Compact Routing on Planar Graphs

by

Newsha Seyedi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis delves into the exploration of shortest path queries in planar graphs, with an emphasis on the utilization of space-efficient data structures. Our investigation primarily targets connected, undirected, static pointer planar graphs, focusing on scenarios where queries predominantly start or end at a select subset of nodes.

The shortest path problem, central to our study, boasts a rich historical context and has profound real-world implications in diverse fields such as web mapping, robotics, and VLSI circuit design. Our research is pivoted on the space-efficient representation of planar graphs, a critical consideration in 2D visualizations and city map representations.

In this thesis, shortest path queries are delineated into three categories: shortest path, distance oracle, and port queries, each with distinct computational characteristics and storage requirements.

A significant portion of our research is focused on center-based configurations in graphs, where a small subset of nodes, designated as 'centers,' plays a pivotal role. These centers are crucial, either due to their strategic importance within the graph, which necessitates more prompt responses to queries, or due to their high frequency in the query list. We explore various scenarios within this configuration. Our approach prioritizes handling queries involving these centers more efficiently, aiming to provide rapid responses for strategically important queries and to enhance overall query processing speed. This method is particularly effective, as addressing the queries linked to these relatively few but significant centers can substantially improve the efficiency of the entire system. Such prioritization reflects practical applications like urban navigation, where focusing on key locations can significantly expedite overall navigation and operational efficiency.

For shortest path queries in a center-based configuration, we have developed a data structure that efficiently answers queries from other nodes to centers in $O(\text{length of the path})$ time. In the first scenario, where all queries are from or to a center, the space requirement is $3n+2m+2km+o(nk)$, where n represents the number of nodes, m the number of edges, and k the number of centers. Additionally, our approach supports distributed storage and processing, facilitating parallel computing.

For distance oracle queries in unweighted graphs within a center-based configuration, our methods manage responses in $O(\log^{1+\epsilon} n)$ time, where ϵ is any constant greater than zero, with an additional $o(nk)$ space requirement. In general, for unweighted graphs without any specific configuration, the distance oracle requires $2n + 2m + 2nm + o(n)$ bits of space, offering responses in a similar time frame. The strength of our approach lies in

its distributability across multiple servers, which enhances concurrent query processing, a feature particularly beneficial in center-based configurations.

Moreover, we introduce a specialized data structure for distributed routing tables, capable of responding to port queries in constant time. This structure efficiently utilizes space, limiting the aggregate bit requirement for all routing tables within graph G to $3.2n^2 + o(n^2)$ bits.

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Professor J. Ian Munro, for his invaluable support and guidance throughout my master's thesis journey. His profound expertise, endless patience, and consistent encouragement were the cornerstones of my research, particularly during challenging phases. His mentorship has been a privilege and a significant factor in my academic growth.

I am also deeply thankful to my husband and best friend, Sayed Mohammad Amin Khodaee. His unwavering support, both spiritually and technically, has been a source of strength and inspiration. His presence and assistance have been pivotal in my journey.

My sincere appreciation goes to my family, especially my mother and father, for their endless love and support. Their belief in me and their encouragement have been my pillars of strength and motivation. I am truly grateful for their presence in my life and the support they have provided me every step of the way.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis explores shortest path queries in planar graphs using space-efficient data structures. The types of queries we aim to address include shortest path, distance oracle, and port queries. The main focus of this thesis is on connected, undirected, static pointer planar graphs, where most queries start or end at a small subset of nodes.

1.1 Motivation

In this thesis, we delve into the compact shortest path problem, primarily focusing on planar graphs.

The shortest path problem, a challenge dating back to 1953 [31], stands as a cornerstone in various domains such as web mapping, robotics, transportation, and the intricate designs of Very-Large-Scale Integration (VLSI) circuits. Its importance is underscored by its profound implications in real-world scenarios, especially navigational queries.

In this thesis, our exploration centers on the space-efficient representation of planar graphs. **Planar graphs** [33], characterized by their ability to be depicted on a plane without intersecting edges, are crucial in a variety of applications. Their significance is especially evident in areas such as 2D visualizations and city map representations.

Our research seeks to bridge two pivotal domains: the longstanding challenges of the shortest path problems and the space-efficient representations of planar graphs. By exploring the compact representation of shortest paths in planar graphs, we aim to further our understanding and enhance the practical applications in these fields.

1.1.1 Potential Applications

The implications of our research extend to numerous problems in computer science. To elucidate, here are two fundamental problems that can benefit from our findings:

1. **Road Network Routing**: Analogous to road network routing, we engage with k designated vertices, striving for rapid responses to shortest path queries from these vertices. In this scenario, a solitary system, inclusive of a CPU and RAM, drives the computation of the shortest path.

2. **Switch Network Routing**: This addresses the transmission of packets in a network comprised of switches. Every switch retains an essential fraction of the data and is tasked with determining the next port on the shortest path to the intended switch.

1.2 Problem Definition

1.2.1 Shortest Path

Consider $G(V, E)$, a simple non-negative weighted undirected connected planar graph with n vertices and m edges.

Definition 1.2.1. Path: Consider a graph $G(V, E)$. A **path** P from a vertex u to a vertex v in this graph can be defined as a sequence of vertices and edges $P = (x_0, e_1, x_1, e_2, ..., e_k, x_k)$, where $x_0 = u$ and $x_k = v$. Each edge e_i in this sequence, for $1 \leq i \leq k$, is the link between the vertices x_{i-1} and x_i. The term "edges in the path" is the set $\{e_1, e_2, ..., e_k\}$, which includes all the edges that sequentially connect the vertices in the path. The **length** of a path is determined by the number of edges in that path, which in this case, is k.

Definition 1.2.2. Weighted Graph: A graph in which each edge e has an associated numerical value, termed its weight, represented as $w(e)$. This weight can represent distances, costs, etc.

Definition 1.2.3. Non-negative weighted graph: A *non-negative weighted graph* is a weighted graph in which all edge weights are non-negative, formally denoted as:

$$\forall (u, v) \in E, w(u, v) \geq 0$$

where E is the set of edges and $w(u, v)$ represents the weight of the edge between vertices u and v.

Definition 1.2.4. Unweighted Graph: In this type of graph, the edges do not have any specific weights assigned to them. Typically, the cost or weight of an edge in an unweighted graph is considered to be 1 for the purpose of algorithms or analyses.

Let $\mathcal{P}_{u,v}$ be the set of all possible paths from vertex u to vertex v in the graph G. Each path $P(u, v)$ in $\mathcal{P}_{u,v}$ has an associated **cost**:

$$\text{Cost}(P(u, v)) = \sum_{e \in P(u,v)} w(e),$$

where $w(e)$ designates the weight of edge e.

The **shortest path problem** involves finding a path $P^*(u, v)$ within the set $\mathcal{P}_{u,v}$ that has the lowest cost. This can be expressed as:

$$P^*(u, v) = \min_{P(u,v) \in \mathcal{P}_{u,v}} \text{Cost}(P(u, v))$$

The objective of the shortest path problem is to find and explicitly present a path $P^*(u, v)$ from the set of all possible paths between u and v such that the cost of $P^*(u, v)$ is minimized. For unweighted graphs, the objective is to minimize the **length** of the path.

Definition 1.2.5. Length of a Path: refers to the number of edges present in the path. For unweighted graphs, the goal is to minimize the length.

Definition 1.2.6. SPL (Shortest Path Length): Given any two vertices u and v in a graph G, the Shortest Path length, **SPL(u, v)**, varies based on the nature of the graph:

- If G is a weighted graph: **SPL(u, v)** denotes the maximum number of edges among all the shortest paths between u and v. It represents the number of edges in the longest shortest path (by number of edges) from u to v, where the shortest is the path with the minimum cost.
- If G is an unweighted graph: **SPL(u, v)** is equivalent to the number of edges of the shortest path between u and v.

Example 1.2.1. ***Shortest Path Length (SPL):*** *Consider a graph with two distinct paths from vertex u to vertex v:*

- ***Path A:*** *Consists of 2 edges, each with a weight of 1000.*

- ***Path B:*** *Comprises 10 edges, each with a weight of 1.*

Analysis:

- *The total weight of Path A is* $2 \times 1000 = 2000$.
- *The total weight of Path B is* $10 \times 1 = 10$.

Despite Path A having fewer edges, Path B has a lower total weight.

Conclusion: *In accordance with the definition of SPL, the Shortest Path Length from u to v in this graph is determined by the path with the lowest total weight. Therefore, the SPL from u to v is 10, as Path B is the shortest in terms of total weight and contains 10 edges.*

Throughout this thesis, we primarily focus on two categories of graphs: **unweighted graphs** and **non-negative weighted graphs**.

Whenever we refer to graph G without explicitly stating its weight status, it should be understood that G is unweighted. Conversely, if we describe a graph as "weighted", we are specifically referring to a non-negative weighted graph. This convention is adopted to maintain clarity and consistency throughout our discussions.

1.2.2 Port Problem

The **port problem**, addressed in our studies, seeks to identify the vertex after u on a shortest path from u to v in a potentially weighted graph G. Notably, when multiple shortest paths exist, the port can be the vertex after u in any of those paths. If the time taken to resolve a port query is denoted as "port query time," then the computation of the shortest path can be performed in $O(SPL(u, v) \times$ port query time) time. Here, **SPL(u, v)** represents the maximum length (number of edges) among the shortest weighted paths linking vertices u and v.

1.2.3 Query List

A **query list** is a sequence of q queries, each comprising a pair of vertices, u and v. For every such pair, the intent is to extract specific information about the shortest path from u to v. While each query list is homogenous, containing questions of the same type, different query lists can pose different types of questions:

1. **Shortest Path**: Here, for every query (u, v) in the list, the response is the exact shortest path from u to v.

2. **Distance Oracle**: In this type, for each query (u, v), the answer is the cost of the shortest path from u to v.

3. **Port**: For this query type, for every query port(u, v), the response should indicate the first vertex after u on a shortest path from u to v.

In addressing these queries, we introduce different configurations.

1.2.4 Center-Based Configurations

In many applications, a subset of vertices hold more importance than others. Within graph G with n vertices, a subset of these vertices, called **centers**, stand out due to their heightened significance. Most queries either start or end at one of these centers. This reflects real-world scenarios, like in city navigation where most journeys either begin or end at a key location or center. With this in mind, we define:

Definition 1.2.7. Centers: A subset of nodes in G such that most or all queries (depending on the scenario) either start or end at one of these nodes. Essentially, these nodes in G are more important than others. This subset has a size of k, where $k \ll n$. Each node in this subset is termed as a **center**.

Definition 1.2.8. Centered and Non-Centered Queries: Queries that either start or end at one of the centers are termed **centered** queries, while those that do not involve any center are termed **non-centered** queries. Because G is undirected, without loss of generality, we can assume the centered queries originate from a center.

In our study, we examine several scenarios concerning the role of centers in query processing. These centers are pivotal points in the graph where queries either originate or conclude. The scenarios vary: centers might be predetermined or identified dynamically; they may remain constant or change over time, with changes either explicitly stated or implicitly inferred. Crucially, queries associated with these centers are often prioritized for expedited processing. This prioritization stems from the significance of these queries, which might be due to factors such as their frequency, criticality, or relevance to key operations. Therefore, our focus is not just on whether most queries involve centers, but on the inherent reasons that make these center-associated queries more important and necessitate their quicker resolution.

Center-Based Scenarios

In the context of our problem, different scenarios emerge depending on the behavior and knowledge concerning the centers in relation to the query list, where a query pertains to a question about the shortest path between a pair of vertices, u and v. In **Complete Association** scenarios, all queries are centered queries. However, in **Non-Complete Association** scenarios, there are centered and non-centered queries. Our algorithm can handle centered queries efficiently, but non-centered queries take more time. Thus, the performance in Non-Complete Association scenarios is related to the proportion of non-centered queries in the query list.

1. **Known Centers with Complete Association**: In this scenario, every shortest path query (u, v) is centered, meaning either u or v is a center. Since we have established that G is an undirected graph, we can, without loss of generality, assume that the first vertex of every centered query is a center. To clarify, if a query's second vertex is a center and the first one is not, the two can be swapped without affecting the answer. Consequently, for every shortest path query (u, v), we can assume that u is a center. Moreover, the identities of these center vertices are known in advance.

2. **Known Centers with Non-Complete Association**: In this scenario, while a substantial number of the shortest path queries (u, v) are centered, there are also non-centered queries present in the query list. The identities of the vertices that serve as these centers are known in advance.

3. **Unknown Centers with Non-Complete Association**: In this scenario, while a substantial number of the shortest path queries (u, v) are centered, there are also non-centered queries present in the query list. However, the identities of the vertices that constitute these centers are not known in advance.

4. **Variable Centers with Known Dynamics**: This scenario falls under the Non-Complete Association category. Unlike previous scenarios where centers remain static, here centers can dynamically change. A vertex designated as a center at one point might later lose its status, reverting to a non-center, and vice versa. At any given moment, the total number of centers does not exceed k, but the specific vertices playing the role of centers can vary over time. Importantly, not only are we aware of which vertices currently act as centers, but we are also informed about any changes in this status. Thus, we are always updated when a vertex joins or leaves the group of centers. Generally, there are three types of operations to modify centers,

and whenever any of these operations occur, it is explicitly communicated to the algorithm:

(a) Add a center: If the current number of centers is less than k, a vertex can be added to the centers.

(b) Remove a center: Provided there is at least one center, a center can be removed, turning it into a regular vertex.

(c) Replace a center with a normal vertex: This operation involves removing a center (turning it into a regular vertex) and promoting a regular vertex to center status. This operation keeps the total number of centers unchanged.

Note that any of these operations, when executed, is explicitly reported to the algorithm. Additionally, the initial set of centers is known in this scenario.

5. **Variable Centers with Unknown Dynamics**: Similar to the previous scenario, this falls under the Non-Complete Association category. In this context, centers are dynamic and can undergo changes as detailed by the three operations in "Variable Centers with Known Dynamics". However, visibility into these changes is restricted. The algorithm lacks knowledge of the initial set of centers and is not informed about any center change operations. This means it does not recognize which operation took place, which vertices were involved, and the timing of the operation. The current set of vertices acting as centers is not disclosed to the algorithm, and when shifts occur in their composition, the specifics and timings of these transitions remain concealed.

The dynamic nature observed in the last two "Variable" scenarios is of considerable practical significance. This adaptability ensures that the system can autonomously adjust to changing real-world conditions, such as shifts in urban area popularity. As a result, the system continuously aligns with the current dynamics without the need for manual recalibration.

1.3 Constraints

In this thesis, we operate under the subsequent technical constraints and assumptions:

1. **Graphs**: All graphs addressed in this thesis are simple and undirected.

2. **Logarithmic Base**: All logarithmic operations denoted as lg use a base of two. When ln is used, it refers to the natural logarithm.

3. **Computational Model**: We utilize the conventional $\Theta(\log n)$-bit unit-cost RAM model for our computations. Within this framework, operations, including reading, writing, and adding on $O(\log n)$ consecutive bits, are carried out in $O(1)$ time.

1.4 Results

1.4.1 Results for Center-Based Configurations

In our study, we dive into understanding the intricacies of center-based configurations within the context of planar graphs. We categorized our study into five distinct scenarios, each presenting its own set of challenges and characteristics:

1. Known Centers with Complete Association.
2. Known Centers with Non-Complete Association.
3. Unknown Centers with Non-Complete Association.
4. Variable Centers with Known Dynamics.
5. Variable Centers with Unknown Dynamics.

For each scenario, a suite of performance attributes is presented. A primary attribute is the **total space**, representing the number of bits required for storage in each scenario. Within these configurations, we pre-store the shortest path trees of certain vertices. It is possible to both remove and add the shortest path tree for a vertex. The act of storing a vertex's shortest path is termed a "**retrieve request**" or simply "**doing a retrieve**". Each retrieve operation has a time complexity of $O(n)$ in unweighted planar graphs, and $O(n \log n)$ in weighted planar graphs. Given this, the cumulative time complexity for resolving all the queries in the query list is represented as:

$$O\left(n \times \text{number of retrieves} + \sum_{\text{each query in the list}} \text{time complexity of the specific query}\right)$$

Definition 1.4.1. Stored Vertex and Stored Vertices: In the context of our algorithms operating under the center-based configuration, a **stored vertex** refers to any vertex for which we have stored the representation of its shortest path tree in memory. The collective set of such vertices, for which these representations are held, is termed **stored vertices**.

The time complexity for resolving the shortest path query in centered queries is $O(\text{Length})$, where Length represents the length of the shortest path. In the case of non-centered queries, a retrieval operation becomes necessary, as detailed in Section 4.1. Regarding the storage of shortest path trees, each tree for the **stored vertices** is distributed. This leads to two primary considerations: the space required for each stored vertex and the maximum number of such vertices. Therefore, the cumulative space requirement is the product of the **space needed for each stored vertex** and the **maximum number of stored vertices**, in addition to some shared space.

A crucial insight into this storage methodology is the labeling uniformity required across all shortest path trees. This consistent labeling means we cannot merely store the shortest path of each vertex in isolation without allocating some shared space. This constraint, rooted in the labeling uniformity, is precisely why it is infeasible to store every shortest path tree within a minimal footprint of $2n + o(n)$ bits.

To ensure clarity in our findings, we outline the results across multiple performance metrics or attributes:

- **Total Space (bits)**: Represents the total number of bits required to store information for each scenario.
- **Space per Stored Vertex (bits)**: Indicates the number of bits necessary for storing the shortest path tree of each individual vertex within the stored vertices. Notably, in all scenarios, the **Space per Stored Vertex (bits)** remains $2mk + o(nk)$ bits.
- **Maximum Number of Stored Vertices**: Specifies the upper limit on the number of vertices for which the representation of their shortest-path trees is maintained in memory.
- **Number of Retrieves**: Gives the total number of retrieve requests made. A retrieve request, detailed in Section 4.1, gets invoked when there is a need to store the shortest path of a certain vertex.

- **Retrieve Complexity**: Describes the computational complexity tied to each retrieve request. It has been determined that each retrieve operation has (n) complexity.
- **Query Answer Complexity**: Represents the time complexity involved in addressing a query. If at least one endpoint of a query is found among the stored vertices, the query can be addressed in $O(\text{SPL})$ time.

In Table 1.1, we present the results in a tabulated format for clarity and easy reference. It is essential to note that, for the scenarios and constraints discussed, we operate under the assumption that the graph G is unweighted.

Scenario	**Total Space(bits)**	**Maximum Number of Stored Vertices**	**Maximum Number of Retrieves**
Known with Complete	$3n + 2m + 2km + o(kn)$	k	0
Known with Non-Complete	$3n + 4m + 2km + o(nk)$	$k + 1$	$k + \frac{q}{f(n)}$
Unknown with Non-Complete	$2n + 10m + 8km + o(nk)$	$4k + 4$	$4k + \frac{4q}{f(n)}$
Variable with Known Dynamics	$3n + 4m + 2km + o(nk)$	$k + 1$	$k + \frac{q}{f(n)} + c$
Variable with Unknown Dynamics	$2n + 10m + 8km + o(nk)$	$4k + 4$	$4k + \frac{4q}{f(n)} + 4c$

Table 1.1: **Performance metrics for Center-Based Configurations in Unweighted Graph** G: Given q as the total number of queries and b as the number of non-centered queries, the ratio $f(n)$ is defined as $\frac{q}{b}$, indicating the relationship between the total number of queries and non-centered queries based on n. For variable scenarios, c represents the number of times elements within the centers undergo operations.

Weighted Graphs Considerations

In the case of a graph G with non-negative weights, the **Retrieve Complexity** changes to $O(n \log n)$ across all scenarios, instead of $O(n)$. This adjustment accounts for the complexities introduced by weighted paths. For scenarios other than 'Known Centers with

Complete Association,' an additional storage requirement of $mW + o(mW)$ bits is needed, where W represents the number of bits necessary to store a weight. It is important to note that the issue of negative weights does not arise when we ensure that all paths in the graph are simple, meaning that no nodes are repeated in any path. This constraint effectively eliminates the complications that negative weights could introduce in path calculations, making our analysis and the Retrieve Complexity applicable under this condition.

1.4.2 Results for Distance Oracle

In every center-based configuration scenario for an unweighted graph G, by allocating an additional $o(n)$ space for each stored vertex, we can manage the distance oracle in $O(\log^{1+\epsilon} n)$ time for any constant $\epsilon > 0$. Conversely, for an unweighted G, the distance oracle can be managed using $2n + 2m + 2nm + o(n)$ bits of space and can respond to each query in $O(\log^{1+\epsilon} n)$ time for any constant $\epsilon > 0$. However, there exists an algorithm [7] capable of managing the distance oracle using $O(n^{\frac{5}{3}} \log n)$ space, delivering a response in $O(\log n)$ time. The strength of our method is its distributability across n servers, with each server requiring no more than $2m + o(n)$ space, alongside a shared space of $2n + 2m + o(n)$. Queries can be processed concurrently. The principal advantage of our algorithm is its applicability to center-based configurations.

1.4.3 Results for Routing Table

In this thesis, we introduce a data structure specifically designed for storing routing tables. Each table requires a maximum space of $\frac{8}{15}nd + o(nd)$, where d represents the degree of the node in question. Notably, our proposed routing table provides the capability to respond to port queries in constant time. Additionally, we demonstrate that the aggregate bit requirement to accommodate all routing tables within graph G is confined to $3.2n^2 + o(n^2)$ bits.

1.5 Thesis Organization

Chapter 2 presents the background and notation used throughout the thesis. It begins with an overview of succinct and compact data structures, and delves into the fundamentals of planar graphs. The chapter concludes by offering an overview of the shortest path problem and its various representations.

In Chapter 3, we examine related work, comparing and contrasting space-efficient data structures and their applications in compact representations of planar graphs. This chapter also reviews existing literature on routing queries and the methods for their compact representation in planar graphs.

Chapter 4 details the methods developed during our research. It thoroughly investigates the center-based configuration, distance oracle, and decentralized routing table, providing insights into our novel approaches and the algorithms we have devised.

The concluding Chapter 5 synthesizes the results and contributions of our study, reflecting on the implications and potential for future work.

Chapter 2

Background and Notation

2.1 Succinct and Compact Data Structure Overview

In the domain of data structures, both compact and succinct models aim to optimize space, albeit with different methodologies and degrees of efficiency. Their primary challenge is not merely storing data close to the theoretical lower bound but doing so while efficiently processing queries.

Compact data structures strive to minimize space usage compared to traditional structures, without overly sacrificing their operational speed. These often employ innovative algorithmic techniques and encoding patterns for enhanced space efficiency, though without strictly adhering to the theoretical lower bound on space. In other words, if the information-theoretic lower bound of a structure is N, a compact data structure aims for a space usage of $O(N)$ bits and efficiently answers the queries.

In contrast, **succinct data structures** prioritize using space close to the theoretical lower bound. If the information-theoretic lower bound of a structure is N, a succinct design aims for a space usage of $N + o(N)$ bits. This approach pushes the structure's design to the very limits of theoretical space efficiency, ensuring efficient query support.

2.1.1 Bit Vector Representation

A bit vector of length n refers to a sequence comprising n bits. Three primary operations are essential for bit vectors:

- **Access(i):** Fetches the bit located at the i^{th} position within the bit vector.
- **Rank(i, b):** Given a position i and a specific bit b (either 0 or 1), this function determines the number of instances of bit b up to, and inclusive of, position i. When the bit b is not specified, it is assumed to be 1.
- **Select(j, b):** For a designated bit b and an integer j, this operation determines the position of the j^{th} occurrence of bit b within the vector.

Lemma 1. *A bit-vector of size n can be represented in $n + o(n)$ bits, supporting all aforementioned operations (access, rank, select) in constant time.*

Proof. Refer to Munro, Raman, and Rao [22]. □

2.1.2 Balanced Parenthesis Representation

The balanced parenthesis representation is a distinctive and efficient technique often used to encode nested structures, such as tree hierarchies. Its simplicity and versatility make it suitable for various applications, especially when combined with succinct data structures.

Definition 2.1.1. Depth-First Unary Degree Sequence (DFUDS) representation [30]: Given a sequence S of n characters, with each character either an opening parenthesis '(' or a closing parenthesis ')'. A sequence is deemed balanced if:

- Both opening and closing parentheses in S are of equal count.
- Any prefix of S has a count of opening parentheses that equals or exceeds that of closing parentheses.

Operations

The following operations are pivotal for managing and querying sequences of balanced parentheses:

- **FindClose(i):** For a position i marking an opening parenthesis, this operation returns the location of the related closing parenthesis.

- **FindOpen(i):** Starting from position i of a closing parenthesis, this seeks out the position of its corresponding opening parenthesis.
- **Enclose(i):** Given a position i, first identify its counterpart: if i points to an opening parenthesis, locate the corresponding closing parenthesis and vice versa, resulting in a pair of parentheses. Subsequently, for this pair, find the nearest enclosing set of matching parentheses.
- **Excess(i):** Returns the net difference in count between opening and closing parentheses up to the i^{th} position.
- **DoubleEnclose(x, y):** For two distinct parenthesis pairs, where the opening brackets are at positions x and y, this function finds, if present, the parenthesis pair that encloses both pairs with the least slack. This is synonymous with determining the least common ancestor in the tree equivalent.

Theorem 1. *All the aforementioned queries (FindClose, FindOpen, Enclose, Excess, DoubleEnclose) can be performed in constant time, consuming* $2n + o(n)$ *bits of space.*

Proof. See Theorem 1 in Munro and Raman [28]. □

2.1.3 Rooted Ordinal Trees Representation

A rooted ordered tree is a tree in which each node has a distinguished parent (except for the root, which has none) and a specific order among its children. The representation of rooted ordered trees is essential in various applications, such as XML document encoding and certain algorithmic constructs. A common approach to represent these trees is by transforming them into sequences of balanced parentheses.

Definition 2.1.2. Let T be a rooted ordered tree with n nodes. Each node in the tree is represented by two symbols: an opening parenthesis '(' when we first encounter the node (during a depth-first traversal) and a closing parenthesis ')' after visiting all its descendants. The resulting sequence of $2n$ balanced parentheses represents the tree structure of T.

This representation ensures that:

- The root of T corresponds to the outermost pair of parentheses.
- The children of a node in T are represented by the sequences enclosed within the pair of parentheses corresponding to that node.

Operations

Given the correspondence between balanced parentheses and rooted ordered trees, all operations on balanced parentheses, like 'FindClose(i)', 'FindOpen(i)', 'Enclose(i)', 'Excess(i)', and 'DoubleEnclose(x,y)', can be interpreted in the context of the tree T. For instance, 'FindOpen(i)' and 'FindClose(i)' can be used to find the opening and closing parentheses of a subtree rooted at a particular node, effectively allowing subtree extraction in constant time.

Theorem 2. *Given a rooted ordered tree with n nodes, represented using a balanced parenthesis structure, we can support the following operations in constant time using $2n + o(n)$ bits of space:*

- ***Parent(i):*** *Retrieve the parent of the node at position i.*
- ***Child(i, k):*** *Get the k^{th} child of node i.*
- ***LCA(i, j):*** *Determine the least common ancestor (LCA) of nodes i and j.*
- ***PreOrderNumber(i), PostOrderNumber(i):*** *Obtain the pre-order and post-order numbers of the node at position i.*
- ***SubtreeSize(i):*** *Determine the size of the subtree rooted at i.*
- ***Degree(i):*** *Calculate the number of children (degree) of node i.*
- ***Depth(i):*** *Get the depth of the node at position i from the root.*
- ***Height(i):*** *Obtain the height of the subtree rooted at node i.*

Proof. The results regarding the support of these queries in constant time using $2n + o(n)$ bits for rooted ordered trees are presented by Raman and Rao in their work [30]. We direct the reader to their paper for a comprehensive exposition of the methods and proofs. □

2.1.4 Succinct Representations of Binary Strings Supporting Rank

If the number of ones in a bit-vector is significantly less than its length, the bit-vector can be compressed. For a bit-vector of length n' with k' ones, this section demonstrates how to reduce its length to $o(n')$ while still supporting rank queries.

Definition 2.1.3. Following Lu [20], we introduce the notation $\tilde{O}(n', t')$. Note that this is not the conventional definition of $\tilde{O}$. Here, $\tilde{O}(n', t')$ is defined as $\tilde{O}(n', t') = O(t') \cdot O((\log \log n')^{O(1)})$.

Definition 2.1.4. Given a string X, the notation $X[i]$ is the i^{th} character in X.

Definition 2.1.5. Encoding Length: Given a string X, the notation $|X|$ represents the number of bits required to encode X.

Lemma 2. *Consider an n'-bit binary string X containing no more than k' one-bits. Encoding X into $Z(X)$ requires $O(n')$ time such that:*

$$|Z(X)| \leq \begin{cases} o(n') & \text{if } k' = o(n') \\ \min(n', k' \log \frac{n'}{k'}) + o(n') & \text{if } k' = \Omega(n') \end{cases}$$

Furthermore, each element $X[i]$ can be accessed from $Z(X)$ in $O(1)$ time. Additionally, for any i, the value of rank(X, i) is determinable from $Z(X)$ in $\tilde{O}(n', 1)$ time.

Proof. See Lemma 4 in Lu [20]. □

2.2 Fundamentals of Planar Graphs

Definition 2.2.1. Planar Graph: A graph G with n vertices and m edges is termed a **planar graph** if it can be embedded (drawn) in the plane such that no edges cross.

2.2.1 Faces

When a planar graph is drawn without any edges crossing, it divides the plane into regions called faces. The concept of faces is essential in planar graphs because it leads to various properties and theorems, such as Euler's formula.

2.2.2 Euler's Formula

For a connected planar graph with n vertices, m edges, and f faces, Euler's formula [19] states: $n - m + f = 2$. It is important to note that the cardinality set of faces f is independent of the specific planar embedding chosen. This formula is fundamental in the study of planar graphs and plays a significant role in a wide range of applications and proofs.

2.2.3 Properties of Planar Graphs

- A significant characteristic of planar graphs is the restriction on the number of edges they can contain in relation to their vertices. If a planar graph G has n vertices with $n \geq 3$, then the number of edges m is restricted by the inequality $m \leq 3n - 6$ [33].
- If G is a planar graph, it can be decomposed into three edge-disjoint forests [12].

2.2.4 Planar Embedding

A fundamental property of planar graphs is their ability to be embedded in the plane without any edge crossings. This means that the vertices of the graph can be represented as points in the plane, and the edges can be represented as non-intersecting curves connecting the corresponding points. Such an embedding clearly visualizes the planarity of a graph.

Definition 2.2.2. Planar Embedding: A **planar embedding** of a graph is a representation of the graph on a plane such that the edges of the graph intersect only at their respective endpoints. In other words, it is a visualization of a planar graph wherein no edges cross or overlap, except at the vertices where they are intended to meet.

One classic algorithm that achieves a planar embedding for a planar graph in linear time is the method proposed by John Hopcroft and Robert Tarjan in the 1970s ([14][21]). Their algorithm not only determines the planarity of a graph but, if the graph is planar, produces a planar embedding.

2.2.5 Importance in Computer Science

Planar graphs are studied extensively in computer science due to their importance in various applications, including circuit design, network design, and geographic information systems (GIS). The non-crossing property of edges ensures optimal utilization of space and minimizes complexity in specific computational problems.

2.3 Overview of the Shortest Path Problem

In the realm of shortest path problems, there are generally two distinct subquestions. One concerns finding the shortest path from a specific node to all other nodes in the graph, while the other seeks the shortest path between any two arbitrary nodes.

2.3.1 Shortest Path from a Node

When addressing the shortest path problem, one of the fundamental questions is determining the shortest path from a given node to all other nodes in the graph. In this context, a specialized structure called the "**shortest path tree**" becomes pivotal.

Definition 2.3.1. Shortest Path Tree (SPT): A shortest path tree of a node v in a connected graph G is a tree rooted at v that spans all nodes in G reachable from v and has the property that the path from v to any other node in the tree is the shortest path from v to that node in G. Essentially, it captures the shortest paths from the source node v to all other nodes.

BFS for Unweighted Graphs

In unweighted graphs, where all edges have the same weight (or, equivalently, no weight), the Breadth-First Search (BFS) algorithm is suitable for constructing the shortest path tree. BFS explores the graph in layers, ensuring that all nodes at distance d from the source node are explored before nodes at distance $d + 1$. A **BFS tree** is essentially a rooted tree that represents the structure of the graph as explored by the BFS algorithm. It is constructed by starting from a source node and exploring all its adjacent nodes at breadth, before moving to the nodes at the next level of breadth, and so on. Thus, when BFS completes, the resulting BFS tree rooted at the source node is, in fact, its shortest path tree. The BFS algorithm takes $\Theta(n + m)$ time complexity, where n is the number of nodes and m is the number of edges [8].

Dijkstra's Algorithm for Positive Weighted Graphs

For graphs with non-negative edge weights, Dijkstra's algorithm [10] is the go-to method. The algorithm maintains a set of explored nodes for which the shortest path from the source has been determined. In each iteration, it selects the node with the shortest tentative distance, explores its neighbors, and updates their tentative distances based on the sum of the current node's shortest distance and the weight of the edge connecting them. The algorithm iteratively refines the estimated shortest path values until the actual shortest paths are ascertained. When Dijkstra's algorithm terminates, the shortest paths tree can be reconstructed from the recorded predecessors of each node. Dijkstra's algorithm takes $\Theta(m + n \lg n)$ time complexity.

2.4 Compact Representation for Planar Graphs

2.4.1 OST: Orderly Spanning Tree

Given a rooted spanning tree, T, of the planar graph G and its planar embedding H, the nodes $v_1, v_2, ..., v_n$ represent the nodes in T traversed using a Depth-First Search (DFS) in a pre-order counterclockwise manner with respect to H.

In a Depth-First Search (DFS) on a tree, the pre-order traversal implies that a parent node is processed before any of its children. This ensures that the root of the tree is the first vertex to be considered. Consequently, in our scenario, the root corresponds to v_1. The essence of pre-order is that the parent of a node will always come earlier in the sequence than the node itself.

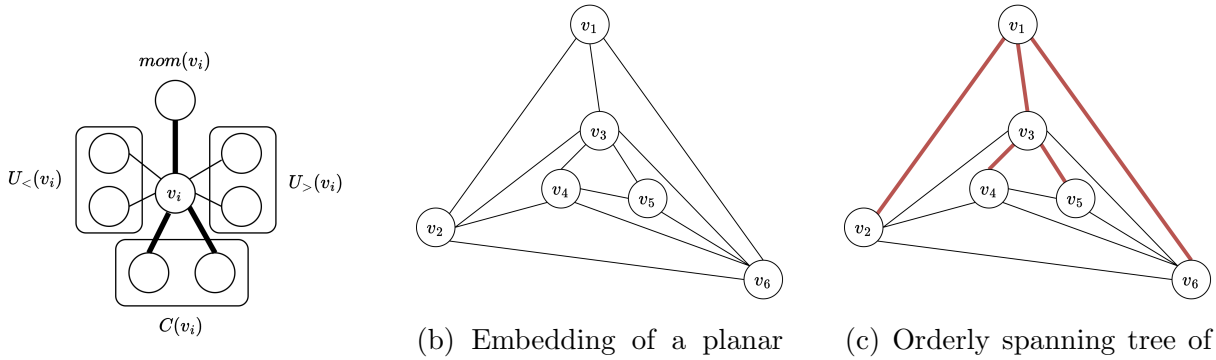
For the planar embedding H, as we conduct this pre-order DFS traversal, the children of any given node are processed in a counterclockwise direction around that parent node. This counterclockwise orientation respects the spatial arrangement defined by the planar embedding. In this traversal, the node v_i holds the i^{th} ordinal position.

Definition 2.4.1. Unrelated Nodes: Two nodes are **unrelated** if neither is an ancestor of the other in the tree T.

Definition 2.4.2. Orderly Node: A node, v_i, is **orderly** if the following four components maintain a counterclockwise orientation with respect to v_i:

- $\text{mom}(v_i)$: This denotes the parent of v_i within T.
- $U_<(v_i)$: These are the unrelated neighbors of v_i that have indices smaller than i.
- $C(v_i)$: Represents the children of v_i in T.
- $U_>(v_i)$: Refers to the unrelated neighbors of v_i with indices greater than i.

T is an **orderly spanning tree** if v_1 is on the boundary and every node v_i is orderly. To the best of our understanding, it was Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu who first introduced the concept of orderly spanning trees, also proposing a method to identify one in linear time [5].



(a) Orderly node

(b) Embedding of a planar graph

(c) Orderly spanning tree of Figure 2.1b

Figure 2.1: **Example of an Orderly Spanning Tree:** Figure 2.1a shows node v_i as an **orderly** node fulfilling orderly conditions. Figure 2.1b presents a sample of H, an embedding of G. Figure 2.1c displays an orderly spanning tree of Figure 2.1b, rooted at v_1, where bold red edges indicate the orderly spanning tree and other edges are excluded.

2.5 Planar Graph Representation using Orderly Spanning Tree

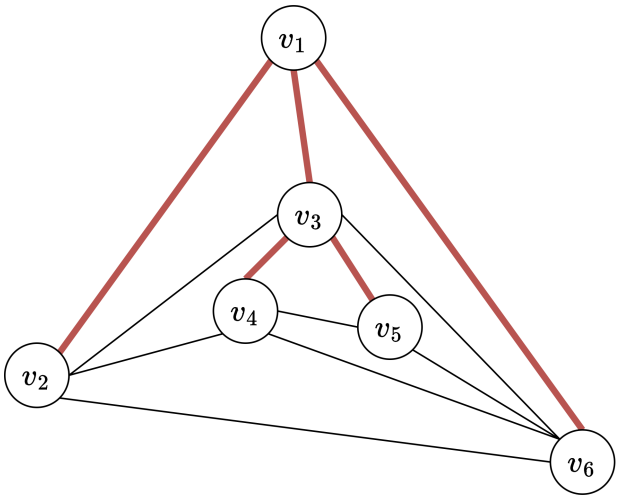
Given T as an orderly spanning tree of G, Lu, in 2010, presented a method that represents G based on T and occupies $2n + 2m + o(n)$ bits [20]. Notably, as m is at most $3n - 6$, the representation is at most $8n + o(n)$ bits. We'll explore this representation technique in the following section.

The nodes $v_1, v_2, ..., v_n$ have been predefined, representing nodes in T traversed by a Depth-First Search (DFS) in a pre-order counterclockwise sequence concerning H.

We will introduce a sequence, P, that represents the planar graph. Subsequently, we optimize storage space for P by segregating it into two separate sequences, namely S' and S''.

Definition 2.5.1. Sequence P consists of brackets and parentheses with a total length of $2m + 2$. Within this sequence, there are $2n$ parentheses and $2m - 2n + 2$ brackets. A noteworthy feature is the balanced nature of both the parentheses and brackets in P (The notion of balanced parentheses is elucidated in the background chapter, and balanced brackets adhere to the same principles). In P, the significance of parentheses and brackets is as follows:

- **For Tree Edges in T:** The parentheses in P offer a parenthetical representation



Description:
Orderly spanning tree of Figure 2.1b. This graph serves as an example of H, which is an embedding of a planar graph G. In this example, the edges in T are represented by red bold lines.

Values in this example:
P: $(_1(_2)_2[_1[_2[_3(_3]_3(_4]_2)_4[_4[_5(_5]_5)_5[_6)_3[_7(_6]_7]_6]_4]_1)_6)_1$
S': $(_1(_2)_2(_3(_4)_4(_5)_5)_3(_6)_6)_1$
S'': 11100010101001010101000011

of T. Every corresponding pair of parentheses in P denotes a node in G. For any given i in the range 1 to n, $(_i$ symbolizes the i^{th} opening parenthesis in P, while $)_i$ represents the matching closing parenthesis. The pairing $(_i$ and $)_i$ equates to node v_i. Additionally, node v_i is an ancestor of v_j in T if and only if $(_i$ and $)_i$ envelop $(_j$ and $)_j$ in P.

- **For Graph Edges Outside T:** Every matching pair of brackets in P signifies an edge not part of T but in G. For each index k from 1 through $m-n+1$, $[_k$ is the k^{th} opening bracket in P and $]_k$ is its corresponding closing bracket. The pair $[_k$ and $]_k$ relates to the edge $e_k = (v_i, v_j)$, where $)_i$ (and analogously $(_j$) is the last parenthesis before $[_k$ (or $]_k$ respectively) in P. This index k is dubbed the identifier for the edge (v_i, v_j) and is denoted as $\mathrm{id}(v_i, v_j) = k$.

To explicitly represent P, one would require $4m + O(1)$ bits. This is deduced from the fact that P has a length of $2m + O(1)$ and is composed of four unique symbols. Given that the orientation of a bracket can be inferred from its immediate preceding parenthesis, P can be transcribed into two sequences: S' and S'', where the combined length of $|S'| + |S''|$ equals $2m + 2n + O(1)$:

- S' has a span of $2n$ bits, indicating the status (open or closed) of each parenthesis in the set of $2n$.
- S'' extends for $2m + O(1)$ bits, designating whether a given element $P[i]$ is a parenthesis or a bracket.

Lemma 3. *Consider a planar graph $G(V, E)$ with n vertices and m edges. Using $2n + 2m + o(n)$ bits, which are computable in $O(n)$ time, the following operations can be executed in constant time for any given indices i and j. This efficiency is a result of utilizing an arbitrary Orderly Spanning Tree, T, taken from an arbitrary planar embedding of G, referred to as H:*

- *Determine if nodes v_i and v_j are adjacent in G;*
- *Ascertain if nodes v_i and v_j are adjacent in T;*
- *Find $id(v_i, v_j)$ for any edge (v_i, v_j) not in T;*
- *Compute $nbr(v_i, j)$: Where $nbr(v_1, j)$ represents the j^{th} neighbor of v_1 in G based on counterclockwise orientation around v_1, starting from v_2. If $i > 1$, $nbr(v_i, j)$ designates the j^{th} neighbor of v_i in G, taking a counterclockwise approach around v_i beginning from $mom(v_i)$;*
- *Identify the number of neighbors of v_i in $U < (v_i)$, $C(v_i)$, and $U > (v_i)$;*
- *Determine the j^{th} neighbor of v_i in $U < (v_i)$ based on counterclockwise orientation around v_i;*
- *Determine the j^{th} neighbor of v_i in $U > (v_i)$, considering a counterclockwise direction around v_i.*
- *$C(v_i, j)$: Determine the j^{th} child of v_i in tree T, when traversing in a counterclockwise direction around v_i.*

Proof. See Lemma 5 in Lu [20]. □

2.6 Pair Cache Problem

Introduced in Khodaee's master's thesis [17], the **Pair Cache Problem** addresses the challenges of managing a limited-size cache while sequentially processing a series of queries. Each query is characterized by a pair of pages, denoted as (a, b), seeking an element found within both pages a and b. Here, a "**page**" refers to a specific segment of memory. The query is considered a "**hit**" if either page a or b is already present in the cache. Conversely, if both are absent, it results in a "**miss**." In the event of a miss, the **FPIFO** (First Pair

In First Out) algorithm — proposed by Khodaee [17] — retrieves both pages. If the cache is full, it removes the pair of pages retrieved at the earliest before retrieving the new ones. Being an online algorithm, FPIFO handles each query on an individual basis, devoid of any insights into subsequent queries.

In Theorem 4 of Khodaee's Master's thesis, it is elucidated that with the FPIFO algorithm, given a cache size that is four times larger, we can handle any sequence of queries with no more than four times the number of retrievals as compared to the optimal approach.

We can adapt the Pair Cache Problem to our center-based configuration. Specifically, if we interpret each page in the cache as a representation of the shortest path tree for a node, then with n nodes in our graph, we have n corresponding pages. Every query (a, b) represents a shortest path query from node a to node b. This path can be found within the shortest path trees of both nodes v_a and v_b. In this analogy, the cache's size limitation corresponds to the maximum number of stored vertices, and the cache itself translates to these stored vertices. Thus, having the representation of a node's shortest path tree in our data structure equates to that node being one of the stored vertices, mirroring the idea of having a page stored in the cache.

Theorem 3. *Given an algorithm that starts with an empty set of stored vertices, if the following conditions hold:*

1. *The maximum number of stored vertices the algorithm uses does not exceed t.*
2. *For any shortest path query (a, b), it requires at least one of a or b to be in the stored vertices.*

Then, if the number of retrievals made by this algorithm is r', the FPIFO algorithm — which operates without prior knowledge about which nodes are centers at any given time and permits up to $4t$ stored vertices — will make at most $4r'$ retrievals.

Proof. The proof of this theorem leverages Theorem 4 of Khodaee's master's thesis combined with the preceding adaptation. □

2.6.1 Network Fundamentals

End systems interconnect through a network of communication links and packet switches. When an end system has data destined for another, it segments this data, appending

header bytes to every segment. These segments, commonly known as **packets** in the realm of computer networking, travel across the network. Once they arrive at the target end system, they are pieced together to recreate the original data [18]. **Routing tables** provide mappings of destinations to their **next-hop** addresses, where the **next-hop** refers to the subsequent immediate destination a packet should be sent to on its path toward its final endpoint. This ensures that packets take optimal or feasible paths. The underlying data structures, whether they be trees, hash tables, or trie structures, play a crucial role in how swiftly routing decisions are made and how efficiently the tables are stored and updated. Additionally, notations such as T_v, R_v, and deg_v, often signify specific network entities or metrics, like routing spanning trees rooted at node v, routing tables for node v, or the degree of node v in a tree, respectively.

In the context of a connected, unlabeled, planar network G comprising n nodes represented by the set V, our focus is on developing compact routing tables. The key elements of this discussion are defined below.

Definition 2.6.1. Routing Spanning Tree T_v for Node v: Rooted at node v, this tree outlines the paths for transmitting packets from v to other nodes within network G. Alternatively, T_v can represent the shortest path tree with its root at v in a weighted version of G. Considering vertices x, u, and v in G, consistency among the routing spanning trees implies that if x is situated on the path of T_v between v and u, then the path defined by T_x from x to u is identical to that of T_v from x to u.

Definition 2.6.2. port(u, v): This represents the vertex after u in the path from u to v within T_u. It can be defined as:

- If v is a neighbor of u in T_u, then $port(u, v) = v$.
- If v is not a neighbor of u in T_u, $port(u, v)$ is given by vertex x. Here, x is a neighbor of u in T_u, and the removal of x disconnects u and v in T_u.

We can also denote $port(u, v)$ as $port_u(v)$. The maximum number of possibilities for $port_u(v)$ is deg_u, with deg_u representing the degree of u in T_u. This is because for each pair u and v, $port_u(v)$ is always a neighbor of u in T_u.

Definition 2.6.3. Let N_v be defined such that:

$$N_v = \{u \in V : u \text{ is a neighbor of } v \text{ in } T_v\}$$

Consequently:

$$port_v : V \to N_v$$

Definition 2.6.4. Routing Table R_v for Node v: A table associated with each node v in the network graph G. The primary purpose of R_v is to determine '$port_v(u)$' based solely on R_v and the label of u. The design objectives for this table, as outlined in [20], are:

- Minimize the number of bits, $\lambda_v(n)$, for the label of node v.
- Minimize the computation time, $\tau_v(n)$, needed to fetch '$port_v(u)$' using R_v and the label of u.
- Minimize the time, $\pi_v(n)$, required to derive R_v from G.
- Minimize the number of bits, $\rho_v(n)$, to represent R_v.
- Minimize the overall representation length, denoted by $\sum_{v \in V} p_v(n)$.

Through this framework, the objective is to institute a proficient routing mechanism within a planar network, optimizing both route identification and resource utilization.

Chapter 3

Related Work

3.1 Space-Efficient Data Structures

While the development of **space-efficient data structures** has been a continual process in the field of computer science, a significant milestone in formalizing this concept was marked by Munro and Suwanda about 45 years ago with their introduction of the **implicit data structure** [26]. These structures are termed "*implicit*" due to the meaningful positions of the elements, establishing relationships between them, a stark contrast to the explicit relationships formed through the use of pointers. Essentially, the data structure represents a permutation of the object values. Williams' heap [34] is an example, supporting a priority queue, that preceded the definition, as, indeed, is a sorted array. Most of the work has focussed on supporting the "dictionary" data type, with operations insert, delete, and find. Munro and Suwanda [26] provided a $\Theta(\sqrt{s})$ technique, where s represents the number of objects, proving its optimality when searches are constrained to comparisons between the sought value and the structure's values. Nonetheless, allowing comparisons between value pairs inside the structure, under the assumption of distinct values, enables support for operations in polylogarithmic time, and ultimately in $O(\log s)$ time, albeit with a substantial constant factor [23].

The **succinct data structure** concept revolves around representing a combinatorial object with a bit count nearing the information-theoretic lower bound, plus an additional smaller term. This lower bound roughly equates to the logarithm (base 2) of the number of such objects. Jacobson, in 1989, coined this term and demonstrated how to efficiently represent a binary tree using $2n' + o(n')$ bits, where n' denotes the number of nodes [15]. This representation facilitates finding children and parents within $O(\lg n')$ bit probes per

operation. Given that there are $\frac{\binom{2n'}{n'}}{n'+1}$ binary trees with n' nodes, a representation requiring about $2n'$ bits becomes crucial. Following this, Clark and Munro [6] presented a technique to replace the $O(\lg n')$ bit probe with a constant number of probes of $\lg n'$ consecutive bits, leading to constant time queries and significantly improving query efficiency. In 1996, Munro [24] introduced a succinct data structure designed for storing ordered rooted trees, while ensuring that fundamental search and navigation operations could be executed in constant time. In 2004, Munro and Rao [25] released a comprehensive survey covering the succinct representation of various data structures.

Compact data structures [29] and **succinct data structures** are both designed to efficiently utilize memory, yet they differ in their approach to space utilization. Compact data structures use O(OPT) bits of space, where OPT represents the information-theoretic lower bound for storing a particular object, ensuring space efficiency within the same order of magnitude as the optimal. Succinct data structures, however, strive for an even tighter space usage, employing OPT + o(OPT) bits. This means that they use the absolute minimum required space plus an additional smaller term, achieving a representation that is extremely close to the theoretical lower bound and making them more space-efficient compared to compact data structures.

3.2 Compact Representation of Planar Graphs

The field of space-efficient graph representation has witnessed substantial progress over the past decades, contributing significantly to data storage and computational efficiency. A noteworthy comprehensive overview of these advancements is provided in the 2019 survey by Besta and Hoefler [2], which offers valuable comparisons across various algorithms and includes a detailed comparative analysis in Table 3 of data structures used in representing planar and planar-like graphs, like Outerplanar and Plane triangulation.

Dating back to 1970, Bonichon et al. [3] introduced the use of well-orderly maps—a special form of planar embedding—to propose a theoretical upper bound of $4.91n$ bits or $2.82m$ bits for representing a planar graph. Fourteen years later, in 1984, Turán [32] presented a method requiring $12n + o(n)$ bits to store planar graphs; however, this method did not support navigational queries efficiently. In a significant stride towards efficiency, Keeler and Westbrook, in 1995 [16], reduced the memory requirement for storing planar graphs to $3.58m + o(n)$ bits, effectively bringing it down to at most $10.74n + o(n)$ bits, a considerable improvement given that in planar graphs, m is less than $3n$.

In 1997, Munro and Raman [28] further pushed the boundaries by leveraging a 4-page

Paper	Space ($+o(n)$) bits	Nav. queries	Find port
Turán [32]	$12n$	No	No
Keeler and Westbrookc [16]	$3.58m$	No	No
Munro and Raman [28]	$8n + 2m$	$O(1)$	No
Chiang et al. [5]	$2n + 2m$	$O(1)$	No
Aleardi et al. [4]	3-connected: $2m$	$O(1)$	No
Aleardi et al. [4]	Triangulated: $3.24n$	$O(1)$	No
Gavoille and Hanusse [11]	$8n$ per node	$O(1)$	$O(lg^{2+eps}n)$
Lu [20]	$7.181n$ per node	$O(1)$	$O(lg^{1+eps}n)$

Table 3.1: **Planar Graph Representation Methods:** This table offers an overview of different methods for planar graph representation. It highlights their space consumption, efficiency in handling basic navigational queries (like degree and adjacency), and the time needed to identify a port from a given source node.

decomposition of planar graphs to propose a pioneering representation. This method not only reduced the space required to merely $8n + 2m + o(n)$ bits, or at most $14n + o(n)$ bits, but also facilitated constant-time degree and adjacency queries, enhancing the query efficiency significantly. Following this, in 2002, Chiang et al. [5] introduced the Orderly Spanning Tree—a linear-time computable spanning tree with distinctive attributes—that enabled a representation method for planar graphs using just $2n + 2m + o(n)$ bits. The resulting data structure was proficient at executing navigation queries, including degree and adjacency, in constant time.

Finally, in 2008, Aleardi et al. [4] achieved a milestone in memory reduction for specific categories of planar graphs: 3-connected and triangulated graphs. By decomposing the graph into smaller segments, they managed to reduce the memory usage to $2m + o(m)$ and $3.24n + o(n)$ bits, respectively. This innovation marked a significant achievement in the ongoing journey to optimize space-efficient representations of planar graphs.

3.3 Compact Representation of Graphs Supporting Routing Queries

Various researchers have explored compact data structures designed to efficiently handle shortest path queries, such as finding the shortest path and distance oracles, across diverse graph types. In 2020, He et al. [13] introduced the inaugural succinct distance oracles

for interval graphs and associated graph types, leveraging an innovative succinct data structure for ordinal trees. This structure enables constant-time mapping between preorder (depth-first) and level-order (breadth-first) ranks of nodes. Additionally, their distance oracles for interval graphs provide efficient support for various navigation queries including adjacency testing, node degree computation, neighborhood identification, and shortest path determination, all executed in optimal time. In 2021, Balakrishnan et al. [1] tackled distance queries on path graphs, noting that these are a superclass of interval graphs. In 2018, Munro and Wu [27] developed a method to approximate distances in chordal graphs (which encompass path graphs as a subclass) with constant time queries, utilizing only $n \lg n + o(n \lg n)$ bits of space.

In 2022, Das et al. [9] introduced an innovative data structure for efficiently finding shortest paths in unweighted interval graphs, where the paths are required to pass through at least one of a designated subset of nodes, termed '**beer vertices**'. These paths, known as '**beer paths**', are distinct in that they do not necessarily have to be simple paths. The data structure achieves space efficiency, requiring $2n \lg n + O(n) + O(|B| \lg n)$ bits for interval graphs and $3n + o(n)$ bits for proper interval graphs, where $|B|$ denotes the number of beer vertices. The structure ensures quick response times for beer distance and shortest beer path queries, while also establishing near-optimal space lower bounds. Notably, the problem tackled in this work bears similarities to the 'center-based configuration' concept explored in our thesis, with the 'beer vertices' playing a critical role comparable to centers, influencing paths and distances throughout the graph.

3.4 Compact Planar Graph Representation for Routing Queries

Our discussion now turns to techniques that are more aligned with our research focus, specifically those that facilitate queries related to the shortest paths in planar graphs. In 1999, Gavoille and Hanusse [11] proposed a method to determine the next vertex (or port) in the routing path within planar graphs. They accomplished this with a time complexity of $O(\log^{2+\epsilon} n)$ where $\epsilon > 0$, utilizing routing tables that occupy $8n + o(n)$ bits per node. In 2010, Lu [20] introduced an approach utilizing orderly spanning trees to diminish the size of routing tables in planar graphs. This method reduced the routing table size to $7.181n + o(n)$ bits per node while ensuring the port query could be supported in $O(\log(1+\epsilon)n)$ time for any constant $\epsilon > 0$. The total memory requirement for this approach stands at $7n^2 + o(n^2)$ bits.

Chapter 4

Methods

In this chapter, we present our contributions, which are organized into three sections, each addressing a different type of query. The primary focus of our work, detailed in Section 4.1, is the center-based configuration. Here, we define this configuration and describe five scenarios within it. For each scenario, we provide a solution for the shortest path query. In Section 4.2, our contribution is a data structure capable of responding to distance oracle queries in both center-based and non-configured settings. Finally, in Section 4.3, we introduce a space-efficient data structure designed to manage port queries. Although our primary contributions pertain to the center-based configuration, we also include results applicable to non-configured settings.

4.1 Center-Based Configuration

In this section, we delve into the analysis of the shortest path problem within the context of the center-based configuration.

The significance of the center-based configuration is underscored by its relevance and applicability to real-world scenarios. Consider, for instance, urban routing challenges. While a city might comprise numerous nodes representing intersections or landmarks, there exists a subset of these nodes that are markedly more frequented—being common origins or destinations for most trips. The center-based configuration adeptly handles these scenarios, managing routing problems in a manner that is both more space-efficient and time-efficient than alternative methods.

In this chapter, we will explore five distinct scenarios within the context of the center-based configuration of connected planar graphs. In the subsequent scenarios, we have q queries, each represented by a pair of nodes. The i^{th} query, denoted by q_i, is represented by a pair of vertices (a_i, b_i). For each query, we are required to determine the shortest path from a_i to b_i. Given that the graph is undirected, this path is equivalently the shortest path from b_i to a_i. Since the planar graph G is connected, there always exists at least one shortest path between a_i and b_i. It is important to note that the queries are online, meaning we do not gain knowledge of the subsequent query until the current one is addressed.

4.1.1 Known Centers with Complete Association

In the scenario titled "**Known Centers with Complete Association**," we are given k vertices termed "centers". In this scenario, we know the exact identities of the nodes that are in the centers. For every query index i ranging from 1 to q, at least one of the vertices a_i or b_i is a center. Without loss of generality, we can assume that a_i is a center. This implies that for every query, its source node is a member of the centers. Initially, we operate under the premise that all edge weights are set to 1. Subsequently, we explore situations where edge weights are positive, signifying they can be any positive value.

Given possession of all Breadth First Search (BFS) trees originating from the centers, we can accommodate any query (a_i, b_i) for i spanning from 1 to q within a complexity of $\Theta(\text{SPL}(a_i, b_i))$. This complexity is optimal as the length of the solution is similarly $\Theta(\text{SPL}(a_i, b_i))$. If we store the BFS trees in the form of an adjacency list, each BFS tree would necessitate a minimum of $n \times \lg(n) + o(n \times \lg(n))$ bits. Therefore, the aggregate space demand is not less than $nk \times \lg(n) + o(nk \times \lg(n))$. A critical observation is that we are unable to retain all BFS trees utilizing the succinct ordinal tree representation described in Section 2.1.3. This is due to the fact that each BFS tree has its unique vertex ordering. Consequently, the i^{th} node in one BFS tree would not match the i^{th} node in another. To reconcile this, we would be compelled to allocate extra storage to discern vertex orderings for every BFS tree, which would require at least $n \times \lg(n)$ bits for each BFS tree. In upcoming sections, we introduce a more proficient method that surpasses the previously mentioned techniques.

We propose a data structure occupying $3n + 2m + 2km + o(nk)$ bits with a construction time of $O(nk)$. This structure can efficiently answer each query (a_i, b_i) in $\Theta(\text{SPL}(a_i, b_i))$.

Consider the vertices in centers as $x_1, x_2, \ldots, x_k$. For each vertex x, we construct its BFS tree in $O(n)$ time. Denote the BFS tree of x_i as BFS_i for each i from 1 to k.

Let T represent an arbitrary Orderly Spanning Tree of H, which is a planar embedding of $G(V, E)$. Both T and H can be obtained in $O(n)$ time [5]. According to Lemma 3, the planar graph G can be represented using $2n + 2m + o(n)$ bits. To store each BFS_i for i ranging from 1 to k, we use T to represent BFS_i with $2m + o(n)$ bits, which ensures constant-time processing for the parent query in BFS_i. To determine the parent of a node in BFS_i, the vertex's preorder in T must be known. The shared vertex ordering across all BFS_i and T is crucial, as it enables consistent referencing. While this paragraph provides a general overview, the detailed explanations of our approach, storage methods, and query processing will be discussed in subsequent sections.

Definition 4.1.1. Let v_i denote the i^{th} vertex in the preorder of T.

Definition 4.1.2. Parent Query of T': For any arbitrary rooted spanning tree T' of G, a parent query is denoted as $\textbf{parent}(T', i)$ and is defined as follows: given an index i (with i ranging from 1 to n), determine the index corresponding to the parent of v_i in T'. Formally, if the parent of v_i is v_j in T', then $\text{parent}(T', i)$ returns j. For the root of T', $\text{parent}(T', i)$ returns null, indicating no parent.

Refer to definition 2.5.1 for a comprehensive definition of P. P is a sequence of length $2m+2$, containing $2n$ parentheses and $2m-2n+2$ brackets. Subsequently, we will introduce new functions related to P.

Definition 4.1.3. rp(i): For any given index i, $rp(i)$ denotes the position of the rightmost parenthesis that comes before the i^{th} bracket in P.

Definition 4.1.4. rv(i): For any given index i, $rv(i)$ is the index of vertex associated with $rp(i)$.

Example 4.1.1. *Consider the following example for better clarity:*

- *Given P as: '()[()]'*
- *For the bracket(1) in red in '()[()]':*
  - *$rv(1)$ is equal to 1, as the parenthesis in red (shown in '()[()]') pertains to v_1.*
- *For the bracket(2) in red in '()[()]':*
  - *$rv(2)$ is equal to 2, as the parenthesis in red (shown in '()[()]') pertains to v_2.*

Lemma 4. *For each i from 1 to n, $rv(i)$ can be obtained in constant time from S' and S'' (see Definition 2.5.1).*

Proof. Apply a rank$(i, 1)$ query in S'' to find the number of parentheses before position i in P. The rank$(i, 1)$ in S'' represents the position of the parenthesis in S'. If that parenthesis is a closed parenthesis, find its corresponding open parenthesis in S' in constant time (see Theorem 1). Upon identifying an open parenthesis, apply a rank query to determine the number of open parentheses before it. This process allows the corresponding vertex to be identified in constant time. Thus, $rv(i)$ can be efficiently determined for any i from 1 to n. □

Lemma 5. *Let $G(V, E)$ be a connected undirected planar graph with n vertices and m edges. Consider H to be an arbitrary planar embedding of G and T as an arbitrary Orderly Spanning Tree of H, represented by sequences S' and S'' (see Definition 2.5.1). Define v_i as the i^{th} element in the preorder of T. For any rooted spanning tree T' of G, there exists a data structure, which can be constructed in $O(n)$ time, that utilizes $2m + o(n)$ bits and is capable of responding to parent(T', i) queries in constant time for any i ranging from 1 to n.*

Proof. To handle the parent(T', i) query in constant time, we introduce two bit-vectors: A and B.

- A has a fixed length of $2(m - n + 1)$, while
- B, being flexible, can have a length up to $2n$.

If the edge leading from a node to its parent in T', denoted as e, lies in $G - T$ (the graph G excluding the edges in T), then its parent can be found in constant time using A. Conversely, if the edge from a node to its parent in T' is within T, identification of this edge occurs in constant time using B. The answer is not retrievable in A if and only if the edge e is in T.

Definition 4.1.5. Bit-vector A: For each bracket in P (see Definition 2.5.1), A has a corresponding bit. Thus, its total length is $2(m-n+1)$. Let bracket$_i$ denote the i^{th} bracket in P. The i^{th} element in A is set to 1 if and only if the edge associated with bracket$_i$ lies between rv(i) and parent$(T', \text{rv}(i))$. Therefore, the count of 1s in A indicates the number of edges present in T' but absent in T.

Definition 4.1.6. Bit-vector B: B is a flexible bit-vector with a maximum possible length of $2n$. To construct B, it is initiated with n 'zeros', where the i^{th} 'zero' in B corresponds to v_i. Prior to the i^{th} 'zero' in B, we append:

- Nothing, if the edge from the i^{th} node to its parent in T' is the same as the edge to its parent in T. This is the case when $\text{parent}(T, i) = \text{parent}(T', i)$.
- j ones, if the edge from the i^{th} node to its parent in T' is the same as the edge from v_i to its j^{th} child in T, following a counterclockwise order (consistent with the child order in the ordinal tree T).

The size of B is constrained to a maximum of $2n$. This is because B inherently contains n 'zeros', and for each node, the number of ones added is at most its number of children in T. As per the lemma in the appendix regarding the total number of children in a tree (see Lemma 7), the aggregate number of children across all nodes of T will not exceed n given that there are $n-1$ edges in the tree, and each edge contributes to one child. Therefore, considering both the 'ones' and 'zeros', B will possess a size of at most $2n$.

It is essential to remember that the final bit in B is invariably a 'zero'. Leveraging this n^{th} 'zero', we can ascertain the length of B in constant time. More precisely, one can invoke the $\text{Select}(n, 0)$ operation to pinpoint the position of the last 'zero' in B. When B is concatenated after A, we can discern that the initial $2(m-n+1)$ bits are derived from A. Moreover, to determine the number of 'zeros' in A, one can execute $\text{rank}(2(m-n+1), 0)$. If this rank is equated to a value s, then by calling $\text{Select}(n+s, 0)$ on the concatenated bit-vector of A and B, we can deduce the end position of the merged A and B.

In order to address the $\text{parent}(T', i)$ query, we will leverage the method established by Lu [20], which has demonstrated that by maintaining S', S'', and an additional $o(n)$ bits, we can perform the access, rank, and select queries on P. Consequently, our discussion henceforth will center around P rather than S' and S''.

To initiate, we must ascertain whether v_i serves as the root of T'. (An allocation of $o(n)$ additional space facilitates this evaluation.) In the event v_i is the root of T', it inherently lacks a parent in T'. Thus, our subsequent approach presumes v_i is not the root of T'.

Let e denote the edge leading from v_i to its parent in T'. First, we conduct a constant-time search using A. If e is in $G-T$, then we can identify v_i's parent in constant time. If the result is not found in A, we then conduct a constant-time search using B. If e is in T, we can again identify v_i's parent in constant time. The procedure can be delineated as follows:

1. If e is in $G-T$:
   (a) Identify two bracket ranges in P corresponding to edges from v_i to its neighbors in $G-T$:

i. Define the first range (l_1, r_1) from the i^{th} open parenthesis to the next parenthesis (open or closed), and the second range (l_2, r_2) from the i^{th} closed parenthesis to the subsequent parenthesis. In this context, "between" means excluding the boundaries, and only brackets can exist between l_1 and r_1, and similarly between l_2 and r_2.

ii. Locate the i^{th} open parenthesis using S'. Represent open parentheses by bit 1 and closed parentheses by bit 0 in S', and apply select$(i, 1)$ on S'. Let the resulting position be z.

iii. Determine the bracket range corresponding to this position by finding the z^{th} and $(z+1)^{th}$ bit 1 by applying a select query in S''. This process establishes the boundaries in P, where the z^{th} 1 in S'' represents l_1 and the $(z+1)^{th}$ 1 in S'' represents r_1.

iv. To find the range starting from the i^{th} closed parenthesis, apply a similar method but use select$(i, 0)$ on S' to locate the i^{th} closed parenthesis.

(b) Find the position of the 'one' in A corresponding to the positions of both ranges (l_1, r_1) and (l_2, r_2) in P:

i. For each range (l_t, r_t) where $t \in \{1, 2\}$, determine the corresponding position in A. Apply rank$(l_t + 1, 0)$ on S'' to find the start of range t in A, and rank$(r_t - 1, 0)$ to find the end. Note that if $l_t + 1 = r_t$, indicating no brackets in the range, skip this and the subsequent steps for this range. Otherwise, if $l_t + 1 < r_t$, a corresponding range exists in A, denoted as $[l'_t, r'_t]$, which includes the range boundary.

ii. For each $t \in \{1, 2\}$, perform rank$(l'_t - 1, 1)$ on A to find the number of 'ones' before position l'_t in A. If l'_t is the first element in A, the number of 'ones' before it is 0. Let this count be z'. Then, execute select$(z' + 1, 1)$ in A to locate the next position containing a 'one' after $l'_t - 1$ in A. If this 'one' is not beyond r'_t, it indicates a 'one' within the range $[l'_t, r'_t]$. Denote this position as z''.

(c) If no 'one' is found in both ranges, it implies $e \in T$. Otherwise, $e \in G - T$.

(d) The position of the z''th bracket in P can be determined. If this bracket is an open (or closed) bracket, its corresponding closed (or open) bracket can be found in constant time. Let this be the j^{th} bracket in P. The rv(j) can also be determined in constant time (see Lemmad 4), and rv(j) represents the parent of i in T'.

2. If e is in T:

(a) In the absence of any ones within both bracket ranges, e is confirmed not to be part of $G - T$, indicating its presence in T. In this case, parent(T', i) is determined using B.

(b) This determination involves finding the i^{th} 'zero' in B, which represents v_i, and if $i > 1$, also the $(i-1)^{th}$ 'zero'.

- If $i > 1$, apply select$(i-1, 0)$ and select$(i, 0)$ in B to find the range of 'ones' between the $(i-1)^{th}$ and i^{th} 'zeros'. Let the count of ones in this range be c'.
- If $i = 1$, apply select$(i, 0)$ in B to find the range of 'ones' before the first 'zero'. Again, let c' be the count of ones in this range.

(c) The parent of v_i in T' is determined using c':

- If $c' = 0$, given e's placement in T, it is inferred that e connects v_i to its parent in T. The parent's index in T is quickly derived from S'.
- If $c' > 0$, it is concluded that e connects v_i to its c'^{th} child in T. Lemma 3 details how this c'^{th} child in T is identified in constant time.

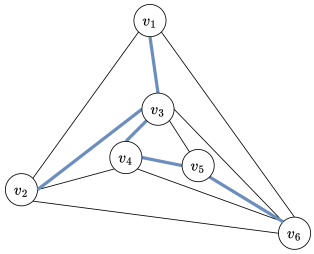
Thus, the parent(T', i) query can be addressed in constant time regardless of whether e is a member of T or not.

In summary, as demonstrated, utilizing vectors A and B alongside an additional $o(n)$ bits enables efficient handling of the parent(T', i) query in constant time. Furthermore, the outlined procedure for constructing both A and B ensures their formation is in $O(m)$, which is $O(n)$ in a planar graph. □

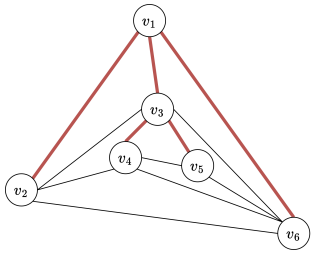
Given Lemma 5, and the representations S' and S'' of G utilizing T, we can formulate a data structure for each BFS$_i$ where i spans from 1 to k. This structure supports the parent query in BFS$_i$ in constant time based on T, and requires $2m + o(n)$ bits. By the specifications of Lemma 5, for each BFS$_i$ (with i ranging from 1 to k), we possess bit-vectors A_i and B_i. Concatenating A_i and B_i yields C_i. If the length of C_i is less than $2m$, pad C_i with 'zeros' until its length matches $2m$.

Considering the space requirements, S' and S'' together demand $2n + 2m + o(n)$ bits. To support the parent query in each BFS$_i$ (with i ranging from 1 to k), storing each C_i along with an extra $o(n)$ space for each becomes essential. The total space required here is $2km + o(nk)$ bits.

Additionally, a distinct bit-vector of size n, termed KB (centers bit-vector), is needed. The i^{th} entry of KB equals 'one' if and only if v_i is among the centers. Knowing the exact nodes within the centers allows for the initial formulation of KB, requiring $n + o(n)$ bits.



(a) T' rooted at v_3



(b) T: Orderly Spanning Tree

Description:
The red bold edges in 4.1b represent the orderly spanning tree T, rooted at v_1. The blue bold edges in 4.1a represent the tree T', rooted at v_3.
Values for this example:
A: 00100001000100
B: 11000000

Figure 4.1: An example illustrating T and T'

In summary, the total space requirement for this scenario is $3n + 2m + 2mk + o(nk)$ bits. Here, the centers themselves serve as the "Stored Vertices." As a result, the "Maximum Number of Stored Vertices" is limited to k. Importantly, retrieve operations are unnecessary in this setup. This is because all the shortest path trees have already been constructed during the preprocessing stage, negating the need for additional retrievals.

The sequences C_i for i ranging from 1 to k can be stored in two distinct ways: a centralized approach and a distributed approach. Let us first discuss the centralized storage method, followed by the distributed method.

Centralized Storage

In the centralized storage approach, all C_i values, where i ranges from 1 to k, are concatenated together. The sequence of concatenation follows the order dictated by the 'ones' in the KB vector. Specifically, the foremost C corresponds to the initial 'one' in KB. This implies it represents the BFS tree of the vertex within the center that possesses the smallest index.

Given a query (a_i, b_i), where both a_i and b_i denote indices of vertices in T based on

a counterclockwise preorder traversal, the initial step involves checking the KB bit-vector to discern which among v_{a_i} or v_{b_i} resides within the centers. For the sake of clarity and without loss of generality, assume that v_{a_i} is a member of the centers.

By invoking a rank query on the KB bit-vector, we can determine the position of a_i within the centers. This reveals the count of vertices v_j that belong to centers and satisfy the condition $j \leq a_i$. Let this count be denoted by z. Given this, we can infer that the C corresponding to the BFS tree rooted at v_{a_i} commences at the position $(z-1) \times 2m$ and culminates at $z \times 2m - 1$.

With this alignment set, one can efficiently resolve the parent query for b_i within this structure in constant time. This procedure should be iteratively executed until the traversal reaches the designated root, which in this context is a_i. Implementing this strategy, we can deduce the shortest path between b_i and a_i in time linear to their distance in the graph, specifically $\Theta(\text{SPL}(a_i, b_i))$.

Distributed Storage

The sequences C_i can be stored using a distributed framework, which offers some distinct advantages for handling and querying data. Here is how it can be set up:

1. **Shared Space for Common Data:** Start by setting up a shared storage space, specifically for holding S' and S''. This ensures that common, frequently accessed data is readily available for all nodes or servers in the distributed system.

2. **Dedicated Storage for Specific Data:** Each C_i, for i ranging from 1 to k, can be saved in its distinct memory partition. This modularity aids in the easy management and retrieval of data.

3. **Parallel Processing:** With each C_i in its memory segment, queries can be processed in parallel, utilizing multiple servers or nodes. This is especially beneficial when dealing with a high volume of queries, as they can be handled concurrently.

4. **Server Configuration:** Visualize the system as consisting of k servers, each equipped with its memory and processing unit (CPU). All servers have access to the shared storage space, which contains S' and S''. The unique advantage here is that a single query is confined to one server, optimizing processing time.

5. **Efficient Query Routing:** One challenge in a distributed setup is directing the query to the appropriate server. Efficient mechanisms or algorithms can be employed to determine which server holds the relevant C_i for a given query.

Opting for a distributed storage approach offers several noteworthy advantages. Foremost, it provides a practical solution to memory management. Instead of consolidating data within a single, vast storage unit, distributed systems harness multiple, smaller memory modules, which can be both cost-effective and adaptable. This modular setup also bolsters performance, especially when met with a high influx of data queries. By allowing parallel processing across different nodes or servers, query response times can be considerably expedited. Beyond these technical merits, distributed storage exhibits a greater resilience and scalability, attributes that make it particularly suitable for addressing real-world challenges. In environments where data loads and query volumes can vary unpredictably, the flexibility and robustness of distributed systems prove invaluable.

In summary, the distributed storage approach not only optimizes memory management and processing speeds but is also well-suited for real-world scenarios, particularly when dealing with substantial data and high query volumes.

Weighted Graphs Considerations

When considering a positively weighted graph G, the fundamental operations and principles largely remain the same. The primary distinction emerges during the construction of BFS trees. In this context, edge weights become crucial, influencing the time complexity of BFS tree generation. For an unweighted graph, BFS construction has a time complexity of $O(n)$. However, in a weighted scenario, this complexity shifts to $O(n \lg n)$ for the creation of the shortest path tree for a node, using Dijkstra's algorithm [10]. It is essential to note that there is no requirement to store the weights separately; the shortest path tree inherently considers them. While weights do affect the paths, the ultimate goal remains the determination of the shortest path, not the exact distance between nodes. Hence, when the algorithm's complexity is expressed as $\Theta(\mathrm{SPL}(a_i, b_i))$, it denotes the order of the number of edges in the shortest path from v_{a_i} to v_{b_i}. It is pivotal to understand that in this scenario, this order is not influenced by the actual weights of the edges.

4.1.2 Known Centers with Non-Complete Association

In this context, knowledge of the vertices designated as centers is established. Nonetheless, the query list may include anomalies, specifically "**non-centered queries**" as defined in 1.2.8. Fortunately, the occurrence of these queries is infrequent.

Assuming the number of 'non-centered queries' to be b, we introduce a function $f(n)$ dependent on n defined as $f(n) = \frac{q}{b}$. We focus on cases where $\Theta(\frac{n}{f(n)}) \in o(n)$. Consequently,

we define b as a function dependent on n.

For instance, if $f(n) = \lg n$, the number of 'non-centered queries' is $\frac{q}{\lg n}$. In this case, the value of $\frac{n}{f(n)}$ becomes $\frac{n}{\lg n}$, which is in $o(n)$.

In addressing the "Known Centers with Non-Complete Association" scenario, two distinct methods have been proposed: the **Static Stored Vertex Method (SSVM)** and the **Dynamic Stored Vertex Replacement (DSVR) Method**. The SSVM employs a fixed set of stored vertices, and when encountering a non-centered query, processes it using the existing representation. On the other hand, the DSVR introduces a flexible approach by allowing one of the stored vertices to be dynamic, facilitating its replacement in response to non-centered queries. While SSVM offers a more direct approach to the problem, the DSVR is particularly significant as it serves as a baseline for other scenarios in this research. The DSVR provides a reference point, aiding in the comparison of performance and outcomes of different methods across various scenarios.

Static Stored Vertex Method (SSVM)

In the **Static Stored Vertex Method (SSVM)**, the shortest path tree of each center is stored. To recall, the shortest path tree of the i^{th} center is referred to as BST_i. Consequently, the information stored in this method mirrors that of the proposed method for the "Known Centers with Complete Association" scenario.

When encountering centered queries, responses are generated following the methodolgy adopted for the "Known Centers with Complete Association" scenario. The distinction in SSVM arises in the context of non-centered queries, which may occasionally appear in this scenario. To address such queries, a two-sided BFS is employed, which has a time complexity of $O(n)$ (given that in planar graphs $m < 3m$).

The ability to execute the two-sided BFS stems from the orderly spanning tree representation of the stored planar graph. Leveraging lemma 3, navigational queries in G can be answered in constant time.

Regarding space requirements, the SSVM demands an allocation identical to that of the method from the "Known Centers with Complete Association" scenario, which is $3n + 2m + 2mk + o(nk)$ bits. Time complexity for centered queries (u, v) stands at $O(SPL(u, v))$, while non-centered queries necessitate $O(n)$ time complexity.

Dynamic Stored Vertex Replacement (DSVR) Method

DSVR serves as a foundational method for other scenarios, allowing for a maximum of $k+1$ stored vertices. Initially, there are no stored vertices. As queries are presented, we store the shortest path trees for certain vertices, thereby designating them as stored vertices.

Upon receiving the i^{th} query (a_i, b_i), two situations might arise:

- **Centered Query** (a_i, b_i): Consider a query a_i, b_i that is centered. Without a loss of generality, let us assume v_{a_i} is a center. If v_{a_i} currently exists as a stored vertex, the query can be addressed in $\Theta(SPL(a_i, b_i))$ time. If not, a retrieval is performed in $O(n)$ to secure the representation of v_{a_i}'s shortest path tree based on T, after which the query can be addressed in $\Theta(SPL(a_i, b_i))$ time. The space requirement for each shortest path tree representation is $2m + o(n)$ bits. Thus, each stored vertex demands $2m + o(n)$ bits. The first k stored vertices are exclusively reserved for the shortest path trees of centers, while the last stored vertex can be replaced by any non-centered vertex.

- **Non-Centered Query** (a_i, b_i): For the query a_i, b_i that is non-centered, if either v_{a_i} or v_{b_i} is among the stored vertices, then the query can be addressed within $\Theta(SPL(a_i, b_i))$ time. If neither vertex is stored, a retrieval is done in $O(n)$ to fetch the representation of v_{a_i}'s shortest path tree based on T. This representation then occupies the designated memory for the $k + 1^{th}$ stored vertex, facilitating the subsequent answering of the query in $\Theta(SPL(a_i, b_i))$ time.

Unlike the SSVM method, DSVR does not demand the creation of shortest path trees in preprocessing. Instead, it leverages stored vertices for all query responses rather than employing navigational queries within the planar graph. While DSVR demands more storage compared to SSVM, their response times differ. Specifically, the total response time for all queries in DSVR is $O((k + b) \times n + \sum_{i=1}^{q} SPL(a_i, b_i))$, whereas for SSVM, it is $O(b \times n + \sum_{i=1}^{q} SPL(a_i, b_i))$. However, DSVR's importance stems from its function as a foundation for other methods. The maximum number of retrievals is bounded by $k + q/f(n)$, as centered queries may necessitate up to k retrievals, and each non-centered query can lead to at most one retrieval. Hence, the cumulative retrievals for non-centered queries is capped at b, equivalent to $\frac{q}{f(n)}$, leading to a total of $k + \frac{q}{f(n)}$ retrievals. The total space requirement for DSVR is $3n + 4m + 2km + o(nk)$, broken down as: $2n + 2m + o(n)$ bits for storing G based on T, $n + o(n)$ bits for the KB bit-vector, and $2(k + 1)m + o(nk)$ bits for the stored vertices.

Weighted Graphs Considerations

Within the DSVR and SSVM algorithms, the dynamic creation of shortest path trees (instead of preprocessing) necessitates the storage of all graph weights. To accommodate this, we allocate mW space, where W is the number of bits required to store a weight.

In our planar graph representation using T, each edge corresponds to a bracket or parenthesis. This allows us to systematically store the weights: starting with the weight of the edge associated with the first opening bracket, followed by the second, and so on, until the edge corresponding to the $m-(n-1)$th opening bracket. This sequence is then followed by weights of edges linked to each subsequent opening parenthesis in relation to its parent, continuing up to the nth parenthesis.

The cumulative storage requirement for all the edge weights then totals $mW + o(mW)$ bits, where W is the number of bits required to store a weight. Given that we can directly access the bracket or parenthesis of a specific edge, we can also instantly access its weight, ensuring a constant-time navigational query, i.e., $O(1)$.

However, when constructing the shortest path tree on weighted graphs, a straightforward BFS becomes inadequate. More sophisticated algorithms, like Dijkstra's, are necessary. While the BFS algorithm runs in $\Theta(m)$, Dijkstra's algorithm has a runtime of $O(m + n \lg n)$, which translates to $O(n \lg n)$ for planar graphs. Thus, every retrieval in a weighted graph scenario demands $O(n \lg n)$ time. Nevertheless, the space requirement for each stored vertex remains unchanged at $2m + o(n)$ bits.

4.1.3 Unknown Centers with Non-Complete Association

In this scenario, the exact nodes serving as centers are not specified. It is only understood that there are k centers and that $q-b$ queries initiate or conclude at these centers, where b represents the number of non-centered queries. Referring to Theorem 3 and incorporating the DSVR algorithm, it is discerned that, using FPIFO algorithm when setting a limit of $4(k+1)$ on the maximum number of stored vertices, the total number of retrievals does not surpass four times the retrievals seen in DSVR, quantified as $k + \frac{q}{f(n)}$. Consequently, the upper limit of retrievals in this situation is confined to $4(k + \frac{q}{f(n)})$.

Given the prescribed maximum number of stored vertices in the algorithm as $4(k+1)$ and considering each stored vertex demands $2m + o(n)$ bits, the overall bit requirement to accommodate these stored vertices stands at a maximum of $8(k+1)m+o(nk)$. Additionally, to have the presentation of G based on T, $2n + 2m + o(n)$ bits are essential.

Without the knowledge of the nodes designated as centers, leveraging the KB bit vector becomes unfeasible. In its place, a sequence denoted as **SVS (Stored Vertices Sequence)** of size $n \times \lg(4(k+1))$ is introduced. For every vertex, this sequence is set to 'zero' exclusively when that particular vertex is not present in the stored vertices at that instance. If it holds a value i, it signifies that the i^{th} stored vertex is that specific node. This sequence is beneficial in two major aspects:

1. To ascertain if a node is among the stored vertices, facilitating a constant time check.
2. To pinpoint the starting position of the representation of the shortest path tree for a node. Understanding that the memory is partitioned into blocks, each of size $2m$, and each block contains the representation of a shortest path tree of a node. Thus, the i^{th} stored vertex initiates at the $2m \times (i-1)$ bit and culminates at the $2mi-1$ bit.

Therefore, accounting for all components, the cumulative space required amounts to $2n + 10m + 8km + o(nk)$.

In this situation, mirroring the "Known Centers with Non-Complete Association" scenario, this method can be expanded for weighted graphs by incorporating an additional $mW + o(mW)$ bits, where W is the number of bits required to store a weight. Consequently, the retrieval cost transitions from $O(n)$ to $O(n \lg n)$.

Furthermore, across all these scenarios, the representation of the shortest path tree for each stored vertex can be stored either centrally or distributed, as previously detailed.

4.1.4 Variable Centers with Known Dynamics

In this scenario, the centers of nodes are always known, and there are consistently k centers. However, these centers can change over time. When one center node is replaced by another non-center node, the original node loses its center status, and the new one becomes a center. Despite these changes, the total count of centers remains k.

Variable Centers DSVR (VDSVR)

We propose an algorithm called VDSVR based on DSVR, with modifications to accommodate this scenario. A limitation is set for a maximum of $k+1$ stored vertices. The first k stored vertices are designated for centers, and the last one is reserved for a non-center

vertex. Initially, the set of stored vertices should be empty. As queries are processed, the algorithm behaves similarly to DSVR. Moreover, when a change in centers occurs (vertex v is removed from centers and vertex u is added to centers), a retrieval for vertex u is executed to generate the shortest path tree representation of vertex u based on T. Subsequently, the shortest path tree representation of vertex v is removed from memory and replaced with the representation of vertex u.

In the VDSVR algorithm, both the overall space and the upper limit on stored vertices mirror those in the DSVR algorithm. Hence, the maximum number of stored vertices stands at $k+1$, with a total space requirement of $3n + 4m + 2km + o(nk)$.

Given r center replacements, there will be r additional retrievals compared to DSVR. Therefore, the maximum number of retrievals is $k + r + \frac{q}{f(n)}$.

In line with the "Known Centers with Non-Complete Association" strategy, the method is adaptable for weighted graphs by incorporating $mW + o(mW)$ bits, where W is the number of bits required to store a weight. This leads to a change in the retrieval cost from $O(n)$ to $O(n \lg n)$.

4.1.5 Variable Centers with Unknown Dynamics

In this scenario, similar to the "Variable Centers with Known Dynamics" setup, centers can be substituted with non-center vertices. However, the primary distinction between this scenario and the "Variable Centers with Known Dynamics" is the absence of knowledge regarding the centers and their replacement process.

Based on Theorem 3 and with the application of the VDSVR algorithm, it becomes apparent that, by utilizing the FPIFO algorithm and imposing a constraint of $4(k+1)$ for the maximum number of stored vertices, the aggregate retrievals will not exceed four times the retrievals observed in VDSVR. This is quantitatively expressed as $k + r + \frac{q}{f(n)}$. Hence, the maximum retrievals in this context are restricted to $4(k + r + \frac{q}{f(n)})$.

In this algorithm, the SVS is employed similarly to the "Unknown Centers with Non-Complete Association" approach. The computation of total space mirrors that of the "Unknown Centers with Non-Complete Association" scenario, amounting to $2n + 10m + 8km + o(nk)$ bits.

In this context, as per the "Known Centers with Non-Complete Association" approach, the method can be extended for weighted graphs by adding $mW + o(mW)$ bits, where W is the number of bits required to store a weight. As a result, the retrieval cost shifts from $O(n)$ to $O(n \lg n)$.

4.2 Distance Oracle

In this section, we explore the **distance oracle** data structure specifically designed for the unweighted planar graphs. The distance oracle allows for the computation of distances between any two given vertices in the graph. In the context of unweighted graphs, the distance oracle between any two arbitrary nodes u and v is represented as $SPL(u, v)$. Thus, in this section, our primary goal is to address the $SPL(u, v)$ query.

In this thesis, we primarily concentrate on two configurations: the **Normal configuration** and the **Center-based configuration**. In the Normal configuration, there are no specific constraints on the query list. However, in the Center-based configuration, certain assumptions are made about the query list, with specific vertices appearing more frequently than others. In this section, we introduce a distance oracle data structure that is applicable to both the Normal and Center-based configurations.

Our proposed algorithm can be adapted to all center-based configuration scenarios by incorporating an additional $o(n)$ bits for each vertex stored. Thus, if v is among the stored vertices, the algorithm can efficiently determine $SPL(v, u)$ for any arbitrary u within $O(\lg^{1+\epsilon} n)$ time, for any constant $\epsilon > 0$.

For the normal configuration, our algorithm requires $2m + o(n)$ bits for each node, distributed individually. Additionally, a shared memory of $2n + 2m + o(n)$ bits is essential. Consequently, with a maximum of $2n + 2m + 2nm + o(n)$ bits, the algorithm can address the SPL for any pair of arbitrary nodes within $O(lg^{1+\epsilon} n)$ time, where $\epsilon > 0$. Given that $m < 3n$ in planar graphs, a total of at most $6n^2 + o(n)$ bits is necessary to manage the distance oracle in the normal configuration.

Definition 4.2.1. Distance from v_i to v_j in T' ($\text{dis}(T', i, j)$): Let T' be a spanning tree of G and v_i be the i^{th} vertex in the counterclockwise preorder of the orderly spanning tree T. The distance from i to j in T', denoted as $\text{dis}(T', i, j)$, represents the number of edges in the path from v_i to v_j in T'. Given that T' is a spanning tree of the connected graph G, there exists a unique path between v_i and v_j within T'. Moreover, if T' is the shortest path tree rooted at v_i or v_j, then $\text{dis}(T', i, j)$ is equivalent to $SPL(i, j)$.

Lemma 6. *Let $G(V, E)$ be a connected undirected planar graph with n vertices and m edges. Consider H to be an arbitrary planar embedding of G and T as an arbitrary Orderly Spanning Tree of H, represented by sequences S' and S''. Define v_i as the i^{th} element in the preorder of T. For every rooted spanning tree T' of G, a data structure can be established in $O(n)$ time, which requires $2m + o(n)$ bits and can effectively answer:*

- *Parent query: parent(T', i) queries in constant time,*

- *Distance query: $dis(T', r, i)$ queries, where r is the root index of T' in the preorder of T, in $O(\lg^{1+\epsilon} n)$ time for any constant $\epsilon > 0$,*

for all i from 1 to n.

Proof. For the proof of this lemma, consider lemma 5, which provides a data structure enabling constant-time parent queries. We denote this data structure as $PD_{T'}$ (representing the parent query data structure for T'). Our goal is to augment $PD_{T'}$ with an additional $o(n)$ bits to address the distance query.

Let L be defined as $\lceil (\lg n)(\lg \lg n) \rceil$. Following the approach in Lu's paper [20], we aim to select $O\left(\frac{n}{L}\right)$ special nodes. To achieve this:

1. Start by picking an arbitrary leaf u in T' that has the maximum depth within T'.
2. Identify the highest(minimum depth) ancestor of u in T' such that the distance from u to this ancestor in T' does not exceed L. Let us call this ancestor v.
3. Choose v as a special node, then eliminate the subtree rooted at v in T'.
4. Repeat the above steps on the residual tree until no nodes are left.

If v is not the root of T', then the exact distance from u to v is L. Hence, whenever a node is elected as a special node and it is not the root, at least L nodes are discarded and are not chosen as special node. Thus, the number of special nodes is $O\left(\frac{n}{L}\right)$.

Consider a bit-vector, SN, of length n. For this vector, $SN[i] = 1$ if and only if v_i is designated as a special node. In simpler terms, if a vertex v_i is a special node, then the corresponding position in the bit-vector will be set to 1.

Given this, the total number of ones in SN is $O\left(\frac{n}{L}\right)$, which is $o(n)$. Leveraging lemma 2, we can represent SN using $o(n)$ space while supporting rank queries in SN within $\tilde{O}(n, 1)$ time. So, in $\tilde{O}(n, 1)$ time, we can determine whether a node is special.

Since the count of special nodes is $o\left(\frac{n}{\lg n}\right)$, storing $O(\lg n)$ bits for each special node results in a total space requirement of $o(n)$. For every special node v, we record its distance to the root of T'. This distance ranges from 0 to $n - 1$, so it can be represented using $O(\lg n)$ bits. Thus, with a bit-vector of size $o(n)$, we can capture the distance from each special node to the root of T'.

To find the distance $\text{dis}(T', r, i)$ for any i from 1 to n, use the following procedure:

1. Initialize u to v_i.
2. Initialize the distance from v_i to u as 0.
3. In $\tilde{O}(n, 1)$ time, determine if u is a special node.
4. While u is not a special node:
    - Use a parent query to find u's parent in constant time.
    - Increment the distance from v_i to u by one.
    - Update u to its parent and repeat for the updated u. This step will be executed no more than L times since each non-special node has an ancestor within L distance that is a special node.
5. If u is identified as a special node:
    - In constant time, retrieve the distance from u to the root of T'.
    - The total distance from v_i to the root of T' is the sum of the computed distance from v_i to u and the stored distance from u to the root. We already have the distance of u to the root because u is a special node.

Given that step 4 will be repeated at most L times (as each non-special node has an ancestor no further than L distance away which is a special node), the algorithm runs in $O(L \times \tilde{O}(n, 1))$ time. This time complexity can be simplified to $O(\lg^{1+\epsilon} n)$ for any constant $\epsilon > 0$.

In summary, through the data structure we have presented, we are able to achieve parent queries in constant time and distance queries in $O(\lg^{1+\epsilon} n)$ for any constant $\epsilon > 0$ using $2m + o(n)$ bits.

It is important to highlight that if, instead of a distance query, we possess any query in T' that meets the following criteria:

- For any nodes u and v (where v is an ancestor of u), the result of query(u) can be derived from the path between u and v combined with the answer for query(v).
- The bit size needed to store the answer of query(u) for any node u is $O(\log n)$.

Under these conditions, we can replace the distance query with this new query. □

Based on the proof from lemma 6, we introduce an additional $o(n)$ bits to the BFS tree representation of each stored vertex to manage the distance query in $O(lg^{1+\varepsilon}n)$ where $\varepsilon > 0$. This does not impact the space requirements of each scenario but enables distance query handling. As discussed in the center-based scenario, this can be stored in either a centralized or distributed manner. It is crucial to note, however, that this structure is designed specifically for scenarios involving unweighted graphs.

Based on lemma 6 for the standard configuration, we are required to represent G based on T using $2n + 2m + o(n)$ bits. Additionally, for every node, we must allocate $2m + o(n)$ bits for each node in G to represent T_{v_i} (which is the shortest path tree of v_i), in order to manage both parent and distance queries within T_{v_i}. Thus, the total space requirement sums up to $2n + 2m + 2nm + o(n^2)$, which is at most $6n^2 + o(n^2)$.

Each node's T_{v_i} representation can also be stored in a distributed manner. For instance, every server can store individual T_{v_i} representations along with the shared data from the representation of G based on T, requiring $2n + 2m + o(n)$ bits. If a request is received, the query can be directed to one of the endpoint servers, where the answer is computed in parallel. This parallel computation capability is another advantage of this approach.

4.3 Decentralized Routing Table

In 1999, Gavoille et al. introduced an algorithm addressing this issue [11]. Their method constructs the shortest path routing table in linear time, occupying 8n + o(n) bits. It answers the port query using the routing table with a time complexity of $O(\lg^{2+\epsilon} n)$ for any positive constant $\epsilon > 0$. Later, in 2010, Lu introduced a superior algorithm to Gavoille's [20]. It also constructs the routing table in linear time but can process the port query in $O(\lg^{1+\epsilon} n)$ for any positive constant $\epsilon > 0$. Each node's routing table in Lu's method requires up to 7.181n + o(n) bits. However, the total space needed by this method is $7n^2 + o(n^2)$ bits. In this section, a data structure is presented for the routing table. This structure is computed in polynomial time, takes up no more than $3.2n^2 + o(n)$ bits in total, and provides constant-time port queries.

We present an algorithm designed for distributed routing tables. Although this algorithm is adaptable to various graph types, our time and space computations here are based on a planar graph G. Initially, each node is arbitrarily labeled with a number between 0 and $n - 1$. Let us denote the vertex labeled i as u_i. For each i in the range 0 to $n - 1$, the following information is stored:

1. **Neighbourhood List**: This list keeps track of the indices of u_i's neighbors. The jth neighbor in the list should be accessible in constant time.

2. **Port Index List**: For all j in the range 0 to $i-1$ and $i+1$ to $n-1$ (essentially all nodes except i), the index of $\text{port}_{u_i}(u_j)$ within u_i's neighbourhood list should be accessible in constant time.

Theorem 4. *There exists a data structure for the port index list corresponding to each u_i which requires no more than $\frac{8}{15}nd_i + o(nd_i)$ bits. This data structure allows constant-time access to any of its elements.*

Proof. **Standard Storage Method:** A typical approach for the port index list involves the following: For any node u_j (given $i \neq j$), an index within the range $[0,\ d_i - 1]$ is allocated to $port_{u_i}(u_j)$ in the neighbourhood list. As indices can be any value from 0 to $d_i - 1$, we need $\lceil \lg d_i \rceil$ bits to represent each. Thus, this conventional method consumes $n \times \lceil \lg d_i \rceil + o(n \times \lceil \lg d_i \rceil)$ bits.

By referencing Lemma 8, for $d_i > 5$, the inequality $n \times \lceil \lg d_i \rceil \leq \frac{nd_i}{2}$ holds. Therefore, for such cases, the storage demands are at most $\frac{nd_i}{2} + o(nd_i)$ bits.

Now, let us consider situations where $d_i \leq 5$:

- **When** $d_i = 1$: Storing the port index list is unnecessary. This is because $port_{u_i}(v)$ is unique for any $v \neq u_i$, given that u_i has just one neighbour, so $port_{u_i}(v)$ directly points to that single neighbour.

- **When** $d_i = 2$ **or** 4: The relationship $\lceil \lg d_i \rceil \leq \frac{d_i}{2}$ is valid. As per our earlier discussion for $d_i > 5$, we conclude that in these cases we need at most $\frac{nd_i}{2} + o(nd_i)$ bits.

- **When** $d_i = 5$: If we possess n numbers, each ranging from 0 to 4, we can group them in sets of three and represent them collectively. With this arrangement, the number of possible states for a group of three values (each between 0 and 4) is $5^3 = 125$. As 125 states can fit within 7 bits (since $2^7 = 128$), the storage requirement is less than $3 \times \frac{5}{2}$ bits. Consequently, for this case, we need no more than $\frac{7}{15}nd_i + o(nd_i)$ bits.

- **When** $d_i = 3$: If we have n numbers, each of which can be either 0, 1, or 2, we can group them in sets of five. The total number of possible states for such a five-number group is $3^5 = 243$. This can be represented using 8 bits (since $2^8 = 256$). Therefore, for this scenario, the maximum storage requirement is $n \times \frac{8d_i}{15} + o(n)$ bits.

□

The presence of both the **Neighbourhood List** and the **Port Index List** ensures that port queries can be resolved in constant time. In the **Neighbourhood List**, neighbor labels of u_i are stored in an array of size d_i. Each of these labels will be represented using $\lceil \lg n \rceil$ bits. Consequently, the most space this list might need is $d_i \times \lceil \lg n \rceil + o(d_i \times \lceil \lg n \rceil)$.

For the **Port Index List**, following theorem 4, the structure uses no more than $\frac{8}{15}nd_i + o(nd_i)$ bits, taking into account that d_i represents the degree of u_i in G. Importantly, unless d_i equals 3, the structure will require at most $0.5nd_i + o(n \times d_i)$ bits. It is only when d_i is strictly 3 that the bit demand goes up to $\frac{8}{15}nd_i + o(nd_i)$.

So, for each index i, the required space is no more than $d_i\lceil \lg n \rceil + \frac{8}{15}nd_i + o(n \lg d_i)$ bits. Given that $d_i \in o(n \lg d_i)$, this expression can be simplified to $d_i \lg n + \frac{8}{15}nd_i + o(n \lg d_i)$. Summing over all i, from 0 to $n-1$, we get:

$$\sum_{i=0}^{n-1}(d_i \lg n + \frac{8}{15}nd_i + o(n \lg d_i))$$

$$= \lg n * \sum_{i=0}^{n-1} d_i + \frac{8}{15}n * \sum_{i=0}^{n-1} d_i + \sum_{i=0}^{n-1} o(n \lg d_i)$$

$$= 2m \lg n + \frac{16}{15}nm + o(nm)$$

Given that G is a planar graph, where $m < 3n$, we deduce:

$$2m \lg n + \frac{16}{15}nm + o(nm) < 6n \lg n + \frac{16}{5}n^2 + o(n^2)$$

Considering that $6n \lg n$ is a term in $o(n^2)$, the final expression becomes:

$$6n \lg n + \frac{16}{5}n^2 + o(n^2) = \frac{16}{5}n^2 + o(n^2)$$

The proposed algorithm efficiently retrieves port queries in constant time. For each node, the maximum space requirement is $\frac{8}{15}nd_i + d_i \lg n + o(n \lg d_i)$ bits. It is feasible to store each node's routing table individually in separate memory locations, enabling distributed storage. In total, the space needed is $3.2n^2 + o(n^2)$ bits. Furthermore, utilizing this structure, we can determine the shortest path between any two arbitrary nodes, u and v, in $\Theta(SPL(u, v))$.

Chapter 5

Conclusion

This thesis undertook the exploration of shortest path, distance oracle, and port queries within the realm of planar graphs, placing a premium on the development of space-efficient data structures. Our approach to tackling these queries varied depending on the configuration of the graph. Specifically, for center-based configurations, we concentrated on optimizing shortest path queries and distance oracles. Meanwhile, in non-configured settings, our focus shifted to refining distance oracles and port queries.

5.1 Main Contributions and Findings

The primary contributions of this thesis are twofold: the development of space-efficient data structures and the facilitation of their distributed implementation. Our methods are not only space-efficient but they are also conducive to parallel processing. This parallelization capability enables the application of our strategies to large-scale graphs, such as those representing city infrastructures.

We have yet to prove that our data structures are succinct, meaning they occupy the least possible space up to a lower order term. However, this potential characteristic suggests that our structures are close to being as space-efficient as theoretically possible. Establishing the lower bounds for space complexity in center-based configurations remains an open question and a compelling direction for future research.

5.2 Advantages and Practical Applications

One of the most significant advantages of our data structures is their inherent suitability for distributed computing. This attribute allows our algorithms to be deployed effectively for large-scale graph applications, where handling massive datasets efficiently is crucial. In practical terms, this means our data structures and algorithms can be instrumental in applications such as urban traffic navigation systems, where real-time data processing and routing decisions are based on extensive city graph data.

5.3 Future Work

Looking forward, there are several avenues for future research that could potentially extend the applicability and efficiency of our work:

- **Lower Bound Proofs:** A key area for future investigation is to establish the lower bounds for the space complexity of center-based configurations. Proving these bounds would be a significant step towards confirming the succinctness of our data structures.
- **Expansion to Beer Graphs:** The notion of shortest paths invariably passing through a center in a center-based configuration presents an intriguing prospect. Future work could explore adapting our data structures for beer graphs, where any shortest path between two vertices is known to pass through a center.
- **Approximation of Distance Oracles:** There is potential to approximate distance oracles in general graphs under the assumption that each shortest path includes a center. This approximation could lead to more efficient routing solutions in complex networks.
- **Incorporation of Weighted Graphs:** Another promising direction is the exploration of weighted graphs for distance oracles. Adjusting our data structures to accommodate edge weights could enhance their applicability to a broader range of graph-based problems.
- **Adaptation to Various Graph Types:** Future research could focus on modifying these data structures for different types of graphs, particularly subclasses of planar graphs. Such adaptations may yield data structures with even lower time complexity or reduced space requirements, offering more efficient solutions for specific graph types.

This thesis has successfully developed space-efficient methods for handling various types of queries in planar graphs. By introducing and exploring novel data structures, we have broadened the understanding of query processing in such graphs. The potential applications of these findings extend to network analysis and related fields. Future research, building upon the foundations laid here, has the opportunity to further refine and expand the scope of these methods, contributing to the evolution of data structures and algorithms within the realm of graph theory and network analysis.

References

[1] Girish Balakrishnan, N S Narayanaswamy, Sankardeep Chakraborty, and Kunihiko Sadakane. Succinct data structure for path graphs. In *2022 Data Compression Conference (DCC)*, pages 262–271, 2022.

[2] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *ArXiv*, abs/1806.01799, 2018.

[3] Nicolas Bonichon, Cyril Gavoille, and Nicolas Hanusse. *An Information-Theoretic Upper Bound on Planar Graphs Using Well-Orderly Maps*, pages 17–46. 01 1970.

[4] L. Castelli Aleardi, O. Devillers, and G. Schaeffer. Succinct representations of planar maps. *Theoretical Computer Science*, 408(2):174–187, 2008. Excursions in Algorithmics: A Collection of Papers in Honor of Franco P. Preparata.

[5] Yi-Ting Chiang, Ching-Chi Lin, and Hsueh-I Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34, 03 2001.

[6] Clark, David. *Compact PAT trees*. PhD thesis, 1997.

[7] Vincent Cohen-Addad, Søren Dahlgaard, and Christian Wulff-Nilsen. Fast and compact exact distance oracle for planar graphs, 2017.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[9] Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, Anurag Murty Naredla, and Kaiyu Wu. Shortest beer path queries in interval graphs, 2022.

[10] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[11] Cyril Gavoille and Nicolas Hanusse. Compact routing tables for graphs of bounded genus. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, ICAL '99, page 351–360, Berlin, Heidelberg, 1999. Springer-Verlag.

[12] Roberto Grossi and Elena Lodi. Simple planar graph partition into three forests. *Discrete Applied Mathematics*, 84(1):121–132, 1998.

[13] Meng He, J. Ian Munro, Yakov Nekrich, Sebastian Wild, and Kaiyu Wu. Distance Oracles for Interval Graphs via Breadth-First Rank/Select in Succinct Trees. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, volume 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[14] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, oct 1974.

[15] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554. IEEE Computer Society, 1989.

[16] Kenneth Keeler and Jeffery Westbrook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58(3):239–252, 1995.

[17] Khodaee, SayedMohammadAmin. The pair cache problem. Master's thesis, 2023.

[18] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, Boston, MA, 7 edition, 2016.

[19] Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press, 1976.

[20] Hsueh-I Lu. Improved compact routing tables for planar networks via orderly spanning trees. *SIAM J. Discrete Math.*, 23:2079–2092, 01 2010.

[21] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[22] J. Munro, Venkatesh Raman, and Srinivasa Rao Satti. Space efficient suffix trees. *Journal of Algorithms*, 39:205–222, 05 2001.

[23] J. Ian Munro. An implicit data structure supporting insertion, deletion, and search in o($\log^2$ n) time. *J. Comput. Syst. Sci.*, 33(1):66–74, 1986.

[24] J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Hyderabad, India, December 18-20, 1996, Proceedings*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.

[25] J. Ian Munro and S. Srinivasa Rao. Succinct representation of data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004.

[26] J. Ian Munro and Hendra Suwanda. Implicit data structures for fast search and update. *J. Comput. Syst. Sci.*, 21(2):236–250, 1980.

[27] J. Ian Munro and Kaiyu Wu. Succinct data structures for chordal graphs. In *International Symposium on Algorithms and Computation*, 2018.

[28] J.I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 118–126, 1997.

[29] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[30] Rajeev Raman and S. Srinivasa Rao. Succinct representations of ordinal trees. 10 2014.

[31] Alfonso Shimbel. Structural parameters of communication networks. *Bulletin of Mathematical Biology*, 15:501–507, 1953.

[32] György Turán. On the succinct representation of graphs. *Discret. Appl. Math.*, 8(3):289–294, 1984.

[33] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, September 2000.

[34] J. W. J. Williams. Algorithm 232 - Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

APPENDICES

Appendix A

Proof of Lemmas

Lemma 7. *In a tree T with n vertices, the total number of children across all nodes does not exceed n.*

Proof. For any given vertex v_i in the tree T, its number of children $c(v_i)$ can be defined as:

$$c(v_i) = deg(v_i) - 1$$

Considering all vertices in T:

$$\sum_{i=1}^{n} c(v_i) = \sum_{i=1}^{n} (deg(v_i) - 1)$$

Using the Handshaking Lemma for trees, the summation of degrees for all vertices is equal to twice the number of edges:

$$\sum_{i=1}^{n} deg(v_i) = 2(n-1)$$

Substituting this into our previous expression:

$$\sum_{i=1}^{n} c(v_i) = 2(n-1) - n$$

$$\sum_{i=1}^{n} c(v_i) = n - 2$$

Thus, the total number of children for all vertices in T is $n-2$, which is less than n. This concludes our proof. □

Lemma 8. *Given any integer d greater than 5, it follows that the ceiling value of $\log_2(d)$ is always less than or equal to half of d, i.e., $\lceil \lg d \rceil \leq \frac{d}{2}$.*

Proof. To prove the lemma, we use induction on d.

Base Cases: For $d = 6, 7, 8, 9, 10$, and 11, it can be easily verified that $\lceil \lg d \rceil \leq \frac{d}{2}$.

Inductive Step: Assume that the statement holds for $d' = \frac{d}{2}$, i.e., $\lceil \lg d' \rceil \leq \frac{d'}{2}$ for some $d \geq 12$.

Given $\lceil \lg d' \rceil \leq \frac{d}{4}$, we know:

$$\lceil \lg d \rceil = \lceil \lg(2 \times d') \rceil = \lceil \lg 2 + \lg d' \rceil = 1 + \lceil \lg d' \rceil$$

Since $1 + \lceil \lg d' \rceil < \frac{d}{2} - \frac{d}{4}$ (because $d \geq 12$ implies $\frac{d}{4} > 1$), we conclude:

$$\lceil \lg d \rceil \leq \frac{d}{2}$$

Thus, by the principle of mathematical induction, the statement is true for all $d > 5$. □