Angle constrained paths in sensor networks

Xiaojun Liang
University of Nevada, Las Vegas

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

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1649

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.
ANGLE CONSTRAINED PATHS IN SENSOR NETWORKS

by

Xiaojun Liang

Master of Science
University of Nevada, Las Vegas
2001

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 2004
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.
Thesis Approval
The Graduate College
University of Nevada, Las Vegas

MAY 18, 2004

The Thesis prepared by
LIANG, XIAOJUN

Entitled
ANGLE CONSTRAINED PATHS IN SENSOR NETWORKS

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

Angle Constrained Paths in Sensor Networks

by

Xiaojun Liang

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

Short-length paths in geometric graphs are not necessarily feasible in sensor networks and robotics. Paths with sharp-turn angles cannot be used by robotic vehicles and tend to consume more energy in sensor networks.

In this thesis, we investigate the development of short-length paths without sharp-turn angles. We present a critical review of existing algorithms for generating angle constrained paths. We then consider the construction of routes having directional properties – d-monotone routes which are special cases of angle constrained paths. We develop a centralized algorithm for computing shortest d-monotone paths in triangulated networks. Since local computations are highly desired in sensor networks, we also consider localized online algorithms for computing length-reduced d-monotone paths in Delaunay Triangulation networks.

The proposed algorithms are implemented in the Java programming language. Performances of the proposed algorithms are evaluated by examining the routes constructed by them on several randomly-generated Delaunay networks.
TABLE OF CONTENTS

ABSTRACT ......................................................................................................................... iii

TABLE OF CONTENTS ........................................................................................................ iv

LIST OF TABLES ............................................................................................................... vi

LIST OF FIGURES ........................................................................................................... vii

ACKNOWLEDGMENTS ..................................................................................................... viii

CHAPTER 1  INTRODUCTION ...................................................................................... 1
Chapter Overview .............................................................................................................. 4

CHAPTER 2 ANGLE CONSTRAINED PATHS .......................................................... 6
Angle Constrained Shortest Path Problem .................................................................... 7
Faster Implementation of Angle Constrained Shortest Path Algorithm ..................... 9
Other Curvature Constrained Paths ............................................................................. 16

CHAPTER 3 MONOTONE ROUTES IN SENSOR NETWORKS ........................... 17
Greedy Routing Algorithms ......................................................................................... 18
Monotone Routes in Triangulated Networks .............................................................. 23
Centralized Algorithms for $d$-Monotone Routes .................................................. 26
Monotone Routes in Delaunay Networks .................................................................... 28
Localized Algorithms for $d$-Monotone Routes ..................................................... 31

CHAPTER 4 IMPLEMENTATION AND EXPERIMENTAL RESULTS ............. 34
Overview of Doubly Connected Edge List Data Structure ........................................ 35
Interactive Construction of Planar Straight Line Graph ...................................... 40
Construction of Planar Straight Line Graph from A Data File ............................ 45
Routes Construction ...................................................................................................... 51
A Centralized Algorithm for Standard Shortest Paths ............................................. 52
A Centralized Algorithm for Angle Constrained Shortest Paths .......................... 54
A Centralized Algorithm for Shortest $d$-Monotone Routes .................................. 56
Localized Algorithms for $d$-Monotone Routes ..................................................... 57
Experimental Results ..................................................................................................... 63

CHAPTER 5 CONCLUSION AND EXTENSIONS .................................................. 68
LIST OF TABLES

Table 4-1  Vertex Records in the vertexVector for Figure 4-2 .........................................39
Table 4-2  Half-edge Records in the halfEdgeVector for Figure 4-2 .............................39
Table 4-3  Face Records in the faceVector for Figure 4-2 .............................................40
Table 4-4  Format of the Input Data File for A PSLG (Format A) ...............................45
Table 4-5  Input Data File for the PSLG in Figure 4-2 ...............................................46
Table 4-6  Average Path Length Comparison ...............................................................64
Table 4-7  Average Path Hops Comparison .................................................................65
Table 4-8  Performance Comparison of MCR and HR by Path Length ......................65
Table 4-9  Performance Comparison of MCR and HR by Path Hops .........................65
LIST OF FIGURES

Figure 2-1 Illustrating Turn Angles .................................................................6
Figure 2-2 Transformation of $G$ into $G'$ ($\theta = 90^\circ$) .....................................9
Figure 2-3 An Angle Constrained Shortest Path Tree .........................................15
Figure 3-1 Illustrating a $d$-Monotone Route .....................................................17
Figure 3-2 Most Forward Routes ......................................................................19
Figure 3-3 Compass Routes .............................................................................21
Figure 3-4 CR Performs Better Than LSR .......................................................22
Figure 3-5 LSR Performs Better Than CR .......................................................23
Figure 3-6 Illustrating Sleeve and non-Sleeve ..................................................24
Figure 3-7 Illustrating the Left-Unbounded Region and Right-Unbounded Region.25
Figure 3-8 Illustrating the Construction of Directed Graph ...............................27
Figure 3-9 Proof for Lemma 3.2 .....................................................................29
Figure 3-10 A Voronoi-DT Path .................................................................30
Figure 3-11 Modified Compass Routing .........................................................32
Figure 4-1 Illustrating a PSLG .....................................................................37
Figure 4-2 Illustrating the DCEL representation of a PSLG ...........................38
Figure 4-3 Illustrating Add-edge ......................................................................41
Figure 4-4 Illustrating Split-edge .....................................................................41
Figure 4-5 Illustrating Move-vertex ..................................................................42
Figure 4-6 Illustrating Split-face ......................................................................43
Figure 4-7 Illustrating Delete-vertex – no face deletion ....................................44
Figure 4-8 Illustrating Delete-vertex – delete one face .....................................44
Figure 4-9 A Shortest Path Constructed by Dijkstra’s Algorithm in a DT .........53
Figure 4-10 A Shortest Path Tree Constructed by Dijkstra’s Algorithm ...........54
in a DT of 1000 Vertices ...........................................................................54
Figure 4-11 A Shortest Path with Angle Constraint of $90^\circ$ in a DT ..............55
Figure 4-12 A Shortest $d$-Monotone Path in a DT .........................................57
Figure 4-13 A $d$-monotone Route Constructed by MCR algorithm in a DT ....59
Figure 4-14 A $d$-monotone Route Constructed by HR algorithm in a DT .......61
Figure 4-15 A $d$-Monotone Route Constructed by MCR in a DT of 1000 Vertices...62
Figure 4-16 A $d$-Monotone Route Constructed by HR in a DT of 1000 Vertices ....63
Figure 4-17 Comparison of Path Lengths by SP, MCR and HR .......................66
Figure 4-18 Comparison of Path Hops by SP, MCR and HR ...........................67
Figure 5-1 A Shortest Path with a Back-up Edge in a DT ...............................69

vii

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

I would like to take this opportunity to express my sincere gratitude to the following people for their invaluable contribution to my research:

My committee chair, Dr. Laxmi Gewali, who initially inspired my interests in computational geometry and sensor network fields, and who guided me through the entire research process of this thesis. He helped me learn and grow, and encouraged me continuously.

My committee members, Dr. John Minor, Dr. Wolfgang Bein, and Dr. Emma Regentova helped me greatly in improving this thesis.

My husband, Tao, offers full support to me all the time.

I would like to extend my thanks to all the professors and staff in the School of Computer Science.
CHAPTER 1

INTRODUCTION

Mobile Computing and Sensor Networks are the two emerging high-tech innovations that have a significant impact on accessing, processing, and delivery of information. A sensor network consists of sensor nodes with short-range radio and on-board processing capability. Any small equipment with embedded wireless sensing and computing devices can be considered as a sensor node. Commercially available Bluetooth devices have been increasingly used to enhance computing and/or sensing equipment with wireless communication capability. A sensor node could be used to sense many physical phenomena that include light, temperature, vibration, or magnetic field in its neighborhood.

There is no fixed network infrastructure in a sensor network. Connectivity among sensor nodes is formed by establishing radio links. Only nodes within the transmission range can establish links. Nodes outside the range can establish connectivity by using other intermediate nodes as relay nodes. The sensor nodes can be either in active or in inactive state. To make the ad-hoc network functional, only active nodes should be linked to form the network. In sensor networks, nodes are usually assumed to be stationary. In some wireless network models, nodes are allowed to change position. Such networks are usually known as Mobile Ad-Hoc Networks (MANET).
One potential application of sensor networks is to process some high-level sensing tasks in a collaborative fashion. The network is periodically queried by an external source to report a summary of sensed data. This area of sensor networks is having a major impact on the computing industry witnessed by a wide range of applications in defense, e-commerce, manufacturing, robotics, and data gathering. Sensor networks have great potential to extend our ability to monitor and control the physical environment from remote locations. Sensor networks improve information gathering accuracy by providing distributed processing of vast quantities of information. Such networks have great scope for processing seismic data, acoustic data, and high-resolution digital images. This approach can provide a rich multi-dimensional view of the environment. Sensor networks are very useful in military and other tactical applications. A large number of sensors can be scattered in risk-prone areas for surveillance purposes to detect certain objects of interest. A typical query could be to report the number of sightings at certain intervals. Sensor networks are very effective in identifying critical events that include the detection of intruders or rogue activities in sensitive regions. Such techniques have immediate application in Homeland Security.

Commercial applications of sensor networks are in situations where there is a need for ubiquitous communication services such as business area wireless networking, on-the-fly conferencing, and communication between mobile robots. Sensor networks are thus expected to play a major role in sensor aided manufacturing. Driven by Moore’s law, sensor nodes are getting smaller and cheaper. As a result, in the near future, it will be economically feasible to use sensor networks even if the applications warrant the deployment of a large number of sensor nodes.
Researchers in computer science are finding opportunities and challenging problems in the area of sensor networks. Most of the traditional centralized algorithms cannot be applied directly to sensor networks. The network connectivity changes frequently and nodes can appear or disappear in unpredictable ways. The challenging problem is how to pool limited local resources at the nodes to obtain global properties.

The routing problem is one of the major fields in sensor network research. Given a source node and a target node, we want to deliver an information packet, or move a vehicle/robot from the source node to the target node. If the target node is within the transmission range of the source node, then the packet can be delivered directly. Otherwise, we have to route the packet from the source node to the target node through one or more relay nodes. A short-length route with a small number of relay nodes is highly desired.

Since the entire network connectivity is not available at each node, a feasible approach would be to develop localized routing algorithms which only require limited resources for the nodes. Normally, a sensor node knows position (coordinates) of itself and positions of its 1-hop neighbors. Availability of positions of additional nodes can lead to the development of good quality localized algorithms for solving specialized problems. Consider the routing problem in which a source node $S$ needs to deliver a packet to a target node $T$ which is outside of its transmission range. This problem can be solved more effectively if the source node knows the position of the target node. Routing algorithms developed on the assumption that the source node knows the position of the target node (in addition to the position of itself and the positions of its 1-hop neighbors)
are usually called location based routing algorithms. Some authors use the term geometric routing to indicate location based routing.

One of the widely-used routing techniques is based on the greedy paradigm. In a greedy routing scheme, the current node delivers the packet to its neighbor that optimizes a certain objective function and also makes positive progress toward the target node. Examples of greedy routing include most forward routing, compass routing, and their variations. Greedy routing algorithms are simple to implement and produce good results for certain classes of networks. For some networks, greedy routing may fail to generate the route.

Routing algorithms based on planar networks are known to construct routes successfully. Face routing is a technique that is guaranteed to construct routes in any connected network. Face routing extracts a planar sub-graph $G$ of a given network and processes it face by face for route construction. As an example, a planar graph called Constrained Gabriel Graph can be constructed locally and its faces can be processed incrementally to generate a source/target route [15]. Several variations of face routing algorithms based on Constrained Gabriel Graphs have been reported [3, 4, 6, 16].

1.1 Thesis Overview

In this thesis, we consider the location based routing problem with turn angle constraint. Although angle constrained routing is an important problem, only a few research papers have been reported. In Chapter 2, we critically review an existing centralized algorithm for angle constrained shortest path planning.
In Chapter 3, we report several results dealing with the generation of $d$-monotone routes. $d$-monotone routing can be viewed as a special case of angle constrained routing. We characterize classes of triangulated networks which do not admit any $d$-monotone route between a given pair of nodes. We propose two localized algorithms for constructing $d$-monotone routes in Delaunay networks. We also present an efficient linear time centralized algorithm for constructing shortest $d$-monotone routes in any triangulated network.

In Chapter 4, we describe the implementation and experimental results of various centralized and localized routing algorithms that include (i) angle constrained shortest path algorithm, (ii) shortest $d$-monotone path algorithm, (iii) modified compass route algorithm (MCR), and (iv) hybrid lateral/compass route algorithm (HR). We also report performances of the proposed algorithms on randomly generated Delaunay networks.

Finally, in Chapter 5, we discuss extensions of the proposed algorithms and discuss some interesting problems dealing with route generation in Delaunay networks.
CHAPTER 2

ANGLE CONSTRAINED PATHS

In this chapter, we present a critical review of angle constrained shortest path algorithms reported in the literature.

The problem of computing shortest paths constrained to have turn angles no more than a certain value has applications in many areas including robotics, motion planning, highway design, and sensor networks. Often, the network is modeled by straight line edges, and a path extracted from such a network consists of a sequence of straight line segments. Two consecutive segments of a path define a turning angle as shown in Figure 2-1.

Figure 2-1: Illustrating Turn Angles
In Figure 2-1, the turn angle $\theta_2$ is much sharper than $\theta_1$. A path with sharp turn angles is not feasible for practical applications. An automobile cannot be pulled through sharp angle turns. In sensor networks, a route with sharp turns tends to consume more energy. It is remarked here that for automobile roads, the paths modeled by sequences of line segments need to be smoothed into curves. Then the turn angles correspond to turn radii. A sharp turn angle becomes a curve with a small turning radius.

2.1: Angle Constrained Shortest Path Problem

The Angle constrained shortest path problem (ACSP) is defined as:

**Given**: A graph $G(V, E)$, a source node $S$, a target node $T$, and a threshold angle $\theta$.

**Question**: Find a shortest path connecting $S$ to $T$ (if it exists) such that the turn angles in the path are no more than the threshold angle $\theta$.

Dijkstra's algorithm is usually used for computing shortest paths in a weighted graph $G(V, E)$. The path produced by Dijkstra's algorithm may contain sharp turn angles. It is known that the Dijkstra's algorithm runs in $O(|V|^2)$ time if a simple array is used for the queue data structure. It can be made to run in $O((|V| + |E|) \log |V|)$ time if the queue is represented by a binary heap [17]. If the queue data structure is implemented by using Fibonacci heaps, then the algorithm runs in time $O(|V| \log |V| + |E|)$ [17]. A fibonacci based implementation runs very fast for large-size sparse graphs.

Dijkstra's algorithm cannot be applied directly to compute the shortest path with angle constraints. A non-trivial transformation is necessary for capturing angle constraints.
constraints. Boroujerdi and Uhlmann [1] have suggested the following transformation to obtain the transformed Graph $G'(V', E')$ from the original graph $G(V, E)$.

1. First, transform edges $e_{ij} \in E$ into vertices $v'_{ij} \in V'$.

2. Second, transform each consecutive pair of edges $e_{ij}, e_{jk} \in E$ into an edge $e'_{ijk} \in E'$, $e'_{ijk}$ is from $v'_{ij}$ to $v'_{jk}$ in $G'$.

It is clear from the definition of transformed graph $G'(V', E')$ that the number of vertices $|V'|$ in the transformed graph is equal to the number of edges $|E|$ in the original graph. The number of edges $|E'|$ in the transformed graph can be estimated as follows. A vertex in $G$ can generate up to $O(|E|)$ edges in $G'$, and there can be up to $O(|V|)$ such vertices. This implies that $|E'| = |E|*|V|$.

An example of the transformation from $G$ into $G'$ under turn angle constraint $\theta = 90^\circ$ is illustrated in Figure 2-2. There are ten edges in $G$. Therefore, there are ten corresponding vertices in $G'$. If a pair of consecutive edges in $G$ meets the angle constraint, the corresponding edge in $G'$ is shown as a solid line, otherwise it is shown as a dash line.
So, if we apply Dijkstra’s algorithm to the entire transformed graph $G'$, a path that minimizes the total turn penalty is generated. If we apply Dijkstra’s algorithm only to the solid edges in $G'$, a shortest path that meets the angle constraint $\Theta$ is generated. The time complexity of applying Dijkstra’s algorithm to the transformed graph is

$$O(|V| \log |V| + |E'|) = O(|E| \log |E| + |E'| \cdot |V|) = O(|E| \log |V| + |E| \cdot |V|) = O(|E| \cdot |V|).$$

### 2.2 Faster Implementation of Angle Constrained Shortest Path Algorithm

To solve the problem of generating shortest paths that do not contain turn angles more than threshold angle $\Theta$, a faster implementation has been developed [1]. The faster version is essentially a clever implementation of the transformed graph $G'$ without explicitly generating it. Range search data structure from computational geometry is
used for the adjacency list (list of neighboring vertices). Two more data structures are used for this implementation: a priority queue $Q$ and a shortest path tree $T$. A vertex can occur in the priority queue more than one time. It can also occur more than one time in the shortest path tree (It is remarked that a vertex can occur only once in the tree/queue in Dijkstra's standard shortest path algorithm.). Sketch of the angle constrained shortest path algorithm is as follows:

**Angle Constrained Shortest Path Algorithm**

1. Construct the adjacency list (range search data structure) for each vertex $v_m$. // a vertex $v_n$ is adjacent to $v_m$ if there is an edge from $v_m$ to $v_n$ in $G$.
2. A priority queue $Q$ is initiated as null. A shortest path tree $T$ is initiated as null.
3. Insert $v_s$ into $Q$ with priority 0. Insert $v_s$ into $T$.
4. While $Q$ is not empty do:
   4.1) Retrieve and delete the vertex $v_j$ with minimum priority $p$ from $Q$;
   4.2) If $v_j$ is $v_t$, then stop and report the path is obtained (by following the parent pointers from $v_j$ to $v_s$ in $T$) with $p$ as the cost of the path.
   4.3) For each vertex $v_k$ in the adjacency list of $v_j$ such that the turn $v_i v_j v_k$ does not violate the turn constraint do: // $v_i$ is parent of $v_j$ in $T$
      4.3.1) Insert $v_k$ into $Q$ with priority $p + c_{jk}$ // $c_{jk}$ is the cost from $v_j$ to $v_k$ in $G$
      4.3.2) Insert $v_k$ into $T$ and set its parent pointer to $v_j$
      4.3.3) Delete $v_k$ from the adjacency list of $v_j$

In order to understand this clever implementation, we illustrate it with a running example for the graph $G$ in Figure 2-2. We want to find a shortest path from $v_0$ to $v_5$ in $G$ under the angle constraint $\theta$ of $90^\circ$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Step 1: Initialize the Adjacency List:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₀</td>
<td>v₁ v₂ v₃ v₄</td>
</tr>
<tr>
<td>v₁</td>
<td>v₂ v₄</td>
</tr>
<tr>
<td>v₂</td>
<td>v₃</td>
</tr>
<tr>
<td>v₃</td>
<td>v₄</td>
</tr>
<tr>
<td>v₄</td>
<td>v₂ v₅</td>
</tr>
<tr>
<td>v₅</td>
<td></td>
</tr>
</tbody>
</table>

Step 2: Insert v₀ into Q and T.

Q:

| v₀ | 0 |

T:

v₀

Step 3.1: Delete v₀ from Q, insert neighbors of v₀ into Q and T, and delete neighbors of v₀ from Adjacency List.

Q:

| v₂ | 183 |
| v₁ | 221 |
| v₃ | 293 |

T:

v₀ ——— v₁ ——— v₂ ——— v₃
Adjacency List:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td></td>
</tr>
<tr>
<td>$v_1$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>$v_2$</td>
</tr>
<tr>
<td>$v_5$</td>
<td></td>
</tr>
</tbody>
</table>

Step 3.2: Delete $v_2$ from $Q$, insert neighbors of $v_2$ that meet angle constraint – only $v_3$ here into $Q$ and $T$, and delete $v_3$ from Adjacency List of $v_2$.

$Q$:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td></td>
</tr>
<tr>
<td>$v_1$</td>
<td>221</td>
</tr>
<tr>
<td>$v_3$</td>
<td>293</td>
</tr>
<tr>
<td>$v_5$</td>
<td>329</td>
</tr>
</tbody>
</table>

$T$:

![Diagram]

Adjacency List:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td></td>
</tr>
<tr>
<td>$v_1$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$v_2$</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>$v_2$</td>
</tr>
<tr>
<td>$v_5$</td>
<td></td>
</tr>
</tbody>
</table>

Step 3.3: Delete $v_1$ from $Q$, insert $v_4$ into $Q$ and $T$ (although $v_2$ is neighbor of $v_1$, turn of $v_0v_1v_2$ does not meet the constraint, so $v_2$ is not inserted in $Q$ and $T$), and delete $v_4$ from Adjacency List of $v_1$.
Adjacency List:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_0</td>
<td></td>
</tr>
<tr>
<td>v_1</td>
<td>v_2</td>
</tr>
<tr>
<td>v_2</td>
<td></td>
</tr>
<tr>
<td>v_3</td>
<td>v_4</td>
</tr>
<tr>
<td>v_4</td>
<td>v_3, v_5</td>
</tr>
<tr>
<td>v_5</td>
<td></td>
</tr>
</tbody>
</table>

Step 3.4: Delete v_3 from Q, insert v_4 into Q and T, and delete v_4 from Adjacency List of v_3.
Adjacency List:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td></td>
</tr>
<tr>
<td>$v_1$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$v_2$</td>
<td></td>
</tr>
<tr>
<td>$v_3$</td>
<td></td>
</tr>
<tr>
<td>$v_4$</td>
<td>$v_3$ $v_5$</td>
</tr>
<tr>
<td>$v_5$</td>
<td></td>
</tr>
</tbody>
</table>

Step 3.5: Since Adjacency List of $v_3$ is empty, just delete $v_3$ from $Q$.

Step 3.6: Although $v_2$ and $v_5$ are neighbor of $v_4$, turns of $v_3v_4v_2$ and $v_3v_4v_5$ do not meet the constraint, so $v_2$ and $v_5$ are not inserted in $Q$ and $T$. Just delete $v_4$ from $Q$.

Step 3.7: Delete $v_4$ from $Q$, insert $v_5$ into $Q$ and $T$, and delete $v_5$ from Adjacency List of $v_4$. 

$Q$:

<table>
<thead>
<tr>
<th>$v_4$</th>
<th>431</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_4$</td>
<td>492</td>
</tr>
</tbody>
</table>

$T$:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
### Adjacency List:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₀</td>
<td></td>
</tr>
<tr>
<td>v₁</td>
<td>v₂</td>
</tr>
<tr>
<td>v₂</td>
<td></td>
</tr>
<tr>
<td>v₃</td>
<td></td>
</tr>
<tr>
<td>v₄</td>
<td>v₂</td>
</tr>
<tr>
<td>v₅</td>
<td></td>
</tr>
</tbody>
</table>

**Step 3.8:** Delete v₅ from Q, since v₅ is the target vertex, stop!!

\[
Q: 
\]

The shortest path from v₀ to v₅ constrained to 90° is v₀ → v₁ → v₄ → v₅. Cost of the path is 592. The angle constrained shortest path tree is as follows.

![An Angle Constrained Shortest Path Tree](image)

**Figure 2-3: An Angle Constrained Shortest Path Tree**

We can see that the vertex v₄ appears in the angle constrained shortest path tree twice. A vertex in the tree is either in the shortest path of itself, or acts as the relay vertex for the shortest path of another vertex.
2.3 Other Curvature Constrained Paths

Not much research results are reported on angle constrained paths in geometric graphs. Some researchers have considered curvature constrained paths that can be constructed to lie within a polygon [13]. Such paths are not required to touch the vertices of the polygon or obstacle but can consist of curves in free space. Some results in such types of paths are reported in [13].
MONOTONE ROUTES IN SENSOR NETWORKS

In this chapter we consider the construction of routes having directional properties. Routes with directional properties can be considered as special cases of angle constrained paths. The notion of directional routes has played important roles in several areas of computer applications that include robotics, computer aided design, VLSI, and of course sensor networks. We start with the formal definition of directional routes.

Definition 3.1: Direction $d$ is the direction of the direct $S/T$ line. A route $R$ is called $d$-monotone, if the projections of the line segments of $R$ into the direct $S/T$ line do not overlap. Figure 3-1 illustrates a $d$-monotone route $R$. 

![Figure 3-1: Illustrating a d-Monotone Route](image-url)
It is observed that a traversal along a $d$-monotone route always makes a positive advance in the direction of $d$.

While developing localized routing algorithms for sensor networks, it is very important to underscore memory usage. For location based routing algorithms the notion of a memoryless algorithm is defined as follows.

**Definition 3.2:** A location based routing algorithm is called **memoryless** if the next node is determined by only examining the coordinates of the current node, its neighbors, and the destination node.

### 3.1 Greedy Routing Algorithms

Several localized greedy routing algorithms proposed by sensor network researchers generate routes that tend to be monotone in the direction of propagation. Brief descriptions of these algorithms are given below. In all the algorithms, the goal is to construct a route connecting a source node $S$ to a target node $T$.

**Most forward routing (MFR).** This is one of the most widely-used greedy routing algorithms in mobile computing and sensor networks [14]. In this scheme, the route is constructed by adding one node at a time, starting from node $S$, to make maximum positive progress toward node $T$. To add a new node to the partially constructed route $<S,v_i>$, the algorithm examines the neighbors $N(v_i)$ of the most recently added node $v_i$. Node $w \in N(v_i)$ which minimizes the Euclidean distance $\text{dist}(w, T)$ is the next node added to the partial route. At each node $v_i$, the next node $w$ is identified by examining the locations of $v_i$, $N(v_i)$, and $T$ and hence the algorithm can be considered to be a purely memoryless local algorithm. The algorithm can successfully construct the route if the
next node selected by the current node is not already in the partially constructed route.

Figure 3-2(a) shows a route constructed by using the MFR scheme. Figure 3-2(b) illustrates a network where the MFR fails to construct the route. A formal sketch of this algorithm is as follows:

**Sketch of Most Forward Routing**

```plaintext
nextNode(S, T, v)
{
    Pick the neighbor w of v that satisfies the following condition:
    (i) Length of line segment (w, T) is minimized.
}
```

![Figure 3-2: Most Forward Routes](image)
**Compass routing (CR).** This is also a greedy algorithm which determines the next node by considering the angle between the target node $T$, the current node $v$, and a candidate node $w \in N(v)$ [11]. The smaller of the two angles measured in a clockwise and a counterclockwise direction from the directed line $(v, T)$ is taken as the *compass angle*. The adjacent node $w$ that minimizes the compass angle $\angle (T, v, w)$ is taken as the next node added to the partial route. By the rule of the next node selection, it is clear that compass routing is a purely memoryless local algorithm. Purely memoryless CR algorithm can construct routes successfully for most instances of triangulated graphs. For certain types of triangulated graphs, the CR algorithm may be trapped in an infinite loop. Figure 3-3(a) illustrates a route constructed by applying CR algorithm. Figure 3-3(b) illustrates a network where the CR fails to construct the route. A variation of CR algorithm that uses $O(1)$ memory is known to generate S/T route successfully in any connected planar straight line graph [14]. A formal sketch of the compass routing algorithm is as follows.

**Sketch of Compass Routing**

```plaintext
nextNode(S, T, v) {
    Pick the neighbor $w$ of $v$ that satisfies the following condition:
    (i) Compass angle $\angle (T, v, w)$ is minimized
}
```
Lateral span routing (LSR). This scheme is developed by considering the distance of the candidate node to the direct segment \((S,T)\) [5]. The new node added to the partial route \(<S,v_i>\) is the neighbor of \(v_i\) that minimizes the distance (perpendicular) to the direct segment and also makes positive progress toward the target node. By looking for a neighbor that is nearest to the direct segment, the constructed route tends to be closer to the direct segment. Intuitively, routes whose nodes are close to the direct segment tend to be of shorter lengths. A formal sketch of the LSR algorithm is given below.
Sketch of Lateral Span Routing

```
nextNode(S, T, v)
{
    Pick the neighbor w of v that satisfies the following condition:
    (i) Distance from w to direct S/T line is minimized.
    (ii) Make positive progress toward the target node.
    (iii) The projection of w on line S/T lies within the segment S/T.
}
```

Figure 3-4 and 3-5 show examples of routes generated by CR, and LSR.

(a) Compass Route                                (b) Lateral Span Route

Figure 3-4: CR Performs Better Than LSR
For each technique, we can construct an example graph where it has better performance than the other.

### 3.2 Monotone Routes in Triangulated Networks

Let $d$ be the direction of the line through the start vertex $S$ and target vertex $T$. We want to characterize triangulated networks that admit a $d$-monotone route from $S$ to $T$. Without loss of generality, we assume that the direction $d$ is along the $y$-axis upward. Each non-horizontal edge is assigned a direction which is closest to the direction $d$. With this direction assignment, the edges of the network can be distinguished into four kinds.
Edges inclined clockwise from $d$ are termed as right-inclined edges, and the edges inclined counterclockwise from $d$ are the left-inclined edges. The other two kinds are the vertical edges and horizontal edges.

Definition 3.3: An ordered sequence of adjacent triangles $t_1, t_2, t_3, \ldots, t_k$ is called a sleeve if the dual of the sequence does not form a cycle.

![Sleeves and non-Sleeves](image)

(a) Sleeve  (b) Sleeve  (c) non-Sleeve

Figure 3-6: Illustrating Sleeve and non-Sleeve

Definition 3.4: A $d$-monotone sleeve is a sleeve whose boundary can be partitioned into two parts. One part is called the left-chain and the other part is the right-chain with obvious meaning. Figure 3-6 (a) and (b) show $d$-monotone sleeves.
**Definition 3.5:** The *cross-segments* connecting right-chain to left-chain can be assigned directions from right to left. A $d$-monotone sleeve is called **blocking** if all cross-segments are directed downward. Figure 3-6 (b) is a blocking sleeve, but Figure 3-6 (a) is not.

Consider the horizontal strip $HS$ bounded by horizontal lines through the topmost and the bottommost nodes of a blocking sleeve $B$. $B$ partitions the $HS$ into three parts: the sleeve itself, *left-unbounded region* $R_L(B)$ and the *right-unbounded region* $R_R(B)$. The notion of $R_L(B)$ and $R_R(B)$ is valid for a single non-horizontal edge (Figure 3-7).

![Diagram](image)

**Figure 3-7:** Illustrating the Left-Unbounded Region and Right-Unbounded Region

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Lemma 3.1: A triangulated network $G$ admits a $d$-monotone $S/T$ path if and only if there is no $d$-monotone blocking sleeve separating $S$ and $T$ into different regions, where $d$ is the direction of line $(S,T)$.

**Proof:** If nodes $S$ and $T$ are separated by a $d$-monotone blocking sleeve, then assume without loss of generality that $S$ is in the right-unbounded region and $T$ in the left-unbounded region (Figure 3-7). Part of the $S/T$ path must connect a node in the right-chain to a node in the left-chain. By the definition of a $d$-monotone blocking sleeve, there is no upward edge connecting right-chain to left-chain. This implies that $S$ and $T$ cannot be connected by a $d$-monotone path. □

3.2.1 Centralized Algorithms for $d$-Monotone Routes

We could develop an algorithm to construct $d$-monotone routes by using the characterization lemma (Lemma 3.1). However, we can develop a simpler algorithm based on searching a path on a directed graph. We construct a directed graph $G$ from the triangulated network $TN$ by assigning directions to edges of $TN$. Left-inclined, right-inclined and vertical edges are given directions to make their directions closer to $d$. Horizontal edges are considered as a pair of twin edges directed in opposite directions. Figure 3-8 shows the directed graph obtained from a Delaunay triangulation. In the resulting directed graph $G$, we can apply a standard graph search technique such as the breath first search (BFS) algorithm [17] to generate a BFS tree rooted at $S$. A $d$-monotone path can be extracted from the BFS tree. The algorithm is sketched formally as Algorithm 3.1.
Algorithm 3.1: (Constructing a $d$-Monotone Route)

(i) Assign directions to left-inclined, right-inclined and vertical edges in $G$ to make their directions closer to $d$. // $d$ is direction of the line through $S$ to $T$
(ii) Assign bi-directions to horizontal edges in $G$.
(iii) Apply BFS algorithm to $G$ starting from $S$.
(iv) If ($T$ is in the resulting BFS tree rooted at $S$)
    Report the monotone path obtained by following the parent pointers from $T$ to $S$ in $G$.
Else
    Report that the monotone path does not exist.

(a) A Delaunay Network $T_N$
(b) Corresponding Directed Graph $G$

Figure 3-8: Illustrating the Construction of Directed Graph

To compute the shortest $d$-monotone route we could apply a shortest path algorithm such as Dijkstra's single source shortest path algorithm. However, if there is no horizontal edge in the graph, the structure of the network can be exploited to develop a faster algorithm. It is easily seen that the directed graph constructed as above does not
contain any cycle, i.e., it is a DAG (Directed Acyclic Graph). We can use the standard single source shortest path algorithm for DAGs [18] to obtain the shortest $d$-monotone route. The algorithm based on this approach is listed as Algorithm 3.2:

**Algorithm 3.2: (Constructing the Shortest $d$-Monotone Route)**

(i) Assign directions to left-inclined, right-inclined and vertical edges in $G$ to make their directions closer to $d$. \quad /\text{Id is direction of the line through S to T}

(ii) Apply the DAG shortest path algorithm to $G$ starting from $S$.

(iii) If ($T$ is in the resulting shortest path tree rooted at $S$)

       Report the shortest monotone path obtained by following the parent pointers from $T$ to $S$ in $G$.

       Else

       Report the monotone path does not exist.

Time Complexity Analysis: Since the BFS algorithm takes $O(|E| + |V|)$ time, it is straightforward to see that the time complexity of both Algorithm 3.1 and Algorithm 3.2 is $O(|E| + |V|)$.

3.3 Monotone Routes in Delaunay Networks

One of the most widely-used network models is the Delaunay triangulation (DT). DT of a set of point nodes is constructed by using empty circle tests [8,12]. Three sensor nodes $a$, $b$, and $c$ are connected to form a triangle if the circle passing through nodes $a$, $b$ and $c$ is empty. Localized algorithms for constructing DT of sensor nodes have been reported [7]. It is interesting to examine the existence of $d$-monotone routes between any pair of nodes in a DT.

**Lemma 3.2.:** Any two nodes of a Delaunay Triangulation network cannot be separated by a $d$-monotone blocking sleeve.
Proof: Assume to the contrary that two vertices $S$ and $T$ are separated by a $d$-monotone blocking sleeve in a DT network $G$. Let $D_1$ be the $d$-monotone blocking sleeve (Figure 3-9 (a)).

![Diagram](image)

**Figure 3-9: Proof for Lemma 3.2**

Consider the last $(k^{th})$ component of the $d$-monotone blocking sleeve $D_1$, and the isolated rectangle $R(v_a m v_b n)$ with $t_k$ as the diagonal (Figure 3-9 (b)). Consider the two triangles adjacent to $t_k$. The other node ($v_c$) must be inside the lower triangle $(v_a m v_b)$ of rectangle $R$ by the definition of a $d$-monotone blocking sleeve. To be $d$-monotone blocked from node $S$, node $T$ must be inside the upper triangle $(v_a m v_b)$ of rectangle $R$. Observe that the
circle through $v_a$, $v_b$, and $v_c$ contains node $T$, implying that $t_k$ is not an edge of the DT network – a contradiction. □

The existence of $d$-monotone routes in a DT network can be established by using the technical lemma for bounding the length of the shortest path in a DT given in [2].

**Lemma 3.3:** There is always a $d$-monotone route between any pair of nodes in a Delaunay triangulation network.

**Proof:** Consider the Voronoi diagram of the given DT network. It is well known that DT is the straight-line dual of the corresponding Voronoi diagram. The direct $S/T$ line intersects with a sequence of Voronoi faces $(f_0, f_1, f_2, \ldots, f_n)$ as shown in Figure 3-10. By connecting all Delaunay vertices $(p_0, p_1, p_2, \ldots, p_n)$ corresponding to this sequence of Voronoi faces, a DT path can always be constructed successfully. We call this path the Voronoi-DT path (Figure 3-10).

![Figure 3-10: A Voronoi-DT Path](image)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Dobkin, Friedman, and Supowit [2] have established that the Voronoi-DT path ($S$ is $p_0$, $T$ is $p_n$) is monotone along the $S/T$ direction. □

To construct the shortest $d$-monotone route in a DT network, we can apply Algorithm 3.2 in a straightforward way. DT is a planar graph and hence the number of edges is linearly related to the number of vertices. The time complexity is linear in the number of vertices.

### 3.3.1: Localized Algorithms for $d$-monotone Routes

The algorithms for constructing $d$-monotone routes presented above (Algorithm 3.1 and Algorithm 3.2) are centralized algorithms. Complete knowledge of the network is necessary to execute centralized algorithms. For sensor network applications, it is desirable to develop localized algorithms that can be executed online without having complete knowledge of the entire network. Memoryless localized algorithms would be highly desirable. We now proceed to propose two new localized algorithms for constructing $d$-monotone routes in DT networks.

One of the most used localized routing algorithms is the compass routing (CR) [11]. The routes generated by CR algorithm tend to be $d$-monotone for most of the DT networks. On rare occasions, the route constructed by the CR algorithm may not be $d$-monotone. Figure 3-11 (a) shows a non-$d$-monotone route constructed by CR algorithm in a DT network. However, the standard CR algorithm can be modified to generate only $d$-monotone routes by requiring a simple check (Figure 3-11 (b)). In the Modified Compass Routing algorithm (MCR), when the current node $A$ examines its neighbors to select the next node, it picks the neighbor $B$ that satisfies two conditions: (i) the node that
minimizes the angle $\angle (T, A, B)$ – the standard CR condition, and (ii) the projected point of the selected node $B$ on the line $(S, T)$ should be within the segment $(S, T)$.

If the route generated by the standard CR algorithm is not $d$-monotone then there is only one occurrence of the back-up edge. In fact it can be established in a straightforward way that a back-up edge can occur only as the last edge in the generated route. This is stated in the following observation.

Observation 3.1: If the route generated by the standard CR algorithm is not $d$-monotone, then the back-edge must be the last edge.
Sketch of Modified Compass Routing

nextNode(S, T, v)
{
    Pick the neighbor w of v that satisfies the following two conditions:
    (i) Compass angle $\angle (T, v, w)$ is minimized
    (ii) The projection of w on line S/T lies within the segment S/T.
}

Routes generated by the Lateral Span Routing algorithm are always $d$-monotone by definition. However, quality of the generated routes may not always be satisfactory. Sometimes, the generated path may cross the direct S/T line many times which tends to increase the total length significantly. For this reason, we propose a variation of the Lateral Span Routing algorithm called the Hybrid Lateral/Compass Routing algorithm (HR). HR works by mixing both the lateral span routing strategy and the compass routing strategy. If LSR crosses the direct S/T line then the MCR scheme is used; otherwise, the LSR scheme is used. A formal sketch is given below.

Sketch of Hybrid Lateral/Compass Routing

nextNode(S, T, v)
{
    (i) Let $w_1$ be the neighbor of v determined by using LSR scheme.
    (ii) Let $w_2$ be the neighbor of v determined by using MCR scheme.
    (iii) If segment $(v, w_1)$ intersects with line $(S, T)$
        If segment $(v, w_2)$ does not intersect with line $(S, T)$
            Return $w_2$;
        Else
            Return $w_1$;
    Else
        Return $w_1$;
}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 4

IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this chapter, we mainly consider the implementation and experimental investigation of routing algorithms. For the representation of the geometric network, we use the planar straight line graph (PSLG) in doubly connected edge list (DCEL) form from computational geometry [12]. We implement DCEL from scratch so that a PSLG can be either constructed interactively from a graphical user interface (GUI) or read from a file. When PSLG is available in DCEL form, the network can be traversed easily and neighbor computations can be done efficiently without exploring the entire network.

By using a DCEL representation, several versions of $d$-monotone route constructing algorithms (both centralized and localized) are implemented. The implementation also includes provisions for constructing (i) shortest paths by using the standard Dijkstra’s algorithm, and (ii) shortest paths constrained to be within a given turn angle. The program can be used to construct Delaunay triangulation and represent it in DCEL form. Performances of localized routing algorithms that include Modified Compass Routing (MCR) and Hybrid Lateral/Compass Routing (HR) are reported. Performance evaluation of these algorithms is done by comparing the routes generated by them to the routes generated by the centralized shortest $d$-monotone path algorithm. Algorithms are tested on randomly generated Delaunay graphs.
4.1 Overview of Doubly Connected Edge List Data Structure

Doubly connected edge list (DCEL) data structure is one of the most popular data structures from computational geometry [12] for representing any planar straight line graph (PSLG). Since we implement this data structure from scratch using the Java programming language, we first briefly sketch the main ingredients of the data structure and then describe the implementation. A DCEL consists of a set of vertices $V$, a set of half-edges $E$, and a set of faces $F$. In DCEL representation, each edge of the original graph is represented by two half-edges in opposite directions. Each half-edge $(e_i)$ is associated with (i) two vertices (its start vertex and its end vertex), (ii) one face (its incident face), and (iii) three other half-edges (twin half-edge, previous half-edge and next half-edge). The incident face of a directed half-edge $e_i$ is the face to its left. Two half-edges $e_i$ and $e_j$ are twin edges of each other if one’s start vertex is the end vertex of the other and vice versa. The boundary of a face can be described by the closed sequence of half-edges. With respect to this closed sequence, each edge in the sequence has a unique previous edge and a unique next edge. Specifically the record of a half-edge can be listed as:

**Half-edge Record** ($e_i$):

- **edgeID**: the index of $e_i$
- **startVertex**: the start point of $e_i$
- **endVertex**: the end point of $e_i$
- **twinEdge**: the twin half-edge of $e_i$
- **nextEdge**: the next half-edge of $e_i$
- **prevEdge**: the previous half-edge of $e_i$
- **incFace**: the face on which $e_i$ is incident

A vertex record in DCEL includes fields for an incident half-edge, the $x$ and $y$ coordinates of the point corresponding to the vertex, and some other optional information.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
such as the list of neighbors, or the degree of the vertex, and other information as needed in specific applications. Specifically, the record of a vertex can be listed as:

**Vertex Record** \((v_i)\):
- vertexID: the index of \(v_i\)
- incHalfEdge: one of the half-edges incident at \(v_i\)
- thePoint: the x and y coordinates of \(v_i\)

A face record includes fields for a bounding half-edge and some other optional information such as the area and size of the face.

**Face Record** \((f_i)\):
- faceID: the index of \(f_i\)
- aBoundingHalfEdge: one of the half-edges bounding \(f_i\)

In our program each of these records (Half-edge, Vertex, and Face) is designed as Java class. We now describe how to construct a DCEL representation for a PSLG. We describe with an example PSLG shown in Figure 4-1. It may be noted that a planar straight line graph (PLSG) is a planar graph embedded in the plane with all edges as line segments. In a planar graph, the number of faces \(numF\), the number of edges \(numE\), and the number of vertices \(numV\) are linearly related:

\[
numV - numE + numF = 2.
\]

Delaunay triangulations, Gabriel graphs, relative neighborhood graphs, and Voronoi diagrams are all PSLGs. The PSLG shown in Figure 4-1 has ten vertices, sixteen edges and eight faces. \(f_0\) is unbounded while \(f_1, f_2, f_3, f_4, f_5, f_6,\) and \(f_7\) are bounded.
To represent a PSLG in DCEL, we include three fields for a PSLG record. Elements in the vertexVector are a list of DCEL vertex records. Elements in the halfEdgeVector are a list of DCEL half-edge records. Elements in the faceVector are a list of DCEL face records.

**PSLG record** \((G_i)\):
- vertexVector: a list of the vertices of \(G_i\)
- halfEdgeVector: a list of half-edges of \(G_i\)
- faceVector: a list of faces of \(G_i\)

A DCEL representation of the PSLG in Figure 4-1 is illustrated in Figure 4-2. We name the unbounded face \(f_0\) as the **outer face**. All other faces are bounded and called
**inner faces.** Bounding edges of the inner faces are running counterclockwise while edges of the outer face are running clockwise.

![Diagram of DCEL representation of a PSLG](image)

Figure 4-2: Illustrating the DCEL representation of a PSLG

A list of the vertex records, a list of the half-edge records, and a list of the face records for the PSLG in Figure 4-2 are shown in Table 4-1, Table 4-2, and Table 4-3 respectively.
Table 4-1: Vertex Records in the vertexVector for Figure 4-2

<table>
<thead>
<tr>
<th>Vertex</th>
<th>xCor</th>
<th>yCor</th>
<th>incHalfEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>505</td>
<td>303</td>
<td>e31</td>
</tr>
<tr>
<td>v1</td>
<td>350</td>
<td>297</td>
<td>e25</td>
</tr>
<tr>
<td>v2</td>
<td>305</td>
<td>467</td>
<td>e17</td>
</tr>
<tr>
<td>v3</td>
<td>513</td>
<td>460</td>
<td>e21</td>
</tr>
<tr>
<td>v4</td>
<td>346</td>
<td>680</td>
<td>e5</td>
</tr>
<tr>
<td>v5</td>
<td>164</td>
<td>463</td>
<td>e29</td>
</tr>
<tr>
<td>v6</td>
<td>501</td>
<td>119</td>
<td>e27</td>
</tr>
<tr>
<td>v7</td>
<td>672</td>
<td>257</td>
<td>e1</td>
</tr>
<tr>
<td>v8</td>
<td>653</td>
<td>556</td>
<td>e3</td>
</tr>
<tr>
<td>v9</td>
<td>235</td>
<td>175</td>
<td>e9</td>
</tr>
</tbody>
</table>

Table 4-2: Half-edge Records in the halfEdgeVector for Figure 4-2

<table>
<thead>
<tr>
<th>Edge</th>
<th>startVertex</th>
<th>endVertex</th>
<th>nextEdge</th>
<th>prevEdge</th>
<th>twinEdge</th>
<th>incFace</th>
</tr>
</thead>
<tbody>
<tr>
<td>e0</td>
<td>v6</td>
<td>v7</td>
<td>e2</td>
<td>e10</td>
<td>e1</td>
<td>f0</td>
</tr>
<tr>
<td>e1</td>
<td>v7</td>
<td>v6</td>
<td>e27</td>
<td>e31</td>
<td>e0</td>
<td>f7</td>
</tr>
<tr>
<td>e2</td>
<td>v7</td>
<td>v8</td>
<td>e4</td>
<td>e0</td>
<td>e3</td>
<td>f0</td>
</tr>
<tr>
<td>e3</td>
<td>v8</td>
<td>v7</td>
<td>e30</td>
<td>e5</td>
<td>e2</td>
<td>f6</td>
</tr>
<tr>
<td>e4</td>
<td>v8</td>
<td>v4</td>
<td>e6</td>
<td>e2</td>
<td>e5</td>
<td>f0</td>
</tr>
<tr>
<td>e5</td>
<td>v4</td>
<td>v8</td>
<td>e3</td>
<td>e21</td>
<td>e4</td>
<td>f6</td>
</tr>
<tr>
<td>e6</td>
<td>v4</td>
<td>v5</td>
<td>e8</td>
<td>e4</td>
<td>e7</td>
<td>f0</td>
</tr>
<tr>
<td>e7</td>
<td>v5</td>
<td>v4</td>
<td>e23</td>
<td>e28</td>
<td>e6</td>
<td>f4</td>
</tr>
<tr>
<td>e8</td>
<td>v5</td>
<td>v9</td>
<td>e10</td>
<td>e6</td>
<td>e9</td>
<td>f0</td>
</tr>
<tr>
<td>e9</td>
<td>v9</td>
<td>v5</td>
<td>e29</td>
<td>e11</td>
<td>e8</td>
<td>f5</td>
</tr>
<tr>
<td>e10</td>
<td>v9</td>
<td>v6</td>
<td>e0</td>
<td>e8</td>
<td>e11</td>
<td>f0</td>
</tr>
<tr>
<td>e11</td>
<td>v6</td>
<td>v9</td>
<td>e9</td>
<td>e25</td>
<td>e10</td>
<td>f5</td>
</tr>
<tr>
<td>e12</td>
<td>v3</td>
<td>v0</td>
<td>e14</td>
<td>e18</td>
<td>e13</td>
<td>f1</td>
</tr>
<tr>
<td>e13</td>
<td>v0</td>
<td>v3</td>
<td>e21</td>
<td>e30</td>
<td>e12</td>
<td>f6</td>
</tr>
<tr>
<td>e14</td>
<td>v0</td>
<td>v1</td>
<td>e16</td>
<td>e12</td>
<td>e15</td>
<td>f1</td>
</tr>
<tr>
<td>e15</td>
<td>v1</td>
<td>v0</td>
<td>e26</td>
<td>e24</td>
<td>e24</td>
<td>f3</td>
</tr>
<tr>
<td>e16</td>
<td>v1</td>
<td>v2</td>
<td>e18</td>
<td>e14</td>
<td>e17</td>
<td>f1</td>
</tr>
<tr>
<td>e17</td>
<td>v2</td>
<td>v1</td>
<td>e25</td>
<td>e29</td>
<td>e16</td>
<td>f5</td>
</tr>
<tr>
<td>e18</td>
<td>v2</td>
<td>v3</td>
<td>e12</td>
<td>e16</td>
<td>e19</td>
<td>f1</td>
</tr>
<tr>
<td>e19</td>
<td>v3</td>
<td>v2</td>
<td>e22</td>
<td>e20</td>
<td>e18</td>
<td>f2</td>
</tr>
<tr>
<td>e20</td>
<td>v4</td>
<td>v3</td>
<td>e19</td>
<td>e22</td>
<td>e21</td>
<td>f2</td>
</tr>
<tr>
<td>e21</td>
<td>v3</td>
<td>v4</td>
<td>e5</td>
<td>e13</td>
<td>e20</td>
<td>f6</td>
</tr>
<tr>
<td>e22</td>
<td>v2</td>
<td>v4</td>
<td>e20</td>
<td>e19</td>
<td>e23</td>
<td>f2</td>
</tr>
<tr>
<td>e23</td>
<td>v4</td>
<td>v2</td>
<td>e28</td>
<td>e7</td>
<td>e22</td>
<td>f4</td>
</tr>
<tr>
<td>e24</td>
<td>v6</td>
<td>v1</td>
<td>e15</td>
<td>e26</td>
<td>e25</td>
<td>f3</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 4-2: Half-edge Records in the halfEdgeVector for Figure 4-2 (Continued)

<table>
<thead>
<tr>
<th>Edge</th>
<th>startVertex</th>
<th>endVertex</th>
<th>nextEdge</th>
<th>prevEdge</th>
<th>twinEdge</th>
<th>incFace</th>
</tr>
</thead>
<tbody>
<tr>
<td>e25</td>
<td>v1</td>
<td>v6</td>
<td>e11</td>
<td>e17</td>
<td>e24</td>
<td>f5</td>
</tr>
<tr>
<td>e26</td>
<td>v0</td>
<td>v6</td>
<td>e24</td>
<td>e15</td>
<td>e27</td>
<td>f3</td>
</tr>
<tr>
<td>e27</td>
<td>v6</td>
<td>v0</td>
<td>e31</td>
<td>e1</td>
<td>e26</td>
<td>f7</td>
</tr>
<tr>
<td>e28</td>
<td>v2</td>
<td>v5</td>
<td>e7</td>
<td>e23</td>
<td>e29</td>
<td>f4</td>
</tr>
<tr>
<td>e29</td>
<td>v5</td>
<td>v2</td>
<td>e17</td>
<td>e9</td>
<td>e28</td>
<td>f5</td>
</tr>
<tr>
<td>e30</td>
<td>v7</td>
<td>v0</td>
<td>e13</td>
<td>e3</td>
<td>e31</td>
<td>f6</td>
</tr>
<tr>
<td>e31</td>
<td>v0</td>
<td>v7</td>
<td>e1</td>
<td>e27</td>
<td>e30</td>
<td>f7</td>
</tr>
</tbody>
</table>

Table 4-3: Face Records in the faceVector for Figure 4-2

<table>
<thead>
<tr>
<th>Face</th>
<th>boundingEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>e0</td>
</tr>
<tr>
<td>f1</td>
<td>e12</td>
</tr>
<tr>
<td>f2</td>
<td>e20</td>
</tr>
<tr>
<td>f3</td>
<td>e24</td>
</tr>
<tr>
<td>f4</td>
<td>e7</td>
</tr>
<tr>
<td>f5</td>
<td>e9</td>
</tr>
<tr>
<td>f6</td>
<td>e3</td>
</tr>
<tr>
<td>f7</td>
<td>e1</td>
</tr>
</tbody>
</table>

In our implementation, there are two ways to construct a PSLG in DCEL representation: (i) Interactive construction from a graphical user interface (GUI); and (ii) Construction from a data file.

4.1.1 Interactive Construction of Planar Straight Line Graph

A PSLG can be interactively constructed by performing the following operations:

- **Add-edge** \((v_i, v_j)\): This operation adds a new vertex \(v_i\) and connects \(v_i\) to an existing vertex \(v_j\) by effectively adding a new edge (two half-edges). If
the initial PSLG is the triangle $G_1$, add $(v_9, v_4)$ operation results in the graph $G_2$ (Figure 4-3).

![Figure 4-3: Illustrating Add-edge](image)

- **Split-edge** $(e_i, e_j, v_i)$: An existing edge is split into two parts (four half-edges) by inserting a new vertex $v_i$ near the mid-point of edge $e_i e_j$. We obtain $G_3$ by the operation *split-edge* $(e_6 e_7, v_5)$ on $G_2$ (Figure 4-4).

![Figure 4-4: Illustrating Split-edge](image)
- **Move-vertex** ($v_i$): This operation moves the position of an existing vertex $v_i$. *Move-vertex* ($v_9$) operation on $G_3$ results in the graph $G_4$ (Figure 4-5).

![Diagram of Move-vertex](image)

**Figure 4-5: Illustrating Move-vertex**

- **Split-face** ($f_i, f_j$): This operation creates a new face $f_i$ by adding a new edge (two half-edges) to split an existing face $f_j$ into two faces. $G_5$ is obtained by the operation *Split-face* ($f_4, f_0$) on $G_4$ (Figure 4-6). We can see that $G_4$ is part of the PSLG in Figure 4-2.
A PSLG can also be interactively updated by performing the delete-vertex \(v_j\) operation.

- **Delete-vertex \(v_j\):** This operation deletes a vertex \(v_j\), and zero or more half-edges incident at \(v_j\) depending on the degree of \(v_j\). If \(v_j\) is a vertex of a triangle face or \(v_j\) belongs to several faces, deletion of \(v_j\) also causes deletion of the corresponding face(s). *Delete-vertex \(v_9\) operation on \(G_5\) produces \(G_6\) (Figure 4-7). Delete-vertex \(v_3\) operation on \(G_6\) produces \(G_7\) (Figure 4-8).
It is straightforward to observe that each of the above operations except for split-face can be done in constant time – it is a matter of updating a constant number of half-edge and vertex records. For split-face operation, we need to update $O(m)$ half-edge records,
where $m$ is the number of edges in the original face. Hence split-face operation takes $O(m)$ time. Also delete-vertex operation takes constant time for simple cases and can take $O(m)$ time if the deletion of the vertex triggers the modification of many faces on the vertex.

4.1.2 Construction of Planar Straight Line Graph from A Data File

We now describe how to construct a PSLG from a data file. The format of the input data file is given in Table 4-4. We further illustrate the input data file format by giving an example in Table 4-5, which is the input data file for the PSLG in Figure 4-2.

Table 4-4: Format of the Input Data File for A PSLG (Format A)

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Data Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Total number of vertices</td>
</tr>
<tr>
<td>2</td>
<td>x and y coordinates of the vertices (x1, y1, x2, y2, ...xn, yn);</td>
</tr>
<tr>
<td>3</td>
<td>Total number of faces</td>
</tr>
<tr>
<td>4</td>
<td>Size of the outer face</td>
</tr>
<tr>
<td>5</td>
<td>Vertex indices of the outer face, listed clockwise (starting from line 6 are data for inner faces, order of the inner faces listed doesn’t matter, with the even lines as the sizes of the faces and odd lines as the vertex indices of the faces, listed counterclockwise)</td>
</tr>
<tr>
<td>6</td>
<td>Size of an inner face</td>
</tr>
<tr>
<td>7</td>
<td>Vertex indices of the inner face</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Size of an inner face</td>
</tr>
<tr>
<td>k+1</td>
<td>Vertex indices of the inner face</td>
</tr>
</tbody>
</table>
Table 4-5: Input Data File for the PSLG in Figure 4-2

<table>
<thead>
<tr>
<th>Description</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of vertices</td>
<td>10</td>
</tr>
<tr>
<td>x and y coordinates of the vertices</td>
<td>505 303 350 297 305 467 513 460 346 680 164 463 501</td>
</tr>
<tr>
<td></td>
<td>119 672 257 653 556 235 175</td>
</tr>
<tr>
<td>Total number of faces</td>
<td>8</td>
</tr>
<tr>
<td>Size of the outer face ($f_0$)</td>
<td>6</td>
</tr>
<tr>
<td>Vertex indices of $f_0$</td>
<td>6 7 8 4 5 9</td>
</tr>
<tr>
<td>Size of $f_1$</td>
<td>4</td>
</tr>
<tr>
<td>Vertex indices of $f_1$</td>
<td>3 0 1 2</td>
</tr>
<tr>
<td>Size of $f_2$</td>
<td>3</td>
</tr>
<tr>
<td>Vertex indices of $f_2$</td>
<td>4 3 2</td>
</tr>
<tr>
<td>Size of $f_3$</td>
<td>3</td>
</tr>
<tr>
<td>Vertex indices of $f_3$</td>
<td>6 1 0</td>
</tr>
<tr>
<td>Size of $f_4$</td>
<td>3</td>
</tr>
<tr>
<td>Vertex indices of $f_4$</td>
<td>5 4 2</td>
</tr>
<tr>
<td>Size of $f_5$</td>
<td>5</td>
</tr>
<tr>
<td>Vertex indices of $f_5$</td>
<td>9 5 2 1 6</td>
</tr>
<tr>
<td>Size of $f_6$</td>
<td>5</td>
</tr>
<tr>
<td>Vertex indices of $f_6$</td>
<td>8 7 0 3 4</td>
</tr>
<tr>
<td>Size of $f_7$</td>
<td>3</td>
</tr>
<tr>
<td>Vertex indices of $f_7$</td>
<td>7 6 0</td>
</tr>
</tbody>
</table>

When we read the input data file of a PSLG, a DCEL representation for the PSLG can be achieved quickly by feeding data into the PSLG record (vertexVector, halfEdgeVector, and faceVector). Face by face, PSLG is constructed starting from the unbounded outer face $f_0$. After the outer face is constructed, other faces can be constructed in any order. At the very beginning, a temporary vector edgeRecordSetLater is created to search for the edge records efficiently. The outer face $f_0$ is constructed by default as the first element of the faceVector. Given the x and y coordinates of a point, we first fill the field thePoint of the corresponding vertex record. The incHalfEdge field is recorded later in the face.
construction stage. If there is only one vertex in a PSLG, then the PSLG has only the outer face $f_0$, and it has no bounding half-edges. If an inner face is constructed, then a new face record is created and the faceVector is updated accordingly. The first edge of the currently constructed face is taken as the bounding half-edge of the face. A vertex can possibly have many incident edges. One of the incident edges is picked arbitrarily as the incHalfEdge. Before we create a half-edge, we have to first check whether the half-edge has been created by examining the edgeRecordSetLater vector to avoid creating duplicates. If the half-edge is found in the edgeRecordSetLater vector, we need to update that edge’s record and then delete it from edgeRecordSetLater. If the half-edge is not in the edgeRecordSetLater, we need to create the half-edge and its twin. Although twin half-edges are created at the same time, only records of the edges belonging to the currently constructed face are set while their twin half-edges are saved in the edgeRecordSetLater vector. Fields in the half-edge records that include twinEdge, prevEdge, nextEdge, and incFace are updated next. We need to be careful while adding the edge that closes the currently constructed face. When a face closing edge is added, its next edge is the first edge of the current face. The vertexID, edgeID, and faceID fields are their respective indices in the corresponding vectors. Construction of the PSLG is completed when all inner faces have been added. A formal sketch of the PSLG construction as described above is listed below as Algorithm Construct PSLG. The sketch is described in terms of DCEL data structure fields and operations.
Algorithm Construct PSLG

Input: An input data file of a PSLG in Format A.
Output: A DCEL representation of a PSLG in three vectors: vertexVector, halfedgeVector, faceVector.

Step 1:

vertexVector = Set of vertices obtained from the 1st and 2nd line of the file;
int numF = the number of faces obtained from the 3rd line of the file;
Vector outerIndexVector = Vertex Indices of the outer face obtained from the 4th and 5th line of the file;
boolean IsOuterFace = true;
Vector innerFaces = Vertex Indices of the inner faces obtained from the 6th line to the end of the file, each element of it is an innerIndexVector;
Vector innerIndexVector = Vertex Indices of an inner face;

Step 2: if (vertexVector.size() > 1)
  ConstructFace(outerIndexVector, IsOuterFace);
  for (int i = 0; i < numF - 1; i++)
    innerIndexVector = innerFaces.elementAt(i);
    ConstructFace(innerIndexVector, !IsOuterFace);
}

Algorithm Construct Face

Input: Vector indexVector, // A vector containing the vertex indices of // of a face
       boolean IsOuterFace. // Flag indicating inner/outer face
Output: A DCEL representation of a PSLG face.

Step 1: Vector edgeRecordSetLater; // temporary vector
  int faceSize = indexVector.size();
  Face currentFace;
  Vertex vFirst, vS, vE;
  half-edge eFirst, eFirstTwin, eCurrent, eCurrentTwin;
  half-edge ePrev, eClose, eCloseTwin;
  int vIndex;

Step 2: if (IsOuterFace)
  currentFace = f0; // f0 is faceID of the outer face
else //It is an inner face
  currentFace = new Face();
Step 3:  

```java
vIndex = indexVector.elementAt(0);
vFirst = vertexVector.elementAt(vIndex);
vIndex = indexVector.elementAt(1);
vE = vertexVector.elementAt(vIndex);

//construct the first half-edge and its twin of the current face
if (edgeBeenConstructed (vFirst, vE) == null) {
    eFirst = new half-edge (vFirst, vE);
    eFirstTwin = new half-edge (vE, vFirst);
    eFirst.incFace = currentFace;
    eFirst.twinEdge = eFirstTwin;
    eFirstTwin.twinEdge = eFirst;
    edgeRecordSetLater.addElement(eFirstTwin);
    edgeVector.addElement(eFirst);
    edgeVector.addElement(eFirstTwin);
}
else { // the half-edge has already been constructed
    eFirst = edgeBeenConstructed (vFirst, vE);
    eFirst.incFace = currentFace;
    eFirstTwin = eFirst.twinEdge;
    //eFirst’s incident face is set, delete it from the
    //edgeRecordSetLater list
    edgeRecordSetLater.removeElement(eFirst);
}
currentFace.aBoudningHalfEdge = eFirst;

if (!IsOuterFace) {
    faceVector.addElement(currentFace);
    vFirst.incHalfEdge = eFirst;
}
else //the outer face
    vE.incHalfEdge = eFirstTwin;

ePrev = eFirst;
```

Step 4:  

```java
//construct the intermediate half-edges and their twins of the current face
for (int i = 1; i < faceSize - 1; i ++){
    vIndex = indexVector.elementAt(i);
    vS = vertexVector.elementAt(vIndex);
    vIndex = indexVector.elementAt(i + 1);
    vE = vertexVector.elementAt(vIndex);
```
if (edgeBeenConstructed (vS, vE) == null) {
  eCurrent = new half-edge (vS, vE);
  eCurrentTwin = new half-edge (vE, vS);
  eCurrent.incFace = currentFace;
  eCurrent.prevEdge = ePrev;
  eCurrent.twinEdge = eCurrentTwin;
  eCurrentTwin.twinEdge = eCurrent;
  edgeRecordSetLater.addElement(eCurrentTwin);
  edgeVector.addElement(eCurrent);
  edgeVector.addElement(eCurrentTwin);
}
else //the edge has been constructed{
  eCurrent = edgeBeenConstructed (vS, vE);
  eCurrent.incFace = currentFace;
  eCurrent.prevEdge = ePrev;
  eCurrentTwin = eCurrent.twinEdge;
  edgeRecordSetLater.removeElement(eCurrent);
}
if (IsOuterFace)
  vE.incHalfEdge = eCurrentTwin;
else
  vS.incHalfEdge = eCurrent;

  ePrev.nextEdge = eCurrent;
  ePrev = eCurrent;
}

Step 5: //construct the close half-edge and its twin of the current face
if (edgeBeenConstructed (vE, vFirst) == null) {
  eClose = new half-edge (vE, vFirst);
  eCloseTwin = new half-edge (vFirst, vE);
  eClose.nextEdge = eFirst;
  eClose.prevEdge = ePrev;
  eClose.twinEdge = eCloseTwin;
  eClose.incFace = currentFace;
  eCloseTwin.twinEdge = eClose;
  edgeRecordSetLater.addElement(eCloseTwin);
  edgeVector.addElement(eClose);
  edgeVector.addElement(eCloseTwin);
}
else  //the edge has been constructed{
    eClose = edgeBeenConstructed (vE, vFirst);
eClose.incFace = currentFace;
eClose.prevEdge = ePrev;
eClose.nextEdge = eFirst;
eCloseTwin = eClose.twinEdge;
    edgeRecordSetLater.removeElement(eClose);
}
eFirst.prevEdge = eClose;
ePrev.nextEdge = eClose;

if (IsOuterFace)
    vFirst.incHalfEdge = eCloseTwin;
else
    vE.incHalfEdge = eClose;

Given a PSLG constructed by the interactive GUI method, we can represent it in Format A to store in a data file in a straightforward way. If a planar graph is available in Format A, then our program can construct the PSLG in DCEL form.

4.2 Routes Construction

Our program includes components for constructing standard shortest paths, angle constrained shortest paths, and $d$-monotone shortest paths by using centralized algorithms. It also includes localized algorithms for computing $d$-monotone paths. All the algorithms are implemented by using the network available in the DCEL form. For all algorithms we take Delaunay triangulation as the underlying network model. We use the Delaunay triangulation Java code available from [9]. Our program converts the Delaunay triangulation from the format given in [9] to the DCEL format. We now describe how several algorithms can be implemented by using the DCEL data structure.
4.2.1 A Centralized Algorithm for Standard Shortest Paths

The sketch of the standard Dijkstra’s shortest path algorithm [17] is as follows.

Input: PSLG DT, vertex S, T.
Output: A shortest path from S to T in DT.
Step 1: int sl = S.vertexID;
         int tl = T.vertexID;
         int numofV = DT.vertexVector.size();

Step 2: boolean Included[numofV] initiated to false;
        double Distance[numofV] initiated to \infty;
        int Path[numofV] initiated to -1;

Step 3: Included[sl] = true;
        Distance[sl] = 0;
        Path[sl] = sl;
        for all neighbors v of S do{
            int vl = v.vertexID;
            Distance[vl] = half-edge(S,v).length;
            Path[vl] = sl;
        }

Step 4: while not every vertex in DT is included in the shortest path tree do{
        double miniDist = the smallest Distance[] value among the
                          vertices which are not yet included;
        vertex w = the vertex which has the miniDist;
        int miniIndex = w.vertexID;
        Included[miniIndex] = true;
        for all neighbors y of w which are not yet included do{
            int yl = y.vertexID;
            if (Distance[miniIndex] + half-edge(w,y).length
                < Distance[yI]) {
                Distance[yl] = Distance[miniIndex] +
                             half-edge(w,y).length;
                Path[yl] = miniIndex;
            }
        }
    }

Step 5: The shortest path from S to T is obtained:
        int k = T.vertexID;
        while (k \neq S.vertexID) do{
            k = Path[k];
        }

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 4-9 is a snapshot of the implementation of the above described algorithm.

Figure 4-10 is a snapshot of a shortest path tree in a large size DT (\( > 900 \) vertices).

Figure 4-9: A Shortest Path Constructed by Dijkstra's Algorithm in a DT
4.2.2 A Centralized Algorithm for Angle Constrained Shortest Paths

This angle constrained shortest path algorithm was presented by Boroujerdi and Uhlmann in 1997 [1]. The sketch of it in DCEL data structure is as follows.

Input: PSLG \( DT \), vertex \( S, T \), double \( \alpha \).
Output: A shortest path from \( S \) to \( T \) in \( DT \), in which the max turning is not greater than angle \( \alpha \).
Step 1: The adjacency list for each vertex \( v_m \) are initiated as \( v_m \)'s neighbors in \( DT \).
Step 2: A priority queue \( Q \) is initiated as \( null \).
A shortest path tree \( Tr \) is initiated as \( null \).
Step 3: Insert \( S \) into \( Q \) with priority 0. Insert \( S \) into \( Tr \).
Step 4: while $Q$ is not empty do {
    Retrive and delete the vertex $v_j$ with minimum priority $p$ from $Q$;
    if $v_j$ is $T$
        Stop and report the path is obtained (by following the parent pointers from $v_j$ to $S$ in $Tr$) with $p$ as the cost of the path.
    for each vertex $v_k$ in the adjacency list of $v_j$ such that the turn $v_jv_k$ is not greater than $a$ do {
        Insert $v_k$ into $Q$ with priority $p + \text{half-edge}(v_j, v_k)$.length;
        Insert $v_k$ into $Tr$ and set its parent pointer to $v_j$;
        Delete $v_k$ from the adjacency list of $v_j$;
    }
}

For comparison purposes, the network, start vertex, and target vertex are the same in Figure 4-9 and Figure 4-11.

Figure 4-11: A Shortest Path with Angle Constraint of $90^\circ$ in a DT
Since there is a sharp turn angle in the shortest path constructed in Figure 4-9, we put an angle constraint of 90° to remove the sharp turn in Figure 4-11.

4.2.3 A Centralized Algorithm for Shortest d-Monotone Routes

\(d\) is the direction of the direct path \(ST\). We first assign directions to all half-edges in the graph. Directions are made close to \(d\). And then we apply the Acyclic Shortest Path algorithm to the updated graph [18].

Input: PSLG \(DT\); vertex \(S, T\);
Output: A \(d\)-monotone shortest path from \(S\) to \(T\) in \(DT\)

Step 1: Assign directions to all half-edges \(e_{ij}\) in \(DT\). The directions are made close to \(d\). (e.g., by BFS)

Step 2: Initiate indegrees to all vertices \(v_i\) in \(DT\) according to the half-edge directions assigned in Step 1 (e.g., by DFS).

Step 3: For all vertices \(v_i\) in \(DT\) do

\[v_i.SPLength = \infty;\]
\[S.SPLength = 0;\]

Step 4: For all vertices \(v_i\) in \(DT\) do

If \((v_i\text{.indegree} == 0)\)

Put \(v_i\) into \(Queue;\)

Step 5: Do

remove vertex \(w\) from \(Queue;\)

For all neighbors \(z_j\) of \(w\) do

\{ if \((w.SPLength + \text{half-edge}(w, z_j).\text{length} < z_j.SPLength)\)

\{ \[z_j.SPLength = w.SPLength + \text{half-edge}(w, z_j).\text{length};\]
\[z_j.parent = w;\]
\}

\[z_j\text{.indegree} = z_j\text{.indegree} - 1;\]

if \((z_j\text{.indegree} == 0)\)

put \(z_j\) into \(Queue;\)

\} while \(Queue\) is not empty

Step 6: The \(d\)-monotone shortest path from \(S\) to \(T\) is obtained by following the parent pointers from \(T\) to \(S\).

Figure 4-12 is a snap shot of the implementation of the above described algorithm.
4.2.4 Localized Algorithms for $d$-Monotone Routes

We propose two localized algorithms for constructing $d$-monotone paths. First is the Modified Compass Routing (MCR) which makes a modification on the standard compass routing algorithm to ensure construction of a $d$-monotone path. Sketch of the algorithm is as follows:

**Algorithm: Modified Compass Routing**

- **Input:** PSLG $DT$; vertex $S$, $T$;
- **Output:** A short $d$-monotone path from $S$ to $T$ in $DT$.
- **Step 1:** Vector MCompassRoute; Vertex $currV$;
- **Step 2:** Put $S$ into MCompassRoute;
  
  $currV = S$;

![Figure 4-12: A Shortest $d$-Monotone Path in a DT](image)
Step 3: while (currV is not T) do {
    currV = findMCRNB(currV);
    put currV into MCompassRoute;
}

Step 4: Put T into MCompassRoute

Algorithm: findMCRNB

Input: vertex v;
Output: vertex mcmb, the minimum compass neighbor of v

Step 1: Vector nbofV = v.Neighbors;
       nbSize = nbofV.size;
       vertex w = nbofV.elementAt(0)
       double miniAngle = ∠Tv w;
       vertex mcmb = w;
       double tempAngle;

Step 2: for i from 1 to nbSize-1 do {
            w = nbofV.elementAt(i);
            tempAngle = ∠Tv w;
            if (tempAngle < miniAngle &&
                the projection of w on the direct line ST does not exceed T ) {
                miniAngle = tempAngle;
                mcmb = w;
            }
        }

return mcmb;

Figure 4-13 is a snap shot of the implementation of the above described algorithm.
The other localized algorithm we propose is the Hybrid Lateral/Compass Routing (HR). We build the route by finding the minimum lateral span neighbor (MLSNB) and the compass routing neighbor (CRNB) for the current vertex on the partially constructed route. If the next segment $\overrightarrow{Ir}$ given by MLSNB does not cause a crossing of the direct path $ST$, we add $\overrightarrow{Ir}$ to the partially constructed route. Otherwise, we examine CRNB. If segment $\overrightarrow{le}$ given by CRNB does not cause a crossing of the direct path $ST$, we add $\overrightarrow{le}$ to the partially constructed route. If both $\overrightarrow{Ir}$ and $\overrightarrow{le}$ cause a crossing, we take $\overrightarrow{Ir}$ as the default choice. Sketch of the algorithms is as follows.

**Algorithm: Hybrid Lateral/Compass Routing**

Input: PSLG $DT$; vertex $S$, $T$;  
Output: A short $d$-monotone path from $S$ to $T$ in $DT$
Step 1: Vector HybridRoute; Vertex currV;
Step 2: Put S into HybridRoute;
currV = S;
Step 3: while (currV is not T) do {
currV = findHybridNB(currV);
put currV into HybridRoute;
}
Step 4: Put T into HybridRoute

Algorithm: findHybridNB

Input: vertex v;
Output: vertex hybridnb, the hybrid lateral and compass neighbor of v
Step 1: Vector nbofV = v.Neighbors;
nbSize = nbofV.size;
Vector monotoneNBofV;
vertex w;
Step 2: for i = 0 to nbSize-1 do {
w = nbofV.elementAt(i);
if the turn of segment vw from the segment ST is not greater than 90°
Put w into monotoneNBofV;
}
Step 3: int monotoneNSize = monotoneNBofV.size;

w = monotoneNBofV.elementAt(0);
double miniAngle = \angle Tvw;
double miniLateral = w.Lateral(segment ST);
vertex mlsnb = w;
vertex crnb = w;
Step 4: for i = 0 to monotoneNSize-1 do {
w = monotoneNBofV.elementAt(i);
double tempAngle = \angle Tvw;
if (tempAngle < miniAngle &&
the projection of w on the direct line ST does not exceed T ) {
miniAngle = tempAngle;
crnb = w;
}
Double tempLateral = w.Lateral(segment ST);
if (tempLateral < miniLateral &&
the projection of w on the direct line ST does not exceed T) {
miniLateral = tempLateral;
mlsnb = w;
}
Step 5: 

if the segment $v, mlsnb$ does not cross the segment $ST$

$hybridnb = mlsnb$;

else{

if the segment $v, crnb$ does not cross the segment $ST$

$hybridnb = crnb$;

else

$hybridnb = mlsnb$;

}

return $hybridnb$;

Figure 4-14 shows the $d$-monotone route generated by executing the HR algorithm. For comparison purposes, the network, start vertex, and target vertex are the same in Figure 4-12, Figure 4-13, and Figure 4-14. The lengths of the routes generated by MCR and HR algorithms on this example network are very close.

Figure 4-14: A $d$-monotone Route Constructed by HR algorithm in a DT
The routes generated by MCR and HR algorithms on a large size graph (>900 vertices) are shown in Figure 4-15 and Figure 4-16 respectively.

Figure 4-15: A $d$-Monotone Route Constructed by MCR in a DT of 1000 Vertices
4.3 Experimental Results

To test the performance of our proposed algorithms, we constructed the Delaunay triangulation of a set of randomly generated vertices in the plane. Each vertex is generated by picking randomly-generated x and y coordinates in a square region of 900 by 900 pixels. In order to obtain long paths, we do not generate the positions for the start vertex $S$ and target vertex $T$ randomly, but select them by mouse click after visually inspecting the generated triangulations. For each generated DT graph, two pairs of $S$ and $T$ vertices are selected manually. The first pair is chosen near the diagonally opposite

Figure 4-16: A $d$-Monotone Route Constructed by HR in a DT of 1000 Vertices
corners of the rectangular canvas (SW-NE). The other pair is selected near the other corners (SE-NW). Delaunay triangulations for seven different vertex set sizes (50, 100, 200, 400, 600, 800, 1000) are considered. For each vertex set size, five DT graphs are randomly generated.

Since the region area (canvas size) is fixed, larger number of nodes results in higher density of node distribution. The routes generated by localized algorithms are compared with the routes generated by the shortest path algorithm for the same pair of $S$ and $T$. The quality of a path is measured in term of total length and the total number of hop-counts in the path. The shorter the total length, the better is the path. Similarly, the smaller the hop count, the better is the path. The $d$-monotone paths are generated by using MCR and HR algorithms. The shortest $d$-monotone paths are generated by the centralized Acyclic Shortest Path algorithm. The lengths and hops of the generated paths are tabulated in Table 4-6 and Table 4-7. Percentage comparisons are shown in Table 4-8 and Table 4-9.

<table>
<thead>
<tr>
<th>Vertices</th>
<th>ASP</th>
<th>MCR</th>
<th>HR</th>
<th>MCR/ASP</th>
<th>HR/ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1001.8</td>
<td>1015.9</td>
<td>1018.5</td>
<td>1.014</td>
<td>1.017</td>
</tr>
<tr>
<td>100</td>
<td>1040.7</td>
<td>1075.3</td>
<td>1074.4</td>
<td>1.033</td>
<td>1.032</td>
</tr>
<tr>
<td>200</td>
<td>951.1</td>
<td>968.9</td>
<td>977</td>
<td>1.019</td>
<td>1.027</td>
</tr>
<tr>
<td>400</td>
<td>1012.5</td>
<td>1033.6</td>
<td>1058.6</td>
<td>1.021</td>
<td>1.046</td>
</tr>
<tr>
<td>600</td>
<td>1042.3</td>
<td>1079</td>
<td>1082</td>
<td>1.035</td>
<td>1.038</td>
</tr>
<tr>
<td>800</td>
<td>1027.6</td>
<td>1048.5</td>
<td>1063.2</td>
<td>1.020</td>
<td>1.035</td>
</tr>
<tr>
<td>1000</td>
<td>1019.2</td>
<td>1041.6</td>
<td>1056.7</td>
<td>1.022</td>
<td>1.037</td>
</tr>
<tr>
<td>Total Avg.</td>
<td>1013.6</td>
<td>1037.543</td>
<td>1047.2</td>
<td>1.024</td>
<td>1.033</td>
</tr>
</tbody>
</table>
Table 4-7: Average Path Hops Comparison

<table>
<thead>
<tr>
<th>Vertices</th>
<th>ASP</th>
<th>MCR</th>
<th>HR</th>
<th>MCR/ASP</th>
<th>HR/ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>8.3</td>
<td>8.2</td>
<td>8</td>
<td>0.988</td>
<td>0.964</td>
</tr>
<tr>
<td>100</td>
<td>12.4</td>
<td>12.1</td>
<td>12.8</td>
<td>0.976</td>
<td>1.032</td>
</tr>
<tr>
<td>200</td>
<td>15.7</td>
<td>16.1</td>
<td>16.2</td>
<td>1.025</td>
<td>1.032</td>
</tr>
<tr>
<td>400</td>
<td>23.6</td>
<td>23.2</td>
<td>24.6</td>
<td>0.983</td>
<td>1.042</td>
</tr>
<tr>
<td>600</td>
<td>29.6</td>
<td>30.5</td>
<td>31.8</td>
<td>1.030</td>
<td>1.074</td>
</tr>
<tr>
<td>800</td>
<td>34.6</td>
<td>33.2</td>
<td>35.6</td>
<td>0.960</td>
<td>1.029</td>
</tr>
<tr>
<td>1000</td>
<td>36.2</td>
<td>37.3</td>
<td>39.7</td>
<td>1.030</td>
<td>1.097</td>
</tr>
<tr>
<td><strong>Total Avg.</strong></td>
<td><strong>22.914</strong></td>
<td><strong>22.943</strong></td>
<td><strong>24.1</strong></td>
<td><strong>1.0012</strong></td>
<td><strong>1.0517</strong></td>
</tr>
</tbody>
</table>

Table 4-8: Performance Comparison of MCR and HR by Path Length

<table>
<thead>
<tr>
<th>Total Number of Cases</th>
<th>Shorter Path by MCR</th>
<th>Shorter Path by HR</th>
<th>Equal Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>43</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>100%</td>
<td>61.43%</td>
<td>20%</td>
<td>18.57%</td>
</tr>
</tbody>
</table>

Table 4-9: Performance Comparison of MCR and HR by Path Hops

<table>
<thead>
<tr>
<th>Total Number of Cases</th>
<th>Fewer Hops by MCR</th>
<th>Fewer Hops by HR</th>
<th>Equal Number of Hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>38</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>100%</td>
<td>54.28%</td>
<td>14.29%</td>
<td>31.43%</td>
</tr>
</tbody>
</table>

An inspection of the data in the Table 4-8 shows that for 61.43% of the cases, Modified Compass Routing generates shorter routes than Hybrid Routing. For 20% of the cases, HR generates shorter routes than MCR. And they have equal performance in 18.57% of the cases in terms of the path length. The plots of lengths and hop-counts of paths generated by various algorithms are shown in Figure 4-17 and Figure 4-18, respectively.
Figure 4-17: Comparison of Path Lengths by SP, MCR and HR
CHAPTER 5

CONCLUSION AND EXTENSIONS

We presented a critical review of the existing algorithms for computing angle constrained paths. We characterized a class of triangulated networks which do not admit any $d$-monotone route between a given pair of nodes. We presented an efficient linear time centralized algorithm for constructing shortest $d$-monotone routes in any triangulated network. We also proposed two localized algorithms: (i) modified compass routing (MCR), and (ii) hybrid lateral/compass routing (HR) for constructing $d$-monotone routes in Delaunay networks. MCR algorithm is a modified version of the standard compass routing algorithm to ensure the $d$-monotone property. The basic idea of the HR algorithm is to improve upon the quality of the routes generated by the lateral span routing algorithm by adding the compass routing scheme.

We implemented several geometric routing algorithms that included the angle constrained shortest path algorithm, shortest $d$-monotone path algorithm, modified compass routing algorithm, and hybrid lateral/compass routing algorithm. Implementation is done in the Java Programming Language. The implemented prototype has a user-friendly graphical user interface (GUI) for planar graph construction, route computation and route display. It also offers an option to generate planar graphs from
input data files. Generated graphs can be saved in a data file for later usage. All planar graphs are represented in Doubly Connected Edge List (DCEL) data structure.

Performances of the proposed algorithms were evaluated by examining total lengths and the number of hops of the routes constructed by the corresponding algorithm on several (thirty-five) randomly generated Delaunay networks. An inspection of the experimental data shows that both MCR and HR generate good quality paths.

One might initially believe (as we did) that the shortest path between two given nodes in a DT is always monotone. For many examples, the shortest paths in DT are monotone. For some rare node distributions, the shortest path need not be monotone. Figure 5-1 shows a shortest path with a back-up edge in a DT.

![Figure 5-1: A Shortest Path with a Back-up Edge in a DT](image-url)
It would be interesting to characterize a class of Delaunay Triangulation networks which admit monotone shortest paths between all pairs of nodes. The Delaunay triangulation in Figure 5-1 can be slightly changed by using a diagonal-flip operation to get a new triangulation that makes shortest paths between all pairs of nodes monotone. So, an interesting question is: can every DT be transformed into a new triangulation $T'$ by diagonal-flip operations so that the shortest path between any pair of nodes in $T'$ is monotone?
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

Liang, Xiaojun

Local Address:
4250 S Arville St, Apt 204
Las Vegas, Nevada 89103

Home Address:
265 Huangpu Road Central, Room 905
Guangzhou, Guangdong, P.R.China 510655

Degree:
Master of Science, Hospitality Administration, 2001
University of Nevada, Las Vegas
Bachelor of Economics, Hospitality Administration, 1997
Hangzhou Institute of Commerce, China

Thesis Title: Angle Constrained Paths in Sensor Networks

Thesis Examination Committee:
Chair: Dr. Laxmi Gewali, Ph. D.
Committee Member: Dr. John Minor, Ph.D.
Committee Member: Dr. Wolfgang Bein, Ph.D
Graduate Faculty Representative: Dr. Emma Regentova, Ph. D.