國 立 交 通 大 學

應用數學系

碩 士 論 文

感應式網路中的剛性性質以及唯一定位問題

The Rigidity Property and the Unique Localization Problem of Sensor Networks

研 究 生：蔡松育

指導教授：陳秋媛　教授

中 華 民 國 九 十 九 年 一 月

感應式網路中的剛性性質以及唯一定位問題

The Rigidity Property and the Unique Localization Problem of Sensor Networks

研 究 生：蔡松育　　Student: Sung-Yu Tsai

指導教授：陳秋媛　　Advisor: Chiuyuan Chen

國 立 交 通 大 學

應 用 數 學 系

碩 士 論 文

A Thesis
Submitted to Department of Applied Mathematics
College of Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Master
in
Applied Mathematics
January 2010
Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 九 年 一 月

感應式網路中的剛性性質以及唯一定位問題

研究生：蔡松育　　指導老師：陳秋媛　教授

國 立 交 通 大 學

應 用 數 學 系

摘　要

在感應式網路中，有些點知道本身的所在位置，而其他的點經由計算它們與鄰居之間的距離去決定自己的所在位置，我們將計算這些點的所在位置的過程稱之為網路定位。如果一個網路定位問題有唯一解，則稱之為可被解決的。在文獻[1]中證明了網路定位問題是可被解決的，如果其對應的基礎圖是具有全範圍剛性性質（亦即三連通、且具有多餘的剛性性質）。在文獻[5]中，Jacobs 和 Hendrickson 提出了一個演算法來辨識一個給定的圖是否具有剛性性質。我們稱一個圖為具有多餘的剛性性質，假如我們移掉任何一個邊之後，此圖還具有剛性性質。在這篇論文中，我們將會提供數個從具有剛性性質的圖去建構一個新的具有剛性性質的圖的方法，我們也會提出一個電腦程式來解決唯一定位問題；換句話說，我們的程式可以判斷一個給定的圖是否具有全範圍剛性性質，我們也將會提出一些實驗的結果。

關鍵詞：感應式網路、網路定位、基礎圖、剛性性質、多餘的剛性性質、全範圍剛性性質。

中 華 民 國 九 十 九 年 一 月

The Rigidity Property and the Unique Localization Problem of Sensor Networks

Student: Sung-Yu Tsai Advisor: Chiuyuan Chen

Department of Applied Mathematics
National Chiao Tung University
Hsinchu, Taiwan 30050

Abstract

In a sensor network, some nodes know their locations and other nodes determine their locations by measuring the distances to their neighbors. The process of computing the locations of the nodes is called network localization. A network localization problem is solvable if it has a unique solution. It has been proven in [1] that a network localization problem is solvable if and only if its corresponding grounded graph is globally rigid (i.e., 3-connected and redundantly rigid). A graph G is redundantly rigid if $G - e$ is rigid for any edge e in G. In [5], Jacobs and Hendrickson have proposed an elegant algorithm to check if a given graph is rigid. In this thesis, we will provide several ways to construct rigid graphs from rigid graphs. We will also implement a computer program for solving the unique localization problem; in other words, our program can check if a given graph is globally rigid. Some experimental results will also be proposed.

Keywords: sensor network, unique localization, grounded graph, rigidity, redundantly rigidity, globally rigidity.

誌　謝

能夠完成這篇論文，首先要感謝我的家人，能夠自小栽培我讀書到在現在，全力支持，讓我無後顧之憂，全力把心思放在課業上，從交通大學大學部到研究所這幾年的期間，因為有了許多人的幫助與支持，進而完成這篇論文，感謝指導老師陳秋媛教授，在讓她指導的兩年多的期間，不論是課業上還是其它方面都受到照顧，也拓展了不少研究上的視野。

此外，系上的很多老師，也在我修課期間教導了我許多相關的知識，像是大學時間的李榮耀老師、林松山老師、張麗萍老師、葉立明老師，以及研究所時期的傅恆霖老師、邵錦昌老師、翁志文老師等。

除了老師們的幫助，在這兩年半我很感謝同指導老師的藍國元學長、邱鈺傑學長、林威雄學長、黃志文學長、陳子鴻學長、黃信菖學長、劉宜君、劉士慶、曾慧棻、黃思綸、吳思賢、羅健峰，針對我的缺失給予了適時的建議，在他們身上也看到了不少值得學習的優點；最後感謝交大羽球隊廖威彰教練，讓我在球隊學習成長，不論是球技上的增進以及為人處事的啟發，都有莫大的幫助，也感謝全體隊友，球場上的幫助以及球場外的加油打氣，大專盃三連霸以及梅竹八連勝是我們最美好的回憶。

Contents

List of Figures

1 Introduction

In [1], Aspnes et al. provided a theoretical foundation for the problem of network localization problem in which some nodes know their locations and other nodes determine their locations by measuring the distances to their neighbors. More precisely, one begins with a network $\mathbf{N}$ in real d-dimensional space (where $d = 2$ or 3) consisting of a set of $m > 0$ nodes labeled 1 through m that represent the special "beacon" nodes together with $n - m > 0$ additional nodes labeled $m + 1$ through n that represent the ordinary nodes. Each node is located at a fixed position in $\Re^d$ and has associated with it a specific set of "neighboring" nodes. It is required that the definition of a neighbor is a symmetric relation on $\{1, 2, \ldots, n\}$ in the sense that node j is a neighbor of node i if and only if node i is also a neighbor of node j. Under these conditions, $\mathbf{N}$'s neighbor relationships can be described by an undirected graph $G_{\mathbf{N}} = (V, E_{\mathbf{N}})$ with vertex set $V = \{1, 2, \ldots, n\}$ and edge set $E_{\mathbf{N}}$ defined so that (i, j) is one of the graph's edges precisely when nodes i and j are neighbors. We assume throughout this thesis that $G_{\mathbf{N}}$ is a connected graph.

The *network localization problem with distance information* is to determine the locations of all nodes p_i in $\Re^d$ given the graph $G_{\mathbf{N}}$ of the network, the positions of the beacons p_j, $j \in \{1, 2, \ldots, m\}$ in $\Re^d$, and the distance $\delta_{\mathbf{N}}(i, j)$ between each neighbor pair $(i, j) \in E_{\mathbf{N}}$. We say that a network localization problem is *solvable* if there exists a unique set of nodes $\{p_{m+1}, p_{m+2} \ldots, p_n\}$ in $\Re^d$ consistent with the given data $G_{\mathbf{N}}$, $\{p_1, p_2, \ldots, p_m\}$, and $\delta_{\mathbf{N}} : E_{\mathbf{N}} \to \mathbb{R}$. The graph $G_{\mathbf{N}} = (V, E_{\mathbf{N}})$ is *unique localizable* if its corresponding network localization problem is solvable.

Before going further, we give an example of a graph which is not unique localizable. Consider the graph in Figure 1. Suppose that vertices a, b and d know their locations. Hence the distances between a and b, b and d, and also a and d are known. Suppose vertex c knows only its distances to a and b, but not the distance to d. Then there are two possible locations for c as shown in Figure 1(a) and (b). Consequently, the graph in Figure 1 is not unique localizable.

A *sensor network* consists of multiple detection stations called *sensor nodes*, each of

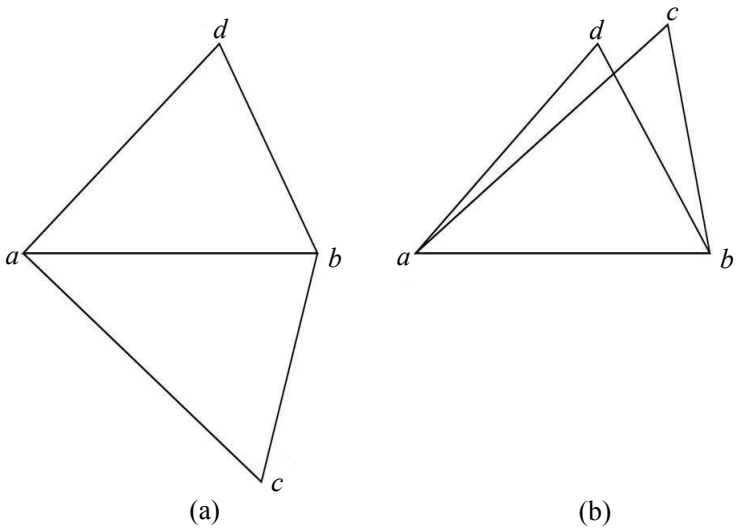


Figure 1: (a) One possible localization. (b) Another localization.

them is small, lightweight and portable. Every sensor node is equipped with a transducer, microcomputer, transceiver and power source. The transducer generates electrical signals based on sensed physical effects and phenomena. The microcomputer processes and stores the sensor output. The transceiver, which can be hard-wired or wireless, receives commands from a central computer and transmits data to that computer. The power for each sensor node is derived from the electric utility or from a battery.

A sensor network can be used to monitor and record conditions at diverse locations. The monitored parameters may be temperature, pressure, humidity, illumination intensity, wind direction and speed, vibration intensity, sound intensity, chemical concentrations, pollutant levels and vital body functions. In a sensor network, every sensor needs to know its location to detect and record events and to route packets [6]. So the network localization problem plays an important role for sensor networks. Recently, several methods [2, 8, 9] have been proposed to determine the location of the sensors in a sensor network.

All the graphs considered in this thesis are simple and undirected. Let $G = (V, E)$ denote a graph with vertex set $V(G)$ and edge set $E(G)$. The number of vertices and the number of edges in G are called the *order* and *size* of G, respectively. In this thesis, a node and a vertex are used interchangeably. A graph G is *k-connected* if it remains

connected upon the removal of any set of $< k$ vertices. The *connectivity* of a complete graph of order n is defined to be $n - 1$.

In [1], Aspnes et al. proposed the idea of grounded graphs. More precisely, in a *grounded graph*, each vertex represents a node in the given network and there is an edge between the two vertices if the distance between the two nodes is known. It has been assumed that when a sensor network is given, the locations of beacons are known. Hence the distance between any two beacons is implicitly known. Consequently, in a grounded graph, any two beacons are connected and therefore the subgraph graph induced by the beacons is a complete graph. In order to determine the locations, other sensor nodes must compute the distances between them and the beacons or the the distances between them and those sensor nodes which have already known their locations. As an example, in Figure 2, the vertices a, b and c are beacons and distances between vertices a and d, b and d, c and d, a and e, c and e, d and e are known. Because the vertices a, b and c are beacons, the distance between each pair of them is implicitly known. The vertex d can compute the distances between itself and the vertices a, b and c to determine its location. After d knows its location, the vertex e can compute the distances between itself and the vertices a, c and d to determine the location of e.

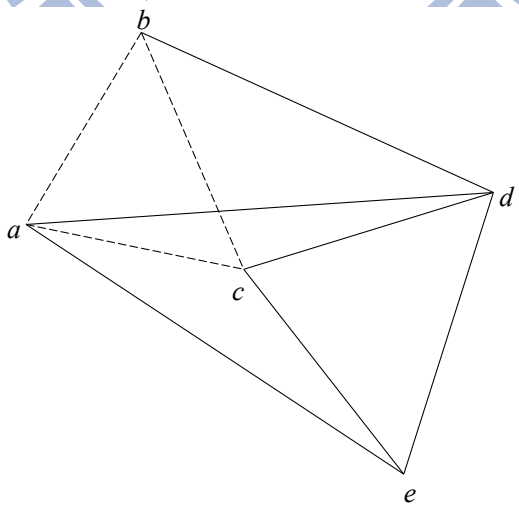


Figure 2: A grounded graph, in which vertices a, b and c are beacons.

It has been proven in [1] that: *A network has an unique localization if and only if its corresponding grounded graph is globally rigid.* Formal definitions of the rigidity and

globally rigidity of a graph is quite copious and can be found in [1]. In [7], it has been proven that a graph of order n must have at least $2n-3$ edges in order to be rigid. In [7], Laman proposed the following theorem for determining if a graph with $2n-3$ edges is rigid; for convenience, call this theorem Laman's Theorem.

Laman's Theorem A graph G of order n and size $2n-3$ is rigid if and only if it has no subgraph with more than $2n'-3$ edges, where n' is the number of vertices in the subgraph.

A graph is *redundantly rigid* if it remains rigid after removing any single edge. It has been proven in [4] that a graph G with ≥ 4 vertices is *globally rigid* in $\Re^2$ if and only if it is 3-connected and redundantly rigid in $\Re^2$. As an example, consider Figure 3. The graph in Figure 3(a) is not rigid. The graph in Figure 3(b) is rigid but not redundantly rigid. The graph in Figure 3(c) is 3-connected and redundantly rigid; hence this graph is globally rigid.

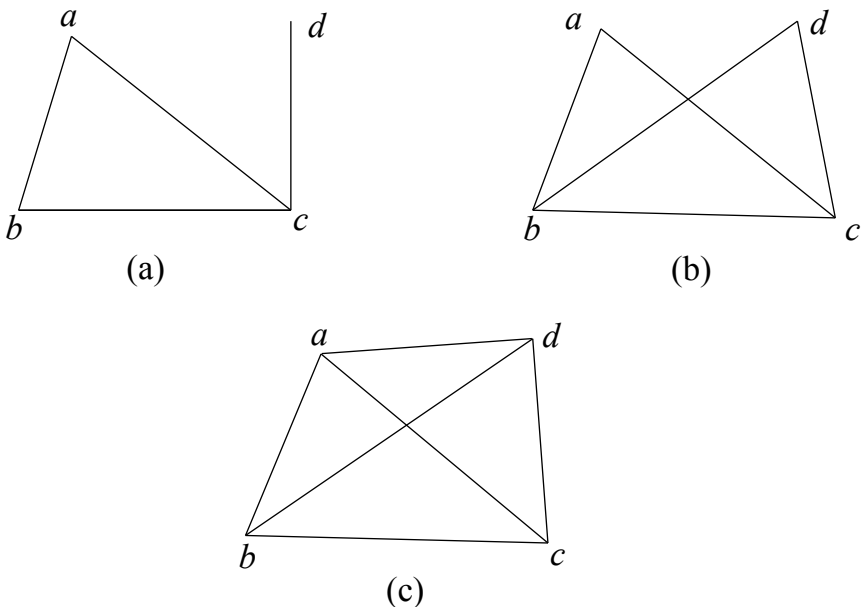


Figure 3: (a) A graph which is not rigid. (b) A graph which is rigid but not redundantly rigid. (c) A graph which is globally rigid.

In this thesis, we will provide several ways to construct rigid graphs from rigid graphs. In particular, we prove:

1. If a graph G is rigid, then the graph G' formed by adding a new vertex v and two new

edges connecting v to G is still rigid.

2. If a graph G is rigid, then the graph G' formed by adding two new vertices u and v and two new edges on u and v respectively is still rigid.

3. If the graph G_1 and G_2 are rigid, then the graph G' formed by adding 3 edges between G_1 and G_2 (under some constraints) is still rigid.

4. If the graph G_1,G_2 and G_3 are rigid, then the graph G' formed by adding 6 edges between G_1, G_2 and G_3 (under some constraints) is still rigid.

We also implement the algorithm proposed in [5] into a computer program in Turbo C programming language. Our computer program can determine if a given graph is globally rigid; hence it can determine if a given graph is 3-connected, rigid, and redundantly rigid. We also run simulation to obtain experimental results for the relation between the transmission range of sensors and globally rigid and the relation between the number of sensors and globally rigid.

The rest of the thesis is organized as follows. In Section 2, we will propose some properties for graph rigidity. In Section 3, we will introduce the algorithm (called the pebble game) proposed in [5]. We will implement the pebble game algorithm with computer program and present the simulation results for the unique localization problem in $\Re^2$. Concluding remarks and future works will be presented in Section 4. Finally, our computer program is listed in Appendix A and the detail data of our experimental results are given in Appendices B and C.

2 Some theoretical results

In this section, we will use Laman's Theorem to obtain a new rigid graph from a given rigid graph.

Theorem 1. *Let $G = (V, E)$ be a rigid graph of order n and size $2n - 3$. Let G' be the graph obtained from G by adding a new vertex v and two new edges (v, p_1) and (v, p_2),*

where p_1 and p_2 are two arbitrary vertices in $V(G)$. Then the new graph G' is rigid. (See Figure 4(a) as an illustration.)

Proof. Consider an arbitrary subgraph $\hat{G}$ of G'. If $\hat{G}$ does not contain v, then since $\hat{G} \subset G$, by the assumption that G is rigid, we have $|E(\hat{G})| \leq 2|V(\hat{G})| - 3$. On the other hand, let $\hat{G} = \dot{G} \cup v$, where $\dot{G} \subset G$. Then either $|E(\hat{G})| = |E(\check{G})| + 1$ or $|E(\hat{G})| = |E(\dot{G})| + 2$. In either case, since $\dot{G} \subset G$, by the assumption that G is rigid, we have $|E(\dot{G})| \leq 2|V(\dot{G})| - 3$. Hence $|E(\hat{G})| \leq |E(\dot{G})| + 2 \leq 2|V(\dot{G})| - 1 = 2(|V(\hat{G})| - 1) - 1 = 2|V(\hat{G})| - 3$. From the above, $|E(\hat{G})| \leq 2|V(\hat{G})| - 3$ holds for any subgraph $\hat{G}$ of G'; hence G' is rigid. ∎

In Theorem 1, we consider how to construct a new rigid graph G' from a given rigid graph G by adding a new vertex. Since the new graph G' is of order $n + 1$, by Laman's Theorem, G' must have at lest $2(n + 1) - 3 = 2n - 1 = (2n - 3) + 2$ edges. Hence we must add at least two new edges to G to obtain G'. By the same token, if we want to add k new vertices to G to obtain G', we have to add at least $2k$ edges. Theorem 2 considers the case of $k = 2$. The cases that $k \geq 3$ can be handled in a similar way. Notice that once a new vertex is added to G, the two new edges incident to this new vertex are added to G at the same time.

Theorem 2. *Let G be a rigid graph of order n and size $2n-3$. Let G' be the graph obtained from G by adding two new vertices v_1, v_2 and four new edges $(v_1, v_2), (p_1, v_1), (p_2, v_2), (p_3, v_1)$, where p_1, p_2 and p_3 are three arbitrary vertices in $V(G)$. Then the new graph G' is rigid. (See Figure 4(b) as an illustration.)*

Proof. Let G" be the graph obtained from G by adding vertex v_1 and edges (v_1, p_1) and (v_1, p_3). By Theorem 1, G" is rigid. Since G' is the graph obtained from G" by adding vertex v_2 and edges $(v_1, v2)$ and (v_2, p_2), by Theorem 1, G' is rigid. ∎

In the following, whenever we say that *three edges have no common endpoint*, we mean that there is no vertex incident to all of the three edges. Figure 5 shows an example of three edges have a common endpoint.

We have already shown how to obtain a new rigid graph from a given rigid graph by adding k new vertices. Now, we show how to obtain a new rigid graph from k rigid graphs

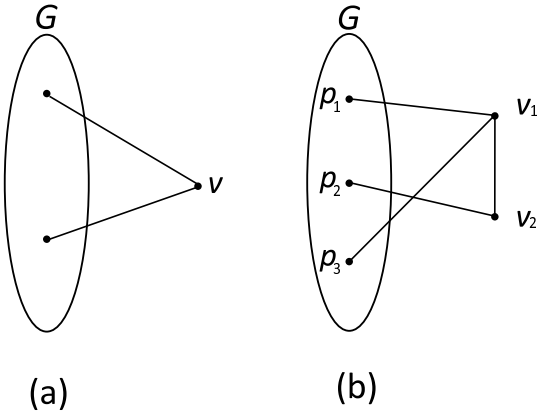


Figure 4: (a) An illustration of Theorem 1. (b) An illustration of Theorem 2.

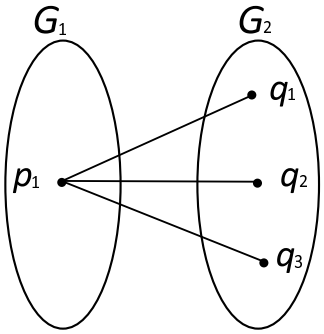


Figure 5: An illustration of Theorem 3.

$G_1, G_2, \ldots, G_k$. Suppose G_i is of order n_i and size $2n_i - 3$, for $i = 1, 2, \ldots, k$. Since the order of G' is $n_1 + n_2 + \cdots + n_k$, the size of G' must be at lest $2(n_1 + n_2 + \cdots + n_k) - 3 = ((2n_1 - 3) + (2n_2 - 3) + \cdots + (2n_k - 3)) + 3(k - 1)$, which means that we have to add at least $3(k - 1)$ edges to G to make the new graph G' be rigid. Theorem 3 is the $k = 2$ case.

Theorem 3. *Let G_1 and G_2 be two rigid graphs such that G_1 is of order n_1 and size $2n_1 - 3$, and G_2 is of order n_2 and size $2n_2 - 3$. Let G' be the graph obtained by adding three edges between G_1 and G_2. If the three edges have no common endpoints (as shown in Figure 6(a), (b), (c) and (d)), then the new graph G' is rigid.*

Proof. Let the three newly added edges be (p_1, q_2), (p_2, q_2) and (p_3, q_3), where $\{p_1, p_2, p_3\} \subseteq V(G_1)$ and $\{q_1, q_2, q_3\} \subseteq V(G_2)$. If the equality does not hold for $p_1 = p_2 = p_3$ *and* $q_1 = q_2 = q_3$. Consider an arbitrary subgraph $\hat{G}$ of G'. If $\hat{G} \subseteq G_1$ or $\hat{G} \subseteq G_2$, by the as-

sumption that G_1 and G_2 are rigid, we have $|E(\hat{G})| \leq 2|V(\hat{G})| - 3$. Now assume that $V(\hat{G}) = V(\hat{G}_1) \cup V(\hat{G}_2)$. Let $\hat{G}_1$ and $\hat{G}_2$ be the subgraph of $\hat{G}$ such that $\hat{G}_1 \subseteq G_1$ and $\hat{G}_2 \subseteq G_2$. Also, let $|V(\hat{G}_1)| = \hat{n}_1$ and $|V(\hat{G}_2)| = \hat{n}_2$. Since G_1 and G_2 are rigid, it follows that $|E(\hat{G}_1)| \leq 2\hat{n}_1 - 3$ and $|E(\hat{G}_2)| \leq 2\hat{n}_2 - 3$. Thus $|E(\hat{G})| \leq |E(\hat{G}_1)| + |E(\hat{G}_2)| + 3 \leq (2\hat{n}_1 - 3) + (2\hat{n}_2 - 3) + 3 = 2(\hat{n_1} + \hat{n_2}) - 3 = 2(|V(\hat{G}_1)| + |V(\hat{G}_2)|) - 3 = 2|V(\hat{G})| - 3$. Thus $|E(\hat{G})| \leq 2|V(\hat{G})| - 3$ holds for any subgraph $\hat{G}$ of G'. Hence G' is rigid. ∎

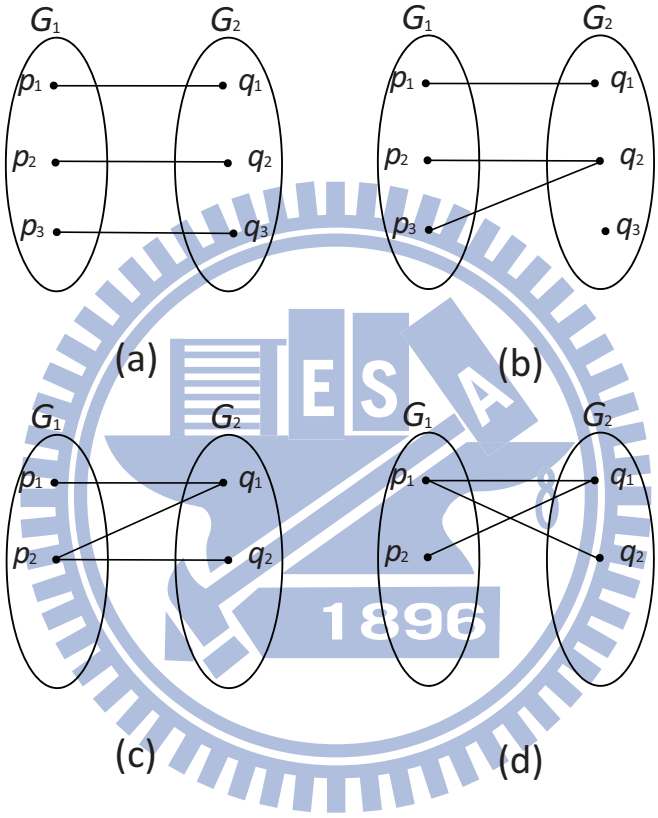


Figure 6: An illustration of Theorem 3.

In Theorem 3, it is required that the three newly added edges between G_1 and G_2 have no common endpoints; the reason is as follows. If $p_1 = p_2 = p_3$ (see Figure 5) or $q_1 = q_2 = q_3$, then consider the subgraph $\tilde{G}$ of G' with $V(\tilde{G}) = V(G_2) \cup \{p_1\}$ and $E(\tilde{G}) = E(G_2) \cup \{(p_1, q_1), (p_1, q_2), (p_1, q_3)\}$. Then $|V(\tilde{G})| = |V(G_2)| + 1 = n_2 + 1$. Since $|E(\tilde{G})| = |E(G_2)| + 3 = 2n_2 = 2(n_2 + 1) - 2 > 2(n_2 + 1) - 3 = 2|V(\tilde{G})| - 3$, G' is not rigid.

Theorem 4. *Let G_1, G_2 and G_3 be three rigid graphs such that G_i is of order n_i and size $2n_i - 3$, for $i = 1, 2, 3$. Then the new graph G' obtained by adding six edges between G_1,*

G_2 and G_3 is still rigid if (1) the number of edges between any two of G_1, G_2 and G_3 is ≤ 3 and (2) if there are three edges between any two of G_1, G_2 and G_3, then the three edges have no common endpoints. (See Figure 7(a), (b), (c).)

Proof. First consider the case that among the six newly added edges, three of them are added between two of G_1, G_2 and G_3. Without loss of generality, we assume that three newly added edges are between G_1 and G_2 and call these three edges e_a, e_b, e_c (see Figure 7(a),(b)). Also assume that the remaining three edges are called e_d, e_e, e_f. Since there are six edges in total, e_d, e_e, e_f are incident to vertices in G_3. If the three edges e_a, e_b, e_c have no common endpoint, then by Theorem 3, the graph $\dot{G}'$ with $V(\dot{G}') = V(G_1) \cup V(G_2)$ and $E(\dot{G}') = E(G_1) \cup E(G_2) \cup \{e_a, e_b, e_c\}$ is rigid. Again, by Theorem 3, when $\dot{G}'$ is rigid and the edges e_d, e_e, e_f have no common endpoint, G' is rigid.

Now consider the case that two of the six newly added edges are between G_1 and G_2, two of them are between G_1 and G_3, and two of them are between G_2 and G_3 (see Figure 7(c)). Let $\hat{G}$ be an arbitrary subgraph of G'. There are three cases.

Case 1: $\hat{G} \subseteq G_i$ for some $i \in \{1, 2, 3\}$. By the assumption that G_1,G_2 and G_3 are rigid, we have $|E(\hat{G})| \leq 2|V(\hat{G})| - 3$.

Case 2: $\hat{G} \subseteq (G_i \cup G_j)$ for some $i, j \in \{1, 2, 3\}$ and $\hat{G} \not\subseteq G_i$ for any $i \in \{1, 2, 3\}$. Without loss of generality, assume $\hat{G} \subseteq (G_1 \cup G_2)$. Let $\hat{G}_1$ and $\hat{G}_2$ be the subgraphs of $\hat{G}$ such that $\hat{G}_1 \subseteq G_1$ and $\hat{G}_2 \subseteq G_2$. Also, let $|V(\hat{G}_1)| = \hat{n}_1$ and $|V(\hat{G}_2)| = \hat{n}_2$. Since G_1 and G_2 are rigid, it follows that $|E(\hat{G}_1)| \leq 2\hat{n}_1 - 3$ and $|E(\hat{G}_2)| \leq 2\hat{n}_2 - 3$. Thus $|E(\hat{G})| \leq |E(\hat{G}_1)| + |E(\hat{G}_2)| + 2 \leq (2\hat{n}_1 - 3) + (2\hat{n}_2 - 3) + 2 = 2(\hat{n}_1 + \hat{n}_2) - 4 \leq 2(\hat{n}_1 + \hat{n}_2) - 3 = 2|V(\hat{G})| - 3$.

Case 3: $V(\hat{G}) = V(\hat{G}_1) \cup V(\hat{G}_2) \cup V(\hat{G}_3)$ and $\hat{G} \not\subseteq (G_i \cup G_j)$ for any $i, j \in \{1, 2, 3\}$ and $\hat{G} \not\subseteq G_i$ for any $i \in \{1, 2, 3\}$. Let $\hat{G}_1$, $\hat{G}_1$ and $\hat{G}_3$ be the subgraph of $\hat{G}$ such that $\hat{G}_1 \subseteq G_1$, $\hat{G}_2 \subseteq G_2$ and $\hat{G}_3 \subseteq G_3$. Also, let $|V(\hat{G}_1)| = \hat{n}_1$, $|V(\hat{G}_2)| = \hat{n}_2$ and $|V(\hat{G}_3)| = \hat{n}_3$. Since G_1, G_2 and G_3 are rigid, it follows that $|E(\hat{G}_1)| \leq 2\hat{n}_1 - 3$, $|E(\hat{G}_2)| \leq 2\hat{n}_2 - 3$ and $|E(\hat{G}_3)| \leq 2\hat{n}_3 - 3$. Thus $|E(\hat{G})| \leq |E(\hat{G}_1)| + |E(\hat{G}_2)| +$

$$|E(\hat{G}_3)|+6 \leq (2\hat{n}_1-3)+(2\hat{n}_2-3)+(2\hat{n}_3-3)+6 = 2(\hat{n}_1+\hat{n}_2+\hat{n}_3)-3 = 2|V(\hat{G})|-3.$$

From the above, $|E(\hat{G})| \leq 2|V(\hat{G})|-3$ holds for any subgraph $\hat{G}$ of G'; thus G' is rigid. ∎

Theorem 4 limits the number of newly added edges between any two of G_1, G_2 and G_3 to three edges. What will happen if we add more than three edges between two of the graphs G_1, G_2 and G_3? Without loss of generality, suppose we add more than three edges between G_1 and G_2 (see Figure 7(d)) and let E' denote the set of these newly added edges between G_1 and G_2. Let $\hat{G}$ be the graph with vertex set $V(\hat{G}) = V(G_1) \cup V(G_2)$ and edge set $E(\hat{G}) = E(G_1) \cup E(G_2) \cup E'$. Then $|E(\hat{G})| > |E(G_1)| + |E(G_2)| + 3 = (2n_1 - 3) + (2n_2 - 3) + 3 = 2(n_1 + n_2) - 3 = 2|V(\hat{G})| - 3$ and hence G' is not rigid.

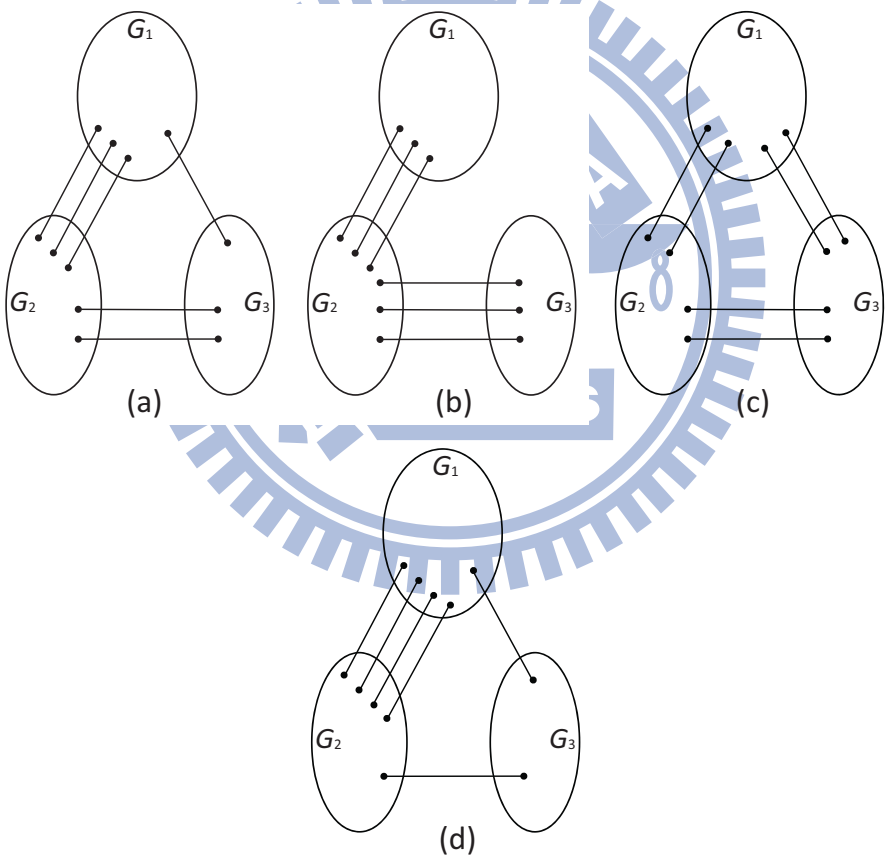


Figure 7: An illustration of Theorem 4, where (a), (b), (c) are allowed and (d) is not. (a)(b)(c) together

In Figure 8, we list the possible cases of constructing a rigid graph from four rigid graphs G_1, G_2, G_3 and G_4, where each G_i is of order n_i and size $2n_i - 3$. In Figure 8 (a),(b) and (c), consider the graph $G_{1,2}$ obtained by adding three edges between G_1 and

G_2, we can use Theorem 3 to prove $G_{1,2}$ is rigid. The graph $G_{3,4}$ obtained by adding three edges between G_3 and G_4 is rigid by the same token. Then we can use the same method to prove that the resultant graph obtained by adding three edges between $G_{1,2}$ and $G_{3,4}$ is rigid. We also can use similar arguments to prove that the graphs in Figure 8(d), (e), (f), (g) and (h) are also rigid. That the graphs in Figure 8(i) and (j) are rigid can not be proven by the above theorem; however, they can be proven by similar methods and we omit the proofs.

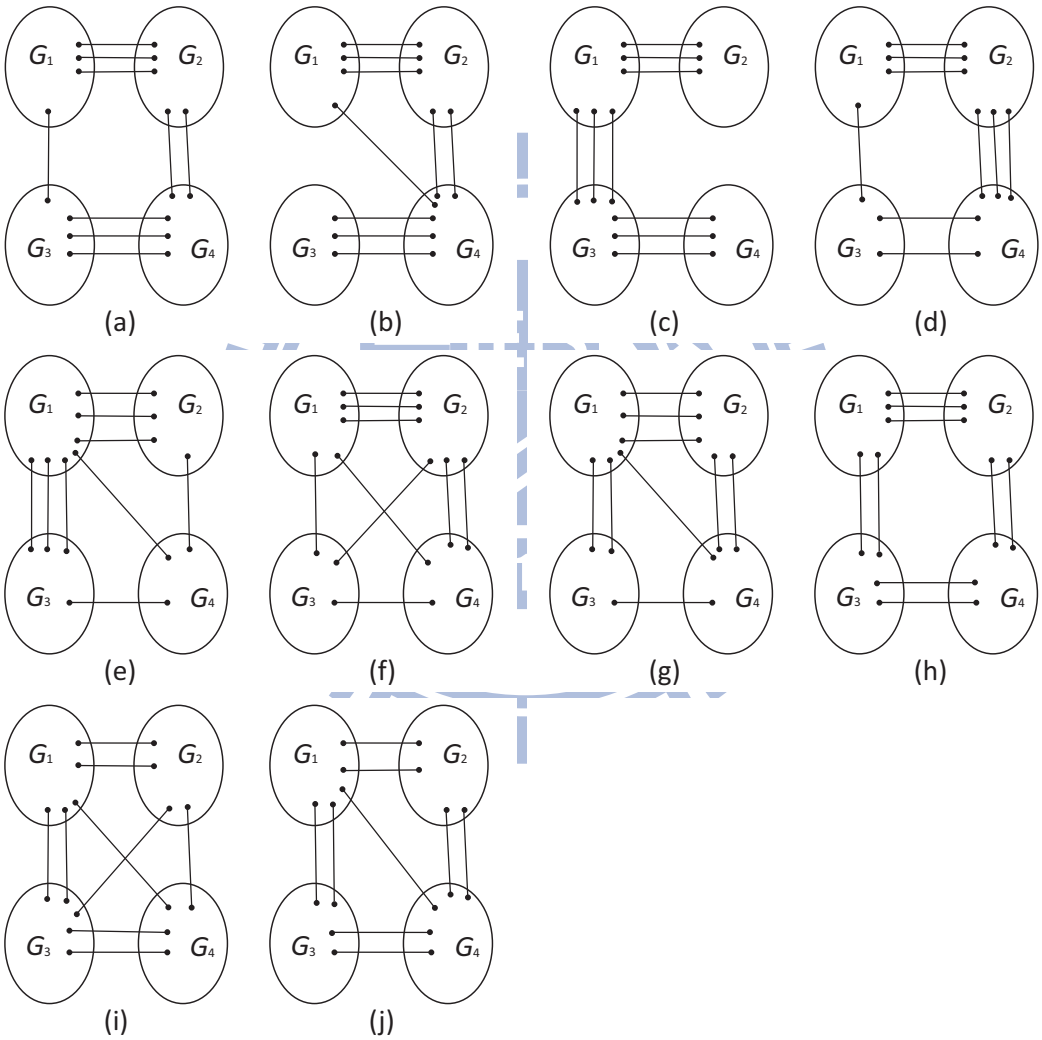


Figure 8: Constructing a rigid graph from four rigid graphs G_1, G_2, G_3 and G_4.

3 Some experimental results for the localization problem

Although Laman's Theorem can be used to check if a given graph is rigid, it is inefficient since in the worst case it has to compute the number of edges for every subgraph of a given graph and a graph has an exponential number of subgraphs. In [5], Jacobs and Hendrickson proposed an efficient algorithm, called *pebble game*, to check if a given graph is rigid. The pebble game algorithm takes only $O(nm)$ time, where n is the order and m is the size of the given graph. In the first part of this section (subsection 3.1), we give the pebble game algorithm proposed in [5]. In the second part of this section (subsection 3.2), we present some experimental results.

3.1 The pebble game algorithm

The following two theorems are foundations of the pebble game algorithm.

Theorem 5. [7] *The edges of a graph $G = (V, E)$ are independent in $\Re^2$ if and only if no subgraph $G' = (V', E')$ of G has more than $2n' - 3$ edges, where n' is the order of G'.*

Theorem 6. [5] *For a graph $G = (V, E)$ of order n and size m, the following statements are equivalent.*

A. *The edges of G are independent in $\Re^2$.*

B. *For each edge (a, b) in G, the graph formed by adding three additional (multiple) edges (a, b) has no induced subgraph G' in which $m' > 2n'$, where n' is the order and m' is the size of G'.*

Let G be a graph of order n and size m. The basic idea behind the pebble game algorithm is to grow a maximal set of independent edges at a time. Denote these basis edges (the edges that belong to a maximal set of independent edges) by $\hat{E}$. A new edge can be added to $\hat{E}$ if it is discovered to be independent of $\hat{E}$. If $2n - 3$ independent edges are found, then G is rigid. The key is the efficient determination of whether or not a new edge is independent to the current basis $\hat{E}$.

Assume that we have a set $\hat{E}$ (may be empty) of independent edges. Combine $\hat{E}$ with the vertices of G and thus form a graph $\hat{G}$. We want to determine if another edge e is independent of $\hat{E}$ by adding e to $\hat{G}$. Let $\overline{G}$ be the graph obtained by adding e to $\hat{G}$. By Theorem 6, e is independent of $\hat{E}$ if and only if there is no subgraph with too many edges such that $m' > 2n'$ after any edge in $\overline{G}$ is quadrupled (i.e., adding three edges between the same pair of vertices). The following lemma suggests that only e needs to be quadrupled.

Lemma 7. [5] *A new edge e is independent of $\hat{E}$ if and only if the graph G_{4e} formed by quadrupling e has no induced subgraph G' in which $m' > 2n'$, where n' is the order and m' is the size of G'.*

Lemma 7 reduces the time complexity of independence testing to that of counting edges in subgraphs once a new edge is quadrupled. The pebble game algorithm is based on Lemma 7 and it works as follows.

The pebble game algorithm:

Initially, each vertex is given two pebbles and can use its pebbles to cover any two edges which are incident to it. We would like to assign the pebbles in such a way that all the edges in the graph are covered. Such an assignment is called a *pebble covering*. In [5], it has been shown that the existence of a pebble covering is equivalent to that there is no induced subgraph on n' vertices with more than $2n'$ edges, which is the independence condition in Lemma 7.

The following is an approach for moving pebbles to cover a new edge. Assume that we have a set of edges $\hat{E}$ that are covered with pebbles. First, look at the two vertices incident to the new edge. If either one of the two vertices has a free pebble, then use it to cover the new edge and exit. Otherwise, their pebbles are used to cover existing edges. If a vertex at the other end of one of these edges has a free pebble, then that pebble can be used to cover the existing edge, freeing up a pebble which can be used to cover the new edge. More precisely, the algorithm searches for free pebbles in a directed graph, in which each edge has a direction. Specifically, if edge (a, b) is covered by a pebble from vertex a, then the edge is directed from a to b. From each vertex v, search along the edges (at

most two) directed away from v. The search for free pebbles continues until either a free pebble is found and a sequence of swaps allows a new edge to be covered, or else no more vertices can be reached and no free pebbles have been discovered. We list in Figure 9 the pebble game algorithm given in [5].

The end of the pebble game algorithm

If Enlarge_Cover (see Figure 9) fails, then all the pebbles owned by the vertices encountered in the search are already covering bonds. If n' vertices were encountered, then they must have at least $2n'$ induced edges to spend all their pebbles. This observation proves that the existence of a pebble covering is equivalent to the edge independence; see the following.

Lemma 8. [5] *If the new edge e is tripled instead of quadrupled to form G_{3e} , then G_{3e} has a pebble covering.*

Lemma 9. [5] *If G_{4e}, the graph constructed by quadrupling e, does not have a pebble covering, then the set of n' vertices encountered in Find_Pebble already have $2n' - 3$ induced edges.*

Therefore, if the pebble game algorithm determines that a new edge is not independent, then it means that a subgraph on n' vertices already contains $2n' - 3$ edges; otherwise, the pebble game algorithm will enlarge the covering successfully.

Theorem 10. [5] *A new edge e is independent of $\hat{E}$ if and only if there exists a pebble covering when e is quadrupled.*

Corollary 11. [5] *If $\hat{E}$ is independent and a corresponding pebble covering $\hat{G}$ is known, then determining whether a new edge is independent requires $O(n)$ time.*

The pebble game algorithm takes $O(nm)$ time, in which each enlargement of a pebble covering requires $O(n)$ time.

```
Algorithm Enlarge_Cover(G, e_ab)
    For Each vertex v
        seen(v) = False, path(v) = -1
    found = Find_Pebble(G, a, seen, path)
    If (found) Then
        Rearrange_Pebbles(G, a, seen)
        Return (Success)
    If (not seen(b)) Then
        found = Find_Pebble(G, b, seen, path)
        If (found) Then
            Rearrange_Pebbles(G, b, path)
            Return (Success)
    Return (Failure)

Function Find_Pebble(G, v, seen, path)
    seen(v) = True, path(v) = -1
    If (v has free pebble) Then
        Return (True)
    Else
        x = neighbor along edge my pebble covers
        If (not seen(x)) Then
            path(v) = x
            found = Find_Pebble(G, x, seen, path)
            if (found) Then Return (True)
        y = neighbor along edge my other pebble covers
        If (not seen(y)) Then
            path(v) = y
            found = Find_Pebble(G, y, seen, path)
            if (found) Then Return (True)
    Return (False)

Subroutine Rearrange_Pebbles(G, v, path)
    While (path(v) ≠ −1)
        w = path(v)
        if (path(w) = −1) Then
            Cover edge (v,w) with free pebble from w
        Else
            Cover edge (v,w) with pebble from edge (w,path(w)
        v = w
```

Figure 9: Algorithm for enlarging a pebble covering.

3.2 Experimental results

In our simulation, we assume each sensor has the same transmission range and we randomly generate various network topology of different settings. For each setting, we

perform the simulation for 500 times and compute the number of 3-connected graphs, the number of rigid graphs, the number of redundantly rigid graphs, and the number of globally rigid graphs among 500 randomly generated graphs. We vary the number of sensors and the transmission range to evaluate the impact of this two factors.

In Figures 10, 11 and 12, we give three examples of our computer program. In each of these examples, the number of sensors is 100, the transmission range is 25, and the region is of size $100 \times 100m^2$.

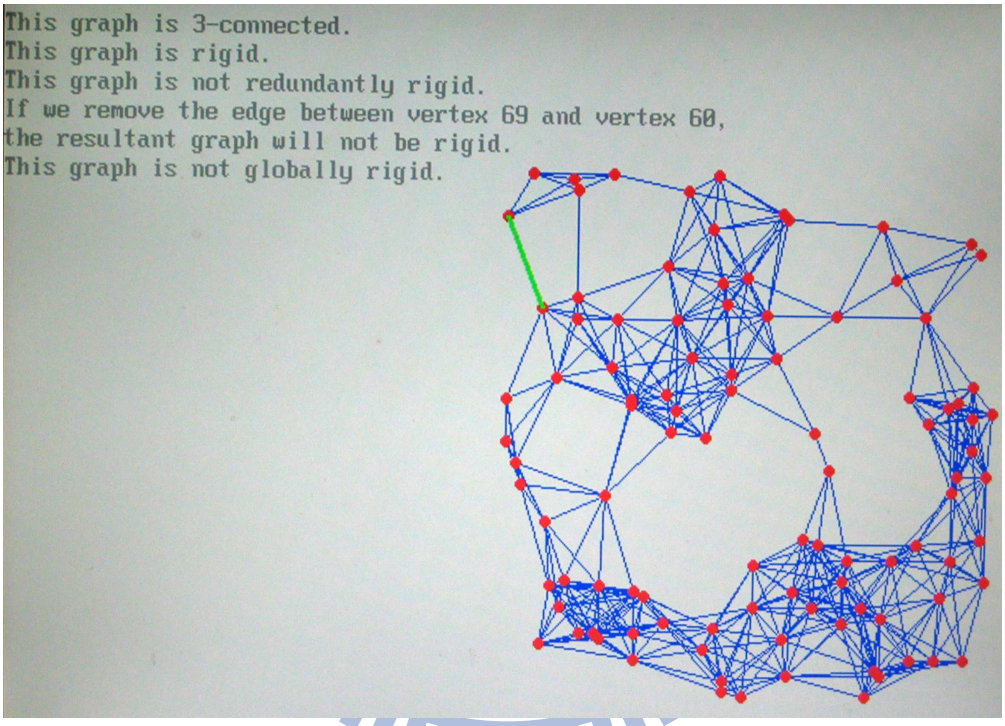


Figure 10: Example 1 of our computer program.

In our simulation, we randomly place sensors in a $100 \times 100m^2$ region. Figure 13 shows our experimental result when the transmission range is fixed; see also Appendix B for the detailed data. In particular, Figure 13(a) is the result of fixing the transmission range at $20m$, which is 1/5 to the area edge ($100 \times 100m^2$); Figure 13(b) is the result of fixing the transmission range at $25m$, which is 1/4 to the area edge ($100 \times 100m^2$); and Figure 13(c) is the result of fixing the transmission range at $33m$, which is about 1/3 to the area edge ($100 \times 100m^2$). In Figure 13, the number of nodes varies from 30 to 120 and we count the number of 3-connected graphs, the number of rigid graphs, the number of

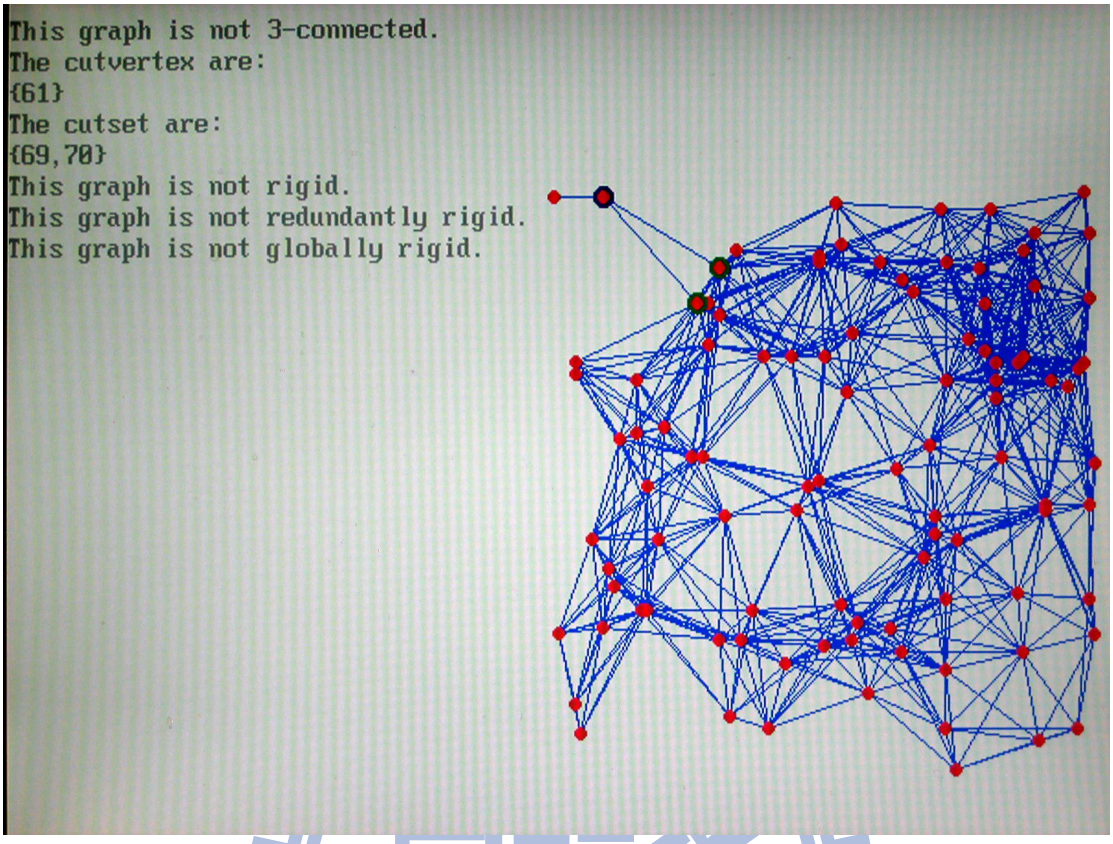


Figure 11: Example 2 of our computer program.

redundantly rigid graphs, and the number of globally rigid graphs among 500 randomly generated graphs. For example, from Appendix B, we know that when the transmission range is $20m$ and the number of sensors is 100, there are 113 3-connected graphs, 289 rigid graphs, 144 redundantly rigid graphs, and 113 globally rigid graphs among the 500 randomly generated graphs. Figure 13 and Appendix B show that when the number of sensors is raised from 30 to 120, the number of 3-connected, rigid, redundantly rigid, and globally graphs also increases.

Figure 14 shows our experimental result when the number of sensors is fixed; see also Appendix C for the detailed data. In particular, Figure 14(a) is the result of fixing the number of sensors at 50; Figure 14(b) is the result of fixing the number of sensors at 60; and Figure 14(c) is the result of fixing the number of sensors at 70. In Figure 14, the transmission range varies from $10m$ to $50m$ and we count the number of 3-connected graphs, the number of rigid graphs, the number of redundantly rigid graphs, and the

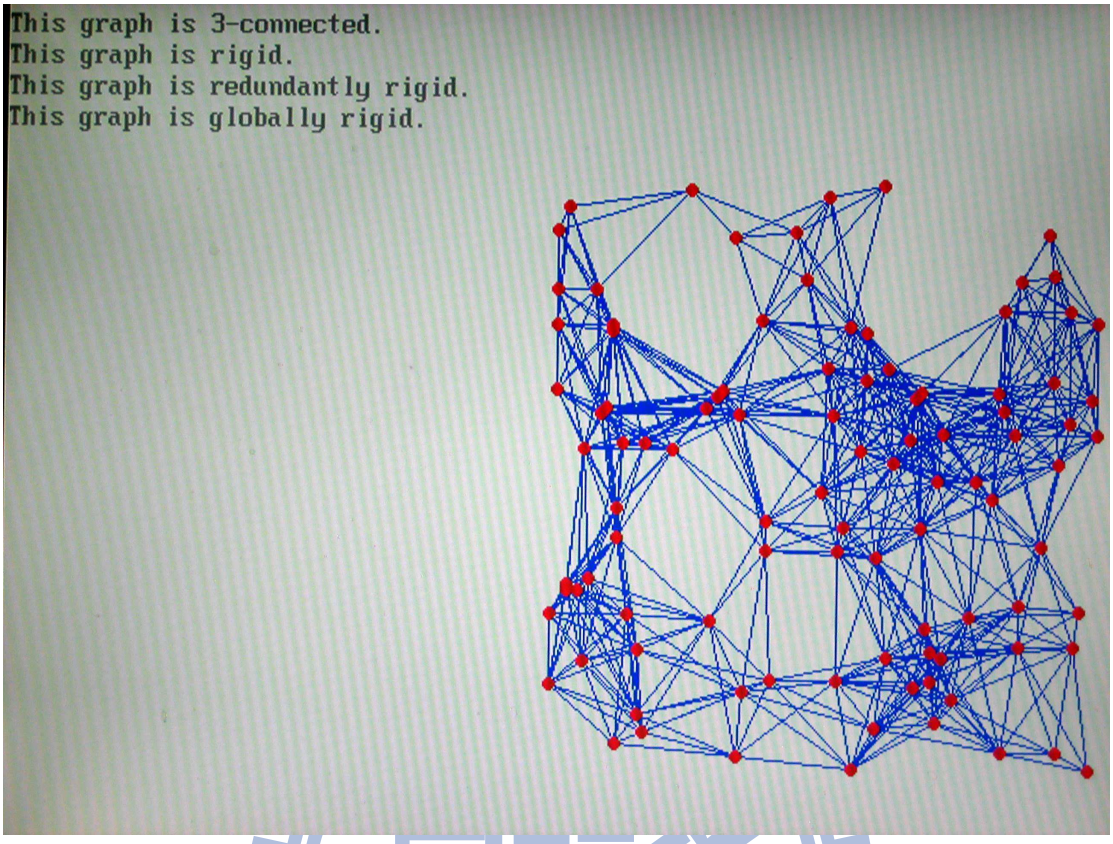


Figure 12: Example 3 of our computer program.

number of globally rigid graphs among 500 randomly generated graphs. For example, from Appendix C, we know that when the transmission range is $30m$ and the number of sensors is 50, there are 137 3-connected graphs, 269 rigid graphs, 155 redundantly rigid graphs, and 137 globally rigid graphs among the 500 randomly generated graphs. Figure 14 and Appendix C show that when the transmission range is raised from $10m$ to $50m$, the number of 3-connected, rigid, redundantly rigid, and globally graphs also increases.

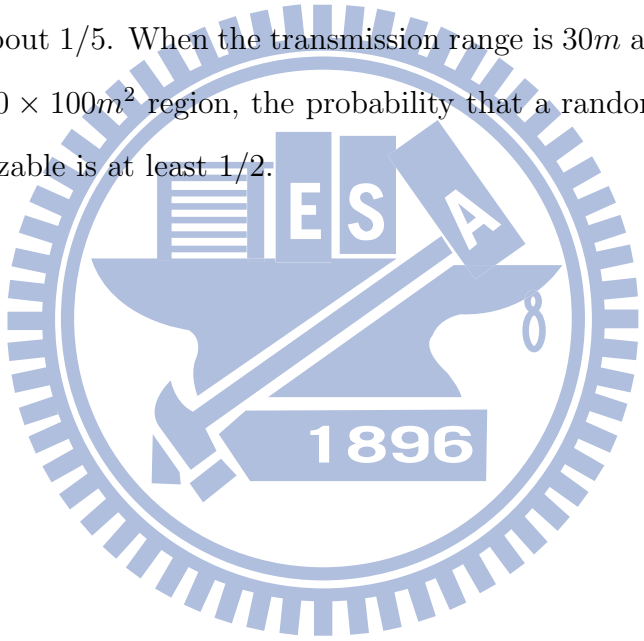
From our experimental results, we have several observations.

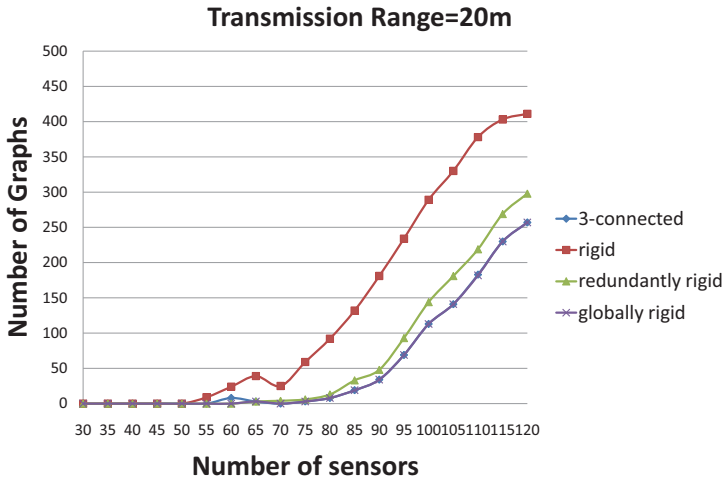
- Our simulation shows that when a randomly generated graph is 3-connected, it has a very high possibility of being globally rigid. Therefore, in our simulation, the number of globally rigid graphs and the number of 3-connected graphs are almost the same.

- Let N_s and N_G denote the number of sensors and the number of globally rigid graphs,

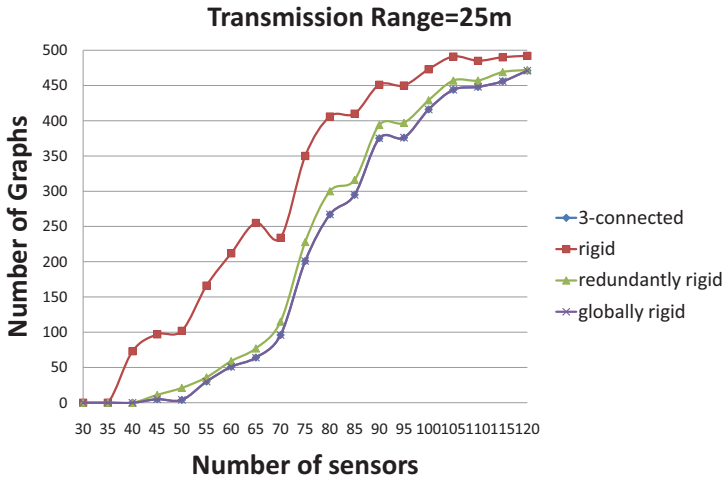
respectively. In our simulation, if we fix N_s, then N_G does not increase very fast when the transmission range varies from $10m$ to $20m$; however, N_G increases very fast when the transmission range varies from $20m$ to $35m$. Moreover, if we fix the transmission range, then the increasing speed of N_G is not obvious when N_s increases from 30 to 120.

- When the transmission range is $25m$ and when we place 60 sensors in a $100 \times 100m^2$ region, the probability that a randomly generated graph is unique localizable is at least 1/10. When the transmission range is $25m$ and when we place 70 sensors in a $100 \times 100m^2$ region, the probability that a randomly generated graph is unique localizable is about 1/5. When the transmission range is $30m$ and when we place 60 sensors in a $100 \times 100m^2$ region, the probability that a randomly generated graph is unique localizable is at least 1/2.

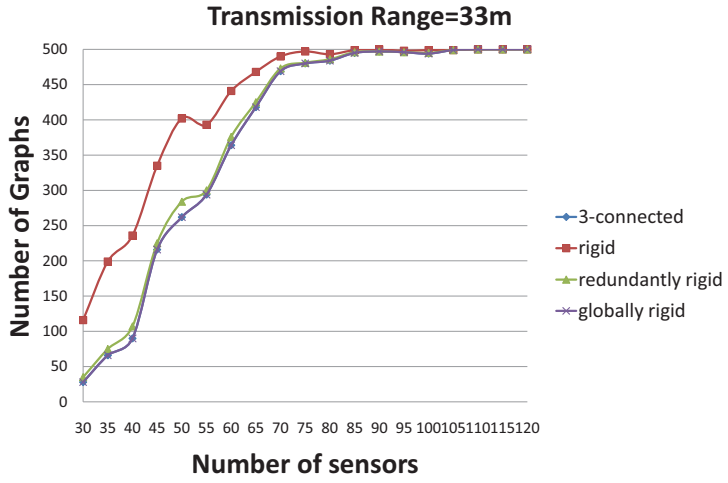




(a)

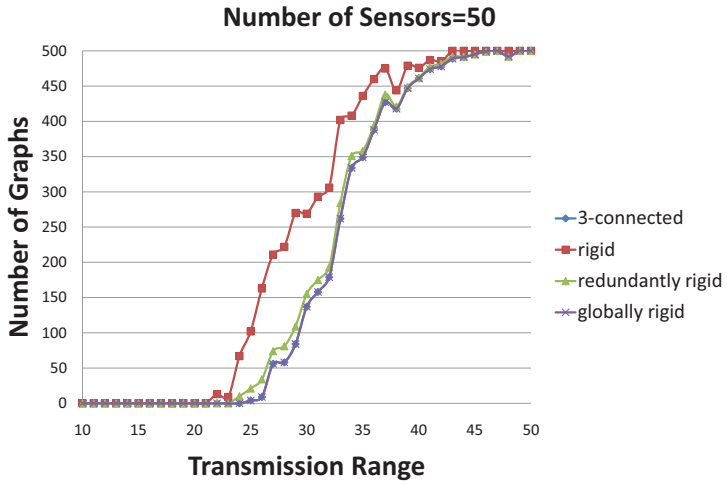


(b)

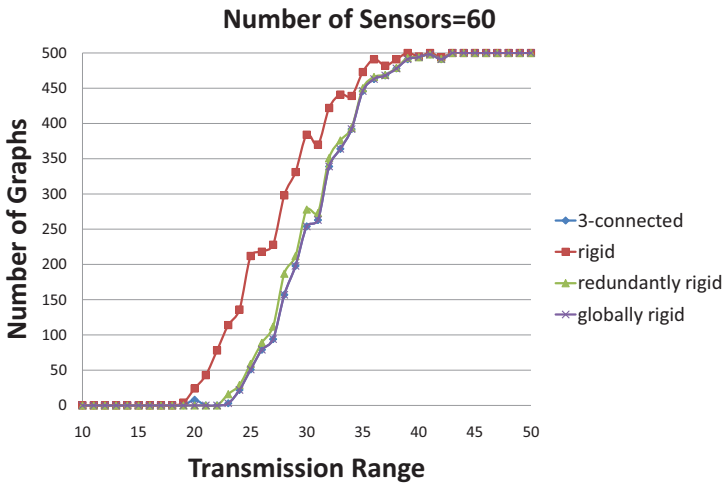


(c)

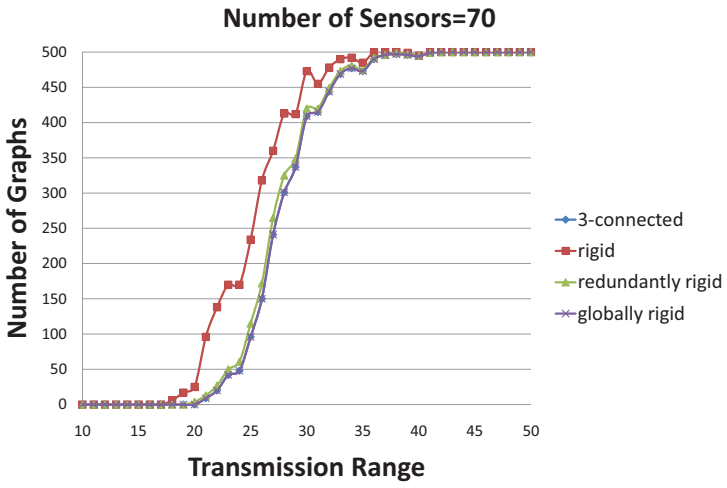
Figure 13: Our experimental result when the transmission range is fixed.



(a)



(b)

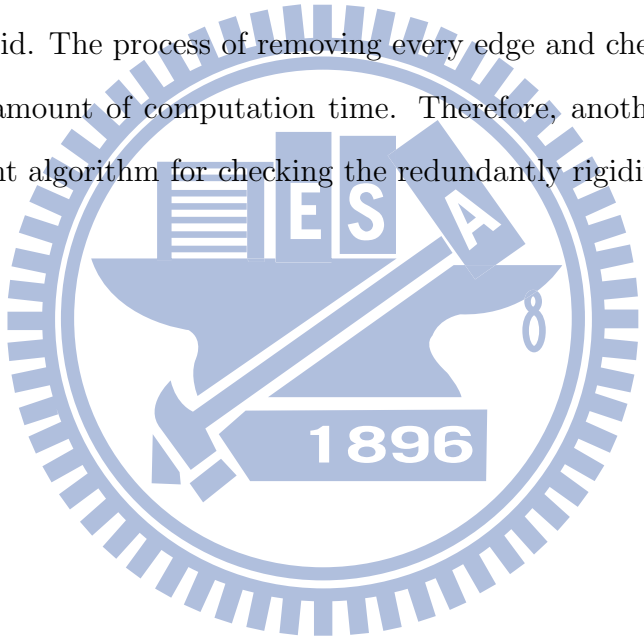


(c)

Figure 14: Our experimental result when the number of sensors is fixed.

4 Concluding remarks

In the thesis, we study the unique localization problem and provide several ways to construct rigid graphs from rigid graphs. We also implement a computer program for the unique localization problem and propose some experimental results. Let N_s and N_G be defined as in the previous section. In this thesis, we try to determine the relation between N_s and N_G, and the relation between transmission range and N_G. One future work is to increase the number of randomly generated graphs to see if there is any trend in the increasing rate. Furthermore, since if we want to check a given graph is redundantly rigid, we have to remove every edge of the randomly generated graph to check if the resultant graph is rigid. The process of removing every edge and checking each resultant graph takes a huge amount of computation time. Therefore, another future work is to design a more efficient algorithm for checking the redundantly rigidity of a given graph.



References

[1] J. Aspnes and T. Eren and D. Goldenberg, et al., "A theory of network localization," IEEE Transactions on Mobile Computing, vol. 5, no. 12, 2006.

[2] T. He, C. Huang, B. Blum, J. Stankovic, and T. Abdelzaher, "Range-free localization schemes in large scale sensor networks," in Proc. Ninth Int'l Conf. Mobile Computing and Networking (MobiCom), pp. 81-95, 2003.

[3] B. Hendrickson, "Conditions for unique graph realizations," SIAM J. Computing, vol. 21, no. 1, pp. 65-84, 1992.

[4] B. Jackson and T. Jordan, "Connected rigidity matroids and unique realizations of graphs," J. Combinatiorial Theory B, vol.94, pp. 1-29, 2005.

[5] D. Jacobs and B. Hendrickson, "An algorithm for two dimensional rigidity percolation: The pebble game," J. Computional Physics, vol. 137, no. 2, pp. 346-365, 1997.

[6] B. Karp and H.T. Kung, "GPSR: Greedy Perimeter Stateless Routing for wireless networks," in Proc. Sixth Int'l Conf. Mobile Computing and networking(mobiCom), 2000.

[7] G. Laman, "On graphs and rigidity of plane skeletal structures," J. Eng. Math., vol. 4, pp. 331-340, 2002.

[8] D. Moore, J. Leonard, D. Rus, and S.Teller, "Robust distributed network localization with noisy range measurements," in Proc. Second ACM Conf. Embedded Networked Sensor Systems (SenSys), Nov. 2004.

[9] A. Savvides, C.-C. Han, and M.B. Strivastave, "Dynamic fine-grained localization in ad-hoc networks of sensors," in Proc.Seventh Int'l Conf. Mobile Computing and Networking (MobiCom), pp. 166-179, 2001.

A Appendix: The source code of our program

```
//File Name: GloblyRigid.cpp
//Author:  蔡松育
//Email Address: tbofdciyib@yahoo.com.tw
//Description: Count the number of globally rigid graphs among 100 randomly generated graphs.
//Input:    None.
//Output: The numbers of 3-connected, rigid, redundantly rigid, and globally rigid graphs among these 100
//            randomly generated graphs.

#include <iostream>
#include <fstream>
#include <ctime>
#include <cstdlib>
#include <conio.h>
#include <math.h>
using namespace std;

#define testtime 100      //number of randomly generate graphs
#define range 100         //range of sensors
#define radius 30         //radius of sensors
#define square (r)*(r);
#define RAND_MAX 32767
const int SIZE=70;        //number of sensors

int absolute[100][100]; //the adjacency matrix of the randomly generated graph
int dir[100][100]={0};  //the adjacency matrix used to record the directed graph obtained by pebble game
int pebble[100];        //pebble[i] is the number of pebbles on vertex i
int temppebble[100]={0}; //If we want to add edge (i,j) into the graph, then temppebble[i] and temppebble[j]
                         //record the number of pebbles borrowed from i and j, respectively
int seen[100];          //If vertex i has been searched in pebble game, then seen[i] is set to 1
int path[100];          //path[i] represents the son of vertex i after the function find_pebble is executed
unsigned xxxx=1;

struct coordinate
{
        int x;
        int y;
}sensor[SIZE];          //the coordinates of sensors

//------------------------------ functions used to generate sensors start here
void srand(unsigned s){xxxx=s;}

int rand()
{
        xxxx=214013*xxxx+2531011;
        return(xxxx>>16)&0x7FFF;
}

void in(int n)
{
        time_t t;
        time(&t);
        srand(static_cast<unsigned>(t));

        for(int i=0;i<n;i++)
        {
                sensor[i].x=rand()%range;
                sensor[i].y=rand()%range;
        }
```

```
}
//------------------------------ functions used to generate sensors end here

//--------------------------------pebble game starts here
void rearrange_pebbles(int dir[100][100], int v, int path[100])
{
	if(path[v]==-1)
		pebble[v]=pebble[v]-1;
	while(path[v]!=-1)
	{
		int w=path[v];

		if(path[w]==-1)
		{
			dir[w][v]=1;
			dir[v][w]=0;
			pebble[w]=pebble[w]-1;
		}
		else
			dir[w][v]=1;dir[v][w]=0;
		v=w;
	}
}

int find_pebble(int dir[100][100], int v, int seen[100], int path[100])
{
	seen[v]=1;
	path[v]=-1;

	int found;
	int x=-1;
	int y=-1;

	if(pebble[v]>0)
	{
		return 1;
	}
	else
	{
		for(int i=0; i<SIZE; i++)
			if(dir[v][i]>0){x=i; break;}
		for(int j=x+1; j<SIZE; j++)
			if(dir[v][j]>0){y=j; break;}

		if(x!=-1)
		{
			if(seen[x]==0)
			{
				path[v]=x;
				found=find_pebble(dir,x,seen,path);
				if(found) return 1;
			}
		}
		if(y!=-1)
			if(!seen[y])
			{
				path[v]=y;
				found=find_pebble(dir,y,seen,path);
```

```
                        if(found) return 1;
                }
        }
        return 0;
}

int enlarge_cover(int dir[100][100], int p, int q)
{
        for(int i=0; i<SIZE; i++)
        {
                seen[i]=0;
                path[i]=-1;
        }
        int found=find_pebble(dir,p,seen,path);

        if(found)
        {
                rearrange_pebbles(dir,p,path);
                temppebble[p]++;
                return 1;
        }

        if(!seen[q])
        {
                found=find_pebble(dir,q,seen,path);
                if(found)
                {
                        rearrange_pebbles(dir,q,path);
                        temppebble[q]++;
                        return 1;
                }
        }

      return 0;
}

int rigid(void)
{

        for(int i=0; i<SIZE; i++)
                pebble[i]=2;

        int pb1,pb2,pb3,pb4;

        for(int i1=0;i1<SIZE;i1++)
                for(int i2=i1+1;i2<SIZE;i2++)
                {
                        if(absolute[i1][i2]!=0)
                        {

                                pb1=enlarge_cover(dir,i1,i2);
                                pb2=enlarge_cover(dir,i1,i2);
                                pb3=enlarge_cover(dir,i2,i1);
                                pb4=enlarge_cover(dir,i2,i1);

                                        if(pb1 && pb2 && pb3 && pb4)
                                {
                                        dir[i1][i2]=1;
```

```
                                pebble[i1]=1;
                                pebble[i2]=2;
                                temppebble[i1]=0;
                                temppebble[i2]=0;
                        }
                        else
                        {
                                pebble[i1]=temppebble[i1];
                                pebble[i2]=temppebble[i2];
                                temppebble[i1]=0;
                                temppebble[i2]=0;
                        }
                }
        }

        int total=0;

        for(int t=0; t<SIZE; t++)
                total=total+pebble[t];

        if(total==3)
                return 1; //If the tatal number of remaining pebbles is 3, the graph is rigid and we return true
        else
                return 0; //Otherwise, we return false
}
//-----------------------------------------------------pebble game ends here


//-----------------------------------------------------test 3-connected starts here
void dfs_visit(int u, int color[100], int pi[100], int b[100][100]) //dfs
{
        color[u]=1;
        for(int v=0;v<SIZE;v++)//if edge(u,v) belong to E(G) and v is never searched,then color v=1
        {
                if (b[u][v]==1 && color[v]==0)
                {
                        pi[v]=u;
                        dfs_visit(v,color,pi,b);
                }
        }
}

void cutvertex(int i, int j, int b[100][100], int& ans)
//remove two vertices i and j from G, and then check if the resultant graph G' is connected
{
        for(int k1=0; k1<=SIZE-1; k1++) //remove all the edges incident to vertex i
        {
                b[i][k1]=0;
                b[k1][i]=0;
        }

        for(int k2=0; k2<=SIZE-1; k2++) //remove all the edges incident to vertex j
        {
                b[j][k2]=0;
                b[k2][j]=0;
        }

        int pi[100];
```

```
        int color[100];

        for(int s=0; s<SIZE; s++) //color every vertex with color 0;
        {
                color[s]=0;
                pi[s]=-1;
        }

        //choose a vertex called candidate that belongs to G-{u,v}
        //varible candidate represents the vertex we start a DFS
        int p=0;
        int candidate;

        if(p!=i) //since i and j are removed, we can not choose them as the candidate
                candidate=p;
        else
                candidate=p+1;
        if (candidate==j) candidate++;

        dfs_visit(candidate,color,pi,b); //do DFS

        color[i]=1; //color the removed vertex i with color 1
        color[j]=1; //color the removed vertex j with color 1

        for(int a=0; a<SIZE; a++) //check if every vertex is reached
                if(color[a]==0)     //If there is vertex which is not reached, the graph is not connected,
                        ans=0;    //ans=0 means the graph is not triconnected
}

int test(int b[100][100])
{
        int ans=1; //varible ans represnets the graph is triconnected or not

        for(int i=0; i<=SIZE-2; i++) //choose two vertices i and j so that they will be reomoved
                for(int j=i+1; j<=SIZE-1; j++)
                {
                        cutvertex(i,j,b,ans); //remove vertices i and j and test if the resultant graph is connected
                        for(int t1=0; t1<SIZE; t1++) //recover matrix b to matrix absolute
                                for(int t2=0; t2<SIZE; t2++)
                                        b[t1][t2]=absolute[t1][t2];
                }

        return ans;
}

//-----------------------------------------------------------test 3-connected ends here


//-----------------------------------------------------------main program starts here
int main()
{
        int tricount=0;     //number of triconnected graphs among 100 randomly generated graphs
        int rigidcount=0;           //number of rigid graphs among 100 randomly generated graphs
        int redundantlyrigidcount=0; //number of redundantly rigid graphs among 100 randomly generated graphs
        int globallyrigidcount=0;       //number of globallyrigid graphs among 100 randomly generated graphs

        ofstream outf;
        FILE *f1,*f2;
```

```
ofstream outfdata3;
outfdata3.open("s70r30.txt",ios::app);

for(int ttime=0; ttime<testtime; ttime++)
{
                //---------------------------------------generate the vertices randomly
        outf.open("data1.txt",ios::out);
        int n=SIZE;
        in(n);

        for(int my_write=0; my_write<n; my_write++)
                outf << "(" << sensor[my_write].x << "," << sensor[write].y << ")" << endl;

        outf << endl;
        outf << endl;
        outf.close();

        //---------------------------------------read the coordinates to construct adjacency matrix
        struct coordinate vt[100];

        if ((f1=fopen("data1.txt","r"))==NULL)
        {
                printf("data1 cannot be opened\n");
                exit(1);
        }
        if ((f2=fopen("data2.txt","w"))==NULL)
        {
                printf("data2 cannot be opened\n");
                exit(1);
        }

        char next1;
        int q1=0;
        int tem[10];
        int data1size=0;

        while((next1=fgetc(f1))!=EOF)
        {
                if(next1=='(')
                continue;

                tem[q1]=atoi(&next1);
                q1++;

                int total=0;

                if(next1==',')
                {
                        for(int i1=0; i1<q1; i1++)
                                total=total+tem[i1]*pow(10,q1-i1-2);
                        q1=0;
                        vt[data1size].x=total;
                        total=0;
                }

                if(next1==')')
                {
```

```
                for(int i2=0; i2<q1; i2++)
                        total=total+tem[i2]*pow(10,q1-i2-2);
                q1=0;
                vt[data1size].y=total;
                total=0;
                data1size++;
        }
}

for(int p1=0; p1<data1size; p1++)
        for(int p2=p1+1; p2<data1size; p2++)
        {
                int length=(vt[p1].x-vt[p2].x)*(vt[p1].x-vt[p2].x)+
                                                (vt[p1].y-vt[p2].y)*(vt[p1].y-vt[p2].y);
                int r=radius;

                if(length<=(r*r))
                {
                        absolute[p1][p2]=1;
                        absolute[p2][p1]=1;
                }
        }

for(int p3=0; p3<data1size; p3++)
{
        for(int p4=0; p4<data1size; p4++)
                if(absolute[p3][p4]==0)
                        fputc('0',f2);
                        else
                        fputc('1',f2);
        fputc('\n',f2);
}

fputc('\n',f2);
fputc('\n',f2);
fputc('\n',f2);
fclose(f1);
fclose(f2);

//---------------------------------------Count the numbers of 3-connected, rigid, redundantly rigid, and
//---------------------------------------globally rigid graphs among 100 randomly generated graphs. int
condition1=0; //0 means not triconnected and 1 means triconnected
int condition2=0; //0 means not rigid and 1 means rigid
int condition3=0; //0 means not redundantly rigid and 1 means redundantly rigid

int b[100][100];

for(int t3=0; t3<SIZE; t3++) //copy the data from matrix absolute to matrix b
        for(int t4=0; t4<SIZE; t4++)
                b[t3][t4]=absolute[t3][t4];

if(test(b)==1) //test if b is 3-connected?
{
        tricount++;
        condition1=1;
}

int reduntrigid=1;
```

```
            if(rigid()==1)
            {
                  condition2=1;
                  rigidcount++;
            }
            else
                  reduntrigid=0;

            if(reduntrigid==1)
                  for(int   t11=0;   t11<data1size-1;   t11++)
                        for(int   t12=t11+1;   t12<data1size;   t12++)
                              if(absolute[t11][t12]==1)
                              {
                                    int temp1=absolute[t11][t12];
                                    int temp2=absolute[t12][t11];

                                    absolute[t11][t12]=0;
                                    absolute[t12][t11]=0;
                                    for(int tt1=0;tt1<size;tt1++)
                                    {
                                          for(int tt2=0;tt2<size;tt2++)
                                          dir[tt1][tt2]=0;
                                          temppebble[tt1]=0;
                                    }
                                    if(rigid()!=1)
                                    {
                                          reduntrigid=0;
                                          t11=data1size-1;
                                          t12=data1size-1;
                                    }

                                          absolute[t11][t12]=temp1;
                                          absolute[t12][t11]=temp2;
                              }
            condition3=reduntrigid;

            if(reduntrigid==1) redundantlyrigidcount++;

            if((condition1==1) && (condition2==1) && (condition3==1)) globallyrigidcount++;

          for(int    tt3=0; tt3<size; tt3++)
                                    {
                                          for(int tt4=0;tt4<size;tt4++)
                                          {dir[tt3][tt4]=0; absolute[tt3][tt4]=0;}
                                          temppebble[tt1]=0;
                                    }

      }

      outfdata3 << tricount << " " << rigidcount << " " << redundantlyrigidcount
                        << " " << globallyrigidcount << endl;
      outfdata3.close();

      return 0;
}
//-----------------------------------------------------main program ends here
```

B Appendix: detail data of our experimental results when the transmission range is fixed

Transmission range = $20m$

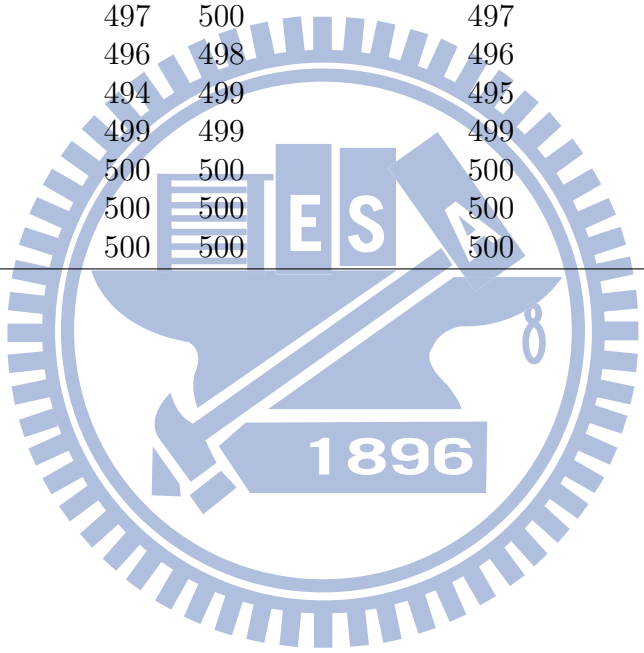
G_s	3-connected	rigid	redundantly rigid	globally rigid
30	0	0	0	0
35	0	0	0	0
40	0	0	0	0
45	0	0	0	0
50	0	0	0	0
55	0	9	0	0
60	8	24	0	0
65	3	39	3	3
70	0	25	4	0
75	3	59	6	3
80	8	92	13	8
85	19	132	33	19
90	34	181	48	34
95	69	234	93	69
100	113	289	144	113
105	141	330	181	141
110	183	378	219	182
115	230	403	269	230
120	257	411	298	257

Transmission range = $25m$

G_s	3-connected	rigid	redundantly rigid	globally rigid
30	0	0	0	0
35	0	0	0	0
40	0	73	0	0
45	5	97	11	5
50	4	102	21	4
55	30	166	36	30
60	51	212	59	51
65	64	255	77	64
70	96	234	115	96
75	201	350	228	201
80	267	406	300	267
85	295	410	316	295
90	375	451	394	375
95	376	450	397	376
100	416	473	429	416
105	444	491	457	444
110	448	485	457	448
115	456	490	469	456
120	471	492	472	471

Transmission range = $33m$

G_s	3-connected	rigid	redundantly rigid	globally rigid
30	28	116	35	28
35	66	199	75	66
40	90	236	107	90
45	216	335	225	216
50	262	402	284	262
55	294	393	300	294
60	364	441	376	364
65	418	468	425	418
70	469	490	473	469
75	480	497	481	480
80	484	493	486	484
85	495	499	496	495
90	497	500	497	497
95	496	498	496	496
100	494	499	495	494
105	499	499	499	499
110	500	500	500	500
115	500	500	500	500
120	500	500	500	500



C Appendix: detail data of our experimental results when the number of sensors is fixed

Number of sensors = 50

transmission range	3-connected	rigid	redundantly rigid	globally rigid
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
18	0	0	0	0
19	0	0	0	0
20	0	0	0	0
21	0	0	0	0
22	0	13	0	0
23	0	9	0	0
24	0	67	10	0
25	4	102	21	4
26	9	163	34	9
27	56	211	74	56
28	58	222	81	58
29	84	270	109	84
30	137	269	155	137
31	158	293	175	158
32	179	306	193	179
33	262	402	284	262
34	334	408	351	334
35	349	436	358	349
36	388	460	394	388
37	427	475	438	427
38	418	444	421	418
39	447	479	449	447
40	461	476	462	461
41	474	487	477	474
42	478	485	482	478
43	489	500	492	489
44	491	500	492	491
45	495	500	495	495
46	499	500	499	499
47	500	500	500	500
48	492	500	492	492
49	500	500	500	500
50	500	500	500	500

Number of sensors = 60

transmission range	3-connected	rigid	redundantly rigid	globally rigid
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
18	0	0	0	0
19	0	4	0	0
20	8	24	0	0
21	0	43	0	0
22	0	78	0	0
23	3	114	16	3
24	22	136	29	22
25	51	212	59	51
26	79	218	89	79
27	94	228	112	94
28	157	298	187	157
29	198	331	212	198
30	254	384	278	254
31	263	370	273	263
32	339	422	351	339
33	364	441	376	364
34	392	439	394	392
35	446	473	450	446
36	463	491	466	463
37	468	482	469	468
38	478	491	479	478
39	491	500	493	491
40	494	495	494	494
41	498	500	498	498
42	491	494	492	491
43	500	500	500	500
44	500	500	500	500
45	500	500	500	500
46	500	500	500	500
47	500	500	500	500
48	500	500	500	500
49	500	500	500	500
50	500	500	500	500

Number of sensors = 70

transmission range	3-connected	rigid	redundantly rigid	globally rigid
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
18	0	6	0	0
19	0	17	0	0
20	0	25	4	0
21	9	96	13	9
22	20	138	27	20
23	42	170	50	42
24	48	170	61	48
25	96	234	115	96
26	150	318	172	150
27	241	360	265	241
28	301	413	325	301
29	337	412	349	337
30	409	473	420	409
31	415	455	420	415
32	444	478	449	444
33	469	490	473	469
34	477	492	481	477
35	473	485	475	473
36	490	500	492	490
37	496	500	496	496
38	497	500	499	497
39	496	499	497	496
40	494	495	495	494
41	499	500	499	499
42	500	500	500	500
43	500	500	500	500
44	500	500	500	500
45	500	500	500	500
46	500	500	500	500
47	500	500	500	500
48	500	500	500	500
49	500	500	500	500
50	500	500	500	500