Optimization Online*, 2024.

[34] Ali Ahmadi, Iman Gholami, MohammadTaghi Hajiaghayi, Peyman Jabbarzade, and Mohammad Mahdavi. Prize-Collecting Steiner Tree: A 1.79 Approximation. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, STOC '24, page 1641–1652, Vancouver, BC, Canada, 2024.

[35] Narek Bojikian and Stefan Kratsch. A tight Monte-Carlo algorithm for Steiner Tree parameterized by clique-width. *arXiv preprint*, 2024.

[36] Bart M. P. Jansen and Céline M. F. Swennenhuis. Steiner Tree Parameterized by Multiway Cut and Even Less. *arXiv preprint*, 2024.

[37] Mikkel Thorup and Uri Zwick. Approximate Distance Oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[38] Stefan Balev, Yoann Pigné, Éric Sanlaville, and Mathilde Vernet. Brief Announcement: The Dynamic Steiner Tree Problem: Definitions, Complexity, Algorithms. In *3rd Symposium on Algorithmic Foundations of Dynamic Networks*, volume 292 of *SAND '24*, pages 24:1–24:6, Dagstuhl, Germany, 2024.

[39] T-H. Hubert Chan, Gramoz Goranci, Shaofeng H. C. Jiang, Bo Wang, and Quan Xue. Fully Dynamic Algorithms for Euclidean Steiner Tree. *arXiv preprint*, 2023.

[40] Parikshit Saikia and Sushanta Karmakar. A simple $2(1 - 1/\ell)$ factor distributed approximation algorithm for Steiner tree in the CONGEST model. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, ICDCN '19, page 41–50, Bangalore, India, 2019.

[41] Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.

[42] Parikshit Saikia and Sushanta Karmakar. Improved distributed approximation for Steiner tree in the CONGEST model. *Journal of Parallel and Distributed Computing*, 158:196–212, 2021.

[43] Parikshit Saikia and Sushanta Karmakar. $2(1-1/\ell)$-Factor Steiner Tree Approximation in $\tilde{O}(n^{1/3})$ Rounds in the CONGESTED CLIQUE. In *Seventh International Symposium on Computing and Networking*, CANDAR '19, pages 82–91, Nagasaki, Japan, 2019.

[44] Parikshit Saikia and Sushanta Karmakar. Round-Message Trade-Off in Distributed Steiner Tree Construction in the CONGEST Model. In *Sixteenth International Conference on Distributed Computing and Internet Technology*, ICDCIT '20, pages 111–126, Cham, 2020.

[45] Parikshit Saikia and Sushanta Karmakar. Distributed approximation algorithms for Steiner tree in the CONGESTED CLIQUE. *International Journal of Foundations of Computer Science*, 31(5):605–627, 2020.

[46] Parikshit Saikia, Sushanta Karmakar, and Aris Pagourtzis. Primal-dual based distributed approximation algorithm for Prize-collecting Steiner tree. *Discrete Mathematics, Algorithms and Applications*, 13(02):2150008, 2021.

[47] Kurt Mehlhorn. A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Information Processing Letters*, 27(3):125–128, 1988.

[48] Ittai Abraham, Shiri Chechik, and Cyril Gavoille. Fully Dynamic Approximate Distance Oracles for Planar Graphs via Forbidden-Set Distance Labels. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 1199–1218, New York, New York, USA, 2012.

[49] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[50] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

[51] Edward F Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[52] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 114–122, Milwaukee, Wisconsin, USA, 1981.

[53] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[54] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *Journal of Computer and System Sciences*, 69(3):485–497, 2004.

[55] Ittai Abraham, Cyril Gavoille, Dahlia Malkhi, and Michael Thorup. Compact Routing on Dynamic Trees. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 17–26, Toronto, Canada, 2008.

# Disseminations from the Thesis

## Journals

- Hemraj Raikwar, Harshil Sadharakiya, and Sushanta Karmakar. Dynamic algorithms for approximate Steiner trees. *Concurrency and Computation: Practice and Experience*, 37(6-8):e70040, 2025.

## Conferences

- Hemraj Raikwar and Sushanta Karmakar. Fully Dynamic Algorithm for Steiner Tree Using Dynamic Distance Oracle. In *23rd International Conference on Distributed Computing and Networking*, ICDCN 2022, page 246–247. IEEE, Association for Computing Machinery, Delhi, India, 2022.

- Hemraj Raikwar and Sushanta Karmakar. Fully dynamic algorithm for the Steiner tree problem in planar graphs. In *Tenth International Symposium on Computing and Networking Workshops*, CANDARW 2022, pages 416–420. IEEE, Himeji, Japan 2022.

- Hemraj Raikwar and Sushanta Karmakar. An incremental algorithm for $(2-\epsilon)$-approximate Steiner tree requiring $o(n)$ update time. In *Eleventh International Symposium on Computing and Networking*, CANDAR 2023, pages 168-174. IEEE, Matsue, Japan, 2023: **Recognized as Outstanding Paper Award**.

## Under Preparation

- Hemraj Raikwar, Apuroop Sigilipalli, Harshil Sadharakiya, and Sushanta Karmakar. A Decremental Algorithm for $(2+\epsilon)$-Approximate Steiner Tree in Planar Graphs- **preparing to submit to *International Journal of Foundations of Computer Science (IJFCS)***.

- Hemraj Raikwar, Miki Patel, and Sushanta Karmakar. Fully dynamic Steiner tree algorithms handling six types of updates- **submitted to *Theoretical Computer Science(TCS)***.