Applications of top trees on Planar Graphs

A Top tree like structure for planar graphs and solution of nearest marked vertex problem on planar graphs

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

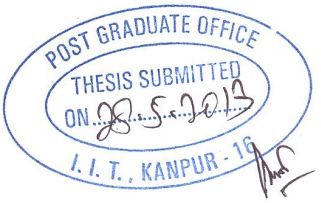
Master of Technology

by
Shahbaz Khan
Roll No. : 11111037

under the supervision of
Dr. Surender Baswana

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
June, 2013

CERTIFICATE



It is certified that the work contained in this thesis entitled
"*Applications of top trees on Planar Graphs, A Top tree like structure for planar graphs and solution of nearest marked vertex problem on planar graphs.*",
by *Shahbaz Khan (Roll No. 11111037)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Surender Baswana

(Dr. Surender Baswana)
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur
Kanpur-208016

May, 2013

Abstract

Top tree is a powerful data structure for maintaining dynamic trees. It can solve problems on trees dealing with properties based on vertices, edges and paths. It solves a variety of problems for trees including aggregating information, nearest marked vertex, maintaining center and diameter, subject to dynamic updates in the tree as edge insertion and deletion. We present two results, each of which is an application of top trees on planar graphs.

In the first result we make a structure similar to top trees for planar graphs having n vertices, that can solve the problems dealing with properties based on vertices, edges and faces of a planar graph. It allows the user to expose the graph on a cycle or cut such that the data structure represents the subgraph enclosed by the cycle or cut. This structure is then used to solve two problems. The first application is aggregating information (Minimum, maximum or sum) on elements(vertices, edges or faces) of planar graph, with online queries and additive weight updates on the elements of a subgraph, enclosed by a cycle or cut on the graph. The second application is maintaining dynamic Minimum Spanning Tree of a planar graph, which allows online updates on edge weights and query of MST on a subgraph enclosed by a cut on the graph. Trivially both the applications takes $O(n)$ time to be solved, whereas using our structure it can be solved in $O(d\log n)$ time where d is the length of cycle or cut mentioned above.

The second result deals with nearest marked vertex problem for planar graphs. Marked ancestor problem for trees is an extensively studied problem and finds application is solving many other theoretical problems. Nearest Marked Vertex problem is a variant of Shortest Path problem which deals with dynamically marking/unmarking any vertex and querying the nearest marked vertex of any given vertex in an online fashion. The problem can be extended to find the k nearest marked vertices for a given vertex. We present a solution to solve the problem in $O(\sqrt{n}\log n)$ time for both update and query. There is currently no efficient solution for the problem in planar graphs, for general graphs similar bounds are used to provide a 3 approximate solution for the problem.

Dedicated to
my alma mater and my family.

Acknowledgement

First and foremost I would like to express my sincere gratitude towards my thesis supervisor Dr. Surender Baswana for his constant support and encouragement. I am indebted to him for his patient guidance and constant motivation, giving a proper direction to my efforts despite various failures that I encountered during the completion of my thesis. I am honoured to have got a chance to work with such an enthusiastic, hard working and supportive person. I would also like to thank Prof. Somenath Biswas and Dr. Satyadev Nandakumar with whom I worked for first six months, which have been a great learning experience for me, especially in terms of various aspects of research studies. I would like to thank all the faculty and staff of the Department of CSE for teaching and supporting me during my entire stay at IIT Kanpur.

I would like to especially thank Mr. Tejas Gandhi(PhD) and Mr. Saiful Islam(PhD) for being a mentor to me, guiding me in every aspect of my life at IIT Kanpur. I am indebted to all my friends of the Mtech 2011 batch and PhD, for making past two years of my life such a memorable experience. I would like to especially thank Ajay, Ashu, Ankit, Atanu, Akhil, Abhimanyu, Hrishit, Mangat, Nitesh, Parveen, Rizwan and Vikram for the great times that we shared together.

Lastly , I would like to thank my alma mater, my family and the Almighty God for making me what I am today, that enabled me to complete this thesis work.

Shahbaz Khan

Contents

Abstract ii

Acknowledgement iv

List of Figures ix

List of Algorithms x

1 Introduction 1

1.1 Problems Solved 1

1.1.1 Top tree for Planar Graphs 1

1.1.2 Nearest Marked Vertex in Planar Graphs 2

1.2 Organisation of thesis 3

2 Background and Related Work 4

2.1 Dynamic Graph Algorithms 4

2.2 Planar Graphs 4

2.2.1 Sparsity Property 5

2.2.2 Planarity of Dual graph 5

2.2.3 Existence of Interdigitating Trees 6

2.3 Top Trees 6

2.3.1 Structure of top tree 7

2.3.2 Interface of top tree 9

2.3.3 Applications 10

2.3.4 Nearest Marked Vertex 11

2.4 Multiple Source Shortest Path in Planar Graphs 12

3 Top tree for Planar Graphs **15**

3.1 Introduction . . . 15

3.2 Overview . . . 17

3.3 Exposing a subgraph in primal and dual graphs . . . 17

3.4 Top Trees Structure for Planar Graphs . . . 18

3.4.1 External Operations on the Graph . . . 19

3.4.2 Internal Operations . . . 20

3.5 Implementation details . . . 20

3.5.1 Expose(Cycle/Cut) and UnExpose: . . . 20

3.5.2 Update(Element): . . . 22

3.5.3 Update(Cluster) and Query: . . . 22

3.6 Analysis . . . 22

3.7 Aggregating Information on Planar Graphs . . . 24

3.7.1 Minimum Weighted Vertex on subgraph . . . 25

3.7.2 Handling other operations . . . 26

3.7.3 Handling faces and edges . . . 27

3.8 Minimum Spanning Tree on a subgraph of planar graph . . . 28

3.8.1 Modified Expose Operation . . . 29

3.8.2 Modified Unexpose Operation . . . 30

3.8.3 Updating edge weight . . . 31

3.8.4 Analysis . . . 32

4 Nearest Marked Vertex in Planar Graph **34**

4.1 Introduction . . . 34

4.2 Overview . . . 35

4.3 Bounded Edge changes property . . . 36

4.4 Nearest marked vertex property . . . 37

4.5 Initial approaches . . . 38

4.6 Nearest Marked Vertex in Planar Graphs . . . 39

4.6.1 Algorithm . . . 39

4.6.2 Analysis . . . 42

4.7 k Nearest Marked Vertices in Planar Graphs . . . 43

5 Conclusions **45**

5.1 Impact 45

5.2 Limitations 46

5.3 Future Work 46

List of Figures

2.1 The figure shows the primal and dual graphs. (a) Shows the primal graph, (b) Shows the dual graph made using faces of primal graph as vertices, (c) Shows the dual graph. 5

2.2 The figure shows the interdigitating trees. (a)Shows the primal graph (using black edges) and corresponding dual graph (using dotted blue edges). (b)Shows the spanning tree on primal graph (using bold red edges) and that of dual graph (using complete blue edges). 6

2.3 The figure shows the structure of top trees. (a) Shows the representation of tree on each level, (b) Shows the top tree made showing base clusters (trapezium), rake(rectangle) and compress (circle) clusters. . 8

2.4 Tree contraction moves showing *Compress* and *Rake* on left and right. 9

2.5 Shortest path from v_1 and v_3 to y has arc xy shown in blue, then shortest path of atleast one of v_2 or v_4 should pass through them shown in red, ... 13

3.1 The figure shows how the expose operation changes divides a planar graph (a) Shows a cycle on the primal graphs, (b) Shows the dual subgraphs enclosed by the cycle of primal graph, (c) Shows a cycle on dual graph (representing a cut on primal graph), (d) Shows the primal subgraphs enclosed by the cut on the primal graph. 18

3.2 The figure shows how the expose operation changes the spanning tree on primal and dual graphs. (a) Shows the primal graphs , (b) Shows the spanning tree of primal and dual graph,indicating a cycle C by dashed lines, (c) Shows the new spanning trees after calling expose on C such that only one edge of C intersects the spanning tree on dual graph. 21

List of Algorithms

3.1 Exposing a cycle C on the dual graph G^* 23

3.2 Unexposing D_{tt} earlier exposed on a cut C^* on the primal graph G . 31

3.3 Updating the weight of an edge e from $wt(e)$ to $wt^*(e)$. 32

4.1 Making the data structure D_{nmv} 41

4.2 Query a vertex v_x on f_∞ 42

4.3 Query a vertex v_x on f_∞ for k nearest marked vertices 44

Chapter 1

Introduction

Top tree is a powerful data structure for maintaining dynamic trees. It can solve problems on trees dealing with properties based on vertices, edges and paths. It solves a variety of problems for trees including aggregating information, nearest marked vertex, maintaining center and diameter, subject to dynamic updates in the tree as edge insertion and deletion.

In the first problem we have tried to develop a structure similar to top trees for planar graphs. It solve two problems namely aggregating information of elements of a planar graph and maintaining dynamic MST for subgraphs of a planar graph. Second problem deals with reporting nearest marked vertex for any vertex of a planar graph where updates and queries are performed in an online fashion.

1.1 Problems Solved

1.1.1 Top tree for Planar Graphs

The main aim was to develop a data structure similar to Top trees, for planar graphs that addresses problems related to properties of edges, vertices and faces. Our solution introduces a data structure D_{tt} similar to top trees for planar graphs. Technically D_{tt} is easily implementable using top trees, the main contribution of D_{tt} is the *design* of the $interface$, providing user an easier access to advanced techniques using top trees to solve problems on planar graphs. We use the phrase *subgraph enclosed* by a *cycle or cut*, which refers the subgraph of the planar graph that is either internal or external to the cycle (or cut). We introduce the notion of *exposing* the planar graph on a subgraph enclosed by a cycle or cut. On an exposed D_{tt} the data will only account for the subgraph that is internal or external to the cycle(or cut) in the primal graph.

For a given embedded planar graph G having n vertices, we make a data structure D_{tt} in $O(n)$ time and using $O(n)$ space. It can update the weights of an element(vertex,edge or face) in $O(\log n)$ time. The *Access* and *Query* operations

are performed in $O(1)$ time. *Expose* operation called on a subgraph, enclosed by a cycle or cut of size d is performed in $O(d\log n)$ time. Since the size of input for *Expose* is d, we cannot expect to perform better than $\Omega(d)$. Also expose operation in top trees require $O(\log n)$ time per boundary vertex to perform expose on a given cluster path. *Expose* operation of D_{tt} also uses $O(\log n)$ time per edge on the cycle or cut to perform expose justifying the bounds achieved in the implementation of D_{tt}.

The first problem solved using D_{tt} is aggregating information (Minimum, maximum or sum) on elements(vertices, edges or faces) of planar graph G having n vertices, with queries and additive weight updates on the elements of a subgraph, enclosed by a cycle (or cut) including or excluding the external cycle(or cut). The queries and updates are performed in an online fashion. The solution takes $O(d\log n)$ time for performing expose and $O(\log n)$ time to perform query and update operations where d is the length of the input cycle(or cut). Trivial solutions for the same problem uses $O(n)$ time per update or query operation.

The second problem solved using D_{tt} is maintaining dynamic Minimum Spanning Tree(referred as MST) of a planar graph G having n vertices, which allows online updates on edge weights and query of MST of a subgraph enclosed by a cut on G. The solution takes $O(d\log n)$ time for performing query of MST on a subgraph and $O(\log n)$ time to perform query and update operations where d is the length of the input cut. Trivial solutions for the same problem uses $O(n\log n)$ time per update or query operation.

1.1.2 Nearest Marked Vertex in Planar Graphs

Nearest Marked Vertex problem is a variant of Shortest Path problem which deals with dynamically marking/unmarking any vertex and querying the nearest marked vertex of any given vertex in an online fashion. The problem defines two *update* operations namely mark and unmark, and a *query* operation for querying nearest marked vertex.

Marked ancestor problem for trees is an extensively studied problem and finds application in solving many other theoretical problems[5]. The problem is then extended to Nearest marked vertex problem in trees and solved by Alstrup et. al [1] using top trees. It supports dynamic link and cuts, performing all operations in $O(\log n)$ time. They also gave a solution for Nearest marked for a fixed undirected graph on n vertices and m edges. They present a $2k-1$ approximate solution using $O(kn^{1+1/k})$ space data structure built in $O(kmn^{1/k}\log n)$ expected time for any positive integer parameter k, supporting both queries and updates in $O(kn^{1/k}\log n)$ time.

Given an embedded planar graph G having n vertices with infinite face f_∞ having d vertices, our aim is to be able to dynamically *mark* and *unmark* vertices such

that for any vertex $v \in f_\infty$ we can *query* the nearest marked vertex in an online fashion. The problem can be extended to find the k nearest marked vertices for a given vertex. We define a parameter $S \in [1, d]$, such that our algorithm performs the update operations in $O(S\log n)$ time and query in $O(n\log n/S)$ time using $O(nS)$ space. Also it supports k queries in $O((n/S+k)\log n)$ time. The balanced solution in terms of equal bounds for updates and queries is obtained taking S as $\sqrt{n}$ resulting in both updates and query to be done in $O(\sqrt{n}\log n)$ time using $O(n\sqrt{n})$ space. Using same bound the solution by Alstrup et. al[1] for general graphs, gives 3 approximation of the solution. Also our initial approaches to solve the problem on planar graphs performs either update or query in $O(1)$ time, taking atleast $O(d\log n)$ time to solve the other operation. We can match bounds of both these solutions at the extreme values of our parameter S. Hence the bounds achieved for our final solution are competitive with the state of the art and direct approaches to solve the problem.

1.2 Organisation of thesis

In Chapter 2, we discuss about the various topics required to develop a better understanding of the problems solved in the thesis. We describe Planar graphs and its important properties that were used to formulate the solutions of the aforementioned problems. Also the interface and applications of top trees are discussed to give an insight of how top trees are used to solve problems. Further some important results related to the MSSP algorithm by Klien[3] are discussed.

In Chapter 3, we describe the solution of the first problem. We describe a top tree like structure for planar graphs, along with its structure, implementation and application for aggregating information on planar graph. We also state its limitation and are areas of future work.

In Chapter 4, we describe a solution for nearest marked vertex problem in planar graphs using top trees. We describe a data structure to support the related queries and updates in an online fashion. We extend the solution to find k nearest marked vertices of a vertex on a planar graph. We further describe its limitations and scope for future work.

Chapter 2

Background and Related Work

2.1 Dynamic Graph Algorithms

Dynamic graph problems deals with a sequence of queries and updates on a given graph in an online fashion. The updates can be in the form of edge insertions and deletions. Each query has to be answered based on current state of the graph. Trivial approach can be to recompute the solution from scratch using an algorithm for the static version of the problem, after every update on the tree. Dynamic algorithms answer such queries efficiently. Based on the types of updates handled the dynamic graph algorithms are classified as:

1. *Incremental Algorithms:* These support only edge insertions.
2. *Decremental Algorithms:* These support only edge deletions.
3. *Fully dynamic Algorithms:* These support both edge insertions and deletions.

2.2 Planar Graphs

Graphs for which a planar embedding exists, i.e. it can be embedded such that no two edges cross each other, are called Planar Graphs. They have some interesting properties that are exploited to make better solutions for planar graphs. For example, considering the problem of Single Source Shortest paths in graphs having negative edge weights. The best known algorithm for general graph is Bellman Ford algorithm, that takes $O(mn)$ time. However, for planar graphs the best known algorithm is by Klien[18] which takes $O(n\log^2 n)$ time. Some of these properties are exploited by us to frame our solutions namely *Sparsity property* and the existence of *Interdigitating trees*.

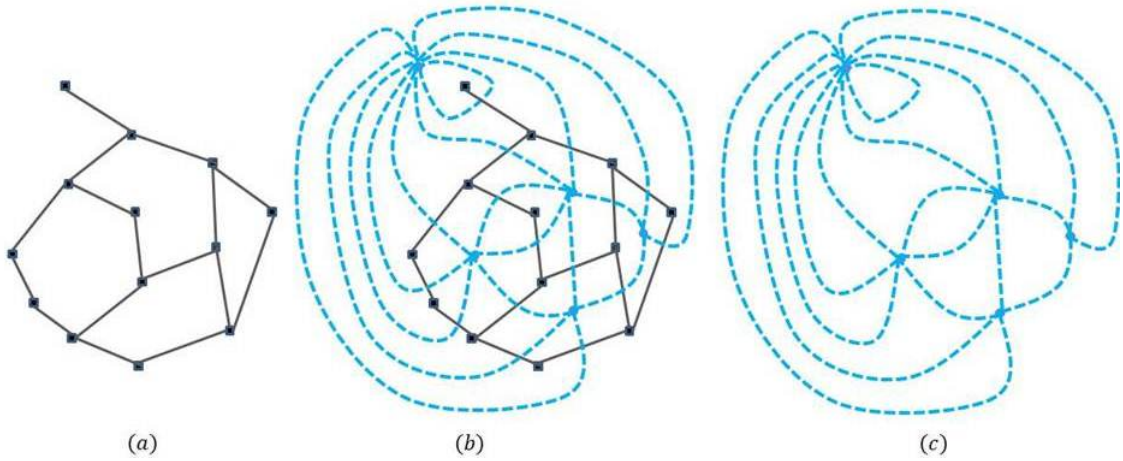


Figure 2.1: The figure shows the primal and dual graphs. (a) Shows the primal graph, (b) Shows the dual graph made using faces of primal graph as vertices, (c) Shows the dual graph.

2.2.1 Sparsity Property

The property is based on the following lemma.

Lemma 2.2.1. *Any simple connected planar graph G with n vertices has at most $3n - 6$ edges.*

Hence, for all planar graphs, the number of edges m is $O(n)$ and hence it is sparse, whereas for general graphs in the worst case m is of $O(n^2)$ making them dense. Importance of this property can be judged by the fact that most algorithms as DFS and BFS, whose complexity is directly affected by the number of edges now perform in linear time.

2.2.2 Planarity of Dual graph

Given a graph $G = (V, E)$, the dual graph $G^* = (V^*, E^*)$ is defined as follows. Each vertex in $V*$ corresponds to a face in G. Also for The dual graph G^* for a planar graph G has a vertex set V^* that corresponds to the faces of G. The original graph G is also called as Primal Graph. Since each edge in a planar graph G is adjacent to exactly two faces, it is used to connect the corresponding vertices in the dual graph G^*. Note that even though the planar graph may be simple but its dual graph may have multi-edges and self loops as two faces can share more than one edge in a planar graph as shown in Figure 3.2. Further note that each face of the dual graph G^* also corresponds to a vertex of the primal graph G. Hence, the dual of G^* is G.

Another interesting feature of the dual graph is that it is also planar. Many algorithms have exploited this property to solve problems that could not be directly solved by exploiting the planarity of primal graph.

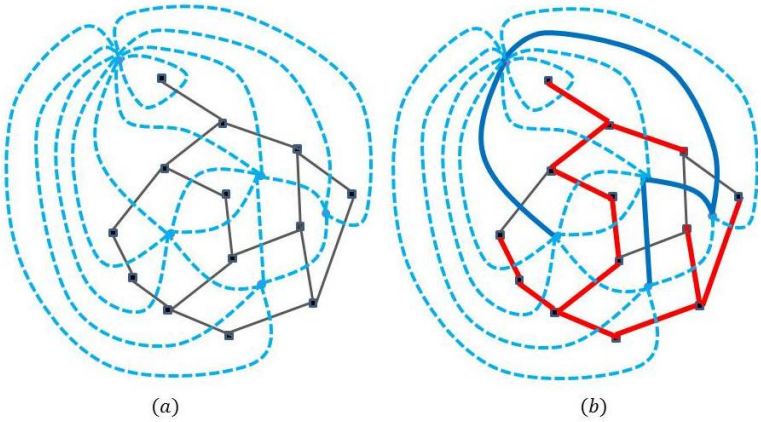


Figure 2.2: The figure shows the interdigitating trees. (a)Shows the primal graph (using black edges) and corresponding dual graph (using dotted blue edges). (b)Shows the spanning tree on primal graph (using bold red edges) and that of dual graph (using complete blue edges).

2.2.3 Existence of Interdigitating Trees

Any spanning tree of a planar graph selects $n-1$ edges of the graph. According to Euler's formula where V represents the set of vertices, F represents the set of faces and E represents the set of edges of a planar graph the following identity holds

$$| V | + | F | -2 =| E | \tag{2.1}$$

It can also be written as

$$(| V | -1) + (| F | -1) =| E | \tag{2.2}$$

Since $| V | -1$ edges are used to make the spanning tree of the primal graph, exactly $F-1$ edges are left. The beautiful aspect of this is the fact, that these remaining $| F | -1$ edges make the spanning tree of the dual graph as shown in Figure 2.2. Hence, these pairs of spanning trees also span all the edges of the planar graph and are called Interdigitating trees. This property is used to make efficient dynamic algorithms for dynamic problems in planar graphs as Maintaining dynamic Minimum Spanning Tree[17].

2.3 Top Trees

Dynamic data structures are developed so as to perform updates and answer queries in an online fashion. In fully dynamic trees, updates may include edge insertions, deletions or modifying weights on edges or vertices etc. Dynamic trees can be based on path decomposition, tree contraction or balancing euler tours. Path decomposition was first used by Sleator and Tarjan[6] to make ST Trees. Its implementation

was later simplified by them using splay trees[7]. Miller and Reif[8] introduced tree contraction and the contraction moves *Rake* and *Compress*. It was later simplified and modelled as RC Trees by Acar et al.[9, 10]. Tree contraction was also used by Fredrickson [11, 12, 13] to build topology trees having a limitation on the degree of vertices to be at most three. A balanced binary tree on euler tours was first used by Henzinger and King[14, 15] to make ET Trees, which was later simplified by Tarjan[16]. However ET Trees con not handle path related problems.

Top tree is a tree contraction based dynamic tree that is known to be the most powerful in terms of number of operations required, problems it solves and easy interface it provides. Top trees address problems that use information of the edges, vertices and paths. It can be used to answer any problem specific queries as minimum weight of an edge on a path, diameter of the tree, sum of edges on a tree, etc. Top trees were introduced by Alstrup et al.[1] and its implementation was later simplified by Tarjan and Werneck[2]. We shall describe the interface of top tree (i.e. various operations it allows) and describe some of its applications based on the works of Alstrup et. al[1] and Werneck [2]. However, details of implementation are not covered here.

2.3.1 Structure of top tree

In a forest each tree is represented by a Top tree, which is a binary tree of clusters. Each cluster represents a tree or a part of tree. Each cluster has two *boundary vertices*. The unique path connecting the boundary vertices is called the *cluster path*. The boundary vertices and the cluster path of a cluster are said to be *exposed* by the cluster. A cluster C can be of three types with the following properties:

1. **Base Clusters:** Each base cluster represents an edge of the tree and appears as a leaf of the top tree. The two vertices of the corresponding edge are called as boundary vertices.

2. **Internal Clusters:** Each internal cluster C represents a subtree of the whole tree. It has two children A and B, which may be base clusters or internal clusters having exactly one common boundary vertex. The subtree represented by C is formed by combining the subtrees represented by its children at the common boundary vertex. It has two vertices x and y called as boundary vertices and the path P_{xy} connecting x and y is called the cluster path.

3. **Root Clusters:** Each root cluster is an internal cluster without a parent. It represents a whole tree in the forest that is represented by the corresponding top tree.

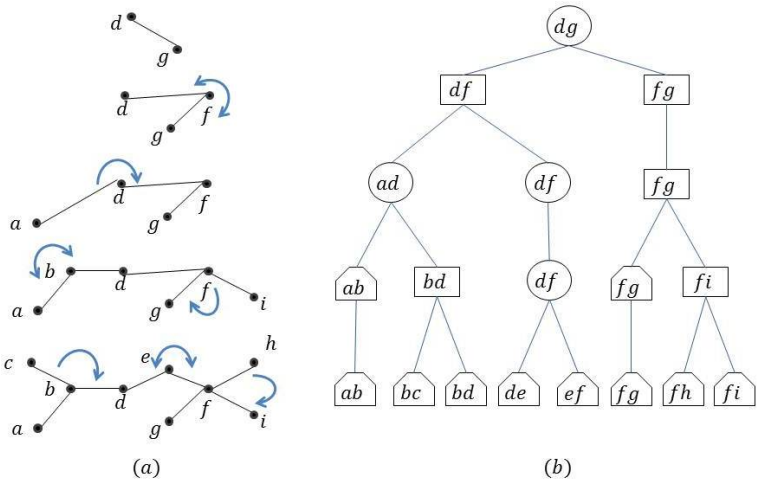


Figure 2.3: The figure shows the structure of top trees. (a) Shows the representation of tree on each level, (b) Shows the top tree made showing base clusters (trapezium), rake(rectangle) and compress (circle) clusters.

Hence, in a forest F represented by a set of top trees TT, each top tree is a binary tree of clusters. It is formed by performing tree contractions at on the tree reducing the tree at each level as shown in Figure 2.3. The root of TT is a root cluster, with the leaves being the base clusters. Every non-base cluster is formed by the combination of two clusters by tree contraction. Two tree contraction moves are allowed namely *compress* and *rake*, as shown in Figure 2.4. The original clusters before they were contracted are shown in Figure 2.4(a). The new cluster formed after contraction is shown in Figure 2.4(b), and the part of the cluster that is exposed is shown in Figure 2.4(c). Contraction moves are described as follows:

1. **Compress:** The two clusters with cluster paths xy and yz (refer to Figure 2.4) are compressed such that new cluster path is xz and common vertex y is no longer exposed. In order to perform this operation the degree of y should be two.

2. **Rake:** The two clusters A and B with cluster paths xy and yz (refer to Figure 2.4) are contracted using rake (called yz raked on xy) such that new cluster path is xy and the vertex z is no longer exposed. A is called the target and B is called the source. In order to perform this operation the degree of z should be one.

In each level of top tree some fraction of clusters are contracted. The clusters that were not contracted are made available to the higher level acting as the cluster of both the levels. It is ensured in the implementation of Top trees that at least a constant fraction of clusters are contracted at each level, such that the height of top tree becomes $O(\log n)$ as the number of base clusters are $n-1$.

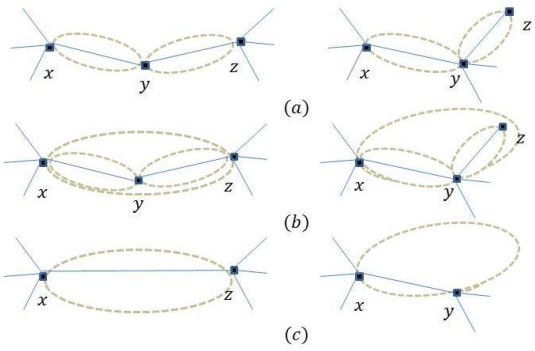


Figure 2.4: Tree contraction moves showing *Compress* and *Rake* on left and right.

2.3.2 Interface of top tree

The user is allowed to store some additional data D on each cluster. There are some operations defined on top trees called *external operations* that can be used to formulate solution of any problem. These are respectively

1. **Link(u,v):** It adds the edge (u, v) to the forest and connects two trees. It is added only if u and v belong to different trees.

2. **Cut(u,v):** It removes the edge (u, v) from the forest and divides a tree into two trees. An edge is deleted only if it originally exists in the forest.

3. **Expose(u,v):** It modifies the top tree such that the root cluster of the top tree consisting of u and v exposes the path uv as its cluster path. It is modified only if u and v are connected.

4. **Access(v):** It allows the user to access the data D associated with the root cluster of the top tree containing vertex v.

The first three external operations *Link*,*Cut* and *Expose* are *update* operations and they change the structure of the top tree. During *Access* operation no change is made to the structure of top tree, however the data D in the root cluster can be modified by the user. Each of these external operations are performed using a series of internal operations. These operations are not directly accessible to the user, however a user needs to define how the data D of the clusters involved is modified by each of these operations. The four internal operations are described as follows:

1. **Create(e):** It creates a new base cluster A on edge e.

2. **Join(A,B):** It creates a new cluster C which is formed by the contraction of clusters A and B.

3. **Split(C):** It splits an internal cluster C into its child clusters A and B.

4. **Destroy(A):** It destroys the base cluster A.

The modifications to the top trees during update operations are performed in a particular fashion which decides the order of execution of the internal operations. Firstly, *Split*s are performed in a top down fashion for all the clusters that are to be changed. Then *Destroy* operation is called if an edge is deleted. It is followed by any updates performed on the original forest. Now new base clusters are created using $Create$. Finally, a series of $Join$ operations are used to build the top tree in bottom up fashion.

It has been proved by Alstrup et. al[1] and Werneck [2] that each external operation can perform at most $O(\log n)$ $Join$ and $Split$ operations and $O(1)$ $Create$ and $Destroy$ operations. Hence, if the updates on data D in each internal operation takes constant time, all the external operations can be performed in $O(\log n)$ time.

In the original paper [1, 2], another operation $Select$ was described to extend top tree and solve problems involving non-local properties. However, it is not relevant to our work, so it is not described here.

2.3.3 Applications

Each cluster is associated with some problem specific data D. This data is manipulated by each of the internal operations. Hence, for describing the algorithm external operations are used and for describing the data manipulation internal operations are used. Following are a few applications of top trees that shall be used in some sections of the thesis report.

Aggregating Information

The problem involves finding the aggregate (minimum, maximum, sum etc.) of information stored on each element (edge or vertex) in a tree, subtree or a path of the tree subject to dynamic updates of $Link$ and Cut and weight increment/decrement over a path or subtree in an online fashion. First the solution is explained for maintaining minimum weighted edge with weight updates to be performed on a path and later the solution is generalised for other problems.

In each cluster C the data D stored comprises of two data values. Local minimum x_{min} stores the weight of the minimum weighted edge on the cluster path. Lazy update x_{upd} stores the value to be incremented to weight of each edge on the cluster path. However, the weight x_{min} is updated according to x_{upd}.

For base cluster x_{min} is initialised by the weight of the edge and x_{upd} is initialised to zero. Consider two clusters A and B having cluster paths xy and yz are children of the cluster C. When a $Join$ is performed $x_{upd}(C)$ is initialised to zero and the value of $x_{min}(C)$ is initialised as

1. For compress resulting in cluster path of C to be xz
   $x_{min}(C) = \min(x_{min}(A), x_{min}(B))$

2. For rake resulting in cluster path of C to be xy
   $x_{min}(C) = x_{min}(A)$

Also when split is performed the value of x_{min} and x_{upd} is updated for both A and B as

1. For compress resulting in cluster path of C to be xz
   $x_{upd}(A) = x_{upd}(A) + x_{upd}(C)$
   $x_{min}(A) = x_{min}(A) + x_{upd}(C)$
   $x_{upd}(B) = x_{upd}(B) + x_{upd}(C)$
   $x_{min}(B) = x_{min}(B) + x_{upd}(C)$

2. For rake resulting in cluster path of C to be xy
   $x_{upd}(A) = x_{upd}(A) + x_{upd}(C)$
   $x_{min}(A) = x_{min}(A) + x_{upd}(C)$

To find the minimum weight edge on a path, the top tree is simply exposed on that path and query is performed on x_{min}. Again to add some weight on each edge on a path, the weight increment is added to x_{upd} after exposing the path. Updates of edge insertion and deletion are simply performed using $Link$ and Cut.

Other operations like $Maximum$ and Sum can be handled similarly. For problems where query and modifications are done on the subtrees rather than paths, Cut is performed to make the subtree independent. Now query and data updates can be performed on the root cluster. The removed edge is added back using $Link$ operation to restore the tree.

Since base clusters of top trees are made on edges, it is easier to handle queries on edges. For handling vertices two approaches can be used. In the first approach, for each vertex a dummy edge can be introduced that stores the weight of the vertex, and weight of every other edge is identity element of aggregating operation, as zero for sum, and ∞ for minimum. In the second approach weights on vertices are stored in some external array. The weight of a vertex is accounted in a the data of a cluster if it is not exposed by the cluster. Since a vertex can be an internal vertex of at most one cluster no vertex is accounted for more than once. Hence, while performing $Join$ only the weight of vertex which no longer remains exposed is accounted.

2.3.4 Nearest Marked Vertex

The problem involves dynamically marking and unmarking any vertex on the tree and querying for the nearest marked vertex for any vertex in an online fashion subject to Link, cut updates in the tree.

For each boundary vertex in a cluster, we store its nearest marked vertex v_{nmv} in the subtree associated with the cluster. However, to find v_{nmv} for each boundary vertex, we do not consider any boundary vertices of the cluster. Also the distance of

the nearest marked vertex is stored in d_{nmv}. Further the length of the cluster path is also maintained in d_{cp}.

For each boundary vertex of each base cluster, v_{nmv} is initialised with $NULL$ and d_{nmv} is initialised with ∞. Also d_{cp} for every base cluster is initialised with the weight of the edge corresponding to the base cluster. Now, when *Join* operation is called on clusters A and B with cluster paths xy and yz, it initialises the value of data D for new cluster C as follows.

1. For compress operation forming C with cluster path xz
   $d_{cp}(C) = d_{cp}(A) + d_{cp}(B)$
   $d_{nmv}(C, x) = min(d_{nmv}(A, x), d_{nmv}(B, y) + d_{cp}(A), d_{cp}(A))$
   $d_{nmv}(C, z) = min(d_{nmv}(B, z), d_{nmv}(A, y) + d_{cp}(B), d_{cp}(B))$

   The last argument of $d_{nmv}(C, x)$ and $d_{nmv}(C, z)$ is considered only if y is marked.

2. For rake with A as target and cluster path or C as xy
   $d_{cp}(C) = d_{cp}(A)$
   $d_{nmv}(C, x) = min(d_{nmv}(A, x), d_{nmv}(B, y) + d_{cp}(A), d_{cp}(A) + d_{cp}(B))$
   $d_{nmv}(C, y) = min(d_{nmv}(A, y), d_{nmv}(B, y), d_{cp}(B))$

   The last argument of $d_{nmv}(C, x)$ and $d_{nmv}(C, y)$ is considered only if z is marked.

Also $v_{nmv}(x)$ and $v_{nmv}(z)$ (or $v_{nmv}(y)$ for rake) is initialised accordingly.

For marking or unmarking a vertex x, the tree is exposed to the path xk for any vertex k. Now the marked status of x can be changed without creating any inconsistency in the underlying top tree as boundary vertex is not accounted for in the data stored.

2.4 Multiple Source Shortest Path in Planar Graphs

Given a planar graph G, having n vertices and m edges with non-negative edge lengths, a data structure is built to report the shortest path between any two vertices on the planar graph.

This section is based on solution for Multiple Source Shortest Path problem(referred as MSSP) given by Klien [3] and the extension to genus g graphs by Cabello [4]. This algorithms finds shortest path trees on all vertices of the infinite face in a planar graph. It starts off with making the shortest path tree for one of the vertex v, and modifies the tree to represent the shortest path tree of the successor of v on the infinite face and so on. The main idea here is that while making these shortest path trees on all the vertices of the infinite face we need to make only $O(m)$ edge changes

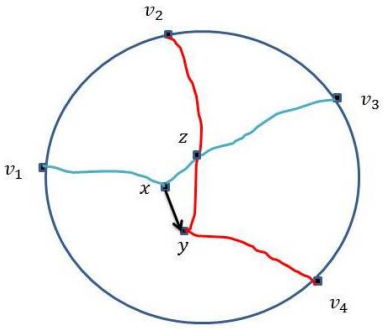


Figure 2.5: Shortest path from v_1 and v_3 to y has arc xy shown in blue, then shortest path of atleast one of v_2 or v_4 should pass through them shown in red,

to the tree. For undirected graph we assume each undirected edge comprises of two directed edges (referred as *arcs*) of opposite directions.

The main idea behind this approach is that an arc can exist in the shortest path tree of a continuous set of vertices on infinite face f_∞. The result can be proved using the following lemma by Klien[3].

Lemma 2.4.1. *Given a planar embedded graph G with infinite face f_∞ on which vertices v_1, v_2, v_3 and v_4 occur in order. A directed edge e that is present in shortest path tree on v_1 and v_3, must be present in the shortest path tree of at least one of v_2 or v_4.*

Proof. Consider figure 2.5, the shortest paths from v_1 and v_3 to y consists of the shown arc xy. Suppose that all shortest paths (in case more than one have the same weight) from v_2 and v_4 to y are such that none of them contains edge xy. Now because of planarity at least one of shortest paths from v_2 and v_4 to y must intersect the path from v_1 or v_2 to y. Without loss of generality consider it to be the path from v_2 intersecting at z.

Now, the path from z to y excluding xy cannot be shorter than the one including it otherwise v_3 can be connected to y using a shorter path following the path that excludes xy after z. Hence, we can use the same path to connect v_2 to y. So it is a contradiction to our earlier assumption hence there exist shortest path tree that contains e for at least one of v_2 or v_4. □

Lemma 2.4.1 can now be used to prove the lemma by Klien[3].

Lemma 2.4.2. *Given a planar embedded graph G with infinite face f_∞ on which shortest path trees are made on each vertex. A directed edge e is present in shortest path trees of a contiguous subset of these vertices around the cycle.*

Proof. Again we give the proof by contradiction. Suppose there exist at least two disconnected subset of vertices around f_∞ which use and arc e. So we can choose v_1 and v_3 from these two subsets and v_2 and v_4 from the vertices of the set preceding

and succeeding the set containing v_3. Hence, the vertices v_1, v_2, v_3 and v_4 are in order. So we can apply Lemma 2.4.1 to prove that our assumption is wrong. Hence, each arc e exist in the shortest path trees of a continuous subset of the vertices around f_∞. □

We can now bound the number of edge changes when we make shortest path tree on each vertex of f_∞ in order from the preceding vertex.

Lemma 2.4.3. *(Limited Edge Changes) Given a planar embedded graph G having m directed edges with infinite face f_∞ on which shortest path trees are made on each vertex. When moving around the f_∞ making shortest path tree of the vertex from previous vertex we make $O(n)$ changes over the whole face.*

Proof. Since each shortest path tree is a spanning tree, it is bound to have exactly $n-1$ edges. Also by Lemma 2.4.2 we know that each arc exist in continuous sequence of vertices, each arc can then enter in the shortest path tree only once as we move around f_∞. In order to maintain the edges in the tree for each incoming edge exactly one edge leaves the tree. Hence, each arc can contribute to one addition and one deletion of edge. As number of arcs are m there can be at most m edge changes (a change corresponds to one edge deletion and one edge insertion). Also for planar graph m is bounded by $O(n)$, so the number of changes can be bounded to $O(n)$. □

After making the shortest path tree on first vertex on f_∞, they slide the root vertex from v_1 to v_2 along the edge connecting the two vertices. Now at v_1 all vertices have distances marked as their shortest distances from v_1. As they move along the edge the distances of vertices change gradually such that at any time distance of only one vertex v_x is incorrect, which is corrected by a single edge change. The tree is maintained using operations of dynamic trees, hence they go on performing edge changes as they reach v_2. All these edge deletions and insertions can be stored in list $EdgAdd_i$ and $EdgDel_i$ respectively for each vertex v_i.

Hence, using MSSP algorithm we can retrieve information of $EdgAdd$ and $EdgDel$ for the given planar graph embedding as described above.

Chapter 3

Top tree for Planar Graphs

3.1 Introduction

Top tree is a powerful data structure for maintaining dynamic trees. It can solve problems on trees dealing with properties based on vertices, edges and paths as introduced in Section 2.3. It solves a variety of problems for trees including aggregating information, nearest marked vertex, maintaining center and diameter, subject to dynamic updates in the tree as edge insertion and deletion. The aim is to make a similar structure for planar graphs that can solve problems related to vertices, edges and faces.

The possible implementation of top trees is described by Alstrup et. al[1] and Werneck[2]. The top tree on a graph G having n vertices can be made in $O(n)$ time using $O(n)$ space. They have implemented each of the three update operations that change the structure of top tree, namely $Link$, Cut and $Expose$ in $O(\log n)$ time. $Expose$ operation on a cluster path, takes two vertices (ends of cluster path) and exposes them to the root cluster, taking $O(\log n)$ for each.

Our solution introduces a data structure D_{tt} similar to top trees, that solves a limited set of problems based on how the local solution for subgraph can be merged to form the solution of the whole graph. Technically D_{tt} is easily implementable using top trees, the main contribution of D_{tt} is the *design* of the $interface$, providing user an easier access to advanced techniques using top trees to solve problems on planar graphs. We use the phrase *subgraph enclosed* by a *cycle or cut*, which refers the subgraph of the planar graph that is either internal or external to the cycle (or cut). We introduce the notion of *exposing* the planar graph on a subgraph enclosed by a cycle or cut. There exist an analogy between top trees and D_{tt}. The two boundary vertices in top trees is analogous to the cycle or cut in D_{tt}. The cluster path connecting the boundary vertices is analogous to the subgraph enclosed by the cycle. In case D_{tt} is not exposed on any cycle or cut, the data on D_{tt} accounts for the whole graph, whereas, on an exposed D_{tt} the data will only account for the subgraph that is internal or external to the cycle(or cut) in the primal graph. A

cycle in the primal graph appears as a cut in the dual graph and vice versa. Hence, both cycle and cut corresponds to the primal graph unless specified otherwise.

For a given embedded planar graph G having n vertices ,we make a data structure D_{tt} in $O(n)$ time and using $O(n)$ space. It can update the weights of an element(vertex,edge or face) in $O(\log n)$ time. The *Access* and *Query* operations are performed in $O(1)$ time. *Expose* operation called on a subgraph, enclosed by a cycle or cut of size d is performed in $O(d\log n)$ time. Note that the size of input for *Expose* is d, hence any algorithm for this problem will require $\Omega(d)$ time.

The first problem solved using D_{tt} is aggregating information (Minimum, maximum or sum) on elements(vertices, edges or faces) of planar graph G having n vertices, with queries and additive weight updates on the elements of a subgraph, enclosed by a cycle (or cut) including or excluding the external cycle(or cut). The queries and updates are performed in an online fashion. The solution takes $O(d\log n)$ time for performing expose and $O(\log n)$ time to perform query and update operations where d is the length of the input cycle(or cut). Whereas a trivial solution for this problem require $O(n)$ time.

The second problem solved using D_{tt} is maintaining dynamic Minimum Spanning Tree(referred as MST) of a planar graph G having n vertices, which allows online updates on edge weights and query of MST of a subgraph enclosed by a cut on G. The solution takes $O(d\log n)$ time for performing query of MST on a subgraph and $O(\log n)$ time to perform query and update operations where d is the length of the input cut. Whereas a trivial solution for this problem require $O(n\log n)$ time.

The main idea used is the existence of interdigitating trees in planar graphs as explained in Section 2.2.3. All the edges of the graph are covered by maintaining two top trees, on the spanning tree of primal and dual graphs respectively. Top trees on primal spanning tree maintains information of vertices, whereas that of dual spanning tree maintains information on faces. The information on edges can be maintained by either or both of the trees. Further the operation expose is used to isolate the information for a subgraph enclosed by a cycle or cut.

We begin with giving an overview of the result we obtained and the intuition behind the approach used in Section 3.2. The the structure of D_{tt} is described giving details of its interface provided to the user in Section 3.4. Then the implementation details of various external operations of D_{tt} are described in Section 3.5 and analysed in Section 3.6. Then the applications of D_{tt} for Aggregating information on Planar graph and to maintain dynamic MST on subgraph of the planar graph are described in Section 3.7 and Section 3.8 respectively.

3.2 Overview

As mentioned in previous section, the main aim was to develop a structure similar to top trees that can address problems on planar graphs. However, several difficulties exist with planar graphs as compared to trees that limits the dynamic nature of the structure D_{tt}. In D_{tt}, we do not allow addition or deletion of edges. *Link* and *Cut* cannot be performed directly as an external operation as it would change the structure of the planar graph, and in particular the faces. Further adding an edge may make the graph non-planar, so only those edges can be added whose vertices share a common face. Each edge addition splits a face into two. Also each edge deletion merges two faces as one.

The property of interdigitating trees explained in Section 2.2.3 is used to represent the graph using two spanning trees. Top trees are made on each of the spanning trees. The main operation *Expose* called on cycle or cut of the primal graph, divides a top tree into two trees representing subgraph internal and external to the cycle. This is done by changing the spanning tree of the concerned top tree such that only one edge is common to the cycle or cut. This can be achieved easily using the top trees present in the structure.

The flexibility of choosing any spanning tree to make T_d and T_p and any spanning tree of the subgraph enclosed by a cut in expose, allows us to solve dynamic MST problem described in Section 3.8. The MST of graph G is chosen as the spanning tree of the primal graph. For making the spanning tree of the subgraph enclosed by the cut while performing expose, we choose the minimum weight edges that makes an MST on the subgraph. The main idea behind this approach is similar to one used by Epstien et al.[17] for maintaining fully dynamic MST for planar graphs.

3.3 Exposing a subgraph in primal and dual graphs

Input of the expose operation is a cycle or a cut on the primal graph representing the subgraph that has to be exposed. A cycle on the primal graph represents a cut on the dual graph and vice-versa. A cycle in the primal graph (shown in Figure3.1(a)) is used to expose the subgraph of the dual graph that is internal or external to the cycle as shown in Figure 3.1(b). Similarly a cut in the graph(represented by a cycle in dual graph shown in Figure3.1(c)) exposes a subgraph of the primal graph internal or external to the cut as shown in Figure 3.1(d). A cycle (or cut) can enclose two subgraphs, one internal and one external to it. To identify uniquely the subgraph enclosed a convention is used based on the convention used by the planar embedding to order the edges around a vertex. It can be either clockwise or anti-clockwise. Without loss of generality assuming the order used is clockwise, the euler tour will traverse all the faces in anti clockwise direction. Hence, given a cycle

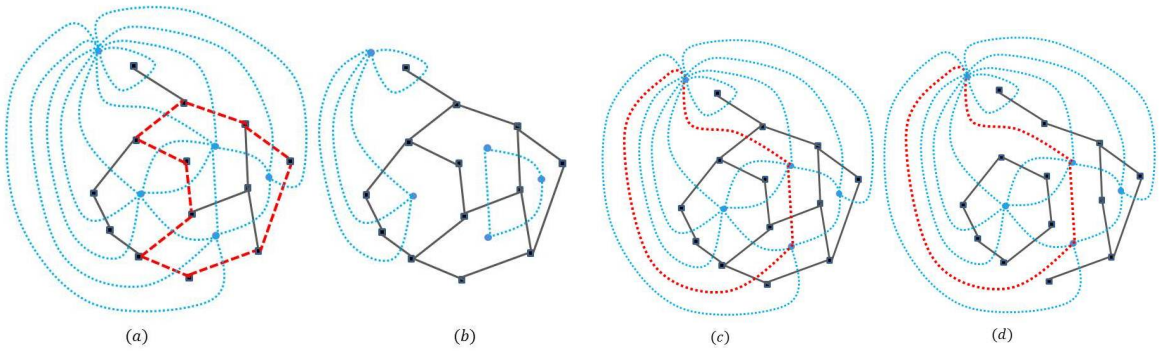


Figure 3.1: The figure shows how the expose operation changes divides a planar graph (a) Shows a cycle on the primal graphs, (b) Shows the dual subgraphs enclosed by the cycle of primal graph, (c) Shows a cycle on dual graph (representing a cut on primal graph), (d) Shows the primal subgraphs enclosed by the cut on the primal graph.

whose edges are ordered in anti-clockwise direction on the embedding will represent the subgraph internal to the cycle and clockwise order will correspond to external of cycle. Also a cut in the primal graph is a cycle in the dual graph. So similarly for the top tree on dual graph, a cycle in anti-clockwise order represents the subgraph internal to it and vice-versa.

3.4 Top Trees Structure for Planar Graphs

Top tree structure D_{tt} can be made for a planar graph G, by representing it with two top trees T_p and T_d, respectively on the interdigitating trees on primal and dual graphs. Any spanning tree can be selected to choose these interdigitating trees. Further, when expose is called on a cut(or cycle) T_p (or T_d) is modified such that only one edge of T_p(or T_d) intersects the cut(or cycle). Now the top tree T_p(or T_d) can be divided into two top trees T_p and T_p^*(or T_d and T_d^*) such that T_p(or T_d) represents the subgraph internal to the cut(or cycle) and T_p^*(or T_d^*) represents the subgraph external to the cut(cycle) by removing the edge e^* intersecting the cut(or cycle). Also an edge e^* is maintained as a base cluster of top tree required to join the two trees T_p and T_p^* (or T_d and T_d^*) when G is unexposed. Weights on vertices and faces are saved in external arrays namely w_v and w_f.

As opposed to top tree made on a tree, making a top tree like structure on a planar graph has some limitations. The fully dynamic updates are not allowed, that is to change the structure of graph using *Link* and *Cut*, because of numerous reasons. Firstly, any edge insertion will split a face in two adjacent faces, also any edge deletion will merge the two faces adjacent to it. Since the structure allows data to be associated with a face, it may lead to inconsistency as a vertex will be added

to T_d, which is not supported by top trees. Further performing a *Link* cannot be done for any edge as it might make the graph non-planar. A *Link* is possible only when the two vertices adjacent to the edge share a common face.

The most non-trivial aspect of the data structure is the operation *Expose*. It makes a top tree on the spanning tree of the subgraph enclosed by the cycle (or cut). Thus any information maintained on trees using top trees, as diameter or center can be maintained for this spanning tree.

User is allowed to store some data on clusters of T_p and T_d respectively. Generally the data on clusters of T_p is dependent on properties of vertices and that stored on clusters of T_d is dependent on properties of faces. The user has to define how this data is modified when internal operations of both top trees are performed. Further the external operations of D_{tt} can be accessed by user to solve the problem.

The main intention behind *Expose* is to reduce the data on T_p (or T_d) to represent the exposed subgraph only, now it can also be used to perform lazy update or query on the subgraph. When a graph is exposed on a cycle/cut the top tree is divided into two sections that were earlier separated by the cycle/cut.

The data structure D_{tt} can be initialised by finding any spanning tree and making a top tree on it. Both these operations can be performed in $O(n)$ time using $O(n)$ space.

3.4.1 External Operations on the Graph

These operations are used to define the algorithm for solving a problem. They modify the structure D_{tt} and are implemented using some internal operations of top trees. The structure supports the following external operations defined as follows:

1. **Expose(Cycle C):** This operation exposes a cycle C of the primal graph dividing the T_d into T_d and T_d^* representing the subgraph enclosed by the cycle and the residual graph respectively. It is used for operations in the dual graph especially on properties related to faces.

2. **Expose(Cut C^*):** This operation exposes a cut C^* of the primal graph dividing the T_p into T_p and T_p^* representing the subgraph enclosed by the cycle and the residual graph respectively. It is used for operations in the primal graph especially on properties related to vertices.

3. **UnExpose:** This operation restores the graph after an expose to represent the whole graph.

4. **Access(Vertex v):** It allows user to access the data stored in the root cluster of the primal top tree T_p having vertex v.

5. **Access(Face f):** It allows user to access the data stored in the root cluster of the dual top tree T_d having vertex corresponding to face f.

6. **Update(Element e):** It involves updating the weight of a vertex, edge or face in a weighted graph.

Note: No operations corresponding to *Link* and *Cut* in top trees are present that changes the structure of the graph. These can be partially simulated by changing the weight of the edge to be ∞ or zero according to application.

3.4.2 Internal Operations

The external operations are performed using a series of internal operations available in top trees. These are not accessible to the user, but defines how the data on clusters of T_p and T_d is modified on application of these internal operations. Each of these operations are separately defined for T_p and T_d. There are four internal operations

1. **Create(Edge e):** It creates a base cluster of the top tree, representing an edge of graph G.

2. **Join(A,B):** It joins the two given clusters that have exactly one common boundary vertex on the corresponding graph(primal or dual). It returns a cluster C that has A and B as its children.

3. **Split(C):** It deletes the cluster C and returns its two child clusters A and B.

4. **Destroy(Edge e):** It deletes the base cluster representing the edge e.

3.5 Implementation details

This section describes how various External operations of D_{tt} are performed using operations of the top tree. Most of these implementations are trivial except for *Expose* and *Update Element*. In order to perform these operations efficiently some extra data is stored on each cluster that is not accessible by the user.

3.5.1 Expose(Cycle/Cut) and UnExpose:

Implementing Expose on a cycle (and hence on T_d) is similar to that on a cut(and hence on T_p), only change being using T_p and T_d interchangibly. For implementing expose on a cycle C (i.e. on dual graph G^*), the top tree T_d is divided into two trees namely T_d and T_d^*, such that they are respectively internal and external to C. T_p remains to be a spanning tree of the primal graph. Since the spanning tree of dual

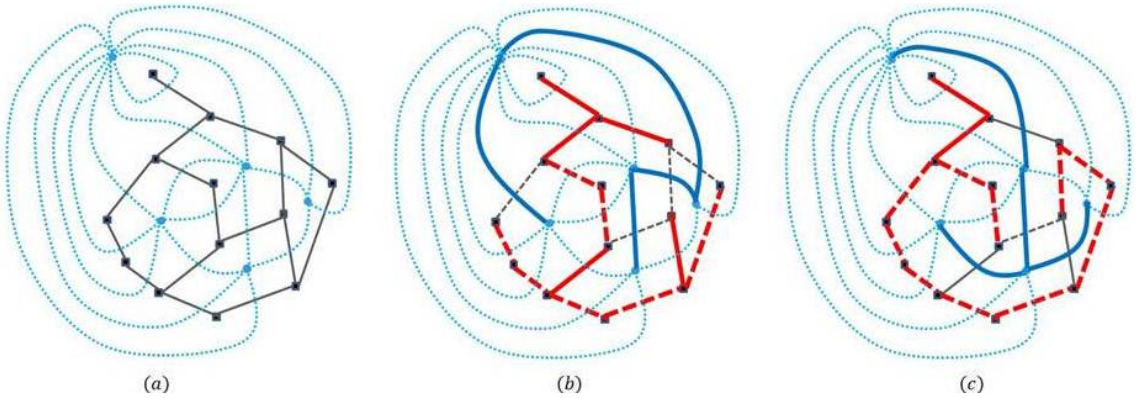


Figure 3.2: The figure shows how the expose operation changes the spanning tree on primal and dual graphs. (a) Shows the primal graphs , (b) Shows the spanning tree of primal and dual graph,indicating a cycle C by dashed lines, (c) Shows the new spanning trees after calling expose on C such that only one edge of C intersects the spanning tree on dual graph.

graph has to be connected, yet it is divided into two trees (T_d and T_d^*), the extra edge e^* not present in either of them is stored separately. Hence we modify the spanning tree of the dual graph such that it contains only one edge that is present in the cycle C. Finally since all edges of G has to be present in either T_p or T_d, all edges of C will now be present in T_p except for e^*.

For each edge e of T_d present in C, Cut is performed on T_d. This edge is to be added to T_p, forming a cycle C_p. The *replacement* edge of e is an edge from T_p present in C_p and not in C. This replacement edge has to be deleted from T_p and added to T_d. To perform this operation efficiently we mark all the edges of C present in T_p. Now replacement edge is any unmarked edge on the cycle C_p formed by adding edge e to T_p.

Edges on the cycle C that are on T_p, are marked by setting a flag in an array $Mark$ defined for all edges. The flag is set to indicate a marked edge or its presence on cycle C. Each cluster stores an available unmarked edge (if present) on the cluster path. An edge from T_p that is present in C_p and not in C, is found by finding the available unmarked edge on C_p.

Data Stored on Cluster

Each cluster stores an available unmarked edge on the cluster path in e_{um}. The value of e_{um} is $NULL$ in case no unmarked edge is present on the cluster path.

Data manipulation by internal operations

A base cluster on edge e, made by $Create$ is initially initialised according to mark flag on e. So $e_{um} = e$ if $Mark(e)$ is zero and $e_m = NULL$ if $Mark(e) = 1$. When $Join$ is performed on clusters A and B to make C, the available unmarked edge

of either A or B present on the cluster path of C is stored in e_{um}. For compress, $e_{um}(C) = e_{um}(A)$, if $e_{um}(A) \neq NULL$ and $e_{um}(C) = e_{um}(B)$ otherwise . For rake, $e_{um}(C) = e_{um}(A)$. No change is performed on data for split and delete.

Algorithm

In Algorithm 3.1, in the first for loop all the edges of C present in T_p are marked. In the second loop, for each edge $e(x, y)$ (where x and y are vertices of dual graph and hence faces of primal graph) of C present in T_d, a replacement edge is found in T_p as explained earlier. The edge e is then replaced by the replacement edge e_{tmp} in T_d. Also e is now marked since it is an edge of C present in T_p.

The replacement edge e_{tmp} cannot be found if all the edges on the cycle C_p are in C. Hence, the cluster path represents the cycle C, indicating that all edges on T_d are now internal to C. Finally in the third loop, all edges on C are unmarked to restore the state of D_{tt}.

Unexpose is performed by simply by performing $Link(e^*)$ on T_d.

3.5.2 Update(Element):

To update a vertex v(or face f) expose is called on the corresponding top tree T_p (or T_d), such that v(or f) to be updated becomes a boundary vertex. Since the data on the cluster does not take the boundary vertices into account, value of v(or f) can be safely updated in wt_v(or wt_f) as it is not internal to any cluster on T_p(or T_d).

The weight on an edge is stored on the base clusters of top tree. *Cut* is performed on the edge and then it can be safely updated, followed by performing *Link* to restore it.

3.5.3 Update(Cluster) and Query:

The root cluster of respective top tree (T_p or T_d) is returned to the user. It can be used to update or query the data stored on the root cluster.

3.6 Analysis

The analysis of all the internal operations of D_{tt} is similar to that of Top trees. Hence, all internal operations are performed in $O(1)$ time. All the external operations of top trees namely *Link*, *Cut* and *Expose* require $O(\log n)$ time as claimed Alstrup[1] and Werneck[2].

External operations of D_{tt} namely *Access* and *Query* are performed in $O(1)$ time as it directly accesses the root cluster of T_p or T_d. *Unexpose* and *Update* requires constant number of *Link*,*Cut* and *Expose* operations on top tree. Hence

Algorithm 3.1: Exposing a cycle C on the dual graph G^*

Input : Planar Graph embedding G having a cycle C in the primal graph, with the data structure D_{tt} built on G.

Output: T_d: Top tree on dual graph G^* internal to cycle C
T_d^*: Top tree on dual graph G^* external to cycle C
e^*: Edge separating the top trees T_d and T_d^*

/*`Marking all the edges of` C `present in` T_p */
foreach *edge* $e \in C$ **do**
  if $e \in T_p$ **then**
    Cut(e) in T_p;
    Mark(e) $\leftarrow 1$;
    Link(e) in T_p;
  end
end

/*`Finding replacement edge for each edge of` C `present in` T_d */
foreach *edge* $e(x, y) \in C$, *with* x *internal to* C **do**
  if $e \in T_d$ **then**
    Cut(e) on T_d;
    Expose T_p on path xy;
    Access(x) and $e_{tmp} \leftarrow e_{um}$;
    /*e_{tmp} `is assigned the value of` e_{um} `stored on the root cluster returned by Access(`x`)` */
    if $e_{tmp} \neq NULL$ **then**
      Cut(e_{tmp}) on T_p;
      Link(e_{tmp}) on T_d;
      Mark(e) $\leftarrow 1$;
      Link(e) on T_p;
    else
      $e^* \leftarrow e$;
      $T_d^* \leftarrow$ Access(y);
    end
  end
end

/*`Unmarking all the edges of` C */
foreach *edge* $e \in C$ **do**
  if $e \in T_p$ **then**
    Cut(e) in T_p;
    Mark(e) $\leftarrow 0$;
    Link(e) in T_p;
  end
end

it is performed in $O(\log n)$ time. Assuming the length of the exposed cycle/cut C to be d, *Expose* operation of D_{tt} performs constant number of external functions of top trees (*Link*, *Cut*, *Expose*) for each edge of C as described in Algorithm 3.1. Hence, *Expose* can be performed on a cycle of size d in $O(d \log n)$ time.

Hence, following can be stated

Theorem 3.6.1. *Given an embedded planar graph G having n vertices ,we can make a data structure D_{tt} in $O(n)$ time and using $O(n)$ space, which allows Update of an element in $O(\log n)$ time and performing Access, Query in $O(1)$ time and Expose in $O(d \log n)$ time, on a subgraph enclosed by a cycle or cut of size d.*

3.7 Aggregating Information on Planar Graphs

It basically includes problems dealing with performing operations (as minimum, maximum, sum) on the elements (vertices, edges or faces) of a subgraph enclosed by a cycle(or cut). In case of edges, the solution may or may not include the cycle (or cut). Many operations can be considered in this category, but current solution limits it to the problems that can be classified using the following properties:

1. **Elements:** Vertices/Faces/Edges
2. **Operation:** Minimum/Maximum/Sum
3. **Limit on External Cycle/Cut:** Including/Excluding

Any combination of the above properties can be currently solved by the data structure D_{tt}. Solutions of these problems differ only in how the data is initialised and manipulated during various internal operations of top trees as Create, Destroy, Join and Split. Faces are handled using T_d, where the graph can be exposed on a Cycle representing a subgraph in the dual graph G^*. Vertices are handled using T_p, where the graph is exposed on a Cut representing a subgraph in the primal graph G. Edges can be handled using either of T_p or T_d based on whether it needs to perform operations and updates on cycles or cuts in the primal graph. Note that for vertices (or faces) the information aggregated on the subgraph enclosed by a cut(or cycle) is same as the information aggregated on the subtree represented by the root cluster of T_p (or T_d) and can be used interchangibly.

Since a face in the dual graph and a vertex in the primal graph can be treated similarly, the solution is presented for performing additive weight updates and finding minimum weighted vertex in the subgraph enclosed by a cut C^* of the primal graph. The solution is then extended to solve the other operations and handle other elements of the problem.

3.7.1 Minimum Weighted Vertex on subgraph

The problem is solved using operations on top tree T_p and the subgraph is implied by using a cut C^* in the primal graph where weights of vertices are stored in an external array w_v.

Operations Allowed

1. Update an Element (Vertex)
2. Add constant to weights of vertices in a subgraph (enclosed by a cut)
3. Query for the minimum weighted vertex in a subgraph (enclosed by a cut)

Solving the problem using data structure D_{tt} involve two parts. Firstly, the data stored on each cluster and its manipulation by the internal operations. Secondly, the algorithm for each operations mentioned above using a the external operations of D_{tt}.

Data on Each Cluster

1. **Data Variable(x_d):** The weight of the minimum weighted vertex in the sub-tree represented by the cluster excluding the boundary vertices.
2. **Update Variable(x_u):** The lazy update information acting as additive update to the weight of each element of the sub-tree represented by the cluster. This update is already applied on the value of x_d and that of external vertices.
3. **Solution(v_{min}):** The minimum weighted vertex in the subtree represented by the cluster excluding the boundary vertices.

Manipulating Data using Internal operations

1. *Create(Edge e):* Since the information stored on data variables exclude the boundary vertices, for a base cluster consisting of two boundary vertices the data will be initialised as $x_d = \infty, x_u = 0$ and $v_{min} = NULL$.
2. *Join(A,B):* The update variable x_u will be initialised with zero for the new cluster formed. Let the cluster path for A be xy and that of B be yz. The data for the cluster C formed is initialized as following
   (a) For compress operation such that the cluster path of C is xz
   $x_d(C) = min(x_d(A), x_d(B), wt_v(y))$
   (b) For rake operation such that the cluster path of C is xy
   $x_d(C) = min(x_d(A), x_d(B), wt_v(z))$

The vertex $v_{min}(C)$ is initialised accordingly.

3. *Split(C):* The update variable will propagate the update to both child clusters A and B, updating the weight of minimum weighted vertex and the disappearing vertex x as
   $x_d(A) = x_d(A) + x_u(C)$
   $x_u(A) = x_u(A) + x_u(C)$
   $wt_v(x) = wt_v(x) + x_u(C)$

4. *Destroy(c):* It has no effect on the x_d or x_u.

Algorithm

1. *Update an Element (Vertex):* External operation of D_{tt} can be directly used to update the weight on vertices.

2. *Additive update to subgraph (enclosed by a cut):* The subgraph is first exposed on the cut using the external operation *Expose* on D_{tt}. Let T_p have boundary vertices v_x and v_y. Then Access(v_x) is used to add the constant x_c, as follows
   $x_u = x_u + x_c$
   $x_d = x_d + x_c$
   $wt_v(v_x) = wt_v(v_x) + x_c$
   $wt_v(v_y) = wt_v(v_y) + x_c$

3. *Query:* The subgraph is first exposed on the cut using the external operation *Expose* on D_{tt}. Let T_p have boundary vertices v_x and v_y. Then Access(v_x) is used to find the answer to the query as follows
   $Answer = min(x_d, wt_x(v_x), wt_x(v_y))$
   The minimum weighted vertex v_{min} is returned accordingly.

3.7.2 Handling other operations

The operation Maximum can be handled similarly by changing the identity element used during initialization and the operation used in *Join*. However, for maintaining Sum an extra data element is used on each cluster namely *Size* (x_s). It stores the number of elements on the cluster excluding the boundary vertices. It is initialized with zero for base clusters and used in other operations as follows:

1. For Join(A,B) in case of both compress or rake to form the cluster C
   $x_s(C) = x_s(A) + x_s(B) + 1$

2. For additive updates of weight x_c to the weights of elements in the subgraph
   $x_d = x_d + x_s * x_c$

3.7.3 Handling faces and edges

Faces can be handled easily by replacing T_p by T_d and vice-versa because of the duality property as explained in Section 2.2.2. Also we use the weights of faces wt_f instead of weights of vertices wt_v.

However, dealing with edges is difficult as all edges are not present in either T_p or T_d. So the information of each edge is stored on the vertices or faces adjacent to it. Note that using this procedure all the edges are counted twice as each edge is adjacent to two vertices and two faces. Now according to whether the expose is used with Cycles or Cuts, we can use either T_d or T_p respectively. Again the solution for both the cases will be similar because they are represented as vertices and faces in T_p and T_d respectively. Also whether or not external cycle or cut defining the subgraph has to be used in aggregation comes into picture only for edges.

Considering expose on Cuts, information is aggregated on respective vertices. The solution will be based on primal graph and hence T_p, with queries including the external cut as well.

Maintaining Minimum/Maximum

The non-trivial part of this problem is the change in value stored on vertex if an edge update on a single edge is applied. The query and additive increment is same as that for vertices. To maintain the minimum(or maximum) edge, stored on a vertex, a segment tree is built on each vertex having entries for each edge maintaining the minimum(or maximum) value at the root. Now whenever the weight of an individual edge is updated, it is updated in the segment tree and hence new minimum(or maximum) can be found in $O(\log n)$ time. Also while performing an additive update on the subgraph similar update is done on the vertices adjacent to the edges on the Cut.

Maintaining Sum

Maintaining Sum of edges is simple as weight of each vertices can be initialised with the sum of edges adjacent to it. The number of edges incident is stored for each vertex in an external array sz_v. sz_v is used to update x_s in join operations with disappearing vertex v_x as
$x_s(C) = x_s(A) + x_s(B) + sz_v(v_x)$
Since weight of each edge is added twice in the sum except for the edges on the external cut, the sum of the edges on cut is taken in $EdgSum$. The corrected Sum is hence calculated as
$Sum^* = (Sum + EdgSum)/2$
Now the updates on edges can be simply handled by adding the change in weight to corresponding vertices. And additive increment of weight x_c is handled by adding

weight to lazy update. The sum is updated as
$x_d = x_d + x_s * x_c$
Also x_c is added to lazy update x_u, but as it is propagated down to the base clusters, each vertex adjacent to and edge adds only half of x_u to the edge, because the edge is updated for the same additive update by two vertices.

Handling exclusion of External Cycle/Cut

If edges on the external cut are to be excluded from the Sum, the corrected sum is now calculated as
$Sum^* = (Sum - EdgSum)/2$
For minimum/maximum the edges of the cycle/cut are first removed from the vertices in the segment tree by performing edge updates. These are later added back to the segment tree to restore the data.

Hence, following theorem can be stated

Theorem 3.7.1. *Given an embedded planar graph G having n vertices, we can make a data structure D_{tt} in $O(n)$ time and using $O(n)$ space, which allows updating an element in $O(\log n)$ time, aggregating information and performing additive weight updates on elements(vertices,edges and faces) of a subgraph enclosed by a cycle or cut of size d in $O(d\log n)$ time.*

3.8 Minimum Spanning Tree on a subgraph of planar graph

Given an embedded planar graph G having n vertices and edges having non-negative weights. We intend to find a spanning tree such that the sum of edges on the spanning tree is minimum, for any subgraph enclosed by a cut of the planar graph subject to online weight updates on edges.

The main idea behind solving this is based on the solution of Epstien et al.[17] for maintaining dynamic Minimum spanning tree(referred as MST). The idea exploits two basic properties of MST namely Cycle property and the Cut property.

Lemma 3.8.1. *(Cycle Property) Given a cycle C in a graph G, the edge on C whose weight is strictly larger than weights of all other edges on C cannot be a part of the MST of G.*

Lemma 3.8.2. *(Cut Property) Given a cut C^* in a graph G, the edge on C^* whose weight is strictly smaller than weights of all other edges on C^* will be a part of MST of G.*

We propose an algorithm to find the MST for any subgraph enclosed by a cut of an embedded planar graph G using data structure D_{tt}. Since spanning tree is

based on vertices of G, MST will be maintained on the primal graph and hence T_p. Firstly, instead of choosing any spanning trees for making T_p and T_d, we choose the MST of G to be the spanning tree on which T_p is made. Secondly, we modify the *Expose* and *Unexpose* operations, such that when ever *Expose* is called on a cut C^*, the modified T_p represents the MST of the subgraph enclosed by the cut C^*.

In the Expose algorithm (refer to Algortihm 3.1) exposing the dual graph on a cycle C, when a Cut is performed on an edge e, its replacement edge is found from the primal graph. Any unmarked edge on T_p is chosen to be the replacement edge.

Here we expose the primal graph on a cut C^*, hence the replacement edge is found on T_d. However, instead of choosing any unmarked edge, the minimum weighted unmarked edge on the cycle C_d formed by adding e to T_d is chosen as the replacement edge. Also for performing $Unexpose$, the MST has to be restored for the whole graph, so for each edge e_d of T_d on cut C^*, we check whether it is the maximum weight edge in the cycle formed by adding e_d to T_p. If not, e_p is added and the corresponding maximum weighted edge is removed from the MST and hence T_p.

3.8.1 Modified Expose Operation

Since *Expose* is performed on a cut, T_p will be divided into two trees namely T_p and T_p^* representing internal and external of the cut respectively. All the edges of C^* will be present in T_d except the extra edge e^* that connects T_p and T_p^* to make the complete spanning tree of the primal graph.

The difference in implementation of Expose operation lies in operation to find the replacement edge $e_{tmp} \in T_d$ for all edges $e \in C^*$ that were originally in T_p. Recall that all edges of C^* on T_d are marked so that they are not selected as replacement edge. Instead of selecting any unmarked edge we now select the unmarked edge having the minimum edge because of the Cut property described in Lemma 3.8.2. To find this replacement edge efficiently, top tree T_d is used by storing some data on T_d as explained.

Data Stored on each cluster of T_d

Each cluster stores the available unmarked edge on the cluster path having minimum weight in e_{um}. The value of e_{um} is $NULL$ in case no unmarked edge is present on the cluster path. The weight of the minimum weighted unmarked edge e_{um} is stored on d_{um}.

Data manipulation by internal operations

A base cluster on edge made by $Create$ is initialised according to mark flag on the edge as follows.

1. For an unmarked edge e with weight x
   $e_{um} = e$, $d_{um} = x$

2. For a marked edge e with weight x
   $e_{um} = NULL$, $d_{um} = \infty$

Cluster C made by $Join$, on clusters A and B having cluster paths xy and yz respectively, d_{um} is initialised as follows initialising e_{um} accordingly.

1. For a compress operation with xz as the cluster path of C
   $d_{um}(C) = \min(d_{um}(A), d_{um}(B))$

2. For a reke operation with xy as the cluster path of C
   $d_{um}(C) = d_{um}(A)$

No change is performed on data for split and delete.

Algorithm

Algorithm is same as original implementation of Expose on a cut C^* mentioned in Section 3.5.1.

3.8.2 Modified Unexpose Operation

Earlier $Unexpose$ was simply performed by adding the edge e^* to merge the trees T_p and T_p^*, restoring the spanning tree of G. However, for restoring a MST of G each edge on the exposed cut C^* is checked for its presence in the MST. Hence for an edge e on C^* present in T_d, we find the maximum weighted edge e_{max} on the cycle C_p formed by adding e to T_p. If weight of e_{max} is greater than the weight of e, by Lemma3.8.1 e_{max} will not be a part of MST. So we remove e_{max} from MST and add e to the MST, and hence T_p. To perform this operation efficiently some data is added to each cluster of T_p as explained.

Data Stored on each cluster of T_p

Each cluster stores the edge on the cluster path having maximum weight in e_{max}. The weight of the maximum weighted edge e_{max} is stored on d_{max}.

Data manipulation by internal operations

A base cluster on edge e, made by $Create$ is initialised with e as e_{max} and $wt(e)$ as d_{max}.

Cluster C made by $Join$, on clusters A and B having cluster paths xy and yz respectively, d_{max} is initialised as follows initialising e_{max} accordingly.

1. For a compress operation with xz as the cluster path of C
   $d_{max}(C) = \max(d_{max}(A), d_{max}(B))$

2. For a reke operation with xy as the cluster path of C
   $d_{max}(C) = d_{max}(A)$

No change is performed on data for split and delete.

Algorithm

Algorithm 3.2: Unexposing D_{tt} earlier exposed on a cut C^* on the primal graph G

Input : Planar Graph embedding G having a cut C^* in the primal graph, with the data structure D_{tt} built on G which is exposed on the cut C^*.
Output: T_p: Top tree on primal graph G
T_d: Top tree on dual graph G^*

```
Perform Link(e*) on T_p;
foreach edge e(x, y) ∈ C*, with x internal to C* do
    if e ∈ T_d then
        Expose T_p on path xy;
        Access(x) and e_tmp ← e_max, d_tmp ← d_max;
        if wt(e) < d_tmp then
            Cut(e_tmp) on T_p;
            Cut(e) on T_d;
            Link(e_tmp) on T_d;
            Link(e) on T_p;
        end
    end
end
```

In Algorithm 3.2, firstly *Link* is performed on e^* to link T_p and T_p^*. Now each edge on cut C^* is checked for its inclusion in MST using the cycle property described in Lemma 3.8.1.

3.8.3 Updating edge weight

This operation is performed only on unexposed D_{tt}. Updates on edge weights are handled similar to checking for inclusion or exclusion of an edge in *Expose* and *Unexpose*. It uses the data stored on clusters of T_p and T_d as described earlier.

As described in Algorithm 3.3, if the weight of an edge present in MST is decreased, it still remains to be a part of MST. Similarly if the weight of an edge not present in MST is increased, it still remains excluded from the MST. However, their weights are updated on the clusters of T_p and T_d respectively. But if the weight of an

edge of MST is increased, or the weight of an edge not present in MST is decreased, it is checked for its inclusion or exclusion from MST as described in Algorithm 3.3.

Algorithm 3.3: Updating the weight of an edge e from $wt(e)$ to $wt^*(e)$.

Input : Planar Graph embedding G , with the data structure D_{tt} built on G which is not exposed on any cycle or cut.

Output: T_p: Top tree on primal graph G representing the new MST.

```
if e(x, y) ∈ T_p then
    Expose T_d on path xy;
    Access(Face adjacent to e)) and e_tmp ← e_um, d_tmp ← d_um;
    if wt*(e) > d_tmp then
        Cut(e_tmp) on T_d;
        Cut(e) on T_p;
        Update weight of e;
        Link(e_tmp) on T_p;
        Link(e) on T_d;
    else
        Expose T_p on path xy(Edge);
        Update weight of e;
        Access(x) and d_min ← wt*(e)
    end
else
    Expose T_p on path xy;
    Access(x) and e_tmp ← e_max, d_tmp ← d_max;
    if wt*(e) < d_tmp then
        Cut(e_tmp) on T_p;
        Cut(e) on T_d;
        Update weight of e;
        Link(e_tmp) on T_d;
        Link(e) on T_p;
    else
        Expose T_d on path xy(Edge);
        Update weight of e;
        Access(Face adjacent to e) and d_um ← wt*(e)
    end
end
```

3.8.4 Analysis

In *Expose* and *Unexpose* operations, for each edge on the cut C^* of size d, constant number of *Link* and *Cut* operations are used, each taking $O(\log n)$ time. Hence *Expose* and *Unexpose* can be performed in $O(d\log n)$ time.

To update the weight on an edge constant number of *Link* and *Cut* operations are used. Hence it can be performed in $O(\log n)$ time.

Hence, following theorem can be stated

Theorem 3.8.3. *Given an embedded planar graph G having n vertices, we can make a data structure D_{tt} in $O(n)$ time and using $O(n)$ space that maintains a top tree T_d on the MST of G, allowing updates on weight of an edges in $O(\log n)$ time and finding the MST of a subgraph enclosed by a cut of size d in $O(d \log n)$ time.*

Chapter 4

Nearest Marked Vertex in Planar Graph

4.1 Introduction

Nearest Marked Vertex problem is a variant of Shortest Path problem which deals with dynamically marking/unmarking any vertex and querying the nearest marked vertex of any given vertex in an online fashion. The problem defines two *update* operations namely mark and unmark, and a *query* operation for querying nearest marked vertex.

Marked ancestor problem for trees is an extensively studied problem and finds application is solving many other theoretical problems[5]. The problem is then extended to Nearest marked vertex problem in trees and solved by Alstrup et. al [1] using top trees. It supports dynamic link and cuts, performing all operations in $O(\log n)$ time. They also gave a solution for Nearest marked for a fixed undirected graph on n vertices and m edges. They present a $2k-1$ approximate solution using $O(kn^{1+1/k})$ space data structure built in $O(kmn^{1/k}\log n)$ expected time for any positive integer parameter k, supporting both queries and updates in $O(kn^{1/k}\log n)$ time.

Given an embedded planar graph G having n vertices with infinite face f_∞ having d vertices, our aim is to be able to dynamically *mark* and *unmark* vertices such that for any vertex $v \in f_\infty$ we can *query* the nearest marked vertex in an online fashion. The problem can be extended to find the k nearest marked vertices for a given vertex.

Two initial approaches for solving the same problem using some existing techniques and data structures are discussed in section 4.5. It also compares the bounds achieved by our final solution with that of these solutions.

We define a parameter $S \in [1, d]$, such that our algorithm performs the update operations in $O(S\log n)$ time and query in $O(n\log n/S)$ time using $O(nS)$ space. Also

it supports k queries in $O((n/S + k)\log n)$ time.

The solution is based on results presented by Klien [3] for Multiple Source Shortest Path problem in planar graphs as explained in 2.4. They described a method to make shortest path tree (referred as SPT) of each vertex on f_∞ from the SPT of its preceding vertex on f_∞, such that total edge changes for making all the SPTs is at most $O(n)$. We store the SPTs of some selected vertices named as *Rooted Vertices*, such that we can make the SPT of any vertex on f_∞ from the SPT of a rooted vertex using $O(n/S)$ edge changes in the worst case. For each of the rooted vertex we maintain a top tree on its SPT and perform marking and unmarking of vertices as the updates are performed on the graph. The solution of Nearest marked vertex in trees by Alstrup [1] is used as explained in Section 2.3.4. We claim that we can achieve these bounds using S rooted vertices.

A solution with equal bounds for both updates and queries is obtained by taking S as $\sqrt{n}$ resulting in both updates and query to be done in $O(\sqrt{n}\log n)$ time using $O(n\sqrt{n})$ space. This result takes same time as one by Alstrup et. al[1] for general graphs, giving 3 approximation of the solution. We have given an exact solution for planar graph with restriction that query can be done only on vertices of f_∞.

We begin by giving an overview of how the result was obtained and the intuition behind the approach we used in Section 4.2. We then state and prove a property that limits the number of edge changes in making of the SPTs as we go around the infinite face of a planar graph in Section 4.3. Also we describe the relation of solution of nearest marked vertex on planar graphs with that on trees in Section 4.4. We then present some initial approaches using existing techniques to solve the same problem and compare it with our final solution in Section 4.5. Then we present our final solution for solving nearest marked vertex problem in planar graphs in Section 4.6. We further extend the solution to find k nearest marked vertices in planar graphs modifying the query algorithm to find k nearest marked neighbours in Section 4.7.

4.2 Overview

Initial approaches for the problem did not perform well for either updates or queries. Our main aim is to provide a solution that would perform well for both queries and updates. We establish a relationship of nearest marked vertex problem for planar graphs and for trees, as a result instead of solving the problem for whole graph, we can simply solve the problem on SPT of the query vertex. For solving the problem of nearest marked vertex on tree we can use the solution explained in Section 2.3.4 using top trees.

Another important observation exploited in our approach is the limited edge changes lemma. Klien [3] established the limited edge changes lemma (discussed in

Section 2.4) claiming that, given any two vertices $u, v \in f_\infty$ we can make SPT_v from SPT_u using $O(n)$ edge changes in worst case. Hence making a top tree on SPT_u, we can make SPT_v for all $v \in f_\infty$ using $O(n)$ link and cuts each requiring $O(\log n)$ time. Once the top tree is made we can query in $O(1)$ time to find the nearest marked vertex, and in $O(\log n)$ time the next nearest marked vertex and so on. Hence, by storing a single top tree on any SPT_u where $u \in f_\infty$, we can solve the problem.

However, this solution performs well only for updates, and uses $O(n\log n)$ time for query (to perform $O(n)$ link and cuts). So the idea was to store more than one top trees at equal intervals (in terms of number of edge changes), such that query time can be reduced thereby increasing update time. Hence, we define a parameter S denoting the number of trees stored, which can be varied from 1 to d for best case of update and query respectively. A good choice of S is $\sqrt{n}$ as it allows both updates and query in the same time which is $O(\sqrt{n}\log n)$.

So we start at any vertex labelling it as a *rooted vertex* and making a SPT on it. We go on making the SPT of the next vertex on f_∞ using a series of edge changes on the previous SPT determined by MSSP algorithm. Clearly, if we take equal interval of edge changes between every two of the S rooted vertices, the number of edge changes for making SPT of any vertex from the rooted vertex preceding it will be bounded by $O(n/S)$.

4.3 Bounded Edge changes property

The main idea used in our various approaches to solve the problem is the one presented by *Limited Edge Changes* lemma, described in Section 2.4. If we select S vertices from the infinite face to be rooted vertices and maintain shortest path trees on them as described in Section 4.2, then the following result holds

Lemma 4.3.1. *Given an embedded planar graph G with infinite face f_∞ of size d. We can maintain shortest path trees on $S \in [1, d]$ rooted vertices such that shortest path tree on any other vertex on f_∞ can be obtained using $O(n/S)$ edge changes to shortest path tree on preceding rooted vertex.*

Proof. We start with any vertex and mark it a rooted vertex. Now we move around the infinite face counting number of edge changes from previous rooted vertex. This can be done efficiently using the information provided by MSSP algorithm discussed in Section 2.4. Let m be the number of directed edges of the planar graph. When the count of edge changes exceeds m/S we mark the current vertex as *rooted*, reset the count to zero and proceed with the algorithm till all vertices are covered.

Now a new rooted vertex will appear after m/S edge changes and the number of edge changes are bounded by m by *Limited Edge Changes* lemma described in

Section 2.4. So the number of rooted vertices are

$$m/(m/S) = S \tag{4.1}$$

Also by the sparsity property of planar graphs the number of edges m, of a planar graph is bounded by $O(n)$ hence edge changes between two rooted vertices are

$$O(n)/S = O(n/S) \tag{4.2}$$

Hence using S rooted vertices we can find shortest path tree for an vertex on infinite face with $O(n/S)$ edge changes if we choose the next rooted vertex after an edge change interval of m/S. □

Choosing S as $\sqrt{n}$ we get the following result:

Corollary 4.3.2. *Given an embedded planar graph G with infinite face f_∞. We can maintain shortest path trees on $\sqrt{n}$ rooted vertices such that shortest path tree on any other vertex on f_∞ can be obtained using $O(\sqrt{n})$ edge changes to shortest path tree on some rooted vertex.*

4.4 Nearest marked vertex property

Maintaining Nearest Marked Vertex on a tree is described in Section 2.3.4. If we can somehow reduce the graph to a tree such that the result of the nearest marked vertex query does not change, we can solve the problem using a top tree. We claim that Shortest Path Tree on the query vertex can be considered for such a reduction.

Lemma 4.4.1. *Given an embedded planar graph G having a vertex v, let T be the shortest path tree on v in G. Then the nearest marked vertex of v in the G is same as the nearest marked vertex of v in T.*

Proof. Since T is the shortest path tree, it is also a spanning tree hence it contains all vertices of the connected graph. Now, if there exists a marked vertex in G that is connected to v, then it is also present in T and hence connected to v. Similarly, if there is no marked vertex connected to v in G, no marked vertex will be present in T. Hence the result of the query holds true for the existence of a marked vertex, i.e. if there is zero or one marked vertex.

Let there be at least two marked vertices x and y, such that x is the nearest marked vertex of v in G and y is the nearest marked vertex of v in T. We claim that x and y are either same or at equal distance from v. Since x is the nearest marked vertex of v in G, the minimum distance from v to reach x and y are related as

$$dist_{min}(v, x) \leq dist_{min}(v, y) \tag{4.3}$$

Also, since T is the shortest path tree, path from v to both x and y are shortest paths. Further, the nearest marked vertex of v on T is y so

$$dist_{min}(v, x) \geq dist_{min}(v, y) \tag{4.4}$$

Hence, we have

$$dist_{min}(v, x) = dist_{min}(v, y) \tag{4.5}$$

It implies that either $x = y$ or both x and y can be considered as the nearest marked vertex of v in both T and G. □

4.5 Initial approaches

Given an embedded planar graph G with n vertices and infinite face f_∞ with d vertices, we intend to mark/unmark any vertex and report the nearest marked vertex for any vertex $v \in f_\infty$ in an online fashion. We present two solutions for the problem, first is a variant of Dijkstra's algorithm and second based on segment tree.

First solution uses Dijkstra's algorithm. It enables us to mark or unmark vertex in constant time. It uses O(n) extra space to store a flag array for vertices, setting the flag to represent a marked vertex. To mark or unmark we simply modify the corresponding flag. For query we use Dijkstra's algorithm in which each time a vertex is added to shortest path tree, it is checked for being marked using its flag's status. We report the first marked vertex added to the shortest path tree. For planar graphs, the sparsity lemma limits the number of edges to $O(n)$, hence the time taken for applying Dijkstra's algorithm and hence query is $O(n\log n)$. We can go on searching for next marked vertex for finding k nearest marked vertices in $O(n\log n)$ time.

Second solution uses a segment tree for each vertex on f_∞ allowing us to query for nearest marked vertex in constant time. It uses d segment trees and two arrays requiring $O(dn)$ extra space. Firstly, we make d flag arrays of size n, storing for each vertex on $v \in f_\infty$ all the vertices in order of their distance from v, setting the flag to represent a marked vertex. We make a segment tree on each of these flag arrays, whose nodes represents the marked vertex which is leftmost (or nearest to v) in its subtree. Hence the root of segment tree of vertex v stores the nearest marked vertex of v. Secondly, we make n index arrays of size d for each vertex $u \in V$ of the graph, storing the index of u in the d flag arrays. These structures can be initialised using $O(dn\log n)$ preprocessing time and $O(dn)$ space. Now, to mark/unmark a vertex u we use its index array to modify the corresponding flag and update the segment tree of every vertex $v \in f_\infty$, using $O(\log n)$ time per update taking $O(d\log n)$ time. To query for a vertex v we report the value of the root of its segment tree in $O(1)$ time. For k queries we go on unmarking the nearest marked vertex in the segment tree

of the query vertex v to get the next nearest marked vertex. All these vertices are later marked again to restore information in the segment tree of v. Thus query for k nearest marked vertices can be done in $O(k\log n)$ time.

Our solution has a parameter $S \in [1, d]$, such that the updates are supported in $O(S\log n)$ time and each query is answered in $O(1)$ time for S vertices and $O(n\log n/S)$ in the worst case. The bounds set by the initial approaches are met at the two extreme values of S. However, the major drawback of the two initial approaches is that they perform very badly for either update or query. Our solution can perform both operations in sublinear time using $S = \sqrt{n}$ as described in Table 4.1 comparing the solutions. Note that for large value of d, both the initial algorithms proves inefficient for either query or updates.

Table 4.1: Comparison of different algorithms

Algorithm	Mark/Unmark	Query	k-Query	Space
Dijkstra's Variant	$O(1)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Using Segment Trees	$O(d\log n)$	$O(1)$	$O(k\log n)$	$O(dn)$
Our Result (for any S)	$O(S\log n)$	$O(n\log n/S)$	$O((n/S+k)\log n)$	$O(Sn)$
Our Result ($S = \sqrt{n}$)	$O(\sqrt{n}\log n)$	$O(\sqrt{n}\log n)$	$O((\sqrt{n}+k)\log n)$	$O(n\sqrt{n})$

4.6 Nearest Marked Vertex in Planar Graphs

We first describe the data structure developed to solve the problem followed by the details of the dynamic operations. The data structure is made by selecting S rooted vertices from f_∞ based on the property described by Lemma 4.3.1. For each rooted vertex we store shortest path tree (referred as SPT) and make a top tree on it as described in Section 2.3.4 to solve nearest marked vertex problem in a tree. To update we mark (or unmark) the vertex on each top tree and for query we generate the top tree on SPT of the query vertex and query on that top tree.

4.6.1 Algorithm

This section presents the algorithm to mark, unmark and query the nearest marked vertex of a planar graph embedding G, having m directed edges and n vertices, with the infinite face f_∞ of size d having vertices indexed as $v_1, v_2, ..., v_d$, using a data structure D_{nmv} that allows us to perform these operations efficiently. The updates and queries are supported in an online fashion.

Data structure

We present a data structure D_{nmv} to support query of nearest marked vertex and mark (or unmark) any vertex efficiently in an online fashion. Following are the main

parts of the data structure:

1. **Rooted Vertices:** We select upto S of the d vertices of f_∞ as *rooted vertices*. We maintain an array of the indices of these rooted vertex in RV. The count of the rooted vertices is stored in rvc.

2. **Top trees on rooted vertices:** We maintain a top tree TT_r for SPT_r of each rooted vertex $v_r \in RV$ with a structure to solve nearest marked vertex problem in trees.

3. **Edge Change Information:** To transform SPT of a rooted vertex to that of any other vertex we may need to add and remove some edges. Hence, we store the list of edges that are added and deleted when we transform SPT_{i-1} to SPT_i in $EdgAdd_i$ and $EdgDel_i$ respectively, for each vertex $v_i \in f_\infty$.

Developing the data structure D_{nmv} for a given planar graph embedding is described in Algorithm 4.1. Firstly the $EdgAdd$ and $EdgDel$ lists are made using the MSSP algorithm as described in Section 2.4. Also we need to select S rooted vertices on f_∞, such that the number of changes $EdgAdd$ or $EdgDel$ for any vertex from the preceding rooted vertex does not exceed $O(n/S)$. We use Lemma 4.3.1 to select these rooted vertices. We begin by selecting the first vertex as the rooted vertex and storing its index in RV. Then we go on marking the vertex that exceeds m/S edge changes from the last rooted vertex, as the next rooted vertex storing the indices of each rooted vertex in RV. A variable rvc maintains a counter for number of root variables encountered.

We use a temporary Top Tree T on SPT_1 with structure to solve nearest marked vertex problem as described in Section 2.3.4. We modify T as we move along the face with edge change operations saving a copy of the top tree for each rooted vertex v_r in TT_r. A counter c is used to keep a count on number of edge changes encountered. If c is greater than m/S we mark the vertex as a rooted vertex. However, we maintain the count of changes c only for edge additions as in a spanning tree the total number of edges has to remain $n-1$, hence an edge change corresponds to a single edge addition and edge deletion. Also note that $Cut(e)$ is performed before $Link(e)$ because we cannot $Link$ two vertices belonging to the same tree.

We also maintain a variable $prev$, which stores the last vertex which resulted in an edge change, in T. This is required to make the $Next$ array to deal with a special case in Algorithm 4.2 for bounding the number of operations in updates.

Algorithm 4.1 uses MSSP algorithm described in Section 2.4 using $O(n\log n)$ time and $O(n)$ space to store each $EdgAdd$, $EdgDel$ and SPT_1. The operations $MakeCopy$ makes a copy of a given top tree and $MakeTopTree$ makes a Top Tree on a given tree, each taking $O(n)$ time. By *Limited Edge Changes* lemma we know that edge changes are bounded by $O(n)$ over the entire algorithm, hence both $Link$

and Cut are performed $O(n)$ times taking $O(\log n)$ time per operation resulting in $O(n\log n)$ time. Most expensive operation is MakeCopy when performed for each rooted vertex. As proved earlier, the number of rooted vertices are bounded by S, performing MakeCopy S times will take $O(nS)$ time. Hence the overall Algorithm 4.1 is performed in $O(nS)$ time. Also TT uses $O(nS)$ space with $EdgAdd$ and $EdgDel$ each using $O(n)$ space. Therefore the total space required for D_{nmv} is $O(nS)$, taking $O(nS)$ time for pre-computation.

Algorithm 4.1: Making the data structure D_{nmv}

Input : Planar Graph embedding G having m directed edges with infinite face f_∞ of size d with vertices indexed $1, 2, ..., d$ and a parameter $S \in [1, d]$

Output: RV : Array of indices of rooted vertices in clockwise order
TT_v : Top tree on shortest path tree of each vertex $v \in RV$
rvc : Number of rooted vertices
$Next$: Array storing for each vertex $v_i \in f_\infty$ the next vertex v_j such that $SPT_i \neq SPT_j$

```
Top Tree T , rvc ← 0, c ← 0, prev ← 1;
Use MSSP to find SPT_1 and populate EdgAdd_i, EdgDel_i ∀v_i ∈ f_∞;
T ← MakeTopTree(SPT_1);
RV[rvc] ← 1;
TT_rvc ← MakeCopy(T);
rvc ← rvc + 1;
for i ← 2 to d do
    if EdgAdd_i ≠ NULL then
        Next[prev] ← i;
        prev ← i;
    end
    foreach edge e ∈ EdgDel_i do Perform Cut(e) on T, c ← c + 1 foreach
    edge e ∈ EdgAdd_i do Perform Link(e) on T
    if c > m/S then
        RV[rvc] ← i;
        TT_rvc ←MakeCopy(T);
        c ← 0, rvc ← rvc + 1;
    end
end
```

Dynamic Operations and Query

Both the operations of update (mark/unmark) and query are are performed on the top trees TT_i as explained in Section 2.3.4. For update, we have to update top tree of each rooted vertex, whereas for query, we first generate the top tree on the SPT of the query vertex and then query on it.

To Mark(or Unmark) a vertex, we mark(or unmark) the information on the top

trees of all rooted vertices. Algorithm 4.2 describes answering a query on a vertex v_x. It is performed by modifying the top tree on the closest rooted vertex v_r preceding v_x to obtain the top tree for the SPT_x. It can be achieved by performing Link and Cut on the edges present in $EdgAdd_i$ and $EdgDel_i$ respectively for each $i \in (r, x]$. This will take at least $O(x-r)$ time. However, $x-r$ can be greater than $O(n/S)$, so we skip the vertices that don't change the tree using $Next$ array. Hence the vertices traversed are bounded by $O(n/S)$ as each vertex corresponds to at least one edge change. After query we revert back the top tree. Notice that we have not copied the top tree and then modified it, though it will allow us to avoid reconstructing the older tree but the copying part may take $O(n)$ time.

Algorithm 4.2: Query a vertex v_x on f_∞

Input : Planar Graph embedding G with infinite face f_∞ and a vertex v_x on f_∞ to be queried with structure D_{nmv} on G

Output: Nearest Marked Vertex v_y of the vertex v_x

$r \leftarrow$Binary Search of largest index less than or equal to x in RV;
$TempAdd \leftarrow NULL, TempDel \leftarrow NULL$

for $i \leftarrow Next[r]$ ***where*** $i \leq x$ **do**
  foreach *edge* $e \in EdgDel_i$ **do**
    Cut(e) on TT_r;
    if $e \in TempAdd$ **then** Remove e from $TempAdd$;
    else Add e to $TempDel$
  end
  foreach *edge* $e \in EdgAdd_i$ **do**
    Link(e) on TT_r;
    if $e \in TempDel$ **then** Remove e from $TempDel$;
    else Add e to $TempAdd$
  end
  $i \leftarrow Next[i]$;
end

$y \leftarrow$Query for Nearest Marked vertex of v_x on TT_r;

foreach *edge* $e \in TempAdd$ **do** Cut(e) on TT_r **foreach** *edge* $e \in TempDel$ **do** Link(e) on TT_r return y;

4.6.2 Analysis

Marking and Unmarking a vertex in a top tree takes $O(\log n)$ time as explained in Section 2.3.4. Since we have to update top trees of S rooted vertices, performing mark/unmark takes $O(S\log n)$ time.

Query of a vertex performs $O(n/S)$ *Link* and *Cut* operations along with a *Query* operation on a top tree taking $O(\log n)$ time per operation, hence $O(n\log n/S)$ time. Had we made a temporary copy of the top tree and performed the updates on it

because the size of top tree is $O(n)$ hence it will be copied in $O(n)$ time. Where as modifying and reverting it will take $O(n/S)$ steps each of $O(\log n)$ time.

Hence we can now state the following theorem.

Theorem 4.6.1. *Given an embedded planar graph G having n vertices with infinite face f_∞ with d vertices and a parameter $S \in [1, d]$ we can make a data structure D_{nmcv} in $O(nS)$ time and using $O(nS)$ space which can allow marking and unmarking of any vertex in the graph in $O(S\log n)$ time and query of nearest marked vertex on vertices of f_∞ in $O(n\log n/S)$ time.*

Choosing S as $\sqrt{n}$ for balancing the time complexity for queries and updates leads to following result:

Corollary 4.6.2. *Given an embedded planar graph G having n vertices with infinite face f_∞ with d vertices, we can make a data structure D_{nmcv} in $O(n\sqrt{n})$ time and using $O(n\sqrt{n})$ space which can allow marking and unmarking of any vertex in the graph in $O(\sqrt{n}\log n)$ time and query of nearest marked vertex on vertices of f_∞ in $O(\sqrt{n}\log n)$ time.*

4.7 k Nearest Marked Vertices in Planar Graphs

This is an extension of previous result where instead of querying for one nearest marked vertex we can query for k nearest marked vertices. To answer a query in a top tree, we first build the top tree on the query vertex in $O(n\log n/S)$ time and then query for nearest marked vertex in it in $O(\log n)$ time. Underlying idea involved is saving the time to construct the top tree separately for each of the k queries. For a single query we derive the top tree for the query vertex from the top tree on preceding rooted vertex. For every next query we don't need to construct the tree again, hence query can be done efficiently. It uses the same data structure D_{nmv} and update algorithm, with just a variation in the query algorithm.

In Algorithm4.3, after the first query we unmark the nearest marked vertex. Then on querying again it reports the next nearest marked vertex. This process of unmark and query is repeated for each query. Each nearest marked vertex is stored in a list NM_k, which is later also used to restore the top tree by marking the vertices in NM_k back again.

The analysis of the result is same as that of previous result, with an exception of the query algorithm. Since the query adds $O(k)$ mark and unmark operations on the top tree T, it uses extra $O(k\log n)$ time. Hence the query of k nearest marked vertices can be performed in $O((n/S + k)\log n)$ time. Note that it does not change the bound for $k = 1$ which remains to be $O(n\log n/S)$.

Hence we can now state the following theorem

Theorem 4.7.1. *Given an embedded planar graph G having n vertices with infinite face f_∞ with d vertices and a parameter $S \in [1, d]$, we can make a data structure D_{nmcv} in $O(nS)$ time and using $O(nS)$ space which can allow marking and unmarking of any vertex in the graph in $O(S\log n)$ time and supporting query of k nearest marked vertices from a vertex on f_∞ in $O((n/S + k)\log n)$ time.*

Choosing S as $\sqrt{n}$ for for balancing the time complexity for queries and updates leads to following result:

Corollary 4.7.2. *Given an embedded planar graph G having n vertices with infinite face f_∞ with d vertices, we can make a data structure D_{nmcv} in $O(n\sqrt{n})$ time and using $O(n\sqrt{n})$ space which can allow marking and unmarking of any vertex in the graph in $O(\sqrt{n}\log n)$ time supporting query of k nearest marked vertices from a vertex on infinite face f_∞ in $O((\sqrt{n} + k)\log n)$ time.*

Algorithm 4.3: Query a vertex v_x on f_∞ for k nearest marked vertices

Input : Planar Graph embedding G with infinite face f_∞ and a vertex v_x on f_∞ to be queried and a constant $k \leq n$ with structure D_{nmv} on G

Output: NMV: k Nearest Marked Vertices of the vertex v_x

$r \leftarrow$Binary Search of largest index less than or equal to x in RV;
$TempAdd \leftarrow NULL, TempDel \leftarrow NULL$

for $i \leftarrow Next_r$ ***where*** $i \leq x$ **do**
  foreach *edge* $e \in EdgDel_i$ **do**
    Cut(e) on TT_r;
    if $e \in TempAdd$ **then** Remove e from $TempAdd$;
    else Add e to $TempDel$
  end
  foreach *edge* $e \in EdgAdd_i$ **do**
    Link(e) on TT_r;
    if $e \in TempDel$ **then** Remove e from $TempDel$;
    else Add e to $TempAdd$
  end
  $i \leftarrow Next_i$;
end

for $i = 1$ *to* k **do**
  $NMV_i \leftarrow$ Query for Nearest Marked vertex of v_x on TT_r;
  Unmark NMV_i on TT_r;
end
for $i = 1$ *to* k **do** Mark NMV_i on TT_r;

foreach *edge* $e \in TempAdd$ **do** Cut(e) on TT_r **foreach** *edge* $e \in TempDel$ **do** Link(e) on TT_r

Chapter 5

Conclusions

5.1 Impact

The two problems we solved, explore the applications of top trees on planar graphs.

The possible implementation of top trees is described by Alstrup et. al[1] and Werneck[2]. The top tree on a graph G having n vertices can be made in $O(n)$ time using $O(n)$ space. They have implemented each of the three update operations that change the structure of top tree, namely *Link*, *Cut* and *Expose* in $O(\log n)$ time. *Expose* operation on a cluster path, takes two vertices(ends of cluster path) as input and exposes them to the root cluster, taking $O(\log n)$ time for each vertex.

We describe the first attempt to make a generalised data structure similar to top trees for planar graphs. Our solution performs all the external operations using time same as that of corresponding operations in top trees. Only exception is *Expose* on a cycle or cut of size d that is performed in $O(d\log n)$ time. Note that the size of input for *Expose* is d, hence any algorithm for this problem will require $\Omega(d)$ time. Also expose operation of top trees takes $O(\log n)$ time for each of the two boundary vertices. Similarly *Expose* of D_{tt} requires $O(\log n)$ time for each edge on the enclosing cycle or cut of size d. Hence the bounds achieved for implementing D_{tt} are competitive with the current state of the art. Trivially both the applications can be solved in $O(n)$ time, hence our solution performs much better if d very small as compared to n.

Our second problem deals with nearest marked vertex problem on planar graphs. The problem of nearest marked vertex on planar graphs is much more complex as compared to trees, also it is fairly less complex than that on general graphs. For trees, Alstrup et al[1] solved the problem taking $O(\log n)$ time per update and query operations. For a general graph having n vertices and m edges, they present a $2k-1$ approximate solution using $O(kn^{1+1/k})$ space data structure built in $O(kmn^{1/k}\log n)$ expected time for any positive integer parameter k, supporting both queries and updates in $O(kn^{1/k}\log n)$ time.

Our balanced solution for planar graph requires $O(\sqrt{n}\log n)$ time per update or

query operation with a limit on the query vertices. Using same bound the solution by Alstrup et. al[1] for general graphs, gives 3 approximation of the solution. Also our initial approaches to solve the problem on planar graphs performs either update or query in $O(1)$ time, taking atleast $O(d\log n)$ time to solve the other operation. We can match bounds of both these solutions at the extreme values of our parameter S. Hence the bounds achieved for our final solution are competitive with the state of the art and direct approaches to solve the problem.

5.2 Limitations

The major limitation of the data structure D_{tt} is that it does not handle path problems. It is because all the edges are divided into two top tree T_p and T_d, and even though top trees can easily handle path problems, not all paths are covered by these top trees.

In case of faces using T_d, the tree contractions are not logically meaningful. The edges in the dual graph that are contracted are not adjacent, leaving limited scope of its application. In particular applications that are dependent on combining the data stored on clusters based on the relative location of clusters are not supported.

Also D_{tt} does not support fully dynamic updates, that is addition and deletion of edges. The main problem is that the top tree on dual graph is made using faces of the primal graph. And top tree does not support addition or deletion of vertices. Addition and deletion of edges leads to split and merge in faces, which is not supported by the underlying structure.

Main limitation of our solution for nearest marked vertex problem in planar graphs is that it answers queries for only vertices located on the outer face of the planar graph. Even though we can Mark/Unmark any vertex the query is limited to the vertices of the infinite face only.

Our algorithm doesn't cover it mainly because at the core of our result we have used MSSP algorithm by Klien [3]. Primarily we need a top tree on the shortest path tree of the vertex to be queried. But since the shortest path trees are not made on inner vertices we cannot make them easily with a limit on edge changes.

Secondly our algorithm is not fully dynamic and hence does not support updates as *Link* or *Cut*. Again since such updates were not handled by MSSP algorithm[3], we were not able to handle them.

5.3 Future Work

The base clusters in top trees are based on edges. It seems the data structure D_{tt} may become more useful if the base clusters can be based on vertices or faces.

Hence, exploring use of topology trees at the heart of this data structure instead of top trees may lead to significant results. Also fully dynamic updates on the graph can be supported if the faces are not represented as vertices by using top trees.

In the nearest marked vertex problem, main limitation is the limit on vertices that can be queried. Klien [3] have tried to overcome the problem of limit on queries using structures called Jordan Curves. It allows recursively applying the same algorithm to find the shortest path distance between any two vertices. It may be interesting to explore its use here. Also the algorithm may be extended to support fully dynamic updates i.e. Link and cut.

Bibliography

[1] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup: Maintaining information in fully dynamic trees with top trees. ACM Trans. Algorithms 1, 2 (October 2005), 243-264.

[2] Renato F. Werneck. Design and analysis of data structures for dynamic trees. PhD thesis, Princeton, NJ, USA, 2006. Adviser-Robert E. Tarjan.

[3] Philip N. Klein: Multiple-source shortest paths in planar graphs. In Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2005). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 146-155.

[4] Sergio Cabello and Erin W. Chambers: Multiple source shortest paths in a genus g graph. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '07). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 89-97.

[5] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe: Marked Ancestor Problems. 39th Annual Symposium on Foundations of Computer Science, FOCS '98 Proceedings, 1998, pages 534-543.

[6] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. Journal of Computer and System Sciences, 26(3):362-391, 1983.

[7] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. Journal of the ACM, 32(3):652-686, 1985.

[8] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pages 478-489, 1985.

[9] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In Pro- ceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 524-533. SIAM, 2004.

[10] U. A. Acar, G. E. Blelloch, and J. L. Vittes. An experimental analysis of change propagation in dynamic trees. In Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX), pages 41-54, 2005.

[11] G. N. Frederickson. Data structures for on-line update of minimum spanning trees, with applications. SIAM Journal of Computing, 14(4):781-798, 1985.

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

[13] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. Journal of Algorithms, 24(1):37-65, 1997.

[14] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarihmic time per operation. In Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC), pages 519-527, 1997.

[15] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarihmic time per operation. Journal of the ACM, 46(4):502-516, 1999.

[16] R. E. Tarjan. Dynamic trees as search trees via Euler tours, applied to the network simplex algorithm. Mathematical Programming, 78:169-177, 1997.

[17] David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert Endre Tarjan, Jeffery Westbrook, Moti Yung: Corrigendum: Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph. J. Algorithms 15(1): 173 (1993)

[18] Philip N. Klein, Shay Mozes, Oren Weimann: Shortest paths in directed planar graphs with negative lengths: A linear-space O(n log2 n)-time algorithm. ACM Transactions on Algorithms 6(2) (2010)