Computational Study on a Branch Decomposition Based Exact Distance Oracle for Planar Graphs

by

Xinjing Wei

B.Sc, Dalhousie University, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Computing Science

in the
School of Computing Science
Faculty of Applied Science

SIMON FRASER UNIVERSITY
Fall 2017

Approval

Name:	**Xinjing Wei**
Degree:	**Master of Computing Science**
Title:	***Computational Study on a Branch Decomposition Based Exact Distance Oracle for Planar Graphs***
Examining Committee:	**Chair:** Ke Wang Professor

Qianping Gu
Senior Supervisor
Professor _____

Andrei Bulatov
Supervisor
Professor _____

Jiangchuan Liu
Internal Examiner
Professor
School of Computing Science
Simon Fraser University _____

Date Defended: October 19, 2017

Abstract

We present a simple exact distance oracle for the point-to-point shortest distance problem in planar graphs. Given an edge weighted planar graph G of n vertices, we decompose G into subgraphs by a branch-decomposition of G, compute the shortest distances between each vertex in a subgraph and the vertices in the boundary of the subgraph, and keep the shortest distances in the oracle. Let $bw(G)$ be the branchwidth of G. Our oracle has $O(bw(G))$ query time, $O(bw(G)n \log n)$ size and $O(n^2 \log n)$ pre-processing time. Computational studies show that our oracle is much faster than Dijkstra's algorithm for answering point-to-point shortest distance queries for several classes of planar graphs.

Keywords: shortest distance, planar graphs, exact distance oracle, branch decomposition

Acknowledgements

There are many people to whom I owe great gratitude for completing this thesis. I would first like to express my gratitude to my senior supervisor, professor Qianping Gu, for the useful comments, patient guidance, and generous financial support, without which my life in Vancouver would be totally different. Professor Gu introduced me into the world of algorithmic graph theory. I always enjoy benefiting from his sharp comments and scientific ideas during our meetings. I am also grateful for professor Gu's help on revising this thesis. I know this must not have been easy, but professor Gu still managed to give me many insightful comments on my thesis so that I could finally finish it.

I am obliged to professor Andrei Bulatov for being my supervisor during my master's study. I would like to thank professor Jiangchuan Liu for being my examiner. My gratitude also goes to professor Ke Wang for taking the time to chair my thesis defence. I am grateful to the help desk for they support during my research experiments.

I am especially grateful to my wife, whose unyielding love have encouraged me and helped me with my research even though she has to sacrifice much of her time on taking care of me. And many thanks to my yet-to-be-born baby, he has always been my inspiration and motivation.

Last, I leave the deepest and warmest part of my heart for my beloved parents, who gave birth to me, fed me, enlightened me and educated me with their unconditional support and continuous love. Without them, everything is impossible.

Table of Contents

List of Tables

List of Figures

1. Introduction

The problem of computing the shortest paths/distances between vertices in graphs is one of the most well-known and well-studied problems in graph theory with many applications. There are many variants of the problem, and thus many practical algorithms are developed to solve them. One of the variants, and also the focus of this thesis, is the point-to-point shortest path/distance problem: Given a graph G and two vertices s (called the source) and t (called the destination), find a shortest path/distance from s to t in G. The point-to-point shortest path/distance problem is the foundation of other shortest path/distance problems and has many applications. One of the most commonly used applications is route-finding in real road networks (which share many properties with planar graphs). Take Google Maps as an example, it asks you to choose a starting location and a destination, and then outputs a route from the starting location to the destination that is the "shortest" with respect to either travel time or travel distance. In computer networks, it is common that a "shortest" route between two computers is required in order to build up a connection between the two computers. Whether the application is in a map system or a computer network, the larger the graph is, the more important the efficiency of answering a shortest path/distance query is: in applications like Geographic Information System (GIS), a delay of a few minutes may not be tolerable. Classic algorithms like Dijkstra's algorithm [13], however, may not be efficient for new applications that require an answer for a shortest path/distance query in a large graph in a very short time. The problem with algorithms like Dijkstra's algorithm is that they use the raw information of the graph only and compute everything on the graph itself. Distance oracles, on the other hand, is an approach to address this new challenge.

A distance oracle (sometimes also called an index or a labeling scheme) is a data structure that is precomputed and stores some information that helps computing the shortest distance to answer a distance query [39]. A distance oracle can be classified into static oracles and dynamic ones. A static oracle is mainly evaluated by the following parameters: (1) the time used for the oracle to answer a distance query (query time), (2) the memory size used by the oracle (oracle size), and (3) the time used for creating the oracle (pre-processing time). In addition to the criteria above, a dynamic oracle is also evaluated by the update time: the time to update the oracle when there is a dynamic change in the input graph.

The oracle can be either centralized or distributed, that is, an oracle can be entirely stored in a server (central node), or be stored separately in each node in a network.

In this thesis, we consider the static oracles. We propose an oracle for the point-to-point distance problem in edge weighted planar graphs. The oracle itself can be either centralized or distributed, but the pre-processing is performed in a centralized way, a more detailed description of the oracle will be given in Section 3.

1.1 Related Work

There are two extreme methods of solving point-to-point exact shortest path/distance problems: one is by classical shortest distance algorithms like Dijkstra algorithm [13] and Bellman-Ford algorithm [5, 17]. These algorithms do not use a distance oracle, instead, they compute the shortest distance completely on the graph itself and on the fly. Another method is to use a 2-dimensional distance array to store the pre-computed all-pairs distances in the graph. When a shortest distance query comes in, the program can then just look up the distance array and return the desired answer. Let G be an edge weighted graph with n vertices and m edges. Dijkstra's algorithm takes $O(m + n \log n)$ time and $O(m)$ memory space to answer a point-to-point shortest distance query for G without negative edge weight. Bellman-Ford algorithm takes $O(mn)$ time and $O(m)$ memory space to answer the query but can handle the problem for graphs with negative edge weights (without negative cycles) [37]. The oracle of 2-dimensional distance array takes $O(1)$ time (query time) to answer the query but requires $O(n^2)$ memory space (oracle size). Further more, it takes $O(n(m + \log n))$ time (pre-processing time) to create the oracle. There is a trade-off between the query time and oracle size. The area that this thesis focuses on is to study and develop oracles that have better trade-offs between the two extreme examples.

1.1.1 Dijkstra Algorithm

Dijkstra algorithm and its variants are so far the most well-known one-to-all exact shortest distance algorithm without the use of a distance oracle. The algorithm starts with a set initially containing only the source vertex, and continues adding vertices with the shortest distance from the source vertex to the set, until the destination vertex is found or there is no vertex that the source vertex can reach [13]. An advantage of this algorithm is that in the process, it computes the shortest distances from the source to all the vertices in the set, thus Dijkstra algorithm is often used as a tool to compute one-to-all shortest distances in a graph. The algorithm is so popular that it is often used as a benchmark in evaluating the efficiency of an oracle. It is used in this thesis as a comparison object as well.

1.1.2 Distance Oracle in General graphs

A distance oracle is a data structure that stores some pre-computed information and answers a shortest distance query efficiently. An exact distance oracle is a distance oracle that gives the shortest distance $d_G(s,t)$ from vertex s to vertex t in graph G. An approximate distance oracle, together with an (α,β) approximation stretch (α is called a multiplicative stretch and β is called an additive stretch), is a distance oracle that gives a distance $\tilde{d}_G(s,t)$ with $d_G(s,t) \leq \tilde{d}_G(s,t) \leq \alpha d_G(s,t) + \beta$. An approximation oracle with stretch $(\alpha, 0)$ is called an α-approximate distance oracle.

In 2003, Cohen, Halperin, Kaplan, and Zwick [8] introduced a distributed exact distance oracle for general graphs using 2-hop cover. Assume the 2-hop cover for a graph G is H, then their oracle size is $O(|H|)$, and has an average query time of $O(\frac{|H|}{n})$. They also pointed out that the size of the oracle is unpredictable, but using a heuristic in the paper, an almost optimal 2-hop cover can be found (a $(\log n)$-approximation algorithm), thus bounding the size of the portal sets to $O(\log n)$ for each vertex [8]. In 2013, Babenko, Gledberg, Gupta, and Nagarajan [3] further improved the 2-hop cover algorithm that gives an $\log n$ approximation on the optimal maximum size of the portal sets, thus reducing the worst-case query time of the oracle up to a log factor. In 2014, Jiang, Fu, Wong, and Xu developed a distributed exact distance oracle for unweighted directed graphs using 2-hop labeling, and provided a oracle size bound of $O(hn)$ on scale-free networks with $O(n \log M \cdot (\frac{n}{M} + \log n))$ pre-processing time and $O(n \cdot \frac{\log n}{M} \cdot \frac{n}{B})$ query time, where h is a small constant, M is the memory size, and B is the disk block size [27].

Sommer pointed out in [39] that Thorup and Zwick in 2005 [41] gave a tight trade-off between approximation ratio and space complexity for general graphs: an oracle of size $O(kn^{1+\frac{1}{k}})$ gives a $(2k-1)$-approximation on the shortest distance and has an $O(k)$ query time for any $k \geq 1$ [41]. He also pointed out that the trade-offs for distance oracles that use embedded information of the graphs is less studied [39].

Many distance oracles have been developed and studied for general graphs, readers may refer to Sommer's survey paper (Section 2) [39] for more details.

1.1.3 Distance Oracle in Planar Graphs

Planar graphs are considered widely in the application of shortest distance queries due to their similarity with real world road networks [39]. Though real road networks are not all planar graphs, they share some properties (small separators, etc.) [15] that can be used as a tool of answering distance queries efficiently. Because of this similarity with the road networks, many researchers came to the realization of the necessity of answering shortest distance queries efficiently in planar graphs.

Theoretical Results

In 1997, Djidjev [14] proved that there exists an exact distance oracle of size $S \in [n, n^2]$ that answers the shortest path/distance query in $O(\frac{n^2}{S})$ time for planar graphs. If $S \in [n^{\frac{4}{3}}, n^{\frac{3}{2}}]$, then there exists an oracle of size S with query time $\tilde{O}(\frac{n}{\sqrt{S}})$ [14]. Djidjev uses r-divisions (a partition of the edge set into $O(\frac{n}{r})$ regions, $R_1, R_2, \cdots$ of size r) to divide a graph G into subgraphs, and uses the set of boundary vertices, denoted as ∂R, (vertices adjacent to edges of different regions) as portal sets (vertices in which can be used to compute/estimate shortest paths). The algorithm then computes the pairwise distances between portals and stores them in a table with $O(\frac{n^2}{\sqrt{r}})$ space and can answer shortest distance queries in $O(r)$ time. For more details and improvements, readers can refer to [7, 14, 29]. In 2004, Gavoille [18] proved a lower bound of $\Omega(n^{\frac{3}{2}})$ for the oracle size of a bounded degree weighted planar graph for any query time. In 2006, Fakcharoenphol and Rao [16] expended the embedded information in planar graphs into exact distance oracle and came up with an exact distance oracle with $O(n \log^3 n)$ query time and $O(n \log n)$ size that is suitable for machines with limited space and preprocessing time. Wulff-Nilsen proposed an exact distsance oracle with constant query time and $O(\frac{n^2 (\log \log n)^4}{\log n})$ size for weighted directed planar graphs in 2010 [42]. Mozes and Sommer [29] then introduced an exact distance oracle for planar graphs with oracle size $O(S)$ and pre-processing time $\tilde{O}(S)$, and answers shortest distance queries in $\tilde{O}(\frac{n}{\sqrt{S}})$ time, given that $S \in [n \log \log n, n^2]$. They also came up with a linear space exact distance oracle for planar graphs with query time $O(n^{\frac{1}{2}+\epsilon})$ for any $\epsilon > 0$. In 2017, Cohen-Addad, Dahlgaard and Wulff-Nilsen [9] developed an exact distance oracle using r-division that achieves $O(n^{\frac{5}{3}})$ space and $O(\log n)$ query time on planar weighted directed graphs. They also provided a way to construct an exact distance oracle of size $O(S)$ that answers shortest distance oracle queries in $O(\frac{n^{5/2}}{n^{3/2}} \log n)$ time for any $S \geq n^{\frac{3}{2}}$, which improves the previous $O(n^{\frac{1}{4}})$ query time.

In 2004, Thorup [40] proposed a $(1+\epsilon)$-approximation oracle with $O(\log \log(nN) + \frac{1}{\epsilon})$ query time and $O(\frac{n(\log(nN)) \log n}{\epsilon})$ size for planar digraphs with edge weights drawn from $\{0, 1, \cdots, N\}$, and a $(1+\epsilon)$-approximation oracle with $O(\frac{1}{\epsilon})$ query time and $O(\frac{n \log n}{\epsilon})$ size for undirected planar graphs. In 2015 [22], Gu and Xu proposed a $(1+\epsilon)$-approximation oracle with an $O(1)$ query time independent of ϵ and $O(n \log n(\frac{\log n}{\epsilon} + f(\epsilon))$ space for undirected planar graphs, where $f(\epsilon) = 2^{O(1/\epsilon)}$. In 2016, Wulff-Nilsen [44] came up with a $(1+\epsilon)$-approximate distance oracle with $O(\frac{(\log \log n)^3}{\epsilon^2} + \frac{\log \log n \sqrt{\log \log(\log \log n/\epsilon^2)}}{\epsilon^2})$ query time and $O(\frac{n(\log \log n)^2}{\epsilon} + \frac{\log \log n}{\epsilon^2})$ space for undirected planar graphs. This improves the previous best product of query time and space.

Computational Results

Due to the popularity of route planning and GIS, many distance oracles specialized for real road networks were developed in the recent years.

In 2008, Geisberger, Sanders, Schultes, and Delling [19] came up with a hierarchical distance oracle called Contraction Hierarchies (CH). The algorithm recursively contracts vertex with "less importance", constructing layers of contracted graphs. It answers shortest distance queries by searching paths from the most contracted layer to the original graph. CH performs well on practical data (road network of Western Europe) due to its small overhead and fast pre-processing time [19, 39]. An improved version of the oracle, PHAST, was then introduced in 2013 in [10], taking the advantages of the modern CPU architectures and making it suitable for shortest distance queries for continental road networks.

Inspired by the recent research findings on graph partitioning, Delling, Goldberg, Pajor, and Werneck [11] proposed an algorithm called Customizable Route Planning (CRP) in 2011. The algorithm partitions a graph into connected subgraphs with no more than N vertices, together with a set of boundary graphs (induced by boundary vertices and arcs) then uses bidirectional Dijkstra algorithm to construct the oracle. The algorithm was tested on continental road networks and was more than 3000-7000 times faster than the Dijkstra algorithm [11].

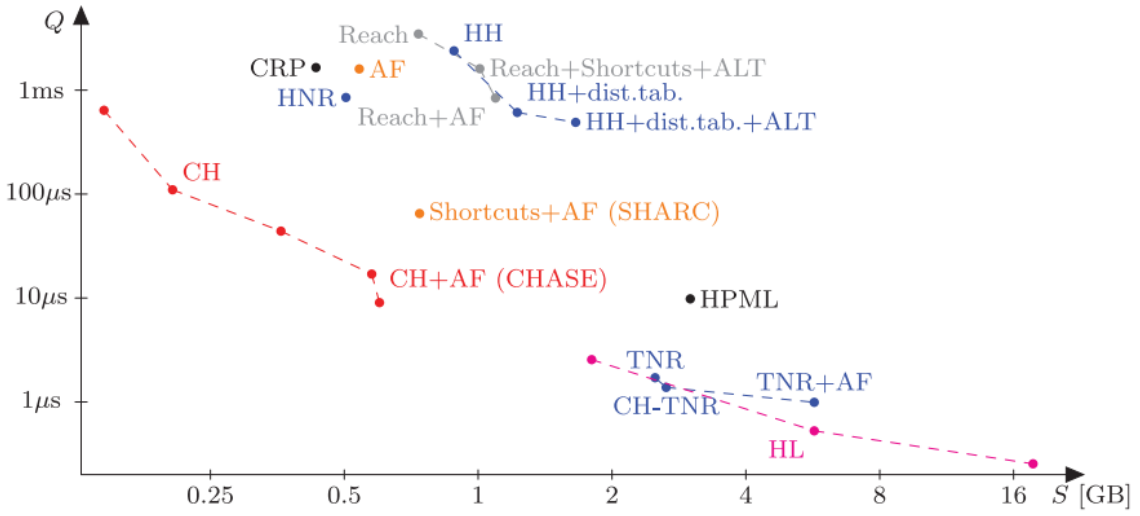


Figure 1.1: Practical results for road planing in real road networks. This figure is extracted from [39]. It shows the trade-offs between space (S) and query time (Q) for many distance oracles developed for real world road networks.

Other distance oracles that performs well on practical data like road networks includes TreeMap [45], Arc Flags (AF) [28], Transit-Node Routing [2], Hub Labels [12], and etc.. Readers may refer to Sommer's survey paper [39] for more detailed descriptions of the above oracles. Most of such distance oracles are tuned using techniques specialized for specific networks like road networks, etc., and the algorithms/heuristics rely on the topological/spatial properties of the networks heavily. Figure 1.1 shows the computational results (space-query-time trade-offs) on some of the above algorithm.

1.1.4 Graph Decomposition and Its Use in Distance Oracles

Graph decomposition is a commonly used approach for developing distance oracles. Common decomposition techniques includes tree decomposition [33], path decomposition [32, 40] and branch decomposition [34]. For any particular decomposition technique, a graph G is decomposed into subgraphs using some kind of graph separator (e.g. a vertex cut set, an edge set, etc.). Each separator cuts the induced graph (a subgraph of G) into subgraphs. A decomposition of a graph G is often represented by a tree like structure called decomposition tree, together with a function τ that maps either vertices or edges of the graph to nodes in the decomposition tree. Recent studies use these decomposition techniques and separators to design distance oracles (whether exact or approximate). The reason is that if some separators separate the source and the destination into two different subgraphs, then the shortest path from the source to the destination has to intersect with those separators, thus reducing the number of vertices needed to compute the shortest distance. Thorup [40], Wulff-Nilsen [43], and Gu and Xu [22] proposed approximate distance oracles based on graph decompositions. Xiang [45] introduced a distributed exact distance oracle using tree decomposition of a graph. It creates a separation tree (whose nodes represents vertex sets that separates the graph) T by recursively removing centroids in the tree decomposition and labeling the nodes using Breadth First Search (BFS). T is then transformed into a binary tree T' by adding dummy nodes to it, and the labels are updated accordingly. The construction takes $O((tw(G))^2 \cdot n \cdot \log^2 n + tw(G) \cdot m \cdot \log n)$. The oracle answers the shortest distance query in $O(tw(G))$ time, where $tw(G)$ is the treewidth (defined in Section 2) of G.

1.2 Contribution of the Thesis

A branch-decomposition is a system of using vertex cut sets as separators to decompose a graph G into subgraphs (a formal definition of branch-decomposition is given in Chapter 2). It is a useful technique for many problems in graph theory, the shortest path problem is one of such problems.

In this thesis, we present a simple exact distance oracle for planar graphs G based on branch decompositions. The oracle has $O(bw(G))$ query time, $O(bw(G)n \log n)$ size and $O(n^2 \log n)$ pre-processing time, where $bw(G)$ is the branchwidth of G. The construction of the oracle starts with computing a branch decomposition of the input planar graph G from which a branch decomposition tree T_B is built. Then a perfect "virtual" rooted binary tree T_V is constructed by transforming T_B into a rooted binary tree T_L (each tree node in T_L corresponds to a tree edge in T_B) and adding dummy nodes. The algorithm then computes the cut set associated with each node in T_L (a tree edge in T_B) and the shortest distances between each vertex in a leaf node and vertices in the cut set associated with each ancestor in T_V, and keeps the shortest distances in the oracle.

In the query phase, assume that the source is s and the destination is t, the cut set C that separates s and t is found using T_V, the shortest distance between s and t is then defined by $d_G(s,t) = \min_{v \in C}\{d_G(s,v) + d_G(v,t)\}$, where $d_G(v,u)$ denotes the shortest distance between u and v in G.

Due to its close relation with the branch decomposition of G, the oracle has a small query time and oracle size for graphs with small branchwidth. Unlike the exact distance oracles that are specialized for real world road networks, our oracle is efficient for a wide range of undirected planar graphs, and is easy to implement from algorithm engineering point of view due to the simplicity of the data structure.

Computational study shows that our oracle performs well on planar graphs, it beats both Dijkstra's algorithm and Bi-directional Dijkstra's algorithm by a factor of at least 30 for planar graphs with 5000+ edges.

1.3 Thesis Structure

In this thesis, we will give the preliminaries of the thesis in Chapter 2 and introduce the branch-decomposition based exact oracle for the point-to-point distance problem in arbitrary planar graphs in Chapter 3. We then present the computational study results for the branch decomposition based oracle and Dijkstra's algorithm, followed by a discussion of the results in Chapter 4. Finally, we conclude the thesis in Chapter 5.

2. Preliminary

We denote $G = (V, E)$ as a graph with V as the vertex set and E as the edge set (i.e., a set of pairs of elements in V). We let $|V| = n$ and $|E| = m$. We denote by $V(G)$ and $E(G)$ as the vertex set and the edge set of a graph G, respectively. In the rest of this thesis, we consider G as a simple connected graph (no multi-edge or loop). G is weighted if each edge e is associated with a real number, denoted as $w(e)$, as the edge weight, otherwise unweighted. For an un-weighted graph, we assume each edge has weight one. If the pairs in E is ordered, we say that the graph G is directed, otherwise, the graph G is undirected. We will use graph for undirected graph and digraph for directed graph in the rest of the thesis. G is said to be *planar* if there is a drawing of G onto a plane with no two edges crossing each other. One interesting property of a planar graph is that $m = O(n)$.

A path P in G is a sequence of edges $e_1e_2 \cdots e_k$ of G, where $e_i = (v_i, v_{i+1})$ for $i = 1, 2, \cdots, k$ such that each vertex of G appears in P at most once. When $v_1 = v_{k+1}$, P is called a cycle. The length of a path P, denoted by $l(P)$, is the sum of the weights of all edges in P. Formally, for $P = e_1e_2 \cdots e_k$, $l(P) = \Sigma_{i=1}^{k} w(e_i)$. A *shortest path* from a vertex s to a vertex t in G is a path from s to t with the minimum length. The length of a shortest path from s to t in G is the *shortest distance* from s to t in G, denoted by $d_G(s, t)$.

A subgraph $H = (V', E')$ of G is a graph such that $V' \subseteq V$ and $E' \subseteq E$, and V' contains all endpoints of the edges in E'. A connected component $Q = (V_Q, E_Q)$ of G is a subgraph of G such that every pair of vertices in V_Q is connected by a path, and no vertex in V_Q is connected to any vertex in $V \setminus V_Q$. A *vertex cut set* $C \subseteq V$ of G is a set of vertices of which the removal (together with their incident edges) will decompose G into at least two components. For a graph G and a subset $A \subseteq E(G)$ of edges, we denote the complement of A, $E(G) \setminus A$, by $\bar{A}$. A *separation* of graph G is a pair $(A, \bar{A})$ of subsets of $E(G)$. Notice that for each separation $(A, \bar{A})$, there is a vertex cut set $C = V(A) \cap V(\bar{A})$ associated with it. The order of a separation $(A, \bar{A})$ is $|V(A) \cap V(\bar{A})| = |C|$.

We will use node for vertex and link for edge in a tree. In a rooted tree T with root r, node v is a child of node u if (u, v) is a link of T and $d_T(v, r) = d_T(u, r) + 1$; and u is called the parent of v. Node v is a descendant of u if u is on the path between v and r, and $d_T(v, r) > d_T(u, r)$; and u is called an ancestor of v. A node of T is called a leaf if it does not have any child, otherwise an internal node. An internal node u of a binary tree T has at

most two children. The *depth* of a node v in T, denoted by $dep(v)$, is the length of the path from the root to this node. The depth of a tree T is defined by $dep(T) = \max_{v \in V}\{dep(v)\}$. A *perfect binary tree* is a tree where every internal node has exactly two children and all leaf nodes have the same depth. For each vertex u in T, the subgraph induced by each child v of u and v's descendants is a subtree of T. We call one subtree the left subtree of u and the other (if u has two children) the right subtree of u (see Figure 2.1 for an example). For a tree T, we denote by $lf(T)$ the number of leaves in T. For a link $e = (u, v)$ of T with root r, removing e partitions T into two subtrees $T_L(e)$ and $T_R(e)$, one contains the root r and the other does not. If the node v is the left child of u, then we say that the subtree containing r is the right subtree (namely $T_R(e)$) induced by the link e. The other subtree $T_L(e)$, which is rooted at v, is called the left subtree induced by the link e (see Figure 2.2(a) for an example). If v is the right child of u, then $T_L(e)$ is the subtree containing r and $T_R(e)$ is the subtree rooted at v (see Figure 2.2(b) for an example). We denote by $lf_L(e)$ the number of leaves in the left subtree $T_L(e)$ and by $lf_R(e)$ the number of leaves in the right subtree $T_R(e)$ induced by e.

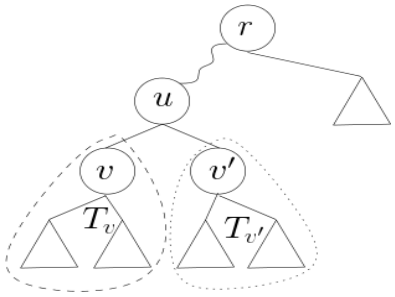
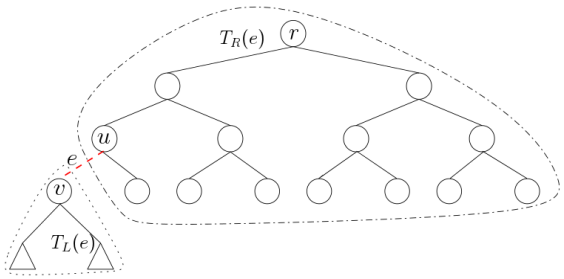


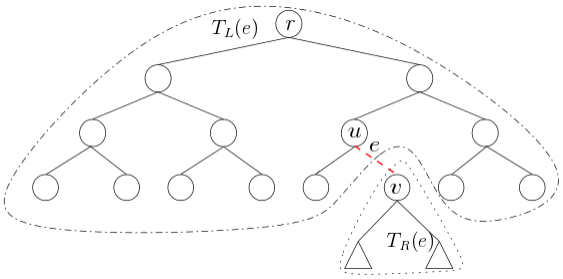
Figure 2.1: An example of the left/right subtrees of a node u in a binary tree rooted at r. Circles represent nodes in the tree, and triangles represent implicit subtrees. In this figure, u has two children v and v'. We say that the binary tree T_v (the dashed region), induced by v and its descendants, is the left subtree of u; and the binary tree $T_{v'}$ (the dotted region), induced by v' and its descendants, is the right subtree of u.

An *inorder search* of a rooted tree is a tree traversal that first visits the left subtree of the root, then the root, then the right subtree of the root, and each subtree is visited recursively. The *inorder search index*, or just the *index*, of a node u is the order (numbered from 1 to n) in which u is visited by an inorder search in the rooted tree. The index of a node u is denoted by index(u). For a pair of nodes u and v in a rooted tree T with root r, a common ancestor of u and v is a vertex w which is on the path from u to r and the path from v to r. The *nearest common ancestor (nca)* is therefore the common ancestor w with highest depth. Notice that if w is the nca of u and v, then one of u and v is in the left subtree of w and the other is in the right subtree of w.

Branch decomposition was first introduced by Robertson and Seymour in 1991 [34]. A *branch decomposition* of a graph G is a pair (T_B, τ), where T_B is a ternary tree with $|E|$



(a) If v is a left child, then the subtree rooted at r is the left subtree $T_L(e)$, the subtree rooted v is the right subtree $T_R(e)$.



(b) If v is a right child, then the subtree rooted at r is the right subtree $T_R(e)$, the subtree rooted v is the left subtree $T_L(e)$.

Figure 2.2: An example for the two cases of left/right subtrees. Circles represent nodes in the tree T, and triangles represent implicit subtrees. The dashed link $e = (u, v)$ is a removed link and r is the root of the tree T.

leaf nodes and τ is a bijection from the edges in G to the leaves in T_B. The removal of any link e in T_B (say the resulting two subtrees are T_1 and T_2) "cuts" G into two subgraphs, one induced by the leaves in T_1, the other induced by the leaves in T_2. Thus each tree link in T_B has a vertex cut set associated with it. We say that the separation $(\tau(T_1), \tau(T_2))$ is induced by the link e. We define the width of a branch decomposition (T_B, τ) to be the largest order of the separations induced by links of T_B. The *branchwidth* of G, denoted by $bw(G)$, is the minimum width of all branch decompositions of G. In the rest of this thesis, we identify a branch decomposition (T_B, τ) with the tree T_B, leaving the bijection implicit and regarding each leaf of T_B as an edge of G. We call a link e in a branch decomposition tree *valid* if $\frac{lf(T_B)}{3} \leq lf_L(e) \leq \frac{2lf(T_B)}{3}$.

For any perfect binary tree T and a leaf node $x \in V(T)$, we define the *forward port set* and *backward port set* as follows:

Definition 1. let v be a node in T with the in-order index $a \cdot 2^i$, where a is some integer, we define v as a level i port.

For each leaf node x of T, let y be the node with the minimum index such that $\text{index}(x) \leq \text{index}(y)$. Assume that y is a level i port and we rename y as y_i. For every $j > i$, let y_j be the level j port with the minimum index such that $\text{index}(x) \leq \text{index}(y_j)$. We call $y_i, y_{i+1}, \cdots$ the forward ports for x. The set F_x containing all the forward ports of x is the forward port set for x. Similarly, let z be the port with the maximum index such that $\text{index}(x) \geq \text{index}(z)$. Assume z is a level p port. For every $q > p$, let z_q be the level q port with the maximum index such that $\text{index}(x) \geq \text{index}(z_q)$. We call $z_p, z_{p+1}, \cdots$ the backward ports for x. The set B_x containing all backward ports of x is the backward port set for x.

Notice that for any node y in F_x, x is in the left subtree of y, and for any node z in B_x, x is in the right subtree of z. Additionally, any ancestor of x is either in F_x or B_x but not in both. These are direct observations of the in-order search on a perfect binary tree.

Throughout the paper, we refer shortest path/distance query problem as the point-to-point shortest distance query problem: Given a pair of vertices s and t in a weighted planar graph G, find the shortest distance $d_G(s,t)$ from s to t in G.

3. Algorithm Design

3.1 Algorithm Description

To construct our oracle, we first compute a branch decomposition T_B of an input planar graph G. An optimal branch-decomposition can be computed in $O(n^3)$ time in worst case and $O(n^2 \log n)$ in average case [20]. A branch-decomposition of width $O(k)$ can be computed in $\min\{O(n \log^3 n \log k), O(nk^2 \log k)\}$ time, where $k = bw(G)$ [21].

As mentioned above, every leaf node in T_B corresponds to an edge in G. For each tree link e of T_B, e is associated with a vertex cut set (the separator) that cuts G into two subgraphs (or regions). For querying the shortest distance of a pair of vertices s and t, one just need to find the cut set that separates s and t into two regions, the shortest path has to go through a vertex in the cut set. Thus to find the shortest distance between s and t using a branch decomposition T_B, one just need to find a cut set in T_B that separates s and t into two regions. Based on the structure of a branch decomposition T_B, the cut set is one of these associated with the three links incident to the nca w of the leaf containing s and the leaf containing t, as shown in Figure 3.1. The shortest distance path between s and t in G will have to intersect with some vertices in at least one of the cut sets associated with e_1, e_2, or e_3 in Figure 3.1. Therefore, if we want to find the shortest path distance from s to t, we will have to consider all three cut sets, thus increasing the query time. To increase the efficiency, we need to use the correct cut set for s and t for computing the distance between s and t, not all of the three cuts. Also, the number of ancestors of a node s is the height of T_B and thus if T_B has a large height (T_B is not balanced) then the oracle has a large size. To address the above problems, we perform the following transformation to get a rooted binary tree T_L, which we will refer to as a logical tree in the rest of the paper, such that for every two vertices s and t in G, the nca of the node containing s and the node containing t in T_L is associated with a cut set which separates s and t in two regions, and the hight of T_L is $O(\log n)$.

Given a branch decomposition tree T_B, we choose a link e that is valid and "virtually remove" it from the tree. Assume that the separation is $(E(G_1), E(G_2))$, where G_1 and G_2 are the two subgraphs induced by e. We then recur the above process on G_1 and G_2, say the valid links chosen in G_1 and G_2 are f and g respectively. We view e, f, and g as logical

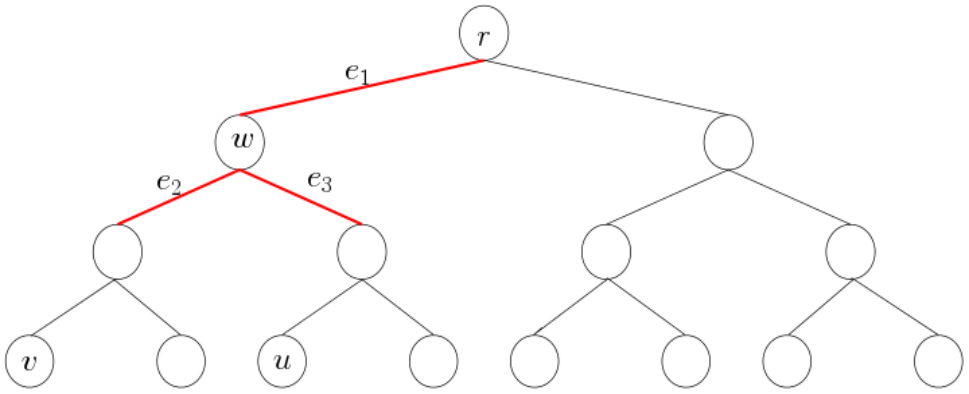


Figure 3.1: In this branch decomposition tree T of G rooted at r, w is the nearest common ancestor of u and v. The three links (bold lines) e_1, e_2, and e_3 are the three potential vertex cut sets that may separate u and v in G.

nodes and connect f and e, and g and e using logical links. We repeat the process until all links are "removed" from T_B. The logical nodes and links form a rooted binary tree, denoted by T_L. Notice that each node in T_L corresponds to a link in T_B. Further more, the order we choose links in T_B to remove in the transformation phase does not affect the theoretical bound of our oracle, as long as the chosen links are valid in every iteration.

To make this process simpler to implement, we first change T_B to a rooted binary tree by first pick up a link $e = (u, v)$ with u, v both internal nodes. Then e is physically removed from the tree and two links (r, u) and (r, v) are added to T_B, where r is a newly added node, and also the root of T_B. The detailed process is described in Algorithm 1 and Algorithm 2.

Algorithm 1 TreeTransformation

Input: G, an input planar graph

1: Find a branch decomposition tree T_B of G
2: Convert T_B to a rooted binary tree rooted at r
3: $Tab \leftarrow$ An empty table with links in T_B as horizontal indexes and L (left side) & R (right side) as vertical indexes.
4: Count $lf_L(e)$ and $lf_R(e)$ for every link e and fill the table Tab by first virtually remove e and then count the number of leaves in each resulting subtree
5: $r_L \leftarrow ModifyDecompTree(T_B, Tab)$
6: **return** T_L, rooted at r_L

We now prove that the cut set associated with the nca of the nodes containing s and t in T_L contains the correct vertex that intersect with the shortest path between s and t in G: at any stage of the transformation, say the tree link that is currently being removed is e, then e separates the graph into two subgraphs, say the two subgraphs are G_1 and G_2, if u resides in G_1 and v resides in G_2, then e contains the cut set that separates u and v. After the transformation, u will be in a node in one subtree of e in T_L and v will be in a node in the other subtree of e in T_L. Thus the nca of the two nodes containing u and v will

Algorithm 2 ModifyDecompTree

Input: r, the root of the current branch decomposition subtree T_r; Tab, the table computed in Algorithm 1; N, the number of leaves in T_r

1: $e_0 = (u, v) \leftarrow ValidLinkSelection(r, N)$ (Algorithm 4)
2: **if** no such e_0 is found **then**
3: **return** NULL
4: **end if**
5: $updateTable(r, Tab, e_0)$ (Algorithm 5)
6: $dir \leftarrow 0$ if v is the right child of u, 1 otherwise
7: $n_r \leftarrow Tab[r][LEFT] + Tab[r][RIGHT]$, the number of leaves in T_r with e_0 removed
8: $n_v \leftarrow Tab[v][LEFT] + Tab[v][RIGHT]$, the number of leaves in T_v
9: $e_1 \leftarrow ModifyDecompTree(r, Tab, n_r)$
10: $e_2 \leftarrow ModifyDecompTree(v, Tab, n_v)$
11: link e_0 & e_1 and e_0 & e_2 with logical links, e_0 is the parent of the two
12: **if** $dir = 0$ **then**
13: e_1 is the left child of e_0 and e_2 is the right child of e_0
14: **else**
15: e_2 is the left child of e_0 and e_1 is the right child of e_0
16: **end if**
17: **return** e_0

be e, hence we can correctly find the cut set that separates them. Notice that because we always select valid links, the resulting tree T_L will be balanced. Thus the height of T_L is $O(\log n)$.

3.1.1 Valid Link Selection

There is, however, one problem remains unsolved: the selection of such a valid link. To address this problem, we count the number of leaf nodes in the induced left and right subtrees for each link, and store the result in a $3 \times O(n)$ table Tab. The first row of Tab is the reference to all the links in T_B, the second and the third rows contain the number of leaves in the left/right subtrees of the corresponding link.

The general idea of valid link selection is to do it in a recursive way, from top to bottom, so that each edge is considered at most once in each selection. In any valid link selection stage of the tree transformation, suppose $e = (u, v)$ (u is the parent of v) is selected in the tree T_r rooted at r that has m leaves (recall that $m = O(n)$ in planar graphs). Link e is then marked as removed, thus we have a separation $(E(T_1), E(T_2))$. Suppose T_1 is the subtree rooted at r and T_2 is the subtree rooted at v. The table Tab is then updated by the following rule: for the subtree T_1, do a search from r and count the number of leaves, stops when reaches a leaf or a removed edge (Algorithm 3); for the subtree T_2, we do a search from v and count the number of leaves. To find a valid child link, we again start from the root to preserve the right/left direction of the links. Suppose we want to select a child in T_1, we first start from r, checking each link incident to r, and recurse on the children of r

until we find one edge that is valid. For more detailed description of the algorithms, please refer to Algorithm 4 and Algorithm 5.

The table update process takes $O(n)$ time and requires $O(n)$ space for the table Tab. To find children links in the worst case, we need to iterate through the table, thus taking $O(n)$ time. Therefore, valid link selection stage takes $O(n)$ time and $O(n)$ space.

Algorithm 3 LeavesCount

Input: r, the root of the current branch decomposition tree

if r is NULL or both children links of r is removed **then**
  return 0
else if r is a leaf node **then**
  return 1
else if one of the child link of r is removed **then**
  return The *LeavesCount* of the other child link
else
  return The sum of the *LeavesCount* for both children links
end if

Algorithm 4 ValidLinkSelection

Input: r, the root of the current branch decomposition tree, Tab, the table, n the total number of leaves in the tree rooted at r, and $e = (u, v)$, the edge being selected

for each child u of r (two at most) **do**
  if (r, u) is valid **then**
    return (r, u)
  else
    return $ValidLinkSelection(u, Tab, Tab[u][LEFT] + Tab[u][RIGHT])$
  end if
end for
return NULL

Algorithm 5 UpdateTable

Input: r, the root of the current tree, Tab, the table, and $e = (u, v)$, the link being removed

Mark e as removed
Suppose the two subtrees are T_1 and T_2, with roots r and v
$LeavesCount(r)$, update Tab accordingly
$LeavesCount(v)$, update Tab accordingly

3.1.2 In-Order Search Index Assignment

After we obtain T_L, the question that remains is to find the nca of two nodes efficiently. Harel and Tarjan [23] proposed an $O(1)$ algorithm that finds the nca of two nodes in 1984. The algorithm, however, requires the tree to be a complete binary tree (for correct node

indexing used in the algorithm), otherwise a complex transformation is required in order to find the nca of two nodes in an arbitrary tree. A detailed algorithm description is given in Section 3.1.3. Recall that by our transformation, T_L is already a rooted binary tree, thus changing T_L into a complete tree is trivial: recall that each node in T_L is a link (associated with a cut set) in T_B. For a leaf node (some link in T_B) u in T_L, suppose the graph induced by the leaves in the subtree T_u of T_B is G_u, then the cut set associated with u cuts G_u into two subgraphs (called atom nodes). Let T_E be the tree obtained from T_L by attaching atom nodes to the corresponding leaf nodes. Let T_V be a perfect binary tree obtained from T_L also, but with the following rules: let the height of T_L be h. For a leaf node u in T_L with depth less than h, create two dummy perfect binary trees (solely for node indexing) T_{u1} and T_{u2} of height $h - dep(u)$ and attach them to u. Suppose the two atom nodes attached to u in T_E are a_1 and a_2, attach a_1 and a_2 to T_{u1} and T_{u2} so that their nca is u (attach one on the rightmost leaf on T_{u1} and other on the leftmost leaf on T_{u2}); for a node u with depth equals h, simply attach the two associated atom nodes. We then add dummy nodes to the leafs so that T_V becomes a perfect binary tree of height $h+1$, which is also complete. After creating T_V, we do an in-order search to assign each node the in-order search index.

Algorithm 6 InorderSearch

Input: v, the current tree node in T_L; i, the last assigned index, initially 0

1: **if** v is NULL **then**
2: **return** i
3: **end if**
4: **if** v has a left child w_1 **then**
5: $i \leftarrow InorderSearch(w_1, i)$
6: **else**
7: Create a new atom node a_1 and attach it to the left of v and assign $i + 1$ to a_1
8: $i \leftarrow i + 2^{h+1-dep(v)}$
9: **end if**
10: Assign the new i to v
11: **if** v has a right child w_2 **then**
12: $i \leftarrow InorderSearch(w_2, i)$
13: **else**
14: $i \leftarrow i + 2^{(h+1)-dep(v)+1} - 1$
15: Create a new atom node a_1 and attach it to the left of v and assign i to a_2
16: **end if**
17: **return** i

One disadvantage of the above algorithm is that we need to physically create and add dummy nodes and dummy links to T_E, thus creating additional memory consumption. The dummy nodes and dummy links are solely for assigning the correct index to the non-dummy nodes in T_E. To further speed up the pre-processing, notice that Algorithm 7 is based on the in-order search index of each node in the perfect binary tree, thus we do not need to physically add dummy nodes and links to T_E, we only need the correct index. Therefore, we

"virtually" add dummy nodes and links to make T_E a perfect binary tree T_V. The process is also trivial: because T_V is a perfect binary tree of height $h+1$, as long as we know that depth of a node v and the last assigned index i, we can assign the indexes to v and its two atom nodes a_1 and a_2 by the following rules:

- if v is a leaf node in T_L (not in T_B) with $dep(v) < h+1$, then
  - index$(v) = i + 2^{(h+1)-dep(v)}$, or in another word, the number of nodes in the left virtual subtree plus i;
  - the left atom node a_1 of v has index$(a_1) = i+1$;
  - the right atom node a_2 of v has index$(a_2) = i + 2^{(h+1)-dep(v)+1} - 1$, or in another word, the number of nodes in the left and right virtual subtree plus i minus 1.
- if v is not a leaf node in T_E, then
  - recur on the left child
  - index$(v) = i+1$.
  - recur on the right child

By using the above rules, we can use an in-order search algorithm to correctly assign the desired index to all the nodes in T_E as if they are in a complete binary tree T_V without physically adding dummy nodes and links. Thus we use the term "virutal". The algorithm is shown in Algorithm 6.

The in-order search step takes only $O(n)$ in time since each node is visited only once in T_E.

3.1.3 Finding the Nearest Common Ancestor

Now we are in position to discuss how the nca of two nodes in T_E can be found efficiently. We first have the following theorem.

Theorem 1. *For a pair of distinct leaf nodes u and v in T_E, with index(u) <index(v), the forward port set of u, F_u, and the backward port set of v, B_v, has exactly one node w in common, which is also their nca. Additionally, index(u) < index(w) < index(v).*

Proof. Suppose the nca of u and v is the node w, and the tree rooted at w is T_w. Since all ancestors of any leaf node x is either in F_x or B_x but not both, and u and v are distinct, w has to be in one of $F_u \cap B_v$, $F_u \cap F_v$, $B_u \cap F_v$, or $B_u \cap B_v$. If w is in $B_u \cap B_v$, then u and v are both in the right subtree of T_w. Say the right child of w is w' (w' exists because u and v are distinct and they are leaf nodes), then u and v has to be in the subtree rooted at w', therefore w' is a common ancestor of u and v with a larger depth than w, contradiction. A similar argument can apply to prove that the nca is not in $F_u \cap F_v$. If w is in $B_u \cap F_v$,

then u is in the right subtree of w and v is in the left subtree of w, hence according to the in-order search, index(u) > index(v), contradiction. Thus the nca of u and v is in the intersection of F_u and B_v.

We now want to prove that $F_u \cap B_v$ has only one element. Suppose that there are at least two elements in $F_u \cap B_v$, clearly the nca w is the one with the largest depth. Say another node, different from w, in $F_u \cap B_v$ is p, then either index(p) > index(v) or index(p) < index(u). Since w and p are both in F_u and B_v, w is in the left subtree and the right subtree of the subtree induced by p, contradiction. This also proves that index(u) < index(w) < index(v). □

To answer nca query (for later use in the shortest distance query) between a pair of nodes u and v in T_E with index(u) < index(v), if we first "virtually remove" (ignore) all nodes in F_u and B_v that are not in the range [index(u),index(v)], then sort the remaining elements in F_u and B_v in ascending depth in T_E, call the resulting sets $F'_u = \{u_{fmin}, \cdots\}$ and $B'_v = \{v_{bmin}, \cdots\}$, We then have the following theorem:

Theorem 2. *The nca of a pair of nodes u and v with index(u) < index(v) is $u_{fmin} = v_{bmin}$.*

Proof. Suppose the nca of u and v is w, and dep(w) > dep(u_{fmin}), then w, as well as v, is in the left subtree of the tree rooted at u_{fmin}, thus index(v) < index(u_{fmin}), contradicts to Theorem 1. Similarly, if dep(w) > dep(v_{bmin}), then index(u) > index(v_{bmin}). Again, it contradicts to Theorem 1. □

The above theorem provides a way to quickly identify the nca of u and v using only F_u or B_v: the first element w in sorted F_u or sorted B_v such that index(u) < index(w) < index(v) is the nca of u and v. Thus when querying the nca of u and v, we need only one of F_u or B_v. Suppose we use F_u only, then all we need to do is to iterate through F_u, as soon as we find a node w in F_u with index(u) < index(w) < index(v), this w will be the nca of u and v. The process takes $O(\log n)$ time in theory. In practice, however, it is good enough, because even if we have a planar graph with one billion edges, the nca search takes only 30 comparisons. The space consumption for each node is $O(\log n)$, because we need to store F_v and B_v for each node v in T_E.

To further improve the time on computing the nca of two nodes in T_E, we can use the algorithm proposed by Harel and Tarjan [23]. In their algorithm, they define an operation $\oplus$ on the index of the nodes: let $bin(i)$ be the binary representation of the integer i, then $i \oplus j$ is the integer k, where $bin(k) = bin(i)$ XOR $bin(j)$. Haerl and Tarjan used this operation to compute the nca of two nodes in a perfect binary tree. The algorithm is shown in Algorithm 7. Here we omit the proof of the correctness of the algorithm. Readers may refer to [23] for the proofs.

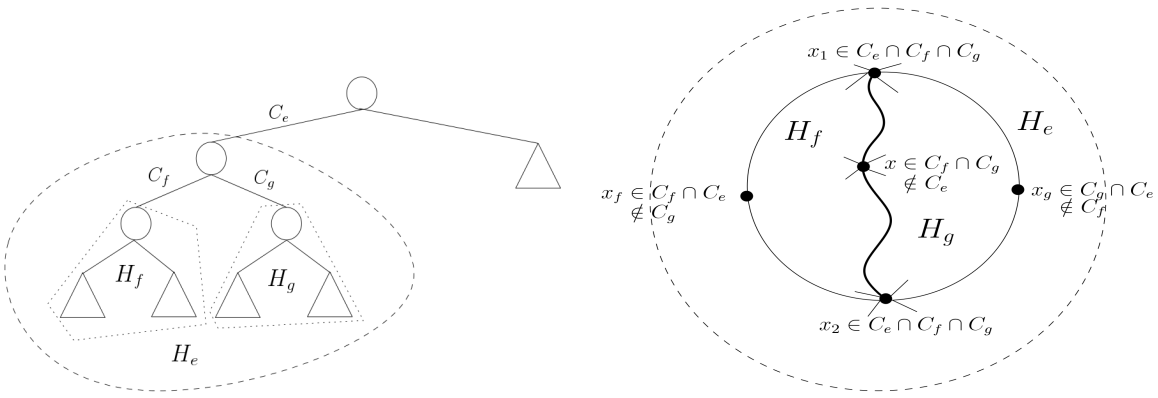
Algorithm 7 NearestCommonAncestor

Input: T, a perfect binary tree, v & u, the two nodes in T
Output: w, the nca of v and u

```
if v is the ancestor of u then
    return v
end if
if u is the ancestor of v then
    return u
end if
h ← ⌊log(index(v)⊕index(u))⌋
return The node w whose index is 2^{h+1}⌊index(v)/2^{h+1}⌋ + 2^h
```

3.1.4 Construction of the Cut Sets

This can be done in T_B. To find a cut set C_e of a link e in T_B, say the two children links of e are f and g, as shown in Figure 3.2(a) . We can do: $C_e = C_f \setminus C_g \cup C_g \setminus C_f$, where C_f and C_g are the cut sets associated with f and g, respectively. However, by doing this, we may ignore at most 2 vertices on the boundary of the region H_e induced by C_e ($x_1 \& x_2 \in C_e \cap C_f \cap C_g$, as shown in Figure 3.2(b)).



(a) An example of T_B, where circles represents nodes and triangles represents subtrees. C_e, C_f, and C_h are the cut sets associated with the three links in T_B. The dotted regions H_f and H_g are the subgraphs induced by the cut sets C_f and C_g. The dashed region H_e is the subgraph induced by the cut set C_e.

(b) An example of regions in G. The dashed circle is the input graph G, the solid line circle represents the region H_e induced by the cut set C_e. H_e is further decomposed into H_f and H_g. x_1 and x_2 are the example of the two ignored vertices in $C_e = C_f \setminus C_g \cup C_g \setminus C_f$.

Figure 3.2: An illustration of the correspondence between links (cut sets) in T_B and regions/subgraphs in G.

To address this, we can either check if there is a vertex in both H_f and H_g which is incident to some other vertex outside H_e; or, we can check the node degree of the vertices, if $\deg_{H_f}(u) + \deg_{H_g}(u) < \deg_G(u)$, then such an u is an ignored vertex. Both method takes $O(n^2)$ time to find in worst case. More specifically, if we initially set all graph edges to be external, and while we are constructing the cut sets from bottom up, say we are merging

H_f (induced by C_f) and H_g (induced by C_g) to form H_e, we mark the edges incident to a vertex in either C_f or C_g (a vertex on the boundary) as internal **with respect** to H_e. We can then check for each node in $C_g \cap C_f$ if some nodes has both internal edges (with respect to H_g and H_f) and external edges (may be internal with respect to some other region), then these nodes are the ignored ones. For each edge, we only need to store the largest region it resides in to keep track of in which regions the edge is internal.

If we have embedded information, we can then use the embedded information to find such ignored vertices. When checking the two possibly ignored nodes, since we have the embedded information, we have an ordering of the nodes in $C_e \cap C_f \cap C_g$. The ignored vertices can only be the first and/or the last nodes in this ordering, thus we only need to check the first and the last node. This approach reduces the time complexity to $O(n)$ in theory.

There are $O(n)$ nodes in T_E, thus the process has a time complexity of $O(n^2)$ and uses $O(bw(G)n)$ space.

3.1.5 Shortest Distances from Vertices to Cut Sets

To finish up the pre-processing, we need to calculate the shortest distances from s to every vertex in the subgraph containing s (in the atom node that contains s), and the shortest distance from s to the vertices in the cut set of every ancestor of w in T_E (there are at most $\log n$ of such ancestors). This can be done by running the one-to-all Dijkstra's algorithm for each vertex $s \in V(G)$ and storing only the shortest distances from s to the vertices described above. This takes n executions of Dijkstra's algorithm and thus taking a total of $O(n^2 \log n)$ time and $O(bw(G)n \log n)$ space.

3.1.6 Answering Shortest Distance Queries

Algorithm 8 Query

Input: T_V, the computed virtual tree, s, the source, t, the destination
Output: $d_G(s,t)$, the shortest distance from s to t in G

```
1: Find the two atom nodes w_1 and w_2 that contain s and t
2: w ← NearestCommonAncestor(T_V, w_1, w_2)
3: C_w ← the cut set associated with w
4: d ← positive infinity
5: for each vertex v ∈ C_w do
6:    if d_G(s,v) + d_G(v,t) < d then
7:       d ← d_G(s,v) + d_G(v,t)
8:    end if
9: end for
10: return d
```

To answer a shortest distance query from s to t in G. Assume the pre-processing is done, we first locate the atom nodes w_1 and w_2 in T_E that contains s and t, respectively. If $w_1 = w_2$, then we just look up the shortest distance. If not, we first find the nca of w_1 and w_2. Assume the nca is w, and the cut set associated with w is C_w. We then find the vertex $u \in C_w$ such that $d_G(s,u) + d_G(u,t)$ is the smallest. The result is the shortest distance from s to t in G. In the worst case, it takes $O(bw(G))$ to find the desired shortest distance. The algorithm is shown in Algorithm 8

3.2 Distributed Version of the Oracle

By the centralized construction of the oracle, it can be stored independent from the input graph G. It can, however, also be stored in a distributed way. As discussed in Section 3.1.6, the information we need to calculate the shortest distance from s to t in G is the distances mentioned in Section 3.1.5 and the nca of the two atom nodes in T_E containing s and t. The nca can be found using the forward port set and the backward port set as discussed in Section 3.1.3. Thus, for each vertex v, we can store the indexes of the ancestors of the atom node containing v in T_E, and the required shortest distances. Therefore, we have a distributed version of the oracle as well. The labeling scheme takes $O(bw(G) \log n)$ space.

3.3 The Complexities and the Trade-Off of the Oracle

The oracle takes $\min\{O(n \log^3 n \log k),\ O(nk^2 \log k)\}$ time to find a branch decomposition tree T_B of width $O(k)$ theoretically for a planar graph G with $k = bw(G)$ [21], $O(n^2)$ time to transform T_B into T_E, $O(n)$ time to assign in-order indexes, $O(n^2)$ time to construct cut sets, and $O(n^2 \log n)$ time to compute selected shortest path distances. Thus it takes $O(n^2 \log n)$ time to construct. It answers the shortest distance query in $O(bw(G))$ time and uses $O(bw(G) n \log n)$ space. The label size for each node in T_E, if the oracle is stored distributively, is $O(bw(G) \log n)$. The product of query time and oracle size of our oracle is $O(k^2 n \log n)$, where $k = bw(G)$. For planar graphs G with $bw(G) = o(n^{\frac{1}{3}})$, our oracle has a better product of query time and oracle size than the best known result of $O(n^{\frac{5}{3}} \log n)$ in [9].

4. Computational Study

The program was implemented in Java 1.7. The implementations of the above algorithms are straight forward using the pseudo code provided above. We used Algorithm 7 to find the nca of a pair of vertices. Notice that because of the structure of a branch decomposition tree, a vertex v in the graph G may appear in multiple atom nodes in the virtual tree T_V. When answering a shortest distance query from s to t, we always choose the first atom node we can find that contains the vertex s, and so does for the vertex t. We tested our implementations on three classes of graph instances against both one-directional Dijkstra's algorithm (or just Dijkstra's algorithm) and bi-directional Dijkstra's algorithm. Both versions of Dijkstra's algorithms are implemented in Java. For the three classes of graph instances, Class (1) instances include Delaunay triangulations of point sets taken from TSPLIB [31]. The instances were used as test instances in the previous studies [6, 25, 24] on branch decompositions. Class (2) instances are generated by LEDA library based on some geometric properties [6]. Class (3) instances are generated by PIGALE library [1], which generates random planar graphs with a given number of edges based on the algorithm [36]. The three classes of planar graphs are commonly used in the previous computational studies on planar graphs. The three classes were chosen as test subjects because the three libraries are popular and available to us. The branch decompositions were calculated using previously implemented algorithms in [6]. The branch decomposition program was written in C++ and was tested on the three classes of graph instances as well, readers may refer to [6] for computation times of the graph instances. We tested the program on the three classes of graph instances for point-to-point shortest distance queries and reported the following attributes:

- G, the name of the graph;
- $E(G)$, the number of edges in the graph G;
- $bw(G)$, the branchwidth of the graph G;
- Preprocessing Time (PT) in seconds, given that we already have the branch decomposition;
- Oracle Size (S) in MB;

- Oracle Query Time (OQT) per 1000 queries in seconds;
- Dijkstra's algorithm Query Time (DQT) per 1000 queries in seconds;
- Bi-directional Dijkstra's algorithm Query Time (BDQT) per 1000 queries in seconds;
- D/O = DQT/OQT, a ratio that tells us how much faster can our oracle answer a query compared to Dijkstra's algorithm; and
- B/O = BDQT/OQT, a ratio that tells us how much faster can our oracle answer a query compared to Bi-directional Dijkstra's algorithm.

All the above data are either rounded to two decimal places, or to one significant digit if two decimal places are not enough. D/O and B/O are calculated using the un-rounded data, then rounded to two decimal places. We may refer to D/O and/or B/O as improvement ratios.

The program was executed on a server with Intel(R) Xeon(R) 2.80GHz x86_64 CPU, 8GB physical memory and 8GB swap memory. The operating system is CentOS 6.9, and the programming language we used is Java.

4.1 Experimental Results

4.1.1 Results for instances in the Three Classes

The computational results for Class (1) instances are shown in Table 4.1. The data show that the shortest distance query time using our oracle is much faster than those using the two versions of Dijkstra's algorithm, by a factor of 270 to 800 for instances of more than 5000 edges compared to Dijkstra's algorithm, and by a factor of 30 to 163 compared to Bi-directional Dijkstra's algorithm.

G	$E(G)$	$bw(G)$	PT	S	OQT	DQT	BDQT	D/O	B/O
d1655	4890	29	15.67	318.08	0.02	3.00	0.65	150.96	32.77
pr1002	2972	21	5.11	131.96	0.01	2.28	0.33	155.79	22.72
pr2392	7125	29	36.18	439.67	0.02	4.86	0.61	239.62	30.01
rl1323	3950	22	11.82	206.82	0.02	2.49	0.75	123.55	37.43
rl1889	5631	22	24.75	376.60	0.01	3.68	0.42	272.41	31.28
fl3795	11326	25	123.97	976.49	0.01	7.03	1.60	718.365	163.46
fnl4461	13359	48	202.28	1369.4	0.02	7.25	1.07	478.87	70.58
pcb3038	9101	40	56.91	867.4	0.01	4.68	0.65	325.82	44.97
rl5915	17728	41	372.45	2139.74	0.02	10.85	0.49	723.28	32.39
rl5934	17770	41	340.06	2289.7	0.01	10.39	0.70	798.37	53.82

Table 4.1: Computational Results (in seconds) of Shortest Distance Query Using This Oracle v.s. Dijkstra's Algorithm and Bi-directional Dijkstra's Algorithm for Class (1) Graphs

The computational results for Class (2) and (3) instances are shown in Tables 4.2 and 4.3. The data in these two tables show a similar result: our oracle answers shortest distance queries much faster than both Dijkstra's algorithm and Bi-directional Dijkstra's algorithm. This shows that the performance of our oracle is independent on the type of the input planar graph.

The data for all three classes also show that the query time is dependent on the branchwidth of the input graph G. The data show that smaller branchwidth yield a faster query time, regardless of the number of edges. This outcome is not surprising, since in section 3.1.6 we showed that the query time is $O(bw(G))$, which depends only on the value of $bw(G)$. This also rule out the possibility that the $O(\log n)$ time algorithm, Algorithm 7, we used in our program to find the nca is a major reason that slows the query process. Thus indicating that in practice, Algorithm 7 is good enough.

The preprocessing time is also relatively small compared to the time needed to compute the optimal branch decomposition as shown in Tables 1, 2, and 3 in [6]. Thus the whole preprocessing time including the branch decomposition process is dominated by the time used to find the branch decomposition. The oracle size is not very small, but it is also not intolerably large. The data show that the oracle size is proportional to the number of edges in the input graph. The oracle size may be reduced if we use another programming language or improve our algorithm.

G	$E(G)$	$bw(G)$	PT	S	OQT	DQT	BDQT	D/O	B/O
rand1160	2081	8	2.32	135.06	0.005	30.86	0.25	156.57	44.62
rand1672	3047	10	6.66	193.06	0.006	1.28	0.24	201.29	38.04
rand2236	4002	10	10.50	462.43	0.007	1.67	0.48	238.71	69.39
rand2780	5024	10	18.08	480.62	0.007	2.00	0.56	291.71	81.05
rand3325	6035	9	26.73	961.84	0.006	2.38	0.67	394.70	111.06
rand3857	7032	11	38.35	970.42	0.007	3.03	0.76	418.43	104.43
rand5446	10093	11	97.12	2000.19	0.007	4.00	0.79	538.76	106.89
rand8098	15031	13	323.84	5155.72	0.01	9.81	3.86	735.15	289.11
rand10701	20044	13	716.45	7310.19	0.01	16.38	1.99	1460.08	117.07
rand15902	30010	14	1466.30	17841.22	0.01	21.59	3.21	1615.78	240.25

Table 4.2: Computational Results (in seconds) of Shortest Distance Query Using This Oracle v.s. Dijkstra's Algorithm and Bi-directional Dijkstra's Algorithm for Class (2) Graphs

One potential reason that slows the query process of our oracle is the number of memory accesses in the query stage. The program needs to do multiple memory accesses in order to get the correct nca as well as the cut set associated with it. While in Dijkstra's algorithms, one to two memory accesses are enough.

The main improvement in our oracle is that when answering a shortest distance query from s to t, the oracle needs only to consider $bw(G)$ number of vertices given the nca of s and t, while in the two versions of Dijkstra's algorithm, a much larger number of redundant

G	$E(G)$	$bw(G)$	PT	S	OQT	DQT	BDQT	D/O	B/O
PI1180	2202	7	3.28	110.06	0.007	1.58	0.37	210.15	56.64
PI1182	2016	7	3.87	93.44	0.007	1.76	0.22	250.50	31.43
PI1186	2029	6	3.81	86.78	0.008	1.38	0.33	179.91	43.45
PI1193	2019	6	3.95	87.83	0.006	1.54	0.28	252.49	45.49
PI1207	2029	9	3.74	88.18	0.006	1.38	0.29	225.53	46.31
PI2995	5043	7	37.94	592.45	0.006	2.85	0.59	476	98.42
p3586	6080	8	40.05	840.81	0.007	3.41	1.68	465.16	229.67

Table 4.3: Computational Results (in seconds) of Shortest Distance Query Using This Oracle v.s. Dijkstra's Algorithm and Bi-directional Dijkstra's Algorithm for Class (3) Graphs

vertices needed to be considered. Intuitively, for a shortest path P that contains a large number of edges, the two Dijkstra's algorithm tend to do a huge number of iterations in order to find P, due to its BFS-like search pattern. This redundancy is even worst when the graph is unweighted: the search is BFS. This improvement is also reflected in the experimental results: the larger the number of edges in a graph, the better the improvement ratios tend to be. Thus one may suspect that for a shortest path distance of a path with a large number of edges, our oracle can have an even better improvement ratios than those shown in Tables 4.1, 4.2, and 4.3. The above observation is experimented and discussed in section 4.1.2.

4.1.2 Experiments and Results for Shortest Distance Queries with Long Paths

As mentioned above, long paths (shortest paths with a large number of edges) may yield a larger improvement ratio and short path (shortest paths with a small number of edges) may yield a less appealing ratio. Consider two vertices s and t, which are relatively close, in the sense that they have a small number of vertices between them in a shortest path. Bi-directional Dijkstra algorithm will find the path using only a small number of iterations. Whereas, if the branchwidth is large, the oracle needs to consider a large number of vertices in the boundary in order to find the shortest distance. For shortest paths with large numbers of vertices, Bi-Dijkstra's algorithm tends to consider more redundant vertices than the oracle does, thus making oracle more efficient than Bi-directional Dijsktra's algorithm.

We tested the query time on paths with a large number of edges and paths with a small number of edges for several graphs, they all yield similar results: long paths queries has a better improvement ratios than short paths. In the experiment, a random source s in G is selected, then the shortest path distance and the number of edges in the shortest path from s to every vertex in G is computed using Dijkstra's algorithm. As a result, a list of vertices is returned. The list is sorted in ascending order such that the shortest path from s to the first vertex in G in the list has the smallest number of edges, and the shortest path from s to the last vertex in G in the list has the largest number of edges. The long paths

Graph G	$E(G)$	$bw(G)$	Number of Edges in the Shortest Path	B/O
rand3857	6035	11	830 - 855	165.83
			10 - 25	87.22
p3586	6080	8	200 - 224	237.55
			5 - 15	165.71
fnl4661	13349	48	780 - 807	78.28
			10 - 25	60.88

Table 4.4: Computational Results of Long/Short Path Shortest Distance Queries Using This Oracle v.s. Bi-Directional Dijkstra's Algorithm

are selected around the tail of the list and the short paths are selected starting from the $(\frac{n}{100})$-th position of the list. Here we listed three graphs from the three classes. The results are shown in Table 4.4. We did not test the oracle on paths with only one to two edges because the associated cut set sizes will be small, and thus both algorithm should yield a good query time.

The table shows that for long paths, our oracle performs better than the average; and for short paths, our oracle performs worse. The data also show that for long paths, the B/O ratio is affected by both branchwidth of the input graph and the number of edges in the shortest path. With similar branchwidth, the B/O ratio of the graph rand3857 is increased by approximately 58.80%, while that for the graph p3586 is only increased by 3.43%. With similar number of edges in a long path for rand3857 and fnl4661, the B/O ratio is increased only by 10.91% for fnl4661. The reason for such dependence is easy to understand: if there are many edges in a path, then Dijkstra's algorithm tends to use more iterations in order to find the desired shortest path distance than our oracle. And the larger the branchwidth is, the more vertices in the cut set our oracle needs to consider when computing the shortest path distance.

For short paths, the reason that the B/O is below what is listed in Tables 4.1, 4.2, and 4.3 is that bi-directional Dijkstra's algorithm needs to explore much fewer vertices than for the long paths. Also, the proportion of the time spend on the memory accesses becomes larger. It is anticipated that if the program is written in programming languages that has less overhead than Java, the performance would be much better. The results, however, still show that our oracle has a better query time than the two versions of Dijkstra's algorithms even for short paths.

5. Conclusion

We proposed and implemented a branch-decomposition based exact oracle for point-to-point distance problem in planar graphs. Computational studies show that our oracle performs well on planar graphs with small branchwidth. The oracle is particularly much more efficient than Dijkstra's algorithm when answering a shortest distance query of a shortest path with a large number of edges, but less improvement over Dijkstra's algorithm is achieved if the path contains a small number of edges. This is due to the fact that when we transform the decomposition tree into a virtual tree, we include every link in the decomposition tree. Thus making the subgraph associated with each atom node very small, namely, one edge. Hence the query rely heavily on the cut sets. This makes it interesting to consider a case where we do not completely transform the decomposition tree into a virtual tree, rather, we stop when there is a certain number of leaf nodes in the subtree. By doing so, the subgraphs associated with each atom node will be a graph with a small number of vertices.

An interesting future work is to implement the above idea and experiment on it to see if this can improve the query time on paths with small number of edges. Another interesting open problem is that is there a way to choose associated atom nodes for s and t so that the nca is as close to s and t as possible?

Moreover, planarity properties used in our oracle are for finding a good branch decomposition. Although planarity give us a good embedded information that we can use to deal with the ignored boundary vertices as mentioned in Section 3.1.4, it is not mandatory. Thus, the input graph does not have to be planar as long as a good branch decomposition can be obtained. Theoretically, a graph G of a constant genus (the minimum number of handles that must be added to the plane to embed the graph without any crossings) or non-orientable genus, a branch decomposition of G with branchwidth $O(bw(G))$ can be computed in polynomial time [26]. For an arbitrary graph G, it is NP-hard to compute an optimal branch decomposition of G [35], but a branch decomposition of G with branchwidth $O(bw(G))$ can be computed in $O(2^{O(bw(G))}n^2)$ time [38]. For graphs that are close to planar, however, there may be some techniques we can use to find a good branch decomposition.

In [4], Bashir finds a good carving decomposition for non-planar graphs G by first removing edges so that the graph becomes planar, say the resulting graph is G', then computing an optimal carving decomposition on G', and finally adding the edges back.

The above idea is based on the fact that a carving decomposition of G' is also a carving decomposition of G, given that $V(G) = V(G')$. Although this fact is not true for branch decomposition, it provide us a potential approach to deal with non-planar graphs: first remove some edges in the input graph G so that the resulting graph G' becomes planar, then compute an optimal branch decomposition T_B of G'. To add back the removed edges, for any removed edge $e = (x, y)$, first find a link $l = (u, e')$ in T_B where e' and e share a common end point, namely x. Then remove the link l, add a new link $l' = (u, e'')$ in T_B, where e'' is a new node. Finally we attach e and e' to e''. The resulting branch decomposition may not be optimal, but it is very close to optimal if the input graph is close to planar. Another way to deal with non-planarity is that we can contract a small non-planar subgraph of the input graph G into a "super vertex", making the resulting graph G' planar, then run our construction. With this approach, however, we need extra space for storing shortest distances between vertices within "super vertices", and shortest distances from each vertex in a "super vertex" to vertices in the cut sets of the parent nodes. The above two approaches for adapting our oracle to non-planar graphs exploit many interesting open problems and hence are good future works.

The program we used in the experiments was implemented in Java, thus everything has to be implemented in an object-oriented point of view. This creates a large amount of overheads and increases the memory usage of the oracle. Therefore it is also interesting to test if the oracle can perform better if it is implemented in another language like C/C++.

References

[1] Public implementation of a graph algorithm library and editor. `http://pigale.sourceforge.net/`, 2008.

[2] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, pages 55–66, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] M. Babenko, A. V. Goldberg, A. Gupta, and V. Nagarajan. Algorithms for hub label optimization. In *Automata, Languages, and Programming: 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, pages 69–80, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] M. Bashir and Q. Gu. Carving-decomposition based algorithms for the maximum path coloring problem. In *Communications (ICC), 2012 IEEE International Conference on*, pages 2977–2982. IEEE, 2012.

[5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 12:87–90, 1958.

[6] Z. Bian and Q. Gu. Computing branch decomposition of large planar graphs. *Lecture Notes in Computer Science*, 5038:87–100, 2008.

[7] S. Cabello. Many distances in planar graphs. *Algorithmica*, 62(1):361–381, Feb 2012.

[8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[9] V. Cohen-Addad, S. Dahlgaard, and C. Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs. *CoRR*, abs/1702.03259, 2017.

[10] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940 – 952, 2013. Best Papers: International Parallel and Distributed Processing Symposium (IPDPS) 2010, 2011 and 2012.

[11] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Experimental Algorithms: 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, pages 376–387, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[12] D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, pages 18–29, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] H. N. Djidjev. Efficient algorithms for shortest path queries in planar digraphs. In *Graph-Theoretic Concepts in Computer Science: 22nd International Workshop, WG '96 Cadenabbia, Italy, June 12–14, 1996 Proceedings*, pages 151–165, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[15] D. Eppstein and M. T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '08, pages 16:1 – 16:10, New York, NY, USA, 2008. ACM.

[16] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868 – 889, 2006. Special Issue on FOCS 2001.

[17] L.R. Ford. *Network Flow Thoery*. RAND Corporation, Santa Monica, CA, USA, 1956.

[18] C. Gavoille, D. Peleg, S. Pérennes, and R/ Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85 – 112, 2004.

[19] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms: 7th International Workshop, WEA 2008 Provincetown, MA, USA, May 30-June 1, 2008 Proceedings*, pages 319–333, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[20] Q. Gu and H. Tamaki. Optimal branch-decomposition of planar graphs in o(n3) time. *ACM Trans. Algorithms*, 4(3):30:1–30:13, July 2008.

[21] Q. Gu and G. Xu. Near-linear time constant-factor approximation algorithm for branch-decomposition of planar graphs. *CoRR*, abs/1407.6761, 2014.

[22] Q. Gu and G. Xu. Constant query time $(1 + \epsilon)$-approximate distance oracle for planar graphs. In *Algorithms and Computation: 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, pages 625–636, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[23] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

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

[25] I.V. Hicks. *Branch decompositions and their applications*. PhD thesis, Rice University, 2000.

[26] Torsten Inkmann. *Tree-based decompositions of graphs on surfaces and applications to the Traveling Salesman Problem*. Georgia Institute of Technology, 2008.

[27] M. Jiang, A. W. Fu, R. C. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *Proc. VLDB Endow.*, 7(12):1203–1214, August 2014.

[28] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *J. Exp. Algorithmics*, 11, February 2007.

[29] S. Mozes and C. Sommer. Exact shortest path queries for planar graphs using linear space. *CoRR*, 2010.

[30] L. F. Muller and M. Zachariasen. Fast and compact oracles for approximate distances in planar graphs. In *Algorithms – ESA 2007: 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007. Proceedings*, pages 657–668, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[31] G. Reinelt. Tsplib-a traveling salesman library. *ORSA J. on Computing*, 3:376 – 384, 1991.

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

[33] N. Robertson and P.D. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

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

[35] Neil Robertson and Paul D Seymour. Graph minors. xiii. the disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.

[36] G. Schaeffer. Random sampling of large planar maps and convex polyhedra. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 760–769. ACM, 1999.

[37] R. Sedgewick. *Section 21.7: Negative Weights*, chapter Shortest Paths. Algorithms in Java. Addison-Wesley Professional, 3rd edition, July 2003.

[38] Paul D Seymour and Robin Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[39] C. Sommer. Shortest-path queries in static networks. *ACM Comput. Surv.*, 46(4):45:1–45:31, April 2014.

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

[41] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, January 2005.

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

[43] C. Wulff-Nilsen. Approximate distance oracles with improved preprocessing time. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 202–208, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[44] C. Wulff-Nilsen. Approximate distance oracles for planar graphs with improved query time-space tradeoff. *CoRR*, abs/1601.00839, 2016.

[45] Y. Xiang. Answering exact distance queries on real-world graphs with bounded performance guarantees. *The VLDB Journal*, 23(5):677–695, Oct 2014.