Disjoint paths in geometric graphs

Daniel Mazzella

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/2019

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
DISJOINT PATHS IN GEOMETRIC GRAPHS

by

Daniel Mazzella

Bachelor of Science
University of Nevada, Las Vegas
1998

A thesis submitted in fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2006
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
The Thesis prepared by

DANIEL MAZZELLA

Entitled

DISJOINT PATHS IN GEOMETRIC GRAPHS

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ABSTRACT

Disjoint Paths in Geometric Graphs

by

Daniel Mazzella

Dr. Laxmi P. Gewali, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Construction of the shortest paths connecting two given nodes in a geometric graph is the quintessential problem in computational geometry. We consider several variations of this problem with applications that include robotics, Geographic Information Systems, and sensor networks. The first problem we address is the development of an efficient algorithm for constructing a pair of short node-disjoint paths connecting start and target nodes. The second problem investigated is the development of efficient algorithms for constructing narrow and in-range broadcast corridors in triangulated geometric graphs. Finally, we consider the development of an approximation algorithm for constructing reduced overlap trees in three-colored geometric graphs. Theoretical analysis and a detailed experimental investigation of the proposed algorithms are also presented.
TABLE OF CONTENTS

ABSTRACT .................................................................................................................................................. iii
TABLE OF CONTENTS ................................................................................................................................ iv
LIST OF FIGURES ................................................................................................................................................... vi
ACKNOWLEDGMENTS ........................................................................................................................................ vii
CHAPTER 1 INTRODUCTION .......................................................................................................................... 1
CHAPTER 2 DISJOINT PATHS IN GEOMETRIC GRAPHS .................................................................................. 4
  Preliminaries ................................................................................................................................................ 4
  Shortest Disjoint Path-Pairs ......................................................................................................................... 6
  Narrow Corridors in Planar Graphs .............................................................................................................. 14
  Shortest Width-Bounded Corridor ............................................................................................................... 21
  Time Complexity Analysis .......................................................................................................................... 23
CHAPTER 3 CONSTRUCTING OVERLAP REDUCED TREES ........................................................................... 25
  Introduction ................................................................................................................................................ 25
  Reduced Overlap Trees ................................................................................................................................. 26
  Construction of 2-SPT .................................................................................................................................. 29
  Improved Weight Reassignment Approximation (I-WRAP) ...................................................................... 31
  Time Complexity Analysis .......................................................................................................................... 32
CHAPTER 4 IMPLEMENTATION, EXPERIMENTATION AND PERFORMANCE RESULTS ................................... 33
  Introduction ................................................................................................................................................ 33
  Implementation Framework in JavaBeans .................................................................................................... 34
  edu.unlv.compgeom JavaBeans Framework ................................................................................................. 34
  Dijkstra’s Algorithm for Shortest Path Trees ............................................................................................... 35
  Delaunay Triangulation for Formation of Connected Graph ...................................................................... 38
  Short Length Disjoint Path Pair Algorithm ............................................................................................... 38
  Narrow Corridor Algorithm ......................................................................................................................... 43
  Short In-Range Corridor Algorithm ........................................................................................................... 47
  Weight Reassignment Approximation Algorithm ...................................................................................... 50
  Performance Results .................................................................................................................................... 53
  Tables, Graphs and Figures ........................................................................................................................ 54

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
LIST OF FIGURES

Figure 2.1: Illustrating Disjoint Paths ................................................................. 5
Figure 2.2: The shortest path from s to t creates a cut in the graph .................. 8
Figure 2.3: Path-pair construction for non-distinct cut case ......................... 10
Figure 2.4: Path-pair construction for distinct cut path ............................... 12
Figure 2.5: Short Length Disjoint Path Pair Algorithm ............................... 13
Figure 2.6: Illustrating a Corridor ................................................................. 15
Figure 2.7: Triangulated Graph and its Dual .................................................. 17
Figure 2.8: Progression of the Narrow Corridor Construction Algorithm ....... 18
Figure 2.9: Constrained Dual Algorithm ....................................................... 19
Figure 2.10: Narrow Corridor Algorithm ....................................................... 20
Figure 2.11: Short In-Rang Corridor Algorithm ........................................... 23
Figure 3.1: Illustrating a 3-Colored Network ................................................ 26
Figure 3.2: Illustrations of red and blue SPTs ................................................. 27
Figure 3.3: Illustrating a graph where non-overlapping solution does not exist ... 28
Figure 3.4: Weight Reassignment Approximation Algorithm (WRAP Algorithm) ... 31
Figure 4.1: UML Diagram of SmartPoint, SmartLine, and Smart Segment ....... 36
Figure 4.2: Algorithm Sketch of Dijkstra's Shortest Path Algorithm ............ 37
Figure 4.3: UML Diagram: TwoPathPairHelper and TwoPathPairControl classes 40
Figure 4.4: User Interface of Short Length Disjoint Path Pair Algorithm .......... 41
Figure 4.5: Code for Short Length Disjoint Path Pair Algorithm ................... 42
Figure 4.6: UML Diagram of the NarrowPathHelper and NarrowCorridorControl .... 44
Figure 4.7: User Interface for Narrow Path Application ............................... 45
Figure 4.8: Code for Narrow Corridor Algorithm ......................................... 46
Figure 4.9: UML Diagram of the InRangeHelper and InRangeControl .......... 47
Figure 4.10: User Interface for In Range Application .................................... 48
Figure 4.11: Pseudo Code for Shortest in-Range Algorithm ....................... 49
Figure 4.12: UML Diagram of TwoTreeHelper, TwoTreeControl and BaseControl ...... 50
Figure 4.13: User Interface of TwoTreeControl ............................................ 51
Figure 4.14a: Pseudo Code for the Weight Reassignment Approximation Algorithm .... 52
Figure 4.15b: Pseudo Code for the Weight Reassignment Approximation Algorithm ................................................................. 53
Figure 4.16: Number of nodes vs. path length difference of path pair (%) ......... 54
Figure 4.17: Number of nodes, path pair lengths, and path length difference (%) .. 55
Figure 4.18: Number of nodes vs. percent difference in tree length for WRAP Algorithm ............................................................... 56
Figure 4.19: Number of Nodes, weights of trees 1 and 2, tree 2 weights after algorithm, and difference in percent between Tree 2 before and after Wrap .......... 56

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ACKNOWLEDGMENTS

There are many people whose help, suggestions and knowledge were very valuable in completing this thesis. I would like to extend my greatest gratitude to my advisor, Dr. Laxmi P. Gewali, for his support, patience, and wisdom contributing to my thesis. Without his time and help the completion of this thesis would just not have been possible.

I would like to thank Dr. Ajoy Datta, Dr. Henry Selvaraj, Dr. Tom Nartker, and Dr. Yfantis for their contribution throughout the research and development of the algorithms, code created and writing of this thesis.

I would like to thank my brilliant mother and father for their commitment to me and my education. Lastly, I would like to thank my soon to be wife, Shellie for listening to me when I needed their help, and understanding the long hours I was away from her on this journey.
CHAPTER 1

INTRODUCTION

Finding shortest paths between two nodes in connected networks is perhaps one of the most fundamental problems in networks studies with a wide range of applications that includes robotics, geographic information systems, wide area networks, transportation networks, mobile computing, and ad-hoc sensor networks. In robotics, it is often required to find a collision-free path of short length to move a robot from a starting point to a target point, which can be effectively done by modeling the collection of obstacles by a geometric graph and finding the shortest path in it. Similarly, transportation engineers use shortest path algorithms for planning quickly traversable paths. Shortest path algorithms are used heavily for generating driving directions in road networks. The driving direction programs available in Google map and Mapquest web pages also use shortest path algorithms. In such applications, the shortest path is generally computed by applying Dijkstra’s shortest path algorithm [5] for the weighted and connected graph. While there are obvious applications of a single shortest path connecting two nodes, such a lone path becomes too restrictive in real world applications. Nodes and links of a network could be unreliable and may cease to function. If a planned single path contains a few faulty nodes or links, the entire pre-computed shortest path may become invalid with the failure of a single node/link in the path. For example, in transportation networks the shortest path may contain link
segments prone to stalled-traffic. It is therefore highly desirable to develop algorithms for constructing more than one path of short length connecting two given nodes in a weighted graph.

Computing more than one path between two nodes is becoming very important in recent years due to the emergence of ad-hoc sensor networks. In an ad-hoc sensor network, nodes are low cost, small devices equipped with a low power radio which can be used to establish links. The nodes have low computing power and a relatively small amount of memory. No fixed infrastructure is available to establish connectivity between nodes. The networks connectivity is established when nodes within the transmission range communicate locally. In order to increase the reliability of a routing task in an ad-hoc network, it is critical to develop multiple paths between two given nodes.

In this thesis, several problems dealing with the construction of disjoint paths and disjoint trees connecting given nodes in geometric graphs are considered.

In Chapter 2, three problems dealing with the construction of a pair of node-disjoint paths connecting two given nodes in a geometric graph are addressed. The first problem that is considered is the construction of a pair of short length disjoint paths connecting two given nodes in a planar geometric graph. An $O(n \log n)$ algorithm is presented for solving this problem. One of the main ingredients of the proposed algorithm is the processing of the boundary edges of the graph that helps in making the constructed paths both node-disjoint and of short lengths. The second problem considered in Chapter 2 is the construction of a 'narrow corridor' of smallest width which is essentially the construction of a pair of node disjoint paths that bounds a smallest width corridor connecting two given nodes in a triangulated planar graph. An $O(n \log n)$ algorithm for constructing the narrow-
corridor is presented for the solution. The algorithm is developed by working on the dual of
the given triangulated planar graph. The third problem that is considered in Chapter 2 is a
variation of the narrow corridor problem, called the 'In-Range corridor problem' which
constructs the shortest length corridor of bounded width in a planar triangulated graph. An
O(n log n) algorithm to solve this problem is proposed.

In Chapter 3, the 'overlap reduced' tree construction problem in a geometric graph
is introduced. This problem, the construction of a pair of trees connecting source nodes and
target nodes in a 3-colored geometric graph that have the minimum number of overlap
edges is considered. This seems to be a difficult problem and presented is a heuristic called
the Weight Reassignment Approximation algorithm which constructs a tree pair with
reduced number of overlap edges. The time complexity of the proposed approximation
algorithm is O(n^2 log W).

Chapter 4 contains a description of the proposed algorithms, sketches of the
programs, and the experimental investigation. All of the algorithms are implemented in the
Java programming language. The implementations are written from scratch, with an object
oriented paradigm in mind. The programs read in and write out formatted text files, SVG
formatted XML files, or XFig files. Various pre-existing algorithms from computational
geometry have been used extensively. The description of the classes and their relationships
are depicted in a UML class diagram. The quality of the generated solutions obtained in the
experimental investigation is presented in both tabular and graphical forms.

In Chapter 5, the scope, limitation, and further extensions of the proposed
algorithms are discussed. Also proposed are several interesting open problems dealing with
disjoint paths.
CHAPTER 2

DISJOINT PATHS IN GEOMETRIC GRAPHS

Preliminaries

In this chapter three problems, dealing with the construction of disjoint paths connecting two given nodes in a geometric graph, are considered. First, is the development of efficient algorithms for constructing a node-disjoint-path pair of short length connecting a start node to a target node in a geometric graph. Second, we looked at the construction of a pair of node disjoint paths between two nodes that bound a narrow corridor in a triangulated planar graph. Third is a variation of the second problem which searches for a pair of node disjoint paths that bound the shortest-width corridor in a planar graph. For all three problems, efficient and practical algorithms to solve them are presented.

A pair of disjoint paths can be node-disjoint or edge-disjoint. A pair of edge-disjoint paths can not share an edge but can possibly share nodes. In a node-disjoint path, the paths cannot share nodes. Note that a pair of node-disjoint paths are trivially edge disjoint as well. Since the construction of disjoint paths connecting two given nodes $s$ and $t$ are being investigated, the term node-disjoint is used to indicate that the paths can not share any nodes except the source and target nodes $s$ and $t$. Figure 2.1 illustrates node-disjoint and edge-disjoint paths. In the rest of the chapter the term disjoint-path is meant to be a pair of node disjoint paths.
a. Node Disjoint Paths

b. Edge Disjoint Paths

Figure 2.1: Illustrating Disjoint Paths
Shortest Disjoint Path-Pairs

Dijkstra’s algorithm for finding the shortest path between two given nodes is the most highly cited path construction algorithms in networks. To solve the shortest path problem, Dijkstra’s algorithm constructs the shortest path tree rooted at the source node. To find the shortest path between a pair of nodes, Dijkstra’s algorithm computes the shortest path from the source to all other nodes in the network. [5] In real world applications, it is necessary to have more than one path of shortest length between two given nodes. In road networks, it may be risky to rely on only one short-length path between two given nodes. The shortest path in a road network may not be reliable due to a traffic jam, road construction, or other unpredictable events. In such situations it is necessary to have more than one path, each as independent of the others as possible, for connecting two given nodes. The first step in this direction is the construction of a pair of node-disjoint paths between two given vertices. The precise definition of the short-length disjoint path pair can be stated as follows:

**Shortest Disjoint Path Pair Problem (SDPP)**

**Given:** A geometric graph G(V,E); a start node s; and a target node t.

**Problem:** Construct a pair of disjoint paths P^1 and P^2 connecting s and t such that the total length of P^1 and P^2 is minimized.

Other algorithms for computing the shortest disjoint path pair (SDPP) have been considered previously. Suurballe developed an $O(n^2 \log n)$ algorithm for solving the SDPP Problem in directed weighted graphs with an anti-symmetric property.[11] A directed
weighted graph is said to be anti-symmetric if the presence of an edge \((v, u)\) implies that the edge \((u, v)\) is not present. Suurballe related the problem of computing the shortest path pair to the problem of computing minimum flow in weighted graphs. An improved algorithm for computing shortest path pair was later given by Suurballe and Tarjan in a paper titled “A Quick Method for Finding Shortest Pairs of Disjoint Paths” [10]. Their algorithm runs in \(O(m \log(1 + m/n)n)\) time and \(O(m)\) space where \(m\) is the number of edges in the graph.

The algorithm given in their publication also assumes that the graph is directed and anti-symmetric. They give a technique to transform an undirected weighted graph to a directed and anti-symmetric graph by using vertex splitting. The algorithm presented by Suurballe and Tarjan is theoretically elegant but not simple enough for implementation.

A simpler algorithm, for computing shortest path pair, can be implemented in a straightforward manner without using any complicated data structures and without going through any vertex splitting transformations. This algorithm is based on Dijkstra’s standard algorithm with added preprocessing of the edges on the boundary of the geometric graph.

For presentation purposes, the algorithm is described by considering planar geometric graphs. However, the proposed algorithm works correctly for non-planar geometric graphs as well. A graph is called 2-connected if there exists a pair of node-disjoint paths between any pair of vertices. If the given graph is not 2-connected then a pair of disjoint paths may not exist between \(s\) and \(t\). It is observed that, for most graphs, solutions for the SDPP contains the shortest \(s-t\)-path. However, this need not be true for all graphs, as stated in Observation 1.
Observation 1: The Shortest s-t-path need not be included in the solution of SDPP. The problem instance shown in Figure 2.2 illustrates this observation.

Figure 2.2: The shortest path from $s$ to $t$ creates a cut in the graph

Note that the shortest s-t-path shown in Figure 2.2 transverses partly along the boundary of the graph. The layout of the shortest s-t-path is such that any other path connecting $s$ to $t$ cannot be disjoint from the s-t-path.

Definition 2.1: A path $R^l$ is called a proper sub-path of $R^2$ if $R^l$ is contained in $R^2$ and $R^l$ does not contain the start node $s$ or the target node $t$.

Definition 2.2: A path $Q^l$ is called a distinct cut-path with respect to path $P$ connecting vertices $v^l$ and $v^2$ if it satisfies these two conditions: 1) $Q^l$ is a proper
subset of $P$, and 2) $v^1$ and $v^2$ are not present in $Q'$. A distinct cut path is marked as Cut 1 in Figure 2.2.

**Definition 2.3:** A path $P'$ connecting $s$ and $t$ is a feasible member of the solution for a SDPP instance $\{G, s, t\}$ if a proper sub-path of $P'$ is not a distinct cut path of $G$ with respect to $s$ and $t$.

If the shortest $s$-$t$-path $P_1$ in $G(V,E)$ does not contain a distinct cut-path, then $P_1$ can be taken as one of the members of the shortest path pair. On the other hand if the shortest $s$-$t$-path $P_1$ contains a distinct cut-path then $P_1$ is discarded and other paths need to be looked at to be in the solution for a short path pair.

First, the construction of the shortest path pair is described when the shortest $s$-$t$-path $P_1$ does not contain a distinct cut path (Figure 2.3a). One of the members of the solution of SDPP is $P_1$. To find the other member directions are assigned to the edges on and near $P_1$. The edges on $P$ are given direction implied by its traversal from $s$ to $t$. Direction is then assigned to the edges incident on $P_1$ so that the incident edges get direction away from $P_1$. This process of assigning direction to edges on and near $P_1$ is referred to as path-tagging. Let $G_1$ denote the graph obtained by path-tagging the shortest $s$-$t$ path $P_1$ (Figure 2.3b). Once path tagging of $P_1$ is compete, the other member of the solution is obtained for SDPP by obtaining shortest path $P_2$ from $t$ to $s$ in $G_1$ (see Figure 2.3c).
Figure 2.3: Path-pair construction for non-distinct cut case

a. Shortest $s$-$t$-path

b. Path-tagging $s$-$t$-path

c. Constructing the companion path
Consider the case when the shortest $s$-$t$-path $P_1$ does indeed contain a distinct cut-path (Figure 2.4a). In this case, counterclockwise direction is assigned to the boundary edges of $G$ to obtain $G_1$ (Figure 2.4b). Path tagging is then performed to the portion of the boundary from $t$ to $s$ to obtain the tagged graph $G_2$ (Figure 2.4c). In this tagged graph the shortest $s$-$t$-path $P_2$ is found in $G_2$. It is observed that path $P_2$ is disjoint from the lower boundary path ($t$ to $s$). Now consider path $P_2$ in the original graph $G$. Path tagging is performed on $P_2$ in $G$ to obtain the graph $G_2$. The shortest path $P_3$ from $s$ to $t$ in $G_2$ is obtained. The solution of SDPP is given by paths $P_2$ and $P_3$. A formal sketch of the algorithm is given as Short Length Disjoint Path Pair Algorithm in Figure 2.5.
Figure 2.4: Path-pair construction for distinct cut path
**Input:** Geometric graph $G(V,E)$, start vertex $s$ and target vertex $t$

**Output:** Pair of Short Paths connecting $s$ and $t$

Step 1: Find the shortest path $P_1$ from $s$ to $t$ in $G$

Step 2: If $P_1$ is a cut-path
   a. Assign counterclockwise direction to the boundary edges from $s$ to $t$
   b. Perform Path Tagging on the path from $t$ to $s$ to obtain graph $G_1$
   c. Find the shortest path $P_x$ from $s$ to $t$ in $G_1$
   d. Replace $P_1$ by $P_x$

Step 3: Find the Companion Path $P_2$
   a. Perform path-tagging to $P_1$ in $G$ to obtain $G_2$
   b. Find the shortest path $P_2$ from $t$ to $s$ in $G_2$

Step 4: The output is given by $P_1$ and $P_2$

Figure 2.5: Short Length Disjoint Path Pair Algorithm

The execution time of the Short Length Disjoint Path Algorithm can be determined by assuming that the given geometric graph is available in an embedded form in the plane (e.g. road networks). Step 1 is done by using Dijkstra’s Shortest Path Algorithm which takes $O(n^2)$ time, where $n$ is the number of vertices in the network. Finding a cut-path can be done by checking intersections between the path and the polygon formed by the boundary of the planar embedded network. This can be done in $O(n)$ time, [8] hence, the conditional check in the “if expression” of Step 2 takes $O(n)$ time. Step 2a can be done in $O(n)$ time. Similarly the path-tagging procedure in Step 2b takes $O(n)$ time. Step 2c can be
done in \( O(n^2) \) time using Dijkstra's algorithm. Hence Step 2 takes \( O(n^2) \) time. Similarly Step 3 can be done in \( O(n^2) \) time. Hence, the total time is \( O(n^2) \). This leads to the following theorem.

**Theorem 2.1**: Short Length Disjoint Pair Algorithm can be executed in \( O(n^2) \) time, where \( n \) is the number of vertices in the embedded network.

Remark 2.1: The number of overlapping edges in road networks is very small, often a constant fraction of the number of nodes. Thus, the number of edges in a road networks is linear to the number of vertices. The Dijkstra's Shortest Path Algorithm can be implemented in \( O(|E| \log |E|) = O(n \log n) \) time for such a network. Therefore, Short Length Disjoint Path Pair Algorithm can be executed in \( O(n \log n) \) time for a road network.

**Narrow Corridors in Planar Graphs**

Consider a planar graph \( G(V,E) \) and two given nodes \( s \) and \( t \) in \( G \). A corridor connecting \( s \) and \( t \) in \( G \) is a pair of node-disjoint paths \( P_1 \) and \( P_2 \) such that the region bounded by \( P_1 \) and \( P_2 \) does not contain any other node of the graph. The notion of a corridor between the two given nodes is a planar triangulated graph as illustrated in Figure 2.6.
Exponentially many corridors could be constructed between two given nodes. The concept of narrow corridors is introduced, which are informally corridors with narrow width. For road network, the width of a strip (i.e., corridor) is defined in terms of its triangulation. A triangulated corridor has an ordered sequence of diagonals $d_1, d_2, ..., d_m$. The width of the corridor is the length of the longest diagonal.
Narrow Corridor Problem (NCP)

Given: A triangulated planar graph G(V,E), a start node s and a target node t

Question: Find a corridor of smallest width in G connecting s and t.

In order to develop an algorithm to solve the NCP, we transform the triangulated graph G(V,E) into its dual graph D(V1,E1) without counting the outer unbounded face. For each triangle Ti, we construct a node of vi whose coordinates are taken as the centroid of Ti. Note that if the vertices of a triangle have coordinates (x1, y1), (x2, y2) and (x3, y3) then the coordinates of the centroid (xc, yc) is given by xc = (x1+x2+x3)/3.0 and yc = (y1+y2+y3)/3.0. The centroids of two triangles are connected to form an edge belonging to E1 if the two triangles share an edge. The weight of an edge in E1 is the length of the side shared by the corresponding triangle. It is remarked that the dual graph D(V1,E1) is similar to the Voronoi Diagram [8]. If the triangulated graph is a Delaunay triangulation, Figure 2.7 illustrates a triangulated graph and its dual. The algorithm progression is shown in Figure 2.8. An algorithm for constructing the dual of a triangulated planar graph is listed as the Constrained Dual Algorithm in Figure 2.9.
Figure 2.7: Triangulated Graph and its Dual
Figure 2.8: Progression of the Narrow Corridor Construction Algorithm

- a. Triangulated Graph and its Dual
- b. Showing the Narrowest Tree from $s^d$ to $t^d$
- c. Narrowest tree from $s$ to $t$
- d. The Narrowest Corridor
**Input:** A triangulated planar graph $G(V,E)$

**Output:** Dual Graph $D(V^/,E^/) \text{ of } G$

**Step 1:** Let $v^1_i, v^2_i, v^3_i$ be the vertexes of triangle $t_i$. Let $c_i$ denote triangle $t_i$ Center

**Step 2:** For all triangles $t_i \ (1 \leq j \leq m )$

   a. $c_i \cdot x = (v^1_i \cdot x + v^2_i \cdot x + v^3_i \cdot x) / 3$
   
   b. $c_i \cdot y = (v^1_i \cdot y + v^2_i \cdot y + v^3_i \cdot y) / 3$

**Step 3:** For all triangle $t_i \ (1 \leq i \leq m )$

   a. Let $t_p, t_q, t_r$ be the neighbors of $t_i$
   
   b. Connect $c_i$ to $c_p, c_q$ and $c_r$
   
   c. Weights of the edges $(c_b, c_p), (c_b, c_q),$ and $(c_b, c_r)$ are set to the length of the corresponding shared edges.

---

Figure 2.9: Constrained Dual Algorithm
Input: A triangulated planar graph $G(V,E)$, a start point $s$, a target point $t$

Output: Pair of node disjoint paths connecting $s$ and $t$ that is a narrow corridor

Step 1: Compute the dual graph $D(V',E')$ corresponding to $G(V,E)$ by using constrained dual algorithm

Step 2: // Initialize $Q$ and $P$
   a. Let $Q$ be the empty priority queue
   b. Let $P$ be the empty path
   c. Insert into $Q$ all edges incident on $s$

Step 3: $e = Q.deleteMin()$

Step 4: Insert into $Q$ all edges incident on the outward endpoint of $e$, the endpoints should not already be in $P$

Step 5: While ($e$ does not contain $t$)
   a. $e = Q.deleteMin()$
   b. Insert into $Q$ edges incident on outward ends of $e$ not already in $P$
   c. Add $e$ to $P$

Step 6: Narrow corridor is given the path pairs corresponding to the dual of $P$.

Figure 2.10: Narrow Corridor Algorithm

The time complexity of the algorithm can be determined in straightforward way.

The input triangulated planar graph can be represented in a doubly connected edge list (DCEL) data structure by following standard techniques from computational geometry [8]. This takes $O(n \log n)$ time. In a DCEL representation, the graph can be transversed face by
face in linear time. By using such a data structure, the dual graph can be constructed in \( O(n) \) time. Therefore, we have the following Lemma:

**Lemma 2.1:** Given a triangulated planar graph \( G(V,E) \), its constrained dual can be constructed in \( O(n \log n) \) time.

**Theorem 2.2:** Narrow corridor algorithm can be executed in \( O(n \log n) \) time.

**Proof:** Step 1 takes \( O(n) \) time. Step 2, Step 3 and Step 4 all take constant time. The *delete-min* and *insert-item* operations in a priority queue can be done in \( O(\log n) \) time by using the standard priority queue data structure. Since each edge is inserted into the priority queue at most once, the total time for Step 5 is bounded by \( O(n \log n) \). This leads to \( O(n \log n) \) total time for executing the entire algorithm.

**Shortest Width-Bounded Corridor**

The shortest width-bounded corridor algorithm can be used to find 2 disjoint paths that are the closest to each other. Practical application of this algorithm can be helpful in situations where two co-travelers along these paths, wish to maximize contact (i.e. visibility by radio) with each other. The narrowest corridor between two given nodes could be exceedingly wide in some node networks (Figure 2.11).
We could consider constructing a corridor which is not necessarily the narrowest, but whose width is within a certain range $\beta$. The range $\beta$ may correspond to the maximum visibility range or the maximum communication range.

*Short In-Range Corridor Problem (SICP)*

**Given:** A triangulated planar geometric graph $G(V,E)$ a start node $s$ and a target node $t$.

**Question:** Find a shortest corridor connecting $s$ and $t$ such that the width of the corridor is within a given range $\beta$.

At first glance, it seems that a variation of the tree growing algorithm can be used with the narrow corridor algorithm. With this approach, each step in the tree construction reveals more that one edge than can be used to grow the tree, subject to the range constraint. It can not be determined which one will give the shortest length. Therefore a different approach is needed. The edges from the dual graph that correspond to those out-of-range will need to be discarded. The dual graph obtained by discarding out-of-range edges is referred to as a *reduced dual graph*. Then, Dijkstra’s shortest-path algorithm can be used on the reduced dual graph to obtain a solution for *SICP*. A formal sketch is given as follows:
**Short In-Range Corridor Algorithm**

**Input:** A triangulated graph $G(V,E)$; A start point $s$; A target point $t$; A broadcast range $\beta$

**Output:** Pair of node disjoint paths representing a short in-range corridor connecting $s$ to $t$.

Step 1: Compute the dual graph $D(V',E')$ corresponding to $G(V,E)$ using the constrained dual algorithm.

Step 2: Remove all edges from the dual graph $D(V',E')$ who’s weight is greater than $\beta$

Step 3: Run Dijkstra’s shortest path algorithm on $D(V',E')$ to produce path $P$.

Step 4: The Short In-Range Corridor is given by the path pairs corresponding to the dual of $P$.

---

**Figure 2.11: Short In-Rang Corridor Algorithm**

**Time Complexity Analysis**

Much like the narrow-corridor problem in Section 2.3, the input triangulated planar graph can be represented in a doubly connected edge list (DCEL). Unlike the narrow-corridor problem, there is no tree to transverse; instead, Dijkstra’s Shortest Path Algorithm is applied. Dijkstra’s Algorithm normally takes $O(n^2)$ time. However, since Dijkstra’s algorithm is run on the dual graph, which is constructed from a Delaunay triangulation, the number of edges $E$ is linear to the number of vertices $n$. In this case, Dijkstra’s Shortest Path Algorithm can be implemented in $O(|E| \log |E|) = O(n \log n)$ time. This leads to the following theorem:
Theorem 2.3: Short In-Range Corridor algorithm can be executed in $O(n \log n)$ time.

Proof: Step 1 takes $O(n)$ time as shown in section 2.3. Step 2 takes $O(n)$ time, since each edge of the dual is examined to determine if it is greater or less than the broadcast range $\beta$. Dijkstra’s Shortest path Algorithm run in Step 3 takes $O(n \log n)$ time. This leads to $O(n \log n)$ total time for executing the entire algorithm.
CHAPTER 3

CONSTRUCTING OVERLAP REDUCED TREES

Introduction
Consider road networks in a metropolitan area. Such networks can be approximated by a weighted graph $G(V,E)$ consisting of a set of vertices $V$ and a set of edges $E$. Assume that the number of vertices is $n = |V|$ and the number of edges is $m = |E|$. The vertices are then marked in three different colors: $h$ of them are red, $q$ of them are blue, and the remaining are white (non colored). One of the red vertices is distinguished as a source vertex and is denoted by $s^1$. The remaining $h-1$ red vertices are called target vertices and are denoted as $t^1_r, t^2_r, \ldots, t^{h-1}_r$. Similarly, the blue vertices consist of one source vertex $s^2$, and $q-1$ target vertices $t^1_b, t^2_b, \ldots, t^{q-1}_b$. An example graph with red, blue and white vertices is shown in Figure 3.1.
The union of the shortest paths connecting the red source vertices to red target vertices gives a red shortest path tree (red-SPT) rooted at s1 (Figure 3.2a). The blue shortest path tree (blue-SPT) can be defined similarly.

Reduced Overlap Trees

With respect to the overlay of the red-SPT and blue SPT shown in Figure 3.2b, a pair of SPTs can have overlap edges as evident from the given example. Our problem of interest is to construct a pair of shortest path trees (blue-SPT and red-SPT) that have a reduced number of overlap edges.

Reduced Overlap 2-SPT Problem (2-SPT)

Given: i. A weighted 3-color graph $G(V,E)$ with h red vertices, q blue vertices, and $|V| - h - q$ white vertices

The red vertices consist of one source vertex $s'_0$ and $h - 1$ target vertices $t_1, t_2, ..., t_{h-1}$. 

26
The blue vertices consist of one source vertex $s^b_0$ and $q - 1$ target vertices $t^b_1, t^b_2, ..., t^b_{h-1}$.

**Question**: Construct a red-SPT and a blue-SPT that have the minimum number of overlap edges.

Figure 3.2: Illustrations of red and blue SPTs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
It can be remarked that a non-overlapping solution for the 2-SPT problem may not exist for graphs with certain input source and target instances. The example in Figure 3.3 shows such an instance.

Many graph covering problems are known to be NP-Hard [4] and it is very likely that 2-SPT problem is also intractable. The 2-SPT problem is like the problem of covering a graph with simpler graphs. It is interesting to hypothesize approximation algorithms for the 2-SPT problem. One approach would be to generate the shortest-path trees for red source/targets and blue source/targets separately, which can be determined by adopting Dijkstra's Algorithm. The separately generated red-SPT and blue-SPT could then be carefully modified to reduce the number of overlapping edges without significantly increasing the weights of the SPTs.
Construction of 2-SPT

Since the shortest path now contains the shortest path information from all vertices in G(V,E) to a given start, generating a SPT can be done by following the edges from the selected targets back to the source. In the red/blue SPT problem, Dijkstra’s algorithm must be run twice, once with the red-source and once with the blue source. When the red-SPT and blue-SPT are generated, the overlap edges can be clearly identified by examining each edge of the paths. If the two SPTs have no overlapping edges, then that is the optimum solution.

One approach is to try to avoid the overlapping edges. When the red-SPT and blue-SPT have been generated, the edges of the tree and the edge of the graph can be compared to identify overlapping edges. If the two SPTs do not have overlapping edges, then the optimum solution has been obtained. If overlapping edges are present then the trees need to be modified in an attempt to reduce the number of overlapping edges. An infinitely large weight is assigned to overlapping edges with the assumption that it is better to avoid overlapping edges altogether from the intended solution. Dijkstra’s algorithm is applied again on the modified graph obtained by assigning infinite weight to the overlapping edges.

Preliminary inspection in a few simplistic graphs revealed that this approach would not generate a good solution most of the time. It is not appropriate for several reasons. First, the true shortest path is not used by either blue or red SPTs. Second, this approach simply forced the blue and red trees to make a detour. In the detour, the two SPTs may again have more overlapping edges.

A better approach is to allow overlapping edges if the detour paths are not shorter than the initial overlapping edges. A question then arises, which of the two SPTs should
use the selected overlapping edges? The total weight of the SPT is considered the sum of the weights of its edges. In order to use the maximum number of short length edges, it is asserted that the tree having the largest total weight will use all the overlapping edges and the other tree will look for detour edges. Assume without the loss of generality, that the red-SPT has larger weight and is kept as it is, while the blue-SPT's edges are to be modified to reduce the number of overlap edges. The main question is how to construct the modified blue-SPT. A weight re-assignment approach is proposed, in which the weights of the overlap edges are increased by a factor of $f$ times the original weight to generate a modified graph $G_l$. The SPT Algorithm is run again in $G_l$ to generate the modified blue-SPT. Since the overlap edges now have larger weight, the SPT algorithm will try to avoid them when constructing the tree. The construction of the modified blue-SPT can be repeated by using larger and larger values of $f$ if necessary. A formal sketch of this approximation scheme is listed below as the Weight Reassignment Approximation Algorithm (WRAP Algorithm). Three conditions are used to sketch the algorithm.

Definition 3.1 (Condition 1): Red-SPT and Blue-SPT have overlapping edges

Definition 3.2 (Condition 2): Let $WT$ be the total weight of all edges of the original graph $G(V,E)$. No edge of the recently updated graph has weight $\geq WT$.

Definition 3.3 (Condition 3): Incremental improvement $> \epsilon$ (Percentage of overlapping edges is reduced every time)
**Input:** A 2-colored connected, graph $G(V,E)$ source/target nodes set identified

**Output:** Overlap Reduced red-SPT and blue-SP

Step 1: Generate red-SPT $T_1$ and blue SPT $T_2$ by running Dijkstra’s algorithm on $G(V,E)$.

Step 2: Assume without loss of generality that $T_1$ has larger number of edges.

Step 3: while( condition 1 and condition 2 and condition 3)
   
   a: Let $G_l$ be the graph obtained by doubling weight of the overlapping edges.
   
   b: Generate red-SPT $T_1$ and blue-SPT $T_2$ by applying Dijkstra’s algorithm to $G$.
   
   c: Identify overlapping edges in $T_1$ and $T_2$

Step 4: Output $T_1$ and $T_2$ as the Overlap-Reduced trees

---

**Figure 3.4: Weight Reassignment Approximation Algorithm (WRAP Algorithm)**

**Improved Weight Reassignment Approximation (I-WRAP)**

Double the multiply factor $a$ after each iteration and take smaller starting value, for example $a = 0.1$. By focusing on and modifying the smaller of the two SPTs, it is anticipated that the overall quality of the solution will be more optimal. In practice, adding weight to edges is preferred vs. removing the edges so the graph can remain connected.

There may be bridges in the graph that all paths must use. This approach will allow the smaller of the SPTs to use those edges, while avoiding the other edges with the higher weights.
Time Complexity Analysis

The execution time of the WRAP algorithm can be done in a clear-cut way. The analysis assumes that the input graph is available in an embedded format in the plane. This algorithm uses Dijkstra's Algorithm, whose execution takes $O(n^2)$ time in the worst case, so Step 1 takes $O(n^2)$ time. Identification of overlapping edges of the two trees can be done $O(n)$ time, hence Step 2 takes $O(n)$ time. The loop in Step 3 executes at most in $\log(W)$ time where $W$ is the total weight of all the edges in the graph. Each execution of Step 3 takes $O(n^2)$ time. Hence, Step 3 takes $O(n^2 \log W)$ time. So, the total execution time of the algorithm is $O(n^2 \log W)$.

**Theorem 3.1** Execution of the WRAP Algorithm can be done in $O(n^2 \log W)$ where $n$ is the number of vertices in the networks and $W$ is the total weight of all the edges in the network.

**Remark:** The actual number of times Step 3 has been observed to execute is $O(1)$ times. As the number of nodes decreases, the more times step 3 has to be repeated. Conversely, as the number of nodes increases, the number of executions of Step 3 stabilized at one. That is to say, a weight of twice the existing weight of the edges was sufficient to make the two trees disjoint. This means, in practice, the actual execution time of the algorithm is $O(n \log n)$ for planar graph.
CHAPTER 4

IMPLEMENTATION, EXPERIMENTATION AND PERFORMANCE RESULTS

Introduction

In this chapter, the implementation, experimentation and performance results are explored. The algorithms proposed in Chapters 2 and 3 have been implemented in the Java Programming Language [1]. Java has become the de facto programming language used in corporate enterprise systems, and is fast becoming the programming language of choice among scientists. Java is an Object Oriented Programming language, whose paradigm enables real world entities to be represented using classes. Built into java is an enormously powerful API called the Swing classes that allows for quick development of graphical user interfaces. The object oriented nature of these swing classes enables implementations computational geometry problems to have a good platform to build upon.

Experiments with the developed algorithms were conducted on randomly generating graphs ranging in size from 25 through 500 nodes. For each of the algorithms (Short Length Path Pairs, Narrow Corridor, and Overlap Reduced Trees) sixty graphs were generated and processed to produce the results given in this text. These various metrics were then applied to the results to gauge the performance of the algorithms on finding hypothesized solutions.
Implementation Framework in JavaBeans

Object oriented programming techniques center around the reuse of code. Many times, frameworks will be constructed that allow problems of a similar nature to share base classes over and over again without re-implementing them. In Java, these frameworks are known as JavaBeans. “JavaBeans technology is the component architecture for the Java 2 Platform, Standard Edition (J2SE). Components (JavaBeans) are reusable software programs that you can develop and assemble easily to create sophisticated applications.” [9]

The use of JavaBeans proved to be very handy in computational geometric programming. The idea of encapsulating points (vertices), segments (Edges), rays, lines, and graphs with the data and the relationships on one another freed up the development process to concentrate on solving the larger problems, and not worry on how to implement the calculation of the distance from a point to a line, for example. Coupled with Java language tools like TreeSets, Comparators, ArrayLists, and Iterators, many iterative and pointer operations were abstracted away from the actual implementation.

edu.unlv.compgeom JavaBeans Framework

The basis of all the software developed in this work is done with the edu.unlv.compgeom package. In Java, basic objects are represented in this package as JavaBeans. The “getter”, “setter”, and calculation methods are the only publicly visible interfaces. The internal data structures can not be manipulated by other outside classes. The interfaces provide all the functionality needed by the higher level classes. The edu.unlv.compgeom package contains the root beans from which everything else is
developed. Some of the classes that constitute the package are SmartPoint, which encaptulates the data and methods relating to a point or vertex, SmartLine, which contains the data and methods relating to a line, and SmartSegment, which is a line with a start point and end point, and an implementation for Dijkstra's Shortest Path Algorithm. In the Dijkstra class, a geometric graph $G(V,E)$ is simulated with an array of SmartPoints, and an array of SmartSegments. A UML model [3] showing the data contained within the classes, the methods implemented, and the relationship between them is illustrated in Figure 4.1. With these base classes contained within a JavaBean, the algorithms presented are easily implemented without worrying about the exact details of how data is stored, how to compare two points, or how to find the distance between a point and a line.

Dijkstra's Algorithm for Shortest Path Trees

The foundation of many of the algorithms presented here is Dijkstra's Shortest Path Algorithm. The algorithm can be described informally in terms of a set of nodes of a partial tree $S$, set of fringe vertices $R$, and a set of bridge edges $BeE$. Initially, the source vertex $v_0$ is included in $S$. Vertices not in $S$ but adjacent to vertices in $S$ are called fringe vertices. The edges connecting fringe vertices to vertices in $S$ are referred to as bridge edges. The algorithm progresses by selecting a light bridge-edge that corresponds to the fringe vertex that has the shortest distance from source vertex. The light bridge-edge is included in the partial tree and distances from the source vertex to fringe vertices are updated. This process of light-edge selection and shortest distance update to fringe vertices is repeated in steps to grow the initial single node tree to the shortest path tree. The algorithm is formally sketched below in Figure 4.2.
<table>
<thead>
<tr>
<th>SmartLine</th>
</tr>
</thead>
<tbody>
<tr>
<td>-vertical : boolean</td>
</tr>
<tr>
<td>-horizontal : boolean</td>
</tr>
<tr>
<td>-slope : double</td>
</tr>
<tr>
<td>-intercept : double</td>
</tr>
<tr>
<td>+SmartLine() : SmartLine</td>
</tr>
<tr>
<td>+areOnSameSide(SmartPoint sp1, SmartPoint sp2) : boolean</td>
</tr>
<tr>
<td>+clone() : SmartLine</td>
</tr>
<tr>
<td>+computeIntersect(SmartLine sl) : SmartPoint</td>
</tr>
<tr>
<td>+draw(Graphics g, Color c)() : void</td>
</tr>
<tr>
<td>+getSlope()() : double</td>
</tr>
<tr>
<td>+getXPoint(double y)() : double</td>
</tr>
<tr>
<td>+getYPoint(double x)() : double</td>
</tr>
<tr>
<td>+getIntercept()() : double</td>
</tr>
<tr>
<td>+isAbove(SmartPoint sp)() : boolean</td>
</tr>
<tr>
<td>+isHorizontal()() : boolean</td>
</tr>
<tr>
<td>+isParallel(SmartLine l)() : boolean</td>
</tr>
<tr>
<td>+linesOn(SmartPoint)() : boolean</td>
</tr>
<tr>
<td>+toSegment()() : SmartSegment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SmartPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>-java.awt.point : object(idl)</td>
</tr>
<tr>
<td>-index : int</td>
</tr>
<tr>
<td>+SmartPoint() : SmartPoint</td>
</tr>
<tr>
<td>+getX()() : int</td>
</tr>
<tr>
<td>+getY()() : int</td>
</tr>
<tr>
<td>+getIndex()() : int</td>
</tr>
<tr>
<td>+setIndex(int i)() : void</td>
</tr>
<tr>
<td>+compareTo(Object o1, Object o2)() : int</td>
</tr>
<tr>
<td>+compareToTo(Object o)() : int</td>
</tr>
<tr>
<td>+equals(Object o)() : boolean</td>
</tr>
<tr>
<td>+distance(Object)() : double</td>
</tr>
<tr>
<td>+draw(Graphics g, Color c, size s)() : void</td>
</tr>
<tr>
<td>+clonePoint()() : SmartPoint</td>
</tr>
<tr>
<td>+collinear(SmartPoint sp2)() : boolean</td>
</tr>
<tr>
<td>+left(SmartPoint sp1)() : boolean</td>
</tr>
<tr>
<td>+intersectProps(SmartPoint sp1, SmartPoint sp2)() : boolean</td>
</tr>
<tr>
<td>+between(Object)() : boolean</td>
</tr>
<tr>
<td>+intersect(SmartPoint sp1, SmartPoint sp2)() : boolean</td>
</tr>
<tr>
<td>+toSVG(String fill, String stroke)() : String</td>
</tr>
<tr>
<td>+toFig(int type)() : String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>-vertices : Array of SmartPoint</td>
</tr>
<tr>
<td>-edges : Array of SmartSegment</td>
</tr>
<tr>
<td>-costMatrix : int[][]</td>
</tr>
<tr>
<td>+setVertices(TreeSet vertices)() : void</td>
</tr>
<tr>
<td>+setEdges(TreeSet edges)() : void</td>
</tr>
<tr>
<td>+setStartVertex(SmartPoint sp)() : void</td>
</tr>
<tr>
<td>+getVertices()() : Array of SmartPoint</td>
</tr>
<tr>
<td>+getEdges()() : Array of SmartSegment</td>
</tr>
<tr>
<td>+getShortestPath(SmartPoint sp)() : ArraySmartSegment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Delaunay</th>
</tr>
</thead>
<tbody>
<tr>
<td>-vertices : SmartPoint</td>
</tr>
<tr>
<td>-edges : SmartSegment</td>
</tr>
<tr>
<td>+getVertices() : SmartPoint</td>
</tr>
<tr>
<td>+getEdges() : SmartSegment</td>
</tr>
<tr>
<td>+triangulate() : SmartSegment</td>
</tr>
<tr>
<td>+dual() : SmartSegment</td>
</tr>
</tbody>
</table>

Figure 4.1: UML Diagram of SmartPoint, SmartLine, and Smart Segment

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Dijkstra's Shortest Path Algorithm

Input: i. A graph $G(V,E)$ with a set of vertices $V$ of size $n$ and set of edges $E$. Each edge $e_i = (u, v)$ has a weight $w_i$ associated with it that represents the Euclidean distance between the vertices represented in a 2x2 cost matrix ii. A source Vertex $v_0$.

Output: An array representing the indexes of the shortest path from all vertices to the source vertex $v_0$

Step 1: for $i = 0$ to the number of vertices $n$
  found[$i$] = false;
  distance[$i$] = cost[$v_0$][$i$];
Step 2: set found[$v_0$] = true; set distance[$v_0$] = 0; set path[$v_0$] = -1;
Step 3: for $i = 1$ to the number of vertices $n$
  $u = \text{choose(distance, n, found)}$;
  found[$u$] = true;
  for ($w = 0$ to the number of vertices)
    if (!found[$w$] && distance[$u$] + cost[$u$][$w$] < distance[$w$])
      distance[$w$] = distance[$u$] + cost[$u$][$w$];
      path[$w$] = $u$;
Step 4: Return the path array

Sub Routine Choose

Given: An array of distances, a boolean array found and a size $n$

Output: The index of the edge with the smallest distance for the list

Step 1: for $i = 0$ to the size $n$; min = distance[$i$]; minpos = $i$;
Step 2: return minpos

Figure 4.2: Algorithm Sketch of Dijkstra's Shortest Path Algorithm
Delaunay Triangulation for Formation of Connected Graph

To test most of the algorithms, a set of random nodes was generated. These random nodes must all be connected to make sure a path from any one node exists to any other node in the graph. Delaunay triangulation can be used to create a connected graph from a set of vertices. In order to understand the Delaunay Triangulation, it is important to understand its dual construct, the Voronoi Diagram. The Voronoi diagram records many properties about the proximity of a graph. For example, for a given node, which nodes are the closest, and which are the furthest? The Voronoi Diagram has the property that, for each site (clicked with the mouse), every point in the region around that site is closer to that site than to any of the other sites. Delaunay Triangulation can be defined as a triangulation of the sites with the additional property that for each triangle of the triangulation, the circumcircle of that triangle is empty of all other sites. The development of the code used in the JavaBean constructed here can be traced to a book by Joseph O'Rourke called “Computational Geometry in C.”

Short Length Disjoint Path Pair Algorithm

With Java, the Short Length Disjoint Path Pair Algorithm was implemented by using a helper class, TwoPathPairHelper, that contains the Dijkstra class that represents the graph $G(V,E)$ and two arrays that represent the paths $P1$ and $P2$. The methods in the TwoPathPairHelper facilitate the addition, deletion, and manipulation of the vertices, and edges as well as, setting of the start point and target point. The operations carried out on $G$, $P1$ and $P2$, to generate the results, were implemented with minimal lines of code. The main application, TwoPathPairControl, handles the user input and display in a Java-swing based
graphical user interface. The TwoPathPairControl class extends a BaseControl class that has many of the methods that are used over and over again, such as mouse control, and the drawing of graphs and paths. The UML Interface diagram is shown in Figure 4.3.

When a user starts the TwoPathPairControl application, they are greeted with a large blank canvas, a menu that contains standard menu file, open, and save options along the top. Along the left side, buttons trigger operations to add vertices and edges, delete vertices and edges, generate a random graph, set the start and target nodes, and to run the disjoint shortest path algorithm. This user interface is based on current software programs to make it easy to understand and work with.

Execution of the Short Length Disjoint Path Pair Algorithm is conducted when the following conditions are met:

1. There are more than three vertices on the canvas
2. There are more than three edges on the canvas
3. The start and target vertices have been identified
4. The "Calculate Disjoint Path" button has been pressed.
Once those conditions are met, the TwoPathPairControl will call

TwoPathPairHelper.calculatePaths() method. The pseudo-code of the execution is shown in Figure 4.5.
Figure 4.4: User Interface of Short Length Disjoint Path Application
void calculatePath()
{
    // Step1: Add to P1 the shortest path from S->T
    this.path1.addAll(dijkstra.shortestPath(this.vertices, this.edges, this.start,
    this.target));

    ArrayList e1;
    // Step 2: if P1 is a cut path
    if (cutPath(this.path1)) {
        path1.clear();
        // find the clockwise boundary
        this.findClockwiseBoundary();
        // the next two lines are path tagging
        e1 = trimPath(e, this.boundary);
        e1 = trimPath(e1, findTtoSBoundaryPath());
        // replace P1
        this.path1.addAll(dijkstra.shortestPath(this.vertices, e1, this.start,
        this.target));
    }
    e1.clear();
    // Step 3: the next two lines are path tagging
    e1 = trimPath(e, pathBlue);
    e1 = trimPath(e1, this.getDirectedIncidentEdgesToPath(pathBlue));
    // Find p2.
    this.path2.addAll(this.shortestPath(this.vertices, e1, this.start, this.target));
}

Figure 4.5: Code for Short Length Disjoint Path Pair Algorithm
Narrow Corridor Algorithm

The implementation of the Narrow Corridor Algorithm is very similar to the implementation of the Short Length Disjoint Path Pair Algorithm. A Graphical User Interface (GUI) Control class called NarrowCorridorControl is in charge of the user action, saving, opening, and display of the graph. NarrowCorridorHelper is the class that implements the algorithm. The key difference in this Narrow Corridor implementation is the requirement for the graph to be a Delaunay Triangulation. An additional JavaBean called Delaunay is used for constructing the edges of the graph. Also used is the notion of a PriorityQueue. The queue will execute the compare(Object o) method inside the objects passed into it to determine which order the objects within the queue are stored. The deleteMin() method will return the object with the smallest value of the compare() method, and remove it from its queue. Pseudo Code for the Narrow Corridor Algorithm is show in Figure 4.8.
Figure 4.6: UML Diagram of the NarrowPathHelper and NarrowCorridorControl
Figure 4.7: User Interface for Narrow Path Application
void calculatePath(){
    // Step 1: Find the Delaunay Triangulation
    this.edges.addAll(this.delaunay.triangulate(this.vertices)) ;
    // Find the dual of the delaunay Triangulation, vertices, edges, s&t
    ArrayList dualEdges(this.delaunay.dualEdges()) ;
    TreeSet dualVertices(this.delaunay.dualVertices());
    SmartPoint dualStart = dualVertices().findClosestEdge(this.start) ;
    SmartPoint dualTarget= dualVertices().findClosestEdge(this.target) ;
    // Step 2: PriorityQueue is a data structure that does not allow
    // duplicates, and arranges items in its Queue according to
    // weight of SmartSegment() ;
    PriorityQueue pQueue = new PriorityQueue(<SmartSegment>);  
    // Step 3: Add the incident edges from start to the queue
    pQueue.addAll(dualEdges.getIncidentTo(dualStart)) ;
    // Step 4: while the target hasn't been reached, loop...
    while( ! pQueue.contains(dualTarget)) {
        // Delete the minimum from the queue
        SmartSegment e = pQueue.deleteMin() ;
        // add all edges incident to the target point to the queue
        pQueue.addAll(dualEdges.getIncidentTo(e.target())) ;
        // add the SmartSegment to the Dual Path
        this.dualPath.add(e) ; }
    // Step 5: find the paths P1 and P2 that are on either side of
    // the dualPath in the dual graph
    findPathsFromDual(dualEdges, dualPath, this.edges, this.path1, this.path2) ;
}

Figure 4.8: Code for Narrow Corridor Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Short In-Range Corridor Algorithm

Much like the Narrow Corridor Problem, the Short In-Range Algorithm requires the graph to be in a Delaunay Triangulation before work can be done. Some changes are made to the GUI from the Narrow Corridor Algorithm, mainly the addition of an input field that takes in the value for the broadcast range and integer value. The ShortInRangeControl class extends the same Base Control class. The method that implements the algorithm is contained in ShortInRangeHelper and is called computePaths(). Pseudo code for the Short In-Range algorithm is shown in Figure 4.10.

![Figure 4.9: UML Diagram of the InRangeHelper and InRangeControl](image)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 4.10: User interface for In Range Application
void calculatePath()
{
    // Step 1: Find the Delaunay Triangulation
    this.edges.addAll(this.delaunay.triangulate(this.vertices));
    // Find the dual of the delaunay Triangulation, vertices, edges, s&t
    ArrayList dualEdges(this.delaunay.dualEdges());
    TreeSet dualVertices(this.delaunay.dualVertices());
    SmartPoint dualStart = dualVertices().findClosestEdge(this.start); 
    SmartPoint dualTarget= dualVertices().findClosestEdge(this.target); 
    // Step 2: Remove edges with weight less than range
    for(i = 0 ; i < dualEdges.size() ; i++){
        if(dualEdges.get(i).getWeight() < this.range){
            dualEdges.delete(i) ; i--  ;
        }
    }
    // Step 3: Find the shortest path in the dual
    dualpath.addAll(dijkstra.shortestPath(dualVertices, dualEdges,
    dualStart, dualTarget));
    // Step 4: find the paths P1 and P2 that are on either side of dualPath in the dG
    findPathsFromDual(dualEdges, dualPath, this.edges, this.path1, this.path2); 
}

Figure 4.11: Pseudo Code for Shortest in-Range Algorithm
Weight Reassignment Approximation Algorithm

The implementation of the Weight Reassignment Approximation Algorithm followed a similar design as the previous algorithms. Some changes were made to handle the multiple sources and target vertices. There are two SmartPoints representing the two source vertices; two ArrayLists of SmartPoints represent the target vertices. Figure 4.11 shows the UML diagram for the TwoTreeHelper, TwoTreeControl, and BaseControl class relationships. The user interface follows the design of the BaseControl class, as show in Figure 4.12. The implementation of WRAP algorithm follows the algorithm sketch fairly closely, as shown in Figure 4.13.

Figure 4.12: UML Diagram of TwoTreeHelper, TwoTreeControl and BaseControl
Figure 4.13: User Interface of TwoTreeControl
```java
void calculatePath()
{

// Step 1: Generate tree 1 and two by using Dijkstra's SP

tree1.addAll(dijkstra.shortestPathTree(vertices, edges, start1, targets1));
tree2.addAll(dijkstra.shortestPathTree(vertices, edges, start2, targets2));

ArrayList edges2 = this.edges.clone();
int lastOverlapCount = overlap(tree1, tree2);

int a = 2;
while(overlapCount(tree1, tree2) && this.edges.getTotalWeight() >= edges2.getTotalWeight() &&
      lastOverlapCount <= overlapCount(tree1, tree2)) {
    ArrayList overlapEdges = overlapEdges(tree1, tree2);
    overlapEdges.addWeight(a++);
    edges.replaceAll(overlapEdges);
    tree1.addAll(dijkstra.shortestPathTree(vertices, edges, start1, targets1));
    tree2.addAll(dijkstra.shortestPathTree(vertices, edges, start1, targets1));
}
}

// returns the count of the edges that overlap
int overlapCount(ArrayList tree1, ArrayList tree2){
    int count = 0;
    for(i = 0; i < tree1.length; i++){
        if(tree2.contains(tree1.get(i))) count++;
    }
    return count;
}
```

Figure 4.14a: Pseudo Code for the Weight Reassignment Approximation Algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Performance Results

To evaluate the performance of the proposed algorithms, many randomly generated graphs were constructed; the algorithms were executed against the random set to obtain the results. In addition, some hand crafted graphs were generated, so the boundary cases could be evaluated. The quality of the solutions differs for the different algorithms. The main characteristics of the experimental investigation for each of the algorithms can be summarized as follows:

Short Disjoint Path Pair Problem: Compare the percentage difference of the true shortest path distance traveled over the two disjoint path pairs.

Narrow Corridor Problem: Compare the maximum width of the corridor.

In-Broadcast Range Problem: Compare the percentage difference of the true shortest path in the broadcast range of the distance traveled over the two disjoint path pair.

Two-SPT Problem: Compare the percentage difference of the length of the true shortest trees against the lengths of the two weight reassigned trees.
Tables, Graphs and Figures

Figures 4.14-4.17 show the results obtained by running randomly generated graphs through each of the algorithms, and the results of the performance metrics listed. Figure 4.14 shows the stabilization of the Disjoint Path Pair Algorithm as the number of nodes increases. Figure 4.15 is a sample of the number of nodes from 10 through 492, the length of the two paths, and the percent difference. Figure 4.16 shows the stabilization of the WRAP algorithm as the number of nodes increases. Figure 4.17 shows a sample of number of nodes vs the weights of the trees before and after the WRAP Algorithm, and finally the percent difference.

Figure 4.16: Number of nodes vs. path length difference of path pair (%)
<table>
<thead>
<tr>
<th>No. Nodes</th>
<th>Path 1</th>
<th>Path 2</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>428</td>
<td>296</td>
<td>30.84%</td>
</tr>
<tr>
<td>52</td>
<td>593</td>
<td>534</td>
<td>9.95%</td>
</tr>
<tr>
<td>104</td>
<td>495</td>
<td>402</td>
<td>18.79%</td>
</tr>
<tr>
<td>132</td>
<td>557</td>
<td>475</td>
<td>14.72%</td>
</tr>
<tr>
<td>162</td>
<td>551</td>
<td>467</td>
<td>15.25%</td>
</tr>
<tr>
<td>214</td>
<td>634</td>
<td>575</td>
<td>9.31%</td>
</tr>
<tr>
<td>260</td>
<td>611</td>
<td>583</td>
<td>4.58%</td>
</tr>
<tr>
<td>306</td>
<td>583</td>
<td>539</td>
<td>7.55%</td>
</tr>
<tr>
<td>356</td>
<td>408</td>
<td>388</td>
<td>4.90%</td>
</tr>
<tr>
<td>442</td>
<td>546</td>
<td>501</td>
<td>8.24%</td>
</tr>
<tr>
<td>492</td>
<td>377</td>
<td>370</td>
<td>1.86%</td>
</tr>
</tbody>
</table>

Figure 4.17: Number of nodes, path pair lengths, and path length difference (%)
Figure 4.18: Number of nodes vs. percent difference in tree length for WRAP Algorithm

<table>
<thead>
<tr>
<th>No. Nodes</th>
<th>Tree 1 weight</th>
<th>Tree 2 weight</th>
<th>Tree 2 weight after WRAP</th>
<th>Difference (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>796</td>
<td>562</td>
<td>562</td>
<td>30</td>
</tr>
<tr>
<td>77</td>
<td>804</td>
<td>199</td>
<td>199</td>
<td>30</td>
</tr>
<tr>
<td>89</td>
<td>1182</td>
<td>894</td>
<td>894</td>
<td>25</td>
</tr>
<tr>
<td>158</td>
<td>1337</td>
<td>3164</td>
<td>3164</td>
<td>13</td>
</tr>
<tr>
<td>474</td>
<td>3228</td>
<td>1494</td>
<td>1494</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 4.19: Number of Nodes, weights of trees 1 and 2, tree 2 weights after algorithm, and difference in percent between Tree 2 before and after Wrap.
CHAPTER 5

CONCLUSIONS AND FUTURE EXTENSIONS

Algorithms for finding short length multiple disjoint paths have many applications in modern day life. We proposed four algorithms dealing with the construction of disjoint paths and overlap reduced trees in a geometric network: (i) Two-Pair Shortest Path Algorithm, (ii) Narrowest In-Range Corridor Algorithm, (iii) Short In-Range Corridor Algorithm, and (iv) Weight Reassignment Approximation (WRAP) Algorithm.

Experimental techniques used to implement the proposed algorithms were critically examined. The actual implementation was done in the Java Programming language, by means of reusable JavaBeans [9] to reduce the total time needed for coding. Randomly distributed vertices were generated, and for some algorithms, Delaunay triangulation was used to generate a connected graph. Each of the algorithm's implementations extended a base control class, that contained the tasks and methods that were common throughout all the implementations. The implementation was properly documented in UML format.

To examine the performance of the algorithms, a large set of randomly generated graphs was used. In particular, for the experimental investigation of Disjoint Pair and WRAP algorithms, about 1500 randomly generated graphs were given as the input. For the other two algorithms, 500 randomly generated and structurally constructed networks were taken as the input. The results were then plotted, and as expected, the graphs with the higher node density yielded smaller difference between the first and next shortest path for...
the Short Disjoint Path Pair problem. Similar results were observed with the WRAP algorithm. Most noticeably sparse graphs had a larger difference in the weights of the two trees, whereas dense graphs had a lower difference.

It seems that the Disjoint Path Pair Algorithm proposed in Chapter 2 produces the shortest disjoint pairs. However, we have not been able to prove this assertion formally. The second algorithm proposed in Chapter 2 produces the optimum solution for the narrowest corridor problem. It would be interesting to look for the development of a similar algorithm in three dimensions where triangles correspond to tetrahedrons. Similar extensions could be carried for the Shortest In-Range problem in three dimensions. The Overlap Reduced Tree problem in Chapter 3 seems very difficult. It would be interesting to either prove it to be NP-Hard or to come up with a polynomial time algorithm.
REFERENCES


VITA

Graduate College
University of Nevada, Las Vegas

Daniel A Mazzella

Local Address:
209 Opera House St
Henderson, NV 89012

Degree:
Bachelors of Science, Computer Science, 1998
University of Nevada, Las Vegas

Thesis Title:
Disjoint Paths in Geometric Graphs

Thesis Examination Committee:
Chair: Dr. Laxmi P. Gewali, Ph. D.
Committee Member: Dr. Ajoy Datta, Ph. D.
Committee Member: Dr. Tom Nartker, Ph. D.
Graduate Faculty Representative: Dr. Henry Selvaraj, Ph. D.