An Experimental Study of Distance Sensitivity Oracles

by

Vincent Troy Williams

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Rahul Tripathi, Ph.D.
Lawrence O. Hall, Ph.D.
Sudeep Sarkar, Ph.D.

Date of Approval:
10/26/2010

Keywords: algorithm analysis; data structures; Dijkstra’s algorithm; dynamic programming; experimental algorithmics; graphs and trees

Copyright © 2010, Vincent Troy Williams
DEDICATION

to my Father, Mother and Brother
ACKNOWLEDGEMENTS

I would like to thank Dr. Tripathi for taking me on as his research assistant and bringing to my attention the recent work on the distance sensitivity problem [BK09]—the problem of constructing a data structure (called distance sensitivity oracle) for any edge-weighted graph $G$ that supports queries on shortest distance and/or path from any vertex $x$ to any vertex $y$ avoiding any vertex $v$ in $G$. Along the way, he has helped me to improve my code, allowing for a more reliable and stable release of the code that I can be proud of.

I would like to thank Aaron Bernstein and David Karger, the authors of the paper “A Nearly Optimal Oracle for Avoiding Failed Vertices and Edges” [BK09], for developing an efficient algorithm in both space and time for the distance sensitivity problem.

I would like to thank John Boyer for posting his code for a Fibonacci heap implementation of min-priority queue [Boy97]. Using his code, I was able to have the most efficient implementation of Dijkstra’s algorithm for computing single-source shortest distances and paths.

I would like to thank Johannes Fischer and Volker Heun for developing code for the Range Minimum Query (RMQ) data structure [FH06] that was used to minimize the total running time of the distance sensitivity oracle construction. I did modify their code to return the largest value in a range instead of the smallest.

I would like to thank Dan Moulding for creating the Visual Leak Detector code [Mou09]. His code when ran on a Win32 machine compiled in Visual Studio helped me to find many memory leaks.
# TABLE OF CONTENTS

LIST OF TABLES iii

LIST OF FIGURES v

LIST OF ALGORITHMS vi

ABSTRACT vii

CHAPTER 1 INTRODUCTION 1

1.1 Background and Problem Description 1
1.2 Motivation 3
1.3 Notations 4

CHAPTER 2 COMPUTING ALL-PAIRS OF SHORTEST PATHS 6

2.1 Definitions of the $D$, $H$, $P$, and $ST$ Tables 6
2.2 Creation of the $D$, $H$, $P$, and $ST$ Tables 6

CHAPTER 3 ASSIGNING PRIORITIES 11

3.1 Definition of Centers 11
3.2 Assign-Priority Algorithm 11

CHAPTER 4 COVERING CHAINS 15

4.1 Definitions of the $Cr$, $Cl$, and $BCP$ Tables 15
4.2 Creation of the $Cr$, $Cl$, and $BCP$ Tables 16
4.3 Finding Centers on Both Sides of an Avoiding Vertex 19

CHAPTER 5 SHORTEST DISTANCES FROM ALL CENTERS AVOIDING THEIR COVERED VERTICES 20

5.1 Definitions of the $D_k$, $D_kE$, $\hat{D}_k$ and $\hat{D}_kE$ Tables 20
5.2 Creation of the $D_k$, $D_kE$, $\hat{D}_k$ and $\hat{D}_kE$ Tables 21

CHAPTER 6 SHORTEST DISTANCES FROM ALL VERTICES AVOIDING THE FIRST EDGES OF THEIR SHORTEST PATHS 24

6.1 Definitions of the $De$, $DeE$, $\hat{D}e$, and $\hat{De}E$ Tables 24
6.2 Creation of the $De$, $DeE$, $\hat{D}e$, and $\hat{De}E$ Tables 25

CHAPTER 7 BOTTLENECK VERTICES 28

7.1 Definition of the $BV$ Table 28
7.2 Creation of the $BV$ Table 29
LIST OF TABLES

Table 1.1  Notations used in this Thesis  

Table 10.1  Memory usage of the nearly optimal oracle on synthetic test data (total memory used is measured in kilobytes)  

Table 10.2  Memory usage of the brute-force method on synthetic test data (total memory used is measured in kilobytes)  

Table 10.3  Memory usage of the nearly optimal oracle and the brute-force method on real-world test data (total memory used is measured in kilobytes)  

Table 10.4  Total running times used by the brute-force method on synthetic test data (running time is measured in seconds)  

Table 10.5  Total running times used by the nearly optimal oracle on synthetic test data (running time is measured in seconds)  

Table 10.6  Total running times for the algorithm All-Pairs-Shortest-Paths on synthetic test data (running time is measured in seconds)  

Table 10.7  Total running times for the algorithm Assign-Priority on synthetic test data (running time is measured in seconds)  

Table 10.8  Total running times for the algorithms Create-Cr-and-BCP-Tables and Create-Cl-Table on synthetic test data (running time is measured in seconds)  

Table 10.9  Total running times for the algorithms Create-D_k-and-D_kE-Tables and Create-De-and-DeE-Tables on synthetic test data (running time is measured in seconds)  

Table 10.10  Total running times for the algorithms Create-BV-Table and Create-DBV-and-FEBV-Tables on synthetic test data (running time is measured in seconds)  

Table 10.11  Construction times of the nearly optimal oracle and the oracle given by the brute-force method on real-world test data (running time is measured in seconds)
<table>
<thead>
<tr>
<th>Table 10.12</th>
<th>Total running times for the algorithm All-Pairs-Shortest-Paths on real-world test data (running time is measured in seconds)</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 10.13</td>
<td>Total running times for the algorithm Assign-Priority on real-world test data (running time is measured in seconds)</td>
<td>62</td>
</tr>
<tr>
<td>Table 10.14</td>
<td>Total running times for the algorithms Create-Cr-and-BCP-Tables and Create-Cl-Table on real-world test data (running time is measured in seconds)</td>
<td>62</td>
</tr>
<tr>
<td>Table 10.15</td>
<td>Total running times for the algorithms Create-$D_k$-and-$D_kE$-Tables and Create-De-and-DeE-Tables on real-world test data (running time is measured in seconds)</td>
<td>62</td>
</tr>
<tr>
<td>Table 10.16</td>
<td>Total running times for the algorithms Create-BV-Table and Create-DBV-and-FEBV-Tables on real-world test data (running time is measured in seconds)</td>
<td>63</td>
</tr>
<tr>
<td>Figure 10.1</td>
<td>Total memory used for the nearly optimal oracle on synthetic test data</td>
<td>46</td>
</tr>
<tr>
<td>Figure 10.2</td>
<td>Total memory used for the brute-force method on synthetic test data</td>
<td>46</td>
</tr>
<tr>
<td>Figure 10.3</td>
<td>Total running times for the nearly optimal oracle on synthetic test data</td>
<td>49</td>
</tr>
<tr>
<td>Figure 10.4</td>
<td>Total running times for the brute-force method on synthetic test data</td>
<td>49</td>
</tr>
<tr>
<td>Figure 10.5</td>
<td>Total running times for the algorithm All-Pairs-Shortest-Paths on synthetic test data</td>
<td>52</td>
</tr>
<tr>
<td>Figure 10.6</td>
<td>Total running times for the algorithm Assign-Priority on synthetic test data</td>
<td>54</td>
</tr>
<tr>
<td>Figure 10.7</td>
<td>Total running times for the algorithms Create-Cr-and-BCP-Tables and Create-Cl-Table on synthetic test data</td>
<td>55</td>
</tr>
<tr>
<td>Figure 10.8</td>
<td>Total running times for the algorithms Create-D_k-and-D_kE-Tables and Create-De-and-DeE-Tables on synthetic test data</td>
<td>56</td>
</tr>
<tr>
<td>Figure 10.9</td>
<td>Total running times for the algorithms Create-BV-Table and Create-DBV-and-FEBV-Tables on synthetic test data</td>
<td>58</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

1. Initialize-Single-Source  
2. Relax  
3. Construct-Shortest-Paths-Tree  
4. All-Pairs-Shortest-Paths  
5. Assign-Priority  
6. Create-Cr-and-BCP-Tables  
7. Create-CI-Table  
8. Find-Centers  
9. Create-$D_k$-and-$D_kE$-Tables  
10. Create-$De$-and-$DeE$-Tables  
11. MTC  
12. Find-Bot  
13. Create-BV-Table  
14. Create-$DBV$-and-$FEBV$-Tables  
15. Shortest-Distance-Avoiding-Vertex  
16. Shortest-Distance-Avoiding-Edge  
17. Shortest-Path-Avoiding-Vertex  
18. Create-Path-Avoiding-Vertex  
19. Shortest-Path-Avoiding-Edge
An Experimental Study of Distance Sensitivity Oracles

Vincent Troy Williams

ABSTRACT

The paper “A Nearly Optimal Oracle for Avoiding Failed Vertices and Edges” by Aaron Bernstein and David Karger lays out a nearly optimal algorithm for finding the shortest distances and paths between vertices with any given single failure in constant time without reconstructing the oracle. Using their paper as a guideline, we have implemented their algorithm in C++ and recorded each step in this thesis. Each step has its own pseudo-code and its own analysis to prove that the entire oracle construction stays within the stated running time and total space bounds, from the authors. The efficiency of the algorithm is compared against that of the brute-force methods total running time and total space needed. Using multiple test cases with an increasing number of vertices and edges, we have experimentally validated that their algorithm holds true to their statements of space, running time, and query time.
CHAPTER 1
INTRODUCTION

1.1 Background and Problem Description

The distance sensitivity problem requires the construction of a data structure (called distance sensitivity oracle or, in short, oracle) for any edge-weighted graph $G$ that supports queries on shortest distance and/or path from any vertex $x$ to any vertex $y$ avoiding any vertex $v$ or any edge $(u,v)$ in $G$. The oracle has been created many times before, each revision improving upon the previous oracle. In 1959, Dijkstra gave the famous “Dijkstra’s algorithm” [Dij59] that can find all-pairs of shortest paths in total running time of $O(mn + n^2 \log n)$ and total space of $O(n^2)$.\(^1\) Although Dijkstra’s algorithm cannot handle any vertex or edge failure without first reconstructing the oracle for the failed vertex or edge, it is still the starting point for all known oracles. In 2008, Demetrescu et al. [DTCR08] created an oracle with a total running time of $O(mn^2 + n^3 \log n)$, total space of $O(n^2 \log n)$, and query time of $O(1)$. Following in their footsteps, Bernstein and Karger [BK08] created their first oracle with a total running time of $\tilde{O}(n^2 \sqrt{m})$, total space of $\tilde{O}(n^2)$, and query time of $O(1)$.\(^2\) Improving upon their earlier result, Bernstein and Karger [BK09] created a nearly optimal oracle that takes a total running time of $\tilde{O}(mn)$, total space of $\tilde{O}(n^2)$, and query time of $O(1)$.

In this thesis, we have implemented the nearly optimal oracle by Bernstein and Karger [BK09] and presented an experimental evaluation of our implementation. The oracle in [BK09] was constructed through a series of steps that built up from one another.

---

\(^1\)Values $n$ and $m$ stand for the total number of vertices and the total number of edges, respectively, in the graph $G$. Refer to Table 1.1 for all notations used throughout this thesis.

\(^2\) $f(n) = \widetilde{O}(g(n))$ if $f(n) = O(g(n) \text{polylog}(n))$. 

1
graphs, Dijkstra’s algorithm must be run first on the graph to obtain all-pairs of shortest
distances and paths. From there, the algorithm deals with assigning vertices with an
integer priority in the range \([1, \log n]\) that determines the number of vertices that a
single vertex can cover. A vertex \(c\) is said to cover another vertex \(v\) if the shortest path
from \(c\) to every other vertex \(y\) that avoids \(v\) is known and stored. The cover vertices \(c\)
are classified into different priority groups. Higher priority vertices are rare and cover
more vertices than the more common lower priority vertices. A cover vertex \(c\) of priority
\(k\), where \(1 \leq k \leq \log n\), can cover all vertices with priority less than or equal to its own,
between all levels 1 and \(O(2^k)\) in its own shortest path tree.

Every vertex in the graph is a cover vertex and, on a shortest path between any two
vertices, intervals of vertices are defined using incremental priorities. Let us say that a
vertex \(x\) has priority \(k\). Then on a shortest path \(\pi_{x,y}\) from \(x\) to \(y\), the first interval \([x,u]\)
contains all vertices from \(x\) to the first vertex \(u\) of priority greater than \(k\). This process
of interval creation is repeated until vertex \(y\) is reached. For each interval, the next
step is to determine which vertex on the interval causes the highest distance if removed
from the graph. On the shortest path \(\pi_{x,y}\), a vertex \(w\) in any interval \([s,t]\) is said to be
the bottleneck vertex with respect to \(x\), \(y\), and \([s,t]\) if, among the vertices in \([s,t]\), the
removal of \(w\) from the graph results in the maximum increase in the shortest distance.
With the bottleneck vertices of all the intervals known, the algorithm then finds the
shortest distance avoiding the bottleneck vertices and stores all these distances. With
the oracle now constructed, queries for computing shortest distances in the presence of
a single failed vertex or edge can be answered in \(O(1)\) time. The oracle can also answer
queries asking for shortest paths avoiding a single failed vertex or edge in \(O(L)\) time,
where \(L\) is the number of edges on the shortest path. The following lemma is crucial
for computing the answers of these queries:

Lemma 1.1 (Bottleneck Lemma [BK09]): For any vertices \(x\), \(y\), and \(v\), let \(d_{x,y}\) denote
the shortest distance from \(x\) to \(y\) and let \(d_{x,y,v}\) denote the shortest distance from \(x\) to \(y\)
avoiding \(v\). Let \(x\), \(s\), \(v\), \(t\), and \(y\) be vertices in that order on the shortest path \(\pi_{x,y}\) from
Let \( v \) be the failed vertex and \( s \neq v \neq t \). Let \( w \) be the bottleneck vertex of the interval \([s, t]\). Then,

\[
d_{x,y,v} = \min\{d_{x,s} + d_{s,y,v}, d_{x,t,v} + d_{t,y}, d_{x,y,w}\}.
\]

In other words, \( d_{x,y,v} \) is the minimum of the following three values: (1) the shortest distance from vertex \( x \) to \( s \) plus the shortest distance from vertex \( s \) to vertex \( y \) avoiding vertex \( v \), (2) the shortest distance from vertex \( x \) to vertex \( t \) avoiding vertex \( v \) plus the shortest distance from vertex \( t \) to vertex \( y \), and (3) the shortest distance from vertex \( x \) to vertex \( y \) avoiding the bottleneck vertex \( w \).

While Bernstein and Karger [BK09] gave theoretical guarantees for their oracle construction, they did not evaluate their oracle for real-world applications. We have taken it upon ourselves to experimentally validate that their algorithm performs to support their statements in implementation with data (both randomly generated and real-world) compared with the worst-case algorithm. Using their guidelines and algorithm, we have implemented a working program that shows that their oracle can be efficiently implemented and evaluated on typical computing platforms. Our program makes it possible to decrease network downtime when a failure occurs, where the network can be anything from a computer to road network. If the problem can be related to a graph and the desired outcome is the shortest path and/or distance, then this algorithm can be used as a more efficient method to produce less down time.

1.2 Motivation

The motivation behind the implementation of this algorithm is justified by the number of applications that can benefit from a decrease in running time and space. One such application is that of vehicular traffic modeling on roads and highways. When there is an obstruction of some kind on a road way or at an intersection, our code can be used to find an alternate shortest path in constant time. This would be extremely helpful
to police and citizens as they can be rerouted by a police detour or a change on their GPS units in their cars. Another real world application would be that of computer networks. No computer is fool proof against downtime and if that occurs the end user is still expecting their data to be managed in a timely manner. With our code, a new network path between computers can be quickly rerouted while the broken computer is fixed.

1.3 Notations

All the notations used in this thesis are taken from [DTCR08]. We are given a non-negative edge-weighted, directed graph $G = (V, E, W)$. We use $m$ to denote the number of edges and $n$ to denote the number of vertices in $G$. W.l.o.g., we assume that all shortest paths in $G$ are unique and that $m \geq n - 1$. (For general graphs, the requirement that all shortest paths in $G$ are unique, can be enforced by having some mechanism for breaking any ties, e.g., by adding perturbations or by using lexicographic selection).

For any set of vertices $S$, $|S|$ denotes the number of vertices in $S$. For any vertex $v$, let $IN(v) = \{u \in V \mid (u, v) \in E\}$.

Let $\hat{G}$ denote the directed graph that is the same as $G$ except that the directions of edges in $\hat{G}$ are the reverse of those in $G$. The unique shortest path from any vertex $x$ to any vertex $y$ in $G$ is denoted by $\pi_{x,y}$ and in $\hat{G}$ is denoted by $\hat{\pi}_{x,y}$. The number of edges on any path $\pi$ is denoted by $|\pi|$. The length of a path $\pi$ is denoted by $W(\pi)$, where the length $W(\pi)$ is the summation of all the edge weights on the path $\pi$. For all vertices $x$ and $y$ of $G$, let $d_{x,y}$ denote $W(\pi_{x,y})$, the length of the shortest path (or shortest distance) between $x$ and $y$ in $G$. For all vertices $x$ and $y$ and subset $S$ of vertices of $G$, let $\pi_{x,y,S}$ denote the shortest path from $x$ to $y$ that avoids $S$ and let $d_{x,y,S}$ denote $W(\pi_{x,y,S})$, the length of the shortest path from $x$ to $y$ that avoids $S$. For brevity, we write $\pi_{x,y,v}$, where $v$ is a vertex in the subset $S$, as $\pi_{x,y,v}$ and write $d_{x,y,v}$ as $d_{x,y,v}$. Analogous terms ($\hat{d}_{x,y}$, $\hat{\pi}_{x,y,S}$, $\hat{d}_{x,y,S}$, and $\hat{d}_{x,y,v}$) are defined for the graph $\hat{G}$.

The shortest path tree rooted at a vertex $x$ in $G$ is denoted by $T_x$ and in $\hat{G}$ is denoted
by $\hat{T}_x$. For all vertices $x$ and $v$ of $G$, let $T_x(v)$ be the subtree of $T_x$ that is rooted at $v$. The subtree $\hat{T}_x(v)$ of $\hat{T}_x$ is defined analogously. The notations used throughout this thesis are summarized in Table 1.1.

Table 1.1: Notations used in this Thesis

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>Non-negative edge-weighted, directed graph $G = (V, E, W)$</td>
</tr>
<tr>
<td>$V$</td>
<td>Set of vertices in $G$</td>
</tr>
<tr>
<td>$E$</td>
<td>Set of edges in $G$</td>
</tr>
<tr>
<td>$n$</td>
<td>Total number of vertices in $G$</td>
</tr>
<tr>
<td>$m$</td>
<td>Total number of edges in $G$</td>
</tr>
<tr>
<td>$W[u, v]$</td>
<td>Weight of the directed edge $(u, v)$ in $G$</td>
</tr>
<tr>
<td>$\text{IN}(v)$</td>
<td>the set of vertices $u$ for which $(u, v)$ is an edge in $G$</td>
</tr>
<tr>
<td>$</td>
<td>\pi</td>
</tr>
<tr>
<td>$\pi_{x,y}$</td>
<td>The unique shortest path from a vertex $x$ to a vertex $y$ in $G$</td>
</tr>
<tr>
<td>$\pi_{x,y,S}$</td>
<td>The unique shortest path from a vertex $x$ to a vertex $y$ avoiding a set of vertices $S$ in $G$</td>
</tr>
<tr>
<td>$\pi_{x,y,v}$</td>
<td>The unique shortest path from a vertex $x$ to a vertex $y$ avoiding a vertex $v$ in $G$</td>
</tr>
<tr>
<td>$d_{x,y}$</td>
<td>The length of a shortest path from a vertex $x$ to a vertex $y$ in $G$</td>
</tr>
<tr>
<td>$d_{x,y,S}$</td>
<td>The length of a shortest path from a vertex $x$ to a vertex $y$ avoiding a set of vertices $S$ in $G$</td>
</tr>
<tr>
<td>$d_{x,y,v}$</td>
<td>The length of a shortest path from a vertex $x$ to a vertex $y$ avoiding a vertex $v$ in $G$</td>
</tr>
<tr>
<td>$T_x$</td>
<td>The shortest path tree rooted at a vertex $x$ in $G$</td>
</tr>
<tr>
<td>$T_x(v)$</td>
<td>The subtree of $T_x$ rooted at a vertex $v$ in $G$</td>
</tr>
</tbody>
</table>
CHAPTER 2
COMPUTING ALL-PAIRS OF SHORTEST PATHS

2.1 Definitions of the $D$, $H$, $P$, and $ST$ Tables

We run Dijkstra’s algorithm from all source vertices of the given directed graph $G$. This allows us to compute all-pairs of shortest paths in time $O(mn + n^2 \log n)$ or $\tilde{O}(mn)$ [Dij59]. In each shortest path computation, we fill in tables as described in [BK09]. These tables are $D$, $H$, $P$ and $ST$, as defined below.

Definition 2.1: For all vertices $x$ and $y$ of $G$, the following tables are defined:

- $D[x, y]$ stores the shortest distance from $x$ to $y$ in $G$ if it exists and stores $-1$ otherwise.
- $H[x, y]$ stores the number of edges on the (unique) shortest path $\pi_{x,y}$ if it exists and stores $-1$ otherwise.
- $P[x, y]$ stores the parent of $y$ on the (unique) shortest path $\pi_{x,y}$ if it exists and stores $-1$ otherwise.
- $ST[x]$ stores $T_x$, the shortest path tree rooted at $x$.

It is clear from the above definition that each of the tables $D$, $H$, $P$, and $ST$ takes $\Theta(n^2)$ space.

2.2 Creation of the $D$, $H$, $P$, and $ST$ Tables

Algorithm 1 (Initialize-Single-Source) takes as input a graph $G$ and a source vertex $s$ and it runs in $O(n)$ time. The algorithm initializes the values in the tables $D$, $H$, and $P$, for a fixed source vertex $s$ and all vertices $u$ of $G$. In Algorithm 2 (Relax), the
input consists of vertices $s$, $u$, and $v$, along with an array of edge weights $W$, and a min-priority queue $Q$. The algorithm compares the distance $D[s,v]$ against the value of $D[s,u]$ plus $W[u,v]$. If $D[s,v]$ is greater than $D[s,u] + W[u,v]$, then the following updates are performed: (1) If $D[s,v]$ is infinity, then $v$ is placed into $Q$ and (2) the shortest distance $D[s,v]$, the number of edges $H[s,v]$, and the parent $P[s,v]$ of $v$ on $\pi_{s,v}$ are set to $D[s,u] + W[u,v]$, $H[s,u] + 1$, and $u$, respectively. In Algorithm 3 (Construct-Shortest-Paths-Tree), the input consists of a graph $G$ and a source vertex $s$, and the output is $ST[s]$. The algorithm iterates over each vertex $u \neq s$ of $G$ and adds the edge $(P[s,u], u)$ into $ST[s]$. In Algorithm 4 (All-Pairs-Shortest-Paths), the input consists of a graph $G$, an array of edge weights $W$, and a min-priority queue $Q$. The algorithm loops through all vertices $s$ in $G$ to determine the shortest distances from $s$ to all vertices $y$ in $G$. It then makes a call to Algorithm 3 to construct the table $ST$.

<table>
<thead>
<tr>
<th>Input: Graph $G$, source vertex $s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Void</td>
</tr>
<tr>
<td>-------------------------------------</td>
</tr>
<tr>
<td>1 begin</td>
</tr>
<tr>
<td>2         foreach vertex $u$ in $G$ do</td>
</tr>
<tr>
<td>3         $D[s,u] \leftarrow \infty$;</td>
</tr>
<tr>
<td>4         $H[s,u] \leftarrow \infty$;</td>
</tr>
<tr>
<td>5         $P[s,u] \leftarrow NULL$;</td>
</tr>
<tr>
<td>6         end</td>
</tr>
<tr>
<td>7         $D[s,s] \leftarrow 0$;</td>
</tr>
<tr>
<td>8         $H[s,s] \leftarrow 0$;</td>
</tr>
<tr>
<td>9         end</td>
</tr>
</tbody>
</table>

Algorithm 1: Initialize-Single-Source
Input: Vertices $s$, $u$, and $v$, array of edge-weights $W$, min-priority queue $Q$

Output: Void

1 begin
2    if $D[s, v] > D[s, u] + W[u, v]$ then
3       if $D[s, v] == \infty$ then
4          $Q.Insert(v)$;
5          $D[s, v] \leftarrow D[s, u] + W[u, v]$;
6          $H[s, v] \leftarrow H[s, u] + 1$;
7          $P[s, v] \leftarrow u$;
8 end

Algorithm 2: Relax

Input: Graph $G$, source vertex $s$

Output: $ST[s]$

1 begin
2    $ST[s] \leftarrow NULL$;
3    foreach vertex $u$ in $G$ do
4       if $u \neq s$ then
5          Add edge $(P[s, u], u)$ in $ST[s]$;
6 end
7 return $ST[s]$;
8 end

Algorithm 3: Construct-Shortest-Paths-Tree
**Input:** Graph $G$, array of edge weights $W$, min-priority queue $Q$

**Output:** Tables $D$, $H$, $P$, $ST$

begin
    foreach vertex $s$ in $G$ do
        Initialize-Single-Source($G$, $s$);
        $Q$.Insert($s$);
        while !$Q$.empty() do
            $u$ ← $Q$.ExtractMin();
            foreach neighbor $v$ of $u$ do
                Relax($s$, $u$, $v$, $W$, $Q$);
            end
        end
        $ST[s]$ ← Construct-Shortest-Path-Tree($G$, $s$);
    end
    return $D$, $H$, $Pred$, $ST$;
end

Algorithm 4: All-Pairs-Shortest-Paths

*Analysis of Algorithm 1:* Every iteration of the For loop in Step 2 runs in time $O(1)$ and there are $n$ iterations. Hence, the running time of the algorithm is $O(n)$.

*Analysis of Algorithm 2:* It is clear that this algorithm runs in constant time.

*Analysis of Algorithm 3:* Every iteration of the For loop in Step 3 runs in time $O(1)$ and there are $n$ iterations. Thus, the running time of the algorithm is $O(n)$.

*Analysis of Algorithm 4:* Every iteration of the For loop in Step 2 calls first Algorithm 1 that takes $O(n)$ run time, followed by a While loop. Over all the iterations of the While loop, $Q$.ExtractMin is called $n$ times (once for each vertex), where each such call takes $O(\log n)$ worst-case time. Also, over all the iterations of the While loop, $Q$.DecrementKey is called $O(m)$ times, where each such call takes $O(1)$ amortized time assuming $Q$ is implemented as a Fibonacci min-heap. It follows that each iteration
of the For loop in Step 2 takes worst-case time $O(m + n \log n)$, assuming the Fibonacci min-heap implementation of $Q$. Thus, the total running time of the algorithm, using a Fibonacci min-heap implementation of $Q$, is $O(n(m + n \log n)) = \tilde{O}(mn)$.

The space used by each of the Algorithms 1, 2, 3, and 4 is $O(n^2)$. 
CHAPTER 3
ASSIGNING PRIORITIES

3.1 Definition of Centers

The next step involves assigning a priority to each vertex based on a random sampling approach [BK09]. All vertices start off with a priority of 1. Then, for all integer values of \( k \) in the range \([1, \log n]\), the priority of each vertex \( x \) is set to \( k \) with probability \( \Theta(1/2^k) \).\(^1\) If the probabilistic trial of assigning priority \( k \) fails, then the vertex \( x \) retains its current priority.

**Definition 3.1 ([BK08, BK09]):** Let \( 1 \leq k \leq \log n \) be an integer. A vertex \( x \) is said to be a \( k \)-center if its priority is \( k \). The set of all \( k \)-centers is denoted by \( R_k \). A vertex \( x \) is said to be a \( k^+ \)-center if its priority is at least \( k \).

The following requirements are important for the construction of a space-efficient distance oracle: For every integer \( 1 \leq k \leq \log n \),

- \(|R_k| = \tilde{O}(n/2^k)\).

- Any shortest path with \( \tilde{O}(2^k) \) vertices contains a \( k \)-center.

3.2 Assign-Priority Algorithm

In Algorithm 5 (Assign-Priority), the input is a graph \( G \) and the output is “Pass” or “Fail.” This algorithm visits each vertex in \( G \) and assigns a priority of \( k \) with probability \( 1/2^{k-1} \), where \( 1 \leq k \leq \log n \). If a vertex is assigned multiple priorities, the vertex keeps the highest assigned priority. Once each vertex \( x \) has been given a priority, test that all

\(^1f(n) \in \Theta(g(n)) \) iff there exists positive constants \( c_1 \) and \( c_2 \) and integer \( n_0 \) such that, for all integers \( n \geq n_0 \), it holds that \( c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \).
paths from $x$ of $2^k + 1$ edges has a vertex of priority more than $x.prior$. If this test fails, then the algorithm will clear the vertices of their priority and try once again for $10 \ln n$ times.
Input: Graph $G$

Output: “PASS” or “FAIL”

1 begin
2    for loop ← 1 to $10 \ln n$ do
3        foreach vertex $x$ in $G$ do
4            for $k ← 1$ to $\log n$ do
5                $x.priority ← \begin{cases} k & \text{with probability } 2^{1-k}, \\ x.priority & \text{otherwise}; \end{cases}$
6            end
7        end
8    end
9    test1 ← test2 ← True;
10   foreach vertex $x$ in $G$ do $R[x.priority] ← R[x.priority] + 1$;
11   for $k ← 1$ to $\log n$ do if $R[k] > \frac{4n}{2^k}$ then test1 ← False;
12   if (test1 == True) then
13      foreach vertex $x$ in $G$ do
14          $k ← x.priority$;
15          Perform a BFS in tree $T_x$ starting from $x$;
16          if there is a path $\pi$ in this BFS such that $|\pi| > 5 \cdot 2^k$ and no vertex in $\pi$ has priority $> k$ then test2 ← False;
17      end
18   end
19   if (test1 == True AND test2 == True) then return “PASS”;
20 end
21 return “FAIL”;
Lemma 3.2 ([BK09]): Algorithm 5 runs in time $O(n^2 \log n) = \tilde{O}(n^2)$ and returns “PASS” with probability $1 - O(1/n)$. If the algorithm returns “PASS” then the following holds: For every integer $1 \leq k \leq \log n$,

- $|R_k| \leq \frac{4n}{2^k}$.

- For every $k$-center $x$ and vertex $y$ such that $|\pi_{x,y}| > 5 \cdot 2^k$, there exists a $(k+1)^+$-center $z$ on $\pi_{xy}$.

Analysis of Algorithm 5: The For loop in Step 2 runs over $O(\log n)$ iterations. In each iteration of the For loop, Steps 3–10 take $O(n \log n)$ time and Steps 11–18 take $O(n^2)$ time. Therefore, the total running time of the algorithm is $O(\log n) \times O(n \log n + n^2) = O(n^2 \log^2 n) = \tilde{O}(n^2)$. It is clear that the space bound of the algorithm is $O(n)$. 

14
CHAPTER 4
COVERING CHAINS

4.1 Definitions of the $Cr$, $Cl$, and $BCP$ Tables

Once the priorities are assigned to all the vertices of the graph, a covering chain is created for each shortest path $\pi_{x,y}$ by selecting vertices by increasing order of priority on $\pi_{x,y}$. For every shortest path $\pi_{x,y}$, the first cover vertex is $x$, the second cover vertex is the first vertex succeeding $x$ on the path $\pi_{x,y}$ that has a higher priority than $x$, and so on. This process of designating cover vertices is repeated until vertex $y$ is reached. After the covering vertices for each shortest path in $G$ are found, the same process is repeated on each shortest path $\tilde{\pi}_{x,y}$ in the graph $\tilde{G}$. The set of vertices in between two adjacent cover vertices, say $s$ and $t$, on any shortest path in $G$ (\( \tilde{G} \)) is denoted by the interval $[s, t]$. The sequence of cover vertices $c_1, c_2, \ldots, c_j$ on any shortest path $\pi_{x,y}$ in $G$ ($\tilde{\pi}_{x,y}$ in $\tilde{G}$) is called a covering chain of $\pi_{x,y}$ (respectively, $\tilde{\pi}_{x,y}$). The intervals $[c_1, c_2], [c_2, c_3], \ldots, [c_{j-1}, c_j]$ partition the shortest path $\pi_{x,y}$ such that each $c_i$ covers all vertices in $[c_i, c_{i+1}]$, and so the intervals are referred to as covering intervals. The highest priority of the covering chain of $\pi_{x,y}$ is stored into $BCP[x, y]$ and used for finding covering vertices.

Definition 4.1 ([BK08, BK09]): For all vertices $x$ and $y$ of $G$ and integer $i$, where $1 \leq i \leq \log n$, the following tables are defined:

- $Cr[x, y, i]$ stores the first $i^+$-center $v \in V(G)$ on $\pi_{x,y}$ if it exists and stores $-1$ otherwise. $Cr$ stands for center right.

- $Cl[x, y, i]$ stores the first $i^+$-center $v \in V(G)$ on $\tilde{\pi}_{y,x}$ if it exists and stores $-1$ otherwise. $Cl$ stands for center left.
- \( \text{BCP}[x,y] \) stores the highest center priority on \( \pi_{x,y} \). The notation BCP stands for biggest center priority.

It is clear from the above definition that the tables \( Cr \) and \( Cl \) take \( \Theta(n^2 \log n) \) space and the table \( BCP \) takes \( \Theta(n^2) \) space.

4.2 Creation of the \( Cr \), \( Cl \), and \( BCP \) Tables

In Algorithm 6 (Create-Cr-and-BCP-Tables), the input is a graph \( G \) and the output consists of tables \( Cr \) and \( BCP \). The algorithm iterates over each shortest path \( \pi_{x,y} \) in the graph \( G \) and sets the covering vertices for each path starting with vertex \( x \). While traversing through each shortest path, the priority of each vertex is tested and the highest priority is stored for the given shortest path. In Algorithm 7 (Create-Cl-Table), the input is a graph \( G \) and the output is the table \( Cl \). The algorithm walks through each shortest path \( \hat{\pi}_{x,y} \) in the graph \( \hat{G} \) and sets the covering vertices starting with vertex \( y \).
**Input:** Graph $G$

**Output:** $Cr$ and $BCP$

begin

foreach vertex $x$ of $G$ do

  foreach vertex $y$ of $G$ do

    for $i \leftarrow 1$ to $\log n$ do

      $Cr[x, y, i] \leftarrow -1$;

    end

  end

for $i \leftarrow 1$ to $x.priority$ do

  $Cr[x, x, i] \leftarrow x$;

end

$BCP[x, x] \leftarrow x.priority$;

foreach vertex $y$ in the pre-order traversal of tree $T_x$ do

  $z \leftarrow P[x, y]$;

  $BCP[x, y] \leftarrow \max\{BCP[x, z], y.priority\};$

  for $i \leftarrow 1$ to $BCP[x, z]$ do

    $Cr[x, y, i] \leftarrow Cr[x, z, i]$;

  end

  for $i \leftarrow BCP[x, z] + 1$ to $y.priority$ do

    $Cr[x, y, i] \leftarrow y$;

  end

end

end

Algorithm 6: Create-Cr-and-BCP-Tables
Input: Graph $G$

Output: $Cl$

\begin{algorithm}
\begin{align*}
1 & \text{begin} \\
2 & \quad \text{foreach vertex } y \text{ of } G \text{ do} \\
3 & \quad \quad \text{foreach vertex } x \text{ of } G \text{ do} \\
4 & \quad \quad \quad \text{for } i \leftarrow 1 \text{ to } \log n \text{ do} \\
5 & \quad \quad \quad \quad Cl[x, y, i] \leftarrow -1; \\
6 & \quad \quad \quad \text{end} \\
7 & \quad \text{end} \\
8 & \quad \text{for } i \leftarrow 1 \text{ to } y.\text{priority} \text{ do} \\
9 & \quad \quad Cl[y, y, i] \leftarrow y; \\
10 & \quad \text{end} \\
11 & \quad \widehat{BCP}[y, y] \leftarrow y.\text{priority}; \\
12 & \quad \text{foreach vertex } x \text{ in the pre-order traversal of tree } \widehat{T}_y \text{ do} \\
13 & \quad \quad z \leftarrow \widehat{P}[y, x]; \\
14 & \quad \quad \widehat{BCP}[x, y] \leftarrow \max\{\widehat{BCP}[z, y], x.\text{priority}\}; \\
15 & \quad \quad \text{for } i \leftarrow 1 \text{ to } \widehat{BCP}[z, y] \text{ do} \\
16 & \quad \quad \quad Cl[x, y, i] \leftarrow Cl[z, y, i]; \\
17 & \quad \quad \text{end} \\
18 & \quad \quad \text{for } i \leftarrow \widehat{BCP}[z, y] + 1 \text{ to } x.\text{priority} \text{ do} \\
19 & \quad \quad \quad Cl[x, y, i] \leftarrow x; \\
20 & \quad \quad \text{end} \\
21 & \quad \text{end} \\
22 & \text{end} \\
\end{align*}
\caption{Create-Cl-Table}
\end{algorithm}
4.3 Finding Centers on Both Sides of an Avoiding Vertex

Algorithm 8 (Find-Centers) takes input a graph $G$, vertices $x$ and $y$, and a vertex $v$ to avoid on the path $\pi_{x,y}$. It outputs the pair $(c_x, c_y)$ of centers that cover $v$ and are on either sides of vertex $v$ on the path $\pi_{x,y}$.

**Input:** Graph $G$, vertices $x$ and $y$, and vertex $v$ to avoid on the path $\pi_{x,y}$

**Output:** Center pairs $(c_x, c_y)$ such that $v$ is a vertex in the covering interval $[c_x, c_y]$ on the covering chain for $\pi_{x,y}$

1. $i \leftarrow BCP[x, v]$;
2. $j \leftarrow BCP[v, y]$;
3. $c_x \leftarrow Cr[x, y, i]$;
4. if $(i > j$ or $i == j)$ then $c_y \leftarrow Cl[x, y, j]$;
5. else $c_y \leftarrow Cr[v, y, i + 1]$;
6. return $(c_x, c_y)$;

Algorithm 8: Find-Centers

*Analysis of Algorithm 6:* The For loop in Step 2 runs over $O(n)$ iterations. In each iteration of the For loop, Steps 3–7 take $O(n \log n)$ time, Steps 8–11 take $O(\log n)$ time, and Steps 12–21 take $O(n \log n)$ time. Hence, the total running time of the algorithm is $O(n^2 \log n) = \tilde{O}(n^2)$. The space bound of the algorithm is $O(n^2 \log n) = \tilde{O}(n^2)$.

*Analysis of Algorithm 7:* The analysis of this algorithm is the same as that of Algorithm 6. Therefore, the algorithm runs in time $\tilde{O}(n^2)$ and uses space $\tilde{O}(n^2)$.

*Analysis of Algorithm 8:* It is clear that this algorithm runs in time $O(1)$, as all data is pulled from tables already computed. The space bound of the algorithm is $O(1)$. 

19
CHAPTER 5

SHORTEST DISTANCES FROM ALL CENTERS AVOIDING THEIR COVERED VERTICES

5.1 Definitions of the $D_k$, $D_kE$, $\hat{D}_k$ and $\hat{D}_kE$ Tables

With the covering chain created for every shortest path $\pi_{x,y}$, the next step is to store the shortest distance $d_{c,y,v}$ and the first edge of the path $\pi_{c,y,v}$ into tables. Here, $c$ is a chosen center, $y$ is any vertex in the shortest path tree $T_c(v)$, and $v$ is any vertex that $c$ covers, which must lie within the first $5 \cdot 2^k$ levels of $T_c$.

**Definition 5.1 ([BK08, BK09]):** A vertex $c$ is said to cover a vertex $v$ in $G$ if we store $d_{c,y,v}$ for every $y \in T_c(v)$. Similarly, a vertex $c$ is said to cover a vertex $v$ in $\hat{G}$ if we store $\hat{d}_{c,y,v}$ for every $y \in \hat{T}_c(v)$.

In other words, $c$ covers $v$ in $G$ if we store the lengths of the shortest paths from $c$ to all vertices $y \in T_c(v)$ avoiding $v$ in $G$. Likewise, $c$ covers $v$ in $\hat{G}$ if we store the lengths of the shortest paths from $c$ to all vertices $y \in \hat{T}_c(v)$ avoiding $v$ in $\hat{G}$.

**Definition 5.2 ([BK08, BK09]):** For every integer priority $1 \leq k \leq \log n$, $k$-center $c$, and for all vertices $v$ and $y$ of $G$ such that $y \in T_c(v) - \{v\}$, we have

- If $c$ covers $v$ in $G$, then $D_k[c,y,v]$ stores $d_{c,y,v}$.
- If $c$ covers $v$ in $G$, then $D_kE[c,y,v]$ stores the first edge $(c,u) \in \pi_{c,y,v}$.
- If $c$ covers $v$ in $\hat{G}$, then $\hat{D}_k[c,y,v]$ stores $\hat{d}_{c,y,v} = d_{y,c,v}$.
- If $c$ covers $v$ in $\hat{G}$, then $\hat{D}_kE[c,y,v]$ stores the first edge $(c,u) \in \pi_{c,y,v}$.

The total space needed for each of the tables $D_k$, $D_kE$, $\hat{D}_k$ and $\hat{D}_kE$ is $\sum_{k=1}^{\log n} (4n/2^k) \times n \times (5 \cdot 2^k) = 20n^2 \log n = O(n^2 \log n) = \tilde{O}(n^2)$. 

20
5.2 Creation of the $D_k$, $D_kE$, $\hat{D}_k$ and $\hat{D}_kE$ Tables

Algorithm 9 (Create-$D_k$-and-$D_kE$-Tables) takes as input a graph $G$ and returns as output the Tables $D_k$ and $D_kE$. The algorithm iterates over each center $c$ and each vertex $v$ in $T_c$, and creates a new edge-weighted graph $G_v = (V_v, E_v, W_v)$ whose vertex set $V_v$ and edge set $E_v$ are defined as follows: $V_v$ contains $c$ and the set $U_v$ of vertices in $T_c(v) \setminus \{v\}$ and $E_v$ contains an edge from $c$ to each vertex in $U_v$ and also the edges in $G$ induced by $U_v$. Once the graph $G_v$ is constructed, Dijkstra’s algorithm is run with center $c$ as the source vertex; all shortest distances are stored in the table $D_k$ and the first edges of the new shortest paths are stored in the table $D_kE$. The tables $\hat{D}_k$ and $\hat{D}_kE$ are created by running the same algorithm on the graph $\hat{G}$. 
Input: Graph $G$

Output: $D_k$ and $D_kE$

begin

foreach vertex $c$ of $G$ do

$k \leftarrow c \text{.priority}$

for $L \leftarrow 1$ to $5 \cdot 2^k$ do

foreach vertex $v$ of $T_c$ at level $L$ do

if $(v \text{.priority} \leq k)$ then

$U_v \leftarrow$ set of all vertices of $T_c(v)$ except the vertex $v$;

Construct a directed weighted graph $G_v = (V_v, E_v, W_v)$:

- $V_v = U_v \cup \{c\}$
- $E_v$ contains an edge from $c$ to each vertex in $U_v$ and also contains the edges in $G$ induced by $U_v$
- $W_v[a, b]$ is the weight of the edge $(a, b)$ in $G_v$ defined as

$$W_v[a, b] = \begin{cases} 
\min_{x \in T_c(v)} \{d_{c,x} + W[x, b]\} & \text{if } a = c, \\
W[a, b] & \text{otherwise},
\end{cases}$$

where we assume that $d_{c,x} = +\infty$ if $x$ is not reachable from $c$ in $G$ and $W[x, b] = +\infty$ if $(x, b)$ is not an edge of $G$;

foreach vertex $y \in U_v$ do

$D_k[c, y, v] \leftarrow$ the shortest distance from $c$ to $y$ in $G_v$;

$D_kE[c, y, v] \leftarrow$ the first edge in the shortest path from $c$ to $y$ in $G_v$;

end

end

end

end

Algorithm 9: Create-$D_k$-and-$D_kE$-Tables
Analysis of Algorithm 9: In every iteration of the For loop in Step 2 and every iteration of the For loop in Step 4, the total computation time of the For loop in Step 5 is $O(m+n \log n)$. Since there are at most $4n/2^k$ centers of priority $k$, the total computation time of the algorithm is at most $\sum_{k=1}^{\log_2 n} (4n/2^k) \times (5\cdot2^k) \times O(m+n \log n) = O(nm \log n + n^2 \log^2 n)$, which is $\tilde{O}(mn)$ as $m \geq n - 1$ by the assumption made in Section 1.3. The space used by the algorithm is mainly in maintaining the tables. Therefore, the total space requirement of the algorithm is $\tilde{O}(n^2)$.
CHAPTER 6
SHORTEST DISTANCES FROM ALL VERTICES AVOIDING THE
FIRST EDGES OF THEIR SHORTEST PATHS

6.1 Definitions of the $De$, $DeE$, $\hat{De}$, and $\hat{DeE}$ Tables

The previous chapter shows how shortest distances from all centers avoiding their covered vertices are computed and stored in the tables. Here, we compute the shortest distances from all vertices avoiding the first edges on their shortest paths using Algorithm 10 (Create-$De$-and-$DeE$-Tables). This algorithm follows the same principles as Algorithm 9 and they can be combined. However, we present the two algorithms separately to keep the process clear and understandable. Algorithm 10 stores the information in the tables defined below.

Definition 6.1 ([DTCR08, BK08]): For all vertices $x$ and $y$ of $G$, let $e_{x,y}$ ($\hat{e}_{x,y}$) denote the first edge of $\pi_{x,y}$ (respectively, $\hat{\pi}_{x,y}$). The following tables are defined: For all vertices $x$ and $y$ of $G$,

- $De[x, y]$ stores the shortest distance from $x$ to $y$ in $G - \{e_{x,y}\}$.
- $DeE[x, y]$ stores the first edge on the shortest path from $x$ to $y$ in $G - \{e_{x,y}\}$.
- $\hat{De}$ stores the shortest distance from $x$ to $y$ in $\hat{G} - \{\hat{e}_{x,y}\}$.
- $\hat{DeE}[x, y]$ stores the first edge on the shortest path from $x$ to $y$ in $\hat{G} - \{\hat{e}_{x,y}\}$.

It is clear from the above definition that each of these tables takes $\Theta(n^2)$ space.
6.2 Creation of the $De$, $DeE$, $\hat{De}$, and $\hat{DeE}$ Tables

Algorithm 10 (Create-$De$-and-$DeE$-Tables) takes as input a graph $G$ and returns as output the tables $De$ and $DeE$. The algorithm iterates over each vertex $x$ and every vertex $v$ such that the edge $(x, v)$ is in $T_x$, and creates a new edge-weighted graph $G_v = (V_v, E_v, W_v)$ whose vertex set $V_v$ and edge set $E_v$ are defined as follows: $V_v$ contains $x$ and the set $U_v$ of all vertices in $T_x(v)$ and $E_v$ contains an edge from $x$ to each vertex in $U_v$ except for the original edge $(x, v)$ (a new edge weight is assigned) and also contains the edges in $G$ induced by $U_v$. Once the graph $G_v$ is constructed, Dijkstra's algorithm is run with $x$ as the source vertex; all shortest distances are stored in the table $De$ and the first edges of the new shortest paths are stored in the table $DeE$. The tables $\hat{De}$ and $\hat{DeE}$ are created by running the same algorithm on the graph $\hat{G}$. 
Input: Graph $G$

Output: $De$ and $DeE$

1 begin
2 foreach vertex $x$ of $G$ do
3     foreach vertex $v$ such that the edge $(x,v)$ is in $T_x$ do
4         $U_v \leftarrow$ set of all vertices of $T_x(v)$;
5     Construct a directed weighted graph $G_v = (V_v, E_v, W_v)$:
6         • $V_v = U_v \cup \{x\}$
7         • $E_v$ contains an edge from $x$ to each vertex in $U_v$
8             except for the original edge weight $(x,v)$ and also
9             contains the edges in $G$ induced by $U_v$
10        • $W_v[a,b]$ is the weight of the edge $(a,b)$ in $G_v$ defined as
11            
12                $W_v[a,b] = \begin{cases} 
13                            \min_{z \notin T_x(v)} \{d_{x,z} + W[z,b]\} & \text{if } a = x, \\
14                            W[a,b] & \text{otherwise}, 
15                \end{cases}$
16
17        where we assume that $d_{x,z} = +\infty$ if $z$ is not reachable from $x$ in $G$
18        and $W[z,b] = +\infty$ if $(z,b)$ is not an edge of $G$
19
20     foreach vertex $y \in U_v$ do
21         $De[x,y] \leftarrow$ the shortest distance from $x$ to $y$ in $G_v$;
22         $DeE[x,y] \leftarrow$ the first edge in the shortest path from $x$ to $y$ in $G_v$;
23     end
24 end
25 end

Algorithm 10: Create-$De$-and-$DeE$-Tables

Analysis of Algorithm 10: In every iteration of the For loop in Step 2, the total computation time of the For loop in Step 3 is $O(m + n \log n)$. Since there are $n$ possible source vertices, the total running time of the algorithm is $O(n) \times O(m + n \log n) = O(mn + n^2 \log n)$, which is $\tilde{O}(mn)$ as $m \geq n - 1$ by the assumption made in Section 1.3. The space used
by the algorithm is mainly in maintaining the tables. Therefore, the total space bound of the algorithm is $O(n^2)$. 
CHAPTER 7
BOTTLENECK VERTICES

7.1 Definition of the BV Table

With the Cr and Cl tables containing the covering intervals for each shortest path \( \pi_{x,y} \), the algorithm then declares one vertex of each covering interval \( I \) to be the bottleneck vertex of \( I \) with respect to its endpoints. The bottleneck vertex \( w \) of any interval \( I \) on any shortest path \( \pi_{x,y} \) is some vertex in \( I \) whose removal from the graph results in the maximum shortest distance between \( x \) and \( y \).

**Definition 7.1 (Bottleneck Vertex [BK09]):** A vertex \( w \) is labeled a bottleneck vertex of a given interval \( I \) of \( \pi_{x,y} \) if and only if \( w = \arg\max_{v \in I} \{d_{x,y,v}\} \).

The following lemma expresses \( d_{x,y,v} \) in terms of \( d_{x,y,w} \), where \( w \) is the bottleneck vertex of the interval \( I \) that \( v \) belongs to on \( \pi_{x,y} \). This lemma is the cornerstone of the nearly optimal oracle.

**Bottleneck Lemma (Lemma 1.1) Restated:** Let \( x, s, v, t, \) and \( y \) be vertices in that order on the shortest path \( \pi_{x,y} \) from \( x \) to \( y \), where \( v \) is the failed vertex and \( s \neq v \neq t \). Let \( w \) be the bottleneck vertex of the interval \([s,t]\). Then, \( d_{x,y,v} = \min\{d_{x,s} + d_{s,y,v}, d_{x,t,v} + d_{t,y}, d_{x,y,w}\} \).

**Definition 7.2 ([BK09]):** The table \( BV \) is defined as follows: For all vertices \( x \) and \( y \) of \( G \) and integer priority \( 1 \leq i \leq \log n \),

- \( BV[x,y,i] \) stores the bottleneck vertex of the \( i \)'th covering interval on the covering chain for \( \pi_{x,y} \).

It it clear from the above definition that the table \( BV \) takes \( O(n^2 \log n) = \tilde{O}(n^2) \).
7.2 Creation of the $BV$ Table

In Algorithm 11 (MTC), the input consists of vertices $x$ and $y$ and a vertex $v$ to avoid. The algorithm returns the length of the shortest path from $x$ to $y$ avoiding $v$, but passing through the two centers $c_x$ and $c_y$ covering $v$. For Algorithm 12 (Find-Bot), the input consists of vertices $x$ and $y$, and a subinterval $I = [s, t]$ on the covering chain for $\pi_{x,y}$ to perform the search for the bottleneck vertex. The output of the algorithm is a vertex $w$ that is the bottleneck vertex of $I$ (with respect to $x$ and $y$). The algorithm is a recursive binary search that is performed on the interval $[s, t]$ by breaking the interval into two subintervals, then comparing the distance values for the subintervals, and finally choosing one subinterval based on the comparison. When the recursive call reaches an interval with at most two vertices, then a candidate for the bottleneck vertex is returned using an exhaustive search over the interval. In Algorithm 13 (Create-BV-Table), the input is a graph $G$ and the output is the bottleneck vertex of every interval $I$ of every shortest path $\pi_{x,y}$. The algorithm iterates over all vertices $x$ and $y$, and over all intervals $I$ on the shortest path $\pi_{x,y}$.

| Input: Vertices $x$ and $y$, vertex $v$ to avoid |
| Output: Length of the shortest path from $x$ to $y$ avoiding $v$, but passing through the two centers $c_x$ and $c_y$ covering $v$ |

\[ (c_x, c_y) \leftarrow \text{Find-Centers}(x, y, v); \]

\[ \text{return } \min\{d_{x,c_x} + d_{c_x,y,v}, d_{x,c_y,v} + d_{c_y,y}\}; \]

Algorithm 11: MTC
Input: Vertices $x$ and $y$, centers $c_x$ and $c_y$ of a covering interval $I = [c_x, c_y]$ on the covering chain for $\pi_{x,y}$, the covering interval $I$, and indices low and high, where low $\leq$ high, of a subinterval $I[\text{low} \ldots \text{high}]$ of $I$.

Output: A vertex $w = \arg\max_{v \in I[\text{low} \ldots \text{high}]} MTC(x, y, v)$

1 if $|I[\text{low} \ldots \text{high}]| \leq 2$ then return $\arg\max_{w \in I[\text{low} \ldots \text{high}]} (MTC(x, y, w))$;

2 mid $\leftarrow \lfloor (\text{low} + \text{high})/2 \rfloor$;

3 $v \leftarrow \arg\max_{w \in I[\text{mid} \ldots \text{high}]} (d_{c_x,y,w})$;

4 $L(x, y, v) \leftarrow d_{x,c_x} + d_{c_x,y,v}$;

5 $R(x, y, v) \leftarrow d_{x,c_y,v} + d_{c_y,y}$;

6 if $L(x, y, v) \leq R(x, y, v)$ then

7 $w \leftarrow \text{Find-Bot}(x, y, c_x, c_y, I, \text{low}, \text{mid})$;

8 $w' \leftarrow \arg\max_{v,w} \{MTC(x, y, v), MTC(x, y, w)\}$;

9 end

10 if $L(x, y, v) > R(x, y, v)$ then $w' \leftarrow \text{Find-Bot}(x, y, c_x, c_y, I, \text{mid}, \text{high})$;

11 return $w'$

Algorithm 12: Find-Bot
**Input**: Graph $G$

**Output**: The bottleneck vertex of every covering interval $I$ of every shortest path $\pi_{x,y}$

1. **foreach** vertex $x$ of $G$ **do**

2.     **foreach** vertex $y$ of $G$ **do**

3.         **for** $i \leftarrow 1$ **to** $\log n$ **do**

4.             $c_x \leftarrow Cr[x,y,i]$;

5.             $c_y \leftarrow Cr[x,y,i+1]$;

6.             $BV[x,y,i] \leftarrow \text{Find-Bot}(x,y,c_x,c_y,[c_x,c_y],0,H[c_x,c_y])$;

7.         **end**

8. **end**

9. **end**

**Algorithm 13**: Create-BV-Table

**Analysis of Algorithm 11**: It is clear that this algorithm runs in time $O(1)$, as all data is pulled from tables already computed. The space bound of the algorithm is $O(1)$.

**Analysis of Algorithm 12**: The algorithm is much like a recursive binary search in that it recurses on half-intervals. This leads to an $O(\log n)$ total running time to recurse through the interval and find the bottleneck vertex. Each invocation of the algorithm takes only $O(1)$ running time, as Step 4 uses RMQ data structure to find the maximum value in a subarray in constant time. Since there are $O(\log n)$ recursive calls, the total running time of the algorithm is $O(\log n)$. The space bound of the algorithm is $O(n)$.

**Analysis of Algorithm 13**: Since the total number of calls to Algorithm 12 is $O(n^2 \log n)$ and each call takes $O(\log n)$ time, the total running time of the algorithm is $O(n^2 \log^2 n) = \tilde{O}(n^2)$. The space bound of the algorithm is $\tilde{O}(n^2)$.
CHAPTER 8
SHORTEST DISTANCES BETWEEN ALL PAIRS OF VERTICES
AVOIDING THE BOTTLENECK VERTICES

8.1 Definition of the DBV and FEBV Tables

With the bottleneck vertices stored in the table $BV$, the oracle then determines, for each pair of vertices $x$ and $y$ and for each covering interval $i$, the shortest distance from $x$ to $y$ while avoiding the bottleneck vertex $BV[x, y, i]$ of $i$ (with respect to $x$ and $y$) and stores this distance into a table $DBV$. To get the shortest distances between all pairs of vertices while avoiding the bottleneck vertices with respect to the pairs, a new non-negative edge-weighted, directed graph $G_{bv} = (V_{bv}, E_{bv}, W_{bv})$ is created. The vertex set $V_{bv}$ consists of a source $s$ and the vertices $v[x, y, i]$, for all vertices $x$ and $y$ and for every integer priority $1 \leq i \leq \log n$ in the original graph $G$. The goal behind the creation of $G_{bv}$ is to reduce the computation of $DBV[x, y, i] = d_{x, y, BV[x, y, i]}$ to the computation of the shortest distance from $s$ to $v[x, y, i]$.

During the creation of $DBV$, an additional table $FEBV$ will be created. The entry $FEBV[x, y, i]$ will store the first edge on the shortest path from $x$ to $y$ avoiding the bottleneck vertex of the $i$’th covering interval on $\pi_{x,y}$. This table will be used for answering queries that deal with finding the shortest path between two vertices for a given failed vertex or edge.

**Definition 8.1 ([BK09]):** For all vertices $x$ and $y$ of $G$ and integer $1 \leq i \leq \log n$, the following tables are defined:

- $DBV[x, y, i]$ stores the shortest distance from $x$ to $y$ avoiding the bottleneck vertex $BV[x, y, i]$ of the $i$’th covering interval on the covering chain for $\pi_{x,y}$.
\* \* FEBV \[x,y,i\] stores the first edge of the shortest path from \(x\) to \(y\) avoiding the bottleneck vertex \(BV[x,y,i]\) of the \(i\)’th covering interval on the covering chain for \(\pi_{x,y}\).

It is clear from the above definition that each of the tables \(DBV\) and \(FEBV\) takes \(O(n^2 \log n) = \tilde{O}(n^2)\) space.

**Lemma 8.2 (Bottleneck Values [BK09]):** For all vertices \(x\) and \(y\) of \(G = (V,E,W)\) and integer \(1 \leq i \leq \log n\), the bottleneck value \(DBV[x,y,i] = d_{x,y,BV[x,y,i]}\) is given by

\[
DBV[x,y,i] = \min\{ \min_{y' \in IN(y)} (MTC(x,y',BV[x,y,i]) + W[y',y]) \text{ (term 1)}, \\
\min_{y' \in IN(y)} (DBV[x,y',j] + W[y',y]) \text{ (term 2)} \},
\]

where \(j\) in \(DBV[x,y',j]\) is the center priority for which \(BV[x,y,i]\) is in the \(j\)’th covering interval on the covering chain for \(\pi_{x,y'}\) in \(G\).

### 8.2 Creation of the DBV and FEBV Tables

In Algorithm 14 (Create-DBV-and-FEBV-Tables), the input is a graph \(G = (V,E,W)\) and the output consists of tables \(DBV\) and \(FEBV\). The algorithm implicitly maintains a new non-negative edge-weighted, directed graph \(G_{bv} = (V_{bv},E_{bv},W_{bv})\) that contains a source vertex \(s\) and vertices \(v[x,y,i]\) corresponding to bottleneck vertices \(BV[x,y,i]\) in the original graph \(G\). An edge from the source vertex \(s\) to each vertex \(v[x,y,i]\) is implicitly created and the weight of this edge is set to the minimum of \(MTC(x,y',v) + W[y',y]\) over all \(y' \in V\) such that \((y',y) \in E\) and \(v\) is the bottleneck vertex \(BV[x,y,i]\) of the \(i\)’th covering interval on the covering chain for \(\pi_{x,y}\) in \(G\). An edge is implicitly added from \(v[x,y',j]\) to \(v[x,y,i]\) if \(y' \in IN(y)\) and \(j\) is the index of the covering interval on the covering chain for \(\pi_{x,y'}\) in \(G\) that contains \(BV[x,y,i]\). With all the edges implicitly created in the graph \(G_{bv}\), Dijkstra’s algorithm is then run on \(G_{bv}\) with \(s\) as the start vertex. The shortest distances from \(s\) to all vertices of \(G_{bv}\) are stored in the table \(DBV\) and the first edge of the shortest paths are stored in the table \(FEBV\).
Define a directed weighted graph $G_{bv} = (V_{bv}, E_{bv}, W_{bv})$ with a designated source vertex $s$ as follows:

- $V_{bv} = \{s\} \cup \{v[x, y, i] \mid x, y \in V \text{ and } 1 \leq i \leq \log n \text{ is an integer priority}\}$;
- $E_{bv}$ contains an edge from $s$ to each vertex $v[x, y, i]$ and an edge from each $v[x, y', j]$ to $v[x, y, i]$, where $y' \in IN(y)$ and $j$ is the interval on $\pi_{x,y'}$ in $G$ that contains $BV[x, y, i]$.
- $W_{bv}$ is a non-negative weight function on edges of $G_{bv}$ such that
  
  (a) $W_{bv}[s, v[x, y, i]] = \min_{y' \in IN(y)} \{MTC(x, y', BV[x, y, i]) + W[y', y]\}$ and
  
  (b) $W_{bv}[v[x, y', j], v[x, y, i]] = W[y', y]$ if $y' \in IN(y)$ and $j$ is the center priority for which $BV[x, y, i]$ is in the $j$'th covering interval on the covering chain for $\pi_{x,y'}$ in $G$.

Note that there may be some $y' \in IN(y)$ for which $BV[x, y, i]$ is not on $\pi_{x,y'}$ in $G$; so, for such $y'$, $MTC(x, y', BV[x, y, i])$ and $BV[x, y', j]$ are not defined. In this case, it is easy to see that $d_{x,y',s}$ equals $d_{x,y'}$. Thus, this special case is handled by defining $MTC(x, y', BV[x, y, i])$ to be $d_{x,y'}$ and defining $DBV[x, y', j]$ to be infinity.

```
foreach (vertices $x, y \in V$ and integer priority $1 \leq i \leq \log n$) do
    $DBV[x, y, i] \leftarrow$ the shortest distance from $s$ to $v[x, y, i]$ in $G_{bv}$;
    $FEBV[x, y, i] \leftarrow$ the first edge on the shortest path from $s$ to $v[x, y, i]$ in $G_{bv}$;
end
```

Algorithm 14: Create-$DBV$-and-$FEBV$-Tables

Analysis of Algorithm 14: Since each vertex $y$ in $G$ is part of $O(n \log n)$ triplets of the form $(x, y, i)$, the number of edges in $G_{bv}$ is $O(n \log n \sum_{y \in V} |IN(y)|) = O(mn \log n)$ or $\tilde{O}(mn)$. It is clear that the number of vertices in $G_{bv}$ is $O(n^2 \log n)$. Once $G_{bv}$ is constructed, Dijkstra’s algorithm is then run on input $G_{bv}$ and $s$, which takes time.
\(O(mn \log n + n^2 \log^2 n)\), which is \(\tilde{O}(mn)\) as \(m \geq n - 1\) by the assumption made in Section 1.3. Thus, the total running time of the algorithm is \(\tilde{O}(mn)\). Notice that the edge information of \(G_{bv}\) is not explicitly stored, rather edges leaving any vertex in \(G_{bv}\) are computed on-the-fly using the definition of \(G_{bv}\) given in Algorithm 14. Thus, the total space bound of the algorithm is \(O(|V_{bv}|) = O(n^2 \log n) = \tilde{O}(n^2)\).
CHAPTER 9
ANSWERING QUERIES

9.1 Types of Queries

There are a total of four different queries that can be asked to the oracle. The first is a query asking for the shortest distance from any vertex $x$ to any vertex $y$ avoiding any failed vertex $v$. The pseudocode for answering this query is taken from Section 6 of [BK08], with one exception that the table $EP[x, y, i]$ is replaced by [BK09]’s table $DBV[x, y, i]$. The second query asks for the shortest distance from any vertex $x$ to any vertex $y$ avoiding any failed edge $(u, v)$. The pseudocode for answering this query is taken from Figure 6.1 of [DTCR08], replacing the “v-dist$(x, y, u)$” function call with the query for the shortest distance from $x$ to $y$ avoiding the failed vertex $u$. The third query asks for the shortest path $\pi_{x,y,v}$ from any vertex $x$ to any vertex $y$ avoiding any failed vertex $v$. This query also uses the shortest distance avoiding a failed vertex, but based on which term is chosen from the “min” function in the Bottleneck Lemma (Lemma 1.1), an edge of the path $\pi_{x,y,v}$ is returned [DTCR08]. Since only a single edge is returned, this query must be ran $O(L)$ times, where $L$ is the number of edges in the shortest path $\pi_{x,y,v}$. The last query asks for the shortest path from any vertex $x$ to any vertex $y$ avoiding any failed edge $(u, v)$. The pseudocode for answering this query follows the same steps as for the query for the shortest distance avoiding a failed edge. Based on which term is chosen from the “min” function in the Bottleneck Lemma (Lemma 1.1), a single edge is returned [DTCR08]. This query must also be ran $O(L)$ times to return each edge of the shortest path avoiding a failed edge.
9.2 Query: Shortest Distance Avoiding A Failed Vertex

Algorithm 15 (Shortest-Distance-Avoiding-Vertex) answers queries that ask for the shortest distance avoiding a single failed vertex. The input to the algorithm consists of vertices \( x, y, \) and \( v \) of a graph \( G \) and the output is the shortest distance from \( x \) to \( y \) avoiding \( v \). The algorithm first determines whether \( v \), the vertex to avoid, is on the path \( \pi_{x,y} \). If \( v \) is not on \( \pi_{x,y} \), then the shortest distance \( d_{x,y} \) is returned. Otherwise, the algorithm finds the endpoints, \( c_x \) and \( c_y \), of the covering interval that \( v \) belongs to on \( \pi_{x,y} \). The algorithm then compares three possible distances from \( x \) to \( y \) that avoid \( v \), as stated in the Bottleneck Lemma (Lemma 1.1), and returns the minimum of the three distances.

\[
\begin{align*}
\text{Input:} & \text{ Vertices } x, y, \text{ and } v \text{ of a graph } G, \text{ where } v \text{ is the vertex to avoid} \\
\text{Output:} & \text{ Shortest distance from } x \text{ to } y \text{ avoiding } v \text{ in graph } G \\
1 & \text{ if } d_{x,v} + d_{v,y} > d_{x,y} \text{ then} \\
2 & \quad \text{return } d_{x,y}; \\
3 & \text{ end} \\
4 & i \leftarrow BCP[x,v]; \\
5 & j \leftarrow BCP[v,y]; \\
6 & \text{ if } i > j \text{ then} \\
7 & \quad \text{break and compute } \hat{d}_{y,x,v} \text{ instead}; \\
8 & \text{ end} \\
9 & c_x \leftarrow Cr[x,y,i]; \\
10 & \text{ if } i = j \text{ then } c_y \leftarrow Cl[x,y,j]; \\
11 & \text{ else } c_y \leftarrow Cr[v,y,i+1]; \\
12 & \text{return } \min\{d_{x,c_x} + D_i[c_x,y,v]), (d_{c_y,y} + \hat{D}_j[c_y,x,v]), DBV[x,y,i]\}; \\
\end{align*}
\]

Algorithm 15: Shortest-Distance-Avoiding-Vertex
Analysis of Algorithm 15: Each step calls upon a variable in an already created table that takes $O(1)$ time to pull the data from the table. Therefore, the total run time is $O(1)$.

### 9.3 Query: Shortest Distance Avoiding A Failed Edge

Algorithm 16 (Shortest-Distance-Avoiding-Edge) answers queries that ask for the shortest distance avoiding a single failed edge. The input consists of vertices $x$ and $y$ and an edge $(u,v)$ of a graph $G$, and the output is the shortest distance from $x$ to $y$ avoiding the edge $(u,v)$. The algorithm first determines whether $(u,v)$, the edge to avoid, is on the path $\pi_{x,y}$. If $(u,v)$ is not on $\pi_{x,y}$, then the distance $d_{x,y}$ is returned. Otherwise, the algorithm calls Algorithm 15 by passing in the vertices $x$ and $y$ along with the first vertex of the failed edge $(u,v)$. The distance returned from this call is then compared against the combined distance of $d_{x,u}$ and $d_{u,y,v}$, and the minimum of the two distances is returned.

```
Input: Vertices $x$ and $y$ and an edge $(u,v)$ of a graph $G$, where $(u,v)$ is the edge to avoid
Output: Shortest distance from $x$ to $y$ avoiding the edge $(u,v)$ in graph $G$

1  if $d_{x,u} + W[u,v] + d_{v,y} > d_{x,y}$ then
2        return $d_{x,y}$;
3  end
4  $d_1 \leftarrow$ Shortest-Distance-Avoiding-Vertex($x,y,u$);
5  $d_2 \leftarrow d_{x,u} + D_e[u,y]$;
6  return $\min\{d_1,d_2\}$;
```

Algorithm 16: Shortest-Distance-Avoiding-Edge

Analysis of Algorithm 16: Each step calls upon a variable in an already created table that takes $O(1)$ time to pull the data from the table. Therefore, the total run time is $O(1)$.

38
9.4 Query: Shortest Path Avoiding A Failed Vertex

Algorithm 18 (Create-Path-Avoiding-Vertex) answers queries that ask for the shortest path avoiding a failed vertex. The input to the algorithm consists of vertices \(x\), \(y\) and \(v\) of a graph \(G\), and the output is the shortest path from \(x\) to \(y\) avoiding \(v\). The algorithm repeatedly calls Algorithm 17 (Shortest-Path-Avoiding-Vertex) to construct the path \(\pi_{x,y,v}\). Algorithm 17 first determines whether \(v\), the vertex to avoid, is on the path \(\pi_{x,y}\). If \(v\) is not on \(\pi_{x,y}\), then the first edge on the path \(\pi_{x,y}\) is returned. Otherwise, the algorithm finds the endpoints, \(c_x\) and \(c_y\), of the covering interval that \(v\) belongs to on \(\pi_{x,y}\). The algorithm then compares three possible distances from \(x\) to \(y\) that avoid \(v\), as stated in the Bottleneck Lemma (Lemma 1.1), chooses one of the minimum distances, and returns a single edge based on the choice made.
**Input:** Vertices $x$, $y$, and $v$ of a graph $G$, where $v$ is the vertex to avoid

**Output:** A single edge in the shortest path from $x$ to $y$ avoiding $v$ in graph $G$ and a boolean value 1 or 0. The value 1 denotes that the returned edge is for the forward path and the value 0 denotes that the returned edge is for the backward path.

1. If $d_{x,v} + d_{v,y} > d_{x,y}$ then
2. return the first edge of $\pi_{x,y}$ and the value 1;
3. end

4. $i \leftarrow BCP[x,v]$;
5. $j \leftarrow BCP[v,y]$;
6. if $i > j$ then
7. break; compute an edge of $\hat{\pi}_{y,x,v}$ and the boolean value;
8. end

9. $c_x \leftarrow Cr[x,y,i]$;
10. if $i = j$ then $c_y \leftarrow Cl[x,y,j]$;
11. else $c_y \leftarrow Cr[v,y,i + 1]$;
12. $d \leftarrow \min\{(d_{x,c_x} + D_i[c_y,y,v]), (d_{c_y,y} + \hat{D}_j[c_y,x,v]), DBV[x,y,i]\};$
13. if Term 1 is chosen in the computation of $d$ then
14. return the first edge of $\pi_{x,c_x}$ and the value 1;
15. end
16. else if Term 2 is chosen in the computation of $d$ then
17. return the edge $(P[c_y,y], y)$ and the value 0;
18. end
19. else
20. return $FEBV[x,y,i]$ and the value 1;
21. end

Algorithm 17: Shortest-Path-Avoiding-Vertex
**Input**: Vertices \(x, y,\) and \(v\) of a graph \(G,\) where \(v\) is the vertex to avoid

**Output**: Shortest path \(\pi_{x,y,v}\)

1. \(\text{Front	extunderscore List} \leftarrow x;\)
2. \(\text{Back	extunderscore List} \leftarrow y;\)
3. **while** \(\text{Front	extunderscore List}.last \neq \text{Back	extunderscore List}.last\) **do**
   4. \((u, v, \text{flag}) \leftarrow \text{Shortest	extunderscore Path	extunderscore Avoiding	extunderscore Vertex}(\text{Front	extunderscore List}.last, \text{Back	extunderscore List}.last, v);\)
   5. **if** \(\text{flag} = 1\) **then** insert \(v\) to \(\text{Front	extunderscore List}.last;\)
   6. **else** insert \(u\) to \(\text{Back	extunderscore List}.last;\)
4. **end**
8. Combine \(\text{Front	extunderscore List}\) and \(\text{Back	extunderscore List}\) to get the sequence of vertices in \(\pi_{x,y,v};\)

Algorithm 18: Create	extunderscore Path	extunderscore Avoiding	extunderscore Vertex

**Analysis of Algorithm 17**: Each step calls upon a variable in an already created table that takes \(O(1)\) time to pull the data from the table. Therefore, the total run time is \(O(1).\)

**Analysis of Algorithm 18**: Each call to Algorithm 17 (Shortest	extunderscore Path	extunderscore Avoiding	extunderscore Vertex) in Step 4 takes constant time. Step 8 and the While loop in Step 3 take a total of \(O(L)\) time, where \(L\) is the number of edges in \(\pi_{x,y,v}.\) Therefore, the algorithm runs in \(O(L).\)

### 9.5 Query: Shortest Path Avoiding A Failed Edge

Algorithm 19 (Shortest	extunderscore Path	extunderscore Avoiding	extunderscore Edge) answers queries that ask for the shortest path avoiding a single failed edge. The input consists of vertices \(x\) and \(y\) and an edge \((u, v)\) of a graph \(G,\) and the output is the shortest path from \(x\) to \(y\) avoiding the edge \((u, v).\) This shortest path may either totally avoid the vertex \(u\) or it passes through \(u\) but avoids the edge \((u, v).\) This is determined by comparing the distance returned by Algorithm 15 (Shortest	extunderscore Distance	extunderscore Avoiding	extunderscore Vertex) with \(d_{x,u}\) plus \(De[u,y],\) and storing the minimum of the two values in a variable \(d.\) If \(d\) is assigned the former value, then a call to Algorithm 18 (Create	extunderscore Path	extunderscore Avoiding	extunderscore Vertex) is made and the shortest path is
returned. Otherwise, the shortest path is constructed by concatenating $\pi_{x,u}$, the second endpoint $z$ of $DeE[u,y]$, and $\pi_{z,y}$.

**Input:** Vertices $x$ and $y$ and an edge $(u,v)$ of a graph $G$, where $(u,v)$ is the edge to avoid.

**Output:** Shortest path from $x$ to $y$ avoiding the edge $(u,v)$ in graph $G$

```plaintext
1 if $d_{x,u} + W[u,v] + d_{v,y} > d_{x,y}$ then
   2 return $\pi_{x,y}$;
2 end
3 $d_1 \leftarrow$ Shortest-Distance-Avoiding-Vertex$(x,y,u)$;
4 $d_2 \leftarrow d_{x,u} + De[u,y]$;
5 $d \leftarrow \min\{d_1,d_2\}$;
6 if Term 1 is chosen in the computation of $d$ then
5 return Create-Path-Avoiding-Vertex$(x,y,u)$;
6 end
7 else
8 $(u,z) \leftarrow DeE[u,y]$;
9 return $\pi_{x,u} \circ z \circ \pi_{z,y}$;
10 end
```

**Algorithm 19: Shortest-Path-Avoiding-Edge**

*Analysis of Algorithm 19:* Steps 8 and 12 take a total of $O(L)$ time, where $L$ is the number of edges in the shortest path avoiding the edge $(u,v)$. The computation time of all other steps is $O(1)$. Therefore, the total run time of the algorithm is $O(L)$. 

42
10.1 Experimental Setup

To determine the amount of speed up in running time and reduction in total space is achieved by the nearly optimal oracle proposed in [BK09], the results are compared against the brute-force method, run on the same machine and the same input. We chose to compare the nearly optimal oracle against that of the brute-force method to show the maximum amount of speedup and memory reduction possible from the worst-case running time and memory usage. Comparing the nearly optimal oracle to the next best known oracle might not give a good comparison of how well the nearly optimal oracle performs. As in any experimental setup, there needs to be a control case to gauge how well the new theory or algorithm performs; the brute-force method is our control case.

The brute-force method keeps the following tables: $D$ with a total space needed of $O(n^3)$ that holds the shortest distance between all vertices while avoiding each vertex in turn and the table $Pred$ with a total space needed of $O(n^3)$ that holds the parent of vertex $y$ on the shortest path $\pi_{x,y,v}$ avoiding the failed vertex $v$. To fill in these tables, the brute-force method must run through each pair of vertices in the graph, remove another vertex from the graph, then run Dijkstra’s algorithm, and record the data. Therefor, the total computation time for filling in the tables is $O(n^2 \cdot (m + n \log n))$.

When running the brute-force method, we took the first ten vertices of the input graph for single source shortest path computations and then extrapolated the computation time over all the source vertices and all single vertex failures. This was done because of the large amount of running time the brute-force method would take to completely traverse every path for every failed vertex. In doing so, the time was recorded for the
ten sampled vertices and then estimated for the final run time. If there were, say 250, vertices being tested, then the recorded time was multiplied by 25 (as $10 \times 25 = 250$). Although this does not accurately portray the final running time for the brute-force method, it seems to be a good estimate of the amount of time that could be needed to run the brute-force method in its entirety.

The tests were built to determine, for each data set, the amount of time and the amount of memory needed for construction of the nearly optimal oracle and the oracle given by the brute-force method. The number of queries asked for is not important to the overall running time or total space required; it is only to determine that the nearly optimal oracle is working correctly.

10.1.1 Environment for Synthetic Data Sets

We first created randomly generated input graphs or synthetic data sets to test our code on. We compiled and ran both the nearly optimal oracle and the brute-force method on a Windows machine with an Intel Core 2 Duo Processor, with both CPU cores running at 2.40GHz and 4GB of memory running the Windows Vista Home Premium service pack 2 64-bit operating system. The code was compiled using Visual Studio 2008 with no optimization flags and with all files stored on local hard drive.

10.1.2 Environment for Real-World Data Sets

In addition to the synthetic data sets, we used with permission three real world data sets. The first data set [Kot04] is modeled after that of a local neural network of 131 frontal neurons. Where each neuron is a vertex in the graph and the nerve paths are the edges between the neurons. This data set is composed of 131 vertices and 764 edges. The second data set [CMK04] is also modeled after that of a global neural network of 277 neurons. Once again, each neuron is a vertex in the graph and the nerve paths are the edges between the neurons. This data set is composed of 277 vertices and 2,105 edges. Finally, the third data set [BM06] is modeled after the position and flight paths of the
United States airports and airplanes. Where each airport is a vertex and the flight paths between each airport are the edges. This data set is composed of 322 vertices and 2,126 edges. We then tested our code, both the nearly optimal oracle and the brute force method, on a Linux Machine using an Intel Xeon dual quad core processor in which all eight cores were running at 2661.126MHz and 32GB of memory with the Linux ROCKS 32-bit OS. The code was compiled with gcc version 3.4.6 with no optimization flags and with all files stored on local hard drive.

10.2 Total Memory Used - Synthetic Data Sets

In Figures 10.1 and 10.2, the total memory usage is shown for both the nearly optimal and the brute-force method. In these figures, the x-axis represents the number of vertices for each input graph and the y-axis represents the total memory used in the construction of the oracles. The total memory for both the nearly optimal oracle and the brute-force method are close in terms of how much is used and are separated by as much as 1 MB in some instances, with the nearly optimal oracle using less memory. This may comes as a surprise because the brute-force method has a memory usage of $\Theta(n^3)$, whereas the nearly optimal oracle has a memory usage of $\tilde{O}(n^2)$. The problem arises though during execution, where the brute-force method has only two tables that require $\Theta(n^3)$ space, while the nearly optimal oracle has many tables that require a total of $\tilde{O}(n^2)$ space. Having so many more tables, it is no surprise that the nearly optimal oracle comes close in terms of memory usage to that of the brute-force method, on small data sets.
Figure 10.1: Total memory used for the nearly optimal oracle on synthetic test data

Figure 10.2: Total memory used for the brute-force method on synthetic test data
To get a better idea of how close the nearly optimal oracle and the brute-force method are in terms of memory, Tables 10.1 and 10.2 contain the recorded data values used in the input graphs. An interesting pattern seen in both data tables is that, regardless of the number of edges in the input graphs, the total amount of memory used does not vary much across each row (i.e., fixing the number of vertices, but varying the number of edges for the input graphs does not result in a big change in the total memory spent in the construction of oracles). This shows that it is the number of vertices that dominates how much memory will be used during the execution of the algorithm. This comes as no surprise as both the nearly optimal oracle and the oracle given by the brute-force method have a total space requirement of $\tilde{O}(n^2)$ and $O(n^3)$, respectively, where $n$ is the number of vertices.
Table 10.3: Memory usage of the nearly optimal oracle and the brute-force method on real-world test data (total memory used is measured in kilobytes)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Nearly Optimal Oracle</th>
<th>Brute-force Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kot04]</td>
<td>34,520</td>
<td>19,516</td>
</tr>
<tr>
<td>[CMK04]</td>
<td>147,932</td>
<td>170,848</td>
</tr>
<tr>
<td>[BM06]</td>
<td>121,444</td>
<td>291,416</td>
</tr>
</tbody>
</table>

10.3 Total Memory Used - Real-World Data Sets

Table 10.3 contains the recorded data values for the total memory needed for the nearly optimal oracle and the oracle given by the brute-force method. Except for the first case, the nearly optimal oracle uses the less memory. The first test case is a good example of how the nearly optimal oracle, in small test cases, can be out done by that of the brute-force method. This is attributed to the numerous tables that the nearly optimal oracle must create in order to answer queries in constant time.

10.4 Total Construction Time - Synthetic Data Sets

10.4.1 The Nearly Optimal Oracle and The Brute-Force Method

In Figures 10.3 and 10.4, the total construction time of the two oracles are shown. In these figures, the x-axis represents the number of vertices for each input graph (test case) and the y-axis represents the total construction time for each test case. Here, it is easily seen that the construction time of the nearly optimal oracle for each test case is much less than that of the oracle given by the brute-force method for the same test case. This clearly demonstrates the superiority in terms of the construction time of the nearly optimal oracle over the oracle given by the brute-force method.
Figure 10.3: Total running times for the nearly optimal oracle on synthetic test data

Figure 10.4: Total running times for the brute-force method on synthetic test data
Table 10.4: Total running times used by the brute-force method on synthetic test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>Number of Edges</th>
<th>45</th>
<th>1,225</th>
<th>11,175</th>
<th>19,900</th>
<th>31,125</th>
<th>44,850</th>
<th>61,075</th>
<th>79,800</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>0.452</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>2.582</td>
<td>68.040</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td>49.771</td>
<td>768.042</td>
<td>5,136.270</td>
<td>8957.459</td>
<td>-1</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>124.999</td>
<td>1,461.319</td>
<td>9,237.279</td>
<td>15,753.620</td>
<td>25,504.080</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td></td>
<td>207.775</td>
<td>2,340.949</td>
<td>14,190.900</td>
<td>23,931.150</td>
<td>38,749.025</td>
<td>60,255.050</td>
<td>123,319.830</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td></td>
<td>367.650</td>
<td>3,802.889</td>
<td>21,335.730</td>
<td>41,750.910</td>
<td>60,088.110</td>
<td>85,095.500</td>
<td>140,116.230</td>
<td>143,387.430</td>
</tr>
<tr>
<td>400</td>
<td></td>
<td>850.920</td>
<td>5,032.159</td>
<td>37,724.040</td>
<td>67,030.760</td>
<td>13,229.039</td>
<td>185,758.880</td>
<td>205,832.519</td>
<td>276,464.519</td>
</tr>
<tr>
<td>Number of Vertices</td>
<td>45</td>
<td>1,225</td>
<td>11,175</td>
<td>19,900</td>
<td>31,125</td>
<td>44,850</td>
<td>61,075</td>
<td>79,800</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.783</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1.920</td>
<td>73.925</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>11.231</td>
<td>817.057</td>
<td>4,973.538</td>
<td>8,556.75</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>20.925</td>
<td>2,163.595</td>
<td>8,998.388</td>
<td>12,308.873</td>
<td>19,759.669</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>30.513</td>
<td>4,300.29</td>
<td>11,668.698</td>
<td>20,060.217</td>
<td>25,497.112</td>
<td>27,577.922</td>
<td>38,740.574</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>49.624</td>
<td>3,469.619</td>
<td>21,306.894</td>
<td>19,429.652</td>
<td>27,414.086</td>
<td>37,065.996</td>
<td>27,822.313</td>
<td>21,342.032</td>
<td></td>
</tr>
<tr>
<td>350</td>
<td>64.025</td>
<td>9,506.404</td>
<td>21,123.537</td>
<td>29,358.258</td>
<td>49,698.926</td>
<td>40,214.438</td>
<td>51,564.714</td>
<td>69,372.652</td>
<td></td>
</tr>
</tbody>
</table>
Tables 10.4 and 10.5 show the recorded data for the final construction times of both the oracles given by the brute-force method and the nearly optimal oracle. Looking down a column (increasing vertices) or across a row (increasing edges), it is no surprise that the total construction times, in most cases, increase the way they do. At the core of each algorithm is Dijkstra’s algorithm whose run time is dependent on the number of vertices and the number of edges. There is no way to avoid running Dijkstra’s algorithm, as the shortest paths between vertices must be known in both the oracle constructions.

Looking closely at each table for the total construction time, there are a few instances where the time recorded does not increase as the number of vertices increase, but in fact decreases. This is clearly evident in Table 10.5 for the nearly optimal oracle. This can be attributed to Dijkstra’s algorithm, as it is used in multiple steps in the construction of the oracle. The running time of Dijkstra’s algorithm not only depends on the number of vertices and the number of edges in the graph, but also depends on the arrangement of edges in the graph. Due to the latter dependence, there can be a difference in the amount of time between two graphs of equal size (number of vertices and number of edges) that are structurally differently.

10.4.2 All-Pairs of Shortest Paths

Figure 10.5: Total running times for the algorithm All-Pairs-Shortest-Paths on synthetic test data
In Figure 10.5, the total time taken for Algorithm 4 (All-Pairs-Shortest-Paths) used in the nearly optimal oracle is shown. In this figure, it is not easy to see a pattern with the given data. A few test cases show an increase in the amount of time needed as the number of vertices grow, but in most test cases, the amount of running time can either increase or decrease without any noticeable pattern. This can be explained as this step of the nearly optimal oracle construction relies on using Dijkstra’s algorithm. Where the running time of Dijkstra’s algorithm not only depends on the number of vertices and the number of edges in the graph, but also the way the graph is structured.

Table 10.6: Total running times for the algorithm All-Pairs-Shortest-Paths on synthetic test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>1,225</td>
</tr>
<tr>
<td>10</td>
<td>0.063</td>
</tr>
<tr>
<td>50</td>
<td>0.239</td>
</tr>
<tr>
<td>150</td>
<td>1.709</td>
</tr>
<tr>
<td>200</td>
<td>3.799</td>
</tr>
<tr>
<td>250</td>
<td>4.672</td>
</tr>
<tr>
<td>300</td>
<td>6.593</td>
</tr>
<tr>
<td>350</td>
<td>9.001</td>
</tr>
<tr>
<td>400</td>
<td>11.692</td>
</tr>
<tr>
<td>1,225</td>
<td>-1</td>
</tr>
<tr>
<td>1,175</td>
<td>-1</td>
</tr>
<tr>
<td>19,900</td>
<td>-1</td>
</tr>
<tr>
<td>31,125</td>
<td>-1</td>
</tr>
<tr>
<td>44,850</td>
<td>-1</td>
</tr>
<tr>
<td>61,075</td>
<td>-1</td>
</tr>
<tr>
<td>79,800</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 10.6 shows the recorded data for the running times for the algorithm All-Pairs-Shortest-Paths. Looking closely down a column (increasing vertices) or across a row (increasing edges), there is a general increase in the amount of running time, although there are a few cases where the amount of time taken does decrease. Once again, Dijkstra’s algorithm is the cause of this.

10.4.3 Assigning Priorities

In Figure 10.6, the total time taken for Algorithm 5 (Assign-Priority) in the nearly optimal oracle is shown. This figure is straightforward to understand and, as expected, the total running time is low even with a high number of edges and a high number of vertices.

Table 10.7 holds the recorded running time data for the algorithm. In most test cases of a fixed number of edges, the total running time increases as the number of vertices
increase. This is easy to explain, as each vertex can be visited at most $O(\log n)$ times and, on each visit, a test is ran $O(\log n)$ times in an attempt to set the highest priority. As for the discrepancies where the time decreases as the vertices increase, this can be caused by obtaining a correct priority for each vertex on a smaller number of tries than that of the other test cases.

### 10.4.4 Covering Chains

In Figure 10.7, the total time taken for Algorithms 6 and 7 (Create-Cr-and-BCP-Tables and Create-Cl-Table) in the nearly optimal oracle is shown. Once again, we get a straightforward plot and, as expected, a low total running time of the algorithm.
Figure 10.7: Total running times for the algorithms Create-Cr-and-BCP-Tables and Create-CI-Table on synthetic test data

Table 10.8: Total running times for the algorithms Create-Cr-and-BCP-Tables and Create-CI-Table on synthetic test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>45</th>
<th>1,225</th>
<th>11,175</th>
<th>19,900</th>
<th>31,125</th>
<th>44,850</th>
<th>61,075</th>
<th>79,800</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.014</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>50</td>
<td>0.056</td>
<td>0.190</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>150</td>
<td>0.399</td>
<td>2.260</td>
<td>3.481</td>
<td>3.304</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>200</td>
<td>0.920</td>
<td>3.877</td>
<td>3.137</td>
<td>9.279</td>
<td>9.162</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 10.8 holds the recorded data for the algorithms during each construction of the nearly optimal oracle. The table shows that in most test cases for a fixed number of edges, the running time grows with an increase in the number of vertices used. This can be explained by the following reasoning: as the number of vertices grow, so do the number of shortest paths $\pi_{x,y}$ that have to be traversed to find all center vertices. For the recorded times that decrease when the number of vertices increase, this is caused by the number of edges on the shortest path. The less amount of edges means shorter path trees and a faster return time.
10.4.5 Shortest Distances from Centers

In Figure 10.8, the total time taken for Algorithms 9 and 10 (Create-\(D_k\) and \(D_k\)E-Tables and Create-\(De\) and \(De\)E-Tables) in the nearly optimal oracle is shown. Here, one can see a bottleneck in the amount of time it takes to create the nearly optimal oracle. This is to be expected though as the creation of the \(D_k\) and \(De\) tables requires running Dijkstra’s algorithm many times on sub-graphs created from the main graph. The only way to speed this process up would be to find a faster way than that of the implemented Dijkstra’s algorithm.

Figure 10.8: Total running times for the algorithms Create-\(D_k\) and \(D_k\)E-Tables and Create-\(De\) and \(De\)E-Tables on synthetic test data
Table 10.9: Total running times for the algorithms Create-$D_k$-and-$D_kE$-Tables and Create-De-and-DeE-Tables on synthetic test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>Number of Edges</th>
<th>45</th>
<th>1,225</th>
<th>11,175</th>
<th>19,900</th>
<th>31,125</th>
<th>44,850</th>
<th>61,075</th>
<th>79,800</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>0.637</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>0.882</td>
<td>58.177</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>150</td>
<td></td>
<td>1.259</td>
<td>334.068</td>
<td>3,538.985</td>
<td>7,343.312</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>1.742</td>
<td>688.943</td>
<td>5,226.499</td>
<td>9,278.650</td>
<td>16,167.356</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>250</td>
<td></td>
<td>2.454</td>
<td>869.192</td>
<td>6,984.117</td>
<td>12,267.551</td>
<td>20,077.349</td>
<td>21,569.927</td>
<td>32,394.640</td>
<td>-1</td>
</tr>
<tr>
<td>300</td>
<td></td>
<td>2.696</td>
<td>1,473.274</td>
<td>9,436.752</td>
<td>12,478.880</td>
<td>17,649.640</td>
<td>26,830.760</td>
<td>18,719.736</td>
<td>12,270.429</td>
</tr>
<tr>
<td>350</td>
<td></td>
<td>2.965</td>
<td>2,187.400</td>
<td>10,160.260</td>
<td>19,474.289</td>
<td>35,740.898</td>
<td>28,256.646</td>
<td>40,641.660</td>
<td>58,301.121</td>
</tr>
<tr>
<td>400</td>
<td></td>
<td>3.324</td>
<td>2,449.580</td>
<td>14,148.271</td>
<td>20,546.458</td>
<td>26,945.380</td>
<td>36,275.750</td>
<td>27,513.763</td>
<td>34,200.804</td>
</tr>
</tbody>
</table>
Table 10.9 gives the recorded running times for each of the test cases. Here, one can see a pattern for each test case in which the total running time increases for a fixed number of edges and a varying number of vertices. This can be explained as the algorithm ran here looks to cover all vertices from a center vertex, where the center vertex $c$ with a priority of $k$ can cover vertices up to $5 \cdot 2^k$ levels on its shortest path tree, $T_c$. With the increase in the number of edges, this causes an increase in the number of vertices on each level, which leads to a large graph to pass as input into Dijkstra’s algorithm. As for the few test cases where the total running time decreases as the number of vertices increase, this can be caused by one of two things: The new graphs created are smaller in size than that of the previous test cases and the way in which the new graphs are constructed (as that also effects the amount of running time Dijkstra’s algorithm takes).

### 10.4.6 Bottleneck Tables

![Figure 10.9: Total running times for the algorithms Create-BV-Table and Create-DBV-and-FEBV-Tables on synthetic test data](image)

In Figure 10.9, the total time taken for Algorithms 13 and 14 (Create-BV-Table and Create-DBV-and-FEBV-Tables) in the nearly optimal oracle is shown. As with the total time for constructing the Create-D_k-and-D_kE-Tables and Create-De-and-DeE-Tables algorithms, these algorithms take a large amount of time to create. Once again the cause of this is Dijkstra’s algorithm. In the creation of the bottleneck tables, a new graph
is created that has $O(n^2 \log n)$ number of vertices and $O(mn \log n)$ implicit number of edges. With this new graph created, Dijkstra’s algorithm is then run upon it. As to be expected, as the number of vertices and number of edges increase, the total time taken usually increases.
Table 10.10: Total running times for the algorithms Create-BV-Table and Create-DBV-and-FEBV-Tables on synthetic test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Number of Vertices</th>
<th>45</th>
<th>1,225</th>
<th>11,175</th>
<th>19,900</th>
<th>31,125</th>
<th>44,850</th>
<th>61,075</th>
<th>79,800</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.066</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>50</td>
<td>0.738</td>
<td>12.791</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>150</td>
<td>7.858</td>
<td>468.038</td>
<td>1,372.413</td>
<td>1,120.259</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>200</td>
<td>14.456</td>
<td>1,450.254</td>
<td>3,695.844</td>
<td>2,821.420</td>
<td>3,077.620</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>250</td>
<td>22.298</td>
<td>3,398.638</td>
<td>4,441.590</td>
<td>7,627.544</td>
<td>4,908.655</td>
<td>5,365.124</td>
<td>5,531.957</td>
<td>-1</td>
</tr>
<tr>
<td>300</td>
<td>38.584</td>
<td>1,957.536</td>
<td>11,584.320</td>
<td>6,370.520</td>
<td>9,259.683</td>
<td>9,481.505</td>
<td>8,566.615</td>
<td>8,376.292</td>
</tr>
<tr>
<td>350</td>
<td>49.918</td>
<td>7,259.956</td>
<td>10,528.278</td>
<td>9,263.304</td>
<td>13,278.729</td>
<td>11,013.987</td>
<td>9,660.939</td>
<td>9,471.423</td>
</tr>
<tr>
<td>400</td>
<td>67.320</td>
<td>15,785.940</td>
<td>23,677.271</td>
<td>17,979.763</td>
<td>20,088.679</td>
<td>14,060.567</td>
<td>19,161.740</td>
<td>20,191.171</td>
</tr>
</tbody>
</table>
Table 10.10 holds the recorded data for the total running time of the algorithms. Looking down a column (increasing vertices) or looking across a row (increasing edges) there are fluctuations in the time recorded, but not a steady increase. This is caused once again by Dijkstra’s algorithm, where the structure of the graph effects the total amount of running time taken to find all-pairs of shortest-paths.

10.5 Total Construction Time - Real-World Data Sets

10.5.1 The Nearly Optimal Oracle and The Brute-Force Method

Table 10.11: Construction times of the nearly optimal oracle and the oracle given by the brute-force method on real-world test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Nearly Optimal Oracle</th>
<th>Brute-force Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kot04]</td>
<td>13.033</td>
<td>13.175</td>
</tr>
<tr>
<td>[CMK04]</td>
<td>85.433</td>
<td>144.000</td>
</tr>
<tr>
<td>[BM06]</td>
<td>43.981</td>
<td>105.701</td>
</tr>
</tbody>
</table>

Table 10.11 contains the recorded data values for the total construction time needed for the nearly optimal oracle and the oracle given by the brute-force method. The values here are measured in seconds, and once again the nearly optimal oracle outperforms the oracle given by the brute-force method in every test case.

10.5.2 All-Pairs of Shortest Paths

Table 10.12: Total running times for the algorithm All-Pairs-Shortest-Paths on real-world test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Nearly Optimal Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kot04]</td>
<td>0.283</td>
</tr>
<tr>
<td>[CMK04]</td>
<td>1.624</td>
</tr>
<tr>
<td>[BM06]</td>
<td>105.701</td>
</tr>
</tbody>
</table>

Table 10.12 shows the total time taken for Algorithm 4 (All-Pairs-Shortest-Paths) used in the nearly optimal oracle. These values were obtained by running the nearly optimal oracle on the real-world data sets. As is to be expected from an algorithm that is based on Dijkstra’s algorithm, as the number of vertices and the number of edges increase, so does the running time for this step.
10.5.3 Assigning Priorities

Table 10.13: Total running times for the algorithm Assign-Priority on real-world test data (run time is measured in seconds)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Nearly Optimal Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kot04]</td>
<td>0.180</td>
</tr>
<tr>
<td>[CMK04]</td>
<td>0.800</td>
</tr>
<tr>
<td>[BM06]</td>
<td>0.500</td>
</tr>
</tbody>
</table>

Table 10.13 shows the total time taken for Algorithm 5 (Assign-Priority) in the nearly optimal oracle. Here, the total running time of this step is dependent on the probability of $\Theta(1/2^k)$. If an assignment of priorities is not correct, then they must be reassigned again, this leads to the differences in time shown in the table.

10.5.4 Covering Chains

Table 10.14: Total running times for the algorithms Create-Cr-and-BCP-Tables and Create-Cl-Table on real-world test data (run time is measured in seconds)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Nearly Optimal Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kot04]</td>
<td>0.266</td>
</tr>
<tr>
<td>[CMK04]</td>
<td>1.290</td>
</tr>
<tr>
<td>[BM06]</td>
<td>1.255</td>
</tr>
</tbody>
</table>

Table 10.14 shows the total time taken for Algorithms 6 and 7 (Create-Cr-and-BCP-Tables and Create-Cl-Table) in the nearly optimal oracle. In this table, data set [CMK04] takes the most amount of time. This could be caused by how the vertices were assigned priorities or just the arrangement of edges producing longer shortest paths than the larger data set [BM06].

10.5.5 Shortest Distances from Centers

Table 10.15: Total running times for the algorithms Create-$D_k$-and-$D_k^E$-Tables and Create-De-and-DeE-Tables on real-world test data (run time is measured in seconds)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Nearly Optimal Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kot04]</td>
<td>8.510</td>
</tr>
<tr>
<td>[CMK04]</td>
<td>62.829</td>
</tr>
<tr>
<td>[BM06]</td>
<td>32.206</td>
</tr>
</tbody>
</table>
Table 10.15 shows the total time taken for Algorithms 9 and 10 (Create-D_k-and-D_kE-Tables and Create-De-and-DeE-Tables) in the nearly optimal oracle. Once again, data set [CMK04] takes the most running time, even though it is a smaller data set in the number of vertices and the number of edges. This step requires the center vertices to cover vertices $O(2^k)$ levels deep in its shortest path tree. With the recorded running time, it is safe to assume that data set [CMK04] has more levels than that of the others.

10.5.6 Bottleneck Tables

Table 10.16: Total running times for the algorithms Create-BV-Table and Create-DBV-and-FEBV-Tables on real-world test data (running time is measured in seconds)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Nearly Optimal Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kot04]</td>
<td>4.415</td>
</tr>
<tr>
<td>[CMK04]</td>
<td>21.237</td>
</tr>
<tr>
<td>[BM06]</td>
<td>9.492</td>
</tr>
</tbody>
</table>

Table 10.16 shows the total time taken for Algorithms 13 and 14 (Create-BV-Table and Create-DBV-and-FEBV-Tables) in the nearly optimal oracle. Following the trend so far, data set [CMK04] does indeed take the longest running time. In this step, the bottleneck vertices are found and then avoided. Data set [CMK04], going off the assumption before of having a high number of vertices with a high priority, could have many covering intervals of great length. If only the endpoints, $x$ and $y$, of the shortest path $\pi_{x,y}$, had the greatest priority, then there would only be one covering interval for the path $\pi_{x,y}$, that of $x$ to $y$. This could cause an increase in the amount of running time as each interval must be recursively broken down till only one or two vertices remain.
11.1 Final Thoughts

In conclusion, the plans laid out by the authors of “A Nearly Optimal Oracle for Avoiding Failed Vertices and Edges” [BK09] have been proven to hold true in our experimental study. In every way the nearly optimal oracle is better than that of the brute-force method: a much shorter running time of $\tilde{O}(mn)$ and a lower memory requirement of $\tilde{O}(n^2)$. The algorithm maintains the shortest distance from each vertex to every other vertex of the input graph, and so it requires at least $\Omega(n^2)$ space. This is a great reduction from the brute-force method that would require $\Theta(n^3)$ space. A decrease in the total time needed is possible if there is an algorithm that outperforms Dijkstra’s algorithm in finding single-source shortest paths in any given graph. However, this does not seem likely at this point in time.

In this thesis, we limited our focus on implementation and experimental validation of the distance sensitivity oracle construction by Bernstein and Karger [BK09] and so we restrained experiments on small data sets. As future work, we would like to test our implementation on larger data sets for studying the performance on time and space requirements as a function of graph size (i.e., the number of vertices and the number of edges of the input graph) and to experimentally find the optimal constants used throughout the code. A good example of this is the number of levels a center vertex $c$ will cover in its shortest path tree. In Algorithms 5 and 9, we found that the constant value of 5 multiplied by $2^k$, $k$ being the center priority, was able to cover the smallest amount of vertices and still answer queries. If the number of vertices were to increase, so would that of the number of levels that can be covered. We could then run many
experiments to see if the value of 5 could be decreased, or since there is an increase in the number of vertices, needs to be increased. This would lead to a new study of graph size versus constants that could be explored and used in evaluating performance at runtime of the nearly optimal oracle. Our code is available for further development and testing and can be made available through requests by email or through USF archives. The code has been written to be cross-platform, meaning that the code can be compiled and run on any Win32 or Win64 machine, Mac OSX machine, or Linux machine.
REFERENCES


ABOUT THE AUTHOR

Vincent Williams is currently a graduate student working on his master’s degree in Computer Science in the Department of Computer Science and Engineering at the University of South Florida (USF), Tampa. Vincent is also part of the BS and MS five year program in the Department of Computer Science and Engineering at USF and will be graduating with both his bachelors and masters degree. Vincent began his college career early by dual-enrolling in a community college while still in high school and obtaining his AA before his high school diploma. Vincent is a self motivated person who backs down from no challenge. When given the opportunity to teach English in China for a year, he took the chance. While in China, Vincent never fell behind in his studies, he stayed enrolled at USF and continued his work on his thesis. With his graduation on the horizon, Vincent has begun working part-time at a company called Wynright. Wynright is a software company that writes applications for conveyor belts and robotics that are used inside warehouses to increase the shipping rate of products. Come January 2011, Vincent will pursue a full-time employment with Wynright until June of 2011 when he will join the Air Force and serve his country.