Node Filtering and Face Routing for Sensor Network

Umang Amatya

University of Nevada, Las Vegas, umangama@hotmail.com
NODE FILTERING AND FACE ROUTING FOR SENSOR NETWORK

by

Umang Amatya

Bachelor of Computer Engineering, I.O.E., Pulchowk Campus
Tribhuvan University, Nepal
2007

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
August 2012
We recommend the thesis prepared under our supervision by

**Umang Amatya**

entitled

**Node Filtering and Face Routing for Sensor Network**

be accepted in partial fulfillment of the requirements for the degree of

**Master of Science in Computer Science**
Department of Computer Science

Laxmi P. Gewali, Committee Chair
John T. Minor, Committee Member
Juyeon Jo, Committee Member
Rama Venkat, Graduate College Representative

Thomas Piechota, Ph. D., Interim Vice President for Research and Graduate Studies
and Dean of the Graduate College

August 2012
ABSTRACT

Node Filtering and Face Routing for Sensor Network

by

Umang Amatya

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

Greedy forward routing and face routing algorithms have been extensively used for sending messages in sensor networks. In this thesis, we consider the problem of filtering redundant nodes in a sensor network as a pre-processing step for face routing. We propose two algorithms for identifying redundant nodes. We test the performance of proposed filtering algorithms on generated networks. The prototype algorithm for testing the proposed algorithms has been implemented in the Java programming language. Experimental investigation shows that the proposed filtering algorithms are effective in removing redundant nodes without compromising the network connectivity.
ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Laxmi P. Gewali for the continuous support of my thesis, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me a lot during my thesis. I feel that I am very lucky to have an advisor like him. I would also like to thank Dr. John T. Minor, Dr. Ju-Yeon Jo and Dr. Rama Venkat for serving as committee members.

I would also like to thank my friends: Bishal, Dinesh, Kishor dai, Krishna, Pinthep and Romas for their help and support.

Last but not least, I would like to thank my dad and my mom for their love and blessing.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2 REVIEW</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Gabriel graphs</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Algorithm to implement Gabriel graphs</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Relative neighborhood graph</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Algorithm to implement Relative neighborhood graph</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Delaunay Triangulation and Variations</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 Restricted Delaunay Graph(RDG)</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Routing</td>
<td>11</td>
</tr>
<tr>
<td>2.4.1 Greedy forward routing</td>
<td>12</td>
</tr>
<tr>
<td>2.4.2 Face routing</td>
<td>13</td>
</tr>
<tr>
<td>2.4.3 Hybrid greedy face routing</td>
<td>14</td>
</tr>
<tr>
<td>CHAPTER 3 NODE FILTERING FOR FACE ROUTING</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Compressing Equivalent Nodes</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Solo-Faced Chains</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Extracting Biconnected Clusters</td>
<td>24</td>
</tr>
<tr>
<td>3.3.1 Planar Graph</td>
<td>24</td>
</tr>
<tr>
<td>3.3.2 Planar Straight Line Graph</td>
<td>24</td>
</tr>
<tr>
<td>3.3.3 Doubly Connected Edge List (DCEL)</td>
<td>24</td>
</tr>
<tr>
<td>3.3.4 Degree of Vertices</td>
<td>26</td>
</tr>
<tr>
<td>3.3.5 Removal of Floating Chains</td>
<td>26</td>
</tr>
<tr>
<td>CHAPTER 4 IMPLEMENTATION</td>
<td>32</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td>2.1</td>
<td>Formation of Gabriel graph for five point sites</td>
</tr>
<tr>
<td>2.2</td>
<td>Formation of relative neighborhood graph of five point sites</td>
</tr>
<tr>
<td>2.3</td>
<td>Illustrating empty lune of influence test</td>
</tr>
<tr>
<td>2.4</td>
<td>Illustrating Delaunay Triangulation</td>
</tr>
<tr>
<td>2.5</td>
<td>Illustrating Empty Circle in Delaunay Triangulation</td>
</tr>
<tr>
<td>2.6</td>
<td>Illustrating formation of restricted delaunay graph</td>
</tr>
<tr>
<td>2.7</td>
<td>Illustrating Greedy routing</td>
</tr>
<tr>
<td>2.8</td>
<td>Illustrating Face Routing</td>
</tr>
<tr>
<td>2.9</td>
<td>Illustrating Hybrid greedy face routing</td>
</tr>
<tr>
<td>3.1</td>
<td>Illustrating Equivalent nodes $p_5$ and $p_6$</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustrating Clusters of Equivalent nodes</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustrating Gabriel Graph and its compressed version</td>
</tr>
<tr>
<td>3.4</td>
<td>Illustrating a Proof of Lemma 3.1.1</td>
</tr>
<tr>
<td>3.5</td>
<td>Gabriel graph showing edges and its equivalent half edge</td>
</tr>
<tr>
<td>3.6</td>
<td>Illustrating Solo-Faced Chains</td>
</tr>
<tr>
<td>3.7</td>
<td>Illustrating Bridge Solo-Faced Chain ($ch_3$-drawn dashed)</td>
</tr>
<tr>
<td>3.8</td>
<td>Illustrating clusters of nodes connected by Bridge Solo-Faced Chains</td>
</tr>
<tr>
<td>3.9</td>
<td>Tree representation of Bridge Solo-Faced Chains</td>
</tr>
<tr>
<td>3.10</td>
<td>Illustrating cluster of nodes without Non Bridge Solo-Faced Chains</td>
</tr>
<tr>
<td>3.11</td>
<td>Illustrating cluster of nodes after removing External-Components and External Solo-Faced Chains</td>
</tr>
<tr>
<td>3.12</td>
<td>Illustrating Planar Straight Line Graph (PSLG)</td>
</tr>
<tr>
<td>3.13</td>
<td>Illustrating Doubly Connected Edge List data structure</td>
</tr>
<tr>
<td>3.14</td>
<td>Graph showing edges and its equivalent half edge</td>
</tr>
<tr>
<td>3.15</td>
<td>Graph showing floating chains</td>
</tr>
<tr>
<td>3.16</td>
<td>Illustrating biconnected graph</td>
</tr>
<tr>
<td>3.17</td>
<td>Illustrating biconnected components and bridge chains</td>
</tr>
<tr>
<td>3.18</td>
<td>DCEL representation of biconnected components and bridge chains</td>
</tr>
<tr>
<td>3.19</td>
<td>Illustrating transition half-edge</td>
</tr>
<tr>
<td>4.1</td>
<td>Illustrating User Interface of Project 1</td>
</tr>
<tr>
<td>4.2</td>
<td>Illustrating Random Nodes</td>
</tr>
<tr>
<td>4.3</td>
<td>Illustrating Gabriel Graph containing source and target nodes</td>
</tr>
<tr>
<td>4.4</td>
<td>Illustrating the routing path taken from source node to target node</td>
</tr>
<tr>
<td>4.5</td>
<td>Illustrating the user interface of Project 2</td>
</tr>
<tr>
<td>4.6</td>
<td>Illustrating solo-faced chains</td>
</tr>
<tr>
<td>4.7</td>
<td>Illustrating graph containing external segments</td>
</tr>
<tr>
<td>4.8</td>
<td>Illustrating graph after removal of external segments</td>
</tr>
<tr>
<td>4.9</td>
<td>Illustrating equivalent nodes</td>
</tr>
</tbody>
</table>
Figure 4.10  Illustrating user interface of project 3 . . . . . . . . . . . . . . . 39
Figure 4.11  Illustrating greedy routing traced by nodes . . . . . . . . . . . . . 40
Figure 4.12  Illustrating face routing traced by nodes . . . . . . . . . . . . . . . 40
<table>
<thead>
<tr>
<th>Table 3.1</th>
<th>Record for each half-edge</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.2</td>
<td>Record for each face</td>
<td>26</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Record for each Vertex</td>
<td>26</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Nodes and their range</td>
<td>41</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Total changes in equivalent nodes</td>
<td>42</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Total changes in hubs (partially connected graph)</td>
<td>43</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Total changes in hubs (completely connected graph)</td>
<td>44</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Development of efficient algorithms and protocols for sensor network application has attracted the interest of many researchers for the last 10 years [6]. Sensor networks have been applied to solve problems in many areas of science and engineering that include robotics, emergency response, environmental monitoring, remote sensing, manufacturing, and law enforcement [16]. A sensor network is formed by wirelessly connecting sensor nodes which are distributed on a two dimensional surface or embedded in some equipment or gadgets. A sensor node is essentially a small electrical device containing (i) a small amount of memory, (ii) a low capacity processing unit, (iii) a radio communication component with range 300 meters, and (iv) some sensing components for measuring physical quantities such as temperature, pressure, humidity, etc. Nodes within the transmission range can exchange information wirelessly. Nodes outside the transmission range can communicate by establishing a sequence of in-range intermediate nodes between them. This kind of establishing in-range intermediate nodes between two distinct nodes is called routing.

Computation in a sensor network is much different than in a traditional wired network. In a sensor network, there is no centralized control for communication. Communication and computation is preferred to be done locally in a distributed manner. Each node only knows the position of itself and its in-range neighbors. A node can explore the presence of other nearby nodes by exchanging information between $k$-hop neighbors, where $k$ is usually 1, 2, or 3. Computing global properties of the sensor network that include connectivity, clustering, and coverage by exploring upto $k$-hop neighbors (for small $k$) are the most challenging problems in this emerging
area of computer science and engineering. Some good progress has been made for performing routing and clustering in recent years [9] and there are a wealth of fertile problems to pursue further research.

In this thesis, we are mainly concerned about filtering of nodes as a pre-processing step for routing in sensor networks. We basically try to identify redundant nodes present on a network so that we can eliminate such nodes and make the routing algorithms and protocols more effective and efficient.

Chapter two presents a review of planar graphs that includes Gabriel graph (gg), relative neighborhood graph (rng), Delaunay triangulation (dt), and restricted Delaunay graph (rdg). This chapter also contains a review of routing algorithms from computational geometry that includes greedy forward, face routing, and hybrid greedy face routing.

The main contributions of the thesis are in Chapter 3. It contains algorithms for filtering nodes as preprocessing for Face routing. We start with the formulation of the concepts “redundant nodes” and “equivalent nodes”. The proposed filtering algorithms are based on retaining only one member from a set of identified redundant nodes. We show how the removal of a node from a pair of equivalent nodes does not alter the connectivity and routing construction.

Chapter four describes the implementation of algorithms for generating planar networks and algorithms for routing that include greedy routing, face routing, and hybrid greedy face routing. These algorithms are implemented in Java programming language. The implementation has a front-end graphical user interface that makes it easy to execute algorithms for generating planar networks and for constructing routing. Chapter four also contains results of the experimental investigation of the performance of face routing algorithms after applying filtering. The results show that the quality of generated routes are preserved after filtering.

In chapter five, we describe the performance of proposed algorithms for several
input node distributions. We also suggest some problems for future research.
CHAPTER 2

REVIEW

In this chapter, we present a comprehensive review of networks that are used for communication and routing in sensor networks. Most networks reviewed in this chapter are locally computable, which include Gabriel graph (GG), relative neighborhood graph (RNG), restricted Delaunay graph (RDG), and Delaunay triangulation (DT).

2.1 Gabriel graphs

A Gabriel graph $GG(S)$ of a set of point sites $S = \{p_0, p_1, p_2, \ldots, p_{n-1}\}$ is a network structure first introduced by Gabriel and Sokal [8] in 1969 for analyzing geographic variation in zoology [10]. This structure is now extensively used for routing in sensor network and related applications. Gabriel graphs are used to model proximity (nearness) relations between nodes in two dimensional surfaces.

Formally, given a set of point sites $S = \{p_0, p_1, p_2, \ldots, p_{n-1}\}$, two point sites $p_i$ and $p_j$ are connected by an edge $e_{i,j}$ if the disk with diameter ending at $p_i$ and $p_j$ does not contain any other point sites. Figure 2.1 illustrates the formation of a Gabriel graph for five point sites. As shown in Figure 2.1a, the disk with diameter $<p_3,p_4>$ is empty and hence $p_3$ and $p_4$ is connected by an edge. On the other hand, the disk
with diameter \( <p_1, p_3> \) is not empty and hence they are not connected by an edge. The Gabriel graph constructed in this manner is shown in Figure 2.1b.

For applications in sensor networks, only those point sites \( p_i \) and \( p_j \) are considered for possible edge connection if the distance between them is less or equal to the wireless transmission range of nodes.

2.1.1 Algorithm to implement Gabriel graphs

Given a set of point sites \( S = \{p_0, p_1, p_2, \ldots, p_{n-1}\} \), perform empty circle tests for each pair of points by creating a circle with diameter \( p_i \) and \( p_j \) and checking if other points lie inside it or not. If the circle is empty, then connect those two points. The graph obtained in this manner is a Gabriel graph. Here, \( C_n^2 \) pair of circles has to be checked for emptiness with \( n \) other points. Hence, the total time complexity of this algorithm is \( O(n^3) \). This algorithm can be called a *Brute Force Algorithm*.

Since a Gabriel graph is a subset of Delaunay triangulation, we can construct Gabriel graphs using Delaunay triangulation. First of all, find Delaunay triangulation of point sites. This can be computed in \( O(n \log n) \) time. Then, perform empty circle tests on the edges of Delaunay triangle and remove the edges that contain other points inside them. Here the circle considered for empty test is the circle with the candidate Delaunay edge as the diameter. Since the edges of Delaunay triangle are of order \( n \) and we need to check for emptiness with \( n \) other points, the total time complexity of this algorithm is \( O(n^2) \). Such an algorithm can be called a Delaunay Guided Empty Circle algorithm.

There is another more efficient algorithm to compute Gabriel graphs where we don’t need to perform any empty circle tests [10], which can be named a Delaunay and Voronoi Guided Algorithm. We can use the concept of Delaunay triangulation and Voronoi diagram to construct it. Every edge of Delaunay triangle has its dual Voronoi edge. The edge belongs to the Gabriel graph if and only if the edge of
Delaunay triangle intersects its dual Voronoi edge. It takes $O(n \log n)$ time to compute both Delaunay triangulation and Voronoi diagram and since all edges are of order $n$, it takes $O(n)$ time to check for intersection. Thus the total time complexity of this algorithm is $O(n \log n)$.

The formal sketch of these three algorithms are given below:

Algorithm 1 Basic Algorithm (Brute Force)
1. $\textbf{for } i = 0 \text{ to } \text{TotalVertices} - 1 \text{ do}$
2. $\quad \textbf{for } j = 0 \text{ to } \text{TotalVertices} - 1 \text{ do}$
3. $\quad p_1 \leftarrow \text{Vertices}[i]$
4. $\quad p_2 \leftarrow \text{Vertices}[j]$
5. $\quad \textbf{if } \text{circle containing } p_1 \text{ and } p_2 \text{ as diameter is empty } \textbf{then}$
6. $\quad \quad \text{connect } p_1 \text{ and } p_2$
7. $\quad \textbf{end if}$
8. $\quad \textbf{end for}$
9. $\textbf{end for}$

Algorithm 2 (Delaunay Guided Empty Circle Algorithm)
1. Find delaunay triangulation induced by point sites in S
2. $\textbf{for } i = 0 \text{ to } \text{TotalDelaunayEdges} - 1 \text{ do}$
3. $\quad p_1 \leftarrow \text{DelaunayEdge[i]}.\text{source}$
4. $\quad p_2 \leftarrow \text{DelaunayEdge[i]}.\text{target}$
5. $\quad \textbf{if } \text{circle containing } p_1 \text{ and } p_2 \text{ as diameter is empty } \textbf{then}$
6. $\quad \quad \text{connect } p_1 \text{ and } p_2$
7. $\quad \textbf{end if}$
8. $\textbf{end for}$

Algorithm 3 (Delaunay and Voronoi Guided Algorithm)
1. Find voronoi diagram induced by point sites in S
2. Find delaunay triangulation induced by point sites in S
3. $\textbf{for } i = 0 \text{ to } \text{TotalDelaunayEdges} - 1 \text{ do}$
4. $\quad \textbf{if } \text{DelaunayEdges[i]} \text{ intersect } \text{DelaunayEdges[i]}.\text{dual} \textbf{then}$
5. $\quad \quad \text{print } \text{DelaunayEdges[i]}$
6. $\quad \textbf{end if}$
7. $\textbf{end for}$
2.2 Relative neighborhood graph

The relative neighborhood graph was proposed by Godfried Toussaint in 1980 [15] as a way of defining a structure from a set of points that would match human perceptions of the shape of the set. This graph is also used to model proximity (nearness) relations between nodes in two dimensional space.

![Diagram of relative neighborhood graph]

Figure 2.2: Formation of relative neighborhood graph of five point sites

Given a set of point sites $S = \{p_0, p_1, p_2, \ldots, p_{n-1}\}$, two point sites $p_i$ and $p_j$ are connected if and only if there does not exist a third point $p_k$ that is closer to both $p_i$ and $p_j$ than they are to each other. Figure 2.2 shows the relative neighborhood graph for the set of five point sites. Here, $p_1$ and $p_3$ are connected by an edge because there are no two edges $<p_1,p_k>$ and $<p_3,p_k>$ which are shorter than the edge $<p_1,p_3>$. On the other hand, point sites $p_1$ and $p_2$ are not connected because there are two edges $<p_1,p_3>$ and $<p_2,p_3>$ which are shorter than $<p_1,p_2>$.

For applications in sensor networks, we consider the transmission range of point sites and connect only those two point sites that are within transmission range.

An alternative definition of RNG can be given as follows. Two points $p_i$ and $p_j$ are defined as relatively close to each other if $d(p_i,p_j) \leq \max\{d(p_i,p_k), d(p_j,p_k)\}$ for all $k=1\ldots n$ and $k \neq i,j$. The graph thus obtained by connecting only points $p_i$ and $p_j$ that are relative neighbors is the relative neighborhood graph.

2.2.1 Algorithm to implement Relative neighborhood graph

Given a set of point sites $S = \{p_0, p_1, p_2, \ldots, p_{n-1}\}$, perform empty lune of influence tests for each pair of points by creating a circle with center at $p_i$ and $p_j$ and
radius equal to \(<p_i,p_j>\) and checking if other points lie inside the lune or not. If the lune is empty, then connect those two points. The graph obtained in this manner is a relative neighborhood graph. Here, \(C^n_2\) lunes have to be checked for emptiness with \(n\) other points. Hence, the total time complexity of this algorithm is \(O(n^3)\).

![empty lune and non-empty lune](image)

Figure 2.3: Illustrating empty lune of influence test

2.3 Delaunay Triangulation and Variations

Another planar graph that has been used for routing in sensor networks is the Delaunay triangulation. Delaunay triangulation was first proposed by Boris Delaunay in 1934 [1]. The Delaunay triangulation of a set of point sites \(\{p_0, p_1, p_2, \ldots, p_{n-1}\}\) is the triangulation such that no circumscribing circle of any triangle in the triangulation contains any other point site.

![Delaunay triangulation](image)

Figure 2.4: Illustrating Delaunay Triangulation

Figure 2.4 shows the Delaunay triangulation of 7 point sites. We can see that circumscribing circles of each triangle contain no point sites. Empty circle property is elaborately illustrated in Figure 2.5.
Figure 2.5: Illustrating Empty Circle in Delaunay Triangulation

Figure 2.5 demonstrates an empty circle test on two end points of an edge. In Figure 2.5(a), we can see that there is an empty circle through $p_1$ and $p_2$ and hence it is a Delaunay edge. On the other hand, any triangle through $p_3$ and $p_4$ can never be made empty and hence the edge connecting $p_3$ and $p_4$ is not a Delaunay edge.

Delaunay triangulation is closely related to the structure known as Voronoi Diagram [4]. In fact it is known that Delaunay triangulation is the dual of Voronoi diagram. Many algorithms have been reported to compute Delaunay triangulation [1]. Since Voronoi diagram is the dual of Delaunay triangulation, algorithms for computing Voronoi diagram can be used for obtaining Delaunay triangulation and vice versa [12]. The most popular algorithm for computing Delaunay triangulation is the sweepline algorithm proposed by Fortune [7] which runs in $O(n \log n)$ time. It is remarked that the problem of computing Delaunay triangulation has a lower bound of $\Omega(n \log n)$ in the comparison tree model of computation [3]. Hence, Fortunes algorithm is also the optimum algorithm. In the context of routing in sensor networks, locally computable structures are highly suitable. Unfortunately Delaunay triangulation can not be computed locally. However, a super-set of Delaunay triangulation briefly described in the next subsection can be computed locally.
2.3.1 Restricted Delaunay Graph (RDG)

Consider a set of point sites $S=\{p_0, p_1, p_2, \ldots, p_{n-1}\}$ in the plane. Each point site corresponds to the position of a sensor node with radio transmission range $r=1$. (All sensor nodes have identical transmission range.) Suppose each node $p_i$ computes the Delaunay triangulation of $p_i$ and its one hop neighbor. Let $T(p_i)$ denote the Delaunay triangulation of $p_i$ and its one hop neighbors $N(p_i)$. The triangulation $T(p_i)$ is called the local Delaunay triangulation of $p_i$. The network obtained by the union of all $T(p_i)$’s is not necessarily planar and it may not be even a triangulation graph. Let $G_u$ denote the union of all $T(p_i)$ for all $p_i$ in $S$. The graph $G_u$ is not necessarily planar and may not contain all Global Delaunay edges of $S$. A method of extracting a planar graph from $G_u$ with 1-hop information exchange was proposed by Gao et al. [9]. Such a graph is called Restricted Delaunay Graph (RDG).

A technique to obtain RDG $G_r$ from $G_u$ is to remove selected crossing edges (sr edges) from $G_u$ [9]. The approach is to check the consistency of edges in the Delaunay triangulation of two neighbor nodes. To understand this method, consider four nodes $p_1, p_2, p_3, p_4$ as shown in Figure 2.6, where the single triangle is the Delaunay triangulation $T(p_1)$ of 1-hop neighbors $N(p_1)$ and $p_1$ itself. On the other hand, the Delaunay triangulation $T(p_2)$ of $N(p_2)$ and $p_2$ is shown in Figure 2.6b. The overlay of $T(p_1)$ and $T(p_2)$ is shown in Figure 2.6c, where two edges cross making $T(p_1) \cup T(p_2)$ non-planar.

Among the crossing pair of edges $p_1p_3$ and $p_2p_4$, we need to remove one to make $T(p_1) \cup T(p_2)$ planar. The one that is not present in both is removed. The sr edge so identified is not a global Delaunay edge. After removing sr-edge, we obtain the RDG $G_r$ of four nodes shown in Figure 2.6d. Although the RDG shown in Figure 2.6 is a triangulation graph, for large number of nodes, it need not be a triangulation. Furthermore $G_r$ may contain some edges that are not present in global Delaunay triangulation.
A formal local algorithm to construct RDG $G_r$ can be described as follows. Each node $p_i$ determines the Delaunay triangulation $T(p_i)$ of $p_i$ and its 1-hop neighbors $N(p_i)$. Each neighbor $p_i$ and $p_j$ exchange $T(p_i)$ and $T(p_j)$. If an edge in $T(p_i)$ is not valid for $T(p_j)$, then that edge is deleted. This local method of removing inconsistency is sketched as Algorithm 4.

**Algorithm 4** Algorithm for resolving inconsistency

1: $E(u) := uv \mid uv \in T(u)$
2: for each edge $uv$ in $E(u)$ do
3:     for each edge $w$ in $N(u)$ do
4:         if $(u,v \in N(w)$ and $uv \not\in T(w))$ then
5:             delete $uv$ from $E(u)$
6:         end if
7:     end for
8: end for

2.4 Routing

Routing is the process of selecting a path in a network for sending information from source node to destination node along the network [13]. Routes are constructed in sensor networks using appropriate planar graphs that include Gabriel graphs, relative neighborhood graphs, and restricted Delaunay graphs. In general, if a source node $s$ wants to send a message to a destination node $t$ which is
outside the range of $s$, then $s$ needs to send the message through a sequence of relay nodes. This sequence of relay nodes together with source and destination nodes define a route connecting $s$ to $t$. The process of generating these routes is called *routing*. Some of the well known route construction techniques are greedy forward, face routing and hybrid greedy face routing.

### 2.4.1 Greedy forward routing

Greedy forward routing is a very simple yet one of the most powerful routing algorithms. It constructs route locally in a sequence of steps. In greedy routing, all of the adjacent nodes that are within transmission range are first detected. The next node to forward the message is selected from among the adjacent nodes which are nearer to the target node than the current node. The node that is closest to the target node gets the message. This process is repeated until target is reached.

![Figure 2.7: Illustrating Greedy routing](image)

Figure 2.7: Illustrating Greedy routing

We can illustrate it with a simple example as shown in Figure 2.7(a). Let $S=\{s, p_1, p_2, p_3, p_4, t\}$ be the set of nodes where $s$ is the source node and $t$ is the target node. The source node $s$ has three adjacent nodes $p_4, p_1$ and $p_2$. Among those three nodes, node $p_4$ is selected because the distance between $<p_4,t>$ is smaller than $<p_1,t>$ and $<p_2,t>$. Similarly, node $p_4$ has 4 nodes $s, p_1, p_3$ and $t$ as adjacent nodes. Since it is directly connected with $t$, we have found the routing path $s \rightarrow p_4 \rightarrow t$ using greedy forward routing algorithm.

Sometimes, a message gets stuck while using greedy forward routing algorithm. This occurs when the node itself becomes the shortest node rather than its adjacent
nodes. In Figure 2.7(b), we can see that node $s$ finds $p_4$ as the next node. Similarly, node $p_4$ finds node $p_3$ as its next node. Node $p_3$ has $p_4$, $p_1$, and $p_2$ as its adjacent nodes but none of the nodes have shorter distance than $<p_3,t>$. Hence the message gets stuck at a local minimum.

### 2.4.2 Face routing

A message delivery method which is guaranteed to work if there is some path connecting source node $s$ to target node $t$ is based on the traversal of the faces of the planar graph formed on the underlying sensor network. A path construction algorithm based on this approach is known as face routing [13]. We can give a brief description of face routing algorithm by considering an example planar graph constructed on the sensor network. The planar graph could be either a Gabriel Graph (GG), or a Relative Neighborhood Graph (RNG) induced by the sensor nodes. In Figure 2.8, a Gabriel graph of eighteen nodes $S=\{s, p_1, p_2, \ldots, p_{15}, p_{16}, t\}$ is shown where there are nine faces that includes the outer unbounded face. Node $s$ and node $t$ are the source node and target node, respectively.

![Figure 2.8: Illustrating Face Routing](image)

The source node $s$ knows the position of itself, its 1-hop neighbors, and the position of the target node $t$. If $t$ is within the transmission range of $s$, then the message is delivered directly. Otherwise, the algorithm construct the correct face $f_c$ of the Gabriel graph such that (i) $s$ is a node of $f_c$ and (ii) $f_c$ is intersected by the guiding line segment $e_g=(s,t)$. The algorithm determines the transition edge $e_w$ of $f_c$. Transition
edge $e_w$ of $f_c$ is the line segment of $f_c$ that is intersected by the guiding segment $e_g$. In Figure 2.8, $(p_8,p_{10})$ is the transition edge. After constructing the current face $f_c$, the message traverses counterclockwise along the edges of $f_c$ and stops at the transition edge. The message is then delivered to the other face ($f_2$ in Figure 2.8) incident on the transition edge.

In the next iteration, one of the nodes of the transition edge becomes the source node and the other face incident on $e_w$ becomes the current face $f_c$. This process of traversing the faces is continued until the target node $t$ is discovered to deliver the message. In Figure 2.8, the constructed route is shown by directed segments.

**Algorithm 5** Algorithm for face routing

1: $p \leftarrow s$
2: repeat
3: let $f$ be the face of $G$ with $p$ on its boundary that intersects $(p,t)$
4: traverse $f$ until reaching an edge $(u,v)$ that intersects $(p,t)$ at some point $p' \neq p$
5: $p \leftarrow p'$
6: until $p = t$

2.4.3 Hybrid greedy face routing

This is a combination of greedy forward routing and face routing. Greedy forward is used as much as possible and when a stuck node is encountered, face routing is used to escape from the stuck node.

![Figure 2.9: Illustrating Hybrid greedy face routing](image)

In Figure 2.9, greedy forward algorithm is used until node gets stuck at node...
$p_7$. Face routing is used to rescue the $p_7$ node and reach the target.
CHAPTER 3

NODE FILTERING FOR FACE ROUTING

In this chapter, we consider the problem of removing redundant or pseudo-redundant nodes from a set of given nodes. We call this process node filtering.

3.1 Compressing Equivalent Nodes

Consider a set of nodes $S = \{p_0, p_1, p_2, \ldots, p_{n-1}\}$ used for face routing in a sensor network. Two nodes close to each other are called equivalent if their transmission ranges cover the same sub-set of nodes. Recall that all nodes are assumed to have an identical transmission range which is taken, without loss of generality, as 1. We can illustrate the notion of equivalent nodes with a specific example. In Figure 3.1, the transmission disks of two nodes $p_5$ and $p_6$ are shown with dashed circle. This shows that nodes $p_5$ and $p_6$ cover the identical set of nodes $p_1, p_2, p_3, p_4, p_7, p_8$.

On the right-side of Figure 3.1, two nodes $p_{10}$ and $p_{11}$ are shown which are not equivalent even though they are very close to each other. There can be many nodes equivalent to each other in some rare distributions that contain clustered nodes in some pocket region. This is illustrated in Figure 3.2.

![Figure 3.1: Illustrating Equivalent nodes $p_5$ and $p_6$](image-url)

![Figure 3.2: Illustrating another set of equivalent nodes](image-url)
Figure 3.2a is a distribution of sensor nodes where there are five distinctly visible cluster sub-sets of nodes. The clusters that cover identical nodes are highlighted by drawing circles for their range in Figure 3.2b. It is observed in Figure 3.2b that circles in a group cover the same set of nodes.

**Definition 3.1.1.** (Compressed Gabriel Graph) Consider a Gabriel Graph $G(V,E)$ of a set of sensor nodes. Let $C_1,C_2,...,C_k$ be the set of equivalent nodes in $G(V,E)$. The nodes in $V$ not in the equivalent sets are referred to as *background nodes* and the set of these nodes is denoted by $V_B$. The set of nodes obtained by adding to $V_B$ exactly one member from each equivalent set is the compressed set of nodes, $V_C$. The resulting Gabriel graph of $V_C$, denoted by $G_C(V_C,E_C)$, is the *compressed Gabriel graph*. Figure 3.3 shows the original Gabriel graph and its compressed version for indicated transmission range.
Lemma 3.1.1. If background nodes \( v_i \) and \( v_k \) are connected in Gabriel graph \( G(V,E) \), then they are also connected in the compressed Gabriel graph \( G_C(V_C,E_C) \).

**Proof:** Consider any route \( R \) connecting node \( v_i \) to \( v_k \) that passes through a cluster \( C_j \). Let \( v_e \) and \( v_t \) be the nodes in the background and in \( R \) that are closest to the cluster \( C_j \) (Figure 3.4).

Let \( v_j \) be the representative node in \( C_j \). Since \( v_e \) and \( v_t \) are connected to some nodes in \( C_j \), they are also connected to \( v_j \) by the “special path” \((v_e,v_j,v_t)\) shown by dashed segments.
3.2 Solo-Faced Chains

A segment can be viewed to consist of two half-edges. A Gabriel graph in which edges are split into two half-edges is shown in Figure 3.5b. Let \( \{e_1, e_2, \ldots, e_{15}\} \) be the edges of a Gabriel graph. The half-edges of \( e_i \) are denoted by \( e_i' \) and \( e_i'' \). Note that \( e_i' \) and \( e_i'' \) are twin half-edges of each other and they are directed reverse to each other.

![Figure 3.5: Gabriel graph showing edges and its equivalent half edge](image)

**Definition 3.2.1.** An edge \( e_i \) is called a solo-faced edge if the twin half-edges of \( e_i \) i.e. \( e_i' \) and \( e_i'' \) are incident on the same face. In Figure 3.5a, there are three faces \( f_0, f_1, \) and \( f_2 \) in the Gabriel graph. Edge \( e_{15} \) is a solo-faced edge since \( e_{15}' \) and \( e_{15}'' \) are incident on the same face \( f_1 \). Similarly \( e_5 \) is a solo-faced edge as \( e_5' \) and \( e_5'' \) are incident on the same face \( f_0 \). On the other hand, \( e_4 \) is not solo-faced as \( e_4' \) and \( e_4'' \) are incident on faces \( f_0 \) and \( f_1 \), respectively.

**Definition 3.2.2.** (Maximal Solo-Faced Chain) A sequence of consecutive solo-faced edges is referred to as a solo-faced chain. A solo-faced chain that is not contained in any other solo-faced chain is called a maximal solo-faced chain. In Figure 3.5 there are two maximal solo-faced chain which are \( e_7e_6e_5 \) and \( e_{15}e_{14} \). Some Gabriel graphs could have many maximal solo-faced chains. A Gabriel graph with five maximal solo-faced chains is shown in Figure 3.6.
Remark 3.2.1. From now onward, unless otherwise stated, the term \textit{solo-faced chain} will be used to indicate a \textit{maximal solo-faced chain}.

\textbf{Definition 3.2.3.} (Bridge Solo-Faced Chain) Solo-faced chains can be distinguished into two types: Bridge type and Non-Bridge type. If the removal of a solo-faced chain $ch_1$ from the Gabriel graph breaks the graph into two connected components then $ch_1$ is called a \textit{bridge solo-faced chain}. In Figure 3.7, there are seven solo-faced chains. Among them only one is a bridge solo-faced chain which is drawn by dashed edges.

\textbf{Definition 3.2.4.} (External Solo-Faced Chains and External Components) Figure 3.8 shows five clusters of nodes A,B,C,D and E. These clusters contain many interconnected nodes and solo-faced chains. Let $ch_1,ch_2,ch_3$ and $ch_4$ be the four bridge solo-faced chains connecting these five clusters of nodes. These five clusters of nodes and their bridge solo-faced chains can also be represented in a tree structure as shown in Figure 3.9.

Let $s$ and $t$ be the source and target node present inside clusters A and E respectively. From Figure 3.9, we can see that the actual path to travel from source node to target node is A $\rightarrow$ $ch_3$ $\rightarrow$ B $\rightarrow$ $ch_1$ $\rightarrow$ E. Only $ch_3$ and $ch_1$ are the two bridge solo-faced chains that lie in the path from A to E. All other bridge solo-faced
chains that don’t lie in the path are called external solo-faced chains. There are also clusters of nodes that are not present in the path. These type of cluster of nodes are called external components. In the given figure, two clusters $<\text{C,D}>$ and two bridge solo-faced chains $<\text{ch}_2,\text{ch}_4>$ are called external components and external solo-faced chains respectively.

Figure 3.10 shows the cluster of nodes when non bridge solo-faced chains of Figure 3.8 are removed from the graph. This removal of chains helps in removing unnecessary edges from the graph. The graph can be further optimized by removing external components and external solo-faced chains as shown in Figure 3.11.
Figure 3.8: Illustrating clusters of nodes connected by Bridge Solo-Faced Chains

Figure 3.9: Tree representation of Bridge Solo-Faced Chains
Figure 3.10: Illustrating cluster of nodes without Non Bridge Solo-Faced Chains

Figure 3.11: Illustrating cluster of nodes after removing External-Components and External Solo-Faced Chains
3.3 Extracting Biconnected Clusters

We could use the depth-first search based on connected component recognition algorithm [14] to extract biconnected clusters. However, a Gabriel graph is a planar graph and a much simpler algorithm can be designed to extract biconnected components. To describe such a simpler algorithm, it is necessary to use the doubly connected edge list (dcel) data structure for representing planar graphs which is described next.

3.3.1 Planar Graph

A graph $G(V,E)$ is called planar if it can be drawn on a plane without intersecting edges.

3.3.2 Planar Straight Line Graph

A planar straight line graph is a planar graph containing all straight edges. They are mostly used to represent maps. Figure 3.12 shows a planar straight line graph containing five vertices and five edges.

![Figure 3.12: Illustrating Planar Straight Line Graph (PSLG)](image)

3.3.3 Doubly Connected Edge List (DCEL)

Double Connected Edge List is a data structure for representing planar straight line graphs. It was originally suggested by Preparata and Muller in 1978 for the representation of 3D convex polyhedra [11]. In DCEL, each edge is viewed as a
pair of twin half-edges pointing in opposite direction. Figure 3.13 shows a doubly connected edge list data structure containing five vertices, ten half edges and two faces.

![Figure 3.13: Illustrating Doubly Connected Edge List data structure](image)

A DCEL consists of a record for each half edge, face, and vertex. Each half edge record consists of a twin edge, a previous edge, a next edge, an incident face, and a source vertex. The face record consists of a bounding half edge, and each vertex record consists of an incident edge, and its co-ordinate. An example illustrating all the records for each half edge, face and vertex of Figure 3.13 is shown in Table 3.1, 3.2 and 3.3 respectively.

<table>
<thead>
<tr>
<th>Half Edge</th>
<th>Twin</th>
<th>Prev</th>
<th>Next</th>
<th>Incident Face</th>
<th>Source Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$</td>
<td>$e_2$</td>
<td>$e_5$</td>
<td>$e_9$</td>
<td>$f_0$</td>
<td>$v_2$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$e_1$</td>
<td>$e_7$</td>
<td>$e_3$</td>
<td>$f_1$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$e_3$</td>
<td>$e_5$</td>
<td>$e_2$</td>
<td>$e_4$</td>
<td>$f_1$</td>
<td>$v_2$</td>
</tr>
<tr>
<td>$e_4$</td>
<td>$e_6$</td>
<td>$e_3$</td>
<td>$e_7$</td>
<td>$f_1$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$e_5$</td>
<td>$e_3$</td>
<td>$e_6$</td>
<td>$e_1$</td>
<td>$f_0$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$e_6$</td>
<td>$e_4$</td>
<td>$e_8$</td>
<td>$e_5$</td>
<td>$f_0$</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$e_7$</td>
<td>$e_8$</td>
<td>$e_4$</td>
<td>$e_2$</td>
<td>$f_1$</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$e_8$</td>
<td>$e_7$</td>
<td>$e_{10}$</td>
<td>$e_6$</td>
<td>$f_0$</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$e_9$</td>
<td>$e_{10}$</td>
<td>$e_{10}$</td>
<td>$f_0$</td>
<td>$v_3$</td>
<td></td>
</tr>
<tr>
<td>$e_{10}$</td>
<td>$e_9$</td>
<td>$e_9$</td>
<td>$e_8$</td>
<td>$f_0$</td>
<td>$v_5$</td>
</tr>
</tbody>
</table>

Table 3.1: Record for each half-edge
<table>
<thead>
<tr>
<th>Face</th>
<th>Bounding Half Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_0</td>
<td>e_9</td>
</tr>
<tr>
<td>f_1</td>
<td>e_7</td>
</tr>
</tbody>
</table>

Table 3.2: Record for each face

<table>
<thead>
<tr>
<th>Node</th>
<th>Incident Edge</th>
<th>Co-ordinate</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_1</td>
<td>e_5</td>
<td>x_1, y_1</td>
</tr>
<tr>
<td>v_2</td>
<td>e_3</td>
<td>x_2, y_2</td>
</tr>
<tr>
<td>v_3</td>
<td>e_2</td>
<td>x_3, y_3</td>
</tr>
<tr>
<td>v_4</td>
<td>e_7</td>
<td>x_4, y_4</td>
</tr>
<tr>
<td>v_5</td>
<td>e_10</td>
<td>x_5, y_5</td>
</tr>
</tbody>
</table>

Table 3.3: Record for each Vertex

3.3.4 Degree of Vertices

The degree of vertex v is defined as the total number of edges incident to that vertex. Let \( V = \{ v_1, v_2, v_3, v_4, v_5, v_6 \} \) be the total vertices and \( E = \{ e_1, e_2, e_3, e_4, e_5 \} \) be the total edges present in a graph \( G(V, E) \) as shown in Figure 3.14(a). Vertex \( v_3 \) has degree three because three edges are incident to it. Similarly, vertex \( v_6 \) and \( v_5 \) have degree one and two respectively. Figure 3.14(b) shows the edges split into two half edges for representing it in a Doubly Connected Edge List data structure. An algorithm to detect vertices of degree one is described in algorithm 6.

**Algorithm 6** Detect vertices of degree one

1: if \( v_i \).getIncidentEdge().getTwin().getNext()==\( v_i \).getIncidentEdge() then
2: return true.
3: else
4: return false.
5: end if

3.3.5 Removal of Floating Chains

Floating chains have at least one vertex with degree one. Figure 3.15 shows a floating chain \( e_3, e_4, e_5 \). In this chain, vertex \( v_6 \) has degree one and hence this node
is removed from the graph. After the removal of vertex \( v_6 \) and edge \( e_5 \), vertex \( v_5 \) becomes another node with degree one. These nodes are removed from the graph one after another until we don’t find any vertex with degree one. We also need to consider if the vertex node is the source or target because these nodes shouldn’t be removed from the graph. The concept of doubly connected edge list data structure is used to remove floating chains from the graph as shown in Figure 3.15(b).

**Algorithm 7 Removing Floating chain**

1: Vertex \( v_1, v_c \)
2: HalfEdge \( e_1, e_2 \)
3: \( v_c \leftarrow v_i \)
4: while \( v_c \neq s \ or \ t \ and \ deg(v_c) == 1 \) do
5: \( v_1 \leftarrow v_c \)
6: \( e_1 \leftarrow v_1.\text{getIncidentHalfEdge}() \)
7: \( e_2 \leftarrow e_1.\text{getTwin}() \)
8: \( v_c \leftarrow e_2.\text{getStartNode}() \)
9: \( e_2.\text{getPrev}().\text{setNext}(e_1.\text{getNext}()) \)
10: \( v_c.\text{setIncidentHalfEdge}(e_1.\text{getNext}()) \)
11: remove \( v_1, e_1, e_2 \) from the graph
12: end while
We assume that the given Gabriel graph $G(V,E)$ contains no floating chain. If there are floating chains, then we can use Algorithm 7 in Section 3.3.5 to remove them.

A graph $G(V,E)$ is called biconnected if there are two distinct paths between any pair of vertices in $G(V,E)$. A biconnected graph remains connected even if some of its vertices are removed. Figure 3.16(a) is a biconnected graph because any two pair of vertices have at least two distinct paths. Figure 3.16(b) is not a biconnected graph because vertex $v_1$ and $v_4$ doesn’t have two distinct paths.

In a connected Gabriel graph $G(V,E)$ without floating chains, there could be many biconnected components and bridge chains as shown in Figure 3.17. A dcel representation of $G(V,E)$ is shown in Figure 3.18.
Figure 3.16: Illustrating biconnected graph

Figure 3.17: Illustrating biconnected components and bridge chains

Figure 3.18: DCEL representation of biconnected components and bridge chains
Definition 3.3.1. A half-edge \( e_i \) is called a *transition half-edge* if it satisfies the following conditions.

i) Half-edge \( e_i \) is in a bridge-chain.

ii) Let \( e_j = e_i.next.twin \), then \( e_j.face \) is not same as \( e_i.face \) or \( e_j.face \) is not main face (outer unbounded face)

![Figure 3.19: Illustrating transition half-edge](image)

Algorithm 8 describes an algorithm to determine transition half edges. This transition half edge algorithm is used for finding bi-connected graphs in Algorithm 9. The three half-edges \( e_1, e_2, \) and \( e_c \) of Algorithm 9 is shown in Figure 3.19(b). Algorithm 10 describes the procedure to remove external components and external solo-faced chains from the graph.

**Algorithm 8** Find all Transition Half Edge

1: for each edge \( e_i \) in bridge solo-faced chain do
2:   if \( e_i.getNext().getTwin().hasMainFace() == false \) then
3:     add \( e_i \) to transition edge list
4:   end if
5: end for
Algorithm 9  Find Bi-Connected graph from transition edge

1: transition edge $e_i$
2: half edge $e_1, e_2, e_c$
3: $e_1 \leftarrow e_i$
4: $e_2 \leftarrow e_1.getNext()$
5: if $e_2.getNext().getTwin().hasMainFace() == true$ then
6:   $e_c = e_2.getNext().getTwin().getNext()$
7: else
8:   $e_c = e_2.getNext()$
9: end if
10: Edgelist $e_l$
11: $e_l.add(e_2)$
12: while $e_c \neq e_2$ do
13:   $e_l.add(e_c)$
14:   if $e_c.getNext().getTwin().hasMainFace() == true$ then
15:      $e_c = e_c.getNext().getTwin().getNext()$
16:   else
17:      $e_c = e_c.getNext()$
18: end if
19: end while
20: create a polygon $p$ using all the edges of edge list $e_l$
21: for each edge $e_1_i$ present in graph do
22:   if $e_1_i$ is present inside $p$ then
23:      $e_l.add(e_1_i)$
24: end if
25: end for
26: output $e_l$ as total edges of one bi-connected graph obtained from transition edge $e_i$

Algorithm 10  Remove External Solo-Faced Chains and External Components from graph

1: label each edge of a bi-connected graph by a tree.
2: use transition half-edge to find connection between two trees
3: find the tree present in source and target
4: use depth first search to find the path from source to target.
5: delete all bridge solo-faced chains that don’t lie in the path
6: delete all vertices and edges that don’t have trees that lie in the path.
CHAPTER 4

IMPLEMENTATION

This chapter describes the implementation of node filtering and face routing in sensor networks. All the programs are implemented in Java. The program is divided into three projects.

The first project shows the implementation of Gabriel Graph, Relative Neighborhood Graph and Unit Disk Graph. These graphs are implemented for some randomly generated nodes. It also shows how a node routes a path from source node to target node using Greedy Routing, Face Routing and Hybrid Routing.

The second project deals with filtering of nodes in a network. This project mainly focuses on equivalent nodes, solo-faced chains and external components. It shows the difference in a network before and after removal of equivalent nodes. It also shows how the network is changed when nodes present on solo-faced chains and external components are removed from the graph.

The third project is the combination of first and second project. This project shows how the node routes a path from source node to target node using different routing algorithms and the difference found when equivalent nodes, solo-faced chains and external components are removed from the graph.

4.1 Introduction to Java

Java is a programming language developed by James Gosling [2] at Sun Microsystems. It is a high-level object-oriented programming language. Java derived much of its syntax from C and C++. Its applications are typically compiled to bytecode that can run on any Java Virtual Machine (JVM) regardless of computer architecture. It is intended to let application developers write once, run anywhere.
(WORA) [5], meaning that code that runs on one platform does not need to be recompiled to run on another.

4.2 Interface Description

4.2.1 Project 1

Figure 4.1 shows the main user interface of project 1. It is drawn by extending the JFrame class. It consists of a menu bar, a drawing panel, a right panel and a bottom panel.

![Figure 4.1: Illustrating User Interface of Project 1](image)

The menu bar consists of a File Menu which contains 4 menu items: Open, Save, Export and Exit. The Open Menu Item is used for opening a saved file and Save Menu Item is used for saving the graph. Similarly, Export Menu Item is used to export the file to XFig format and Exit Menu Item helps to exit the program.

A big center panel is the drawing panel in which all nodes are drawn. It shows all the graphic output of the program.

A right panel consists of eight checkboxes. The first checkbox is DrawNode. This checkbox helps in drawing nodes on the drawing panel. A user can check this checkbox
and draw nodes on the drawing panel using mouse. *EditNode* and *DeleteNode* checkboxes are used for editing and deleting nodes respectively. If we select the *SelectSource* checkbox and select any node, the node becomes the source. Similarly, we can select the Target node by selecting the *TargetNode* check box. The next check box is *Compare* which helps to compare the graphs. If this check box is selected and we try to draw a Gabriel Graph, it opens a new Frame containing Gabriel Graph and Unit Disk Graph. *Steps* checkbox helps in displaying steps of routing. The last checkbox is *UseFaceRouting* which helps on using the face routing when nodes get stuck during greedy routing.

A right panel also contains four buttons *Random*, *Clear*, *Overall*, and *NextStep*. *Random* button helps in displaying random nodes. These nodes can be cleared by using *Clear* button. *Overall* Button helps in comparing the path travelled by Greedy and Face Routing on RNG and GG. *NextStep* button is used to display the next step in routing. There is also a range slider to change the range of sensor nodes.

Finally, there is a bottom panel containing seven buttons. *DisplayRNG*, *DisplayGG* and *DisplayUDG* buttons are used to display Relative neighborhood
graph, Gabriel graph and Unit disk graph respectively. ClearCanvas button is used to clear the graph. FaceRouting, Greedy and HybridRouting are the three routing buttons to demonstrate the routing operations.

Figure 4.2 shows a few red random nodes drawn on the user interface. It is drawn by clicking Random button. We can also add nodes to it by selecting DrawNode checkbox. The Gabriel Graph of the above nodes is shown in Figure 4.3. It also shows the source and target nodes. The greedy routing path taken from source node to target node is illustrated in Figure 4.4.

![Figure 4.3: Illustrating Gabriel Graph containing source and target nodes](image)

4.2.2 Project 2

The user interface of project 2 is similar to project 1. Figure 4.5 illustrates the user interface of Project 2. It consists of menu bar, drawing panel, range panel and checkboxes. The menu bar consists of a File Menu which contains 3 menu items Open, Save and Exit which are similar to Project 1.

The big white panel is a drawing panel where nodes are drawn. The bottom panel contains a range panel which helps in changing the range of sensor nodes.
Figure 4.4: Illustrating the routing path taken from source node to target node

The right panel consists of many checkboxes. There are five checkboxes present at the top of right node. They are DrawNode, EditNode, DeleteNode, DisplayGG and Displayrange. The top three checkboxes are used for drawing, editing and deleting nodes. The DisplayGG check box helps in displaying the Gabriel graph of nodes. The Display range checkbox helps in displaying the range of sensor nodes.

Figure 4.5: Illustrating the user interface of Project 2

There are two tabs, Segment and EqNode. The Segment tab contains five
checkboxes. The first two checkboxes are used to select source and target. The third and fourth checkboxes are used to remove and show solo-faced segments respectively. The last checkbox is used to remove external segments present on the graph. The graph showing solo-faced segments is illustrated in Figure 4.6. Figure 4.8 shows a graph when external segments are removed from Figure 4.7.

Figure 4.6: Illustrating solo-faced chains

Figure 4.7: Illustrating graph containing external segments
Figure 4.8: Illustrating graph after removal of external segments

EqNode tab contains 3 checkboxes. The first checkbox is used to show only one equivalent node among all equivalent nodes. The second checkbox displays only one equivalent node picked randomly. The third checkbox is used to display equivalent nodes. A graph illustrating equivalent nodes is shown in Figure 4.9. There are also two buttons, Random and Clear which generates and clears all nodes respectively.

Figure 4.9: Illustrating equivalent nodes
4.2.3 Project 3

The user interface of project 3 consists of a menu bar, drawing panel and right panel. The menu bar consists of three menu items, *open*, *save* and *exit*. The drawing panel is used for drawing the graph.

![User Interface of Project 3](image)

**Figure 4.10: Illustrating user interface of project 3**

The right panel consists of 15 checkboxes. The first three checkboxes, *DrawNode*, *EditNode* and *DeleteNode* are used to draw, edit and delete nodes present on graph. *DisplayGG* checkbox helps in displaying the Gabriel Graph of nodes. *DisplayRange* checkbox is used to display range of each sensor nodes.

There are 2 other checkboxes, *SelectSource* and *SelectTarget*, to select source and target nodes. *DisplayEq.Nodes* and *Only1Eq.Node* checkboxes are used to display equivalent nodes and only one equivalent nodes respectively. The *rand* checkbox is used for selecting only one equivalent node in random from clusters of equivalent nodes.

The *ExternalSegment* checkbox is used to show only external segments obtained after removing equivalent nodes, solo-faced chains and external components.

Finally, there are three routing checkboxes, *GreedyRouting*, *FaceRouting* and
HybridRouting, to show how the path is traced when these routing algorithms are used.

In Figure 4.11, we can see the path traced when we apply the greedy routing algorithm. We can also see that the node got stuck while using greedy routing.
Figure 4.12 shows the path traced when we applied the face routing algorithm.

The rightmost bottom textboxes are used for getting information from the graph. It provides information regarding total nodes, total equivalent nodes and total equivalent groups. It also provides information regarding total faces and total hubs as illustrated in Figure 4.12.

4.3 Observation

Table 4.1a and 4.1b shows the random nodes and their respective ranges taken for the experiment. Table 4.1a shows the total nodes and their ranges when they are partially connected and Table 4.1b shows the total nodes when they are completely connected.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>104</td>
</tr>
<tr>
<td>300</td>
<td>87</td>
</tr>
<tr>
<td>400</td>
<td>75</td>
</tr>
<tr>
<td>500</td>
<td>66</td>
</tr>
<tr>
<td>700</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>238</td>
</tr>
<tr>
<td>300</td>
<td>192</td>
</tr>
<tr>
<td>400</td>
<td>156</td>
</tr>
<tr>
<td>500</td>
<td>133</td>
</tr>
<tr>
<td>600</td>
<td>116</td>
</tr>
</tbody>
</table>

Table 4.1: Nodes and their range

We drew Gabriel graphs with the above nodes and tested them in different ranges. We found out that the equivalent nodes varied with the range of nodes. Table 4.2a and 4.2b show the results obtained when the graphs were partially and completely connected respectively. The equivalent nodes got reduced by 12.28% when nodes were partially connected. On the other hand, the nodes got reduced by 5.17% when the nodes were completely connected. This shows that equivalent nodes were mostly found when nodes have a small range (partially connected).

We also counted the total hubs used while routing from source node to target node under different routing conditions. We experimented with greedy, face and
hybrid routing and calculated the total hubs present before and after removing equivalent nodes. We also tested it when the external components were removed from the graph. Table 4.3 gives all the details that we calculated when the graph was partially connected. Table 4.3a gives the changes in hubs when we used greedy routing. Similarly, Table 4.3b and 4.3c gives all the details about changes obtained while using face and hybrid routing respectively. We can see that most of the nodes got stuck when we used greedy routing on partially connected graphs. One of the most significant results can be seen on Table 4.3b and 4.3c where the total hubs reduced a lot during face and hybrid routing respectively.

Table 4.2 shows the total change in hubs when different routing algorithms were used before and after removing equivalent nodes and external segments on completely connected graphs. Table 4.4a, 4.4b and 4.4c shows the changes obtained when greedy, face and hybrid routing were used. The table shows that there wasn’t significant
<table>
<thead>
<tr>
<th>Total Nodes</th>
<th>Range</th>
<th>Before Removing Eq. Nodes</th>
<th>After Removing Eq. Nodes</th>
<th>After Removing External Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>104</td>
<td>11(S)</td>
<td>11(S)</td>
<td>11(S)</td>
</tr>
<tr>
<td>300</td>
<td>87</td>
<td>3(S)</td>
<td>3(S)</td>
<td>3(S)</td>
</tr>
<tr>
<td>400</td>
<td>75</td>
<td>0(S)</td>
<td>0(S)</td>
<td>0(S)</td>
</tr>
<tr>
<td>500</td>
<td>66</td>
<td>2(S)</td>
<td>2(S)</td>
<td>2(S)</td>
</tr>
<tr>
<td>700</td>
<td>60</td>
<td>1(S)</td>
<td>1(S)</td>
<td>1(S)</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Total Nodes</th>
<th>Range</th>
<th>Before Removing Eq. Nodes</th>
<th>After Removing Eq. Nodes</th>
<th>After Removing External Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>104</td>
<td>51</td>
<td>46</td>
<td>42</td>
</tr>
<tr>
<td>300</td>
<td>87</td>
<td>782</td>
<td>726</td>
<td>446</td>
</tr>
<tr>
<td>400</td>
<td>75</td>
<td>800</td>
<td>713</td>
<td>402</td>
</tr>
<tr>
<td>500</td>
<td>66</td>
<td>1044</td>
<td>938</td>
<td>436</td>
</tr>
<tr>
<td>700</td>
<td>60</td>
<td>386</td>
<td>350</td>
<td>209</td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>Total Nodes</th>
<th>Range</th>
<th>Before Removing Eq. Nodes</th>
<th>After Removing Eq. Nodes</th>
<th>After Removing External Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>104</td>
<td>37</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>300</td>
<td>87</td>
<td>289</td>
<td>273</td>
<td>165</td>
</tr>
<tr>
<td>400</td>
<td>75</td>
<td>180</td>
<td>165</td>
<td>88</td>
</tr>
<tr>
<td>500</td>
<td>66</td>
<td>1443</td>
<td>1493</td>
<td>431</td>
</tr>
<tr>
<td>700</td>
<td>60</td>
<td>437</td>
<td>389</td>
<td>243</td>
</tr>
</tbody>
</table>

(c)

Table 4.3: Total changes in hubs (partially connected graph)

improvement when the nodes were completely connected.

From Table 4.3 and 4.4, we can see that more redundant nodes were found on partially connected graphs.
### Table 4.4: Total changes in hubs (completely connected graph)

<table>
<thead>
<tr>
<th>Total Nodes</th>
<th>Range</th>
<th>Total hubs present</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Before Removing Eq.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>After Removing Eq.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nodes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>After Removing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>External Segments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>238</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>300</td>
<td>192</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>400</td>
<td>156</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>500</td>
<td>133</td>
<td>35</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>600</td>
<td>116</td>
<td>31</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>200</td>
<td>238</td>
<td>35</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>300</td>
<td>192</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>400</td>
<td>156</td>
<td>39</td>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>500</td>
<td>133</td>
<td>42</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>600</td>
<td>116</td>
<td>45</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>200</td>
<td>238</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>300</td>
<td>192</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>400</td>
<td>156</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>500</td>
<td>133</td>
<td>35</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>600</td>
<td>116</td>
<td>31</td>
<td>28</td>
<td>28</td>
<td>28</td>
</tr>
</tbody>
</table>
CHAPTER 5

CONCLUSION

We presented a comprehensive review of networks used for communication in sensor networks. The reviewed networks included relative neighborhood graphs (RNG), Gabriel graphs (GG), Delaunay triangulation (DT), and restricted Delaunay graphs (RDG). Well-known routing algorithms for message forwarding in sensor networks were also examined. The examined routing algorithms include (i) greedy forward routing (ii) face routing, and (iii) greedy face hybrid routing.

We formulated the notion of redundant nodes in sensor networks. We showed that only one member from a set of equivalent nodes can be retained. This leads to a filtering algorithm for removing most redundant nodes. We proved that the removal of redundant nodes does not affect the connectivity between source node and target node. Another technique presented to filter unnecessary nodes is the removal of solo-faced chains. We also show that for fixed source node $s$ and fixed target node $t$, some two-connected network components can be removed without compromising the connectivity between $s$ and $t$.

We presented an experimental study of the proposed algorithms. The proposed algorithms were implemented in Java programming language. The resulting prototype program is easy to use and has a user friendly graphical interface. Observed results show that the proposed algorithms are effective in filtering redundant nodes. In many randomly generated networks, 12.28% of nodes can be identified as redundant.

This study can be pursued further. We studied the filtering problem for a two dimensional network. It would be interesting to extend the proposed algorithm to non-planar networks including three dimensional graphs.

Another avenue for further research would be to examine the performance of the
proposed algorithms for actual sensor networks formed by distributing sensor nodes on the surface of outdoor fields.
REFERENCES


VITA

Graduate College
University of Nevada, Las Vegas

Umang Amatya

Home Address:
4441 ESCONDIDO STREET APT 1205
Las Vegas, NV 89119

Degree:
Bachelor of Computer Engineering
Institute of Engineering, Pulchowk Campus, Tribhuvan University, Nepal

Thesis Title: Node Filtering and Face Routing for Sensor Network

Thesis Examination Committee:
Chairperson, Dr. Laxmi P. Gewali, Ph.D.
Committee Member, Dr. John T. Minor, Ph.D.
Committee Member, Dr. Ju-Yeon Jo, Ph.D.
Graduate Faculty Representative, Dr. Rama Venkat, Ph.D.