Distance Oracles for Planar Graphs

by

Gengchun Xu

M.Sc., University of Science and Technology of China, 2012
B.Sc., University of Science and Technology of China, 2009

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Science

Approval

Name:	**Gengchun Xu**
Degree:	**Doctor of Philosophy (Computing Science)**
Title:	**Distance Oracles for Planar Graphs**
Examining Committee:	**Chair:** Binay Bhattacharya Professor School of Computing Science
	Qianping Gu Senior Supervisor Professor School of Computing Science
	Ramesh Krishnamurti Supervisor Professor School of Computing Science
	Andrei Bulatov Internal Examiner Professor School of Computing Science
	Kshirasagar Naik External Examiner Professor Department of Electrical and Computer Engineering University of Waterloo
Date Defended:	**November 22, 2019**

Abstract

The shortest distance/path problems in planar graphs are among the most fundamental problems in graph algorithms and have numerous new applications in areas such as intelligent transportation systems which expect to get a real time answer or a distance query in large networks. A major new approach to address the new challenges is distance oracles which keep the pre-computed distance information in a data structure (called oracle) and provide an answer for a distance query with the assistance of the oracle. The preprocessing time, oracle size and query time are major criteria for evaluating two-phase algorithms. In this thesis, we first briefly review the previous work and introduce some preliminary results on exact and approximate distance oracles. Then we present our research contributions, which includes improving the preprocessing time for exact distance oracles for planar graphs with small branchwidth and providing the first constant query time $(1+\epsilon)$-approximate distance oracle with nearly linear size and preprocessing time for planar graphs.

Keywords: distance oracle; planar graph; algorithm; branch-decomposition

Dedication

Dedicated to my family.

Acknowledgements

First and foremost, I would like to thank my advisor Dr. Qian-Ping Gu, for his supervision throughout my study at SFU. Dr. Gu has always been supportive of my research and I could always get prompt and insightful feedback from him. Additionally, I would like to thank the rest of the examining committee: Dr. Ramesh Krishnamurti, Dr. Andrei Bulatov, Dr. Kshirasagar Naik and Dr. Binay Bhattacharya. Thank you all for taking your time to help improve this thesis. I am sincerely grateful to lab mates Leo Liang and Youjiao Sun who have helped me throughout my study. Special thanks to my parents and my husband. I would not have finished my thesis without their support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Finding the distance between two vertices in a graph is a fundamental computational problem and has wide range of applications. For this problem, there is a rich literature of algorithms. This problem can be solved by a single source shortest path algorithm such as the Dijkstra and Bellman-Ford algorithms. The running time in this case, however, may not be sufficiently small for certain applications. In many applications, it is required to compute the shortest path distance in an extreme short time. One approach to meet such a requirement is to use distance oracles.

A distance oracle consists of a data structure that keeps the pre-computed distance information and a query algorithm that provides the distance between any given pair of vertices efficiently. There are two phases in the distance oracle approach. The first phase is to compute the distances between vertices for a given graph G and a data structure (called *oracle*) to store the distances. The second phase is to provide an answer for a query on the distance between a pair of vertices in G based on the information in the data structure (answer the query by calling the oracle). The efficiency of distance oracles is mainly measured by three criteria: the time to create the data structure (*preprocessing time*), the time to answer a query (*query time*), and the memory space required for the data structure (*oracle size*). A simple approach to compute a distance oracle for a graph G of n vertices and m edges is to solve the all pairs shortest paths problem in G and keep the distances in an $n \times n$ distance array. This gives a simple oracle, the preprocessing time of the oracle is the time it takes to solve the all pairs shortest path problem (e.g., $O(mn + n^2 \log n)$), the query time is $O(1)$ and the oracle size is $O(n^2)$. The large oracle size is a major disadvantage of this approach. Another simple approach is to run a single source shortest path algorithm (e.g., Dijkstra's algorithm) to answer the query, which gives an oracle with no preprocessing time and $O(m)$ oracle size. But the query time in this approach is the running time of the single source shortest path algorithm that can be too large for a distance query. Typically, there is a trade-off between the query time and the oracle size. Main research topics include minimizing the product of the query time and oracle size, minimizing the query time subject to a given oracle size, and minimizing the oracle size subject to a given query time. Improving

Oracle	**Oracle size**	**Query time**	**Preprocessing time**
Djidjev [32]	$S \in [n, n^2]$	$O(n^2/S)$	$O(S)$ for $S \in [n^{3/2}, n^2]$ $O(n\sqrt{S})$ for $S \in [n, n^{3/2})$
Djidjev [32]	$S \in [n^{4/3}, n^2]$	$O(n \log(n/\sqrt{S})/\sqrt{S})$	$O(n\sqrt{S})$
Mozes and Sommer [65]	$S \in [n \log\log n, n^2]$	$\tilde{O}(n/\sqrt{S})$	$\tilde{O}(S)$
Cohen-Addad et al. [27]	$S \in [n^{3/2}, n^2]$	$O(\frac{n^{5/2}}{S^{3/2}} \log n)$	$\tilde{O}(S)$
Gawrychowski et al. [38]	$S \in [n, n^2]$	$\tilde{O}(\max\{1, n^{1.5}/S\})$	$\tilde{O}(\min\{S\sqrt{n}\}, n^2)$

Table 1.1: Exact distance oracles for planar digraphs with different trade-offs between the query time and the oracle size. S is the oracle size and $\tilde{O}(\cdot)$ notation hides the logarithmic factors.

the preprocessing time subject to a given oracle size and query time is also studied. Oracles with constant query time and size smaller than $O(n^2)$ have received much attention.

1.1 Related Works

1.1.1 Exact Distance Oracles for Planar Graphs

Planar graphs are an important model for many networks such as the road networks. In this thesis we focus on distance oracles for planar graphs. Distance oracles for planar graphs have been extensively studied. An exact distance oracle is a data structure which provides exact shortest path distances for queries.

Some important exact distance oracles for planar digraphs are listed in Table 1.1. Djidjev proves that for any size $S \in [n, n^2]$, there is an exact distance oracle with $O(n^2/S)$ query time for planar digraphs with non-negative edge lengths [32]. The preprocessing time is $O(S)$ for $S \in [n^{3/2}, n^2]$ and $O(n\sqrt{S})$ for $S \in [n, n^{3/2})$. There are several exact distance oracles for weighted planar digraphs with more efficient query times for smaller ranges of S. For example, Djidjev [32] improves the query time for size $S \in [n^{4/3}, n^2]$ to $O(n \log(n/\sqrt{S})/\sqrt{S})$. The preprocessing time is $O(n\sqrt{S})$ for this distance oracle. Mozes and Sommer show an oracle with $O((n/\sqrt{S}) \log^{2.5} n)$ query time, $O(S \log^2 n)$ preprocessing time for size $S \in [n \log\log n, n^2]$ [65]. This distance oracle works for a larger range of S than [32] without increasing the query time or preprocessing time when logarithmic factors are ignored. Cohen-Addad et al. give an oracle with $O(\log n)$ query time, $O(n^2)$ preprocessing time and $O(n^{5/3})$ size [27]. This is the first exact distance oracle with $O(\log n)$ query time and truly subquadratic (i.e. $O(n^{2-\epsilon})$ for some $\epsilon > 0$) size. They also show a distance oracle

Oracle	**Oracle size**	**Query time**	**Preprocessing time**
Thorup [78] Klein [59]	$O(n\epsilon^{-1}\log n)$	$O(1/\epsilon)$	$O(n\epsilon^{-2}\log^3 n)$
Thorup [78], directed	$O(n\epsilon^{-1}\log\Delta\log n)$	$O(1/\epsilon+\log\Delta)$	$O(n\epsilon^{-2}\log\Delta\log^3 n)$
Kawarabayashi et al. [51]	$O(n)$	$O(\epsilon^{-2}\log^2 n)$	$O(n\log^2 n)$
Kawarabayashi et al. [57]	$\bar{O}(n\log n)$	$\bar{O}(1/\epsilon)$	$\bar{O}(n\epsilon^{-2}\log^4 n)$
Wulff-Nilsen [83]	$O(n(\log\log n)^2/\epsilon+\log\log n/\epsilon^2))$	$O((\log\log n)^3/\epsilon^2+$ $\log\log n\sqrt{\log\log((\log\log n)/\epsilon^2)}/\epsilon^2)$	N/A

Table 1.2: $(1+\epsilon)$-Approximate distance oracles for planar graphs with different trade-offs between the query time and the oracle size. Δ is the largest finite distance between any pair of vertices in the directed graph. $\bar{O}$ hides $\log\log n$ factors and $\log(1/\epsilon)$ factors.

with oracle size $S \in [n^{3/2}, n^2]$, $O(\frac{n^{5/2}}{S^{3/2}}\log n)$ query time and $\tilde{O}(S)$ preprocessing time, which improves the previous trade-offs for a smaller range of S. This work is further improved by Gawrychowski et al. [38] which shows that there is an exact distance oracle for $S \in [n, n^2]$ with $\tilde{O}(\max\{1, n^{1.5}/S\})$ query time. This is currently the best trade-off for the entire range of S and the distance oracle can be constructed in $\tilde{O}(\min\{S\sqrt{n}\}, n^2)$ time. Readers may refer to Sommer's survey paper [75] for more information.

1.1.2 Approximate Distance Oracles for Planar Graphs

Approximate distance oracles have been developed to achieve fast query and near linear size for planar graphs. For any vertices u and v in a graph G, let $d_G(u,v)$ denote the distance between u and v. A distance oracle is called an (α,β)-approximate oracle for $\alpha \geq 1, \beta \geq 0$ if it provides a distance $\tilde{d}(u,v)$ with $d_G(u,v) \leq \tilde{d}(u,v) \leq \alpha d_G(u,v)+\beta$ for any u and v in G, here α is called the *(multiplicative) stretch* and β is called the *additive stretch*. An oracle with stretch α and additive stretch 0 is called an α-approximate distance oracle.

Some important $(1+\epsilon)$-approximate distance oracles are listed in Table 1.2. For $\epsilon > 0$, Thorup gives a $(1+\epsilon)$-approximate distance oracle with $O(1/\epsilon)$ (resp. $O(1/\epsilon+\log\Delta)$, where Δ is the largest finite distance between any pair of vertices in G) query time, $O(n\epsilon^{-1}\log n)$ (resp.$O(n\epsilon^{-1}(\log\Delta)\log n)$) size and $O(n\epsilon^{-2}(\log n)^3)$ (resp. $O(n\epsilon^{-2}(\log\Delta)(\log n)^3)$) preprocessing time for undirected (resp. directed) planar graphs with non-negative edge lengths [78]. A similar result for undirected planar graphs is found independently by Klein [59]. Kawarabayashi et al. [51] give the first linear size $(1+\epsilon)$-approximate distance oracle with $\tilde{O}(\text{polynomial}(1/\epsilon))$ query time. The preprocessing time is $O(n\log^2 n)$. Kawarabayashi et al. give a $(1+\epsilon)$-approximate distance oracle with $\bar{O}(1/\epsilon)$ query time, $\bar{O}(n\log n)$ size and $\bar{O}(n\epsilon^{-2}\log^4 n)$ preprocessing time for undirected planar graphs with non-negative edge lengths, where $\bar{O}$ is defined to hide $\log\log n$ and $\log(1/\epsilon)$ factors [57]. This distance oracle has a better trade-off between the query time and the oracle size than the one in [59, 78] when $\log\log n$ factors and $\log(1/\epsilon)$ factors are ignored. Recently, Wulff- Nilsen gives a $(1+\epsilon)$-approximate distance oracle with $O(n(\log\log n)^2/\epsilon + \log\log n/\epsilon^2))$ size and $O((\log\log n)^3/\epsilon^2 + \log\log n\sqrt{\log\log((\log\log n)/\epsilon^2)}/\epsilon^2)$ query time for undirected planar graph with non-negative edge lengths [83]. This result has a better trade-off between the query time and the oracle size than those in [57, 59, 78].

1.1.3 Tree-/Branch-Decomposition Based Distance Oracles

For graphs with small tree-/branchwidth (see related definitions in section 2.2), there are efficient distance oracles based on tree-/branch-decompositions. Let k be the branchwidth of the graph and let $T(n,k)$ be the time complexity of computing a branch-decomposition with width $O(k)$. Chaudhuri and Zaroliagis [23] give an oracle for weighted general digraphs with $O(k^3\alpha(n))$ query time, $O(k^3 n)$ size, and $O(k^3 n + T(n,k))$ preprocessing time where $\alpha(n)$ is the inverse of Ackermann function [3] and is a very slowly growing function. For size $S \in [n\log\log k, n^2]$, Mozes and Sommer give a distance oracle for weighted planar digraphs with $O(\min\{k\log n\log^2 k\log\log k, n\sqrt{n}\log^{3.5} n\}$ query time and $O(T(n,k)\log n + S\log^2 n)$ preprocessing time [65]. This distance oracle has better query time for graph with branchwidth $O(n^{o(1)})$ for $S \in [n\log\log k, n^{1.5})$ than the oracle in [38].

1.1.4 Distance Oracles for General Graphs

Cohen et al. [26] give a 2-hop cover labeling scheme for general digraphs in which each vertex v is associated with a set of landmarks $L(v)$ (called a label) such that for any pair of vertices x and y, $L(x)$ and $L(y)$ contains at least one vertex on a shortest path between x and y. By keeping the distances between each v and its landmarks, a distance query can be answered in $O(|L(x)| + |L(y)|)$ time for any vertices x and y. Let $\mathcal{L} = \Sigma_{v\in V(G)} L(v)$ be the total label size. This yields an exact distance oracle with $O(\mathcal{L})$ size and $O(\mathcal{L}/n)$ average query time. There is no absolute guarantee on $\mathcal{L}$ or the maximum label size. However $O(\log n)$-approximates for the total landmarks with the minimum size (and thus the size of the oracle) [26] and the maximum label size (and thus the query time) [10] can both be computed in polynomial time.

Thorup and Zwich [79] show that for any integer $k \geq 1$ there is a $(2k-1)$-approximate distance oracle for general graphs with $\tilde{O}(mn^{1/k})$ preprocessing time, $O(kn^{1+\frac{1}{k}})$ size and $O(k)$ query time for weighted undirected graphs. The size is essentially optimal for the stretch. Chechik [24] improves the query time to $O(1)$. Wulff-Nilsen [82] improves the preprocessing time to almost proportional to the oracle size. Patrascu and Roditty [68] show that a 2-approximate distance oracle for general graphs can be computed in $O(\text{poly}(n))$ time with $O(m^{1/3}n^{4/3})$ size and $O(1)$ query time. For unweighted graphs, Abraham and Gavoille [2] give a $(2k-2, 1)$-approximate distance oracle with $\tilde{O}(n^{1+\frac{2}{2k-1}})$ size, $O(k)$ query time and $O(\text{poly}(n))$ preprocessing time for any integer $k \geq 1$.

1.1.5 Dynamic Distance Oracles

There is a vast literature on dynamic distance oracles that can handle one or more edge/vertex updates. A dynamic distance oracle is said to be *incremental* if it only handles insertion updates, *decremental* if it only handles deletion updates and *fully dynamic* if it handles both types of updates. King [58] give a fully dynamic exact distance oracle with $O(1)$ query time and $O(n^{2.5}\sqrt{M \log n})$ update time for general digraphs with edge length drawn from $[1, 2, \ldots, M]$. Demetrescu and Italiano [31] give a fully dynamic exact distance oracle for general digraphs with non-negative edge length with $O(1)$ query time and amortized update time $\tilde{O}(n^2)$. There are also works on incremental/decremental distance oracles (see e.g. [7, 11]).

Let D denote the sum of all edge lengths. For planar graphs with non-negative edge length, Klein and Subramanian [62] give a fully dynamic $(1 + \epsilon)$-approximate distance oracle that has $O(\epsilon^{-1} n \log^2 n \log D)$ preprocessing time, $O(n + (\epsilon^{-1} n^{2/3} \log n \log D))$ size, $O(\epsilon^{-1} n^{2/3} \log^2 n \log D)$ query time and amortized update time. Abraham et al. [1] give a fully dynamic $O(1 + \epsilon)$-approximate distance oracle for planar graphs with edge lengths drawn from $[1, 2, \ldots, M]$ with $O(\epsilon^{-1} n \log^2 n)$ preprocessing time, $O(n \log n(\epsilon^{-1} + \log n))$ size, $O(\epsilon^{-1} n^{1/2} \log^2 n \log(nM)(\epsilon^{-1} + \log n))$ query time and worst case update time.

1.2 Research Questions and Contributions

In this thesis we study both exact and approximate distance oracles for planar graphs. More specifically we study the following two questions.

1.2.1 Improving the Preprocessing Time for Branch-Decomposition Based Exact Distance Oracles

For graphs with small treewidth/branchwidth, tree-/branch-decompositions are a commonly used tool for developing distance oracles. The preprocessing time of tree-/branch-decomposition based distance oracles are often dominated by the computation of a desired tree-/branch-decomposition. In Chapter 3 we give an algorithm [42, 45] that for an input planar graph G of n vertices and an integer k, in $\min\{(n \log^3 n), O(nk^2)\}$ time either constructs a branch-decomposition of G of width at most $(2 + \delta)k$, where $\delta > 0$ is a constant, or a $(k + 1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G implying $\mathrm{bw}(G) > k$, where $\mathrm{bw}(G)$ is the branchwidth of G. This is the first $\tilde{O}(n)$ time constant-factor approximation for branchwidth/treewidth and largest grid/cylinder minors of planar graphs and improves the previous $\min\{O(n^{1+\epsilon}), O(nk^2)\}$ (where $\epsilon > 0$ is a constant) time constant factor approximations. For a planar graph G and $k = \mathrm{bw}(G)$, a branch-decomposition of width at most $(2 + \delta)k$ and a $g \times \frac{g}{2}$ cylinder/grid minor with $g = \frac{k}{\beta}$, where $\beta > 2$ is a constant, can be computed by our algorithm in $\min\{O(n \log^3 n \log k); O(nk^2 \log k)\}$ time. Using this algorithm, we improve the preprocessing time of the branch-decomposition based distance oracle in [65] from $O(n^{1+\epsilon}+S\log^2 n)$ to $O(\min\{O(n\log^4 n\log k); O(nk^2\log n\log k)\}+S\log^2 n)$, where $S\in[n\log\log k, n^2]$ and $\epsilon>0$ is a constant. For $S\in[n\log\log k, \min\{n\log^2 n\log k, nk^2\log k/\log n\}]$, the new preprocessing time is better. Branch-decompositions are an important tool for developing efficient algorithms for many problems in graphs. Our result has independent interests in graph algorithms as well.

1.2.2 Improving the Query Time for Approximate Distance Oracles

Distance oracles with constant query time are of both theoretical and practical importance [24, 28]. Many $(1+\epsilon)$-approximate distance oracles that have have $O(1/\epsilon)$ query time and size nearly linear in n have been proposed before our result yet none of them have constant query time independent of ϵ. In Chapter 4, we give a $(1+\epsilon)$-approximate distance oracle [43, 44] with $O(1)$ query time for an undirected planar graph G with n vertices and non-negative edge lengths. For $\epsilon>0$ and any two vertices u and v in G, our oracle gives a distance $\tilde{d}(u,v)$ with stretch $(1+\epsilon)$ in $O(1)$ time. The oracle has size $O(n\log n(\log^{1/6} n+\log n/\epsilon+f(\epsilon)))$ and preprocessing time $O(n\log n((\log^3 n)/\epsilon^2+f(\epsilon)))$, where $f(\epsilon)=2^{O(1/\epsilon)}$. This is the first $(1+\epsilon)$-approximate distance oracle with $O(1)$ query time independent of ϵ and the size and preprocessing time nearly linear in n, and improves the query time $O(1/\epsilon)$ of previous $(1+\epsilon)$-approximate distance oracle with size nearly linear in n.

We compute our distance oracle in three steps. In the first step we compute a basic distance oracle DO_0 with $O(\epsilon D)$ additive stretch and $O(1/\epsilon)$ query time using some commonly used techniques. In the second step, we show how to use some classification scheme to reduce the query time of DO_0 and get our second distance oracle DO_1 with $O(\epsilon D)$ additive stretch and $O(1)$ query time. And in the final step, we use some scaling techniques to reduce the stretch of DO_1 and get a distance oracle DO_2 with $(1+\epsilon)$ stretch and $O(1)$ query time. In order to make the scaling techniques work for our needs, we present a data structure that is critical to guarantee $O(1)$ query time while maintaining nearly linear (in n) oracle size.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 contains preliminaries that are related to both Chapter 3 and 4, such as basic definitions and a more detailed review of previous work and main techniques for distance oracles. Chapters 3 and 4 explain the main contributions of this thesis, as outlined in Section 1.2. In Chapter 3 and 4, we first give an introduction of the problem that is studied and then introduce some basic definitions and notations that are used in the corresponding chapter before elaborating our work. The final section gives conclusions and future works.

Chapter 2

Definitions and Previous Works

2.1 Basic Definitions on Graphs

Definition 2.1.1. *A* graph *$G = (V, E)$ consists of a set V of vertices and a set E of edges, where each edge $e \in E$ is a subset of V with two elements.*

Definition 2.1.2. *A* directed graph *(digraph) $G = (V, A)$ consists of a set V of vertices and a set A of directed edges (also called arcs), where each arc $\overrightarrow{a} \in A$ is an ordered pair of vertices in V.*

Definition 2.1.3. *An* (edge-)weighted *graph is a graph $G = (V, E)$ associated with a weight function $w : E \to \mathbb{R}$. A* vertex-weighted *graph is a graph $G' = (V', E')$ associated with a weight function $w' : V' \to \mathbb{R}$.*

Definition 2.1.4. *Two vertices u and v in a graph G are* adjacent *if $\{u, v\}$ is an edge in $E(G)$. For any vertex u, a vertex v is a* neighbour *of u if u and v are adjacent. The* degree *of a vertex v, denoted as* $\deg(v)$*, is the number of neighbours of v.*

Definition 2.1.5. *A graph $G' = (V', E')$ is a* subgraph *of another graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.*

Definition 2.1.6. *Let $G = (V, E)$ be a graph. Let $V' \subseteq V$ be any subset of V and let $E' \subseteq E$ be any subset of E. The* vertex induced subgraph *$G[V']$ of G is the graph with vertex set V' and edge set $\{(u, v) \in E | u, v \in V'\}$. The* edge induced subgraph *$G[E']$ of G is the graph with edge set E' and vertex set $V' = \{v | v \in \cup_{e \in E'} e\}$.*

For any subgraph G' of G, let $G \backslash G'$ denote $G[E(G) \backslash E(G')]$ and let $G - G'$ denote $G[V(G) \backslash V(G')]$. For any subset $A \subseteq E(G)$, let $\overline{A}$ denote $E(G) \backslash A$ and let $\partial(A)$ denote the vertex set $V(G[A]) \cap V(G[\overline{A}])$. The pair $(A, \overline{A})$ is called a *separation* of G and the *order* of separation $(A, \overline{A})$ is $|\partial(A)|$.

Definition 2.1.7. *A* walk *in G is a sequence of edges $e_1, .., e_k$, where $e_i = \{v_{i-1}, v_i\}$ for $1 \leq i \leq k$. A* path *is a walk in which all the vertices are distinct.*

Definition 2.1.8. *A* cycle *is a walk in which the first vertex and the last vertex are the same and the number of edges in the walk is more than one. A* simple cycle *is a cycle in which all the vertices except the first and last vertices are distinct.*

Let $l(e)$ be the edge weight (length) of edge e in G. The *length* of path $Q = e_1, ..., e_k$ is $l(Q) = \sum_{1 \leq i \leq k} l(e_i)$.

Definition 2.1.9. *Let $\mathcal{Q}_{uv}$ be the set of all paths in G between vertices u and v. The* distance *between u and v in G is $d_G(u,v) = \min\limits_{Q \in \mathcal{Q}_{uv}} l(Q)$. A path $Q' \in \mathcal{Q}_{uv}$ is a* shortest path *between u and v if $l(Q') = d_G(u,v)$.*

We may write $d_G(u,v)$ as $d(u,v)$ when G is clear from context. We say path Q *intersects* path Q' (at vertex v) if $V(Q) \cap V(Q') \neq \emptyset$ (and $v \in V(Q) \cap V(Q')$). We say that two paths $Q_1 = (v_1, \ldots, v_{i-1}, v_i, v_{i+1} \ldots, v_{i_1})$ and $Q_2 = (u_1, \ldots, u_{i-1}, v_i, u_{i+1} \ldots, u_{i_2})$ *cross with* each other at vertex v_i if edges/arcs $\{v_{i-1}, v_i\}$, $\{u_{i-1}, v_i\}$, $\{v_i, v_{i+1}\}$, $\{v_{i-1}, u_{i+1}\}$ appear in clockwise (or counterclockwise) order.

Definition 2.1.10. *The* radius *of G is $r(G) = \min\limits_{u \in V(G)} \max\limits_{v \in V(G)} d_G(u,v)$. The* diameter *of G is $d(G) = \max\limits_{u \in V(G), v \in V(G)} \mathrm{dist}_G(u,v)$.*

Definition 2.1.11. *A* tree *is a graph in which any two vertices are connected by exactly one path. A* forest *is a disjoint union of trees.*

For a tree T, any vertex $r \in V(T)$ can be selected as the *root* of T. For any $v \in V(T)$, the *depth* of v is the number of edges of the path between v and the root in T. The depth of T is the largest depth of any vertex in T.

Let T be a rooted tree. For any vertex $v \in V(T)$, the path between the root of T and v, denoted as $T(v)$, is called the *root path* of v. For any vertex $v \in V(T)$ that is not the root of T, the *parent* of v is the vertex that is connected to v in the root path of v. A *child* of a vertex v is a vertex whose parent is v. A *leaf* in a rooted tree is a vertex with no children. A *leaf* in an unrooted tree is a vertex with degree one.

Definition 2.1.12. *A* binary tree *(resp.* ternary tree*) is a tree T where every $v \in V(T)$ has no more than two (resp. three) children.*

Definition 2.1.13. *An* unrooted binary tree *is an unrooted tree T where every $v \in V(T)$ that is not a leaf has degree three.*

Definition 2.1.14. *A* spanning tree *T of a graph G is a subgraph of G that is a tree which includes $V(G)$. A* shortest path spanning tree *rooted at vertex r is a spanning tree T of G such that for any $v \in V(G)$, $d_T(r,v) = d_G(r,v)$.*

Definition 2.1.15. *A graph G is* connected *if there is a path between any pair of vertices in G. G is* biconnected *if for any $v \in V(G)$, $G[V(G) \backslash \{v\}]$ is connected.*

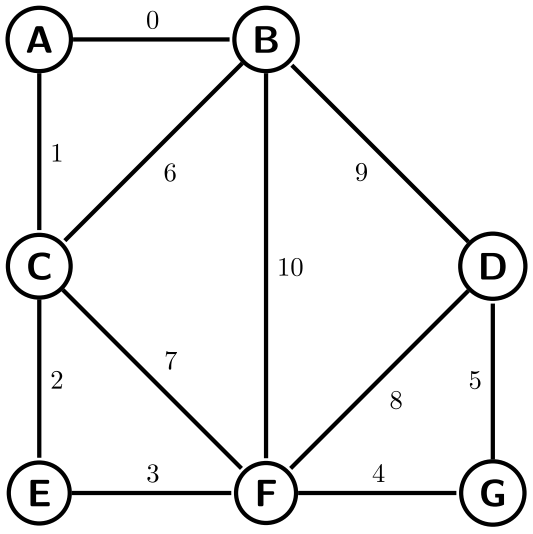


Figure 2.1: A plane graph with 7 nodes and 11 edges

Definition 2.1.16. *A* vertex cut set *in a graph G is a subset S of $V(G)$ such that $G[V(G) \backslash S]$ is not connected. A vertex cut set is said to be a* (α-)balanced (vertex) separator *if each connected component of $G[V(G) \backslash S]$ has no more than αn vertices for some previously fixed constant $\alpha > 0$.*

Let Σ be a sphere. For an open segment s homeomorphic to $\{x | 0 < x < 1\}$ in Σ, we denote by $\text{cl}(s)$ the closure of s. A planar embedding of a graph G is a mapping $\rho : V(G) \cup E(G) \to \Sigma \cup 2^{\Sigma}$ such that

- for $u \in V(G)$, $\rho(u)$ is a point of Σ, and for distinct $u, v \in V(G)$, $\rho(u) \neq \rho(v)$;
- for each edge $e = \{u, v\} \in E(G)$, $\rho(e)$ is an open segment in Σ with $\rho(u)$ and $\rho(v)$ the two end points in $\text{cl}(\rho(e)) \setminus \rho(e)$;
- for distinct $e_1, e_2 \in E(G)$, $\text{cl}(\rho(e_1)) \cap \text{cl}(\rho(e_2)) = \{\rho(u) | u \in e_1 \cap e_2\}$.

Definition 2.1.17. *A graph G is* planar *if it has a planar embedding ρ, and (G, ρ) is called a* plane graph.

We may simply use G to denote the plane graph (G, ρ), leaving the embedding ρ implicit. Figure 2.1 gives an example of plane graph.

Definition 2.1.18. *A* curve *is a continuous image of a closed unit line segment. A curve is* simple *if it does not intersect itself. For a plane graph G, a curve μ on Σ is* normal *if μ does not intersect any edge of G. The length of a normal curve μ is the number of connected components of $\mu \setminus \bigcup_{v \in V(G)} \{\rho(v)\}$.*

Definition 2.1.19. *A* noose *of a plane graph G is a closed normal curve on Σ that does not intersect itself.*

Definition 2.1.20. *A* face *of a plane graph G is a connected component of* $\Sigma \setminus (\cup_{e \in E(G)} \mathrm{cl}(\rho(e)))$.

We denote by $V(f)$ and $E(f)$ the set of vertices and the set of edges incident to face f, respectively. We say that face f is bounded by the edges of $E(f)$. For two faces f and g, a cycle *separates* f and g if the cycle separates Σ into two regions, one region contains f and the other region contains g.

Given a plane graph G, we can specify any face of G as the outer face f_{out} of G. A planar graph G is *triangulated* (resp. *almost triangulated*) if every face in G (resp. other than the outer face) is incident to exactly three edges.

Definition 2.1.21. *For two faces f and g, a* minimum (f, g)-separating cycle *C is a cycle separating f and g with the minimum length. We also call C a* minimum face-separating cycle *(for f and g).*

Let C be a simple cycle in G. Then $\Sigma \setminus (\cup_{e \in E(C)} \mathrm{cl}(\rho(e)))$ contains exactly two connected components, one containing f_{out}, called the *external region of C*, and the other called the *internal region of C*.

Definition 2.1.22. *For an assignment $W(\cdot)$ of non-negative weights to faces, edges, and vertices of G, a simple cycle C is a* balanced cycle separator *if the total weight of faces, edges, and vertices embedded in the external/internal region of C is no more than a constant fraction of the total weight of G.*

2.2 Tree-/Branch-Decompositions

The notions of *tree-decomposition* and *branch-decomposition* are introduced by Robertson and Seymour [71, 72, 73] in graph minor theory.

Definition 2.2.1. *A tree-decomposition of $G = (V, E)$ is a pair (T, B) where T is a tree and B is a function that maps each node $\nu \in V(T)$ to a subset of V, called a bag, such that*

1. $\cup_{\nu \in V(T)} B(\nu) = V$
2. *for each $e = (u, v) \in E$, there exists a $\nu \in V(T)$ such that $u, v \in B(\nu)$*
3. *for each $v \in G$, $I_v = \{\nu \in V(T) | v \in B(\nu)\}$ induces a subtree of T.*

The *width* of a tree-decomposition (T, B) is the size of its largest bag minus 1. The *treewidth* of a graph G, denoted as $\mathrm{tw}(G)$, is the minimum width among all possible tree-decompositions of G. Figure 2.2 gives an example of tree-decomposition.

Definition 2.2.2. *A branch-decomposition of $G = (V, E)$ is a pair (T', ϕ), where T' is a unrooted binary tree and ϕ is a bijection from the set of leaves of T' to E.*

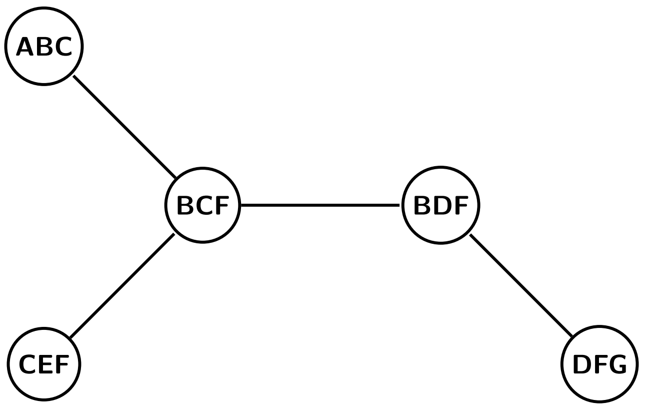


Figure 2.2: A tree-decomposition with width 2 for the graph in Figure 2.1

For a branch-decomposition (T', ϕ), we refer to the vertices of T' as *nodes* and the edges of T' as *links* to distinguish them from the vertices and edges of G. Removing any link $e \in E(T')$ partitions T' into two subtrees $T'_1(e)$ and $T'_2(e)$. Let $L_1(e), L_2(e) \subseteq V(T')$ be the sets of leaves of $T'_1(e)$, $T'_2(e)$ respectively. Then for $i = 1, 2$, $\phi(L_i(e)) = E(G)\backslash\phi(L_{3-i}(e))$ and $\partial(\phi(L_i(e)))$ is a vertex cut set of G if $\partial(\phi(L_i(e))) \neq (\phi(L_i(e)))$. We say that the separation $(\phi(L_1(e)), \phi(L_2(e)))$ is induced by link e. The *width* of a branch-decomposition (T', ϕ) is the largest order of the separations induced by links of T'. The *branchwidth* of G, denoted by $\mathrm{bw}(G)$, is the minimum width of all branch-decompositions of G. In the rest of this paper, we identify a branch-decomposition (ϕ, T') with the tree T', leaving the bijection implicit and regarding each leaf of T' as a edge of G. Figure 2.3 gives an example of branch-decomposition.

For a graph G with more than one edge, the branchwidth and the treewidth are linearly related: $\max\{bw(G), 2\} \leq tw(G) + 1 \leq \max\{\lfloor \frac{3}{2} bw(G) \rfloor, 2\}$, and there are simple $O(n\mathrm{tw}(G))$ time translations between branch-decompositions and tree-decompositions that meet the linear relations [73]. Therefore we use only the terms branch-decomposition and branchwidth for the rest of this thesis for ease of representation unless otherwise stated.

2.3 General Techniques and Results for Exact Distance Oracles for Planar Graphs

2.3.1 Portals

Portals are a set of carefully selected vertices and are commonly used in both exact and approximate distance oracles. Portals can be either global portals which are computed for all the vertices, or local portals which are computed for some specific vertex. Portals must be selected such that for every pair of vertices s and t in the graph, at least one portal is in the (approximate) shortest path between s and t. Common ways of selecting portals include random sampling, high-degree vertices, and vertices on separators etc. [75]. The distances

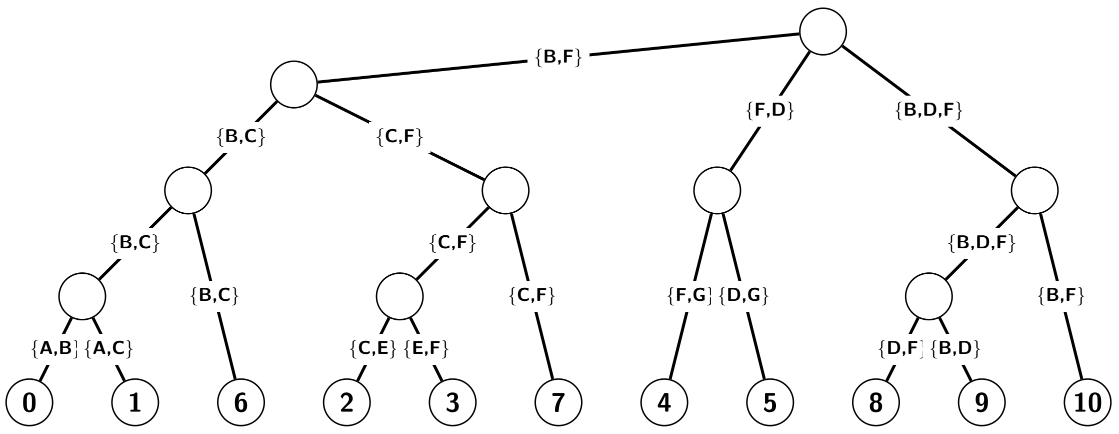


Figure 2.3: A branch-decomposition with width 3 for the graph in Figure 2.1. $\partial(\phi(L_1(e)))$ for each e is shown.

from each vertex to certain portals (usually those that are close to the vertex) and distances among the portals are pre-computed and stored in the distance oracle, either explicitly or implicitly. In each query, these pre-computed distances are used to find out (the portals in, and therefore the length of) the desired shortest path.

2.3.2 Small Balanced Separators and r-division

Lipton and Tarjan [63] prove that any n vertex planar graph has a balanced separator with $O(\sqrt{n})$ size and that the separator can be computed in $O(n)$ time. Using this small balanced separator theorem recursively and with additional care, Frederickson [34] shows that a planar graph can be divided in a balanced way to an *r-division* with some nice properties. Given a parameter r, an *r-division* of a graph G is a division of G into edge induced subgraphs, called *regions*, that are edge disjoint from each other. A vertex v is a *boundary vertex* of the r-division (and the regions v is in) if it is in more than one region.

Definition 2.3.1. *An* r-division *is a division of G into $\theta(n/r)$ regions such that each region has $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices.*

In distance oracles for planar graphs, the given graph is usually divided into smaller regions and the boundary vertices are usually selected as portals. By dividing the input graph using small balanced separators recursively, choosing the vertices in the separators as portals and storing the distances from/to the portals, Djidjev [32] presents a distance oracle with $O(n^{3/2})$ preprocessing time, $O(n^{3/2})$ oracle size, and $O(\sqrt{n})$ query time for weighted planar digraphs. Djidjev also gives a representative result in [32] based on r-division which presents a general trade-off between the space requirement and the query time for weighted digraphs. It's proved that any query can be answered in $O(n^2/S)$ time using $O(S)$ space for $S \in [n, n^2]$. The preprocessing time is $O(S)$ for $S \in [n^{3/2}, n^2]$ and $O(n\sqrt{S})$ for $S \in [n, n^{3/2})$. A similar result for $S \in [n^{3/2}, n^2]$ is proved concurrently by Arikati et al. [6].

Note that these distance oracles do not rely on planarity except for using small separator, r-division and the linear time single source shortest path (SSSP) algorithm in [48]. Therefore they can be modified to work for other minor-closed graph classes with $O(\sqrt{n})$ size balanced separator, such as bounded-genus graphs and H-minor-free graphs for fixed $|H| = O(1)$. More specifically, the trade-off between the space requirement and the query time remains the same for minor-closed graph classes that satisfy two conditions: the existence of $O(\sqrt{n})$ size balanced separator and a $O(n^{1-\epsilon})$ size balanced separator can be computed in $O(n)$ time. And for these graphs, the the preprocessing time is increased to $\max\{O(n^{3/2}\log n), O(S)\}$ for $S \in [n^{3/2}, n^2]$ and $\max\{O(n^{3/2}\log n), O(n\sqrt{S})\}$ for $S \in [n, n^{3/2})$.

For biconnected triangulated planar graphs, Miller [64] proves that a balanced simple cycle separator of size $O(\sqrt{n})$ can be computed in $O(n)$ time. By adding dummy vertices and dummy edges and then using Miller's simple cycle separator instead of Lipton and Tarjan's separator, Klein and Subramanian [62] show that an r-division of a planar graph can be computed such that the boundary vertices of each region lie on the boundary of a constant number of faces (called *holes*). This division is called an *r-division with few holes.*

2.3.3 Monge Property and FR-Dijkstra

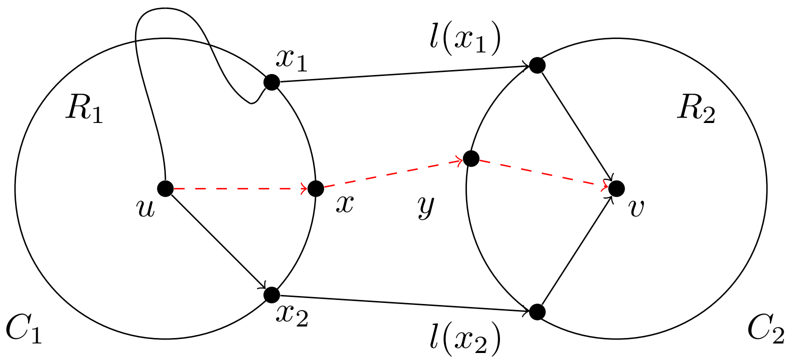
In this subsection, we introduce some distance oracles that further exploit planarity. Recall that in the query phase, distances from/to/between the portals are examined/relaxed to find a shortest path. For planar graphs, Djidjev [32] takes advantage of topological properties of planar graphs and shows how to examined such distances more efficiently. For ease of presentation, let us assume that for every pair of vertices there is exactly one shortest path between them. For any two vertices u and v in the graph, let $Q_{u,v}$ be the path from u to v. For any triangulated planar graph G, the boundary vertices of each region in an r-division of G form a union of simple edge-disjoint cycles [32]. Let R_1, R_2 be two regions in such an r-division containing vertices u and v respectively. Let C_1, C_2 be two boundary cycles in R_1, R_2 respectively that separate u and v. Then every path from u to v must contain some vertex in C_1 and some vertex in C_2. The shortest path can be computed by examining every pair of vertices in C_1 and C_2, which is $O(|V(C_1)||V(C_2)|)$ vertices pairs. This can be improved for planar graphs by taking advantage of planarity. See Figure 2.4a for illustration. For every vertex x on C_1, define $\mathcal{Q}(u, v, x)$ to be the set of paths from u to v which go through x with x being the last vertex in the path that is in C_1. Let $Q(u, v, x)$ be the path that has minimum length among all the paths in $\mathcal{Q}(u, v, x)$ and let $l(x)$ be any vertex in $V(Q_{x,v}) \cap V(C_2)$. Let x_1, x_2 be any two vertices in C_1. Then for any vertex x in C_1 such that x_1, x, x_2 appear in clockwise (resp. counterclockwise) order, there is always some vertex $y \in V(Q_{x,v}) \cap V(C_2)$ such that $l(x_1), y, l(x_2)$ appear in counterclockwise (resp. clockwise) order. Therefore not every subpaths from x to v needs to be examined to compute $Q(u, v, x)$ and the problem of finding $Q(u, v, x)$ for every $x \in V(C_1)$ can be solved in a divide-and-conquer way. By exploiting this *non-crossing* property, Djidjev [32] improves their trade-off between the space requirement and the query time for size $S \in [n^{4/3}, n^2]$ to $O(n \log(n/\sqrt{S})/\sqrt{S})$ query time. The preprocessing time is $O(n\sqrt{S})$.[1] Chen and Xu [25] give a similar result to for undirected planar graphs.

A similar idea is depicted as the *Monge property* in [33]. Let A, B be two ordered sets and $d : A \times B \to \mathcal{R}$ be a function between pairs of elements in A and B. Then function d is said to have the Monge property if for all $a_1 < a_2$ in A and $b_1 < b_2$ in B, $d(a_1, b_1) + d(a_2, b_2) \leq d(a_1, b_2) + d(a_2, b_1)$. Let f be a face in a plane graph G such that $E(f)$ form a simple cycle $C(f)$. Then for any vertices $v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}$ that appear in clockwise/counterclockwise order in C, $d_G(v_{i_1}, v_{i_4}) + d_G(v_{i_2}, v_{i_3}) \leq d_G(v_{i_1}, v_{i_3}) + d_G(v_{i_2}, v_{i_4})$ (see Figure 2.4b). Therefore if we split and rank the vertices in $V(f)$ into two consecutive ordered sets A and B according to their position in $C(f)$, the distance function between vertices in A and B has the Monge property. And the problem of finding the best "counterpart" in B for every element in A can be solved in a divided-and-conquer way. This is called the *Monge property* of planar graph.

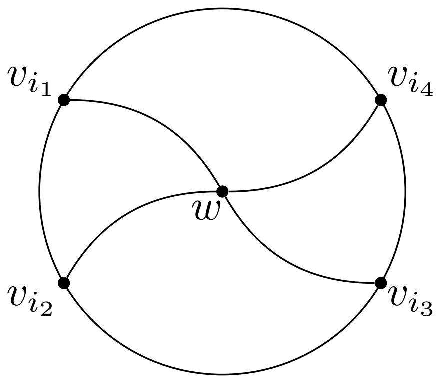
Let R be a region in an r-division and ∂R be the set of boundary vertices of R. The *dense distance graph* of R is the complete graph on ∂R such that the weight of each edge (u, v) equals $d_R(u, v)$. Fakcharoenphol and Rao [33] observe that the dense distance graph can be decomposed to $O(\log n)$ bipartite graphs where Monge property holds (see Figure 2.4c) and show that all edges in the dense distance graph of any region in an r-division with few holes can be relaxed in time nearly linear in the number of vertices, instead of edges, in the dense distance graph. More specifically, they show that for any graph H which consists of a set of dense distance graphs of regions in an r-division with few holes, a shortest path tree in H from any vertex can be computed in $O(|V(H)| \log^2 n)$ time. This is called the *FR-Dijkstra* algorithm and yields an exact distance oracle for planar digraphs with $O(n \log^3 n)$ preprocessing time, $O(\sqrt{n} \log^2 n)$ query time and $O(n \log n)$ size [33]. This distance oracle has better query-preprocessing time product and query-size product than previous results. Using FR-Dijkstra algorithm, Mozes and Sommer [65] and Nussbaum [67] independently give a distance oracle with $O(n \log n)$ preprocessing time, $O(n)$ space and $O(n^{1/2+\epsilon})$ query time for any constant $\epsilon > 0$. These are the first linear space exact distance oracles with provably sublinear query time.

A *partial Monge array* is an array that may have some blank entries and the non-blank entries satisfy the Monge property when we view its rows as ordered set A and columns as ordered set B. It is known that for certain $m \times n$ partial Monge arrays, the minimum/maximum non-blank entry can be found in $O(m + n)$ time if each entry can be accessed in $O(1)$ time [4]. Note that for a partial Monge array remains a partial Monge array if we add a constant

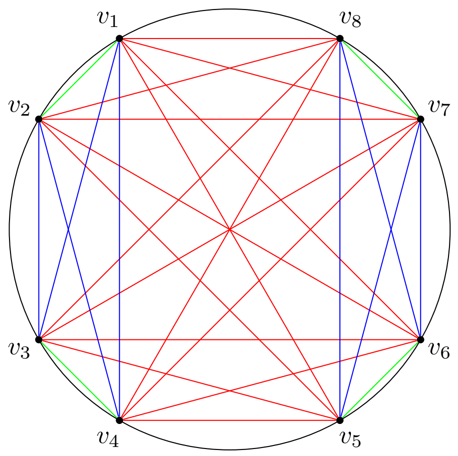
[1]The original result in [32] is a distance oracle with $O(n\sqrt{S})$ preprocessing time, $O(n \log n/\sqrt{S})$ query time for $S \in [n^{4/3}, n^{3/2}]$. However the same distance oracle works for $S \in [n^{4/3}, n^2]$ and the query time is actually $O(n \log(n/\sqrt{S})/\sqrt{S})$. This is first pointed out in [67].



(a) For x_1, x, x_2 appear in clockwise order, there is a $y \in V(Q_{x,v} \cap C_2)$ such that $l(x_2), y, l(x_1)$ appear in counterclockwise order



(b) Monge property in planar graph. For v_1, v_2, v_3, v_4 lying on a face, Q_{v_1,v_3} and Q_{v_2,v_4} must cross with each other at some vertex w. Therefore $d(v_{i_1}, v_{i_4}) + d(v_{i_2}, v_{i_3}) \le d(v_{i_1}, v_{i_3}) + d(v_{i_2}, v_{i_4})$.



(c) A dense distance graph with three different color edges representing the three bipartite graphs where Monge property holds

Figure 2.4: Monge Property

to all the non-empty entries in an entire row or an entire column. Consider the same example in Figure 2.4a. To find the distance from vertex u to vertex v, it suffices to find the portal pair $x \in C_1$ and $y \in C_2$ that minimize $d_G(u,x) + d_{G\setminus(R_1 \cup R_2)}(x,y) + d_G(y,v)$. It is observed in [67] that the distances between the portals can be divided into to two classes such that the distances in each class form a partial Monge array. Therefore the distance from u to v can be found in $O(|V(C_1)| + |V(C_2)|)$ time by using the minimum entry searching algorithm in [4] directly if the distances to/from/among the portals are precomputed. Using this method, Nussbaum [67] improves the trade-off between space requirement and query time in [32] and gives an exact distance oracle with $O((S^{3/2}/\sqrt{n}) \log^2 n)$ preprocessing time, $O(S)$ size and $O(n/\sqrt{S})$ query time for $S \in [n^{4/3}, n^2]$.

Klein gives a multiple-source shortest path (*MSSP*) data structure [60] for planar digraphs with non-negative edge weights. Given an n-vertex plane digraph G with non-negative weights and a face f in G, the MSSP data structure answers the distance from any vertex of G to any vertex incident to f in $O(\log n)$ time. The preprocessing time and space requirement for MSSP are $O(n \log n)$. Note that this result can be used to compute the dense distance graph of a region in an r-division with few holes in $O(r \log n)$ time and instantly removes a $O(\log n)$ factor in the preprocessing time of Fakcharoenphol and Rao's distance oracle in [33]. Combing r-division with few holes and FR-Dijkstra algorithm, Mozes and Sommer [65] extend MSSP so that it answers distance from any vertex of G to all vertices on a cycle (as opposed to a vertex incident to a face in MSSP) of size c in $O(c \log^2 c \log \log c)$ time. This data structure is called the cycle MSSP and has $O(n \log \log c)$ size and $O(n \log^3 n)$ preprocessing time. Note that the query to a single vertex of the cycle and the query to all vertices on the cycle have the same time complexity. For small cycles, the amortized query time is better than MSSP.

Combining the linear space exact distance oracle [65] and the cycle MSSP data structure, Mozes and Sommer give an exact distance oracle [65] with $O(S)$ size and $\tilde{O}(S)$ preprocessing time, that answers a distance query in $\tilde{O}(n/\sqrt{S})$ time for $S \in [n \log \log n, n^2]$, where the $\tilde{O}$ notion hides poly-logarithmic factors. This extends the range of S for the trade-off in [32] without increasing the preprocessing time or query time when poly-logarithmic factors are ignored.

2.3.4 Voronoi Diagram and Point Location Structure

Recently, a series of breakthroughs have been made following Cabello's [20, 21] novel use of *Voronoi diagrams* in planar graphs. Let $\{s_1, s_2, \dots, s_n\}$ be a set of points, called *sites*, in a plane. A Voronoi Diagram is a partition of the plane into *Voronoi cells* such that each cell $\mathrm{Vor}(s_i)$ contains the points on which s_i has more influence than any other site. Consider a region R in an r-division with few holes, a hole H in R, and a vertex $u \notin V(R)$. Each boundary vertex on H can be viewed as a site s_i with a weight $\omega(s_i)$ that equals $d_G(u, s_i)$. The vertices of R can be partitioned into Voronoi cells (with respect to H and u), one cell $\mathrm{Vor}(s_i)$ for each s_i. Each Voronoi cell $\mathrm{Vor}(s_i)$ contains all the vertices v in P such that s_i minimizes $\omega(s_i) + d_P(s_i, v)$ among all sites in H for v. We say s_i is the site of every vertex in $\mathrm{Vor}(s_i)$ with respect to u and H. Note that there is a hole H' in R and a site $s_{i'} \in V(H')$ of v (with respect to H' and u) such that $d_G(u, v) = \omega(s_{i'}) + d_P(s_{i'}, v)$. The dual of the edges whose endpoints are in different Voronoi cells form the Voronoi diagram of R with respect to H and u. To represent a Voronoi diagram compactly, each maximal path whose interior vertices have degree two is replaced by a single edge. Each Voronoi diagram has $O(\sqrt{r})$ vertices and if R is almost triangulated, the Voronoi diagram of R is a ternary tree [38].

An u-independent data structure called the *point location structure* is computed for each region R and each hole H in R such that given any vertex $u \notin V(R)$ and $v \in V(R)$, it returns the site of v with respect to u and H efficiently. To answer a query for a given source $u \notin V(R)$ and destination $v \in V(R)$, the distance oracle uses the point location structure to find out the site of v with respect to u and each hole H in R. The value $\omega(s_i) + d_P(s_i, v)$ is computed for each site s_i of v with respect to each H and the minimum value is the distance from u to v in G.

Consider a vertex $u \notin V(R)$, a vertex $v \in V(R)$, a hole H in R and the site s_i of v with respect to u and H. Let s_{i_1}, s_{i_2} be two sites in H. We say that sites s_{i_1} and s_{i_2} *cover* s_i if s_{i_1}, s_i, s_{i_2} appear in clockwise order along H. The general idea of how a point location structure works is that it answers the basic question in time t: do s_{i_1} and s_{i_2} cover s_i? And then by a binary search the point location structure finds s_i in $O(t \log r)$ time. To answer the basic question in constant time, Cohen-Addad et al. [27] define a set of boundaries with respect to each edge in R and each pair of sites in H such that if s_i is covered by s_{i_1} and s_{i_2} then v is enclosed by the boundary determined by edge (x, y) and s_{i_1}, s_{i_2}, where (x, y) is an edge determined by u. This reduces the total number of boundaries that need to be considered to $O(r^2)$. A simple method to answer the basic question in constant time is to store for each vertex v' in R and every possible boundary whether v' is enclosed by the boundary, which leads to $O(r^3)$ space requirement. To reduce the space requirement to $O(r^2)$, the point location structure stores a constant number of indices for every vertex v' with respect to every edge e' in R, such that the indices represents ranges of sites and can be used to determine whether s_i is covered by s_{i_1} and s_{i_2} in constant time by comparing s_{i_1}, s_{i_2} with the indices. This leads to an exact distance oracle for directed planar graphs with $\tilde{O}(n^{5/3})$ preprocessing time, $O(n^{5/3})$ space and $O(\log n)$ query time, which is the first exact distance oracle for planar graphs with truly subquadratic size that answers a query in $\tilde{O}(1)$ time. The paper also provides a trade-off between the space requirement and the query time: for size $S \in [n^{3/2}, n^2]$, the query time is $O(\frac{n^{5/2}}{S^{3/2}} \log n)$ and the preprocessing time is $\tilde{O}(S)$. This improves the trade-offs in [32, 65] for $S \geq n^{3/2}$.

Gawrychowski et al. [38] significantly simplify and improve the point location structure in [27] by reducing the space requirement for each region R from $O(r^2)$ to $O(r^{3/2})$ without increasing the time complexity of answering the basic question. Instead of storing indices for each vertex v' with respect to edge e', they store a shortest path tree from each of the sites. It is observed that whether v is enclosed by a boundary can be determined in $O(1)$ time by checking the preorder number of v in the shortest paths trees. This improves the result in [27] to a distance oracle with $\tilde{O}(n^2)$ preprocessing time, $O(n^{1.5})$ space and $O(\log n)$ query time. A trade-off between the space requirement and the query time is also given: for size $S \in [n, n^2]$, the query time is $\tilde{O}(\max\{1, n^{1.5}/S\})$. This trade-off is currently the best (up to polylogarithmic factors) for the entire range of S and improves all the previously known trade-offs for the range $S \in [n, n^{5/3}]$ by polynomial factors [38].

2.3.5 Tree-/Branch-Decomposition

As stated in section 2.3.2, small balanced separator and r-division are important tools used in distance oracles for planar graphs. As we know, any graph with branchwidths $O(\text{bw}(G))$ have balanced separators of sizes $O(\text{bw}(G))$ and these balanced separators can be easily found if given a branch-decomposition with width $O(\text{bw}(G))$ (see section 2.2). For planar graphs that are with branchwidth $o(\sqrt{n})$, using the balanced separators achieved from a branch-decomposition instead of the $O(\sqrt{n})$ size small balanced separators in [63, 64] may result in a better distance oracle. Gu and Tamaki [40] show that a branch-decomposition T with width $O(\text{bw}(G))$ for G can be computed in $O(n^{1+\epsilon})$ time for any constant $\epsilon > 0$. Using balanced separators achieved from branch-decomposition together with Miller's balanced cycle separators, Mozes and Sommer present a variant of r-division with few holes, called the $[r, k]$-division with few holes which can improve distance oracles for planar graphs with small treewidth.

Definition 2.3.2. *An* $[r, k]$-division with few holes *of a graph G with branchwidth k is a division of G into $\theta(n/r)$ regions such that each region has $O(r)$ vertices and* $\min\{O(\sqrt{r}), k\}$ *boundary vertices and the boundary vertices of each region lie on the boundary of a constant number of faces/holes.*

When computing a division of a graph, either Miller's cycle separator for planar graphs or the separator from branch-decomposition is chosen as a balanced separator, whichever is of smaller size. Such a division can be computed in $O(n^{1+\epsilon} \log n + n \log r + nr^{-\frac{1}{2}} \log n)$ time using Gu and Tamaki's branch-decomposition. Recall that by using r-division and FR-Dijkstra algorithm, Mozes and Sommer give an exact distance oracle with $O(S)$ size and $\tilde{O}(S)$ preprocessing time, and query time $\tilde{O}(n/\sqrt{S})$ for $S \in [n \log \log n, n^2]$. By using $[r, k]$-division instead of r-division for graphs with branchwidth k, the query time is improved to $O(\min\{k \log n \log^2 k \log \log k, nS^{-1/2} \log^{3.5} n\}$ for graphs with small branchwidth and the preprocessing time is $O(n^{1+\epsilon} + S \log^2 n)$ [65][2].

2.3.6 Price Function

A price function [52] f is a function that maps the set of vertices to a set of real numbers. The *reduced cost function* l_f induced by price function f is a function that assigns each arc (u, v) a new length: $l_f(u, v) = l(u, v) + f(u) - f(v)$. A price function is feasible if and only if all reduced arc lengths are non-negative. Note that the reduced cost function preserves negative cycles and shortest paths. By using a feasible price function, we can adapt an exact distance oracle for directed graphs with non-negative arc lengths to work for directed

[2]The original result in [65] is size $O(S)$, query time $O(\min\{k \log^2 k \log \log k, nS^{-1/2} \log^{2.5} n\}$ and preprocessing time $O(n^{1+\epsilon} + S \log^2 n)$ for $S \in [n \log \log k, n^2]$. However the result is wrong and is corrected by its author.

graphs with negative arc lengths. A feasible price function can be obtained from computing a shortest path tree from an arbitrary vertex u and for every v in the graph set $f(v) = d(u, v)$. For planar graphs, this can be done in $O(n \log^2 n/ \log \log n)$ time using the algorithm in [66].

2.4 General Techniques and Results for Approximate Distance Oracles for Planar Graphs

Approximate distance oracles are computed similarly as exact distance oracles for planar graphs. Generally speaking, the input graph is divided into smaller regions using balanced separators. Vertices on the separators are selected as portals and shortest paths are computed as the concatenation of the shortest paths in regions and between portals. The key difference is that approximate distance oracles do not consider every possible shortest paths. More specifically, not all vertices on the separators are selected as portals. This, however, would introduce error/detour for the shortest paths. Therefore the separators and portals need to be selected carefully such that the error introduced is not too big.

2.4.1 Shortest Path Separator

Lipton and Tarjan [63] observe that for any rooted spanning tree T of an undirected n-vertex planar graph G, there are two vertices u and v such that $V(T(v) \cup T(v))$ is a $\frac{2}{3}$-balanced separator of G, where $T(u)$ and $T(v)$ are the root paths of u and v respectively. Moreover u and v can be found in $O(n)$ time. Thorup [78] uses this result on a shortest path spanning tree to find balanced separators that consist of a constant number of shortest paths. Each such separator is called a *shortest path separator*. Unlike the small balanced separator in 2.3.2, shortest path separators can have as many as $O(n)$ vertices. However, the property of shortest path makes it possible to select a small subset of vertices in the separator as portals without introducing too much error/detour.

2.4.2 Vertex Dependent Portal Set

For any two vertices u and v in G, let $Q_{u,v}$ denote a shortest path between u and v. Consider a shortest path Q in a shortest path separator and a shortest path $Q_{u,v}$ that intersects path Q at some vertex p. Now suppose we are willing to accept an additive error of $O(\epsilon d(u, v))$ for $Q_{u,v}$. The general idea is to find two portals p_1, p_2 in Q, such that $d(u, p_1) + d(p_1, p) \leq d(u, p) + \epsilon d(u, v))$ and $d(p, p_2) + d(p_2, v) \leq d(v, p) + \epsilon d(u, v))$. And then $Q_{u,v}$ can be approximated as the concatenation of Q_{u,p_1}, $Q_{p_1,p}$, Q_{p,p_2}, $Q_{p_2,v}$. For a portal p_i and a vertex p in a path Q, we say that p_i *ϵ-covers* p with respect to a vertex u if for any shortest path $Q_{u,v}$ that contains p, $d(u, p_i) + d(p_i, p) \leq d(u, p) + \epsilon d(u, v))$ holds. For each path Q in a separator and each vertex u in G, it is remained to find a set of portals $\mathcal{P}(Q, u)$ such that for any $p \in V(Q)$, there is a portal $p \in \mathcal{P}(Q, u)$ that ϵ-covers p with respect to u. Let $p_0 \in V(Q)$ be the vertex in Q that is closest to u. It is observed that [78] any shortest path $Q_{u,v}$ that intersects Q is no shorter than Q_{u,p_0}. Therefore $\mathcal{P}(u, Q)$ can be selected by first adding p_0 as a portal, and then traverse along Q towards each end point of Q and add a vertex p_{i+1} as a portal as long as $d(u, p_i) + d(p_i, p_{i+1}) > d(u, p_{i+1}) + \epsilon d(u, p_0)$, where p_i is the previously added portal. Every time a portal p_i is added, it reduces the value of $d(u, p_i) + d(p_i, a)$ by at least $\epsilon d(u, p_0)$, where a is the corresponding end point of Q. Using this and the fact that for any p_i, $d(u, p_i) \geq d(p_i, p_0) - d(p_0, u)$, the total number of portals is $O(1/\epsilon)$ [78].

By dividing the input graph into regions using shortest path separators recursively, and then storing for each vertex u in G a vertex dependent portal set (as well as the distance to the portals) for each of the $O(\log n)$ separators that separates u from the rest of the graph, Thorup [78] give an efficient $(1+\epsilon)$-approximate distance oracle for planar graphs with $O(n(\log^3 n)/\epsilon^2)$ preprocessing time, $O(n(\log n)/\epsilon)$ space and $O(1/\epsilon)$ query time. Note that when ϵ is a constant, the query time is constant. A similar result is found independently by Klein [59].

Vertex dependent portal set can also be used for planar directed graphs. However, portals can not be selected the same way as in undirected planar graphs because edges are directed and some properties that are used to bound the number of portals no longer hold. To solve this problem, paths of different lengths are handled separately: paths with the same source but different length ranges need to use different vertex dependent portal sets. Let Δ be the sum of lengths of all edges in the given directed graph $\overrightarrow{G}$. For each directed path $\overrightarrow{Q'}$ in $\overrightarrow{G}$, there is a scale $\gamma = 2^i, i = 0, 1, 2, \ldots, 2^i, \ldots, 2^{\lceil \log \Delta \rceil}$, such that $\gamma \leq l(\overrightarrow{Q'}) < 2\gamma$. We say that $\overrightarrow{Q'}$ is in scale γ. For each scale γ, a collection of subgraphs of $\overrightarrow{G}$ are computed to handle the directed shortest paths in scale γ such that for each vertex u and each shortest path $\overrightarrow{Q}$ in the separators the size of the portal set is $O(1/\epsilon)$.

Using this method, Thorup [78] gives a $(1+\epsilon)$-approximate distance oracle for directed graphs with $O(n(\log^3 n)(log\Delta)/\epsilon^2)$ preprocessing time, $O(n(\log n)(\log \Delta)/\epsilon)$ space and $O(1/\epsilon + \log\log \Delta)$ query time.

2.4.3 Global Portal Set

Let Q be a path of length $l(Q)$ in an undirected graph G. Klein and Subramanian [62] show that a set $\mathcal{P}(Q)$ of $O(1/\epsilon)$ equally spaced vertices on Q can be selected as the portals such that for any shortest path that intersects Q, the error/detour caused by using only the portals instead of all the vertices on the Q is no more than $\epsilon l(Q)$. More specifically, for any pair of vertices u and v whose shortest path intersects Q, $d_G(u, v) \leq \min_{p \in \mathcal{P}(Q)} dist_G(u, p) + dist_G(p, v) \leq d_G(u, v) + \epsilon l(Q)$. Note that when using global portal set instead of vertex dependent set, the error/detour introduced is proportional to the length of the path Q on which the portals are computed. Therefore, it is also needed to compute $O(\log \Delta)$ different scales when using global portal sets in planar undirected graphs, where Δ is the sum of the lengths of all edges.

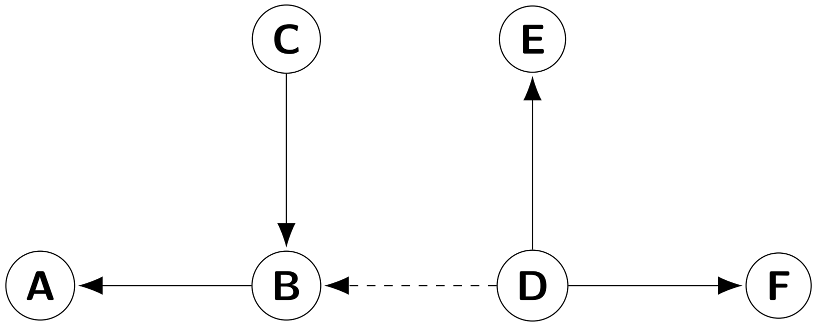
2.4.4 Sparse Neighbourhood Cover

Sparse neighbourhood cover [19] is an essential tool for global portal set to work. To answer the approximate length of a path Q', it needs to be guaranteed that the paths in the shortest path separators are of length $O(l(Q'))$. For $\gamma > 0$, Busch et al. [19] show that a set of subgraphs of G (sparse neighbourhood cover) of total size $O(n)$ can be computed such that the diameter of each subgraph is $O(\gamma)$ and for each vertex $v \in V(G)$ there is at least one subgraph that contains all its neighbours within distance γ. Using this tool with scales $\gamma = 2^i, i = 0, 1, 2, \ldots, 2^i, \ldots, 2^{\lceil \log \Delta \rceil}$, where Δ is the sum of lengths of all edges, and precomputing a distance oracles for each subgraph in each scale, one can handle shortest paths of different lengths. Moreover, it is observe [57] that when computing each scale (and the corresponding distance oracles), edges that are short enough can be contracted as they do not introduce much error. This reduces the total number of edge in all scales from $O(n \log \Delta)$ to $O(n \log n)$ and therefore reduces the total size of the distance oracle. Note that this edge contraction technique can not be applied on directed graphs as it may introduce fake paths that do not exist in the original graph, see Figure 2.5 for an example.

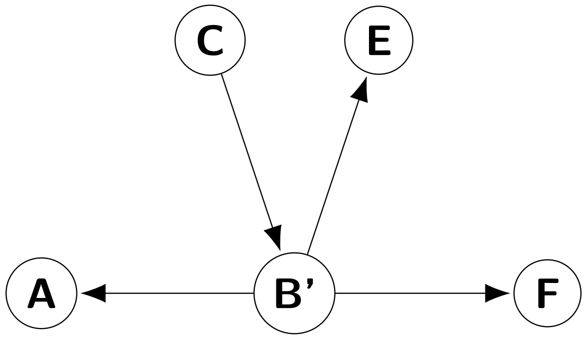
Combining the shortest path separators, global portal sets, sparse neighbourhood cover and cycle MSSP, Kawarabayashi et al. [57] give a $(1 + \epsilon)$-approximate distance oracle for undirected planar graph with $\bar{O}(n\epsilon^{-2} \log^4 n)$ preprocessing time, $\bar{O}(n \log n)$ space and $\bar{O}(1/\epsilon)$ query time, where $\bar{O}$ hides $\log \log n$ and $\log(1/\epsilon)$ factors. This oracle has a better trade-off between space and query time than the oracles in [59, 78].

2.4.5 Classification Scheme

Let G be an undirected planar graph and R be a region of G. Let $Q_{u,v}$ denote a shortest path between vertices u and v in G. Let $\partial R = \{p_1, p_2, \ldots, p_k\}$ be the set of boundary vertices of R. Now consider a vertex $v \in R$ and a vertex $u \in V(G \backslash R)$. Then the shortest path between u and v must contain some boundary vertex $p_i \in \partial R$. Notice that p_i is completely determined by the distances between v and each $p_j, j = 1, 2, \ldots, k$ and the distances between u and each $p_j, j = 1, 2, \ldots, k$. More specifically, $d_G(u, p_i) - d_G(u, p_j) \leq d_G(v, p_j) - d_G(v, p_i)$ holds for every $j, j = 1, 2, \ldots, k$. Therefore for any two vertices u_1 and u_2 in $G \backslash R$ such that $d_G(u_1, p_j) - d_G(u_1, p_1) = d_G(u_2, p_j) - d_G(u_2, p_1), j = 1, 2, \ldots, k$ holds, there is a $p_{i'} \in \partial R$ that is in both Q_{v,u_1} and Q_{v,u_2}. Therefore we can define a map $\phi : u \in (V(G \backslash R)) \to \mathbb{R}^k$ by $\phi(u)[j] = d_G(u, p_j) - d_G(u, p_1)$ such that any two vertices u_1 and u_2 with $\phi(u_1) = \phi(u_2)$ share a same boundary vertex of R on their shortest paths to any vertex in R (notice that the shared boundary vertex differs for different vertices in R). This property can be used to speed up the computation of distances for graphs by classifying the vertices that are mapped into a same value by ϕ into a class and do the time consuming computation such as finding a boundary vertex of R on their path to some vertex in R only once. For this method to work efficiently, it is also required that the vertices outside of R are mapped



(a) A directed graph with a short arc from vertex D to vertex B



(b) The graph obtained from contracting the arc from vertex D to vertex B. The path from vertex C to vertex E in this graph is a fake path that does not exist in the original graph

Figure 2.5: An example showing that the edge contraction technique can not be applied on directed graphs

into a small number of values by ϕ. By adapting the distances between pairs of vertices to $1\epsilon D, 2\epsilon D, \ldots, \frac{1}{\epsilon}\epsilon D$, Weimann and Yuster [80] show that the vertices outside of R can be mapped into $O(2^{O(1/\epsilon)})$ different values while restraining the errors introduced to $O(\epsilon D)$, where D is the diameter of G. This allows them they to compute the $(1+\epsilon)$-approximate diameter of G in $O(n \log^4 n/\epsilon^4 + n2^{O(1/\epsilon)})$ time.

Recall that to compute $(1+\epsilon)$-approximate distance for any pair of vertices u and v we are willing to accept an additive error of $O(\epsilon d(u,v))$. Also recall that sparse neighbourhood covers can be used to compute a set of smaller graphs with different diameters to handle shortest paths of different lengths. Now suppose we are computing $(1+\epsilon)$-approximate distances between vertices u and v in a graph G' with diameter $D' = O(d(u,v))$, the distances between pairs of vertices in G' can be adapted to $1\epsilon D', 2\epsilon D', \ldots, \frac{1}{\epsilon}\epsilon D'$ while restraining the errors introduced to $O(\epsilon d(u,v))$. Therefore the classification scheme can be adopted in computing $(1+\epsilon)$-approximate distance oracles.

Chapter 3

Imporoving Preprocessing Time for Exact Distance Oracle for Planar Graphs

3.1 Introduction

Planar graphs are known to have small balanced cycle separators and this property is heavily relied on in developing distance oracles for planar graphs. A commonly used tool for exact distance oracle for planar graphs is the r-division with few holes, in which the given planar graph G is decomposed into $O(n/r)$ regions such that each region has size $O(r)$, $O(\sqrt{r})$ common vertices with the rest of the graph and these common vertices lie on a constant number of faces of the region. A recursive r-division with few holes can be computedin $O(n)$ time [61].

It is observed that for a planar graph with branchwidth k, the r-division with few holes can be improved such that each region has $O(\min\{\sqrt{r}, k\})$ common vertices with the rest of the graph. This is called the $[r, k]$-division. Based on the $[r, k]$-division, Mozes and Sommer give a distance oracle for weighted directed graphs with size $O(S)$, query time $O(\min\{k \log n \log^2 k \log\log k, nS^{-1/2} \log^{3.5} n\}$ and preprocessing time $O(T(n, k) \log n + S \log^2 n)$ for $S \in [n \log\log k, n^2]$ [65], where $T(n, k)$ is the time complexity of computing a branch-decomposition with width $O(k)$. This distance oracle has better query time for graph with branchwidth $O(n^{o(1)})$ for $S \in [n \log\log k, n^{1.5})$ than the oracle in [38]. For planar graphs, we improve the time complexity of computing a constant-factor approximate branch-decomposition from $O(\min\{n^{1+\epsilon} \log k, nk^3 \log k\})$, where $\epsilon > 0$ is a constant, to $O(\min\{n \log^3 n \log k, nk^2 \log k\})$. Therefore we improves the preprocessing time for Mozes and Sommer's distance oracle to $O(\min\{O(n \log^4 n \log k), O(nk^2 \log n \log k)\}$ for $S \in [n \log\log k, \min\{n \log^2 n \log k, nk^2 \log k / \log n\}]$.

Now we focus on the improvement of the time complexity of computing a constant-factor approximate branch-decomposition for planar graphs. We first introduce some related results. Fast algorithms for computing small width branch-/tree-decompositions of planar graphs have received much attention. Given a planar graph G of n vertices, Tamaki gives an $O(n)$ time heuristic algorithm for branch-decomposition [76]. Gu and Tamaki give an algorithm that for an input planar graph G of n vertices and an integer k, either constructs a branch-decomposition of G with width at most $(c+1+\delta)k$ or outputs $\mathrm{bw}(G) > k$ in $O(n^{1+\frac{1}{c}})$ time, where c is any fixed positive integer and $\delta > 0$ is a constant [40]. By this algorithm and a binary search, a branch-decomposition with width at most $(c+1+\delta)k$ can be computed in $O(n^{1+\frac{1}{c}} \log k)$ time, where $k = \mathrm{bw}(G)$. Kammer and Tholey give an algorithm that for input G and k, either constructs a tree-decomposition of G with width $O(k)$ or outputs $\mathrm{tw}(G) > k$ in $O(nk^3)$ time [53, 54]. The time complexity of the algorithm is improved to $O(nk^2)$ later [55]. This implies that a tree-decomposition with width $O(k)$ can be computed in $O(nk^2 \log k)$ time, where $k = \mathrm{tw}(G)$. Computational study on branch-decomposition can be found in [13, 14, 15, 49, 50, 74, 76]. Fast constant-factor approximation algorithms for branch-/tree-decompositions of planar graphs have important applications such as that in shortest distance oracles in planar graphs [65].

Grid minor of graphs is another notion in graph minor theory [70]. A $k \times k$ grid is a Cartesian product of two paths, each of k vertices. For a graph G, let $\mathrm{gm}(G)$ be the largest integer k such that G has a $k \times k$ grid as a minor. Computing a large grid minor of a graph is important in algorithmic graph minor theory and bidimensionality theory [30, 29, 70]. It is shown in [70] that $\mathrm{gm}(G) \leq \mathrm{bw}(G) \leq 4\mathrm{gm}(G)$ for planar graphs. Gu and Tamaki improve the linear bound $\mathrm{bw}(G) \leq 4\mathrm{gm}(G)$ to $\mathrm{bw}(G) \leq 3\mathrm{gm}(G)$ and show that for any $a < 2$, $\mathrm{bw}(G) \leq a\mathrm{gm}(G)$ does not hold for planar graphs [41]. Other studies on grid minor size and branchwidth/treewidth of planar graphs can be found in [16, 39]. The upper bound $\mathrm{bw}(G) \leq 3\mathrm{gm}(G)$ is a consequence of a result on cylinder minors. A $k \times h$ cylinder is a Cartesian product of a cycle of k vertices and a path of h vertices. For a graph G, let $\mathrm{cm}(G)$ be the largest integer k such that G has a $k \times \left\lceil \frac{k}{2} \right\rceil$ cylinder as a minor. It is shown in [41] that $\mathrm{cm}(G) \leq \mathrm{bw}(G) \leq 2\mathrm{cm}(G)$ for planar graphs. The $O(n^{1+\frac{1}{c}})$ time algorithm in [40] actually constructs a branch-decomposition of G with width at most $(c+1+\delta)k$ or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor.

We propose an $\tilde{O}(n)$ time constant-factor approximation algorithm for branch-/tree-decompositions of planar graphs. Our main result is as follows.

Theorem 3.1.1. *There is an algorithm that given a planar graph G of n vertices and an integer k, in $\min\{O(n \log^3 n), O(nk^2)\}$ time either constructs a branch-decomposition of G with width at most $(2+\delta)k$, where $\delta > 0$ is a constant, or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G.*

The $O(n \log^3 n)$ time result is randomized and can be made deterministic with an additional $\log^3 n$ factor in the running time. Theorem 3.1.1 gives an $\tilde{O}(n)$ time constant-factor approximation for branchwidth/treewidth and largest grid/cylinber minor of planar graphs. Since a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder has branchwidth at least $k+1$ [41], a cylinder minor given in Theorem 3.1.1 implies $\mathrm{bw}(G) > k$. Therefore, for a planar graph G and $k = \mathrm{bw}(G)$, by Theorem 3.1.1 and a binary search, a branch-decomposition of G with width at most $(2+\delta)k$ can be computed in $\min\{O(n\log^3 n\log k), O(nk^2\log k)\}$ time.

Corollary 3.1.1.1. *There is an algorithm that given a planar graph G of n vertices, in $\min\{O(n\log^3 n\log k), O(nk^2\log k)\}$ time constructs a branch-decomposition of G with width at most $(2+\delta)k$, where $\delta > 0$ is a constant and $k = \mathrm{bw}(G)$.*

This improves the previous result of a branch-decomposition with width at most $(c + 1+\delta)k$ in $O(n^{1+\frac{1}{c}}\log k)$ time [40]. It is shown (Statement (5.1)) in [73] that given a branch-decomposition of G with width k, a tree-decomposition of G with width at most $1.5k$ can be computed in $O(nk)$ time. By this result, Theorem 3.1.1 implies an algorithm that for an input planar graph G and an integer k, in $\min\{O(nk + n\log^3 n), O(nk^2)\}$ time constructs a tree-decomposition of G with width at most $(3+\delta)k$ or outputs $\mathrm{tw}(G) > k$. Similarly, for a planar graph G and $k = \mathrm{tw}(G)$, a tree-decomposition with width at most $(3+\delta)k$ can be computed in $\min\{O(nk + n\log^3 n\log k), O(nk^2\log k)\}$ time.

Corollary 3.1.1.2. *There is an algorithm that given a planar graph G of n vertices, in $\min\{O(nk + n\log^3 n\log k), O(nk^2\log k)\}$ time constructs a tree-decomposition of G with width at most $(3+\delta)k$, where $\delta > 0$ is a constant and $k = \mathrm{tw}(G)$.*

Kammer and Tholey give an algorithm that computes a tree-decomposition of G with width at most $48k + 13$ in $O(nk^3\log k)$ time or with width at most $(9+\delta)k + 9$ in $O(n\min\{\frac{1}{\delta}, k\}k^3\log k)$ time (where $0 < \delta < 1$) [53, 54]. They also give an algorithm for computing the weighted treewidth for vertex weighted planar graphs [55]. Applying this algorithm to a planar graph G, a tree-decomposition of G with width at most $(15+\delta)k+O(1)$ can be computed in $O(nk^2\log k)$ time. This improves the result of [53, 54]. Our $O(nk^2\log k)$ time algorithm is an independent improvement over the result of [53, 54][1] and has a better approximation ratio than that of [55]. Our algorithm can be used to compute a $g\times\lceil\frac{g}{2}\rceil$ cylinder (grid) minor with $g = \frac{\mathrm{bw}(G)}{\beta}$, where $\beta > 2$ is a constant, and a $g\times g$ cylinder (grid) minor with $g = \frac{\mathrm{bw}(G)}{\beta}$, where $\beta > 3$ is a constant, of G in $\min\{O(n\log^3 n\log k), O(nk^2\log k)\}$ time. This improves the previous results of $g\times\lceil\frac{g}{2}\rceil$ with $g \geq \frac{\mathrm{bw}(G)}{\beta}$, $\beta > (c+1)$, and $g\times g$ with $g \geq \frac{\mathrm{bw}(G)}{\beta}$, $\beta > (2c+1)$, in $O(n^{1+\frac{1}{c}}\log k)$ time.

Our algorithm for Theorem 3.1.1 uses the approach in the previous work of [40] described below. Given a planar graph G and an integer k, let $\mathcal{Z}$ be the set of biconnected components of G of a normal distance (a definition is given in Section 3.2) $h = ak$, where $a > 0$ is a constant, from a selected edge e_0 of G. For each $Z \in \mathcal{Z}$, a minimum vertex cut set $\partial(A_Z)$ which partitions $E(G)$ into edge subsets A_Z and $\overline{A}_Z = E(G)\setminus A_Z$ is computed such that

[1]The $O(nk^2\log k)$ time algorithm in [55] was first announced in July 2015 [56] while our result was reported in March 2015 [46].

all edges in Z are in A_Z and $e_0 \in \overline{A}_Z$, that is, $\partial(A_Z)$ separates Z and e_0. If $|\partial(A_Z)| > k$ for some $Z \in \mathcal{Z}$ then $\mathrm{bw}(G) > k$ is concluded. Otherwise, a branch-decomposition of graph H obtained from G by removing all A_Z is constructed. For each subgraph $G[A_Z]$ induced by A_Z, a branch-decomposition is constructed or $\mathrm{bw}(G[A_Z]) > k$ is concluded recursively. Finally, a branch-decomposition of G with width $O(k)$ is constructed from the branch-decomposition of H and those of $G[A_Z]$ or $\mathrm{bw}(G) > k$ is concluded.

The algorithm in [40] computes a minimum vertex cut set $\partial(A_Z)$ for every $Z \in \mathcal{Z}$ in all recursive steps in $O(n^{1+\frac{1}{c}})$ time. Our main idea for proving Theorem 3.1.1 is to find a minimum vertex cut set $\partial(A_Z)$ for every $Z \in \mathcal{Z}$ more efficiently based on recent results for computing minimum face-separating cycles and vertex cut sets in planar graphs. Borradaile et al. give an algorithm that in $O(n \log^4 n)$ time computes an oracle for the all pairs minimum face-separating cycle problem in a planar graph G [18]. The time for computing the oracle is further improved to $O(n \log^3 n)$ [17]. For any pair of faces f and g in G, the oracle in $O(|C|)$ time returns a minimum (f, g)-separating cycle C (i.e., C cuts the sphere on which G is embedded into two regions, one contains f and the other contains g). By this result, we show that a minimum vertex cut set $\partial(A_Z)$ for every $Z \in \mathcal{Z}$ in all recursive steps can be computed in $O(n \log^3 n)$ time and get the first result.

Theorem 3.1.2. *There is an algorithm that given a planar graph G of n vertices and an integer k, in $O(n \log^3 n)$ time either constructs a branch-decomposition of G with width at most $(2+\delta)k$ or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G, where $\delta > 0$ is a constant.*

For an input G and integer k, Kammer and Tholey give algorithms that construct a tree-decomposition with width $O(k)$ or outputs $\mathrm{tw}(G) > k$ as follows [53, 54, 55] (related definitions are given in Section 3.4): Convert G into an almost triangulated planar graph $\hat{G}$. Use *crest separators* to decompose $\hat{G}$ into subgraphs such that each subgraph contains one *crest* with a normal distance k from a selected set of edges called the *coast*. For each crest compute a vertex cut set of size at most $3k-1$ to separate the crest from the coast. If such a vertex cut set can not be found for some crest then the algorithms conclude $\mathrm{tw}(\hat{G}) > k$. Otherwise, the algorithms compute a tree-decomposition for the graph $\hat{H}$ obtained by removing all crests from $\hat{G}$ and works on each crest recursively. Finally, the algorithms construct a tree-decomposition of $\hat{G}$ from the tree-decomposition of $\hat{H}$ and those of the crests.

Viewing each biconnected component $Z \in \mathcal{Z}$ as a crest and e_0 as the coast, we get an $O(nk^2)$ time algorithm for Theorem 3.1.1 by applying the ideas of triangulating G and crest separators in [53, 54] to decompose $\hat{G}$ into subgraphs such that each subgraph contains one $Z \in \mathcal{Z}$. Instead of finding a vertex cut set of size at most $3k-1$ for each crest, we apply minimum face-separating cycle techniques to find a minimum vertex cut set $\partial(A_Z)$ in each subgraph. We show that either a vertex cut set $\partial(A_Z)$ with $|\partial(A_Z)| \leq k$ for every $Z \in \mathcal{Z}$

in all recursive steps or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor can be computed in $O(nk^2)$ time and get the result below.

Theorem 3.1.3. *There is an algorithm that given a planar graph G of n vertices and an integer k, in $O(nk^2)$ time either constructs a branch-decomposition of G with width at most $(2+\delta)k$ or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G, where $\delta > 0$ is a constant.*

Theorem 3.1.1 follows from Theorems 3.1.2 and 3.1.3.

The next section gives the preliminaries of this chapter. We prove Theorems 3.1.2 and 3.1.3 in Sections 3.3 and 3.4, respectively. The final section concludes the chapter.

3.2 Definitions and Notations

In this chapter, it is convenient to view a vertex cut set $\partial(A_Z)$ in a graph as an edge in a *hypergraph* in some cases.

Definition 3.2.1. *A* hypergraph *H is an ordered pair $H = (V, E)$ with a set V of vertices and a set E of edges, where each edge $e \in E$ is a subset of V with at least two elements. H is a* graph *if every $e \in E'$ has exactly two elements.*

A *biconnected component* of a graph G is a maximal biconnected subgraph of G. It suffices to prove Theorems 3.1.2 and 3.1.3 for a biconnected graph G because if G is not biconnected, the problems of finding branch-decompositions and cylinder minors of G can be solved individually for each biconnected component.

Definition 3.2.2. *For a plane graph G and vertices $u, v \in V(G)$, the* normal distance *$\mathrm{nd}_G(u,v)$ is the smallest length of a normal curve between $\rho(u)$ and $\rho(v)$. The* normal distance *between two vertex-subsets $U, W \subseteq V(G)$ is $\mathrm{nd}_G(U,W) = \min_{u \in U, v \in W} \mathrm{nd}_G(u,v)$.*

Figure 3.1 (a) shows a normal curve and the normal distance between two vertices in a plane graph. We also use $\mathrm{nd}_G(U,v)$ for $\mathrm{nd}_G(U,\{v\})$ and $\mathrm{nd}_G(u,W)$ for $\mathrm{nd}_G(\{u\},W)$. A noose ν of G separates Σ into two open regions R_1 and R_2 and induces a separation $(A, \overline{A})$ of G with $A = \{e \in E(G) \mid \rho(e) \subseteq R_1\}$ and $\overline{A} = \{e \in E(G) \mid \rho(e) \subseteq R_2\}$. We also say ν induces edge subset A (or $\overline{A}$) (see Figure 3.1 (a) for an example). A separation (resp. an edge subset) of G is called *noose-induced* if there is a noose which induces the separation (resp. edge subset). A noose ν separates two edge subsets A_1 and A_2 if ν induces a separation $(A, \overline{A})$ with $A_1 \subseteq A$ and $A_2 \subseteq \overline{A}$. We also say that the noose-induced subset A separates A_1 and A_2.

Recall that a noose is a closed normal curve on Σ that does not intersect itself. For a plane graph G and a noose ν induced subset $A \subseteq E(G)$, we denote by $G|A$ the *plane hypergraph* obtained by replacing all edges of A with edge $\partial(A)$ (i.e., $V(G|A) = (V(G) \setminus V(A)) \cup \partial(A)$ and $E(G|A) = (E(G) \setminus A) \cup \{\partial(A)\}$). An embedding of $G|A$ can be obtained from that

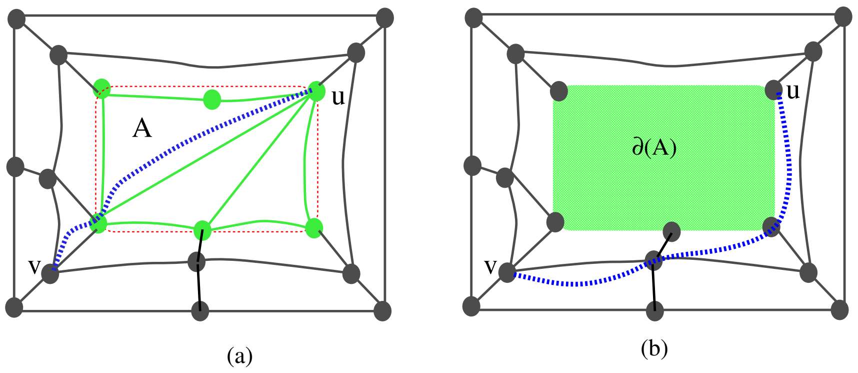


Figure 3.1: (a) A plane graph G. Solid segments are edges of G. A normal curve with the smallest length between vertices u and v in G is expressed by blue thick dashed segments. The normal distance $\mathrm{nd}_G(u, v) = 2$. A noose ν in G is expressed by red thin dashed segments. Noose ν induces an edge subset A and the edges of A are expressed by green solid segments. (b) A plane hypergraph $G|A$. A normal curve with the smallest length between vertices u and v in $G|A$ is expressed by blue thick dashed segments. The normal distance $\mathrm{nd}_{G|A}(u, v) = 3$.

of G with $\rho(\partial(A))$ an open disk (homeomorphic to $\{(x, y)|x^2 + y^2 < 1\}$) that is the open region separated by ν and contains A. For a collection $\mathcal{A} = \{A_1, .., A_r\}$ of mutually disjoint noose induced edge-subsets of G, $(..(G|A_1)|..)|A_r$ is denoted by $G|\mathcal{A}$. The *normal curve* and *normal distances* in a plane hypergraph are defined from the definitions for the normal curve and normal distances in a plane graph by replacing a plane graph with a plane hypergraph. Figure 3.1 (b) gives a normal curve and the normal distance between two vertices in a plane hypergraph.

3.3 $O(n \log^3 n)$ Time Algorithm

We give an algorithm to prove Theorem 3.1.2. Our algorithm follows the approach of the work in [40]. Let G be a plane graph (hypergraph) of n vertices, e_0 be a given edge of G, and $k, h > 0$ be integers. Let $\mathcal{Z}$ be the set of biconnected components of the subgraph of G induced by the vertices with normal distance at least h. We first try to separate e_0 and the set $\mathcal{Z}$.

Definition 3.3.1. *A* good-separator *$\mathcal{A}$ for $\mathcal{Z}$ and e_0 is a set of noose-induced subsets with the following properties:*

1. *for every $A_Z \in \mathcal{A}$, $|\partial(A_Z)| \leq k$;*
2. *for every $A_Z \in \mathcal{A}$, $G|\overline{A_Z}$ is biconnected;*
3. *for every $Z \in \mathcal{Z}$, there is an $A_Z \in \mathcal{A}$ that contains Z and separates Z and e_0;*
4. *for distinct $A_Z, A_{Z'} \in \mathcal{A}$, $A_Z \cap A'_Z = \emptyset$.*

For each $Z \in \mathcal{Z}$, our algorithm computes a minimum noose (and the noose-induced subset A_Z that contains Z) separating Z and e_0. If for some $Z \in \mathcal{Z}$, $|\partial(A_Z)| > k$ then the algorithm constructs a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G in $O(n)$ time by Lemma 3.3.1 proved in [40]. Otherwise, a good-separator $\mathcal{A}$ is computed.

Lemma 3.3.1. *[40] Given a plane graph G and integers $k, h > 0$, let X and Z be edge subsets of G satisfying the following conditions: (1) each of separations $(X, \overline{X})$ and $(Z, \overline{Z})$ is noose-induced; (2) $G[Z]$ is biconnected; (3) $\mathrm{nd}_G(V(\overline{X}), V(Z)) \geq h$; and (4) every noose of G that separates $\overline{X}$ and Z has length $> k$. Then G has a $(k+1) \times h$ cylinder minor and given $(G|\overline{X})|Z$, such a minor can be constructed in $O(|V(X \cap \overline{Z})|)$ time.*

Given a good-separator $\mathcal{A}$ for $\mathcal{Z}$ and e_0, our algorithm constructs a branch-decomposition of the plane hypergraph $G|\mathcal{A}$ with width at most $k + 2h$ by Lemma 3.3.2 shown in [41, 76]. For each $A_Z \in \mathcal{A}$, the algorithm computes a cylinder minor or a branch-decomposition for the plane hypergraph $G|\overline{A}_Z$ recursively. If a branch-decomposition of $G|\overline{A}_Z$ is found for every $A_Z \in \mathcal{A}$, the algorithm constructs a branch-decomposition of G with width at most $k + 2h$ from the branch-decomposition of $G|\mathcal{A}$ and those of $G|\overline{A}_Z$ by Lemma 3.3.3 which is straightforward from the definitions of branch-decompositions.

Lemma 3.3.2. *[41, 76] Let $k > 0$ and $h > 0$ be integers. Let G be a plane hypergraph with each edge of G incident to at most k vertices. If there is an edge e_0 such that for any vertex v of G, $\mathrm{nd}_G(e_0, v) \leq h$ then given e_0, a branch-decomposition of G with width at most $k+2h$ can be constructed in $O(|V(G)| + |E(G)|)$ time.*

The upper bound $k + 2h$ is shown in Theorem 3.1 in [41]. The normal distance in [41] between a pair of vertices is twice of the normal distance in this paper between the same pair of vertices. Tamaki gives a linear time algorithm to construct a branch-decomposition with width at most $k + 2h$ [76].

The following lemma is straightforward from the definition of branch-decompositions and allows us to bound the width of the branch-decomposition of the whole graph.

Lemma 3.3.3. *Given a plane hypergraph G and a noose-induced separation $(A, \overline{A})$ of G, let T_A and $T_{\overline{A}}$ be branch-decompositions of $G|\overline{A}$ and $G|A$ respectively. Let $T_A + T_{\overline{A}}$ to be the tree obtained from T_A and $T_{\overline{A}}$ by joining the link incident to the leaf $\partial(A)$ in T_A and the link incident to the leaf $\partial(A)$ in $T_{\overline{A}}$ into one link and removing the leaves $\partial(A)$. Then $T_A + T_{\overline{A}}$ is a branch-decomposition of G with width $\max\{|\partial(A)|, k_A, k_{\overline{A}}\}$ where k_A is the width of T_A and $k_{\overline{A}}$ is the width of $T_{\overline{A}}$.*

To make a concrete progress in each recursive step, the following technique in [40] is used to compute $\mathcal{A}$. For a plane hypergraph G, an edge e_0 of G and an integer $d \geq 0$, let

$$\mathrm{reach}_G(e_0, d) = \bigcup \{v \in V(G) | \mathrm{nd}_G(e_0, v) \leq d\}$$

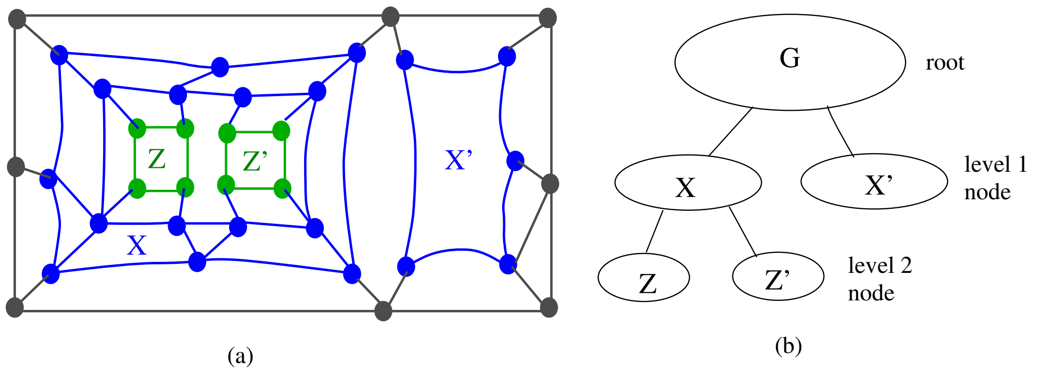


Figure 3.2: (a) A plane graph G with two (level 1) biconnected components X and X'. X contains two (level 2) biconnected components Z and Z'. (b) A layer tree for G and components X, X', Z, Z'.

denote the set of vertices of G with the normal distance at most d from e_0.

Definition 3.3.2. *For an arbitrary constant $\alpha > 0$ and integer $k \geq 2$, the* layer tree $\mathrm{LT}(G, e_0, k, \alpha)$ *is defined as follows:*

- *the root of the tree is G;*
- *each biconnected component X of $G[V(G) \setminus \mathrm{reach}_G(e_0, \lceil \frac{\alpha k}{2} \rceil - 1)]$ is a node in level 1 of the tree and is a child of the root;*
- *each biconnected component Z of $G[V(G) \setminus \mathrm{reach}_G(e_0, h-1)]$, where $h = \lceil \frac{\alpha k}{2} \rceil + \lceil \frac{k+1}{2} \rceil$, is a node in level 2 of the tree and is a child of the biconnected component X in level 1 that contains Z.*

Figure 3.2 gives an example of a layer tree.

Recall that $\mathcal{Z}$ is a set of biconnected components of the subgraph induced by the vertices with normal distance at least h from e_0. Then, $\mathcal{Z}$ is the set of leaf nodes of $\mathrm{LT}(G, e_0, k, \alpha)$ in level 2. For a level 1 node X in $\mathrm{LT}(G, e_0, k, \alpha)$ that is not a leaf, let $\mathcal{Z}_X \subseteq \mathcal{Z}$ be the set of child nodes of X. Based on the plane hypergraph $(G|\overline{X})|\mathcal{Z}_X$, we find a minimum noose in G separating Z and $\overline{X}$ for every $Z \in \mathcal{Z}_X$ to get a good-separator $\mathcal{A}_X$ for $\mathcal{Z}_X$ and $\overline{X}$. If a minimum noose in G separating Z and $\overline{X}$ for some $Z \in \mathcal{Z}_X$ has length $> k$ then we compute a $(k+1) \times \lceil \frac{k+1}{2} \rceil$ cylinder minor of G by Lemma 3.3.1. Otherwise, we compute a good-separator $\mathcal{A}_X$ for $\mathcal{Z}_X$ and $\overline{X}$ and the union of $\mathcal{A}_X$ for every X gives a good-separator $\mathcal{A}$ for $\mathcal{Z}$ and e_0.

Notice that if Z is a single vertex then Z will not be involved in any further recursive step; and if Z is a single edge then there is a noose of length $2 \leq k$ separating Z and $\overline{X}$, and it is trivial to compute the branch-decomposition of Z. So we assume without loss of generality that each $Z \in \mathcal{Z}_X$ has at least three vertices.

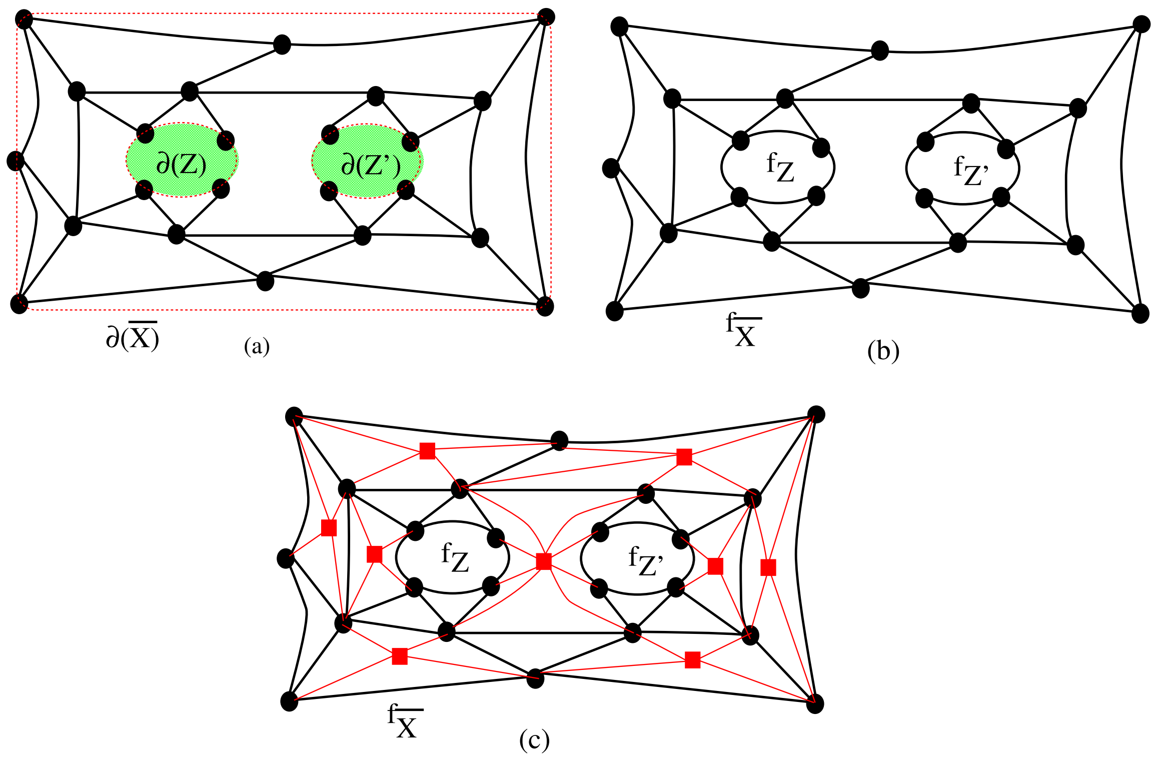


Figure 3.3: (a) A plane hypergraph $(G|\overline{X})|\mathcal{Z}_X$ ($\mathcal{Z}_X = \{Z, Z'\}$). The nooses defining the embeddings of $\partial(\overline{X})$, $\partial(Z)$ and $\partial(Z')$ are expressed by red dashed segments. (b) The plane graph G_X constructed from $(G|\overline{X})|\mathcal{Z}_X$. (c) The weighted plane graph H_X constructed from G_X. The new added vertices and edges are expressed in red.

To compute $\mathcal{A}_X$, we convert $(G|\overline{X})|\mathcal{Z}_X$ to a weighted plane graph H_X and compute a minimum noose (and the noose-induced subset A_Z that contains Z) separating every $Z \in \mathcal{Z}_X$ and $\overline{X}$ by finding a minimum face-separating cycle in H_X. We use the results by Borradaile et al. [17, 18] to compute the face-separating cycles.

We first convert the hypergraph $(G|\overline{X})|\mathcal{Z}_X$ into a plane graph G_X as follows (see Figure 3.3): Remove $\rho(\partial(\overline{X}))$. For each edge $\partial(Z)$ in $(G|\overline{X})|\mathcal{Z}_X$, let ν_Z be a noose which induces the separation $(Z, \overline{Z})$ in G. Then $E_Z = \{\nu_Z \setminus \rho(u) | u \in \partial(Z)\}$ is a set of open segments. Replace edge $\rho(\partial(Z))$ by the set of edges which are segments in E_Z. G_X has a face which contains $\rho(\partial(\overline{X}))$ and we denote this face by $f_{\overline{X}}$. For each $Z \in \mathcal{Z}_X$, the embedding $\rho(\partial(Z))$ of edge $\partial(Z)$ becomes a face f_Z in G_X with $E(f_Z) = E_Z$. A face in G_X that is not $f_{\overline{X}}$ or any of f_Z is called a *natural face* in G_X. Notice that G_X is a minor of G when the embeddings of the graphs are not considered because every $Z \in \mathcal{Z}_X$ is biconnected and has at least three vertices.

Next we convert G_X to a weighted plane graph H_X (see Figure 3.3): For each natural face f in G_X with $|V(f)| > 3$, we add a new vertex u_f and new edges $\{u_f, v\}$ in f for every vertex v in $V(f)$. Each new edge $\{u_f, v\}$ is assigned the weight 1/2. Each edge of G_X is assigned the weight 1. Notice that $|V(H_X)| = O(|V(G_X)|)$.

A noose in G_X is called a *natural noose* if it intersects only natural faces in G_X. It is shown (Lemma 5.1) in [40] that for each $Z \in \mathcal{Z}_X$, there is a minimum noose in G_X separating $E(f_Z)$ and $E(f_{\overline{X}})$ that is a natural noose. If for any $Z \in \mathcal{Z}_X$, a minimum natural noose separating $E(f_Z)$ and $E(f_{\overline{X}})$ has length $> k$, then by Lemma 3.3.1, we can compute a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G_X that is also a minor of G. Otherwise, we find a good-separator $\mathcal{A}_X$ for $\mathcal{Z}_X$ and $\overline{X}$. By Lemma 3.3.4 below, a minimum natural noose ν separating $E(f_Z)$ and $E(f_{\overline{X}})$ can be computed by finding a minimum $(f_Z, f_{\overline{X}})$-separating cycle C in H_X. The subsets of $E(G_X)$ induced by noose ν are also called *cycle C induced subsets.*

Lemma 3.3.4. *Let H_X be the weighted plane graph obtained from G_X. For any $(f_Z, f_{\overline{X}})$-separating cycle C in H_X, there is a natural noose ν that separates $E(f_Z)$ and $E(f_{\overline{X}})$ in G_X with the same length as that of C. For any minimum natural noose ν in G_X separating $E(f_Z)$ and $E(f_{\overline{X}})$, there is a $(f_Z, f_{\overline{X}})$-separating cycle C in H_X with the same length as that of ν.*

Proof. Let C be a $(f_Z, f_{\overline{X}})$-separating cycle in H_X. For each edge $\{u, v\}$ in C with $u, v \in V(G_X)$, $\{u, v\}$ is incident to a natural face f because f_Z is not incident to $f_{\overline{X}}$ by $\text{nd}_{G_X}(V(\overline{X}), V(Z)) = \left\lceil \frac{k+1}{2} \right\rceil$. We draw a simple curve with u, v as its end points in face f. For each pair of edges $\{u, u_f\}$ and $\{u_f, v\}$ in C with $u, v \in V(G_X)$ and $u_f \in V(H_X) \setminus V(G_X)$, we draw a simple curve with u, v as its end points in the face f of G_X where the newly added vertex u_f is placed. Then the union of the curves form a natural noose ν that separates $E(f_Z)$ and $E(f_{\overline{X}})$ in G_X. Each edge $\{u, v\}$ with $u, v \in V(G_X)$ has weight 1. For a newly added vertex u_f, each of edges $\{u, u_f\}, \{u_f, v\}$ has weight 1/2. Therefore, the lengths of ν and C are the same.

Let ν be a minimum natural noose separating $E(f_Z)$ and $E(f_{\overline{X}})$ in G_X. Then ν contains at most two vertices of G_X incident to a same natural face of G_X, otherwise a shorter natural noose separating $E(f_Z)$ and $E_{\overline{X}}$ can be formed. The vertices on ν partition ν into a set of simple curves such that at most one curve is drawn in each natural face of G_X. For a curve with the end points u and v in a natural face f, if $\{u, v\}$ is an edge of G_X then we take $\{u, v\}$ in H_X as a candidate, otherwise we take edges $\{u, u_f\}, \{u_f, v\}$ in H_X as candidates, where u_f is the vertex added in f in getting H_X. These candidates form a $(f_Z, f_{\overline{X}})$-separating cycle C in H_X. Because each edge of G_X has weight 1 and each newly added edge has weight 1/2 in H_X, the lengths of C and ν are the same. $\square$

We use the results by Borradaile et al. for computing minimum face separating cycles [17, 18] to compute $\mathcal{A}_X$ for every X.

Definition 3.3.3. *A* minimum cycle basis tree (MCB tree) *[18] for a plane graph G is an edge-weighted tree $\tilde{T}$ such that*

- *there is a bijection from the faces of G to the nodes of $\tilde{T}$;*
- *removing each edge e from $\tilde{T}$ partitions $\tilde{T}$ into two subtrees $\tilde{T}_1$ and $\tilde{T}_2$; this edge e corresponds to a cycle that separates every pair of faces f and g with f in $\tilde{T}_1$ and g in $\tilde{T}_2$; and*
- *for any distinct faces f and g, the minimum-weight edge on the unique path between f and g in $\tilde{T}$ has a weight equal to the length of a minimum (f, g)-separating cycle.*

It is shown in [18] that an MCB tree $\tilde{T}$ of a plane graph G can be computed in $O(n \log^4 n)$ time and a minimum (f, g)-separating cycle C can be found in $O(|C|)$ time, where $|C|$ is the number of edges in C, from $\tilde{T}$ for distinct faces f and g in G. By the result in [17], the time for computing $\tilde{T}$ is improved to $O(n \log^3 n)$. The next lemma gives a base for our $O(n \log^3 n)$ time algorithm.

Lemma 3.3.5. *[17, 18] Given a plane graph G of n vertices with positive edge weights, a MCB tree of G can be computed in $O(n \log^3 n)$ time. Further, for any distinct faces f and g in G, given a minimum weight edge in the path between f and g in the MCB tree, a minimum (f, g)-separating cycle C can be obtained in $O(|C|)$ time, $|C|$ is the number of edges in C.*

A cycle C in H_X partitions Σ into two regions and one region contains face $f_{\overline{X}}$. We denote by $\text{ins}(C)$ the region that does not contain $f_{\overline{X}}$. The minimum face separating cycles in the MCB tree of H_X computed by Lemma 3.3.5 have the following property which is important for computing a good-separator $\mathcal{A}_X$.

Observation 1. *[17, 18] Let $\tilde{T}$ be an MCB tree of H_X computed using Lemma 3.3.5. For any $Z, Z' \in \mathcal{Z}_X$, let C be the minimum $(f_Z, f_{\overline{X}})$-separating cycle and C' be the minimum $(f_{Z'}, f_{\overline{X}})$-separating cycle obtained from $\tilde{T}$. Then $\text{ins}(C) \cap \text{ins}(C') = \emptyset$ or $\text{ins}(C) \subseteq \text{ins}(C')$ or $\text{ins}(C') \subseteq \text{ins}(C)$.*

Lemma 3.3.5 requires that there is a unique shortest path between any two vertices in the input graph [17, 18]. This requirement can be guaranteed with high probability by a random perturbation on the edge weight of the input graph, making the $O(n \log^3 n)$ time for computing the MCB tree randomized [17], or guaranteed deterministically, increasing the time for computing the MCB tree to $O(n \log^6 n)$ [18]. Using Lemma 3.3.5 to compute an MCB tree $\tilde{T}$ of H_X and thus $\mathcal{A}_X$, our algorithm is summarized in Algorithm 1 below.

Now we prove Theorem 3.1.2 which is re-stated below. The $O(n \log^3 n)$ time result in the theorem is randomized and can be made deterministic with an additional $\log^3 n$ factor in the running time.

Theorem 3.1.2. *There is an algorithm that given a planar graph G of n vertices and an integer k, in $O(n \log^3 n)$ time either constructs a branch-decomposition of G with width at most $(2 + \delta)k$, $\delta > 0$ is a constant, or a $(k + 1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G.*

Algorithm 1: Branch-Minor-MCB-Based$(G|U, \partial(U), k, \alpha)$

```
Input: Integer $k \geq 2$; constant $\alpha > 0$; a biconnected plane hypergraph $G|U$ with
       $\partial(U)$ specified, where $|\partial(U)| \leq k$ and every other edge has two vertices.
Output: Either a branch-decomposition of $G|U$ with width at most $k + 2h$, where
        $h = \lceil \frac{\alpha k}{2} \rceil + \lceil \frac{k+1}{2} \rceil$, or a $(k+1) \times \lceil \frac{k+1}{2} \rceil$ cylinder minor of $G$.
1  if $\mathrm{nd}_{G|U}(\partial(U), v) \leq h$ for every $v \in V(G|U)$ then
2      return a branch-decomposition of $G|U$ with width at most $k + 2h$ by Lemma
       3.3.2;
3  else
4      compute the layer tree $\mathrm{LT}(G|U, \partial(U), k, \alpha)$;
5      for every level 1 node $X$ of $\mathrm{LT}(G|U, \partial(U), k, \alpha)$ that is not a leaf do
6          compute $G_X, H_X$ from $(G|\overline{X})|\mathcal{Z}_X$, where $\mathcal{Z}_X$ is the set of child nodes of $X$;
7          compute an MCB tree $\tilde{T}$ of $H_X$ by Lemma 3.3.5;
8          while $\tilde{T}$ contains any $f_Z$ for $Z \in \mathcal{Z}_X$ do
9              find a face $f_Z$, $Z \in \mathcal{Z}_X$, in $\tilde{T}$ by a breadth first search from $f_{\overline{X}}$ such that
               the path between $f_Z$ and $f_{\overline{X}}$ in $\tilde{T}$ does not contain $f_{Z'}$ for any $Z' \in \mathcal{Z}_X$
               with $Z' \neq Z$;
10             find the minimum weight edge $e_Z = \{u, v\}$ in the path between $f_Z$ and
               $f_{\overline{X}}$, and the cycle $C$ from edge $e_Z$;
11             if $C$ has length $> k$ then
12                 return a $(k+1) \times \lceil \frac{k+1}{2} \rceil$ cylinder minor by Lemma 3.3.1;
13             else
14                 compute the cycle $C$ induced subset $A_Z$ that contains $E(f_Z)$ and
                   include $A_Z$ to $\mathcal{A}_X$; for each node $f$ of $\tilde{T}$, if edge $e_Z$ is in the path
                   between $f$ and $f_{\overline{X}}$ in $\tilde{T}$ then delete $f$ from $\tilde{T}$;
15             end
16         end
17     end
18     Let $\mathcal{A} = \cup_{X:X \text{ is a level 1 node of } \mathrm{LT}(G|U,\partial(U),k,\alpha)} \mathcal{A}_X$;
19     for $A \in \mathcal{A}$ do
20         Let $T_A$ = Branch-Minor-MCB-Based$(G|\overline{A}, \partial(\overline{A}), k, \alpha)$;
21         if $T_A$ is a $(k+1) \times \lceil \frac{k+1}{2} \rceil$ cylinder minor then
22             return $T_A$;
23         end
24     end
25     construct a branch-decomposition $T_0$ of $(G|U)|\mathcal{A}$ with width at most $k + 2h$ by
       Lemma 3.3.2;
26     combine $T_0$ and $T_A$, $A \in \mathcal{A}$ into a branch-decomposition $T$ of $G|U$ with width at
       most $k + 2h$ by Lemma 3.3.3;
27     return $T$
28 end
```

Proof. The input hypergraph $G|\overline{A}$ of Algorithm 1 in each recursive step for $A \in \mathcal{A}$ is biconnected. In Line 12 the algorithm returns a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G. For the $\mathcal{A}_X$ computed in Lines $5-17$, obviously we have

1. for every $A_Z \in \mathcal{A}_X$, $|\partial(A_Z)| \leq k$;
2. because $G|U$ is biconnected and $\partial(A_Z) \in E((G|U)|\overline{A_Z})$, $(G|U)|\overline{A_Z}$ is biconnected;
3. from the way we find the cycles from the MCB tree, for every $Z \in \mathcal{Z}_X$, there is exactly one noose-induced subset $A_Z \in \mathcal{A}_X$ separating Z and $\overline{X}$;
4. from the unique shortest path in H_X, for distinct $A_Z, A_{Z'} \in \mathcal{A}_X$, $A_Z \cap A_{Z'} = \emptyset$.

Therefore, by Definition 1, $\mathcal{A}_X$ is a good-separator for $\mathcal{Z}_X$ and $\overline{X}$. From this, $\mathcal{A}$ is a good separator for $\mathcal{Z}$ and U and Algorithm 1 computes a branch-decomposition or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G. The width of the branch-decomposition computed is at most

$$k + 2h = k + 2(\left\lceil \frac{\alpha k}{2} \right\rceil + \left\lceil \frac{k+1}{2} \right\rceil) \leq k + 2(\left\lceil \frac{\alpha k}{2} \right\rceil) + (k+2) \leq (2+\delta)k,$$

where δ is the smallest constant with $\delta k \geq \alpha k + 4$.

Let M, m_x and m be the numbers of edges in $G[\text{reach}_{G|U}(\partial(U), h)]$, $(G|\overline{X})|\mathcal{Z}_X$ and H_X, respectively. Then $m = O(m_x)$. In Line 4, the layer tree $\text{LT}(G|U, \partial(U), k, \alpha)$ can be computed in $O(M)$ time. For each level 1 node X, it takes $O(m)$ time to compute $\partial(\overline{X})$, G_X and H_X (Line 6). By Lemma 3.3.5, it takes $O(m \log^3 m)$ time to compute an MCB tree $\tilde{T}$ of H_X (Line 7). All executions of Line 9 take $O(m)$ time. It takes $O(m)$ time to compute a cylinder minor by Lemma 3.3.1 (Line 12). From Property 4 of a good-separator (Definition 1), each edge of H_X appears in at most two cycles that induce the subsets in $\mathcal{A}_X$. So it takes $O(m)$ time to compute $\mathcal{A}_X$ (Lines $8-16$). Therefore, the total time for Lines $6-16$ is $O(m \log^3 m)$. For distinct level 1 nodes X and X', the edge sets of subgraphs $(G|\overline{X})|\mathcal{Z}_X$ and $(G|\overline{X'})|\mathcal{Z}_{X'}$ are disjoint. So $\sum_{X:X \text{ is a level 1 node}} m_x = O(M)$. Therefore, the total time for Lines $4-17$ is

$$\sum_{X:X \text{ is a level 1 node of } \text{LT}(G|U,\partial(U),k,\alpha)} O(m_x \log^3 m_x) = O(M \log^3 M).$$

The time for other steps in Algorithm 1 is $O(M)$. In every recursion call, only the vertices in $\text{reach}_{G|U}(\partial(U), h)$ are involved. So the number of recursive calls in which each vertex of $G|U$ is involved is $O(\frac{1}{\alpha}) = O(1)$. Therefore, the running time of the algorithm is $O(n \log^3 n)$. □

3.4 $O(nk^2)$ Time Algorithm

To get an algorithm for Theorem 3.1.3, we follow the framework of Algorithm 1 in Section 3.3 but use a different approach from that in Lines 7-16 to compute face-separating cycles and a good separator for $\mathcal{Z}_X$ and $\overline{X}$. We first introduce some notions for our approach.

Given $\mathcal{Z}_X$ and H_X as those in Algorithm 1, for each $Z \in \mathcal{Z}_X$ the edges incident to face f_Z in H_X form a $(f_Z, f_{\overline{X}})$-separating cycle, denoted by C_Z and called the *boundary cycle* of Z. Notice that $|E(C_Z)| = |\partial(Z)|$. Given a plane graph G, we choose an arbitrary face as the outer face f_0 of G. A cycle C in G partitions Σ into two regions and exactly one region contains f_0. Let $\text{ins}(C)$ denote the region that does not contain f_0.

Definition 3.4.1. *Two cycles C and C' in a plane graph* cross with *each other if they satisfy the following conditions:*

1. $\text{ins}(C) \cap \text{ins}(C') \neq \emptyset$;
2. $\text{ins}(C) \setminus \text{ins}(C') \neq \emptyset$;
3. $\text{ins}(C') \setminus \text{ins}(C) \neq \emptyset$.

A set $\mathcal{C}$ of at least two cycles is crossing if there are two cycles $C, C' \in \mathcal{C}$ such that C and C' cross with each other, otherwise $\mathcal{C}$ is non-crossing.

Our approach has the following major steps:

(s1) For each $Z \in \mathcal{Z}_X$ with $|C_Z| \leq k$, we take C_Z as a "minimum" $(f_Z, f_{\overline{X}})$-separating cycle and the cycle C_Z induced subset Z as a candidate for a noose induced edge subset A_Z that separates Z and $\overline{X}$.

(s2) Let $\mathcal{W}_X = \{Z \in \mathcal{Z}_X \mid |C_Z| > k\}$. We apply the techniques in [53, 54] to decompose H_X into subgraphs that are called *extended components*, each extended component contains face f_Z for exactly one $Z \in \mathcal{W}_X$.

(s3) For each extended component containing one f_Z with $Z \in \mathcal{W}_X$, we find a minimum $(f_Z, f_{\overline{X}})$-separating cycle using the approaches in [18, 69].

(s4) From the $(f_Z, f_{\overline{X}})$-separating cycles computed above, we find non-crossing face-separating cycles to get a good separator for $\mathcal{Z}_X$ and $\overline{X}$.

As shown later, the set of boundary cycles C_Z for $Z \in \mathcal{Z}_X$ is non-crossing. The approach in [69] is a basic tool for Step (s3). The efficiency of the tool can be improved by pre-computing some distances between the vertices in the vertex cut set that separates the extended component from the rest of the graph as in [18]. The subgraphs computed in Step (s2) have properties that allow us to use a scheme in [53, 54] to pre-compute some distances (called *pseudo shortcut set* in [53, 54]) for every extended component to further improve the efficiency of the tool when it is applied to the extended components. To get an $O(nk^2)$ time algorithm, we can not use the techniques in [18] to guarantee unique shortest paths deterministically as in section 3.3. Therefore the set of separating cycles computed in Step (s3) may not be non-crossing. We develop a new technique to clear this hurdle. New ingredients in our approach also include: To find separating cycles, we simply take C_Z as a "minimum" $(f_Z, f_{\overline{X}})$-separating cycle for Z with $|C_Z| \leq k$ and use the complex techniques only for Z with $|C_Z| > k$ instead of every $Z \in \mathcal{Z}_X$ as in [53, 54]. This reduces the time complexity by a $O(k)$ factor for finding the separating cycles. Combining the approaches of [18, 69] for finding the minimum face-separating cycles, the scheme in [53, 54] for pre-computing the pseudo shortcut set and a newly developed technique to extract non-crossing separating cycles from the cycles computed in Steps (s1)-(s3), we find non-crossing separating cycles of length at most k instead of $3k-1$ as in [53, 54, 55].

3.4.1 Review on Previous Techniques

We now briefly review some notions and techniques introduced in [53, 54] for plane graphs without vertex weight. These notions and techniques are also used in [55] but are described for vertex weighted plane graphs. We review these notions and techniques in their form for plane graphs without vertex weight as our algorithm uses these notions and techniques for such graphs.

Let G be a plane graph and f_0 be the outer face of G.

Definition 3.4.2. *The* height *of a vertex u in G is $h_G(u) = \mathrm{nd}_G(V(f_0), u)$.*

For any integer $k \geq 1$, G is *k-outerplanar* if $h_G(u) < k$ for every vertex u in G.

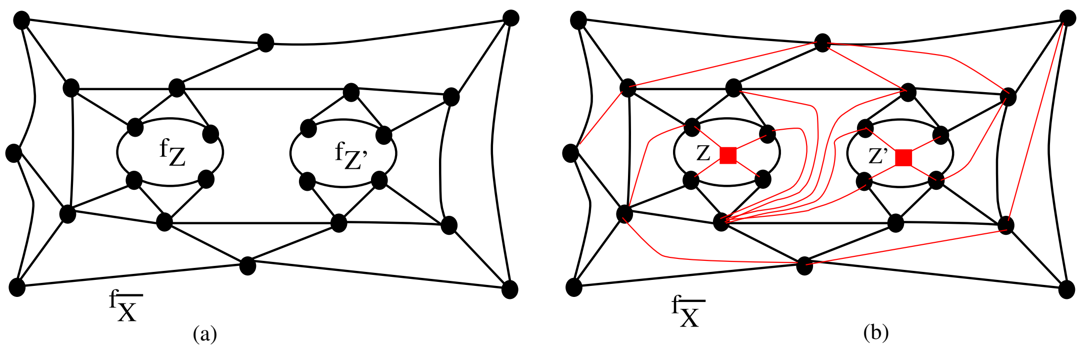
Definition 3.4.3. *The* depth *of a face f of G is $d_G(f) = \min_{u \in V(f)} h_G(u)$. The* depth *of a path L in G is $d_G(L) = \min_{u \in V(L)} h_G(u)$.*

Definition 3.4.4. *A* crest *in G is a maximal connected set $Z \subseteq V(G)$ such that every vertex of Z has the largest height in G.*

Recall that a plane graph is almost triangulated if every face of the graph other than the outer face is incident to exactly three vertices and three edges. Let $\hat{G}$ be an almost triangulated graph. For each u with $h_{\hat{G}}(u) > 0$, an arbitrary vertex v adjacent to u with $h_{\hat{G}}(v) < h_{\hat{G}}(u)$ (such a v always exists) is selected as the *down vertex* of u and the edge $\{u, v\}$ is called the *down edge* of u. When the down edge of every vertex in $\hat{G}$ is selected, each vertex u in $\hat{G}$ has a unique *down path* consisting of the selected down edges only.

Definition 3.4.5. *Let v be any vertex in $\hat{G}$. The* down path *of v is the unique path between v and some vertex of f_0, where f_0 is the outer face of $\hat{G}$, that contains only down edges.*

Definition 3.4.6. *Let Z and Z' be two crests in G. A* ridge *between Z and Z' is a path R between Z and Z' such that R has the maximum depth among all paths between Z and Z'.*



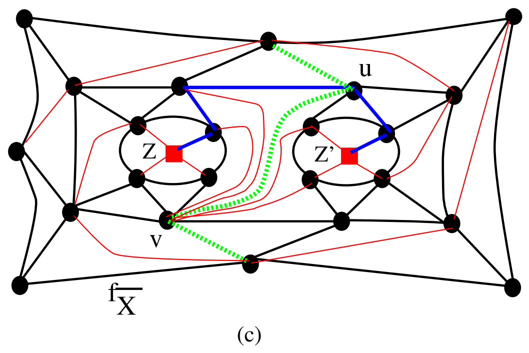


Figure 3.4: (a) A plane graph G_X. (b) An almost triangulated plane graph $\hat{G}_X$ constructed from G_X. The added edges are expressed by red thin segments. $\hat{G}_X$ has two crests Z and Z' expressed by red squares. (c) A ridge R (expressed by blue thick segments) between Z and Z' and a crest separator S (expressed by green thick dashed segments). Vertices u and v are the top vertices, and edge $\{u, v\}$ is the top edge of S.

Definition 3.4.7. *A* crest separator *is a subgraph $S = L_1 \cup L_2$ of $\hat{G}$, where L_1 is the unique down path of a vertex u and L_2 is a path composed of the edge $\{u, u'\}$ and the unique down path of u', where u' is not in L_1 and $h_{\hat{G}}(u') \leq h_{\hat{G}}(u)$.*

Vertices in S of the largest height are called the *top vertices* and edge $\{u, u'\}$ is called the *top edge* of S. Figure 3.4 gives an almost triangulated plane graph with two crests, a ridge R between the two crests and a crest separator. Notice that each crest separator S has $t \in \{1, 2\}$ top vertices. The *height* $h_{\hat{G}}(S)$ of a crest separator S is the height of its top vertices. We say a crest separator $S = L_1 \cup L_2$ is *on a ridge* R if a top vertex u of S is on R and $h_{\hat{G}}(S) = d_{\hat{G}}(R)$. A crest separator $S = L_1 \cup L_2$ is called *disjoint* if path L_1 and the down path of u' do not have a common vertex, otherwise *converged*. For a converged crest separator S, the paths L_1 and L_2 have a common sub-path from a vertex other than u to a vertex w in $V(f_0)$. The vertex $v \neq u$ in the common sub-path with the largest height is called the *low-point* and the sub-path from v to w is called the *converged-path* of S, denoted by $\text{cp}(S)$.

For the outer face f_0 of $\hat{G}$ on the sphere Σ, let $\overline{f_0}$ be the region of $\Sigma \setminus f_0$. Given two crests Z and Z' in $\hat{G}$, We say a crest separator S separates Z from Z' if removing S from

$\overline{f_0}$ cuts $\overline{f_0}$ into two regions, one region contains Z and the other contains Z'. Given a set of $r-1$ crest separators $S_1, .., S_{r-1}$, removing $S_1, .., S_{r-1}$ from $\overline{f_0}$ cuts $\overline{f_0}$ into r regions $R_1, .., R_r$. Let $P_i, 1 \leq i \leq r$, be the subgraph of $\hat{G}$ consisting of the edges of $\hat{G}$ in R_i and the edges of every crest separator with its top edge incident to R_i. Each P_i is called an *extended component.*

Crest separators of minimum height are important in computing extended components with some properties required by our algorithm.

Definition 3.4.8. *A* critical crest separator for crests Z and Z' *in $\hat{G}$ is a crest separator S that satisfies the following conditions:*

- *S is on a ridge between Z and Z' and*
- *S separates Z from Z' in $\hat{G}$.*

The crest separator in Figure 3.4 is a critical crest separator for Z and Z'.

The next observation summarizes some properties of a set of crest separators. These propterties are implicit in the proofs of Lemmas 6-8 of [54] and explicit in Lemmas 3.8-3.12 of [55], and give a base for our algorithm.

Observation 2. *[54, 55] For any subset $\mathcal{W}$ of r crests $Z_1, .., Z_r$ in $\hat{G}$, there is a set $\mathcal{S}$ of $r-1$ crest separators with the following properties:*

(A) The crest separators of $\mathcal{S}$ decompose $\hat{G}$ into extended components $P_1, ..., P_r$ such that each extended component P_i contains exactly one crest $Z_i \in \mathcal{W}$. Moreover, no crest separator in $\mathcal{S}$ contains a vertex of any Z_i in $\mathcal{W}$.

(B) For each pair of extended components P_i and P_j, there is a critical crest separator $S \in \mathcal{S}$ for Z_i and Z_j. Moreover, S has the minimum number of top vertices among all critical crest separators for Z_i and Z_j.

(C) Let $T_{\mathcal{S}}$ be the graph that $V(T_{\mathcal{S}}) = \{P_1, ..., P_r\}$ and there is an edge $\{P_i, P_j\} \in E(T_{\mathcal{S}})$ if there is a crest separator $S = E(P_i) \cap E(P_j)$ in $\mathcal{S}$. Then $T_{\mathcal{S}}$ is a tree.

The tuple $(\hat{G}, \mathcal{S}, \mathcal{W})$ is called a *good mountain structure tree* (GMST). We call $T_{\mathcal{S}}$ the *underlying tree* of the GMST $(\hat{G}, \mathcal{S}, \mathcal{W})$. For each edge $\{P_i, P_j\}$ in $T_{\mathcal{S}}$, the $S \in \mathcal{S}$ with $E(S) = E(P_i) \cap E(P_j)$ is called *the crest separator on edge* $\{P_i, P_j\}$. The following result is implied implicitly in [54] and later stated in [46] and [55]. Our algorithm will use this result for computing a GMST.

Lemma 3.4.1. *[46, 54, 55] Given an arbitrary subset $\mathcal{W}$ of crests in an $O(k)$-outerplanar $\hat{G}$, a GMST $(\hat{G}, \mathcal{S}, \mathcal{W})$ can be computed in $O(|V(\hat{G})|k)$ time.*

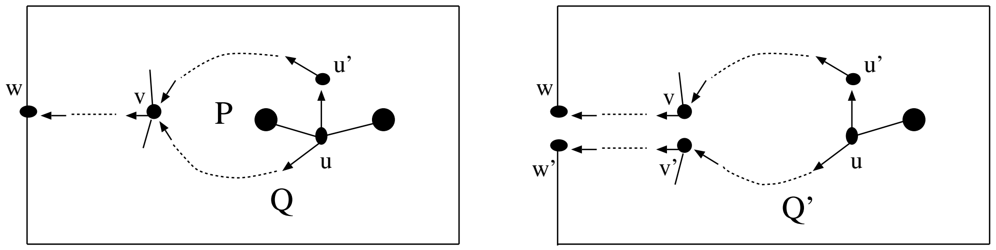


Figure 3.5: Graph Q' obtained from cutting Q along $\mathrm{cp}(S)$.

Given a GMST $(\hat{G}, \mathcal{S}, \mathcal{W})$, we choose an arbitrary vertex P_i in $T_{\mathcal{S}}$ as the root. Each crest separator $S \in \mathcal{S}$ decomposes $\hat{G}$ into two subgraphs, one contains P_i, called the *upper component* by S, and the other does not, called the *lower component* by S. An extended component P is *enclosed* by a converged crest separator S if $P \setminus \mathrm{cp}(S)$ does not have any edge incident to f_0. For vertices u and v in an extended component P, let $\mathrm{dist}_P(u, v)$ denote the length of a shortest path in P between u and v. For vertices u and v in a disjoint crest separator S, let $\mathrm{dist}_S(u, v)$ be the length of the path in S between u and v. For vertices u and v in a converged crest separator S, let $\mathrm{dist}_S(u, v)$ be the length of the shortest path in S between u and v if at least one of u and v is in $\mathrm{cp}(S)$, otherwise let $\mathrm{dist}_S(u, v)$ be the length of the path in S between u and v that does not contain the the low-point of S.

A disjoint crest separator S decomposes $\hat{G}$ into two extended components P and Q. For P (resp. Q), let G_{SP} (resp. G_{SQ}) be the weighted graph on the vertices in S such that for every pair of vertices u and v in S, if $\mathrm{dist}_P(u, v) < \mathrm{dist}_S(u, v)$ (resp. $\mathrm{dist}_Q(u, v) < \mathrm{dist}_S(u, v)$) then there is an edge $\{u, v\}$ with weight $\mathrm{dist}_P(u, v)$ in G_{SP} (resp. with weight $\mathrm{dist}_Q(u, v)$ in G_{SQ}). Edge $\{u, v\}$ is called a *pseudo shortcut*. If P is the upper component by S then G_{SP} is called the *upper pseudo shortcut set* of S, denoted by *up*PSS(S) and G_{SQ} the *lower pseudo shortcut set* of S, denoted by *low*PSS(S), otherwise G_{SP} is called the *low*PSS(S) and G_{SQ} the *up*PSS(S).

A converged crest separator S decomposes $\hat{G}$ into two extended components and exactly one extended component P is enclosed by S. For P, let G_{SP} be defined as in the previous paragraph. Let Q be the other extended component that is not enclosed by S. A plane graph Q' can be created from Q by cutting Q along $\mathrm{cp}(S)$: create a duplicate v' for each vertex v in $\mathrm{cp}(S)$ and create a duplicate e' for each edge e in $\mathrm{cp}(S)$ (see Figure 3.5). Let S' be the subgraph induced by the edges of S and the duplicated edges. For every pair of vertices u, v in S', let $\mathrm{dist}_{S'}(u, v)$ be the length of the path in S' between u and v. For Q', let $G_{SQ'}$ be the weighted graph on the vertices on S' such that for every pair of vertices u and v, if $\mathrm{dist}_{Q'}(u, v) < \mathrm{dist}_{S'}(u, v)$ then there is an edge $\{u, v\}$ with weight $\mathrm{dist}_{Q'}(u, v)$ in $G_{SQ'}$. If P is the upper component by S then G_{SP} is called the *up*PSS(S) and $G_{SQ'}$ the *low*PSS(S), otherwise G_{SP} is called the *low*PSS(S) and $G_{SQ'}$ the *up*PSS(S).

Each crest separator $S \in \mathcal{S}$ decomposes $\hat{G}$ into two extended components P and Q and we assume P is the upper component and Q is the lower component. For each edge

$e = \{u, v\}$ in *up*PSS(S) (resp. *low*PSS(S)), the weight of e is used to decide whether a minimum face-separating cycle should use a shortest path between u and v in P (resp. Q) or not, and if so, any shortest path between u and v in P (resp. Q) can be used (there may be multiple shortest paths between u and v in P (resp. Q)). So we say edge $\{u, v\}$ in *up*PSS(S) (resp. *low*PSS(S)) *represents* any shortest path between u and v in P (resp. Q). The computation of *up*PSS(S) (resp. *low*PSS(S)) also includes computing one shortest path between u and v in P (resp. Q) for every edge $\{u, v\}$ in *up*PSS(S) (resp. *low*PSS(S)).

The next observation summarizes some properties that provide a base for our algorithm.

Observation 3. *[54, 55] For down path, GMST $(\hat{G}, \mathcal{S}, \mathcal{W})$, up*PSS($S$) *and low*PSS($S$)*, the following properties hold:*

(I) For any pair of vertices u and v in $\hat{G}$, $\mathrm{dist}_{\hat{G}}(u, v) \geq |h_{\hat{G}}(u) - h_{\hat{G}}(v)|$.

(II) For any pair of vertices u and v in a same down path of S, $\mathrm{dist}_S(u, v) = |h_{\hat{G}}(u) - h_{\hat{G}}(v)| = \mathrm{dist}_{\hat{G}}(u, v)$.

*(III) If $\hat{G}$ is $O(k)$-outerplanar then for every $S \in \mathcal{S}$, there are $O(k)$ vertices and $O(k)$ edges in S and every edge in up*PSS(S) *(low*PSS(S)*) has weight $O(k)$.*

(IV) If $\hat{G}$ is $O(k)$-outerplanar, $\sum_{P_i \in T_{\mathcal{S}}} |E(P_i)| = |E(\hat{G})| + O(|\mathcal{S}|k)$.

*(V) For every $S \in \mathcal{S}$ and each edge $e = \{u, v\}$ in up*PSS(S)*/low*PSS(S)*, any shortest path represented by e has length smaller than $\mathrm{dist}_S(u, v)$ and any path between u and v with length smaller than $\mathrm{dist}_S(u, v)$ contains no vertex of height greater than $h_{\hat{G}}(S)$ and no more than $t - 1$ vertices of height $h_{\hat{G}}(S)$, where t is the number of top vertices of S.*

*(VI) Let S be the crest separator on edge $\{P_i, P_j\}$ in $T_{\mathcal{S}}$. Assume that Z_i and Z_j are in the upper component P and lower component Q by S, respectively. For every edge $e = \{u, v\}$ in up*PSS(S)*/low*PSS(S)*, any shortest path represented by e has length smaller than $\mathrm{dist}_S(u, v)$ and for any path P_e between u and v in P (resp. Q) with length smaller than $\mathrm{dist}_S(u, v)$, P_e and the segment of S between u and v that contains a top vertex of S form a cycle which separates Z_i (resp. Z_j) from f_0.*

Properties (I)-(IV) in Observation 3 are straightforward from the related definitions. Property (V) is shown in Lemma 12 of [54] and Property (VI) is proved in Lemma 19 of [54] and Lemma 5.2 of [55]. From the properties in Observation 3, the pseudo shortcut sets *up*PSS(S) and *low*PSS(S) can be computed as shown in the next lemma (Lemma 4.12 of [55] and stated in [46]). Our algorithm will use Lemma 3.4.2 for computing the pseudo shortcut sets.

Lemma 3.4.2. *[55] Given a GMST $(\hat{G}, \mathcal{S}, \mathcal{W})$, up*PSS($S$) *and low*PSS($S$) *for all $S \in \mathcal{S}$ can be computed in $O(|V(\hat{G})|k + |\mathcal{W}|k^3)$ time.*

3.4.2 Algorithm for Theorem 3.1.3

For a level 1 node X and the set $\mathcal{Z}_X$ of child nodes in the layer tree $\text{LT}(G|U, \partial(U), k, \alpha)$ in Algorithm 1, recall that G_X is the plane graph converted from the plane hypergraph $(G|\overline{X})|\mathcal{Z}_X$ and H_X is the weighted plane graph computed from G_X as described in Section 3.3. Recall that $\mathcal{W}_X = \{Z \in \mathcal{Z}_X \mid |C_Z| > k\}$. We apply the techniques in [53, 54] to decompose H_X into extended components such that each extended component contains face f_Z of H_X for exactly one $Z \in \mathcal{W}_X$. It may not be straightforward to decompose H_X directly by the techniques of [53, 54] because some of the techniques are described for (unweighted) graphs while H_X is a weighted graph (edges have weight 1/2 or 1). To get a decomposition of H_X as required, we first construct an almost triangulated graph $\hat{G}_X$ from G_X with each $Z \in \mathcal{Z}_X$ represented by a crest of $\hat{G}_X$; then by the techniques of [53, 54] find a GMST of $\hat{G}_X$ that decomposes $\hat{G}_X$ into extended components, each extended component contains exactly one crest; next construct an almost triangulated weighted graph $\hat{H}_X$ from H_X with each $Z \in \mathcal{Z}_X$ represented by a crest of $\hat{H}_X$; and finally compute a set of crest separators in $\hat{H}_X$ based on the GMST of $\hat{G}_X$ to decompose $\hat{H}_X$ into extended components such that each extended component $\hat{H}_X$ contains exactly one crest $Z \in \mathcal{W}_X$ (and thus each extended component of H_X contains exactly one face f_Z).

We first describe the construction of $\hat{G}_X$. Let $f_{\overline{X}}$ be the outer face of G_X. For every $Z \in \mathcal{Z}_X$, we add a vertex, also denoted by Z, and edges $\{u, Z\}$ for every $u \in V(f_Z)$ to face f_Z in G_X. For every natural face f of G_X with $|V(f)| > 3$, we select an arbitrary vertex v of $V(f)$ with $h_{G_X}(v) = d_{G_X}(f)$ ($h_{G_X}(v)$ is the height of vertex v and $d_{G_X}(f)$ is the depth of face f) as the *low-point* of f, denoted by $\text{lp}(f)$, and we add edges $\{u, \text{lp}(f)\}$ to face f for every $u \in V(f)$ and not adjacent to $\text{lp}(f)$. Let $\hat{G}_X$ be the graph obtained from adding the vertices and edges above. Then $\hat{G}_X$ is almost triangulated. Figure 3.4 shows a $\hat{G}_X$ constructed from a G_X.

G_X is a subgraph of $\hat{G}_X$. For every $u \in V(G_X) \cap V(\hat{G}_X)$, $h_{G_X}(u) = h_{\hat{G}_X}(u)$. Every vertex Z added to face f_Z of G_X is a crest of $\hat{G}_X$ and every crest of $\hat{G}_X$ is a vertex Z added to f_Z. Recall that $\mathcal{W}_X = \{Z \in \mathcal{Z}_X \mid |C_Z| > k\}$ is a subset of crests in $\hat{G}_X$. By Lemma 3.4.1, we can find a GMST $(\hat{G}_X, \mathcal{S}, \mathcal{W}_X)$.

Next we describe how to construct $\hat{H}_X$ from H_X. For every $Z \in \mathcal{Z}_X$, we add a vertex, also denoted by Z, and edges $\{u, Z\}$ for every $u \in V(f_Z)$ to face f_Z of H_X. We assign each edge $\{u, Z\}$ weight 1. Let $\hat{H}_X$ be the graph computed above. Then $\hat{H}_X$ is almost triangulated. Notice that $V(G_X) \subseteq V(\hat{G}_X) \subseteq V(\hat{H}_X)$, $V(H_X) \subseteq V(\hat{H}_X)$, $E(G_X) \subseteq E(\hat{G}_X)$, $E(G_X) \subseteq E(H_X) \subseteq E(\hat{H}_X)$ and $|V(\hat{H}_X)| = O(|V(G_X)|)$. Each vertex $u \in V(\hat{H}_X)$ is either a vertex in $\hat{G}_X$ or a vertex added to a natural face of G_X. We define the height $h_{\hat{H}_X}(u)$ of each vertex u of $\hat{H}_X$ as follows:

- $h_{\hat{H}_X}(u) = h_{\hat{G}_X}(u)$ if $u \in V(\hat{H}_X) \cap V(\hat{G}_X)$.
- $h_{\hat{H}_X}(u) = d_{G_X}(f) + \frac{1}{2}$ if $u = u_f$ is the vertex added to a natural face f of G_X.

Then each vertex Z is a crest of $\hat{H}_X$ and each crest of $\hat{H}_X$ is a vertex Z. The height of a crest separator, the depth of a path and a face, and a ridge in $\hat{H}_X$ are defined based on $h_{\hat{H}_X}(u)$ similarly as those in Section 3.4.1. Notice that for every edge e of $\hat{G}_X$, either $e \in E(\hat{G}_X) \cap E(\hat{H}_X)$ or e is an edge added to a natural face f of G_X during the construction of $\hat{G}_X$.

Lemma 3.4.3. *Let Z and Z' be two crests in $\hat{G}_X$ and $\hat{H}_X$. For any ridge R in $\hat{G}_X$ and any ridge R' in $\hat{H}_X$ between Z and Z', either $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R)$ or $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R) + \frac{1}{2}$.*

Proof. For ridge R, if we replace every edge $e = \{u, v\}$ of R that is added to a natural face f with edges $\{u, u_f\}$ and $\{u_f, v\}$ in $\hat{H}_X$, where u_f is the vertex added to face f when constructing H_X from G_X, we get a path Q between Z and Z' in $\hat{H}_X$. Since $h_{\hat{H}_X}(u_f) = \min\{h_{\hat{H}_X}(u), h_{\hat{H}_X}(v)\} + \frac{1}{2}$, $d_{\hat{H}_X}(Q) = d_{\hat{G}_X}(R)$, that is, Q and R have the same depth. Recall that a ridge between Z and Z' is a path between Z and Z' that has the maximum depth among all paths between Z and Z'. $d_{\hat{H}_X}(R') \geq d_{\hat{G}_X}(R)$.

Ridge R' can be partitioned into subpaths such that each subpath either is an edge $e \in E(\hat{G}_X) \cap E(\hat{H}_X)$ or has two edges $\{u, u_f\}, \{u_f, v\}$, where u_f is the vertex added to a natural face of G_X when constructing H_X from G_X. Since $h_{\hat{H}_X}(u_f) = d_{G_X}(f) + \frac{1}{2}$, for each subpath $\{u, u_f\}, \{u_f, v\}$ in R', there is a path P_f between u and v in $\hat{G}_X$ with $d_{\hat{G}_X}(P_f) \geq d_{\hat{G}_X}(f) = h_{\hat{H}_X}(u_f) - \frac{1}{2}$. If we replace every subpath $\{u, u_f\}, \{u_f, v\}$ of R' by path P_f, we get a path P between Z and Z' in $\hat{G}_X$ with $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(P)$ or $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(P) + \frac{1}{2}$. From this and $d_{\hat{H}_X}(R') \geq d_{\hat{G}_X}(R)$, either $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R)$ or $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R) + \frac{1}{2}$. □

Similar to the down vertex and down edge in $\hat{G}_X$, we define the down vertex and down edge for each vertex of $\hat{H}_X$. Recall that any vertex v adjacent to vertex u with $h_{\hat{H}_X}(v) < h_{\hat{H}_X}(u)$ can be selected as the down vertex of u. We choose the down vertex for each u of $\hat{H}_X$ as follows:

- if $u \in V(\hat{G}_X)$ and the down edge $\{u, v\}$ of u in $\hat{G}_X$ is in $E(\hat{G}_X) \cap E(\hat{H}_X)$, then let v be the down vertex of u in $\hat{H}_X$;
- if $u \in V(\hat{G}_X)$ and the down edge $\{u, v\}$ of u in $\hat{G}_X$ is an edge added to a natural face f of G_X, then let the vertex u_f added to f in $\hat{H}_X$ be the down vertex of u;
- otherwise, u is not in $\hat{G}_X$ and is the vertex u_f added to a natural face f of G_X in $\hat{H}_X$; then let the low-point $\text{lp}(f)$ be the down vertex of u.

The edge between vertex u and its down vertex is the down edge of u.

Given a GMST $(\hat{G}_X, \mathcal{S}, \mathcal{W}_X)$, for every crest separator $S \in \mathcal{S}$, we convert each $S \in \mathcal{S}$ into a subgraph D of $\hat{H}_X$ as follows: for every edge $e = \{u, v\}$ in S, if e is also an edge of $\hat{H}_X$ then e is included in D, otherwise edges $\{u, u_f\}, \{u_f, v\}$ of $\hat{H}_X$ are included in D, where u_f is the vertex added to face f of G_X when H_X is created from G_X. D is called the *$\hat{H}_X$-converted graph* of S.

Lemma 3.4.4. *Let $S \in \mathcal{S}$ be a critical crest separator for crest Z and Z' in $\hat{G}_X$ such that S has the minimum number of top vertices among the critical crest separators for Z and Z'. Let D be the $\hat{H}_X$-converted graph of S. Then either*

- *D is a critical crest separator for Z and Z' in $\hat{H}_X$, and D has the minimum number of top vertices among the critical crest separators for Z and Z'; or*
- *D is a crest separator that separates Z from Z', $h_{\hat{H}_X}(D) = d_{\hat{H}_X}(R') + 1/2$, and every critical separator for Z and Z' in $\hat{H}_X$ has two top vertices.*

Proof. Recall that a crest separator S in $\mathcal{S}$ consists of two paths L_1 and L_2, where L_1 is the down path of some vertex u and L_2 is composed of the top edge $\{u, u'\}$ and the down path of u', where u' is not in L_1 and $h_{\hat{H}_X}(u') \leq h_{\hat{H}_X}(u)$. Let $\{u, u'\}$ be the top edge of S. The $\hat{H}_X$-converted graph of S (i.e. D) consists of two paths L'_1 and L'_2. There are three cases:

1. $\{u, u'\} \in E(\hat{H}_X) \cap E(\hat{G}_X)$. L'_1 is the down path from vertex u in $\hat{H}_X$ and L'_2 is composed of the top edge $\{u, u'\}$ and the down path from u' in $\hat{H}_X$.
2. $\{u, u'\} \in E(\hat{G}_X) \setminus E(\hat{H}_X)$ and $h_{\hat{G}_X}(u) = h_{\hat{G}_X}(u') + 1$. L'_1 is the down path from vertex u in $\hat{H}_X$ and L'_2 is composed of the top edge $\{u, u_f\}$ and the down path from u_f in $\hat{H}_X$.
3. $\{u, u'\} \in E(\hat{G}_X) \setminus E(\hat{H}_X)$ and $h_{\hat{G}_X}(u) = h_{\hat{G}_X}(u')$. L'_1 is the down path from vertex u_f. If $u = \mathrm{lp}(f)$ then L'_2 is composed of the top edge $\{u_f, u'\}$ and the down path from u', otherwise, L_2 is composed of the top edge $\{u_f, u\}$ and the down path from u.

In all cases, D is a crest separator that separates Z from Z' in $\hat{H}_X$. Let R be a ridge between in $\hat{G}_X$ and R' be a ridge in $\hat{H}_X$ between Z and Z'. By Lemma 3.4.3, $d_{\hat{H}_X}(R') \geq d_{\hat{G}_X}(R)$. Since D separates Z from Z' in $\hat{H}_X$, D intersects R', implying $h_{\hat{H}_X}(D) \geq d_{\hat{H}_X}(R')$. Because S is a critical crest separator for Z and Z' in $\hat{G}_X$, $h_{\hat{G}_X}(S) = d_{\hat{G}_X}(R)$. Therefore, $h_{\hat{H}_X}(D) \geq d_{\hat{H}_X}(R') \geq d_{\hat{G}_X}(R) = h_{\hat{G}_X}(S)$.

In Cases 1 and 2, $h_{\hat{H}_X}(D) = h_{\hat{G}_X}(S)$, implying $h_{\hat{H}_X}(D) = d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R) = h_{\hat{G}_X}(S)$. So D is a critical separator for Z and Z' in $\hat{H}_X$. In these two cases, D has the same number of top vertices as that of S. In Case 2, S has one top vertex and thus D has the minimum number of top vertices. In Case 1, if S has one top vertex then D has the minimum number of top vertices. Assume that S have two top vertices and there is a critical crest separator D' for Z and Z' in $\hat{H}_X$ with one top vertex. Because $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R)$, there is a critical separator S' for Z and Z' in $\hat{G}_X$ with one top vertex, contradicting with the fact that S is a critical crest separator for Z and Z' with the minimum number of top vertices. Therefore, D has the minimum number of top vertices among all the critical crest separators for Z and Z'.

In Case 3, S has two top vertices and $h_{\hat{H}_X}(D) = h_{\hat{G}_X}(S) + \frac{1}{2} = d_{\hat{G}_X}(R) + \frac{1}{2}$. If $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R) + \frac{1}{2}$, then $h_{\hat{H}_X}(D) = d_{\hat{H}_X}(R')$ and D is a critical crest separator for Z and Z' in

$\hat{H}_X$ with exactly one top vertex. Otherwise, by Lemma 3.4.3, $d_{\hat{H}_X}(R') = d_{\hat{G}_X}(R)$ and $h_{\hat{H}_X}(D) = d_{\hat{H}_X}(R') + 1/2$. Assume that there is a critical crest separator for Z and Z' in $\hat{H}_X$ with one top vertex. Then there is a critical crest separator in for Z and Z' in $\hat{G}_X$ with one top vertex, contradicting with the fact that S has two top vertices and is a a critical crest separator for Z and Z' with the minimum number of top vertices. □

The next lemma is straightforward from Observation 2 and Lemma 3.4.4.

Lemma 3.4.5. *Given a GMST $(\hat{G}_X, \mathcal{S}, \mathcal{W}_X)$, let $\mathcal{D} = \{D | D$ is the $\hat{H}_X$-converted graph of S for every $S \in \mathcal{S}\}$ and assume that $\mathcal{W}_X$ has r crests $Z_1, .., Z_r$. Then the following properties holds for $\mathcal{D}$:*

(A) The crest separators of $\mathcal{D}$ decompose $\hat{H}_X$ into r extended components $P_1, ..., P_r$ such that each extended component P_i has exactly one crest Z_i. Moreover, no crest separator D in $\mathcal{D}$ contains a crest in $\mathcal{W}_X$, that is, each extended component contains the edges in $E(f_Z)$ of H_X for exactly one $Z \in \mathcal{W}_X$.

(B) For each pair of extended components P_i and P_j, either

(B1) there is a critical crest separator $D \in \mathcal{D}$ for Z_i and Z_j such that D has the minimum number of top vertices among the critical crest separators for Z_i and Z_j in $\hat{H}_X$; or

(B2) D is a crest separator that separates Z from Z', and $h_{\hat{H}_X}(D) = d_{\hat{H}_X}(R') + 1/2$; and every critical crest separator for Z and Z' in $\hat{H}_X$ has two top vertices.

(C) Let $T_{\mathcal{D}}$ be the graph that $V(T_{\mathcal{D}}) = \{P_1, ..., P_r\}$ and there is an edge $\{P_i, P_j\} \in E(T_{\mathcal{D}})$ if there is a crest separator $D = E(P_i) \cap E(P_j)$ in $\mathcal{D}$. Then $T_{\mathcal{D}}$ is a tree.

Definition 3.4.9. *The tuple $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$ is called a* pseudo good mountain structure tree (pseudo GMST) *for $\mathcal{W}_X$.*

For each $D \in \mathcal{D}$ and vertices u, v of D, $\text{dist}_D(u, v)$, *up*PSS(D) and *low*PSS(D) are defined similarly as $\text{dist}_S(u, v)$, *up*PSS(S) and *low*PSS(S) for crest separator S and u, v of S in Section 3.4.1. As shown in the next lemma, Properties (I) to (VI) in Observation 3 hold for a pseudo GMST, *up*PSS(D) and *low*PSS(D).

Lemma 3.4.6. *Properties (I) to (VI) in Observation 3 hold for a pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$, *up*PSS$(D)$ and *low*PSS(D), $D \in \mathcal{D}$.*

Proof. Property (I) and Property (II) hold trivially from the definition of $h_{\hat{H}_X}(u)$ for vertex u and the definition of down path.

If $\hat{H}_X$ is $O(k)$-outerplanar, then the height of D is $O(k)$ and there are $O(k)$ vertices and $O(k)$ edges in D because the weight of each edge in $\hat{H}_X$ is 1 or 1/2. From Property (II), every edge in $up\mathrm{PSS}(D)$ ($low\mathrm{PSS}(D)$) has weight $O(k)$ because $\mathrm{dist}_D(u, v) = O(k)$ for any u and v in D. Thus, Property (III) holds.

Every $D \in \mathcal{D}$ is the $\hat{H}_X$-converted graph of an $S \in \mathcal{S}$ and $|E(D)| = O(|E(S)|)$. From this, and $|\mathcal{D}| = |\mathcal{S}|$, $\sum_{P_i \in T_{\mathcal{D}}} |E(P_i)| = |E(\hat{H}_X)| + O(|\mathcal{D}|k)$. Thus, Property (IV) holds.

From Property (II), for each edge $e = \{u, v\}$ in $up\mathrm{PSS}(S)/low\mathrm{PSS}(S)$, u and v must be in different down paths of D and $\mathrm{dist}_D(u, v) = 2h_{\hat{H}_X}(D) - h_{\hat{H}_X}(u) - h_{\hat{H}_X}(v) + (t - 1)$, where $t \in \{1, 2\}$ is the number of top vertices of D. Any vertex w of height greater than $h_{\hat{H}_X}(D)$ in $\hat{H}_X$ has height at least $h_{\hat{H}_X}(D) + 1/2$. From Property (I), any path between u and v that contains w has length at least $2h_{\hat{H}_X}(D) + 1 - h_{\hat{H}_X}(u) - h_{\hat{H}_X}(v) \geq \mathrm{dist}_D(u, v)$. Similarly, any path between u and v that contains t vertices of height $h_{\hat{H}_X}(D)$ has length at least $\mathrm{dist}_D(u, v)$. Therefore, any path between u and v with length smaller than $\mathrm{dist}_D(u, v)$ contains no vertex of height greater than $h_{\hat{H}_X}(D)$ and no more than $t - 1$ vertices of height $h_{\hat{H}_X}(D)$. Thus, Property (V) holds.

We prove Property (VI) by contradiction. Let $D \in \mathcal{D}$ be the crest separator on edge $\{P_i, P_j\}$ in $T_{\mathcal{D}}$. We assume that Z_i and Z_j are in the upper component and lower component by D, respectively. For each edge $e = \{u, v\}$ in $up\mathrm{PSS}(D)$ (resp. $low\mathrm{PSS}(D)$), let P_e be any path between u and v in the upper component P (resp. lower component Q) with length smaller than $\mathrm{dist}_D(u, v)$ and let $D(u, v)$ be the subpath of D between u and v and containing a top vertex of D. Then P_e and $D(u, v)$ form a cycle in $\hat{H}_X$. Assume for contradiction that the cycle formed by P_e and $D(u, v)$ does not separate Z_i (resp. Z_j) from $f_{\overline{X}}$. Let R' be a ridge between Z_i and Z_j in $\hat{H}_X$. Let D' be the subgraph of $\hat{H}_X$ obtained by replacing $D(u, v)$ with P_e in D. Then D' separates Z_i from Z_j and thus, intersects with R', and D' contains at least one vertex w with $h_{\hat{H}_X}(w) = d_{\hat{H}_X}(R')$. Notice that every vertex of $D' \setminus P_e$ has height smaller than $d_{\hat{H}_X}(R')$. So P_e contains w. Consider Property (B1) in Lemma 3.4.5 where $h_{\hat{H}_X}(D) = d_{\hat{H}_X}(R')$ and D has the minimum number of top vertices among all critical crest separators for Z_i and Z_j. If D has one ($t = 1$) top vertex, from Property (V), P_e contains no ($t - 1 = 0$) vertex of height $d_{\hat{H}_X}(R')$, a contradiction. If D has two top vertices then every critical crest separator for Z_i and Z_j has two top vertices and w is the only vertex of height $d_{\hat{H}_X}(R')$ in P_e. However, this means that D' is a critical crest separator for Z_i and Z_j with one top vertex w, a contradiction. Consider Property (B2) in Lemma 3.4.5, where $h_{\hat{H}_X}(D) = d_{\hat{H}_X}(R') + 1/2$ and every critical crest separator for Z_i and Z_j in $\hat{H}_X$ has two top vertices. As proved, if P_e has one vertex of height $d_{\hat{H}_X}(R')$, we have a contradiction. If P_e has two vertices of height greater than or equal to $d_{\hat{H}_X}(R')$, the length of P_e is at least $\mathrm{dist}_D(u, v)$, a contradiction. This gives Property (VI). □

Given a GMST $(\hat{G}_X, \mathcal{S}, \mathcal{W}_X)$, a pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$ can be computed in $O(|\mathcal{S}|k)$ time. Lemma 3.4.2 holds for a pseudo GMST because the lemma only relies on Properties (I)-(VI) in Observation 3. The computation of $up\mathrm{PSS}(D)$ and $low\mathrm{PSS}(D)$ in Lemma 3.4.2 (implicitly) uses a linear time algorithm by Thorup [77] for the single shortest path problem in graphs with integer edge weight as a subroutine. If the unique shortest paths are required, the techniques in Section 3.3 to guarantee the unique shortest paths will introduce additional time overhead for computing $up\mathrm{PSS}(D)$ and $low\mathrm{PSS}(D)$. To avoid the additional time overhead, we do not assume the uniqueness of shortest paths in the algorithm for Theorem 3.1.3 when we compute minimum separating cycles. In Section 3.3, we do not have crossing minimum separating cycles because of the uniqueness of shortest paths. However, two minimum separating cycles computed without the assumption of unique shortest paths may cross with each other. We eliminate each crossing cycle set by exploiting some new properties of pseudo GMST, $up\mathrm{PSS}(D)$ and $low\mathrm{PSS}(D)$, and then compute a good separator $\mathcal{A}_X$.

Recall that $\hat{H}_X$ is constructed from H_X by adding vertex (crest) Z and edges $\{u, Z\}$, $u \in V(f_Z)$, to face f_Z in H_X. Each extended component P of a pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$ contains exactly one crest $Z \in \mathcal{W}_X$. We show in the proof of Lemma 3.4.7 later that a minimum $(f_Z, f_{\overline{X}})$-separating cycle can be computed based on $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$, $up\mathrm{PSS}(D)/low\mathrm{PSS}(D)$, $D \in \mathcal{D}$, and the approaches of [18, 69].

Given a set $\mathcal{W}_X$ of crests in $\hat{H}_X$, our algorithm for Theorem 3.1.3 computes a pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$, calculates $up\mathrm{PSS}(D)$ and $low\mathrm{PSS}(D)$ for every crest separator $D \in \mathcal{D}$, and finds a minimum $(f_Z, f_{\overline{X}})$-separating cycle for every $Z \in \mathcal{W}_X$ using the pseudo GMST, $up\mathrm{PSS}(D)$ and $low\mathrm{PSS}(D)$. More specifically, we replace Lines $7 - 16$ in Algorithm 1 by Procedure Separator-Based below.

Recall that for each $Z \in \mathcal{Z}_X$, the boundary cycle C_Z for Z is the cycle formed by all edges incident to face f_Z. Procedure Separator-Based first tries to find $(f_Z, f_{\overline{X}})$-separating cycles from the boundary cycles C_Z of $Z \in \mathcal{Z}_X$. The property of boundary cycles shown in the next proposition implies that the set of boundary cycles is non-crossing and there are $O(m)$ edges in these cycles.

Proposition 1. *For any distinct $Z, Z' \in \mathcal{Z}_X$, C_Z and $C_{Z'}$ share at most one common vertex.*

Proof. Assume that C_Z and $C_{Z'}$ share more than one vertex. Then $Z \cup Z'$ is biconnected, contradicting with the fact that each $Z \in \mathcal{Z}_X$ is a biconnected component (i.e., a maximal biconnected subgraph). □

We now analyze the time complexity of Procedure Separator-Based.

Lemma 3.4.7. *The time complexity of Lines $1 - 18$ in Procedure Separator-Based is $O(|E(H_X)|k^2)$.*

Proof. Let m be the number of edges in H_X. Then $|E(\hat{G}_X)| = |E(\hat{H}_X)| = O(m)$. So Lines $1 - 2$ takes $O(m)$ time. By Proposition 1, $\sum_{Z \in \mathcal{Z}_X} |C_Z| = O(m)$. Therefore, for all $Z \in \mathcal{Z}_X \setminus \mathcal{W}_X$, the $(f_Z, f_{\overline{X}})$-separating cycles can be found in $O(m)$ time (Lines $3 - 5$). For each crest $Z \in \mathcal{W}_X$, C_Z has more than k edges in H_X. Therefore, $|\mathcal{W}_X| = O(m/k)$. By

Procedure Separator-Based

```
   Input: A plane hypergraph $(G|\overline{X})|\mathcal{Z}_X$ and an integer $k \geq 2$.
   Output: Either a good separator $\mathcal{A}_X$ for $\mathcal{Z}_X$ and $\overline{X}$, or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder
           minor of $G$.
 1 compute $G_X, H_X, \hat{G}_X, \hat{H}_X$ from $(G|\overline{X})|\mathcal{Z}_X$;
 2 mark every $Z \in \mathcal{Z}_X$ as un-separated;
 3 for every $Z \in \mathcal{Z}_X$ with $|C_Z| \leq k$ do
 4     take $C_Z$ as a $(f_Z, f_{\overline{X}})$-separating cycle and mark $Z$ as separated;
 5 end
 6 let $\mathcal{W}_X = \{Z \in \mathcal{Z}_X \mid |C_Z| > k\}$;
 7 compute a GMST $(\hat{G}_X, \mathcal{S}, \mathcal{Z}_X)$ by Lemma 3.4.1;
 8 compute a pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$ from $(\hat{G}_X, \mathcal{S}, \mathcal{W}_X)$;
 9 compute upPSS($D$) and lowPSS($D$) for every crest separator $D \in \mathcal{D}$ by Lemma 3.4.2;
10 while not every $Z \in \mathcal{Z}_X$ is marked as separated do
11     choose an arbitrary un-separated $Z$, compute a minimum $(f_Z, f_{\overline{X}})$-separating
         cycle $C$ using the pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$, upPSS($D$) and lowPSS($D$);
12     if the length of $C$ is greater than $k$ then
13         return a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor by Lemma 3.3.1;
14     else
15         take $C$ as the minimum $(f_Z, f_{\overline{X}})$-separating cycle for every $Z \in \text{ins}(C)$;
16         mark every $Z$ in ins($C$) as separated;
17     end
18 end
19 Compute $\mathcal{A}_X$ from the (minimum) face-separating cycles obtained in Lines 4 and 15.
```

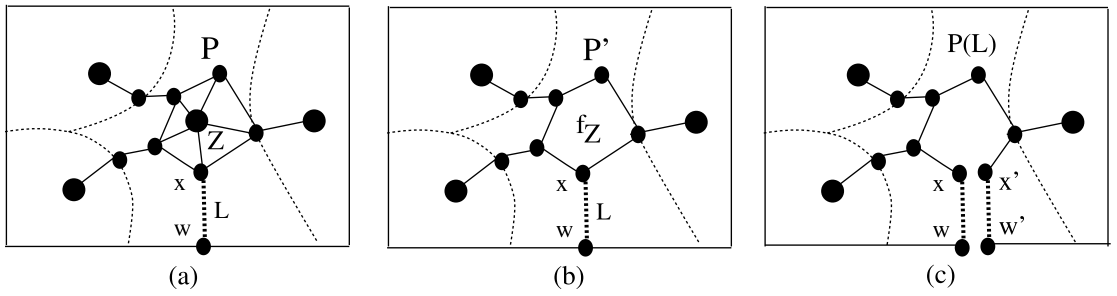


Figure 3.6: (a) Extended component P of $\hat{H}_X$, (b) extended component P' of H_X and (c) graph $P(L)$.

Lemmas 3.4.1 and 3.4.2, a pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$, upPSS(D) and lowPSS(D) for all $D \in \mathcal{D}$ can be computed in $O(mk + |\mathcal{W}_X|k^3) = O(mk^2)$ time (Lines $6-9$).

For an un-separated crest $Z \in \mathcal{W}_X$, let P be the extended component in the pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$ containing Z (see Figure 3.6 (a)). By Property (A) in Lemma 3.4.5, removing Z and the edges incident to Z from P gives an extended component P' of H_X containing face f_Z of H_X (see Figure 3.6 (b)). Let x be an arbitrary vertex in P incident to Z and L be the down path from x to a vertex $w \in V(f_{\overline{X}})$. Then L is a shortest path between x and w in H_X and can be found in $O(k)$ time. Let $P(L)$ be the weighted plane graph obtained from P' by cutting along L: for each vertex u in L create a duplicate u', for each edge e in L create a duplicate e' and create a new face bounded by edges of $E(f_{\overline{X}})$, $E(f_Z)$, L and their duplicates (see Figure 3.6 (c)). Let $H_X(L)$ be the weighted graph obtained from H_X by replacing P' with $P(L)$. For each vertex u in path L, let C_u be a shortest path between u and its duplicate u' in $H_X(L)$. Let y be a vertex in L such that C_y has the minimum length among the paths C_u for all vertices u in L. Then C_y gives a minimum $(f_Z, f_{\overline{X}})$-separating cycle in H_X [69].

For each extended component P in pseudo GMST $(\hat{H}_X, \mathcal{D}, \mathcal{W}_X)$, let $\tilde{D}$ be the crest separator on the edge between P and its parent node and let $\mathcal{D}_P$ be the set of crest separators on an edge between P and a child node of P in $T_{\mathcal{D}}$. From Property (A) in Lemma 3.4.5, Property (V) in Lemma 3.4.6, any shortest path represented by an edge in upPSS(D) or lowPSS(D), $D \in \mathcal{D}$, does not contain any vertex of $V(\hat{H}_X) \setminus V(H_X)$. Therefore, C_u for every u in L can be partitioned into subpaths such that each subpath is either entirely in $P(L)$ or is represented by an edge in upPSS$(\tilde{D})$ or lowPSS(D), $D \in \mathcal{D}_P$. Let P^* be the weighted graph consisting of the edges of $P(L)$, upPSS$(\tilde{D})$ and lowPSS(D), $D \in \mathcal{D}_P$. Notice that for every edge e in upPSS$(\tilde{D})$ and lowPSS(D), $D \in \mathcal{D}_P$, a shortest path represented by e is also computed. Then it is known (Section 2.2 in [18]) that a minimum face separating cycle C_y can be computed in $O(t(P^*) \log |V(L)| + |V(C_y)|) = O(t(P^*) \log k + |V(C_y)|)$ time, where $t(P^*)$ is the time to find a shortest path C_u in P^* for any u in L. For each edge of P^*, we can multiply the edge weight by 2 to make each edge weight a positive integer. By the algorithm in [77], a shortest path C_u can be computed in linear time, that is, $t(P^*) = O(|E(P^*)|)$. Therefore, from the fact that each crest separator has $O(k)$ vertices (Property (III) in Lemma 3.4.6), it takes

$$\begin{aligned} & O((|E(P(L))| + |E(up\mathrm{PSS}(\tilde{D}))| + |\cup_{D\in\mathcal{D}_P} E(low\mathrm{PSS}(D))|)\log k + |V(C_y)|) \\ = \; & O((|E(P)| + \Delta(P)k^2)\log k + |V(C_y)|) \end{aligned}$$

time to compute a minimum $(f_Z, f_{\overline{X}})$-separating cycle, where $\Delta(P)$ is the number of edges incident to P in $T_{\mathcal{D}}$. If C_y has length at most k then $|V(C_y)| = O(k)$, otherwise $|V(C_y)| = O(m)$. Assume that C_y for every $Z \in \mathcal{W}_X$ has length at most k. From Property (C) in Lemma 3.4.5, Property (IV) in Lemma 3.4.6 and $|V(T_{\mathcal{D}})| = |\mathcal{W}_X| = O(m/k)$, the time for computing the minimum $(f_Z, f_{\overline{X}})$-separating cycles for all crests $Z \in \mathcal{W}_X$ is

$$\begin{aligned} \sum_{P\in V(T_{\mathcal{D}})} O((|E(P)| + \Delta(P)k^2)\log k + k) &= O((m + |\mathcal{W}_X|k + |\mathcal{W}_X|k^2)\log k + |\mathcal{W}_X|k) \\ &= O(mk\log k). \end{aligned}$$

Therefore, Line 11 takes $O(mk\log k)$ time in the whole while loop.

If there is a minimum face-separating cycle C_y with length greater than k, it takes

$$O((|E(P)| + \Delta(P)k^2)\log k + m) = O(mk\log k)$$

time to compute this C_y and $O(m)$ time to compute a cylinder minor (Lines $12-13$). Lines $15-16$ is trivial and takes $O(m)$ time in total.

Summing up, the total time for Lines $1 \to 18$ is $O(|E(H_X)|k^2)$. □

We now show how to compute a good separator $\mathcal{A}_X$ in Line 19 of Procedure Separator-Based. By Proposition 1, any two boundary cycles do not cross with each other. Each boundary cycle is the boundary of a face in G_X and faces are disjoint. The height of any vertex in a boundary cycle is at least the height of any $D \in \mathcal{D}$. Therefore, from Property (B) in Lemma 3.4.5 and Property (V) in Lemma 3.4.6, a boundary cycle does not cross with a minimum separating cycle for any crest computed in Line 12 of Procedure Separator-Based. However, a minimum separating cycle for one crest may cross with a minimum separating cycle for another crest. The next lemma gives a base for eliminating crossing separating cycles. We call C computed in Line 12 of Procedure Separator-Based the *cycle computed for Z*.

Lemma 3.4.8. *Let C_1 be the minimum $(f_Z, f_{\overline{X}})$-separating cycle computed for crest Z in Line 12 of Procedure Separator-Based. Let C_2 be the minimum $(f_{Z'}, f_{\overline{X}})$-separating cycle computed for crest Z' after C_1 in Line 12 of Procedure Separator-Based. If C_1 and C_2 cross with each other, then there is a cycle C such that* $\mathrm{ins}(C) = \mathrm{ins}(C_1) \cup \mathrm{ins}(C_2)$ *and the length of C is the same as that of C_2.*

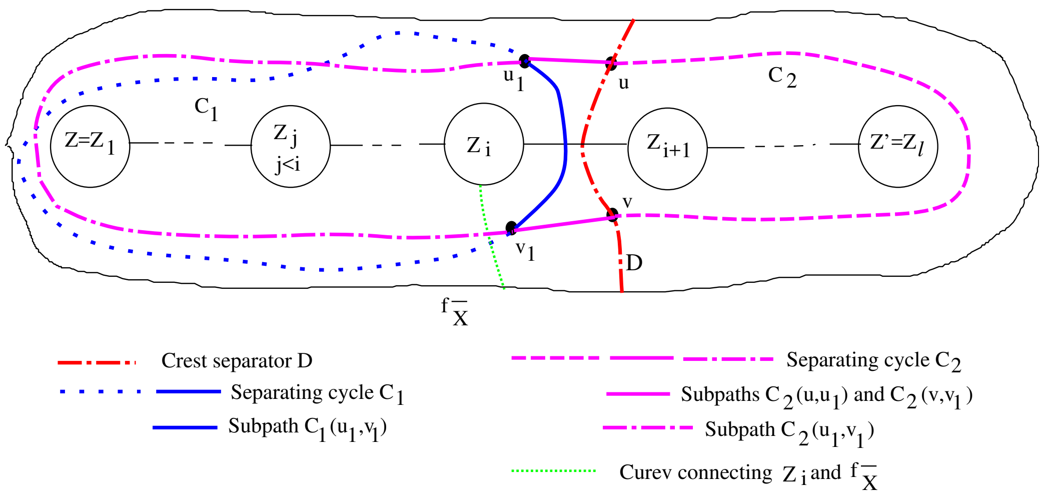


Figure 3.7: A pair of crossing minimum separating cycles C_1 and C_2.

Proof. Assume that P_1 and P_l in the underlying tree $T_{\mathcal{D}}$ contain Z and Z', respectively. Let $\{P_1, P_2\}, \{P_2, P_3\}, ..., \{P_{l-1}, P_l\}$ be the path between P_1 and P_l in $T_{\mathcal{D}}$ and let the crest in P_i be Z_i for $1 \leq i \leq l$ ($Z = Z_1$ and $Z' = Z_l$). It is shown in (Lemma 30) in [54] and (Lemma 5.9) in [55] that if $Z_i \in \text{ins}(C_1)$ then every $Z_j \in \text{ins}(C_1)$ for $j < i$, and if $Z_i \in \text{ins}(C_2)$ then every $Z_j \in \text{ins}(C_2)$ for $j > i$. Because C_2 is computed after C_1, $Z' \notin \text{ins}(C_1)$. So there is a Z_i, $1 \leq i < l$, such that $Z_i \in \text{ins}(C_1)$ but $Z_j \notin \text{ins}(C_1)$ for $j > i$.

Let D be the crest separator for Z_i and Z_{i+1} in the pseudo GMST (see Figure 3.7). We assume without loss of generality that Z (resp. Z') is in the upper component P (resp. lower component Q) by D. From Property (VI) in Lemma 3.4.6 and the fact that $Z_{i+1} \notin \text{ins}(C_1)$, C_1 contains no (shortest path represented by) edge in *low*PSS(D). Notice that that D separates Z from Z' in $\hat{H}_X$. From the fact that C_1 and C_2 cross with each other and the fact that C_1 contains no edge in *low*PSS(D), C_2 has a subpath $C_2(u,v)$ represented by an edge $e = \{u, v\}$ in *up*PSS(D), where u and v are in different down paths of D. By Property (VI) in Lemma 3.4.6, $Z_i \in \text{ins}(C_2)$.

Since C_1 and C_2 cross with each other, they intersect at at least two vertices. Let u_1 and v_1 be the first vertex and last vertex at which cycle C_2 intersects cycle C_1, respectively, when we proceed on $C_2(u,v)$ from u to v. Subpath $C_2(u,v)$ consists of three subpaths: $C_2(u,u_1)$ between u and u_1, $C_2(u_1,v_1)$ between u_1 and v_1 and $C_2(v,v_1)$ between v and v_1. Because cycle C_1 separates Z_i from Z_{i+1}, it contains a subpath $C_1(u_1,v_1)$ between u_1 and v_1 that intersects the ridge between Z_i and Z_{i+1}. Figure 3.7 shows the subpaths above. Let P_e be the path between u and v consisting of subpaths $C_2(u,u_1)$, $C_1(u_1,v_1)$ and $C_2(v,v_1)$. Let $D(u,v)$ be the subpath of D between u and v that contains a top vertex of D. Then P_e and $D(u,v)$ form a closed walk W. On the other hand, there is a curve between Z_i and $f_{\overline{X}}$ such that the curve intersects C_1 and C_2 at $((C_1 \setminus C_1(u_1,v_1)) \cup C_2(u_1,v_1)) \setminus \{u_1,v_1\}$ only and does not intersect D (see Figure 3.7). Therefore, the closed walk W does not separates Z_i from

$f_{\overline{X}}$. From this and Property (VI) in Lemma 3.4.6, the length of P_e is at least $\text{dist}_D(u, v)$. Let l_1 be the length of P_e, l_2 be the length of $C_2(u, v)$ and l_s be the sum of the length of $C_2(u, u_1)$ and that of $C_2(v, v_1)$. Then the length of $C_1(u_1, v_1)$ is $l_1 - l_s$ and the length of $C_2(u_1, v_1)$ is $l_2 - l_s$. Since $l_2 < \text{dist}_D(u, v) \leq l_1$, the length of $C_2(u_1, v_1)$ is smaller than that of $C_1(u_1, v_1)$. Therefore, $Z \in \text{ins}(C_2)$ because otherwise, we can replace $C_1(u_1, v_1)$ with $C_2(u_1, v_1)$ to get a separating cycle for Z with a smaller length, a contradiction to that C_1 is a minimum separating cycle for Z.

For each connected region R in $\text{ins}(C_1)\setminus\text{ins}(C_2)$, the boundary of R consists of a subpath $C_1(R)$ of C_1 and a subpath $C_2(R)$ of C_2. The lengths of $C_1(R)$ and $C_2(R)$ are the same, otherwise, we can get a separating cycle for Z or Z' with length smaller than that of C_1 or C_2, respectively, a contradiction to that C_1 is a minimum separating cycle for Z and C_2 is a minimum separating cycle for Z'.

We construct the cycle C for the lemma as follows: Initially $C = C_2$. For every connected region R in $\text{ins}(C_1) \setminus \text{ins}(C_2)$, we replace $C_2(R)$ by $C_1(R)$. Then $\text{ins}(C) = \text{ins}(C_1) \cup \text{ins}(C_2)$ and has the same length as that of C_2. □

By applying Lemma 3.4.8 repeatedly, we get the next lemma to eliminate a set of crossing minimum separating cycles.

Lemma 3.4.9. *Let $C_1, C_2, ...C_t$ be a set of crossing minimum separating cycles computed in Line 15 of Procedure Separator-Based such that every C_i, $1 < i \leq t$, is computed after C_{i-1}. Then there is a cycle C such that $\text{ins}(C) = \text{ins}(C_1) \cup \ldots \text{ins}(C_t)$ and the length of C is the same as that of C_t.*

Given a set of separating cycles computed in Line 4 and Line 15 in Procedure Separator-Based, our next job is to eliminate the crossing minimum separating cycles and find a good separator $\mathcal{A}_X$ for $\mathcal{Z}_X$ and $\overline{X}$. The next lemma shows how to do this (Line 19 of Procedure Separator-Based).

Lemma 3.4.10. *Given the set of separating cycles of length at most k computed in Procedure Separator-Based, a good separator $\mathcal{A}_X$ for $\mathcal{Z}_X$ and $\overline{X}$ can be computed in $O(|E(\hat{H}_X)|)$ time.*

Proof. Let $m = |E(H_X)|$ and $\mathcal{C}$ be the set of $(f_Z, f_{\overline{X}})$-separating cycles for all $Z \in \mathcal{Z}_X$. Let Γ be the subgraph of H_X induced by the edges of all cycles in $\mathcal{C}$. We orient each separating cycle $C \in \mathcal{C}$ such that $\text{ins}(C)$ is on the right side when we proceed on C following its orientation and give C a distinct integer label $\lambda(C)$. We create a directed plane graph $\vec{\Gamma}$ with $V(\vec{\Gamma}) = V(\Gamma)$ and

$$E(\vec{\Gamma}) = \{(u, v)_{\lambda(C)} \quad | \quad \{u, v\} \in E(C), C \in \mathcal{C}, \text{ and the orientation of } C \text{ is from } u \text{ to } v.\}.$$

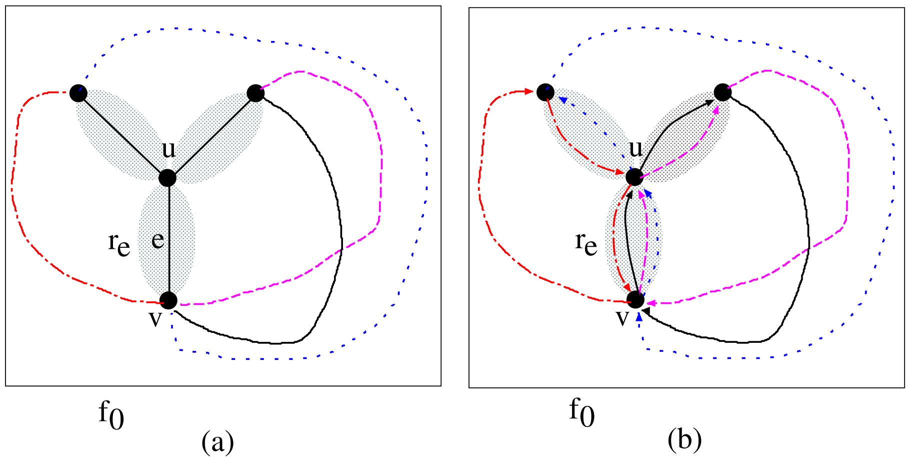


Figure 3.8: (a) embedding of Γ and (b) embedding of $\vec{\Gamma}$.

Notice that if edge $\{u, v\}$ of Γ appears in multiple cycles then $\vec{\Gamma}$ may have parallel arcs from u to v. For simplicity, we may use (u, v) for arc $(u, v)_{\lambda(C)}$ when the label $\lambda(C)$ is not needed in the context. For each cycle $C \in \mathcal{C}$, we denote the corresponding oriented cycle in $\vec{\Gamma}$ by $\vec{C}$. The planar embedding of $\vec{\Gamma}$ is as follows: For each vertex u in $\vec{\Gamma}$, the embedding of u is the point of Σ that is the embedding of vertex u in Γ. For each edge $e = \{u, v\}$ in Γ, let r_e be a region in Σ such that $e \subseteq r_e$, r_e does not have any point of Γ other than e, and $r_e \cap r_{e'} = \emptyset$ for distinct edges e and e' of Γ (see Figure 3.8 (a)). Each arc $\vec{e} = (u, v)_{\lambda(C)}$ in $\vec{\Gamma}$ is embedded as a segment in region r_e, $e = \{u, v\}$ (see Figure 3.8 (b)). We further require the embeddings of arcs in $\vec{\Gamma}$ satisfying the *left-embedding property*: For each edge $e = \{u, v\}$ in Γ, if there is at least one arc from u to v and at least one arc from v to u in $\vec{\Gamma}$ then for any pair of arcs $\vec{e} = (u, v)$ and $\vec{e'} = (v, u)$, the embeddings of $\vec{e}$ and $\vec{e'}$ form an oriented cycle in r_e such that none of f_Z and $f_{\overline{X}}$ is on the left side when we proceed on the cycle following its orientation (see Figure 3.8 (b)). Γ has a face which includes $f_{\overline{X}}$ and we take this face as the outer face f_0 of $\vec{\Gamma}$. Since each edge of H_X appears in at most one boundary cycle C_Z, there are $O(m)$ arcs $(u, v)_{\lambda(C_Z)}$ for all $Z \in \mathcal{Z}_X \setminus \mathcal{W}_X$. Since $|\mathcal{W}_X| = O(m/k)$ and the minimum separating cycle C for every $Z \in \mathcal{W}_X$ has at most k edges, $Z \in \mathcal{W}_X$. Therefore, $\vec{\Gamma}$ can be computed in $O(m)$ time. For each face f in Γ, let $\vec{E}(f)$ be the set of arcs (u, v) and (v, u) in $\vec{\Gamma}$ for each $\{u, v\} \in E(f)$ in Γ.

A search on arc $\vec{e} = (u, v)$ means that we proceed on arc $\vec{e}$ from u to v. For each arc $\vec{e} = (u, v)_{\lambda(C)}$, we define its *next arc* $\mathrm{nx}(\vec{e}) = (v, w)_{\lambda(C)}$ and *previous arc* $\mathrm{pv}(\vec{e}) = (t, u)_{\lambda(C)}$. For arc $\vec{e} = (u, v)_{\lambda(C)}$, let $C(\vec{e})$ be the oriented cycle that contains $\vec{e}$ and let $L(\vec{e}) = \{\vec{h_1} = \mathrm{nx}(\vec{e}), .., \vec{h_t}\}$, $t \geq 1$, be the set of outgoing arcs from v that $\vec{h_i}, 2 \leq i \leq t$, are on the left side of $C(\vec{e})$ when we proceed on $C(\vec{e})$ following its orientation. We assume that arc $\vec{h_i}$, $1 \leq i \leq t$, in $L(\vec{e})$ is the ith outgoing arc from v when we count the arcs incident to v in the counter-clockwise order from $\mathrm{nx}(\vec{e})$ to $\vec{e}$. We define the *leftmost arc* from $\vec{e}$, denoted by $\mathrm{lm}(\vec{e})$, as the $\vec{h_i} \in L(\vec{e})$ with the largest i such that for every $1 \leq j \leq i$,

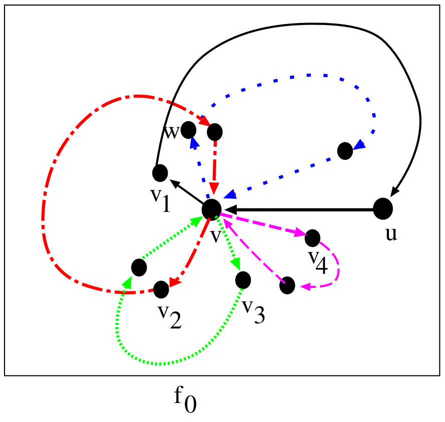


Figure 3.9: For arc $\vec{e} = (u, v)$, $\text{nx}(\vec{e}) = (v, v_1)$, $L(\vec{e}) = \{(v, v_1), (v, v_2), (v, v_3), (v, v_4)\}$ and $\text{lm}(\vec{e}) = (v, v_3)$.

$\text{pv}(\vec{h_j}) \in \text{ins}(C(\vec{h_1})) \cup \cdots \cup \text{ins}(C(\vec{h}_{j-1}))$ holds (see Figure 3.9 for an example). For each arc $\vec{e} = (u, v)_{\lambda(C)}$, $\text{lm}(\vec{e})$ can be found by checking the arcs in $L(\vec{e})$, starting from $\text{nx}(\vec{e})$, in the counter-clockwise order they are incident to v. A search on a sequence of arcs $\vec{e_1}, \vec{e_2}, ..$ is called a *leftmost search* if every $\vec{e}_{i+1}$ is $\text{lm}(\vec{e_i})$ for $i \geq 1$.

By performing a leftmost search on arcs of $\vec{\Gamma}$, starting from an arbitrary arc in $\vec{E}(f_0)$, we can find a separating cycle $\vec{C_m}$ such that for any cycle $\vec{C}$, $C \in \mathcal{C}$, if $\text{ins}(\vec{C_m}) \cap \text{ins}(\vec{C}) \neq \emptyset$ then $\text{ins}(\vec{C}) \subseteq \text{ins}(\vec{C_m})$. We call $\vec{C_m}$ a *maximal cycle*. According to Lemma 3.4.9 and the fact that every cycle $C \in \mathcal{C}$ has a length at most k, the length of $\vec{C_m}$ is at most k.

After finding $\vec{C_m}$, we delete arcs in cycles $\vec{C}$ from $\vec{\Gamma}$ if $\text{ins}(\vec{C}) \subseteq \text{ins}(\vec{C_m})$ to update $\vec{\Gamma}$. We continue the search on the updated $\vec{\Gamma}$ from an arbitrary arc in the updated $\vec{E}(f_0)$ until all arcs are deleted. Then for each $Z \in \mathcal{Z}_X$, there is a unique maximal cycle which separates Z and $\overline{X}$. For every arc $\vec{e}$ in a maximal cycle $\vec{C_m}$, all the arcs in $\text{ins}(\vec{C_m})$ are deleted after $\vec{C_m}$ is found. Each arc is counted $O(1)$ time in the computation for all leftmost arc searches. Therefore, the total time complexity of finding all the maximal cycles is $O(m)$.

For each maximal cycle $\vec{C_m}$ in $\vec{\Gamma}$ computed above, let C_m be the cycle in H_X consisting of edges corresponding to the arcs in $\vec{C_m}$. For each $Z \in \mathcal{Z}_X$, there is a cycle C_m which separates Z and $\overline{X}$. Let A_Z be the edge subset induced by C_m. Then the following holds.

1. $|\partial(A_Z)| \leq k$ (if A_Z is induced by the boundary cycle C_Z then $|\partial(A_Z)| = |C_Z| \leq k$, otherwise A_Z is induced by a cycle C_m of length at most k as shown in Lemma 3.4.8, implying $|\partial(A_Z)| \leq k$).

2. Because $G|U$ is biconnected and $\partial(A_Z) \in E((G|U)|\overline{A_Z}$, $(G|U)|\overline{A_Z}$ is biconnected.

3. Due to the way we find the maximal cycles above, for every $Z \in \mathcal{Z}_X$, there is exactly one subset $A_Z \in \mathcal{A}_X$ separating Z and $\overline{X}$.

4. For distinct $A_Z, A_{Z'} \in \mathcal{A}_X$, $A_Z \cap A_{Z'} = \emptyset$.

By Definition 1, $\mathcal{A}_X$ is a good-separator for $\mathcal{Z}_X$ and $\overline{X}$. □

We are ready to show Theorem 3.1.3 which is re-stated below.

Theorem 3.1.3. *There is an algorithm that given a planar graph G of n vertices and an integer k, in $O(nk^2)$ time, either constructs a branch-decomposition of G with width at most $(2+\delta)k$ or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G, where $\delta > 0$ is a constant.*

Proof. First, as shown in Lemma 3.4.10, a good separator $\mathcal{A}_X$ for $\mathcal{Z}_X$ and $\overline{X}$ is computed by Procedure Separator-Based. From this and as shown in the proof of Theorem 3.1.2, given a planar graph G and an integer k, Algorithm 1 computes a branch-decomposition of G with width at most $(2+\delta)k$ or a $(k+1) \times \left\lceil \frac{k+1}{2} \right\rceil$ cylinder minor of G.

Let M, m_x, m be the numbers of edges in $G[\text{reach}_{G|U}(\partial(U), h)], (G|\overline{X})|\mathcal{Z}_X, \hat{H}_X$, respectively. Then $m = O(m_x)$. By Lemmas 3.4.7 and 3.4.10, Procedure Separator-Based takes $O(mk^2)$ time. For distinct level 1 nodes X and X', the edge sets of subgraphs $(G|\overline{X})|\mathcal{Z}_X$ and $(G|\overline{X'})|\mathcal{Z}_{X'}$ are disjoint. From this, $\sum_{X:X \text{ is a level 1 node}} m_x = O(M)$. Therefore, it takes $\sum_{X:X \text{ is a level 1 node}} O(m_x k^2) = O(Mk^2)$ time to compute a good separator $\mathcal{A}$ when Procedure Separator-Based is used for computing $\mathcal{A}_X$.

The time for other steps in Algorithm 1 is $O(M)$. The number of recursive calls in which each vertex of $G|U$ is involved is $O(\frac{1}{\alpha}) = O(1)$. Therefore, we get an algorithm with running time $O(nk^2)$. □

3.5 Conclusions

In this chapter, we discussed how to improve the construction time of branch-decomposition for planar graphs and therefore improve the preprocessing time of branch-decomposition based exact distance oracles for planar graphs. The practical efficiency of the algorithms for Theorems 3.1.2 and 3.1.3 heavily depends on the efficiency of computing the minimum face separating cycles, especially Line 7 of Algorithm 1 or Procedure Separator-Based. When the size of $\mathcal{Z}_X$ is small, it may be efficient in practice to compute the face-separating cycles using a straightforward approach. Using the near-linear time constant-factor approximate algorithm for branch-decomposition of planar graphs proposed in this chapter, we are able to remove the bottleneck in the preprocessing phase of the branch-decomposition based distance oracle in [65] and improve the preprocessing time from $O(n^{1+\epsilon} + S\log^2 n)$, where $S \in [n \log\log k, n^2]$ and $\epsilon > 0$ is a constant, to $O(\min\{O(n\log^4 n \log k), O(nk^2 \log n \log k)\} + S\log^2 n)$.

Chapter 4

Constant Query Time Approximate Distance Oracle for Planar Graphs

4.1 Introduction

In section 1.1.2, we introduced some representative results on approximate distance oracles. The query times of those oracles are small but still at least $O(1/\epsilon)$. Distance oracles with constant query time are of both theoretical and practical importance [24, 28]. In this chapter we present the first $O(1)$ query time $(1+\epsilon)$-approximate distance oracle with nearly linear preprocessing time and size for undirected planar graphs with non-negative edge lengths.

Theorem 4.1.1. *Let G be an undirected planar graph with n vertices and non-negative edge lengths and let $\epsilon > 0$. There is a $(1+\epsilon)$-approximate distance oracle for G with $O(1)$ query time, $O(n \log n(\log^{1/6} n + \log n/\epsilon + f(\epsilon)))$ size and $O(n \log n(\log^3 n/\epsilon^2 + f(\epsilon)))$ preprocessing time, where $f(\epsilon) = 2^{O(1/\epsilon)}$.*

The oracle in Theorem 4.1.1 has a constant query time independent of ϵ and size nearly linear in the graph size. This improves the query time that are (nearly) linear in $1/\epsilon$ for non-constant ϵ in the previous works [57, 78]. Wulff-Nilsen gives an $O(1)$ time exact distance oracle for G with size $O(n^2(\log\log n)^4/\log n)$ [81]. For ϵ with $\frac{1}{\epsilon} < c_0 \log n$ for some constant c_0, our oracle has a smaller size.

The result in Theorem 4.1.1 can be generalized to an oracle described in the next theorem.

Theorem 4.1.2. *Let G be an undirected planar graph with n vertices and non-negative edge lengths, $\epsilon > 0$ and $1 \leq \eta \leq 1/\epsilon$. There is a $(1+\epsilon)$-approximate distance oracle for G with $O(\eta)$ query time, $O(n \log n(\log^{1/6} n + \log n/\epsilon + f(\eta\epsilon)))$ size and $O(n \log n(\log^3 n/\epsilon^2 + f(\eta\epsilon)))$ preprocessing time, where $f(\eta\epsilon) = 2^{O(1/(\eta\epsilon))}$.*

Our results build on some techniques used in the previous approximate distance oracles for planar graphs. Thorup [78] gives a $(1+\epsilon)$-approximate distance oracle for planar graph

G with $O(1/\epsilon)$ query time. Informally, some techniques used in Thorup's distance oracle are as follows: Decompose G into a balanced recursive subdivision; G is decomposed into subgraphs of balanced sizes by shortest paths and each subgraph is decomposed recursively until every subgraph is reduced to a pre-defined size. Recall that a path Q intersects a path Q' if $V(Q) \cap V(Q') \neq \emptyset$. A set $\mathcal{Q}$ of paths is a path-separator for vertices u and v if every path between u and v intersects a path $Q \in \mathcal{Q}$. Vertices u and v are *shortest-separated* by a path Q if there exist a shortest path between u and v that intersects Q. If vertices u and v have a path-separator $\mathcal{Q}$ then u and v is shortest-separated by some path $Q \in \mathcal{Q}$. For each subgraph X of G, let $\mathcal{P}(X)$ be the set of shortest paths used to decompose X. For each path $Q \in \mathcal{P}(X)$ and each vertex u in X, a set $P_Q(u)$ of $O(1/\epsilon)$ vertices called *portals* on Q is selected. For vertices u and v that are shortest-separated by some path Q in $\mathcal{P}(X)$, $\min_{p \in P_Q(u), q \in P_Q(v), Q \in \mathcal{P}(X)} d_G(u,p) + d_G(p,q) + d_G(q,v)$ is used to approximate $d_G(u,v)$. The oracle keeps the distances $d_G(u,p)$ and $d_G(p,v)$ explicitly.

The portal set $P_Q(u)$ above is vertex dependent. For any path Q in G of length $d(Q)$, there is a set P_Q of $O(1/\epsilon)$ portals such that for any vertices u and v that are shortest-separated by Q, $\min_{p \in P_Q} d_G(u,p) + d_G(p,v) \leq d_G(u,v) + \epsilon d(Q)$ [62]. Based on this and a scaling technique, Kawarabayashi et al. [57] give another $(1+\epsilon)$-approximate distance oracle: Create subgraphs of G such that the vertices in each subgraph satisfy certain distance property (scaling). Each subgraph H of G is decomposed by shortest paths into a r-division of H which consists of $O(|V(H)|/r)$ subgraphs of H, each of size $O(r)$. For each subgraph X of H, let $\mathcal{B}(X)$ be the set of shortest paths used to separate X from the rest of H. For each path $Q \in \mathcal{B}(X)$, a portal set P_Q is selected. For vertices u and v that are shortest-separated by some path $Q \in \mathcal{B}(X)$, $\min_{p \in P_Q, Q \in \mathcal{B}(X)} d_H(u,p) + d_H(p,v)$ is used to approximate $d_H(u,v)$. This oracle does not keep the distances $d_H(u,p)$ and $d_H(p,v)$ explicitly but uses the distance oracle in [65] to get the distances instead. By choosing an appropriate H, the oracle gets an approximate distance for $d_G(U,V)$. By choosing an appropriate value r, the oracle has a better product of query time and oracle size than that of Thorup's oracle.

We also use the scaling technique to create subgraphs of G. We decompose each subgraph H of G into a balanced recursive subdivision as in Thorup's oracle. For each subgraph X of H and each shortest path Q used to decompose X, we choose one set P_Q of $O(1/\epsilon)$ portals on Q for all vertices in X. A new ingredient in our oracle is to use a more time efficient data structure to approximate $d_G(u,v)$ instead of $\min_{p \in P_Q, Q \in \mathcal{P}(X)} d_H(u,p) + d_H(p,v)$. Using an approach in [80], we show that the vertices in $V(X)$ can be partitioned into $s = f(\epsilon)$ classes $A_1, ..., A_s$ such that for every two classes A_i and A_j, there is a key portal $p_{ij} \in P_Q$ and for any $u \in A_i$ and $v \in A_j$, if u and v are shortest-separated by Q then $d_H(u, p_{ij}) + d_H(p_{ij}, v)$ is used to approximate $d_H(u,v)$ and can be computed in $O(1)$ time. Combining with a new technique to select an appropriate subgraph H in $O(1)$ time, we get a $(1+\epsilon)$-approximate distance oracle with $O(1)$ query time.

Our computational model is word RAM, which models what we can program using standard programming languages such as C/C++. In this model, a word is assumed big enough to store any vertex identifier or distance. We also assume basic operations, which include addition, subtraction, multiplication, bitwise operations (AND, OR, NEGATION) and left/right cyclic shift on a word have unit time cost.

4.2 Definitions and Notations

We now introduce some important tools that are used in this chapter.

4.2.1 Recursive Subdivision by Shortest Path

A basic approach in this distance oracle is to decompose graph G into subgraphs by shortest paths.

Definition 4.2.1. *Let $\mathcal{P}$ be any set of shortest paths in graph G and let $W = \cup_{Q \in \mathcal{P}} V(Q)$. $\mathcal{P}$ is a* shortest path separator *of G if $G[V(G) \backslash W]$ has at least $t \geq 2$ connected nonempty subgraphs $G_1, .., G_t$ of G. For $\alpha > 0$, a shortest path separator $\mathcal{P}$ of G is called* α-balanced *if $|V(G_i)| \leq \alpha |V(G)|$ holds for every subgraph G_i.*

Definition 4.2.2. *An* α-balanced recursive subdivision *of G is a structure in which G is decomposed into subgraphs $G_1, .., G_t$ by an α-balanced separator $\mathcal{P}$ and each G_i, $1 \leq i \leq t$, is decomposed recursively until each subgraph is reduced to a pre-defined size.*

Let $\mathcal{Q}$ be any set of paths and let G' be any subgraph of G. $\mathcal{Q}$ *separates* G' and $G[V(G) \backslash V(G')]$ if for any vertex u in G' and any vertex v in $G[V(G) \backslash V(G')]$, any path in G between u and v intersects a path in $\mathcal{Q}$.

We now briefly describe the construction of a recursive subdivision. Readers may refer to Section 2.5 in [78] for more details. When computing an α-balanced recursive subdivision of G, we are given a shortest path spanning tree T_r of G rooted at any vertex r. Recall that every path in T_r from the root r to any vertex is called a root path. A recursive subdivision of G can be viewed as a rooted tree T_G with each vertex of T_G (called a *node*, to be distinguished from a vertex of G) representing a subgraph of G and the root node representing G. Figure 4.1 gives illustration for recursive subdivision. Each node in T_G with node degree one is called a *leaf node*, otherwise an *internal node*. We identify subgraphs of G with their corresponding nodes in T_G when convenient. For each node X of T_G, let $\mathcal{B}(X)$ be the minimum subset of root paths in T_r that separates X and $G[V(G) \backslash V(X)]$ ($\mathcal{B}(G) = \emptyset$). Let $X \cup \mathcal{B}(X)$ denote the subgraph of G induced by $V(X) \cup V(\mathcal{B}(X))$. Let $X + \mathcal{B}(X)$ denote the graph obtained by removing some vertices from $X \cup \mathcal{B}(X)$ as follows: for every vertex v of $\mathcal{B}(X)$ that has degree two in $X \cup \mathcal{B}(X)$, its incident edges (u, v) and (v, w) are replaced by edge (u, w) whose length is the sum of the length of (u, v) and that of (v, w). For each internal node X, a $\frac{1}{2}$-balanced shortest path separator $\mathcal{P}(X)$ of root paths in T_r is used to decompose X into subgraphs $X_1, .., X_t$, $t \geq 2$, as follows: Let $W = V(\mathcal{P}(X))$ and $X_1^*, .., X_t^*$ be the connected components of $G[V(X + \mathcal{B}(X)) \setminus W]$. Then $E(X_i) = E(X) \cap E(X_i^*)$, $1 \leq i \leq t$. Note that $\mathcal{P}(X)$ separates X_i from X_j in X and $\mathcal{B}(X) \cup \mathcal{P}(X)$ separates X_i from X_j in G for $1 \leq i, j \leq t$ and $i \neq j$.

We now state some important properties of the $\frac{1}{2}$-balanced recursive subdivision in the next Lemma.

Lemma 4.2.1. *[78] Given a graph G and any shortest path spanning tree T_r of G, a $\frac{1}{2}$-balanced recursive subdivision T_G of G can be computed in $O(n \log n)$ time such that for each internal node X of T_G, $|V(X_i)| \leq |V(X)|/2$ $(1 \leq i \leq t)$ and $|\mathcal{P}(X)| = O(1)$, and for each node X, $|\mathcal{B}(X)| = O(1)$. Moreover, for each node X of T_G and each root path Q of T_r, if $Q \in \mathcal{B}(X)$, then $Q \in \mathcal{P}(X')$ for some ancestor X' of X in T_G.*

Note that since the size of a subgraph is reduced by at least a factor of 1/2, the depth of T_G is bounded above by $\log n$. For every vertex $v \in V(G)$, let X_v denote the node of T_G of largest depth that contains v. For any $u, v \in V(G)$, we define $X_{u,v}$ to be the *nearest common ancestor* of X_u and X_v in T_G. The next result of [47] shows the nearest common ancestor of any two nodes in a tree can be found in $O(1)$ time.

Lemma 4.2.2. *[47] Given a rooted tree T with n vertices and two vertices x and y in T, there is an algorithm with $O(n)$ preprocessing time and $O(n)$ space that answers the near common ancestor of x and y in $O(1)$ time.*

4.2.2 $O(1/\epsilon)$ Query Time $(1 + \epsilon)$-Approximate Distance Oracle

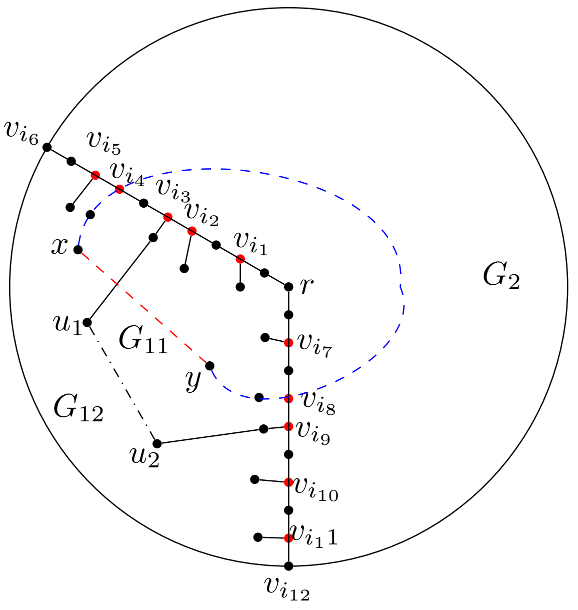
Let Q be a shortest path in G and $\epsilon > 0$. Thorup shows that for every vertex u in G, there is portal set $P_Q(u) \subseteq V(Q)$ of $O(1/\epsilon)$ portals such that for any vertices u and v shortest-separated by Q

$$d_G(u, v) \leq \min_{p \in P_Q(u), q \in P_Q(v)} d_G(v, p) + d_G(p, q) + d_G(q, v) \leq (1 + \epsilon) d_G(u, v).$$

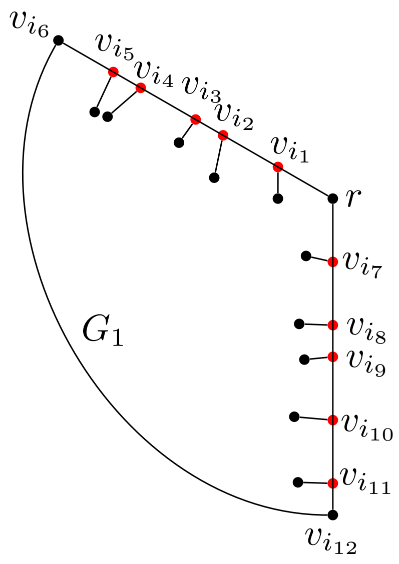
For every subgraph X in a $\frac{1}{2}$-balanced recursive subdivision of G and every shortest path $Q \in \mathcal{B}(X) \cup \mathcal{P}(X)$, by keeping the distance from each vertex u in X to every portal in $P_Q(u)$ explicitly, Thorup shows the following result.

Lemma 4.2.3. *[78] For graph G and $\epsilon > 0$, there is a $(1 + \epsilon)$-approximate distance oracle with $O(1/\epsilon)$ query time, $O(n \log n/\epsilon)$ size and $O(n \log^3 n/\epsilon^2)$ preprocessing time. Especially for $\epsilon = 1$, there is a 2-approximate distance oracle for G with $O(1)$ query time, $O(n \log n)$ size and $O(n \log^3 n)$ preprocessing time.*

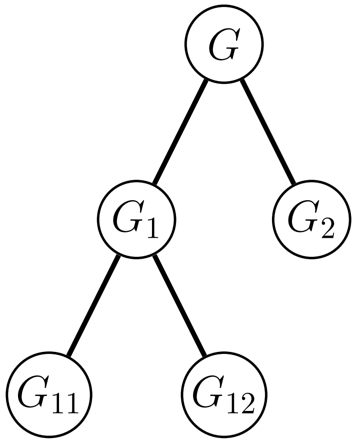
Our oracles will use this oracle for $\epsilon = 1$ (any constant works) to get a rough estimation of $d_G(u, v)$.



(a)



(b)



(c)

Figure 4.1: (a) Root path $T_r(v_{i_6})$ and root path $T_r(v_{i_{12}})$ form a balanced separator of G and decompose G into subgraphs G_1 (not shown in the figure) and G_2. G_1 is further decomposed into G_{11} and G_{12} by $T_r(u_1) \cup T_r(u_2)$. For $X = G_1$, $\mathcal{B}(X) = T_r(v_{i_6}) \cup T_r(v_{i_{12}})$ and $\mathcal{P}(X) = T_r(u_1) \cup T_r(u_2)$. For vertices $x \in V(G_{12})$ and $y \in V(G_{11})$, the shortest path between them must intersect some path in $\mathcal{B}(G_1) \cup \mathcal{P}(G_1)$. (b) $G_1 + \mathcal{B}(G_1)$, vertices in G_1 that are not neighbours of vertices in $\mathcal{B}(G_1)$ are not shown in the figure. (c) T_G representing the recursive subdivision.

4.2.3 Vertex Independent Portal Set

To reduce the query time to a constant independent of ϵ, we will use a portal set P_Q independent of vertex u. For vertices u and v shortest-separated by a path Q, $d_G(u,v) = \min_{p \in V(Q)} d_G(u,p) + d_G(p,v)$. For a $P_Q \subseteq V(Q)$, $\min_{p \in P_Q} d_G(u,p) + d_G(p,v)$ approximates $d_G(u,v)$. The following result will be used.

Lemma 4.2.4. *[62] For a path Q in G, $\epsilon > 0$ and $D \geq d(Q)$, a set P_Q of $O(1/\epsilon)$ vertices in $V(Q)$ can be selected in $O(|V(Q)|)$ time such that for any pair of vertices u and v shortest-separated by Q, $d_G(u,v) \leq \min_{p \in P_Q} d_G(u,p) + d_G(p,v) \leq d_G(u,v) + \epsilon D$.*

The set P_Q in Lemma 4.2.4 is called the *ϵ-portal set (with respect to D)* and every vertex in P_Q is a portal. Given a path Q starting from a vertex r, $\epsilon > 0$ and $D \geq d(Q)$, P_Q can be computed as follows: add r to P_Q, traverse along Q from r and add a vertex $v \in V(Q)$ to P_Q if $d_G(u,v) \geq \epsilon D/2$, where u is the last added portal in P_Q.

4.2.4 Sparse Neighborhood Covers

To apply the ϵ-portal set to our oracle, we further need to guarantee $d_G(u,v) = \Omega(D)$ for vertices u and v in question. We will use the sparse neighborhood covers introduced in [8, 9, 19] of G to achieve this goal.

Lemma 4.2.5. *[19] For G and $\gamma \geq 1$, connected subgraphs $G(\gamma,1), \ldots, G(\gamma,n_\gamma)$ of G with the following properties can be computed in $O(n \log n)$ time:*

1. *For each vertex u in G, there is at least one $G(\gamma,i)$ that contains u and every v with $d_G(u,v) \leq \gamma$.*
2. *Each vertex u in G is contained in at most 18 subgraphs.*
3. *Each subgraph $G(\gamma,i)$ has radius $r(G(\gamma,i)) \leq 24\gamma - 8$.*

4.3 Oracle with Additive Stretch

We first give a distance oracle DO$_1$ which for any vertices u and v in G, and any $\epsilon_0 > 0$, returns $\tilde{d}(u,v)$ with $d_G(u,v) \leq \tilde{d}(u,v) \leq d_G(u,v) + 7\epsilon_0 d(G)$ in $O(1)$ time. Based on the scaling technique in [57], Lemma 4.2.5 and a technique to locate vertices in $O(1)$ time, this oracle will be extended to an oracle stated in Theorem 4.1.1 for G in the next section.

4.3.1 Additive Stretch Distance Oracle with $O(1/\epsilon)$ Query Time

We start with a basic distance oracle DO$_0$ that keeps the following information:

- A $\frac{1}{2}$-balanced recursive subdivision T_G of G in which each leaf node has size $O(2^{(1/\epsilon_0)})$ (Lemma 4.2.1).

- A table storing X_v for every $v \in V(G)$.
- A data structure that answers the nearest common ancestor $X_{u,v}$ for any pair of nodes X_u and X_v in T_G in $O(1)$ time. (Lemma 4.2.2).
- For each internal node X of T_G, keep an ϵ_0-portal set P_Q for every shortest path $Q \in \mathcal{P}(X) \cup \mathcal{B}(X)$. For every P_Q, every $u \in V(X)$ and every portal $p \in P_Q$, keep distance $\hat{d}(u,p)$ with

$$d_G(u,p) \leq \hat{d}(u,p) \leq d_G(u,p) + \epsilon_0 d(G).$$

- For every leaf node X of T_G and every pair of u and v in X, keep

$$\hat{d}_X(u,v) = \min\{d_X(u,v), \min_{p \in P_Q, Q \in \mathcal{B}(X)} \hat{d}(u,p) + \hat{d}(p,v)\}.$$

We now give the query algorithm for distance oracle DO_0.

```
Algorithm 2: DO_0-Query(u, v)
  Input: vertices $u$ and $v$ in $G$
  Output: distance $\tilde{d}(u,v)$ such that $d_G(u,v) \leq \tilde{d}(u,v) \leq d_G(u,v) + 3\epsilon_0 d(G)$
1 Find $X_u, X_v, X_{uv}$;
2 if $X_{uv}$ is a leaf node of $T_G$ then
3     $\tilde{d}(u,v) \leftarrow \hat{d}_X(u,v)$;
4 else
5     $\tilde{d}(u,v) \leftarrow \min_{p \in P_Q, Q \in \mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})} \hat{d}(u,p) + \hat{d}(p,v)$;
6 end
7 return $\tilde{d}(u,v)$;
```

Lemma 4.3.1. *For any vertices u and v in G, DO_0 returns an approximate distance $\tilde{d}(u,v)$ in $O(1/\epsilon_0)$ time such that $d_G(u,v) \leq \tilde{d}(u,v) \leq d_G(u,v) + 3\epsilon_0 d(G)$.*

Proof. Given vertices u and v in G, X_u, X_v, X_{uv} can be found in $O(1)$ time. If X_{uv} is a leaf node of T_G, $\tilde{d}(u,v)$ is computed in $O(1)$ time. Otherwise $\tilde{d}(u,v)$ is computed in $O(1/\epsilon_0)$ time because $|P_Q| = O(1/\epsilon_0)$ and $|\mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})| = O(1)$.

In the case where X_{uv} is not a leaf node in T_G, u and v must be shortest-separated by some path in $\mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})$. Let $P = \cup_{Q \in \mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})} P_Q$ and we define

$$q = \arg_{p \in P} \min\{d_G(u,p) + d_G(p,v)\}.$$

From $\hat{d}(u,q) \leq d_G(u,q) + \epsilon_0 d(G)$, $\hat{d}(q,v) \leq d_G(q,v) + \epsilon_0 d(G)$ and Lemma 4.2.4,

$$\begin{aligned} d_G(u,v) &\leq \tilde{d}(u,v) = \min_{p \in P_Q, Q \in \mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})} \hat{d}(u,p) + \hat{d}(p,v) \leq \hat{d}(u,q) + \hat{d}(q,v) \\ &\leq d_G(u,q) + d_G(q,v) + 2\epsilon_0 d(G) \leq d_G(u,v) + 3\epsilon_0 d(G). \end{aligned}$$

Similarly, $d_G(u,v) \leq \tilde{d}(u,v) = \hat{d}_X(u,v) \leq d_G(u,v) + 3\epsilon_0 d(G)$ holds in the case where X_{uv} is a leaf node in T_G. □

4.3.2 Additive Stretch Distance Oracle with $O(1)$ Query Time

We first reduce the query time for internal nodes in DO_0 to a constant independent of ϵ_0 and then analyse the preprocessing time of the distance oracle. For $z > 0$, let $f(z) = 2^{O(1/z)}$. Based on an approach in [80], we show that for each internal node X and each path $Q \in \mathcal{B}(X) \cup \mathcal{P}(X)$, the vertices in $V(X)$ can be partitioned into $f(\epsilon_0)$ classes such that for any two classes A_i and A_j, there is a key portal $p_{ij} \in P_Q$ and for every $u \in A_i$ and every $v \in A_j$ shortest-separated by Q, $\hat{d}(u,p_{ij}) + \hat{d}(p_{ij},v) \leq d_G(u,v) + 7\epsilon_0 d(G)$. By keeping the classes and key portals, the query time is reduced to $O(1)$. We first define the classes.

Definition 4.3.1. *Let Q be a shortest path in G, $r(G) \leq D \leq d(G)$ and $P_Q = \{p_1 ..., p_l\}$ be an ϵ_0-portal set (with respect to D) on Q. The vertices of G are partitioned into classes based on $\hat{d}(u,p_i), p_i \in P_Q$ as follows. For each vertex u, a vector $\vec{\Gamma}_u = (a_1, ..., a_l)$ is defined such that for $1 \leq i \leq l$, $a_i = \left\lceil \hat{d}(u,p_i)/(\epsilon_0 D) \right\rceil$. Vertices u and v are in the same class if and only if $\vec{\Gamma}_u = \vec{\Gamma}_v$.*

The following property of the classes defined above is straightforward.

Property 1. *Let Q be a shortest path in G, $r(G) \leq D \leq d(G)$ and P_Q be an ϵ_0-portal set with respect to D on Q. Let A be any class of vertices in G defined in Definition 4.3.1. For any two vertices $u, v \in A$ and any portal $p \in P_Q$, $\hat{d}(u,p) - \epsilon_0 D \leq \hat{d}(v,p) \leq \hat{d}(u,p) + \epsilon_0 D$.*

We show more properties of the classes defined above in the next two lemmas.

Lemma 4.3.2. *Let Q be a shortest path in G, $r(G) \leq D \leq d(G)$ and P_Q be an ϵ_0-portal set with respect to D on Q. Let A_i and A_j be any two classes of vertices in G defined in Definition 4.3.1. There is a key portal $p_{ij} \in P_Q$ such that for any vertices $u \in A_i$ and $v \in A_j$ shortest-separated by Q, $d_G(u,v) \leq \hat{d}(u,p_{ij}) + \hat{d}(p_{ij},v) \leq d_G(u,v) + 7\epsilon_0 d(G)$.*

Proof. Figure 4.2 gives an illustration for this proof. We choose arbitrarily a vertex $x \in A_i$ and a vertex $y \in A_j$. Let the key portal be $p_{ij} = \arg_{p_i \in P_Q} \min\{\hat{d}(x,p_i) + \hat{d}(p_i,y)\}$. For any $u \in A_i$ and $v \in A_j$ shortest-separated by Q, let $q = \arg_{p_i \in P_Q} \min\{d_G(u,p_i) + d_G(p_i,v)\}$ and let $p = \arg_{p_i \in P_Q} \min\{\hat{d}(u,p_i) + \hat{d}(p_i,v)\}$. Then

$$\begin{aligned} \hat{d}(u,p) + \hat{d}(p,v) &\leq \hat{d}(u,q) + \hat{d}(q,v) \\ &\leq d_G(u,q) + d_G(q,v) + 2\epsilon_0 d(G) \leq d_G(u,v) + 3\epsilon_0 d(G), \end{aligned}$$

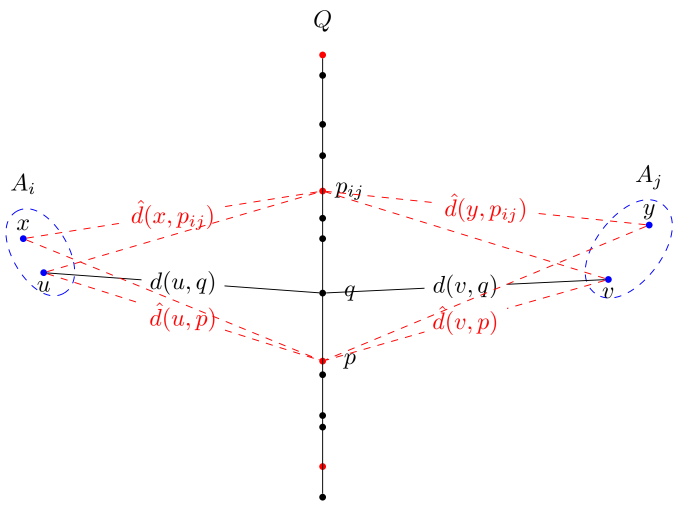


Figure 4.2: Illustration for Lemma 4.3.2. x and y are arbitrary vertices in A_i and A_j respectively. u and v are two vertices in A_i and A_j respectively that are shortest-separated by shortest path Q. Red dots are portal vertices in Q. Solid black lines represents exact distances in G and dashed red lines represents approximate distances $\hat{d}(\cdot,\cdot)$.

because $\hat{d}(u,q) \le d_G(u,q) + \epsilon_0 d(G)$, $\hat{d}(q,v) \le d_G(q,v) + \epsilon_0 d(G)$, P_Q is an ϵ_0-portal set and Lemma 4.2.4. From $u, x \in A_i$, Property 1 and $D \le d(G)$,

$$\hat{d}(u,p_i) \le \hat{d}(x,p_i) + \epsilon_0 D \le \hat{d}(x,p_i) + \epsilon_0 d(G) \le \hat{d}(u,p_i) + 2\epsilon_0 d(G)$$

for every $p_i \in P_Q$. The same relations hold for v, y because they are in A_j. So

$$\begin{aligned}\hat{d}(u,p_{ij}) + \hat{d}(p_{ij},v) &\le \hat{d}(x,p_{ij}) + \hat{d}(p_{ij},y) + 2\epsilon_0 d(G) \\ &\le \hat{d}(x,p) + \hat{d}(p,y) + 2\epsilon_0 d(G) \le \hat{d}(u,p) + \hat{d}(p,v) + 4\epsilon_0 d(G).\end{aligned}$$

Therefore,

$$\begin{aligned}d_G(u,v) &\le \hat{d}(u,p_{ij}) + \hat{d}(p_{ij},v) \le \hat{d}(u,p) + \hat{d}(p,v) + 4\epsilon_0 d(G) \\ &\le d_G(u,v) + 7\epsilon_0 d(G).\end{aligned}$$

This completes the proof of the lemma. □

Lemma 4.3.3. *The total number of classes by Definition 4.3.1 is $f(\epsilon_0)$.*

Proof. Essentially, this result is proved by Weimann and Yuster in [80] but somehow hidden in other details. Below we give a self-contained proof of the lemma. For each vector $\vec{\Gamma}_u = (a_1, .., a_l)$, let $\vec{\Gamma}^*_u = (a_1, (a_2 - a_1), (a_3 - a_2), .., (a_l - a_{l-1}))$. Then $\vec{\Gamma}_u = \vec{\Gamma}_v$ if and only if $\vec{\Gamma}^*_u = \vec{\Gamma}^*_v$. So we just need to prove that the total number of different $\vec{\Gamma}^*_u$ is $f(\epsilon_0)$. From

Definition 4.3.1,

$$\begin{aligned}|a_i - a_{i-1}| &= \left|\left\lceil\frac{\hat{d}(u,p_i)}{\epsilon_0 D}\right\rceil - \left\lceil\frac{\hat{d}(u,p_{i-1})}{\epsilon_0 D}\right\rceil\right| \\ &\le \left|\frac{\hat{d}(u,p_i) - \hat{d}(u,p_{i-1})}{\epsilon_0 D}\right| + 1 \\ &\le \left|\frac{d_G(u,p_i) - d_G(u,p_{i-1})}{\epsilon_0 D}\right| + 2 \le \frac{d_G(p_{i-1},p_i)}{\epsilon_0 D} + 2.\end{aligned}$$

Since P_Q is an ϵ_0-portal set, $l = O(1/\epsilon_0)$. So

$$\sum_{2\le i\le l} |a_i - a_{i-1}| \le \frac{d_G(p_1,p_l)}{\epsilon_0 D} + 2l = O(1/\epsilon_0).$$

Therefore there are $2^{O(1/\epsilon_0)}$ different vectors of $(a_1, |a_2 - a_1|, |a_3 - a_2|, .., |a_l - a_{l-1}|)$. The i'th element of $(a_1, (a_2 - a_1), (a_3 - a_2), .., (a_l - a_{l-1}))$ is either $|a_i - a_{i-1}|$ or $-|a_i - a_{i-1}|$. Therefore, there are $2^{O(1/\epsilon_0)}$ different $\vec{\Gamma}^*_u$. □

Notice that we can assume that for each internal node X, the number of classes Definition 4.3.1 is at most $|V(X)|^2$ because otherwise, instead of partitioning the vertices into classes, we can simply use a $|V(X)| \times |V(X)|$ distance array to keep the shortest distance between every pair of vertices in X.

Now we are ready to show a distance oracle DO$_1$ with $7\epsilon_0 d(G)$ additive stretch. DO$_1$ contains the information kept for DO$_0$ as well as the following additional information:

- For each internal node X of T_G and each shortest path $Q \in \mathcal{B}(X)\cup\mathcal{P}(X)$, let $A^Q_1, ..., A^Q_s$ be the classes of vertices in $V(X)$ defined in Definition 4.3.1. For each vertex $u \in V(X)$, we give an index $I^Q_X(u)$ with $I^Q_X(u) = i$ if $u \in A^Q_i$; and an $s \times s$ array C_Q with $C_Q[i,j]$ containing the key portal p^Q_{ij} for classes A^Q_i and A^Q_j.

Lemma 4.3.4. *For any graph G with n vertices and $\epsilon_0 > 0$, the space requirement for data structure* DO$_1$ *is $O(n(\log n/\epsilon_0 + f(\epsilon_0)))$.*

Proof. Let T_G be the recursive subdivision of G in DO$_1$ and $b = 2^{(1/\epsilon_0)}$. Each leaf node X has $O(b)$ vertices and requires $O(b^2)$ space to keep the distances $\tilde{d}(u,v)$ for u, v in the node. From this and the fact that the sum of $|V(X)|$ for all leaf nodes is $O(n)$, the space for all leaf nodes is $O(nb) = O(nf(\epsilon_0))$. By Lemma 4.2.1, the sum of $|V(X)|$ for all nodes X in T_G is $O(n\log n)$. From $|\mathcal{B}(X) \cup \mathcal{P}(X)| = O(1)$ for every X and $|P_Q| = O(1/\epsilon_0)$ for each $Q \in \mathcal{B}(X) \cup \mathcal{P}(X)$, the total space for keeping the distances $\hat{d}(u,v)$ between vertices and portals is $O(n\log n/\epsilon_0)$. By Lemma 4.3.3, the space for the classes $A^Q_1, .., A^Q_s$ in each internal node X is $f(\epsilon_0)$ for every $Q \in \mathcal{B}(X) \cup \mathcal{P}(X)$. Since there are $O(n)$ internal nodes, the total space for the classes in all nodes is $O(nf(\epsilon_0)) = O(nf(\epsilon_0))$.

Therefore the space requirement for the oracle is $O(n(\log n/\epsilon_0 + f(\epsilon_0)))$. □

We now give the query algorithm and query time for distance oracle DO_1.

Algorithm 3: DO_1-Query(u, v)

```
   Input: vertices $u$ and $v$ in $G$
   Output: distance $\tilde{d}(u,v)$ such that $d_G(u,v) \le \tilde{d}(u,v) \le d_G(u,v) + 7\epsilon_0 d(G)$
 1 Find $X_u, X_v, X_{uv}$;
 2 if $X_{uv}$ is a leaf node of $T_G$ then
 3     $\tilde{d}(u,v) \leftarrow \hat{d}_X(u,v)$;
 4 else
 5     $\tilde{d}(u,v) \leftarrow \infty$;
 6     for each path $Q \in \mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})$ do
 7         $i \leftarrow I_X^Q(u)$, $j \leftarrow I_X^Q(v)$;
 8         $p_{ij}^Q \leftarrow C_Q[i,j]$;
 9         $\tilde{d}(u,v) \leftarrow \min\{\tilde{d}(u,v), \hat{d}(u,p_{ij}^Q) + \hat{d}(p_{ij}^Q, v)\}$;
10     end
11 end
12 return $\tilde{d}(u,v)$;
```

Lemma 4.3.5. *For any vertices u and v in G, DO_1 returns an approximate distance $\tilde{d}(u,v)$ in $O(1)$ time such that $d_G(u,v) \le \tilde{d}(u,v) \le d_G(u,v) + 7\epsilon_0 d(G)$.*

Proof. Similar to the proof of Lemma 4.3.1, if X_{uv} is a leaf node of T_G, DO_1 returns $d_G(u,v) \le \tilde{d}(u,v) \le d_G(u,v) + 3\epsilon_0 d(G)$ in $O(1)$ time. If X_{uv} is an internal node of T_G, u and v are shortest-separated by some path in $\mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})$. Then by Lemma 4.3.2 and the fact that $|\mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})| = O(1)$, DO_1 returns $\tilde{d}(u,v) = \min_{p_{ij}^Q, Q \in \mathcal{B}(X_{u,v}) \cup \mathcal{P}(X_{u,v})} \hat{d}(u, p_{ij}^Q) + \hat{d}(p_{ij}^Q, v) \le d_G(u,v) + 7\epsilon_0 d(G)$ in $O(1)$ time. □

We now describe how to compute the distances $\hat{d}(d,p)$ for internal nodes as defined in DO_0. The method is essentially the same as in the fast construction in [78], but simpler as the portal sets we use are not vertex dependent. We use Lemma 4.2.1 to get the recursive subdivision T_G of G. Let T_r be the shortest path spanning tree of G used in the computation of T_G (see section 4.2.1) and let D be the largest length of any root path of T_r. By a depth first search of T_r from r, we compute an ϵ_0-portal set P_Q (with respect to D) and an *auxiliary* $(\epsilon_0/\log n)$*-portal set* Γ_Q (with respect to D) for every $Q \in \mathcal{P}(X) \cup \mathcal{B}(X)$, $X \in V(T_G)$. We compute the distances $\hat{d}(u,p)$ for every internal node X in a top-down traversal on T_G from root G. We use Dijkstra's algorithm to compute $\hat{d}(u,p)$ for every u in X and every $p \in P_Q \cup \Gamma_Q$, $Q \in \mathcal{P}(X)$. Recall that $X \cup \mathcal{B}(X)$ denotes the subgraph of G induced by $V(X) \cup V(\mathcal{B}(X))$. Let $X \star \mathcal{B}(X)$ denote the graph obtained from $X \cup \mathcal{B}(X)$ as follows (see Figure 4.3 for illustration): for every u in X and every p' in $\cup_{Q' \in \mathcal{B}(X)} P_{Q'} \cup \Gamma_{Q'}$, add edge $\{u, p'\}$ with length $\hat{d}(u,p')$ for every u in X and every p', and then remove degree two vertices of $\mathcal{B}(X)$ as what we do for $X + \mathcal{B}(X)$. For the root node, the computation is on

G. For an internal node $X \neq G$, the computation is on $X \star \mathcal{B}(X)$. Note that $X \star \mathcal{B}(X)$ may not be planar and since $|\mathcal{B}(X)| = O(1)$, $|V(X \star \mathcal{B}(X))|$ is linear in the number of edges of G incident to vertices of X plus the number of portals in each path. Notice that Q' is in $\mathcal{P}(X')$ for some internal node X' which is an ancestor of X in T_G. So the distances $\hat{d}(u, p')$ have been computed when we construct $X \star \mathcal{B}(X)$. Note that for some vertex u and portal p, $\hat{d}(u, p)$ may be computed multiple times. But the value of $\hat{d}(u, p)$ does not change: let $H_i, 1 \le i$, be the graph on which $\hat{d}(u, p)$ is computed for the ith time; $\hat{d}(u, p)$ does not increase because the edge $\{u, p\}$ is contained in H_i for $i \ge 2$; and $\hat{d}(u, p)$ does not decrease because H_{i+1} is a subgraph of H_i for $i \ge 2$. For X with $|V(X)| \ge \log n/\epsilon_0$, we run Dijkstra's algorithm using every $p \in P_Q \cup \Gamma_Q$ as the source. For X with $|V(X)| < \log n/\epsilon_0$, we run Dijkstra's algorithm using every u in X as the source. After the distances $\hat{d}(u, p)$, $p \in P_Q \cup \Gamma_Q$, for all internal nodes have been computed, we only keep the distances $\hat{d}(u, p)$ to the portals $p \in P_Q$ for every internal node.

In the next lemma, we show that the distances $\hat{d}(u, p)$ computed above meet the requirement of DO_0 and DO_1.

Lemma 4.3.6. *For every internal node X of T_G, every vertex u in X and every portal $p \in P_Q \cup \Gamma_Q$, $Q \in \mathcal{P}(X)$, $\hat{d}(u, p) \le d_G(u, p) + \epsilon_0 d(G)$.*

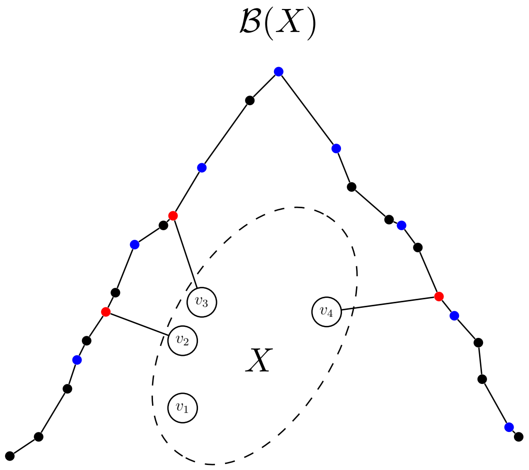
Proof. For every internal node X of depth k in T_G, every vertex u in X and every portal $p \in P_Q \cup \Gamma_Q, Q \in \mathcal{P}(X)$, we prove by induction that $\hat{d}(u, p) \le d_G(u, p) + \frac{k\epsilon_0}{\log n} d(G)$. For the root node (of depth 0), $\hat{d}(u, p) = d_G(u, p)$ because the distances are computed on G. Assume that for every internal node of depth at most $k - 1 \ge 0$, $\hat{d}(u, p) \le d_G(u, p) + \frac{(k-1)\epsilon_0}{\log n} d(G)$. Let X be a node of depth k. For u in X and $p \in P_Q \cup \Gamma_Q$, $Q \in \mathcal{P}(X)$, let $P(u, p)$ be a shortest path between u and p. If $P(u, p)$ contains only edges in X then $\hat{d}(u, p) = d_G(u, p)$ and the statement is proved. Otherwise, $P(u, p)$ can be partitioned into two subpaths $P(u, y)$ and $P(y, p)$, where every vertex of $P(u, y)$ except y is in X and y is a vertex of a path $Q' \in \mathcal{B}(X)$. Note that y is incident to some vertex of X so y appears in $X \star \mathcal{B}(X)$. From the way $\Gamma_{Q'}$ is computed and the fact that $D \le d(G)$, where D is used for computing $\Gamma_{Q'}$, there is a portal $p_y \in \Gamma_{Q'}$ such that $d_{Q'}(y, p_y) = d_G(y, p_y) \le \frac{\epsilon_0}{2\log n} d(G)$. Let X' be an ancestor of X such that $Q' \in \mathcal{P}(X')$. Note that $\hat{d}(u, p_y)$ is computed in $X' \star \mathcal{B}(X')$ and that y, p_y and X (and therefore $P(u, y)$) are all contained in $X' \star \mathcal{B}(X')$. Therefore,

$$\hat{d}(u, p_y) \le d(P(u, y)) + d_{Q'}(y, p_y) \le d(P(u, y)) + \frac{\epsilon_0}{2\log n} d(G)$$

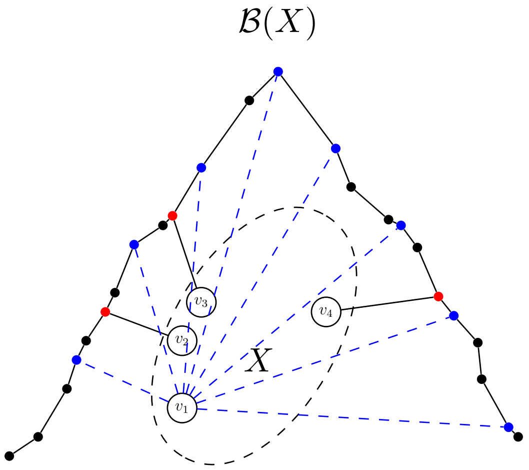
and

$$d_G(p_y, p) \le d(P(y, p)) + d_G(y, p_y) \le d(P(y, p)) + \frac{\epsilon_0}{2\log n} d(G).$$

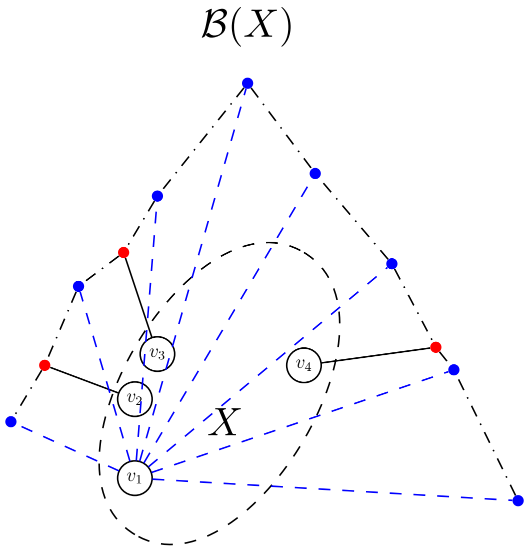
The distance $\hat{d}(p_y, p)$ has been computed in a node X' which is an ancestor of X and has depth at most $k - 1$. So $\hat{d}(p_y, p) \le d_G(p_y, p) + \frac{(k-1)\epsilon_0}{\log n} d(G)$. Because edges $\{u, p_y\}$ and $\{p_y, p\}$



(a) $X \cup \mathcal{B}(X)$. Edges in X are not shown. Blue dots in $\mathcal{B}(X)$ are portals. Red dots in $\mathcal{B}(X)$ are vertices with neighbours in X.



(b) $X \cup \mathcal{B}(X)$ and the edges added between vertices in X and the portals (only edges incident to v_1 are shown).



(c) $X \star \mathcal{B}(X)$. Edges between vertices in X are not shown. Edges between $v_i, i = 2, 3, 4$ and the portals are not shown.

Figure 4.3: Illustration for $X \star \mathcal{B}(X)$.

with lengths $\hat{d}(u, p_y)$ and $\hat{d}(p_y, p)$ are contained in the graph $X \star \mathcal{B}(X)$,

$$\begin{aligned}\hat{d}(u,p) &\leq \hat{d}(u,p_y) + \hat{d}(p_y,p)\\ &\leq d(P(u,y)) + \frac{\epsilon_0}{2\log n} + d(P(y,p)) + \frac{\epsilon_0}{2\log n} + \frac{(k-1)\epsilon_0}{\log n} d(G)\\ &= d_G(u,p) + \frac{k\epsilon_0}{\log n} d(G).\end{aligned}$$

Since each node in T_G has depth at most $\log n$, $\hat{d}(u,p) \leq \epsilon_0 d(G)$. □

The next lemma gives the preprocessing time for distance oracle DO_1.

Lemma 4.3.7. *For any graph G with n vertices and $\epsilon_0 > 0$, distance oracle DO_1 can be computed in $O(n(\log^3 n/\epsilon_0^2 + f(\epsilon_0)))$ time.*

Proof. Let T_G be the recursive subdivision of G and $b = 2^{(1/\epsilon_0)}$. It takes $O(n \log n)$ time to compute T_G (Lemma 4.2.1). It takes $O(n)$ time to compute the data structure that can answer the least common ancestor of any two nodes in T_G in $O(1)$ time [47], $O(n \log n)$ time to compute X_v for every $v \in V(G)$, and $O(n)$ time to compute $P_Q \cup \Gamma_Q$ for every path $Q \in \mathcal{B}(X) \cup \mathcal{P}(X)$, $X \in V(T_G)$.

For every node X, let $M(X)$ be the number of edges in G incident to vertices in X. Then $\sum_{X \in T_G} M(X) = O(n \log n)$. For every path Q, $|P_Q \cup \Gamma_Q| = O(\log n/\epsilon_0)$ and for every node X in T_G, $|\mathcal{B}(X) \cup \mathcal{P}(X)| = O(1)$. From this, for each internal node X, $X \star \mathcal{B}(X)$ has $O(M(X) + \log n/\epsilon_0)$ vertices and $O(M(X) + \log n/\epsilon_0)$ edges. Dijkstra's algorithm is executed $\min\{M(X), \log n/\epsilon_0\}$ times for each node X. It takes $O(M(X)(\log n/\epsilon_0)^2)$ time to compute all $\hat{d}(u,p)$ for node X. Since the sum of $M(X)$ for all nodes X of the same depth is $O(n)$, it takes $O(n(\log n/\epsilon_0)^2)$ time for all internal nodes of the same depth. Since T_G has depth $O(\log n)$, it takes $O(n \log^3 n/\epsilon_0^2)$ time to compute all distances $\hat{d}(u,p)$ for all internal nodes.

To find $\tilde{d}(u,v)$ for a leaf node X, we first use Dijkstra's algorithm to compute $d_X(u,v)$, taking every vertex of X as the source. This takes $O(b^2 \log b) = O(b^2/\epsilon_0)$ time for one leaf node since $|V(X)| = O(b)$ and $O(nb/\epsilon_0)$ time for all leaf nodes since the sum of $|V(X)|$ for all leaf nodes X is $O(n)$. Then we compute

$$\tilde{d}(u,v) = \min\{d_X(u,v), \min_{p \in P_Q, Q \in \mathcal{B}(X)} \hat{d}(u,p) + \hat{d}(p,v)\}.$$

From $|P_Q| = O(1/\epsilon_0)$ for $Q \in \mathcal{B}(X)$ and $|\mathcal{B}(X)| = O(1)$, this takes $O(b^2/\epsilon_0)$ time for one leaf node and $O(nb/\epsilon_0)$ time for all leaf nodes. The total time to compute $\tilde{d}(u,v)$ for all leaf nodes is $O(nb/\epsilon_0)) = O(nf(\epsilon_0))$.

The value D for computing the classes can be found in $O(n)$ time. Since there are $O(n)$ internal nodes, by Lemma 4.3.3, it takes $O(nf(\epsilon_0)(1/\epsilon_0)) = O(nf(\epsilon_0))$ time to compute all classes and key portals. Therefore, DO_1 can be computed in $O(n(\log^3 n/\epsilon_0^2 + f(\epsilon_0)))$ time. □

From Lemmas 4.3.4, 4.3.5 and 4.3.7, we have the following result.

Theorem 4.3.8. *For any graph G with n vertices and $\epsilon_0 > 0$, there is an oracle which gives a distance $\tilde{d}(u,v)$ with $d_G(u,v) \leq \tilde{d}(u,v) \leq d_G(u,v) + 7\epsilon_0 d(G)$ for any vertices u and v in G with $O(1)$ query time, $O(n(\log n/\epsilon_0 + f(\epsilon_0)))$ size and $O(n(\log^3 n/\epsilon_0^2 + f(\epsilon_0)))$ preprocessing time.*

We can make the oracle in Theorem 4.3.8 a more generalized one: For integer η satisfying $1 \leq \eta \leq 1/\epsilon_0$, we partition each path $Q \in \mathcal{B}(X) \cup \mathcal{P}(X)$ into η segments $Q_1, .., Q_\eta$, compute the classes $A_1^{Q_l}, .., A_s^{Q_l}$ of vertices in $V(X)$ for each segment Q_l, $1 \leq l \leq \eta$, and key portal $p_{ij}^{Q_l}$, and use

$$\tilde{d}(u,v) = \min_{p_{ij}^{Q^l}, 1 \leq l \leq \eta, Q \in \mathcal{B}(X) \cup \mathcal{P}(X)} \hat{d}(u, p_{ij}^{Q_l}) + \hat{d}(p_{ij}^{Q_l}, v)$$

to approximate $d_G(u,v)$. By this generalization, we get the following result.

Theorem 4.3.9. *For any graph G with n vertices, $\epsilon_0 > 0$ and $1 \leq \eta \leq 1/\epsilon_0$, there is an oracle which gives a distance $\tilde{d}(u,v)$ with $d_G(u,v) \leq \tilde{d}(u,v) \leq d_G(u,v) + 7\epsilon_0 d(G)$ for any vertices u and v in G with $O(\eta)$ query time, $O(n(\log n/\epsilon_0 + f(\eta\epsilon_0)))$ size and $O(n(\log^3 n/\epsilon_0^2 + f(\eta\epsilon_0)))$ preprocessing time.*

4.4 Oracle with $(1+\epsilon)$ stretch

For $\epsilon > 0$, by choosing an $\epsilon_0 = \frac{\epsilon}{7c}$ where $c > 0$ is a constant, the oracle in Theorem 4.3.8 gives a $(1+\epsilon)$-approximate distance oracle for graph G with $d_G(u,v) \geq d(G)/c$ for every u and v in G. For graph G with $d_G(u,v)$ much smaller than $d(G)$ for some u and v, we use a scaling approach [57] to get a $(1+\epsilon)$-approximate distance oracle. The idea is to compute a set of oracles as described in Theorem 4.3.8, each for a computed subgraph H of G. We further develop a data structure that given any u and v in G, find in $O(1)$ time a constant number of subgraphs (and the corresponding oracles) such that the minimum value returned by these oracles is a $(1+\epsilon)$-approximation of $d_G(u,v)$. Therefore a $(1+\epsilon)$-approximate distance for any u, v can be computed in constant time. We assume $\epsilon > 5/n$, otherwise a naive exact distance oracle with $O(1)$ query time and $O(n^2)$ space can be used to prove Theorem 4.1.1.

4.4.1 Scaling

Let l_m be the smallest edge length in G. We assume $l_m \geq 1$ and the case where $l_m < 1$ can be easily solved in a similar way by normalizing the length of each edge e of G to $l(e)/l_m$. For each scale $\gamma \in \{2^i | 0 \leq i \leq \lceil \log d(G) \rceil\}$, we contract every edge e with $l(e) < \gamma/n^2$

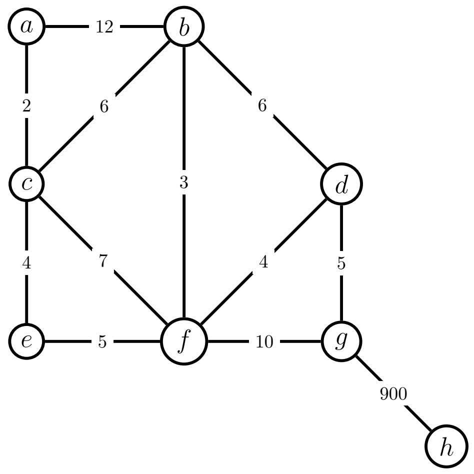
in G and remove every edge e with $l(e) > 24\gamma$, self loops and degree 0 vertices to get a contracted graph G_γ, and then compute a sparse cover $\mathcal{C}_\gamma = \{G(\gamma, j), j = 1, ..., n_\gamma\}$ of G_γ as in Lemma 4.2.5. When an edge $\{u, v\}$ is contracted in a graph, $\{u, v\}$ is removed, vertices u and v are replaced by a new vertex w, and every edge other than $\{u, v\}$ incident to u or v in the graph is made incident to w. Figure 4.4 gives an example for the construction for each G_γ. We say the new vertex w covers vertices u and v. We say a vertex u covers u itself and if a vertex x covers a vertex w then x covers every vertex covered by w. We say an edge $\{u, v\}$ of G appears in scale γ if G_γ has an edge $\{u, v\}$ such that x and y cover u and v, respectively. We say a vertex x appears in scale γ if G_γ contains x. From the construction of G_γ, each edge of G only appears in scales γ satisfying $l(e)/24 \leq \gamma \leq l(e)/n^2$ and thus appears in $O(\log n)$ different scales.

We now show how to effectively contract edges and construct G_γ in each level. As shown in Figure 4.5, we keep a sorted list L containing all the edges of G such that the lengths of the edges are in a non-decreasing order. Besides we keep three pointers p_h, p_f and p_l, such that in each scale γ these three pointers point to the first edge that appears in scale $\gamma - 1$, the first edge that appears in scale γ and the first edge to be removed in scale γ respectively. Let E_C denote the edges to be contracted in the current scale. We first find out all vertices in $V(E_C)$, create new vertices for the contractions and then update the edges in G_γ with the new vertices. Since each edge is contracted at most once and each edge in G_γ is updated in $O(1)$ time, the time complexity of contraction is $O(n)$ and the time complexity of constructing G_γ for all scales is $O(n \log n)$.

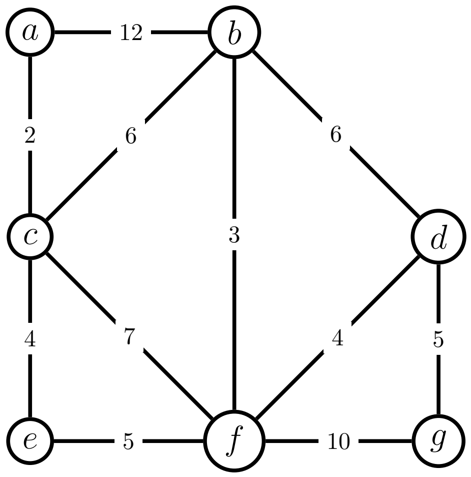
Each G_γ either contains no vertex or contains at least one edge. A scale is *non-trivial* if G_γ contains at least one edge. Since each edge of G appears in $O(\log n)$ scales, there are $O(n \log n)$ non-trivial scales and the total number of vertices appearing in each non-trivial scale is $O(n \log n)$. We can have a bijection $\phi : \{\gamma | \gamma \text{ is a non-trivial scale}\} \rightarrow \{0, 1, ..., N\}$, where $N = O(n \log n)$, such that for every non-trivial scales $\gamma < \gamma'$, $\phi(\gamma) < \phi(\gamma')$. By the bijection ϕ, we can assume that the non-trivial scales are $0, 1, ..., N$. In what follows, we use *scale* for a non-trivial scale.

4.4.2 $O(1)$ Query Time and $\tilde{O}(n)$ Space

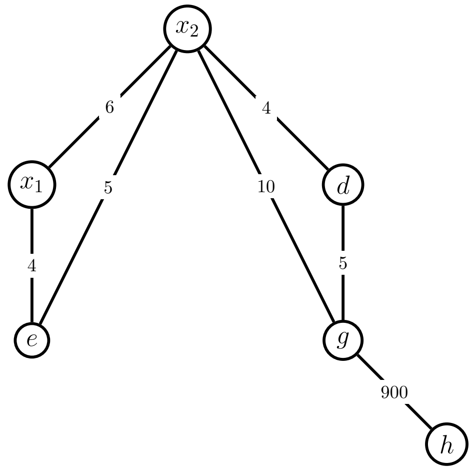
Our $(1+\epsilon)$-approximate distance oracle first estimates roughly the distance $d(u, v)$ to get a right scale γ and then uses oracles $\text{DO}_0(\gamma, j)$ and the vertices in subgraphs $G(\gamma, j)$ that cover u and v to find an approximate distance $\tilde{d}(u, v)$. Though the scaling approach help reduce the total size of the distance oracle, it creates difficulty to find the correct distance oracle. It is not trivial to find the vertices in $G(\gamma, j)$ that cover u and v in $O(1)$ time and $\tilde{O}(n)$ space. A simple approach to find the vertices in $G(\gamma, j)$ that cover u and v in $O(1)$ time is to keep for each vertex u of G explicitly every vertex covering u, but this requires $O(n^2 \log n)$ space, too large. To reduce the space, we create a rooted tree T_C of size $O(n \log n)$ on the vertices in each scale such that if vertex x covers vertex u then x is an ancestor of u in T_C. We



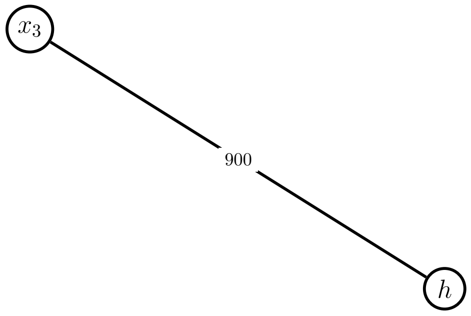
(a) Graph G with 8 vertices and diameter $d(G) = 919$. This is also G_γ for $\gamma = 64, 128$.



(b) G_γ for $\gamma = 1, 2, 4, 8, 16, 32$. Edge $e = (g, h)$ is removed because $l(e) > 24\gamma$ so it will not appear in any subgraph in the sparse neighbourhood cover for scale γ.



(c) G_γ for $\gamma = 256$. Edge (a, c) is contracted into a new vertex x_1 and edge (b, f) is contracted into a new vertex x_2.



(d) G_γ for $\gamma = 512$.

Figure 4.4: Illustration for scaling.

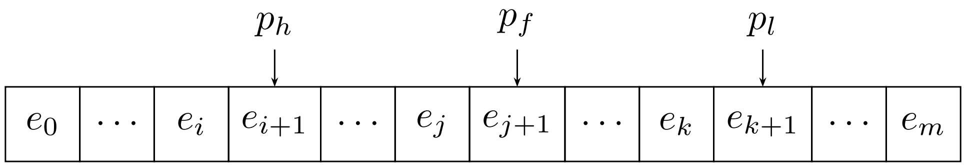


Figure 4.5: A sorted list containing all the edges $e_0, \dots, e_m$ in G. Pointers p_h, p_f and p_l point to the corresponding edges in some scale γ. Edges $e_0, \dots, e_i$ are contracted in scales 1 to $\gamma - 1$. $e_{i+1}, \dots, e_j$ are the edges to be contracted in scale γ. $e_{j+1}, \dots, e_k$ are the edges appearing in scale γ. Edges $e_{k+1}, \dots, e_m$ are to be removed in scale γ.

further create a data structure of size $O(|T_C| \log^{1/6} n)$ (called *Find-Ancestor*) which, given a vertex u of G and a scacle γ such that u is covered by some vertex x in scale γ, answers x in $O(1)$ time. A vertex w (either a vertex of G or one from the contraction of edges) may appear in multiple scales. Let $w(\gamma)$ denote vertex w appearing in scale γ . For $\gamma \neq \gamma'$, $w(\gamma)$ and $w(\gamma')$ are distinct vertices in T_C. The construction of T_C is as follows:

1. Initially, each vertex u of G is given a scale label $\text{sl}(u) = 0$.
2. For each scale $\gamma = 1, 2, \dots, N$ and each vertex $w(\gamma)$ that appears in scale γ, if $w(\gamma)$ is a new vertex caused by the contraction of edges $e_1, \dots, e_k$ then for each vertex $u \in \cup_{1 \leq i \leq k} e_i$, include edge $\{w(\gamma), u(\text{sl}(u))\}$ in T_C, otherwise include edge $\{w(\gamma), w(\text{sl}(w))\}$ in T_C; update $\text{sl}(w)$ to γ.
3. After Step 2, T_C is a forest. Create a vertex r and an edge between r and the root of each tree in the forest to get a tree T_C with root r.

Figure 4.6 gives an illustration of the rooted tree T_C for the graph in Figure 4.4a. Since each vertex of T_C except the root r is a vertex of G_γ and the total number of vertices in all scales is $O(n \log n)$, the size of T_C is $O(n \log n)$. There is a bijection between the set of leaf vertices of T_C and $V(G)$. For each vertex x of T_C, each ancestor of x covers x in some G_γ. Given a vertex u of G and a scale γ, the next lemma in [12] provides a base to find the ancestor $w(\gamma)$ of u in T_C.

Lemma 4.4.1. *[12] Let T be any tree with n vertices. There is an algorithm which given any vertex $x \in V(T)$ and an integer d no larger than the depth of x in T, answers the ancestor of x with depth d in $O(1)$ time. The space requirement of the algorithm is $O(n \log n)$.*

We now describe the algorithm of Lemma 4.4.1. First T is decomposed into disjoint paths: find a longest root-leaf path P in T and remove P from T; the removal of P breaks the remaining of T into subtrees $T_1, T_2, \dots$; each subtree is decomposed recursively by removing the longest root-leaf path in the subtree to get a set of disjoint paths. Each vertex is in exactly one such path and each path contains exactly one leaf vertex of T. For every

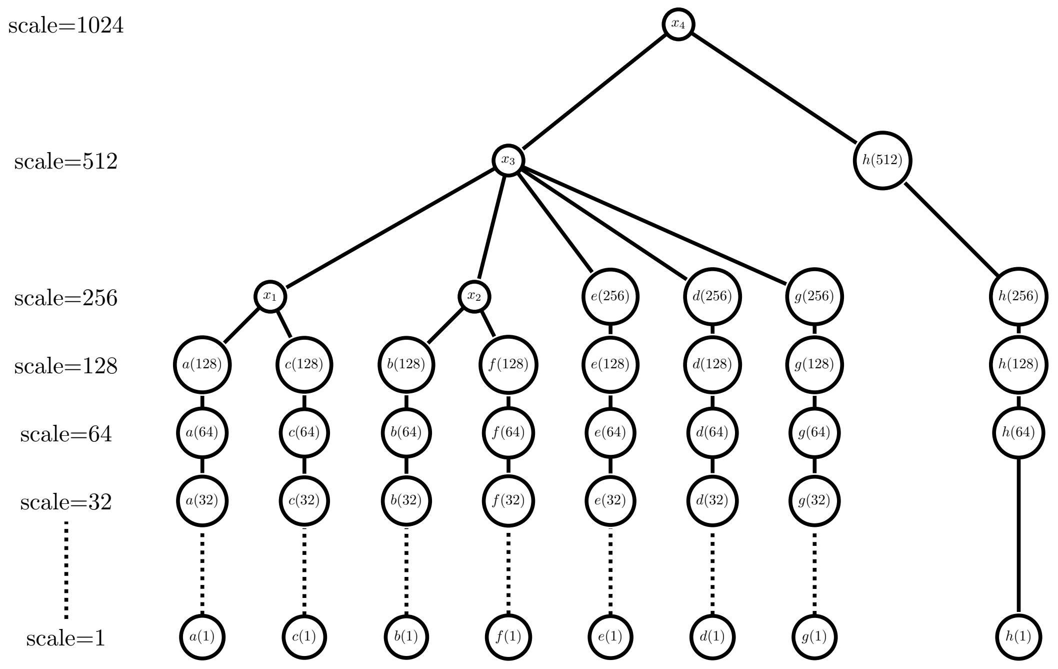


Figure 4.6: Rooted tree T_C for the graph in Figure 4.4a.

leaf u in T, let P_u denote the path that contains u. For every vertex x in T, let $\mathrm{d}(x)$ denote the depth of x (i.e. the number of edges in the path between x and the root in T). For every vertex x of T and $0 \leq k \leq \mathrm{d}(x)$, an ancestor y of x is the kth ancestor of x if y is an ancestor of x and $\mathrm{d}(y) - \mathrm{d}(x) = k$. Secondly for every path P_u, a ladder Q_u containing no more than $2|V(P_u)|$ vertices is created such that Q_u is the path in T between u and its kth ancestor, where $k = \min\{2|V(P_u)| - 2, \mathrm{d}(u)\}$. Store each Q_u in an array. Notice that each vertex of T is in one root-leaf path but may be in multiple ladders. But the size of all Q_u is $O(|T|)$. For each vertex v in P_u, we say that the ladder Q_u created for P_u is v's ladder. For each vertex x of T, store $\mathrm{d}(x)$ and a pointer to x's ladder. Lastly for each vertex x of T, store pointers (shortcuts) $e_i, i = 0, 1, \ldots, \lfloor \log d(x) \rfloor$, such that e_i points to the 2^ith ancestor y_i of x. We call the ancestors pointed to by shortcuts the *critical ancestors* of x. The next lemma shows an important property of the shortcut and ladder scheme.

Lemma 4.4.2. *[12] For any x in T, every kth ancestor y of x with $2^i \leq k < 2^{i+1}$ is in y_i's ladder.*

For each vertex x of tree T, the depths of all ancestors of x are consecutive integers $0, 1, \ldots, d(x)$. By this property and Lemma 4.4.2, it is easy to find the depth d ancestor of x in $O(1)$ time as follows: First use d and the depth of x to find in $O(1)$ time (the pointer pointing to) the correct critical ancestor y_i of x such that the depth d ancestor of x is in y_i's ladder; Then use d and the depth of y_i to locate the depth d ancestor of x in the array containing y_i's ladder in $O(1)$ time. Figure 4.7 gives an example.

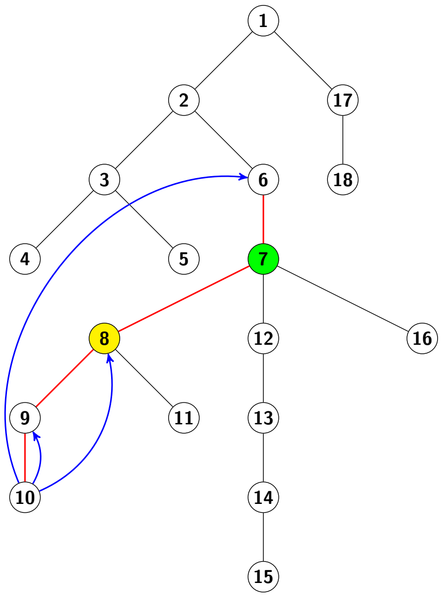


Figure 4.7: An example of finding the third ancestor (vertex 7) of vertex 10. Shortcuts for vertex 10 are expressed by blue darts. Vertex 8 is the critical ancestor whose ladder must contain the third ancestor (vertex 7) of vertex 10. The ladder of vertex 8 is expressed by red lines.

Our problem of finding the ancestor $w(\gamma)$ of u is slightly different from the one in [12] as we do not know the depth of $w(\gamma)$ in T_C and we only need to find $w(\gamma)$ for each leaf u of T_C. First we compute the ladders for tree T_C as the algorithm for Lemma 4.4.1 and then for each leaf u of T_C we create shortcuts connecting u to all its critical ancestors. For every vertex x in T we also store a pointer to x's ladder. Given a vertex u of G and a scale γ, we need to find the the ancestor $w(\gamma)$ of u in T_C using γ instead of the depth of $w(\gamma)$. As in [12], we also first find a correct ancestor x_i of u such that x_i's ladder contains $w(\gamma)$ and then find $w(\gamma)$ from x_i's ladder in $O(1)$ time. But our tasks are more complicated than those in [12] because the scales of all ancestors of u may not be consecutive integers and we can not find the depth of $w(\gamma)$ in $O(1)$ time from the scale γ. Therefore we use following result from [36] to find x_i in $O(1)$ time.

Lemma 4.4.3. *[36] Let b be the machine word size. Given a set B of $|B| \leq \lfloor b^{1/6} \rfloor$ non-negative integer from $0, 1, \ldots, 2^b - 1$, there is a data structure that given any integer k, returns in $O(1)$ time the largest integer in B that is no larger than k. The construction time of the data structure is $O(|B|^4)$ and the size of the data structure is $O(|B|^2)$.*

The next Lemma is an immediate result of Lemma 4.4.3 by creating a B-tree with $|B| = \theta(w^{1/6})$.

Lemma 4.4.4. *For any set S of non-negative integers from $0, 1, \dots, 2^w - 1$, where w is no larger than the machine word size, there is a data structure on the word RAM model that given any integer k, returns in $O(\log_w |S|)$ time the largest integer in S that is no larger than k. The construction time of the data structure is $O(|S|w^{1/2})$ and the size of the data structure is $O(|S|w^{1/6})$.*

For each leaf u of T_C, let A_u be the set of critical ancestors of u and let S_u be the set of scales of the vertices in A_u. Then S_u is a subset of $0, 1, \dots, N$. Since the length of a root-leaf path in T_C is at most N, there are $O(\log N)$ shortcuts for each leaf of T_C. So $|S_u| = O(\log N) = O(\log n)$. To find x_i, we create a data structure of Lemma 4.4.4 with $w = \theta(\log n)$ for each S_u and create a hash tabel [35] for each A_u, with each scale $\gamma' \in S_u$ being a key and the pointer to the critical ancestor of u in scale γ' being the corresponding value. For each ladder Q_u, let R_u be the set of scales of the vertices in Q_u. To find $w(\gamma)$ from the ladder in $O(1)$ time we create a hash table for each ladder Q_u with each scale γ'' in R_u being a key and $w(\gamma'')$ in Q_u being the corresponding value.

Now we summarize the data structure Find-Ancestor for T_C as follows:

- The ladder data structure and a hash table for each ladder Q_u
- For each leaf in T_C, keep a hash table for all the shortcuts to its critical ancestors
- For each S_u, keep a data structure of Lemma 4.4.4 with $w = \theta(\log n)$
- For each vertex x in T_C, keep a pointer to its ladder
- For each leaf u in T_C, keep a pointer to S_u

Lemma 4.4.5. *Data structure Find-Ancestor has $O(n \log^{7/6} n)$ size and can be computed in $O(n \log^2 n)$ time.*

Proof. The rooted tree T_C has size $O(N) = O(n \log n)$. So the size of all ladders is $O(n \log n)$ since each ladder Q_u only doubles the size of the root-leaf path P_u. The size of the hash tables for all ladders is $O(n \log n)$ since the size of the hash table for each Q_u is $O(|Q_u|)$. The size of shortcuts for all u of G is $O(n \log n)$ because each u has $O(\log N) = O(\log n)$ shortcuts. The size for the hash tables for the shortcuts is also $O(n \log n)$. For each u of G, $|Su| = O(\log n)$ and $\sum_{u \in V(G)} |Su| = O(n \log n)$. From this, the size of the data structure of Lemma 4.4.4 for all S_u is $O(n \log^{7/6} n)$ as the size of the data structure for each $|Su|$ is $O(|Su| \log^{1/6} n)$. The size of other pointers associated with the vertices in T_C is $O(N) = O(n \log n)$. So the size of data structure Find-Ancestor is $O(n \log^{7/6} n)$.

T_C, the ladders and short cuts can be computed in $O(n \log n)$ time. The data structure of Lemma 4.4.4 for all S_u can be computed in $O(n \log^{3/2} n)$ time. The hash table for one ladder Q_u can be computed in $O(|Q_u| \log^2 n)$ time [5] and thus the hash tables for all ladders can be computed in $O(n \log^2 n)$ time. □

Given a vertex u of G and a scale γ the data structure Find-Ancestor finds $w(\gamma)$ as follows:

```
Algorithm 4: Find-Ancestor-Query($u, \gamma$)
  Input: a vertex $u$ in $G$ and a level $\gamma$
  Output: the ancestor $w(\gamma)$ of $u$ in $T_C$ that appears in scale $\gamma$
1 if $\gamma = 0$ then
2     return $u$;
3 else
4     Find the largest $\gamma' \in S_u$ with $\gamma' \leq \gamma$;
5     Find the critical ancestor $w'(\gamma')$ of $u$ from the hash table for $A_u$;
6     Find $w(\gamma)$ from the hash table for $w'(\gamma')$'s ladder;
7     return $w(\gamma)$
8 end
```

Lemma 4.4.6. *Given any vertex u of G and any scale γ such that there is an ancestor $w(\gamma)$ of u in T_C, data structure Find-Ancestor finds the $w(\gamma)$ in $O(1)$ time.*

Proof. Each value in S_u is in $0, 1, \ldots, N$ and can be represented using $\theta(\log n)$ bits. Since $|S_u| = O(\log n)$ and we choose $w = \theta(\log n)$, it takes $O(1)$ time to find the largest $\gamma' \in S_u$ with $\gamma' \leq \gamma$ by Lemma 4.4.4. The critical ancestor $w'(\gamma')$ and $w(\gamma)$ can both be found in $O(1)$ time using hash tables.

Let $k = \mathrm{d}(u) - \mathrm{d}(w(\gamma))$. Vertex $w(\gamma)$ is the kth ancestor of u. If $\gamma = 0$ then $w(\gamma) = u$. Otherwise, for the largest $\gamma' \in S_u$ with $\gamma' < \gamma$, the ancestor $w'(\gamma')$ of u is a critical ancestor x_i of u such that $2^i \leq k < 2^{i+1}$. By Lemma 4.4.2, $w(\gamma)$ is in x_i's ladder. Therefore data structure Find-Ancestor finds $w(r)$ in $O(1)$ time. □

The data structure DO_2 for our $(1+\epsilon)$-approximate distance oracle keeps the following information:

- A 2-approximate distance oracle DO_T of G in Lemma 4.2.3.
- For every scale γ, subgraphs $G(\gamma, j)$ and for each subgraph $G(\gamma, j)$, an oracle $\mathrm{DO}_1(\gamma, j)$ in Theorem 4.3.8 with $\epsilon_0 = \epsilon/c'$, $c' > 0$ is a constant to be specified below.
- For every scale γ and every vertex x appearing in scale γ , the index j of every subgraph $G(\gamma, j)$ that contains x.
- Data structure Find-Ancestor.

Lemma 4.4.7. *Data structure* DO_2 *requires* $O(n \log n(\log^{1/6} n + \log n/\epsilon + f(\epsilon)))$ *space and can be computed in* $O(n \log n(\log^3 n/\epsilon^2 + f(\epsilon)))$ *time.*

Proof. DO_T requires $O(n \log n)$ space. Each $\mathrm{DO}_1(\gamma, j)$ requires $O(n_{\gamma j} \log n_{\gamma j}/\epsilon + n_{\gamma j} f(\epsilon))$ space, where $n_{\gamma j} = |V(G(\gamma, j))|$. Each edge e of G appears in $O(\log n)$ different scales γ and in each scale, e appears in $O(1)$ subgraphs $G(\gamma, j)$. From this, $\sum_{\gamma,j} n_{\gamma j} = O(n \log n)$. For every scale γ, DO_2 keeps every vertex x in scale γ and index j of every subgraph $G(\gamma, j)$ that contains x. This requires $O(n \log n)$ space since each edge of G appears in $O(\log n)$ scales and each x appears in $O(1)$ subgraphs. By Lemma 4.4.5, data structure Find-Ancestor has size $O(n \log^{7/6} n)$. Therefore DO_2 requires space $O(n \log n(\log^{1/6} n + \log n/\epsilon + f(\epsilon)))$.

DS_T can be computed in $O(n \log^3 n)$ time and the sparse neighborhood covers can be computed in $O(n \log^2 n)$ time. The time for computing $\mathrm{DO}_1(\gamma, j)$ for each $G(\gamma, j)$ is $O(n_{\gamma j} \log^3 n_{\gamma j}/\epsilon^2 + f(\epsilon))$. By Lemma 4.4.6, data structure Find-Ancestor can be computed in $O(n \log^2 n)$ time. Therefore, DO_2 can be computed in $O(n \log n(\log^3 n/\epsilon^2 + f(\epsilon))$ time. □

Now we give the query algorithm for distance oracle DO_2 in Algorithm 5.

```
Algorithm 5: DO_2-Query(u, v)
   Input: vertices $u$ and $v$ in $G$
   Output: distance $\tilde{d}(u,v)$ such that $d_G(u,v) \le \tilde{d}(u,v) \le (1+\epsilon)d_G(u,v)$
 1 $\tilde{d}_T(u,v) \leftarrow \mathrm{DO}_T(u,v)$;
 2 if $\tilde{d}_T(u,v) = 0$ then
 3     return 0
 4 else
 5     Find $\gamma$ with $\gamma/2 < \tilde{d}_T(u,v) \le \gamma$;
 6     $x \leftarrow$ Find-Ancestor-Query$(u, \gamma)$;
 7     $y \leftarrow$ Find-Ancestor-Query$(v, \gamma)$;
 8     $\tilde{d}(u,v) \leftarrow \infty$;
 9     for $G(\gamma, j)$ that contains both $x$ and $y$ do
10         $\tilde{d}(u,v) \leftarrow \min\{\tilde{d}(u,v), \mathrm{DO}_1(\gamma,j)(u,v)\}$
11     end
12     return $\tilde{d}(u,v)$
13 end
```

Lemma 4.4.8. *For any vertices u and v in G, DO_2 returns an approximate distance $\tilde{d}(u,v)$ in $O(1)$ time such that $d_G(u,v) \le \tilde{d}(u,v) \le (1+\epsilon)d_G(u,v)$.*

Proof. By Lemma 4.2.3, oracle DO_T gives $\tilde{d}_T(u,v)$ with $d_G(u,v) \le \tilde{d}_T(u,v) \le 2d_G(u,v)$ in $O(1)$ time. If $\tilde{d}_T(u,v) = 0$, $d_G(u,v) = 0$ and the correct result is returned in $O(1)$ time. Otherwise, given $\tilde{d}_T(u,v)$, a scale γ with $\gamma/2 < \tilde{d}_T(u,v) \le \gamma$ can be found by computing the most significant bit of $\left\lceil \tilde{d}_T(u,v) \right\rceil$. In the word RAM model with unit costs for basic operations, this can be computed in $O(1)$ time[36]. Notice that each of u and v is covered by a vertex in scale γ and let x and y be the vertices in scale γ covering u and v, respectively. By Lemma 4.4.6, x and y can be computed in $O(1)$ time respectively. By Lemma 4.2.5, there is a $G(\gamma, j)$ that contains x and every w with $d_{G_\gamma}(x,w) \le \gamma$ and $d(G(\gamma,j)) = O(\gamma) =$

$O(d_G(u,v))$. Therefore there exists a constant $c_1 > 0$ (96 would do) such that $d(G(\gamma,j)) \leq c_1 d_G(u,v)$. It is easy to see that $\mathrm{DO}_1(\gamma,j)(u,v)$ returns a minimum distance among all oracles at this scale containing x,y and would be returned as $\tilde{d}(u,v)$. From Lemma 4.2.5, there are $O(1)$ graphs $G(\gamma,j)$ containing x and y. From this and Theorem 4.3.8, it takes $O(1)$ time to compute $\tilde{d}(u,v)$.

By oracle $\mathrm{DO}_1(\gamma,j)$, we get a distance $\tilde{d}_0(x,y)$ with $d_{G(\gamma,j)}(x,y) \leq \tilde{d}_0(x,y) \leq d_{G(\gamma,j)}(x,y) + 7\epsilon_0 d(G(\gamma,j))$. Since $G(\gamma,j)$ is a subgraph obtained from G with every edge e with $l(e) < \gamma/n^2$ contracted, $d_{G(\gamma,j)}(u,v) \leq d_G(u,v)$. Let L be the largest sum of the lengths of the contracted edges in any path in G. Then $d_G(u,v) \leq d_{G(\gamma,j)}(u,v) + L$ and $L < \gamma/n \leq \frac{4}{5}\epsilon d_G(u,v)$, from $\gamma < 2\tilde{d}_T(u,v) \leq 4d_G(u,v)$ and $\epsilon > 5/n$. Let $\tilde{d}(u,v) = \tilde{d}_0(u,v) + \gamma/n$. Then

$$\begin{aligned} d_G(u,v) &\leq \tilde{d}(u,v) \leq d_{G(\gamma,j)}(x,y) + 7\epsilon_0 d(G(\gamma,j)) + \gamma/n \\ &\leq d_G(u,v) + 7c_1\frac{\epsilon}{c'}d_G(u,v) + \frac{4}{5}\epsilon d_G(u,v). \end{aligned}$$

By choosing $c' = 35c_1$, we have $d_G(u,v) \leq \tilde{d}_G(u,v) \leq (1+\epsilon)d_G(u,v)$. □

From Lemmas 4.4.7 and 4.4.8, we get Theorem 4.1.1 which is restated below.

Theorem 4.1.1. *For $\epsilon > 0$, there is a $(1+\epsilon)$-approximate distance oracle for G with $O(1)$ query time, $O(n\log n(\log^{1/6} n + \log n/\epsilon + f(\epsilon)))$ size and $O(n\log n(\log^3 n/\epsilon^2 + f(\epsilon)))$ preprocessing time.*

Using the oracle in Theorem 4.3.9 instead of DO_1, we get Theorem 4.1.2.

4.5 Conclusions

In this Chapter, we discussed how to achieve constant query time independent of ϵ while remaining nearly linear (in n) space and preprocessing time for $(1+\epsilon)$-approximate distance oracle for planar graphs. We showed how to put different tools to work properly to support constant query time, especially how to locate a vertex after contraction in constant time, which was not addressed in previous works. It is open whether there is a $(1+\epsilon)$-approximate distance oracle with $O(1)$ query time and size nearly linear in n for weighted directed planar graphs. Experimental studies for fast query time distance oracles are worth investigating.

Chapter 5

Conclusions and Future Works

In this thesis, we studied both exact distance oracles and approximate distance oracles for planar graphs. In Chapter 3, we showed how to improve the construction time of branch-decomposition for planar graphs and therefore improve the preprocessing time of branch-decomposition based exact distance oracles for planar graphs. And as a direct result, we improved the preprocessing time of the branch-decomposition based distance oracle in [65]. The practical efficiency of the algorithms in this chapter heavily depends on the efficiency of computing the minimum face separating cycles. When the number of faces that need to be separated is small, it may be efficient in practice to compute the face separating cycles using a straightforward approach. Also, the result in this chapter is randomised and can be made deterministic with an additional $log^3 n$ factor in the running time. For graphs where shortest paths are unique, this additional factor can be removed. For graphs with small branchwidth, it is open whether the branch-decompositions techniques and Voronoi diagram techniques can be combined to achieve better exact distance oracles for planar graph.

In Chapter 4, we gave the first $(1 + \epsilon)$-approximate distance oracle with constant query time independent of ϵ and nearly linear (in n) size and preprocessing time. This distance oracle has two technical parts, one is to construct an additive stretch distance oracle with $O(1)$ query time and the other is to convert the first part into a $(1+\epsilon)$-approximate distance oracle while maintaining constant query time after performing edge contraction. A main drawback in the first part is that the dependency on $1/\epsilon$ is exponential for the preprocessing time and oracle size. This has been solved recently by Chan and Skrepetos [22]. They replace the exponential dependency on $1/\epsilon$ on the preprocessing time and space with a polynomial one based on the recent breakthroughs on Voronoi diagram [20]. Gawrychowsk et al. [37] improved the Voronoi diagram techniques in [20] so it is worth investigating whether the improvements can be applied on the result in [22]. Also it is worth investigating whether Voronoi diagram and improved point location structure in [38] for exact distance oracles can be applied to approximate distance oracles.

Bibliography

[1] Ittai Abraham, Shiri Chechik, and Cyril Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1199–1218. ACM, 2012.

[2] Ittai Abraham and Cyril Gavoille. On approximate distance labels and routing schemes with affine stretch. In *International Symposium on Distributed Computing*, pages 404–415. Springer, 2011.

[3] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

[4] Alok Aggarwal and Maria Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2):3–23, 1990.

[5] Noga Alon and Moni Naor. Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[6] Srinivasa Arikati, Danny Z Chen, L Paul Chew, Gautam Das, Michiel Smid, and Christos D Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *Algorithms—ESA'96*, pages 514–528. Springer, 1996.

[7] Giorgio Ausiello, Giuseppe F Italiano, Alberto Marchetti Spaccamela, and Umberto Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–638, 1991.

[8] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998.

[9] Baruch Awerbuch and David Peleg. Sparse partitions. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 503–513. IEEE, 1990.

[10] Maxim Babenko, Andrew V Goldberg, Anupam Gupta, and Viswanath Nagarajan. Algorithms for hub label optimization. *ACM Transactions on Algorithms (TALG)*, 13(1):16, 2016.

[11] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms*, 62(2):74–92, 2007.

[12] Michael A Bender and Martın Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[13] Zhengbing Bian and Qian-Ping Gu. Computing branch decomposition of large planar graphs. In *International Workshop on Experimental and Efficient Algorithms*, pages 87–100. Springer, 2008.

[14] Zhengbing Bian, Qian-Ping Gu, Marjan Marzban, Hisao Tamaki, and Yumi Yoshitake. Empirical study on branchwidth and branch decomposition of planar graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 152–165. Society for Industrial and Applied Mathematics, 2008.

[15] Zhengbing Bian, Qian-Ping Gu, and Mingzhe Zhu. Practical algorithms for branch-decompositions of planar graphs. *Discrete Applied Mathematics*, 199:156–171, 2016.

[16] Hans L Bodlaender, Alexander Grigoriev, and Arie MCA Koster. Treewidth lower bounds with brambles. *Algorithmica*, 51(1):81–98, 2008.

[17] Glencora Borradaile, David Eppstein, Amir Nayyeri, and Christian Wulff-Nilsen. All-pairs minimum cuts in near-linear time for surface-embedded graphs. In *International Symposium on Computational Geometry (SoCG 2016) Symposium on Computational Geometry*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016.

[18] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. Min st-cut oracle for planar graphs with near-linear preprocessing time. *ACM Transactions on Algorithms (TALG)*, 11(3):16, 2015.

[19] Costas Busch, Ryan LaFortune, and Srikanta Tirthapura. Improved sparse covers for graphs excluding a fixed minor. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 61–70. ACM, 2007.

[20] Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2143–2152. SIAM, 2017.

[21] Sergio Cabello. Subquadratic algorithms for the diameter and the sum of pairwise distances in planar graphs. *ACM Transactions on Algorithms (TALG)*, 15(2):21, 2018.

[22] Timothy M Chan and Dimitrios Skrepetos. Faster approximate diameter and distance oracles in planar graphs. In *25th Annual European Symposium on Algorithms (ESA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[23] Shiva Chaudhuri and Christos D Zaroliagis. Shortest paths in digraphs of small treewidth. part i: Sequential algorithms. *Algorithmica*, 27(3-4):212–226, 2000.

[24] Shiri Chechik. Approximate distance oracles with constant query time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 2014.

[25] Danny Z Chen and Jinhui Xu. Shortest path queries in planar graphs. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 469–478. ACM, 2000.

[26] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[27] Vincent Cohen-Addad, Søren Dahlgaard, and Christian Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. In *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*, pages 962–973. IEEE, 2017.

[28] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Robust exact distance queries on massive networks. *Microsoft Research, USA, Tech. Rep*, 2, 2014.

[29] Erik D Demaine, Fedor V Fomin, Mohammadtaghi Hajiaghayi, and Dimitrios M Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and h-minor-free graphs. *Journal of the ACM (JACM)*, 52(6):866–893, 2005.

[30] Erik D Demaine and MohammadTaghi Hajiaghayi. Graphs excluding a fixed minor have grids as large as treewidth, with combinatorial and algorithmic applications through bidimensionality. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 682–689. Society for Industrial and Applied Mathematics, 2005.

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

[32] Hristo N Djidjev. Efficient algorithms for shortest path queries in planar digraphs. In *Graph-Theoretic Concepts in Computer Science*, pages 151–165. Springer, 1996.

[33] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006.

[34] Greg N Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[35] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[36] Michael L Fredman and Dan E Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.

[37] Pawel Gawrychowski, Haim Kaplan, Shay Mozes, Micha Sharir, and Oren Weimann. Voronoi diagrams on planar graphs, and computing the diameter in deterministic õ (n 5/3) time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 495–514. SIAM, 2018.

[38] Pawel Gawrychowski, Shay Mozes, Oren Weimann, and Christian Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 515–529. SIAM, 2018.

[39] Alexander Grigoriev. Tree-width and large grid minors in planar graphs. *Discrete Mathematics and Theoretical Computer Science*, 13(1):13, 2011.

[40] Qian-Ping Gu and Hisao Tamaki. Constant-factor approximations of branch-decomposition and largest grid minor of planar graphs in $\mathcal{O}(n^{1+\epsilon})$ time. *Theoretical Computer Science*, 412(32):4100–4109, 2011.

[41] Qian-Ping Gu and Hisao Tamaki. Improved bounds on the planar branchwidth with respect to the largest grid minor size. *Algorithmica*, 64(3):416–453, 2012.

[42] Qian-Ping Gu and Gengchun Xu. Near-linear time constant-factor approximation algorithm for branch-decomposition of planar graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 238–249. Springer, 2014.

[43] Qian-Ping Gu and Gengchun Xu. Constant query time $(1+\epsilon)$-approximate distance oracle for planar graphs. In *International Symposium on Algorithms and Computation*, pages 625–636. Springer, 2015.

[44] Qian-Ping Gu and Gengchun Xu. Constant query time $(1+\epsilon)$-approximate distance oracle for planar graphs. *Theoretical Computer Science*, 761:78–88, 2019.

[45] Qian-Ping Gu and Gengchun Xu. Near-linear time constant-factor approximation algorithm for branch-decomposition of planar graphs. *Discrete Applied Mathematics*, 257:186–205, 2019.

[46] Qian-Ping Gu and Gengchun Xu. Near-linear time constant-factor approximation algorithm for branch-decomposition of planar graphs. *arXiv preprint arXiv:1407.6761v2*, March 2015.

[47] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

[48] Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *journal of computer and system sciences*, 55(1):3–23, 1997.

[49] Illya V Hicks. Planar branch decompositions i: The ratcatcher. *INFORMS Journal on Computing*, 17(4):402–412, 2005.

[50] Illya V Hicks. Planar branch decompositions ii: The cycle method. *INFORMS Journal on Computing*, 17(4):413–421, 2005.

[51] Ken ichi Kawarabayashi, Philip N. Klein, and Christian Sommer. Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, pages 135–146, 2011.

[52] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[53] Frank Kammer and Torsten Tholey. Approximate tree decompositions of planar graphs in linear time. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms*, pages 683–698. Society for Industrial and Applied Mathematics, 2012.

[54] Frank Kammer and Torsten Tholey. Approximate tree decompositions of planar graphs in linear time. *arXiv preprint arXiv:1104.2275v2*, 2013.

[55] Frank Kammer and Torsten Tholey. Approximate tree decompositions of planar graphs in linear time. *Theoretical Computer Science*, 645:60–90, 2016.

[56] Frank Kammer and Torsten Tholey. Approximate tree decompositions of planar graphs in linear time. *arXiv preprint arXiv:1104.2275v3*, July 2015.

[57] Ken-ichi Kawarabayashi, Christian Sommer, and Mikkel Thorup. More compact oracles for approximate distances in undirected planar graphs. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 550–563. Society for Industrial and Applied Mathematics, 2013.

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

[59] Philip Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 820–827. Society for Industrial and Applied Mathematics, 2002.

[60] Philip N Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155. Society for Industrial and Applied Mathematics, 2005.

[61] Philip N Klein, Shay Mozes, and Christian Sommer. Structured recursive separator decompositions for planar graphs in linear time. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 505–514. ACM, 2013.

[62] Philip N Klein and Sairam Subramanian. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica*, 22(3):235–249, 1998.

[63] Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[64] Gary L Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and system Sciences*, 32(3):265–279, 1986.

[65] Shay Mozes and Christian Sommer. Exact distance oracles for planar graphs. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 209–222. Society for Industrial and Applied Mathematics, 2012.

[66] Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $\mathcal{O}(n \log^2 n/ \log \log n)$ time. In *Algorithms–ESA 2010*, pages 206–217. Springer, 2010.

[67] Yahav Nussbaum. Improved distance queries in planar graphs. In *Workshop on Algorithms and Data Structures*, pages 642–653. Springer, 2011.

[68] Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup-zwick bound. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 815–823. IEEE, 2010.

[69] John H Reif. Minimum s-t cut of a planar undirected network in o(n\logˆ2(n)) time. *SIAM Journal on Computing*, 12(1):71–81, 1983.

[70] Neil Robertson, Paul Seymour, and Robin Thomas. Quickly excluding a planar graph. *Journal of Combinatorial Theory, Series B*, 62(2):323–348, 1994.

[71] Neil Robertson and Paul D Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[72] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[73] Neil Robertson and Paul D Seymour. Graph minors. x. obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991.

[74] J Cole Smith, Elif Ulusal, and Illya V Hicks. A combinatorial optimization algorithm for solving the branchwidth problem. *Computational Optimization and Applications*, 51(3):1211–1229, 2012.

[75] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):45, 2014.

[76] Hisao Tamaki. A linear time heuristic for the branch-decomposition of planar graphs. In *European Symposium on Algorithms*, pages 765–775. Springer, 2003.

[77] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.

[78] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM (JACM)*, 51(6):993–1024, 2004.

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

[80] Oren Weimann and Raphael Yuster. Approximating the diameter of planar graphs in near linear time. *ACM Transactions on Algorithms (TALG)*, 12(1):12, 2016.

[81] Christian Wulff-Nilsen. *Algorithms for planar graphs and graphs in metric spaces.* PhD thesis, PhD thesis, University of Copenhagen, 2010.

[82] Christian Wulff-Nilsen. Approximate distance oracles with improved preprocessing time. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 202–208. Society for Industrial and Applied Mathematics, 2012.

[83] Christian Wulff-Nilsen. Approximate distance oracles for planar graphs with improved query time-space tradeoff. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 351–362. SIAM, 2016.