PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGH EDUCATION AND SCIENTIFIC RESEARCH

Order N°:……..

MOHAMED BOUDIAF UNIVERSITY- M'SILA
MATHEMATICS AND COMPUTING FACULTY

Computer Science Department

DISSERTATION

Submitted in Partial Fulfillment of The Requirements
for The Degree of MASTER

Field: Computer Sciences
Major: Networks
By: Ibrahim BOURAS

Theme

Rigidity Test Using The Pebble Game
Distributed Implementation

Discussed publicly on 31st may 2016 by the jury:

Mr. S. Gasmia	University of M'sila	President
Mr. Adel MOUSSAOUI	University of M'sila	Research Director
Mr. A. Guemmougui	University of M'sila	Member

June 2016

To my parents

Aknowledgement

I would like to express my deep sense of gratitude and heartfelt thanks to my advisor, Mr. Adel MOUSSAOUI Who proposed the right theme for the right student.

I feel very grateful and privileged to have worked on this difficult theme and I remember very well when the supervisor said: This theme is of high degree, that time I thought it is easy, but it was the hardest algorithm I have ever seen in my entire life. So, I thank my advisor again for the confidence he put on me.

I thank him once more for the importance of theme he proposed.

I would like to thank the jury members for their patience in reading this dissertation as well as all my professors, teachers, instructors, helpers and advisers in my entire career one by one, THANK YOU from the bottom of my heart.

Table of Contents

Figure List

INTRODUCTION

The Rigidity theory has a long history. It begun since the 18th century, we found some scientists talked about the closed spatial figure[12]. In the 19th century, Maxwell [23] was studying whether structures are stable or deformable, and a host of other engineers who worked on bridges and mechanical linkages [10].

Rigidity theory for plane bar and joint frameworks (2D) have known a mile stone in the second half of the 20th century. Henneberg introduced important inductive technique so as to construct planar bar and joint frameworks [17]. Henneberg's techniques are still widely used, particularly to prove and analyze rigidity and flexibility of such structure [36].

The Rigidity theory in modern era has known a remarkable advancement with G.Laman who introduced a theorem in 1970 [19] which brought the rigidity problem completely to the set of combinatorial problem.

In this combinatorial characterization of rigidity, one can simply count the vertices and edges in a graph in order to determine its flexibility or rigidity. Even though the ancient scientists and other mechanical engineers [12], were counting vertices and edges to evaluate rigidity of a given graph.

Since that time, there has been an evolving interest in this fascinating subject, and our understanding and characterization of rigid and flexible structures has remarkably grown over the past four decades.

Researchers are continuing to apply rigidity theory in many field such as computer-aided design (CAD), molecular biology, glass network flexibility predictions, sensor networks (localization, communication), and linkages in mechanical engineering [26,28], and host of others.

Throughout our work in this research paper, we will focus on testing the combinatorial graph rigidity of the structures using the famous pebble game algorithm in totally decentralized systems. So, the asynchronous decentralization of this algorithm.

The aim of this dissertation is the distributed implementation of the pebble game algorithm in order to evaluate rigidity of a distributed system using threads, inter-process communications techniques in asynchronous message passing.

To achieve this aim we have to put an outline. So, our work will be divided into three major chapters.

The first chapter contains general overview and background of rigidity theory, including the basic notions of graphs, frameworks, rigidity theory, the pebble game algorithm, some theorems for counting the combinatorial properties, and some details about the sufficient and

necessary condition for 2D and 3D, as we will talk about the famous open problem of Double Banana in 3D.

The second chapter will show the interest of rigidity theory in various application such as Architecture, Protein, Computer-Aided Design, Bridges, Glass frameworks, Chemistry, as we will focus on some examples of the needle towers.

The third chapter contains a detailed description of the distributed implementation of the pebble game algorithm using java threads. As we will mention some detail about java thread operations and synchronization, and at last we will mention the detailed execution steps of a given graph.

At the end, we will conclude a general conclusion about the whole work, then we will mention the list of references used in our research.

CHAPTER 1

RIGIDITY THEORY: BACKGROUND AND CONCEPTS

1.Overview

In this chapter we give some essential concepts of the rigidity theory. Starting with the formal definition of Rigidity and its types. We give also a review of graph theory terminology, then we will give the combinatorial condition for checking rigidity in dimension 2. We describe the pebble game algorithm as the best combinatorial solution of evaluating rigidity in planar networks, in its centralized version. We will provide illustrations of this algorithm, leaving other details which can be found in the mentioned references.

1.2. Basic concepts

Let us start with the most studied structures, the bar and joint frameworks, which are consists of bars and joints. We will give more formal definitions in next sections. We will primarily illustrate the concepts and definitions using the 2-dimensional structures in the Euclidean space.

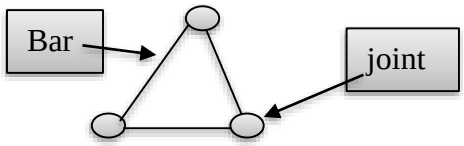


Figure 1.1 : Bar and Joint Framework

In bar and joint frameworks (or frameworks in short) (Figure1) we consider that the bars are solid and rigid, they cannot be changed in distances nor broken, and the joints are freely rotatable [24]. The joints in 2-dimensional space play the role of a connector between bars, which impose the restriction that the bars share the common endvertices. The bars act as distance constraints, which keep the distance between two joints. If we move continuously such a framework in the plane, the distances among pairs of joints that are connected by a bar will remain the same throughout the motion[i].

Yet, the question is: will the distances between other non-connected joints remain fixed?

We are interested in the motions of framework, as its possible motions will guide us to the answers about rigidity or flexibility. A deformation or flex is a motion which preserves the distances of all the bars of the framework but changes the distance between some pair of unconnected joints of the framework.

[i] There are also other types of structures called tensegrities, where some bars are replaced with cables (the distance between joints can shrink but not expand) and struts (the distance can expand but not shrink) We will not be looking at tensegrities in our work.

If no deformation exists, then any motion is said to be a *rigid motion* of a framework, in other words, a *rigid motion* of the framework *preserves* the distance between all joints in the framework.

A framework is rigid if it has no deformations, that is to say, all of its motions are rigid motions, and is flexible otherwise [i].

Rigid motions or rigid body motions are often called trivial motions, and any motions arising from a deformation are known as non-trivial motions.

A key concept in analysis and description of rigidity and motions is the degrees of freedom of a framework, or DOF in short.

The term *degrees of freedom* is often encountered in many other fields (in chemistry, engineering, robotics, etc.). Since our work is very interdisciplinary with various applications, and because technical vocabulary and terminology often presents a barrier in communication of ideas among researchers from different fields, we offer several intuitive descriptions of degrees of freedom. The formal mathematical definition is given in the next section.

Roughly speaking, the degrees of freedom is the number of parameters, that is to say, independent coordinates or measurements needed to describe the position of a framework in the plane or in space [1].

If we consider a single point (joint) moving in the plane, in order to bring this point to any position in the plane, a translation in the x-axis and in y-axis are enough. So, a point in the plane has 2 DOF. Another way to think of this is to coordinatize the plane and see that it takes two real numbers to identify the location of the point (each coordinate changes independently of the other one). Similarly, in three-space a single point (joint) has 3 DOF.

We assume two distinct points P and Q in the plane. P and Q together have 4 DOF. Assume we have connected P and Q with a bar so as to save the distance among them, then we have reduced the total DOF to 3. More specifically, we can bring P to any position in the plane with vertical and horizontal translation (2 DOF). Then we can place Q in any desired position with a rotation about P (1 degree of freedom). So, a single bar (a rigid object) in the plane has three DOF.

Moreover, any rigid framework in the plane with at least two distinct points has 3 DOF: two translations and a rotation. Once the position of any two distinct points of a rigid framework are fixed in the plane, the entire framework is fixed (all other points will be in the fixed positions) [6].

[i] Note that there is an ambiguity in the terminology.

Any bar and joint framework in the plane (with at least two distinct joints) has at least 3 DOF, corresponding to its rigid body motions (generated by two translations and a rotation). We will often refer to the 3 DOF of a rigid body as the trivial DOF. Since trivial DOF are always present in (unpinned or floating) framework, it is a useful practice to ignore the 3 trivial DOF and look for any additional DOF corresponding to non-trivial motions. Flexible frameworks have additional DOF, which are called the internal degrees of freedom of a framework.

Before we give a formal definition, we anticipate that a framework in dimension 2 with more than one joint has *n* internal DOF, and *n+3* total DOF, where *n* is the number of bars that must be added to rigidify the framework. This is generally a good definition, but we will see shortly that we will need an assumption that joints are not in some special geometry. In order to detect rigidity of a framework, it is of clear interest to know how many internal DOF are present. Rigid frameworks have 0 internal DOF.

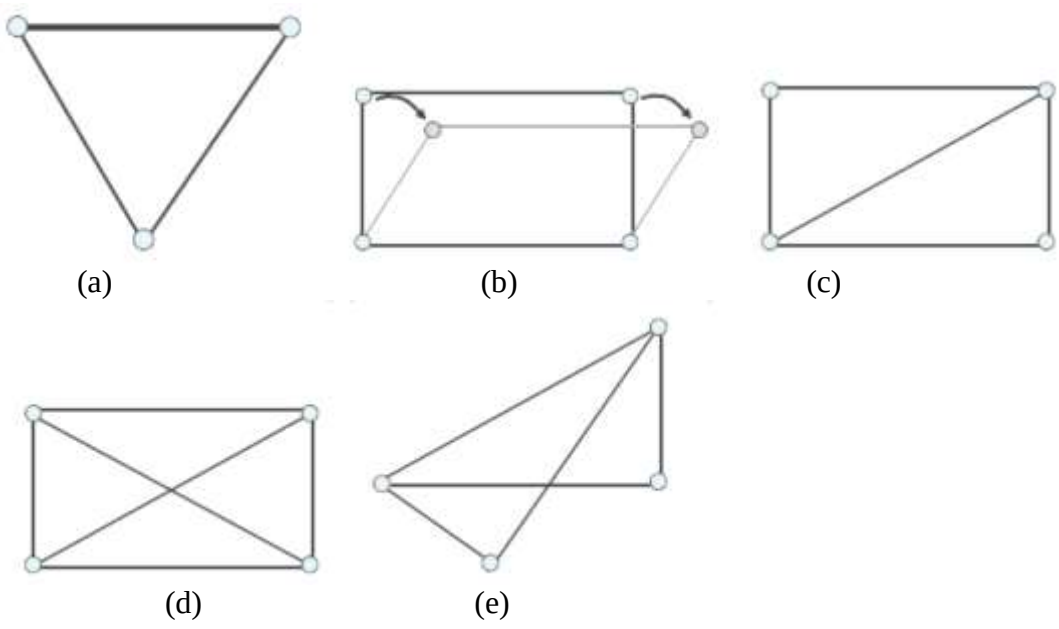


Figure 1.2 :Rigid And Flexible Framework In Th Plane

In the previous figure, we have illustrated some simple bar and joint frameworks. Triangle (a) is clearly a rigid framework. Rectangle (b) is flexible, since it deforms into a parallelogram. The rectangle has four total DOF in the plane, the 3 trivial DOF, and 1 internal degree of freedom. The internal DOF corresponds to the extra motion (deformation) when we fix (hold) the bottom two joints, allowing the top and two side bars to move. On the other hand, a rectangle with a diagonal (extra bar present) (c) is rigid, it has 3 total DOF (0 internal DOF). Adding

another diagonal bar (d) is clearly redundant as the framework is already rigid. Frameworks with redundant edges will be called *stressed.*

Remark. A rigid framework in 3-space, with at least three non-collinear points (joints) has 6 degrees of freedom.

1.3. Formal Definitions, Rigidity and Infinitesimal Rigidity

All the discussion on rigidity so far was informal. In this section we give precise formal definitions and notations. We first state several standard definitions and terms from graph theory.

1.3.1. Graph theory terminology

A graph $G = (V, E)$ has a vertex set $V = \{1, 2,\ldots,n\}$ and an edge set E, where E is a collection of unordered pairs of vertices called the edges of the graph. In more extended form, the vertex set and edge set on G may sometimes be denoted as $V(G)$ and $E(G)$, respectively. We say that two vertices v and w are adjacent (or neighbors) if edge $e=\{v,w\}$ is present in the graph (i.e. an edge e connects a pair of vertices), otherwise they are non-adjacent.

The edge $\{v,w\}$ is said to be incident to vertices v and w, conversely, the vertices v and w are incident to the edge $\{v,w\}$.

To shorten the notation, when no confusion can arise, we may sometimes denote the edge $e = \{v,w\}$ as vw. v, w are called the endvertices (or ends or endpoints) of edge vw. The degree or valence of a vertex v is the number of edges that are incident to v (i.e. the number of edges that have v as an endvertex).

A graph G is complete if all its vertices are pairwise adjacent (i.e. connected by an edge).

A subgraph of $G = (V,E)$ is a graph $G' = (V', E')$, with $V' \subseteq V$ and $E' \subseteq E$, and we simply write $G' \subseteq G$. If $G' \subseteq G$ and G' contains all edges $vw \in E$ with $v,w \in V'$, then G' is called an induced subgraph of G. Alternatively, the subgraph induced or spanned by a set of vertices is the graph consisting of those vertices and all edges that are only incident to those vertices.

The intersection of two graphs G_1 and G_2 is a graph $G=G_1 \cap G_2$, where $V(G) = V(G_1) \cap V(G_2)$ and $E(G) = E(G_1) \cap E(G_2)$. The union of two graphs G_1 and G_2 is a graph $G= G_1 \cup G_2$, where $V(G) = V(G_1) \cup V(G_2)$ and $E(G) = E(G_1) \cup E(G_2)$.

A path P from (or between) v_0 to w_k is a (nonempty) graph $P = (V,E)$, where $V=v_0, v_1,\ldots,v_k$ and $E = v_0v_1, v_1v_2,\ldots v_{k-1}v_k$, where the v_i are all distinct. The vertices v_0 and v_k are called the ends of the path P. The length of a path is the number of edges in the path. For brevity, we may

denote a path using its natural sequence of vertices (i.e. $P= v_0, v_1, \ldots, v_k$). We say that a graph is connected if every pair of its vertices is joined by a path.

A cycle of a graph *G* is a subset of the edge set of *G* which forms a path such that the start vertex and end vertex are the same.

In *directed* graphs, edges are ordered pairs of vertices (i.e. edges get a preferred direction, which is usually identified by an arrow on the graph). We say an edge is directed from an *initial vertex* to a *terminal vertex*. The *out-degree* of a given vertex in a directed graph is the number of edges directed out of that vertex.

An edge is *multiple* if there is another edge with same endvertices. A *loop* is an edge which joins a vertex to itself (i.e. $e =\{v,v\}$).

A graph is *simple* if it does not contain any multiple edges or loops. Multiple edges are permitted in *multigraphs*.

The *multiplicity* of an edge is the number of multiple edges sharing the same endvertices. We are abusing the standard notation for multigraphs and identifying a pair of vertices for edges, although technically each edge gets a unique label. When there exist multiple directed edges between the same pair of vertices, *v* and *w*, such graphs are called *directed multigraphs*.

More details, with illustrations, and other graph-theoretical definitions can be found in any introductory book to Graph Theory (see [6]).

1.4. Formal Definition of Rigidity

We move on to the formal definitions of rigidity that were briefly introduced in the previous section. We continue to use 2-dimensional bar and joint frameworks, as definitions easily extend to 3-dimensional (and higher) frameworks.

We define a 2-dimensional bar and joint framework, or framework in short, as a triple *(V,E,p)*, where $G = (V,E)$ is a simple graph (no loops or multiple edges) and a corresponding configuration $p : V \rightarrow R^2$, which assigns each vertex to a point in the plane. The bars are represented by edges and joints by vertices. For simplicity, we will always assume that the endvertices of every edge in the framework have distinct points (i.e. all edge lengths have positive values). Note that in 3-space, the definition is the same, except $p : V \rightarrow R^3$. For brevity, the framework (V,E,p) is denoted as *G(p)*.

As a convenient abuse of notation we denote the point *p(i)* as p_i, $i \in V$, where in the usual extended form we denote the coordinates of p_i (in 2-dimensions) by (x_i,y_i). A standard and useful practice (which we adapt) is to identify *p* as a single point in R^{2n} $(n = |V|)$ (i.e. $p = (p_1, p_2,\ldots,p_n)$)

A motion (or finite motion) *p(t)* of the bar and joint framework *G(p)* is a family of smooth functions $p(t) = (p_1(t), p_2(t),\ldots,p_n(t))$, $0 \le t \le 1$, such that $p(0)=p$ (i.e. $p_i(0) = p_i$ for all i), and $|p_i(t) - p_j(t)| = |p_i - p_j|$ = constant, for all $\{i,j\} \in$ E and $0 \le t \le 1$ ..... (1)

(eg. Under the motion *p(t)*, the length (Euclidean) of each edge in the framework is kept fixed)

We say that a motion *p(t) of G(p)* is a flex (also non-trivial motion, deformation) if $|p_i(t) - p_j(t)| \neq |p_i - p_j|$ for all t > 0, and some $\{i,j\} \notin E$ (i.e. *p(t)* is not congruent to p(0) = p for all t > 0 { distance between some pair of vertices is changed).

A framework is *flexible* if it has a flex, and is rigid otherwise (has only trivial (rigid body) motions) [5]. More specifically.

In a rigid framework, $|p_i(t) - p_j(t)| = |p_i - p_j|$, for all $i,j \in$ V and t > 0. So, in a rigid framework, every motion *p(t)* preserves the distances of all pairs of vertices (both adjacent and non-adjacent)(for more details see [12] [5] 18] [9,32]).

Finding a solution to this system of quadratic equations is very difficult even for very small frameworks with few edges.

Since rigidity of *G(p)* is difficult to establish, one approach that engineers and mathematicians have used for centuries is to linearize the problem. More specifically, instead of looking for a flex (deformation) directly, one can simplify the quadratic algebra to a more manageable linear algebra.

The *rigidity matrix* notion emerged. The rigidity matrix $R_G(p)$ has |E| rows (one for each edge) and 2|V| columns. For each edge $e \in E$, the corresponding row in the matrix has only four nonzero entries corresponding to the difference in the coordinate values of its two associated incident vertices (see below). The rigidity matrix [i] (in any dimension) has this general form [5]:

$$R_G(p) \quad \begin{matrix} 1 & \ldots & i & \ldots & j & \ldots & n \end{matrix}$$
$$\{i,j\} = \begin{bmatrix} : & . & : & \ldots & : & . & : \\ 0 & \ldots & (pi - pj) & \ldots & (pj - pi) & \ldots & 0 \\ : & . & : & \ldots & : & . & : \end{bmatrix}$$

Consider the graph of a tetrahedron in 2-dimensions (complete graph on four vertices) The rigidity matrix for this framework is:

[i] Maxwell in the 19th century was studying the rigidity matrix.

$R_G(p)$	$v1$	$v2$	$v3$	$v4$
{1,2}	$(p1-p2)$	$(p2-p1)$.	0	0
{1,3}	$(p1-p3)$	0	$(p3-p1)$	0
{1,4}	$(p1-p4)$	0	0	$(p4-p1)$
{2,3}	0	$(p2-p3)$	$(p3-p2)$	0
{2,4}	0	$(p2-p4)$	0	$(p4-p2)$
{3,4}	0	0	$(p3-p4)$	$(p4-p3)$

In the extended form *(where $p_i = (x_i, y_i)$)* this rigidity matrix becomes:

$R_G(p)$	$vx1$	$vy1$	$vx2$	$vy2$	$vx3$	$vy3$	$vx4$	$vy4$
{1,2}	$(x1-x2)$	$(y1-y2)$.	$(x2-x1)$	$(y2-y1)$	0	0	0	0
{1,3}	$(x1-x3)$	$(y1-y3)$	$(x3-x1)$	0	0	$(y3-y1)$	0	0
{1,4}	$(x1-x4)$	$(y1-y4)$	0	0	0	0	$(x4-x1)$	$(y4-y1)$
{2,3}	0	0	$(x2-x3)$	$(y2-y3)$	$(x3-x2)$	$(y3-y2)$	0	0
{2,4}	0	0	$(x2-x4)$	$(y2-y4)$	0	0	$(x4-x2)$	$(y4-y2)$
{3,4}	0	0	0	0	$(x3-x4)$	$(y3-y4)$	$(x4-x3)$	$(y4-y3)$

Here is the figure of the aforementioned tetrahedron.

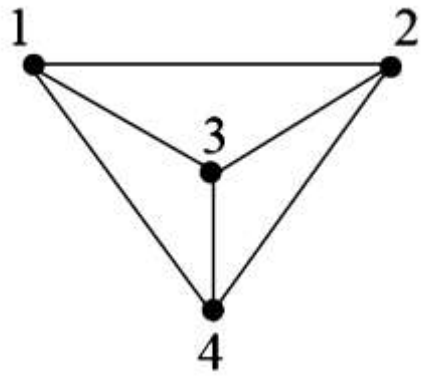


Figure 1.3 Graph Of Tetrahedron, a K_4 Graph.

we are interested in understanding the linearized, infinitesimal motions (i.e. solution $p' = (p'_1, p'_2, \ldots, p'_n)$ of the linear system $R_G(p)p'^T = 0$) of the framework $G(p)$. Infinitesimal rigidity is a natural approximation to rigidity, and their interconnection has been extensively studied (for more details see [9,32]). As we will ultimately rely only on the combinatorial property of rigidity which captures the situations where the two ideas are equivalent (given in next section), we only state the key results along with the standard definitions.

An infinitesimal motion p' is called a trivial infinitesimal motion (or infinitesimal rigid motion) if it has velocities that arise from some congruence [32] (i.e. translations and rotations). If the solution p' is not trivial then p' is called an infinitesimal flex (or infinitesimal deformation).

We say that *G(p)* is infinitesimally flexible if it has an infinitesimal flex, and is infinitesimally rigid otherwise. Since infinitesimal (snap-shot) motions of the framework are solutions of the system of homogenous linear equations (solution space), they form a vector space. We are clearly interested in the dimension of this solution space. The space of trivial infinitesimal motions (in the plane) is of dimension three, generated for instance by two translations and a rotation around an origin [18] .

A set of edges in a framework is said to be independent if their associated rows in the rigidity matrix are linearly independent [12] (i.e. removal of an independent edge decreases the rank by one). An edge is said to be dependent or redundant if removing it from the framework the rank of the rigidity matrix stays the same.

The total degrees of freedom of the framework *G(p)* is the dimension of the solution space of $R_G(p)p^T = 0$ (i.e. the dimension of the space of infinitesimal motions). The internal degrees of freedom of the framework is the dimension of the space of infinitesimal motions minus the dimension of the space of trivial motions (trivial degrees of freedom, i.e. unavoidable solutions). That is, if *G(p)* has at least two vertices, the *internal DOF = total DOF* - **3** *trivial DOF*. An *infinitesimally rigid* framework *G(p)* has 0 *internal* DOF.

Since infinitesimal rigidity can be analyzed using the rigidity matrix (i.e. knowing the size of the solution space), we can make use of the linear algebra tools. A standard result in the linear algebra for a homogeneous system of linear equations connects equations, solutions and variables:

dimension of the solution space = # (number) columns (variables) - # (number) of independent equations (rank).

We can now state a result which says that infinitesimal rigidity of a framework can be determined by simply computing the rank of the rigidity matrix [13,14]

Theorem 1:

A 2-dimensional framework G(p), with |V| > 1, is infinitesimally rigid if and only if the rigidity matrix $R_G(p)$ has (maximal) rank = 2|V| - 3.

Edges (bars) act as constraints on the possible motions of the framework, and intuitively, this says we need 2|V|- 3 independent edges to attain infinitesimal (first order) rigidity (note that 3 DOF of a rigid body are not removed).

For completeness, the result for the 3-dimensional frameworks is :

Theorem 2:

A 3-dimensional framework G(p) with |V| > 2, is infinitesimally rigid if and only if the rigidity matrix $R_G(p)$ has (maximal) rank = 3|V|- 6.

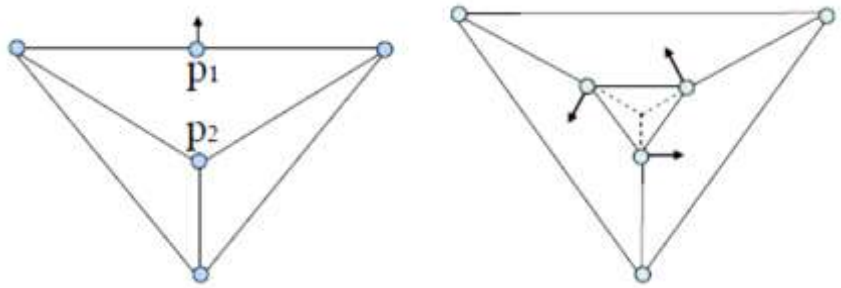
we can simply compute the rank of the rigidity matrix (for instance by Gaussian elimination) and determine if a framework is infinitesimally rigid. Note that any (2-dimensional) infinitesimally rigid framework will have an infinitesimally rigid subframework with exactly 2|V|-3 independent edges (simply discard the redundant edges {edges corresponding to rows of zeros after row-reduction).

The natural question is: what is the relationship between the regular rigidity (finite - continuous motions) and the infinitesimal rigidity (virtual- infinitesimal motion)?

A well-known result in rigidity theory, says that if we have regular flex (finite deformations, non-trivial motion) it follows that we have infinitesimal flex [18].

Theorem 3:

If a framework G(p) is flexible, then G(p) is infinitesimally flexible. Equivalently (by a contrapositive), if G(p) is infinitesimally rigid then G(p) is rigid.



(b)

Figure 1.4 : Nongeneric (Degenerate) Frameworks.

Both frameworks are rigid but not infinitesimally rigid (i.e. rank is less than maximum rank), the arrows represent non-zero velocities. These cases are extremely rare, and occurs because of the special geometry, top three vertices are on a line (a), three lines from big triangle to small center triangle meet at a center point (b).

In some special and extremely rare configurations *p* of the framework, the converse of *Theorem 3* may not be true (i.e. frameworks behave atypically). That is, for some particular *p*, a framework can be rigid and yet not infinitesimally rigid. For example, the framework in Figure 4 (a) is rigid (has only trivial rigid body motions), but it is not infinitesimally rigid (it has an infinitesimal deformation). To see this, we can assign a zero vector to every vertex in the framework, except to vertex p_1 we assign a small non-zero velocity vector (p'_1) (indicated by an arrow) perpendicular to the "chain" (top segment). This (non-trivial) infinitesimal motion (flex) will not distort any edge lengths in the framework at first order, but it will distort the

distance (by an infinitesimal amount) say between p_1 and p_2. One might think of this infinitesimal motion as the vibration of the chain (see[1]).

This type of rigidity can be described as "unstable" or "shaky". Similarly, in the previous figure, we can assign zero velocities to the vertices of the large triangle, but the velocities assigned to the vertices of the small triangle correspond to an infinitesimal flex (non-trivial infinitesimal motion) (i.e. hold large triangle and slightly rotate small triangle about the intersection point of the three lines extending the edges joining the two triangles) These degenerate (or singular) cases like the ones in Figure 4 are fortunately very uncommon, and only occur because the framework has special geometry.

The special configurations *p* of a framework that lead to this type of atypical behavior in rigidity are often referred as degenerate (singular, non-generic) configurations. For instance, in Figure 4 (a) this occurs because the top three vertices are collinear). In fact, if we randomly pick the positions of the vertices of the framework (configuration *p*), it is probability 0 that the framework will be in a degenerate configuration (see [12]). We are interested in the generic configurations (not degenerate), which occur for almost all configurations (with probability 1 when randomly chosen) [15].

Loosely speaking, a configuration *p* of a framework is generic if we can slightly perturb the coordinates without altering any of its rigidity properties [18]. This can be explained and defined more precisely again from the rigidity matrix. If we vary the configuration *p* of a framework (for fixed graph), the rank of the rigidity matrix may change. For instance, if in the framework in Figure 4 (a) the three top vertices are slightly perturbed so they are no longer collinear, then the rank of the rigidity matrix will be increased by one (i.e. from 6 to maximum rank = 2(5) - 3 = 7 on five vertices, and the framework would become infinitesimally rigid by *Theorem 1*.

We say that the generic configurations *p* are those configurations which achieve the maximum possible rank of the rigidity matrix on all subgraphs (and configurations that do not give maximum rank are non-generic, singular)[i]. A framework *G(p)* with a generic configuration *p* is often referred to as a *generic framework* (sometimes called *regular*).

We state this important result due to Gluck (1975) [7] which was also formulated in [6] that connects rigidity with infinitesimal rigidity for generic configurations (almost all configurations) of a framework.

Theorem 4:

[i] a configuration *p* is *generic* if all its vertex coordinates are algebraically independent over the rationales.

For a generic configuration p, a framework *G(p)* is rigid if and only if *G(p)* is infinitesimally rigid.

Equivalently (by the contrapositive), a generic framework *G(p)* is flexible if and only if *G(p)* is infinitesimally flexible.

This result assures us that (for generic frameworks) tracking infinitesimal rigidity is actually tracking regular rigidity (corresponding to finite motions). Note that this result also tells us that if the framework is rigid for one generic configuration (realization) of the framework, it is rigid for any other generic configuration (i.e. almost all configurations) [35].

1.5.Counting for rigidity in plane graphs and Laman's Theorem

The significance of Gluck's result *Theorem 4* is that rigidity and flexibility of generic frameworks is entirely a property of the graph *G = (V,E)*. In other words, the geometry (i.e. configuration *p*, edge lengths) of the generic framework *G(p)* is not important in deciding whether the framework is *rigid or flexible*. That is to say, generic rigidity is a *combinatorial* property. We will step back from our discussion about rigidity of frameworks and strictly talk about the rigidity of a graph *G=(V,E)* (graph rigidity can be taken to be synonymous with generic rigidity). It is certainly intriguing to know that rigidity is a property of a graph, but it is not clear how to *efficiently* check if the graph *G=(V,E)* is rigid.

Since (generic) rigidity is only a property of the graph, it would be desirable to have a pure graph theoretical (combinatorial) way of deciding rigidity of graphs (i.e. by counting vertices and edges in the graph and its subgraph). We will shortly offer an argument on how to decide (in combinatorial way) if the graph is rigid. Most of this discussion can be anticipated from Theorem (1) since the rank of the rigidity matrix offers important clues on what the counting condition for rigidity may look like. The rigidity matrix and rank computation is important as it elucidates rigidity behavior of the framework and provides us with precise mathematical definitions of rigidity. However, since direct computation of the rank of the rigidity matrix is not practical for our purposes and in applications (i.e. distributed implementation), we will now switch the vocabulary and translate some definitions that were stated in terms of the rigidity matrix to pure graph and combinatorial statements to make them more effective. This will match more closely the vocabulary of the graph rigidity literature, and in rigidity-based algorithms (i.e. pebble game), and will prove to be central as most of our work in this dissertation relies on combinatorial rigidity and the pebble game algorithm.

We say that a graph is *minimally rigid* or *isostatic* if it is rigid and after any one of its edges (but not vertices) is removed, the graph becomes *flexible*. The main problem in combinatorial

rigidity is to find a combinatorial condition with a fast algorithm for the graph to be minimally rigid.

Flexibility in the graph occurs due to unconstrained internal degrees of freedom. Since a single vertex in the plane has 2 DOF, a graph with |V| vertices has at most 2|V| DOF (possible independent motions); it has exactly 2|V| DOF when no edges are present. Each edge (constraint) eliminates at most a single DOF so in order to eliminate all the internal DOF (recall that 3 trivial DOF are always present), then one could anticipate that at least 2|V| - 3 (|V| > 1) edges are necessary for rigidity of a graph.[27]

Having 2|V|-3 edges is clearly not sufficient, as the edges could be crowded among only a select few vertices, leaving the other vertices unconstrained (see the next Figure (a)). The goal then, is to have *2|V|- 3 well-distributed edges*. That is, in graphs with 2|V|-3 edges, no subgraph on *V'* vertices should have more than its fair share of 2|V'|-3 edges (see next Figure (b), This criterion is sometimes called the Laman condition, due to Laman [6], and the graphs (subgraphs) that satisfy this condition are known as Laman graphs (subgraphs)). If any subgraph $G(V')$ has more than 2|V|-3 edges, then some of these edges are redundant. Non-redundant edges are independent.

We say that a graph is *stressed* if it has at least one redundant edge. A graph is said to be *independent* if all its edges are independent. A graph is said to be *redundantly rigid* if removal of any edge leaves the graph rigid (next Figure (c)). Clearly *redundantly rigid* graphs are *stressed* (any edge in redundantly rigid graph is redundant). Redundantly rigid graphs (or subgraphs) are sometimes also called *overconstrained* [11,15]. Removing a redundant edge from the graph does not change rigidity (i.e. number of DOF in the graph before and after its removal is the same). It is the independent edges (i.e. well-distributed edges) that eliminate the degrees of freedom from the graph, so the presence of 2|V|-3 well distributed edges should be sufficient for rigidity. This basic intuition is correct in 2-dimensions and it has been stated and used since the time of Maxwell in the 19'th century [10]. It was finally confirmed in the following famous theorem in rigidity theory, due to Laman (1970) [6]:

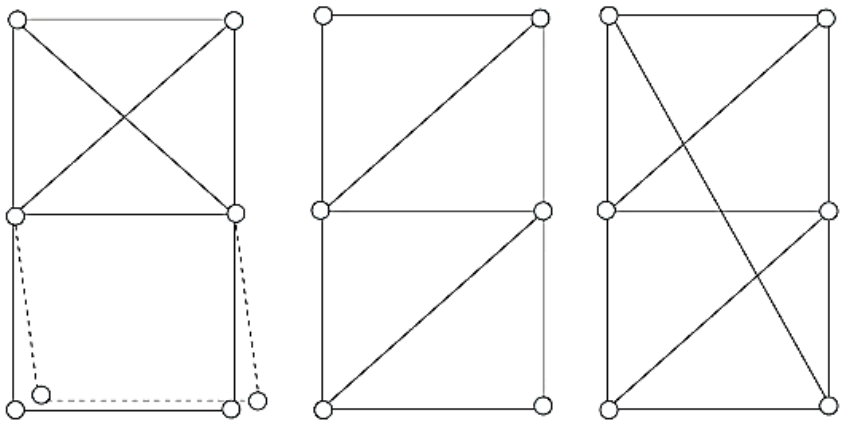
Theorem 5:

Laman said : The edges of a graph G = (V,E) are independent in 2 - dimensions if and only if no subgraph $G' = (V', E')$ has more than 2|V|-3 edges (|V| >1).

Corollary 1: A graph G = (V,E) with 2|V|-3 edges (|V|>1) is minimally rigid in 2-dimensions if and only if no subgraph $G' = (V', E')$ has more than 2|V|-3 edges, (|V| > 1).

Laman's Theorem is the key graph theoretical result which completely characterizes minimally rigid graphs in 2-dimensional space. Several other equivalent characterizations (see

Theorem 6) have since been discovered [12], and some of these play crucial roles in finding polynomial time algorithms for testing rigidity of graphs, such as the pebble game algorithm, which will be discussed in the next chapter. We will refer to the Laman's counting condition as Laman's count or a "2|V|-3 count".



(a) (b) (c)

Figure 1.5: 2|V|-3 Edges Not Sufficient

Capturing well-distributed (i.e. independent) edge is a key step in characterization of rigidity. Both graphs in (a) and (b) have the required minimum number of edges for rigidity, 2(6)-3=9 edges. However, in (a) the edges are not well distributed (independent) so the graph is not rigid. The subgraph induced by the top four vertices has more edges than required (i.e. overconstrained), 2(4)-3 = 5 edges are required, but it has 6 edges, which means that one edge is wasted (redundant). In fact, any one of the six edges in that subgraph is redundant (i.e. it is a redundantly rigid subgraph). In (b) all edges are well-distributed (i.e. independent), so the graph is minimally rigid. In (c) the graph is redundantly rigid, removal of any edge leaves the graph rigid.

Let us briefly illustrate the power and elegance of this theorem. In Figure 7 we have several different graphs (in 2-space) on same sets of vertices. We apply the Laman's theorem, by simply counting vertices and edges in the graph and its subgraphs. Each graph has seven vertices, so the graph will be minimally rigid if we have 11 edges (2x(7) - 3 = 11), and all of the edges are well-distributed (independent).

The graph in Figure 7 (a) has 11 edges and no subgraph of V' vertices has more than $2|V|-3$ edges, guaranteeing its 11 edges are independent. Therefore, from Laman's theorem this graph is minimally rigid (isostatic). In Figure7 (b), we have added an additional edge to the graph, so the graph is rigid (but not minimally rigid). This graph is not redundantly rigid since removal

of some edge at a 2-valent vertex makes the graph flexible. In Figure 7 (c) we have an example of a flexible graph. This graph has 11 edges, but one subgraph (indicated by blue edges and its endvertices) has too many edges and does not satisfy the Laman count. All edges in this subgraph are redundant. Finally, in Figure 7 (d) the graph is clearly flexible as it has less than 11 edges. These kind of graphs are sometimes referred to as underconstrained graphs, as they lack the minimum number of needed edges.

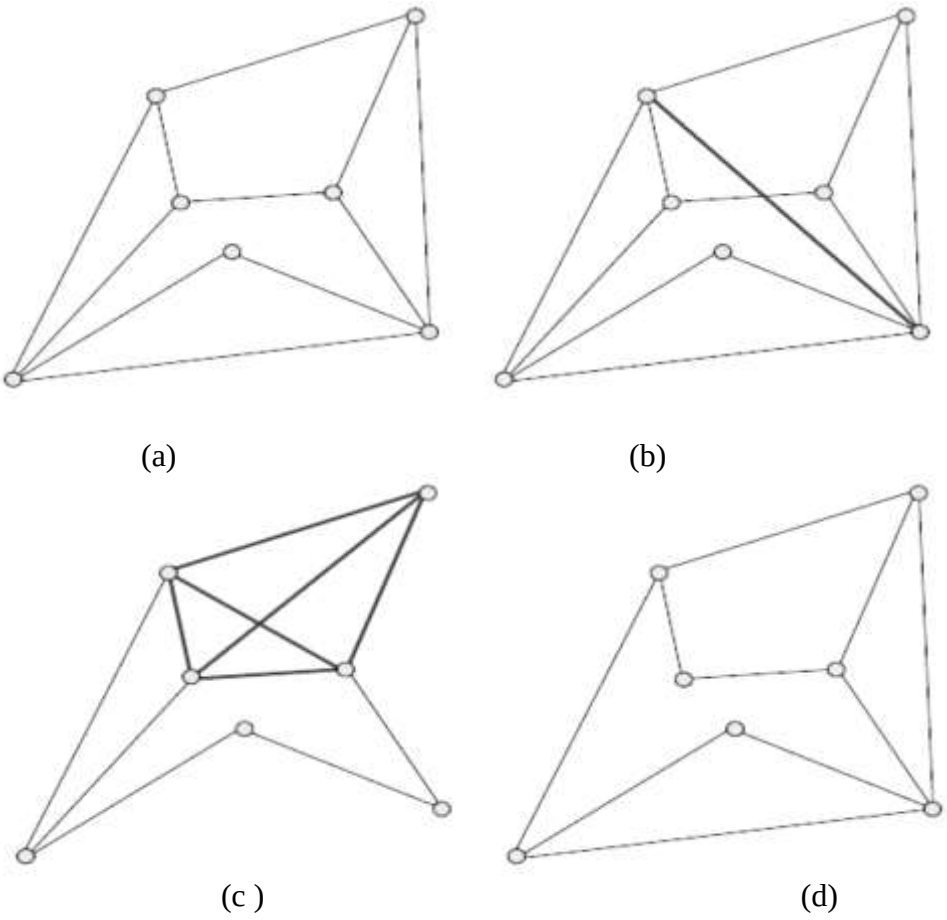


Figure 1.6: Minimally Rigid (Isostatic) Graph

(a) as it satisfies the Laman's counts. Extra edge is added (b), so the graph is rigid but not minimally rigid. The graph in (c) has the minimum number of edges, but the blue subgraph violates Laman's count (i.e. has redundant edges), so it is flexible. This graph illustrates that even flexible graphs can have stress (redundant edges), although it is localized in a certain subgraph. In (d) the graph does not have enough edges, so it is flexible.

In Figure 8 we have a complete bipartite graph $K_{4,4}$ (see [10] for definition). This graph is rigid, in fact it is redundantly rigid (remove any edge and it remains rigid). By overall 2|V|-3

count, it is overconstrained by two edges. This graph illustrates that graphs do not need triangles to be rigid in the plane.

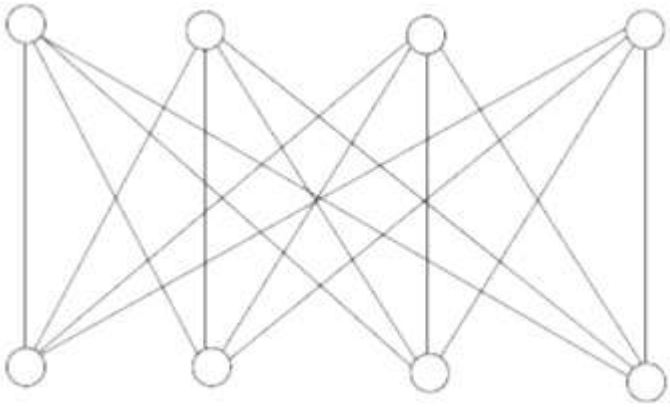


Figure 1.7: A Rigid Graph with No Triangles.

Computing the rank of the rigidity matrix is a significant improvement over the complicated quadratic algebra, but Laman's Theorem is even more powerful.

However, in its original form it still leads to a poor algorithm, as it requires counting the number of edges in every subgraph, of which there are an exponential number [19]. The graphs in Figure 6 were small enough that we could visually inspect the Laman's 2|V|-3 count. The natural next step, which is particularly important for applications, is to find a fast and efficient algorithm that can check the 2|V|-3 count in the graph and subgraphs. A standard practice in combinatorial rigidity theory is to look for equivalent characterizations that capture the counting conditions.

As we said earlier, Laman's Theorem has several equivalent formulations. One particular form is [5]:

Theorem 6

For a graph $G=(V,E)$ having m edges and n vertices, the following are equivalent:

(i) The edges of G are independent in 2-dimensions.

(ii) For each edge $\{i,j\}$ in G, the graph formed by adding three additional edges $\{i,j\}$ has no V' vertex subgraph with more than $2|V|$ edges.

This formulation was initially used by Hendrickson [5], extending the work of Sugihara [30] (as a bipartite matching algorithm) and later revised by Jacobs and Hendrickson [19], and developed into a particularly intuitive and very efficient algorithm, *the pebble game algorithm*.

The basic idea behind the pebble game algorithm is to grow a maximal set of independent edges one at a time by matching them to the DOF in the graph (pebbles).

A new edge is added if it is determined to be independent of the existing set. If 2|*V*|-3 independent edges are found, then the graph is rigid. The pebble game gives a unique and visually appealing way to test edges for independence (track the count), and outputs several key rigidity properties.

In dimension 3 and higher, the rigidity evaluation of bar and joint frameworks is very complicated. Unfortunately, for all of the power that Laman's counts provides in the plane, the 3|V|-6 counting conditions are *not sufficient* for minimally rigid graphs.

The graph of the "*double banana*" in the next Figure is the classical counterexample. This graph satisfies the counts; it has the exact required number of edges (3.(8)-6 = 18) no subgraph has more than 3|*V*|-6 edges connecting |*V*| vertices. In the context of the previous discussion, all eighteen edges are *well-distributed*, yet the graph is still flexible. Each banana (a rigid subgraph) is made of two tetrahedra sharing a triangle. several other examples have also been generated [22].

Since many important applications are found in dimension 3, the solution to this problem would have great practical significance. We now switch our discussion to combinatorial rigidity of different structures which do have necessary and sufficient counting conditions.

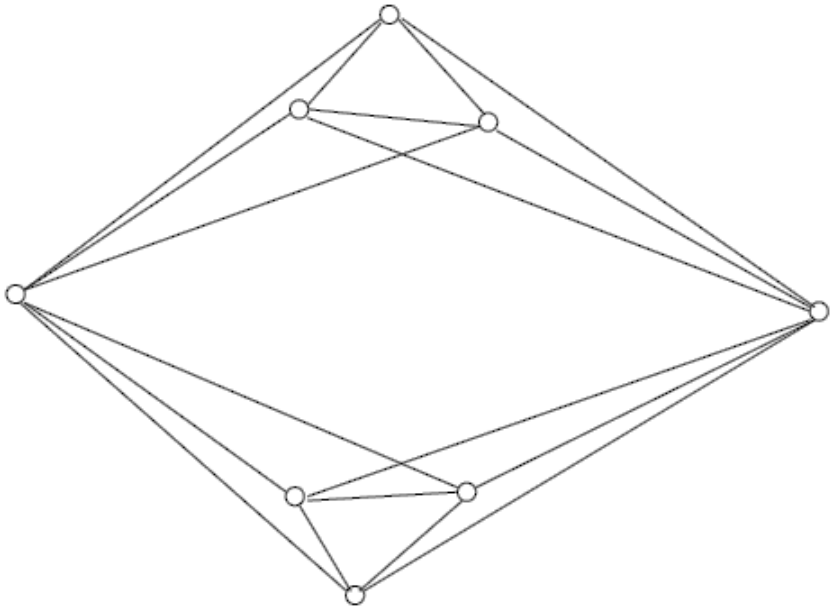


Figure 1.8 A double banana graph

shows that Laman type of counts are not sufficient in 3-dimensions. This graph has 18 well-distributed edges, but it is still flexible.

1.6.Rigidity in Body-Bar frameworks

Why these structures?

So far, we have seen that Laman's Theorem provides both necessary and sufficient counting condition for 2-dimensional bar and joint (generic) graphs, with lack of such counting characterizations in dimension 3. We now turn to a different types of structures, that are generally called body-bar frameworks (or structures) and more importantly include a special subclass of structures, called molecular structures.

A surprising result is that these structures have rich and complete combinatorial (counting) characterization (i.e. Laman type of theorems) of minimal (generic) rigidity in dimension 3, with very intuitive and well defined pebble game algorithm. Moreover, these structures have a well-developed theory and number of strong conjectures in very difficult area in rigidity - the global rigidity [5].

We should emphasize that we are only interested in the pure combinatorial results which capture the (generic) rigidity behaviour. For background purposes and wider context, we will give a brief introduction of the general body-bar and closely related body-hinge frameworks and their combinatorial results.

The nice feature about the molecular structures is that they have an equivalent form in the special class of bar and joint frameworks (bond bending), where the $3|V|-6$ counts do work on the (generic) underlying graph [14,17,38]. Our primary interest in the molecular rigidity models is because of their uses in applications. In particular, molecular structures are used to mathematically model molecules and together with the corresponding *pebble game algorithm*.

Recall from the earlier discussion in this chapter that a rigid structure (rigid body) with at least three non-collinear points (joints) has 6 degrees of freedom in 3-dimensional space (i.e. trivial rigid body motions). In 3-dimensional bar and joint frameworks, each joint (vertex) is a point with 3 DOF, and then we place a maximum of a single bar (edge) between some pairs of joints, which act as distance constraints. In contrast, in body-bar frameworks [36], a body in 3D is a fully dimensional rigid object (i.e. has 6 DOF). Some pairs of bodies are then connected by multiple bars, where each bar preserves the distance between the two attaching points (joints) which lie on different bodies. The bodies are free to move preserving the distance between any two attaching points (joints) connected by a (rigid) bar. Of course, all joints on a same rigid body move together with the body as a single rigid object and have fixed distances between them. As the bodies move, if they preserve the distance between all pairs of points belonging to different bodies, then the body-bar framework is rigid, otherwise it is flexible.

The body-bar framework may at first be perceived as an unusual structure, although they are very familiar in the robotics and engineering community. Some types of linkages and robotic mechanisms have the structure of a body-bar framework.

One famous example is the Stewart platform [29], which is two bodies joined by six bars, which have many important applications (for instance in flight simulations).

We now state one of the remarkable results, due to Tay [29], which naturally extends the Laman's theorem as a both necessary and sufficient counting (i.e. graph property) characterization of rigidity of the (generic) body-bar frameworks.

Theorem 8 :

Known as Tay's Theorem[31], A 3-dimensional (generic) body-bar framework *G(p)* on the multigraph *G=(V,E)* is minimally rigid (isostatic) if and only if it has 6|V|-6 edges and for every nonempty subset E' ⊆ E with vertices *V'*, $|E'| \leq 6|V'|-6$.

As is the case with Laman's 2|V|-3 counting condition, directly applying Tay's $6|V|-6$ counts on the multigraph would lead to a poor algorithm, as we would need to count the number of edges on all subgraphs. We now turn to the pebble game algorithm and its various extensions which can efficiently track these counts.

1.6.The pebble game Algorithm for evaluating rigidity

1.6.1.Overview

The pebble game algorithm was initially introduced by Jacobs and Hendrickson [19] as an efficient and visually appealing algorithm that can track the Laman's 2|V|-3 count in the graph (i.e. rigidity of generic bar and joint 2-dimensional frameworks). This extended Hendrickson's work on bipartite matching and unique realizable graphs [5]. Mike Thorpe's group has extensively applied and revised the pebble game algorithm initially on the flexibility-rigidity predictions of amorphous materials such as glasses (glass-networks) and then to proteins which has resulted in the development of program FIRST [4, 18,33]. In another thread, Lee, Streinu and others have mathematically verified many useful properties of the pebble game algorithm [20,21]. We will describe the pebble game algorithm (sometimes we will just say *pebble game* for brevity), illustrate it on a few examples and highlight important properties.

1.6.2.The Pebble Game Algorithm

We have seen, according to Laman condition, a graph is rigid if it contains a Laman graph, but taking Laman’s condition literally leads to poor algorithm, as it involves checking all subgraphs, efficient and intuitive algorithm exists, based on counting degrees of freedom. So, how to test Laman condition quickly? By *The pebble game* algorithm. In this section, we will describe in detail the pebble game algorithms.

The 2|V|-3 Pebble Game Algorithm description

Input: A non-directed graph G = (V, E).

Output: A directed graph (Rigid or flexible).

Steps of the Algorithm

Initialize Ind(G) [independent edge set] and *R(G)* [Redundant edge set] to an empty set of edges.

Assign two pebbles on each vertex of *G*.

Test the edges of *E* in an arbitrary order.

Step1: Until every edge *e=uv* in the graph *G* has been tested,

take any untested edge *e*, and go to *step 2*.

Otherwise go to *step 3*.

Step 2*: Count* the number of *free pebbles* on the endpoints of *e*, say vertex *u* and *v*.

(i). If the vertices u and v have **four** (two for each) free pebbles,

Add the edge in the independent edge set,

Direct the current edges from *u* to *v*,

Decrement the account of *u* by 1.

Return to step 1.

(ii). Else, *Search* for free pebbles originating from *u* and *v* by following only

the directed (covered) edges in the partially constructed directed graph *Ind(G)*:

(a). If the free pebble is found on some vertex *w* at through a directed

path *P* (which starts at *u* or *v*):

Reverse the entire path *P*, until a free pebble appears on the initial vertex (*u* or *v*) of the path *P* (i.e. *w* loses one free pebble, and *u* or *v* gains one free pebble).

Return to step 2.

b. Else, we could not find the fourth free pebble: the edge is declared redundant (could not be covered by the pebble).

Add e into *R(G)* (redundant edges).

Return to step 2.

Step 3**:** Once all edges have been tested, **Stop**.

Output the sets: I(G) and R(G) = E - I(G).

When the algorithm is finished, *I(G)* is the maximal independent set of edges (edges that are covered by pebbles). *R(G)* is the set of redundant edges (edges that were not covered by a pebble). The Total DOF in a graph = number of remaining free pebbles.

At the end of the pebble game there will always be *three* remaining free pebbles in the graph (indicating the 3 trivial DOF-rigid body motions). We started with 2|V| free pebbles (prior to covering (pebbling) of any edge), and each independent edge used up one free pebble (DOF). So, once all edges have been tested, the total remaining DOF in the graph is 2|V|-|*I(G)*| = total remaining free pebbles. To get the internal (non-trivial) DOF we take the total remaining free pebbles and subtract 3 (trivial DOF), that is internal DOF = 2|V|-|*I(G)*|-3. Since pebbled (covered) edges become directed, the graph on *I(G)* is directed. If *total independent edge* number = 2|*V*|-*3*, then the graph is Rigid.

The centralized pebble game Algorithm of Jacobs and B. Hendrickson can be depicted as follows [25]:

```
1:  procedure PEBBLEGAME (G = (V, E))
2:  Assign each vi   two pebbles, ∀i ∈ I
3:  E* ← {}
4:          for all (i, j) ∈E do
5:                  Quadruple (i,j) over G
6:                  Search for free pebbles, originating from u_i  and v_j
7:                  if found then
8:                          Rearrange pebbles to cover quadrupled (i, j)
9:                          Expand independent set, check rigidity:
10:                         E* ← E* ∪ (i, j)
11:                                 if |E*|= 2|V|-3 then
12:                                 return E*
13:                                 end if
14:                 end if
15:         end for
16: end procedure
```

1.6.3.The Pebble Game Illustration For Testing Rigidity

Here is an illustration of the pebble game Algorithm on a graph of four vertices (*a,b,c* and *d*) where *links* between *a and (b,c,d), link* between *(c and d),* and another *link* between *(b and d)* as depicted in the following figure, where the input is the previous undirected graph, and the output is a rigid directed graph, and here is the steps of the algorithm, this example is a demonstration presented by KINARI (KINematics And RIgidity)[41].-

Input Graph

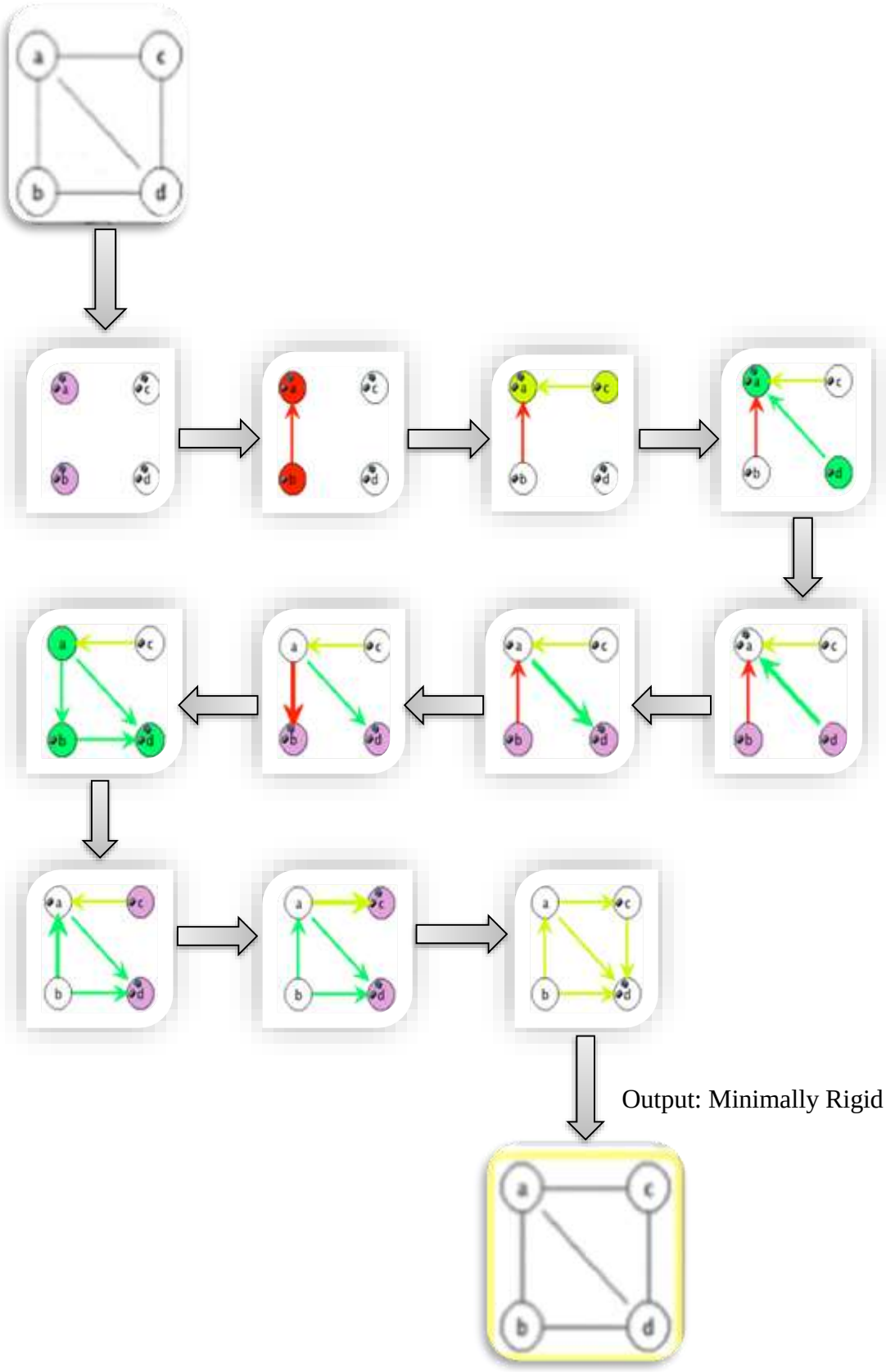


Figure 1.9 2*n*-3 Pebble Game Algorithm Illustration

In this chapter, we have provided a detailed explanation of the rigidity theory and its basic notions, outlined several important properties of graph theory. We have also described the

pebble game algorithm which is considered as the best proposed solution for checking rigidity, and illustrated its execution steps.

We now turn to the important applications of this algorithm where we mention some relevant application domains.

CHAPTER 2 :

THE INTEREST OF THE RIGIDITY THEORY

2.1.Overview

Rigidity and flexibility are two notions that have imposed themselves in many fields of science such as Biology, Medicine, Chemistry, Physics, Architecture, Automatism, Mechanic, computer sciences and so many others. In this chapter we will see the most famous application areas of Rigidity.

2.2.Rigidity Theory in Molecular Biology

Molecular biology is a branch of science concerning biological activity at the molecular level. The field of molecular biology overlaps with biology and chemistry and in particular, genetics and biochemistry. A key area of molecular biology concerns understanding how various cellular systems interact in terms of the way DNA, RNA and protein synthesis function, as well as the regulation of these interactions.

The Flexibility and Rigidity in molecules is used based on mathematical models and the related algorithms. These Mathematical models are used to describe the snap-shot flexibility in molecules from a single snap-shot of a molecule. The molecules are modeled by a body-bar framework which replaces the bar and joint model in this field.

In this model, where each atom with locked bonds is a fully rigid body with 6 degrees of freedom. A covalent bond is a rotatable hinge leaving one degree of freedom between the rigid bodies, imposing constrains between the atoms. A double bond is non-rotatable locking all the six degrees of freedom of the two atoms. Additional type of constrains can also be modeled (see next figure [40]).

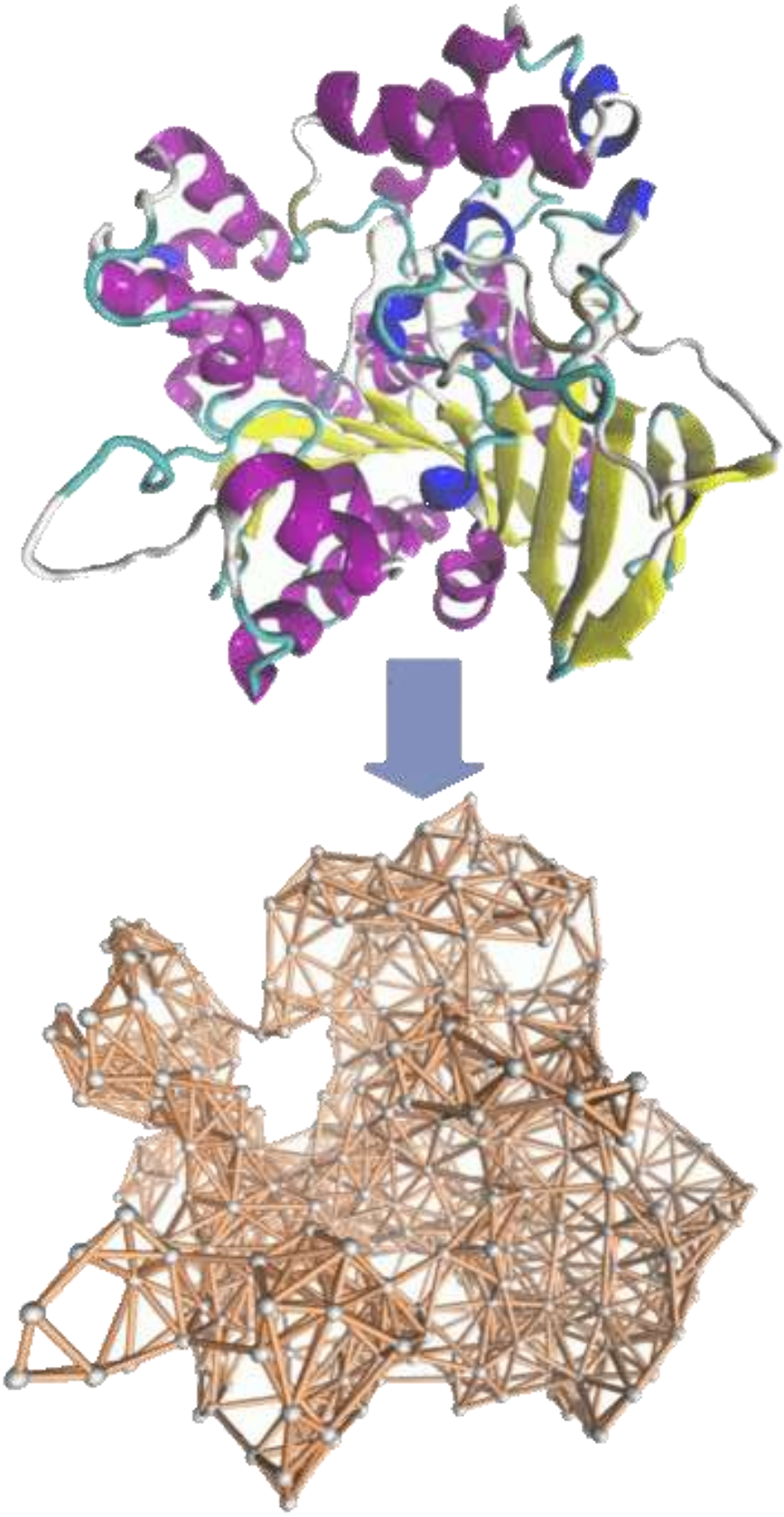


Figure 2.1 Protein modeling

For the body and bar structure a new matrix can be generated like the matrix for bar and joint. A body in 3D has 6-degrees of freedom. These motions can be represented by 6D vector *Si*. An individual constrain will fix the distance between two points on two distinct bodies. This rigidity matrix will have |E| rows and 6|V| columns and its rigidity can be captured by a simple counting property.

A fast algorithm that helps in this count is called the Body-Bar pebble game. It is the same as the pebble game only in the number of independent edge. Total independent edge $E = 6|V|-6$. So, it is *The 6|V|-6 Pebble Game Algorithm*(see next figure).[27]

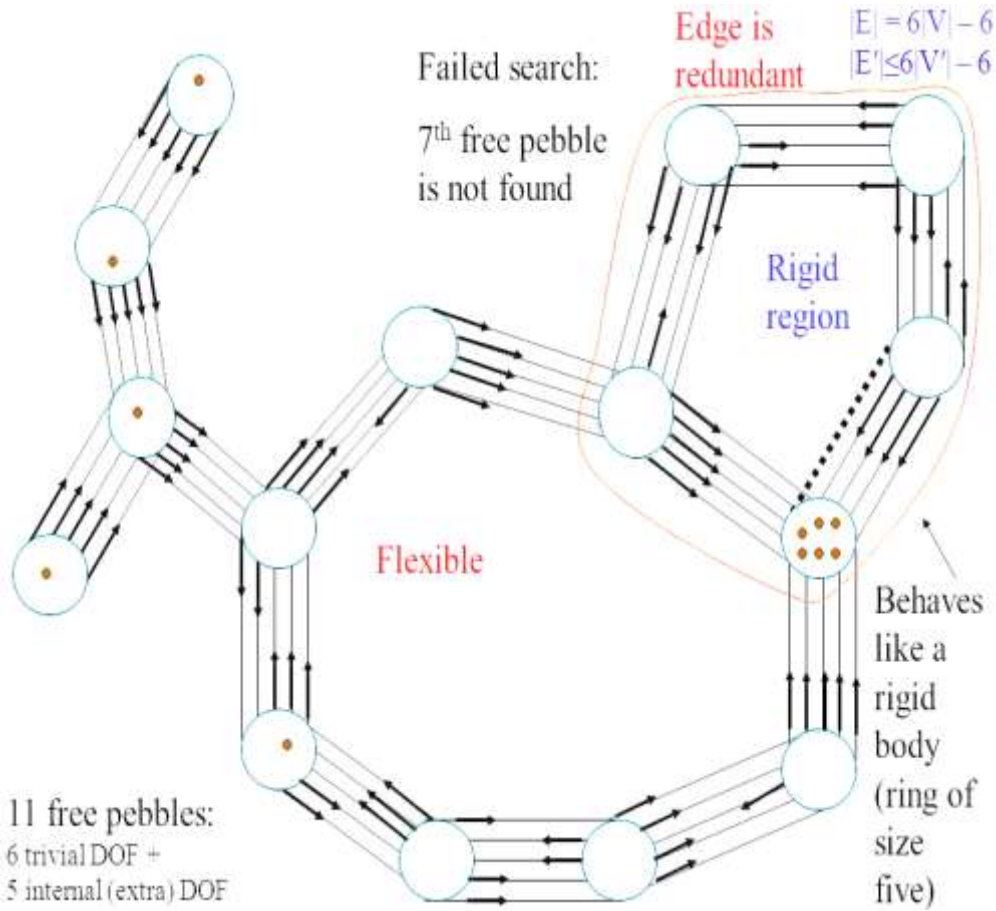
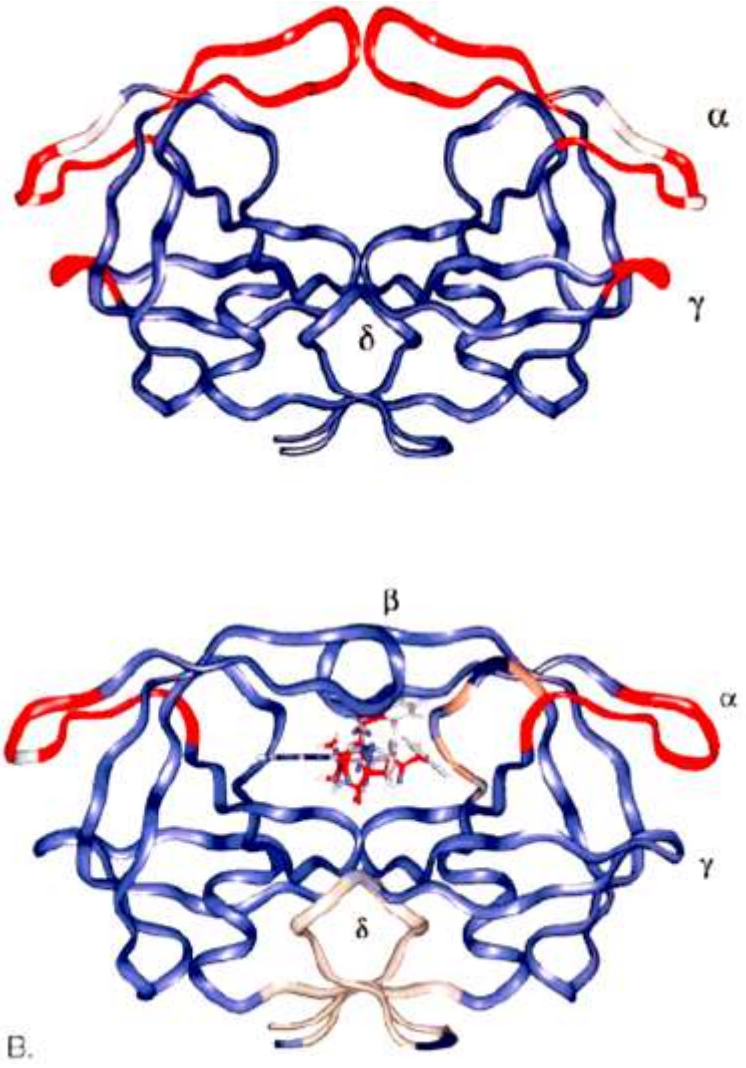


Figure 2.2 6|V|-6 pebble game snap-shot

In this regard, Donald J. Jacobs et al, studied Protein Flexibility Predictions using graph theory, they showed the difference of rigidity and flexibility of the same HIV protein in its two states, the open form of HIV protein and the closed one, they used FIRST program (Floppy Inclusions and Rigid Substructure Topology) and the output of |6V|-6 on HIV protease found as in the next figure [16].



The open Form

The closed Form

Figure 2.3 HIV protease flexibility (rigidity) in two forms (the open and the closed)

2.3.Rigidity theory in Localization

The applications of graph theory in sensor network localization is one of the recent and curious study fields, this application aims to know and determine the sensors positions in a given graph and the distances between some pairs of sensors.

One approach to localize a large network is to divide the network into smaller subnetworks whereby each subnetwork is localized in its own coordinate system. Many algorithms which use linear algebra methods for computing the actual sensor positions given the local solutions of a collection of subnetworks are present. The Graph Rigidity theory is also used to characterize collections of subnetworks for which the algorithms are applicable [39].

In this regard, many studies and applications were done, and the recent ones are in auto-localization of sensor networks and robots etc.

2.4.Rigidity theory in Computer-Aided Design (CAD)

Computer-aided design (CAD), also known as computer-aided drafting and design (CADD), is the use of computer technology for the process of design and design-documentation. CADD software, or environments, provide the user with input-tools for the purpose of streamlining design processes; drafting, documentation, and manufacturing processes. CADD output is often in the form of electronic files for print or machining operations. CADD environments often involve more than just shapes. As in the manual drafting of technical and engineering drawings, the output of CAD must convey information, such as materials, processes, dimensions, and tolerances, according to application-specific conventions [44].

CAD is an important industrial art extensively used in tensegrity structural applications. Tensegrity or tensional integrity is a property of structures with an integrity based on a balance between tension and compression components. It is also widely used to produce computer animation of such models, origamizers to design origami structures and folding robot papers. It is also used in Architecture, when designing bridges, towers, needle towers and buildings. (see next figure for more details).

Here is an example of a CAD of a Bridge design where the rigidity theory is an essential study factor.

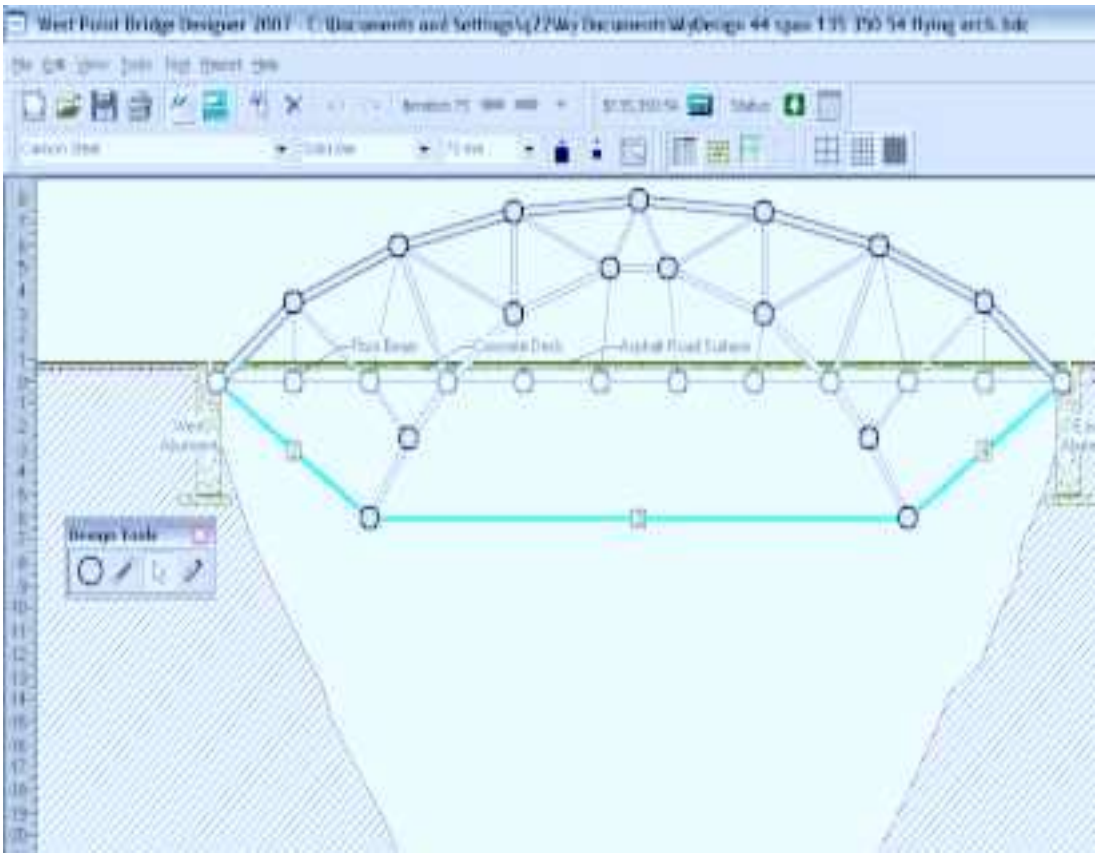


Figure 2.4 : Rigidity Theory in CAD of Bridge Design in Architecture

And this is another application of rigidity theory in tensegrities where the design is used by CAD.

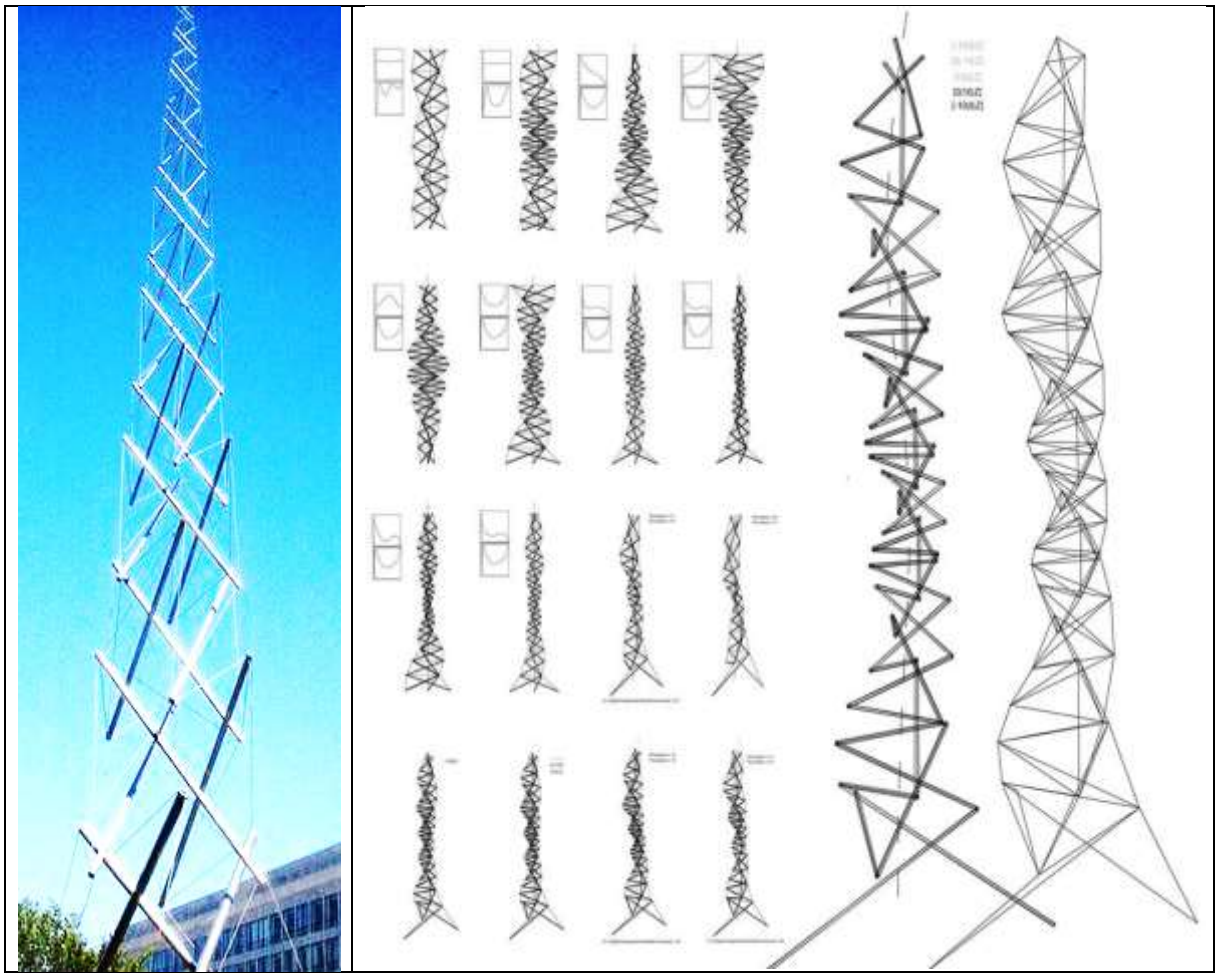


Figure 2.5 : Rigidity Theory in CAD of Needle Tower

Here is another application in a skyscraper : the Street View with embedded Parametric Modelling of the Tensegrity Tower by Gustav Fagerström [45].

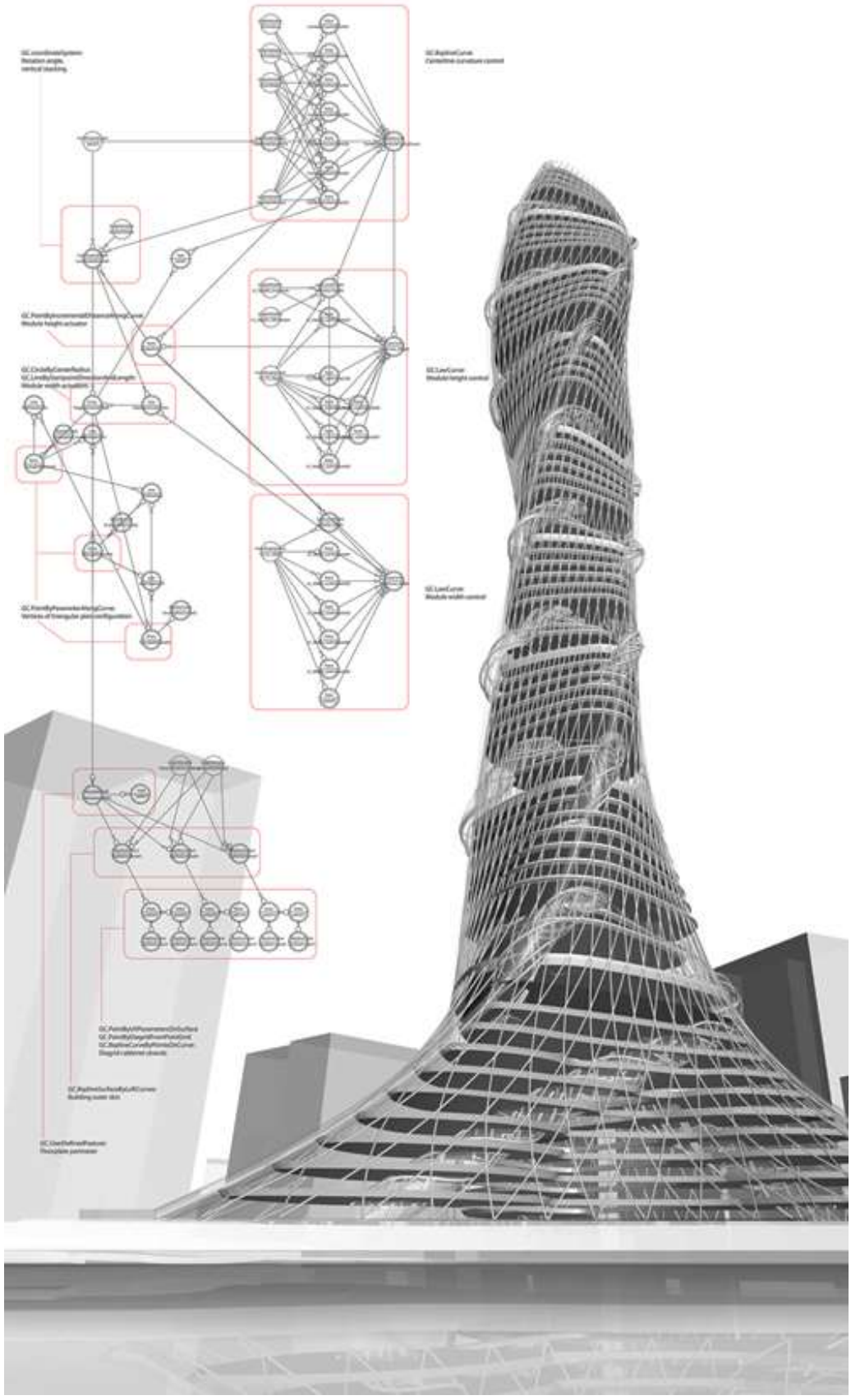


Figure 2.6: Rigidity in CAD for design a Skyscraper

One of the curious application area of rigidity theory is origami, and origamizers are the CAD programs designed for this aim, the next figure shows that.

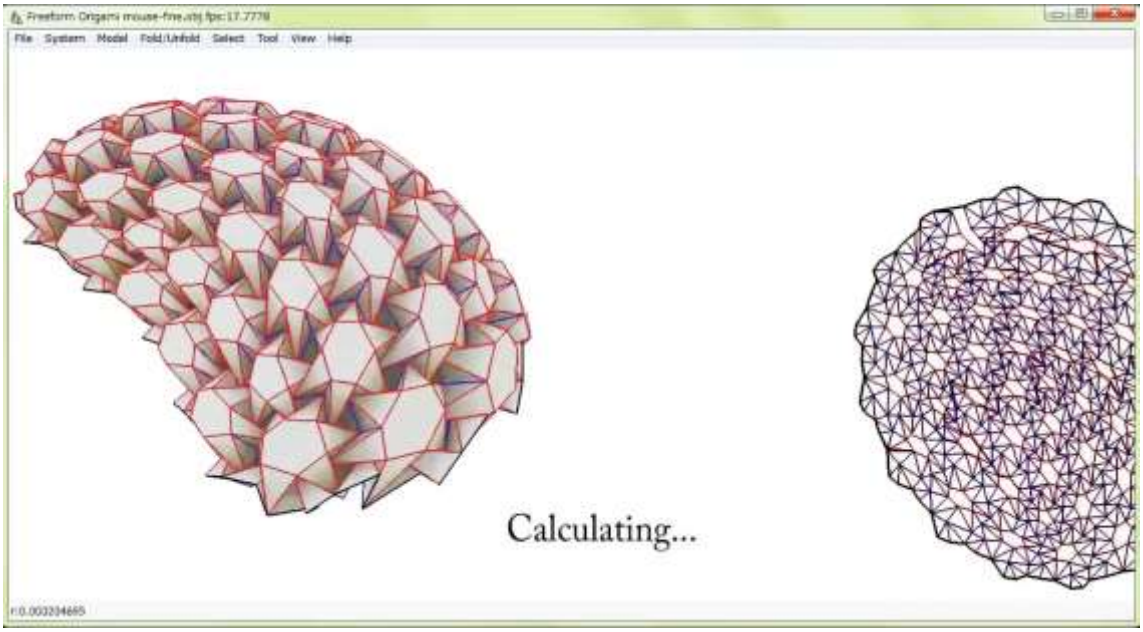


Figure 2.7: Rigidity Theory in CAD of Origami

There are many other applications such the famous folding Algorithm which makes any 3D structure from ONE sheet of paper. So, using this algorithm the robots will be multi-forms and have different shapes.

In this chapter, We have seen some of the major applications in which rigidity theory has a decisive role in deciding whether the strucutre is rigid or flexible such as buildings, towers, molecular biology, glass abd bar structires.Yet we can classify all of thel in one big title which is CAD. Without forgetting the mecanical linkage and robotics.

CHAPTER 3

DISTRIBUTED IMPLEMENTATION

3.1.Overview

In this research paper, as we saw before, we have seen the problem of evaluating combinatorial rigidity of an interconnected system in the plane, without a priori knowledge of the network's topological properties. Ryan K. Williams et. al proposed the decentralization of the pebble game algorithm of Jacobs et. al., a method that determines the rigidity of a network.

This decentralization is based on asynchronous inter-agent message-passing and a distributed memory architecture, coupled with consensus-based auctions for electing leaders in the system. The auction construction allows the sequential nature of the pebble game to extend to a decentralized context, additionally rendering direct control over the identified minimally rigid subcomponent. For instance, in order to guarantee stability and relative localizability in multi-robot system.

So, in this empirical part, we will see the implementation of this algorithm, not using sockets and so on, but using the famous java threads.

3.2.The Basic of Pebble Game Algorithm Decentralization[25]

The primary considerations in decentralizing the pebble game lie in the incremental nature of edge quadrupling, the *storage of E** and *associated* pebble assignments, and *the search* and rearrangement of pebbles under the *asynchronicity* of realistic interacting systems. The proposition in this work is to arbitrate such issues through the election of leaders in the network using such auction algorithm, each expanding *E** by examining their incident edges, querying the network for free pebbles, and then exchanging leadership when the local neighborhood has been evaluated. The independent edges and pebble assignments are thus localized to each agent, distributing network storage and relying on inter-agent messaging to support asynchronicity. So, there are three major steps in this decentralization *Leader Auction*, *Leader Execution and The inter-agent messaging.*

a .Leader Election

An execution of the decentralized pebble game begins with an agent that initiates the algorithm in response to network conditions or mission objectives (e.g. verifying link deletion in *G*). The initiating agent begins by triggering an auction for electing an agent in the network to become the leading agent (or leader); an auction also runs after each leader considers all members of E_i for quadrupling.

The leader election order has a decisive role in decentralized pebble game.

b .Leader Execution

After election and initialization, the task of the leader i is to continue the expansion of E^* by evaluating the independence of each $(i,j) \in E_i$. The leader executes the procedure proposed to accomplish this task, where we assume such execution occurs relative to the leader's local clock, facilitating edge evaluation. First, recall that in checking independence a pebble covering for each quadrupled edge $e_i \in E_i$ must be determined. As the pebble information is distributed across the network, the lead agent must therefore request pebbles through messaging in an attempt to assign pebbles to e_i.

c .Inter-Agent Messaging

As each leader attempts to expand E_i^* through quadrupling, free pebbles are needed to establish a pebble covering. We accommodate the pebble search by defining *asynchronous message (request),* accompanied by *response messages (response found or not found),* indicating the existence of free pebbles.

The reception of a *request* message initiates a local test for free pebbles, if there is a free pebble it sends a response message to the requester and reverse the edges, else it initiates a further research by sending another request to their neighbors, if no free pebble is found, it sends a not found response message to the requester.

And the main idea in this work is the asynchronicity, i.e. each agent works autonomously independently and never stopping for waiting such response.

3.3.The Asynchronous Decentralized Pebble Game Algorithm [25]

Ryan K. Williams et. al proposed an asynchronous decentralized pebble game algorithm for evaluating rigidity in interconnected systems, for this aim our interconnected system will be considered as a set of agent linked through specific network model.

Our system is composed of n agents indexed by $I=\{1,\ldots,n\}$ operating in the plan, each agent autonomously computes and communicates, denoting by (i,j) a bi-directional communication link between agent i and agent j.

To describe the interconnected system formally, we define undirected graph $G= (V,E)$, having vertices (nodes) $V = \{v_1, \ldots, v_n\}$ associated with each agent $i \in$ I, and edge set $E \subset V \subset V$ with members (i,j), where by definition $(i,j) \in E \Leftrightarrow (j,i) \in E$, $\forall\, i \neq j \in I$, excluding the possibility for self-loops, $(i,i) \notin E, \forall\, i \in I$. Agents i and j with an edge $(i, j) \in E$ are referred to as *neighbors*, where the set of neighbors for the i^{th} agent is given by $N_i = \{v_i \in V \mid (i, j) \in E\}$.

Finally, for our purposes, we assume the network topology G is *connected for all time* to guarantee all agents can participate in rigidity evaluation. And we do not forget the principle of asynchronous i.e. no global clock.

Specifically, each agent $i \in I$ possesses pebble assignment set P_i having at most two edges $\{(i,j) \in E \mid j \in N_i\}$, that is incident edges (i,j) to which a pebble is associated.

For clarity, we let $f_i = 2-|P_i| \in \{0,1,2\}$ denote agent i's free pebble count, and use shorthand notation $P_i(k,l)$ for the l^{th} element of the k^{th} edge in P_i, with $k,l \in \{1,2\}$.

Local independent edge storage is denoted by $E_i^* = \{(i,j) \in E \mid j \in N_i\}$, containing edges for which quadrupling and pebble covering succeeds, where by construction $E^* = \cup E_i^*$.

To manage leadership election, each agent has booleans *isLeader(i)* and *beenLeader(i)* indicating current and prior leadership status. In order to handle pebble queries and responses, each agent retains variables *requester(i)* $\in I$ storing the agent making a request, *pathsSearched(i)* $\in \{0,1,2\}$ enumerating the local paths searched, and *requestID(i)* $\in \mathbb{R}$ uniquely identifying received requests. All variables will be initialized as it will be shown in the pertinent Algorithm, at the start of an execution of the proposed algorithm.

The pebble game distributed algorithm is composed of many procedures that are:

- Auction algorithm.
- Agent Initialization whatever *i.*
- Leader initialization after election.
- Leader execution logic.
- Pebble request handler for agent *i.*
- Pebble found handler for agent *i*.
- Pebble not found handler for agent *i.*

The pseudo-code of these algorithms can be described as follows:

Algorithm 0 : Auction Algorithm .

```
1:  ELECTLEADER(i) {
2:      Foreach agent(j){
3:          elect the agent j who has the biggest number of
4:          neighbors that have not yet been leader.
5:          return j
6:      }
7:  }
```

Algorithm 1 : Agent initialization Algorithm $\forall$ *i* $\in$*I.*

```
1:  AGENTINITIALIZE(i) {
2:      isLeader(i)← No          Leadership status
3:      beenLeader(i)← No        Precedent leadership status
4:      requester(i)← 0          pebble requester index
5:      pathsearched(i)←0        local paths searched
6:      requestID(i)←0           pebble request identifier
7:      f_i ←2                   free pebbles
```

```
8:          Pi←{}                    assigned pebbles
9:          Ei* ←{}                  local independent edge
10:   }
```

Algorithm 2 : Leader initialization after election

```
1:    LEADERINITIALIZE(i) {
2:          isLeader(i)← Yes         I am the current leader
3:          beenLeader(i)← Yes       I was leader, yes
4:          reqwait(i)← No           Waiting for free pebbles
5:          quad(i)←1                Quadrupling index
6:          Ei ← {(i,j)∈E | j ∈Ni ∧  Neighbors and not beenleader
7:          !beenLeader(j)}          Current edge
8:          ei ←  (j,k) ∈Ei          Total independent edge
9:          ind(i)←{|E*(t)|}
      }
```

Algorithm 3 : Leader execution logic

```
1:    LEADERRUN(i) {
2:    If (reqWait(i)){
3:          return
4:    }
5:    While  (ei =(j,k)≠0) {
6:          While  (quad(i)≤4) {
7:                If (fi>0)  {
8:                  Pi ←Pi ∪ ei
9:                  fi --
10:                 quad(i)++
11:                       }else {
12:                       PEBBLEREQUESTMSG(i,Pi(1,2),uID)
13:                       ReqWait(i)← Yes
14:                       pathSearched ← 1
15:                       return
16:                      }
17:               }
18:         // quad success, return 3 pebbles
19:         Pi ← { }
20:         fi ← 2
21:         // return 1 pebble to ei(2)
22:         //add independent edge and check rigidity
23:         Ei* ← Ei* ∪ ei
24:         ind(i) ++
25:         if (ind(i) = 2n-3) {
26:               send network rigidity notification
27:               return
28:         }
29:         // Go to next incident edge
30:         Ei ←Ei − ei
31:         ei = (j,k) ∈ Ei
```

```
32:             quad(i)← 1
33:   }
34:  All local edes checked
35:  Initiate leadership transfer auction
36:  }
```

Algorithm 4 : Pebble request handler for agent *i*

```
1:   HANDLEPEBBLEREQUEST(from,I,uID) {
2:   If (ALREADY REQUESTED){
3:     PebbleNotFoundMsg(I,from)
4:         return
5:   }
6:   requestID(i)= uID
7:    If (fi>0)  {
8:             Pi ←Pi ∪ (i,from)
9:              fi --
10:            PEBBLEFOUNDMSG(i,from)
11:    } else {
12:    PEBBLEREQUESTMSG(i,Pi(1,2),uID)
13:    pathSearched ← 1
14:    requester(i)← from
15:    }
16:  }
```

Algorithm 5 : Pebble FOUND handler for agent *i*

```
1:   HANDLEPEBBLEFOUND ( from , i ) {
2:    Pi ←Pi – (i,from)
3:   If (isLeader(i)){
4:      Pi ←Pi ∪ ei
5:      quad(i)++
6:      reqWait(i)← No
7:   } else {
8:   Pi ←Pi ∪ (i,requester(i))
9:             PEBBLEFOUNDMSG(i,from)
10:    }
11:  }
```

Algorithm 6 : Pebble NOT FOUND handler for agent *i*

```
1:   HANDLEPEBBLENOTFOUND ( from , I ) {
2:   If (PathSearched(i) < 2){
3:     PebbleRequestMsg(i , Pi, uID)
4:     PathSearched(i) ←2
5:   } else {
6:    If (isLeader(i))  {
7:   Return pebbles assigned to ei
8:   Go to next incident edge
9:   Ei ← Ei - ei
```

```
10:   e_i ← (j,k) ∈ E_i
11:   quad(i) ← 1
12:   reqWait(i)← No
13:     } else {
14:    PEBBLENOTFOUNDMSG( i, requester(i) )
15:               }
16:   }
17:   }
```

3.4.Tools for implementation

The distributed algorithms implementation is generally accomplished using threads not real systems and sockets. Using multithreading is the most useful way. In our case, we have considered each agent as an independent thread and created a shared means of communication between them in order to guarantee the communication between these threads. So, we have to introduce threads, their intercommunication way (ICP) and their synchronization, all in java programming language.

3.4.1.Java threads

we describe how threads are implemented in the Java programming language, we start with a general overview of threads.

A thread is a program in execution. Most programs written today run as a single thread, causing problems when multiple events or actions need to occur at the same time. Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input lacking the ability to handle more than one event at a time. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows us to do this.[43]

3.4.2.Creating Java thread

Java's creators have graciously designed two ways of creating threads: extending a class and implementing an interface.

The first method of creating a thread is to simply extend from the Thread class. The Thread class is defined in the package *java.lang* which needs to be imported so that our classes are aware of its definition. Here is an example:

```
import java.lang.*;
public class agent extends Thread
{
      public void run()  { ...
      }}
```

The same class can be created by implementing the interface Runnable:

```
import java.lang.*;
public class agent implements Runnable
{
        Thread T;
        public void run()
        {
        ....
        }
}
```

3.4.3.Start, Stop, suspend and resuming java threads

After the thread is created it needs to be started. The call to *start()* will call the target's *run()* method, which is *agent.run()*. The call to *start()* will return right away and the thread will start executing at the same time. Note that the *run()* method is an infinite loop.

Note that it is important to sleep somewhere in a thread. If not, the thread will consume all CPU time for the process and will not allow any other methods such as threads to be executed. Another way to cease the execution of a thread is to call the *stop()* method.

Instead you can pause the execution of a thread with the *sleep()* method. The thread will sleep for a certain period of time and then begin executing when the time limit is reached. But, this is not ideal if the thread needs to be started when a certain event occurs. In this case, the *suspend()* method allows a thread to temporarily cease executing and the *resume()* method allows the suspended thread to start again.

There is also *wait()* method which block the thread at this level until a condition is verified, and *notify()* which resume the blocked thread. These two methods are frequently used in synchronization among processes.[43]

3.4.4.Mutual exclusion in Java

The Java `synchronized` keyword is an essential tool in concurrent programming in Java.

Its overall purpose is to only allow one thread at a time into a particular section of code thus allowing us to protect variables or data from being corrupted by simultaneous modifications from different threads.[42]

Using a `synchronized` block for mutual exclusion, at its simplest level, a block of code that is marked as `synchronized` in Java tells the JVM: *"only let one thread in here at a time"*, for instance:

`synchronized` (cpt){ cpt++} ;

This means that only one process can modify the variable *cpt* and it is in mutual exclusion until the process is ended up.

3.4.5.Inter-thread communication

Inter-Process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated. For this mode, there are many methods such as shared memory, pipes, channels, mailboxes, ports, message passing, semaphores, monitors etc.[31]

3.4.6.Producer and Consumer paradigm

The producer and consumer problem is one where two processes must coordinate to exchange data. In this system, a producer process is periodically creating new data elements and consumer process is waiting for these data items to be created and is using them for some other task. In order for this system to function, the producer and consumer require a communication process to allow them to coordinate when the producer has created a new item so that the consumer can successfully read the data. Such a system can be built using either message passing or a shared memory approach.[31]

3.4.7.Message Passing

Message passing is a form of communication used in interprocess communication. Communication is made by the sending of messages to recipients. Each process should be able to name the other processes. The producer typically uses *send()* system call to send messages, and the consumer uses *receive()* system call to receive messages. These system calls can be either synchronous or asynchronous, and could either be between processes running on a single machine, or could be done over the network to coordinate machines in a distributed system. This allows the producer to transfer data to the consumer as it is created.[31]

3.4.8.Shared Memory

Shared Memory is an OS provided abstraction which allows a memory region to be simultaneously accessed by multiple programs with an intent to provide communication among them. One process will create an area in RAM which other processes can access (this is typically done using system calls mmap, shmget etc). Normally the OS prevents processes from accessing the memory of another process, but the Shared Memory features in the OS can allow data to be shared. Since both processes can access the shared memory area like regular working

memory, this is a very fast way of communication (as opposed to other mechanisms of IPC). On the other hand, it is less powerful, as for example the communicating processes must be running on the same machine (whereas other IPC methods can use a computer network), and care must be taken to avoid issues if processes sharing memory are running simultaneously and may try to edit the shared buffer at the same time.[31]

3.5.1.Model for Distributed Pebble Game Algorithm

We have modeled each agent (robot, sensor network etc.) as a single thread, each agent has an initialization algorithm which we have put in the constructor of the agent thread.

The auction algorithm we used is our own contribution, we used for this purpose a FIFO list that store the discovered neighbors that have not been leader yet, and elect among them the one that has the biggest number of neighbors (in fact, this idea is logic, to be elected as a leader or president, you have to be supported and has many militants, supporters and neighbors), so our auction algorithm is like spiral.

The leader run algorithm is modeled as a void method in the agent thread in which the process of quadrupling start locally for the current edge, then pebbling the agents through asynchronous message passing mechanism where the leader sends a request message and does not blocking the work.

The send (request) message is modeled as a method in the agent thread exactly like the receive messages (pebble found message and pebble not found message).

The means of communication among these threads is synchronized using specific java key words.

Once the program knows the number of the nodes (agents, robots, etc) in the network, it created *n* similar a threads that cooperate together to test the rigidity in our distributed system.

Yet, without forgetting the fundamental principle in the distributed systems which says that each member system in the network knows only its neighbors, so it does not communicate immediately with the distant (not neighbors, to which it has no direct link) system, i.e each system communicate with the sytems that have link with. To implement this idea, we used synchorinzed a stack by overloading its methods.

Our program test whether the number of independent edges is equal to twice the number of nodes (agents) minus three, if it is the case, our distributed system is rigid, otherwise it is not rigid. And this is our goal.

3.5.2.Java source code

The chared channel Java source code as we synchronized :

```
private class SharedChannel {
        private Stack channel;
        private SharedChannel() {
           this.channel = new Stack();
        }
        public synchronized boolean isEmpty() {
           return channel.isEmpty();
        }
        public synchronized Object peek() throws InterruptedException {
           while (channel.isEmpty()) {
              wait(); }
           return channel.peek();
        }
        public synchronized void push(Object str) {
           channel.push(str);
           notify();
        }
        public synchronized void pop() throws InterruptedException {
           while (channel.isEmpty()) {
              wait();
           }
           channel.pop();
        }
        public boolean contains(Object o) {
           return channel.contains(o);
        }
        public synchronized int size() {
           return channel.size();       }
        public synchronized int search(Object o) {
           return channel.search(o);
        }
        public synchronized boolean removeElement(Object str) throws InterruptedException
{
```

```
            while (channel.isEmpty()) {
                return true;
            }
            channel.removeElement(str);
            return true;
        }
        public synchronized Object elementAt(int index) throws InterruptedException {
            while (channel.isEmpty()) {        wait();     }
            return channel.elementAt(index);
        }
        public void clear() {
            if (!channel.isEmpty()) {
                channel.clear();
            }
        }
    }
```

The data structure used to represent the initial graph is :

```
public static int graph[][] = new int[n][n];
```

The data structure used to represent the total number of cevered edges is:

```
public static int Total_covered_edges;
```

the data structure used to represent the shared IPC (inter-process communication) is :

```
SharedChannel sharedbuffer = new SharedChannel();
```

The data structure used to represent the list of agent that have been leader and the the current leader respectively is :

```
public static boolean beenleader[] = new boolean[n] , isleader_i[] = new boolean[n];
```

The data structure used to represent the number of free pebbles is :

```
public static int[] freePebbles = new int[n];
```

The data structure used to assign the number of neighbors and not being leader yet to each agent is :

```
public static int[] auction = new int[n];
```

The data structure used to represent the assigned pebbles of an edge *i* is:

```
SharedChannel[] Pi = new SharedChannel[n];
```

and at max has 2 vertices.

The data structure used to represent the list of total independent edge is :

SharedChannel epsilon_iS = new SharedChannel();

The data structure used to represent the order of electing leader is :

SharedChannel Leader_elechn = new SharedChannel();// 4

The doled proposed for each node (agent) is an independent thread defined as follows :

```
private class Agent extends Thread {
    int current_edge = -1;
    protected int i;
    int bgn() {…
    }
    void Ni(int i) throws InterruptedException {…
    }
    void init_request_ID() {…
    }
    void leaderInitialize(int i, int Total_covered_edges) throws InterruptedException {…
    }
    void leaderRun(int i) throws InterruptedException {…
    }
    void PebbleRequestMsg(int from, int i, int uIDi) throws InterruptedException {…
    }
    void PebbleFoundMsg(int from, int i) throws InterruptedException {…
    }
    void PebbleNotFoundMsg(int from, int i) throws InterruptedException {…
    }
    @Override
    public void run() { … }
}
```

And the main mthod for creating our distributed system consists in creating *n* thread and each thread represents an autonomous agent as depicted in follows :

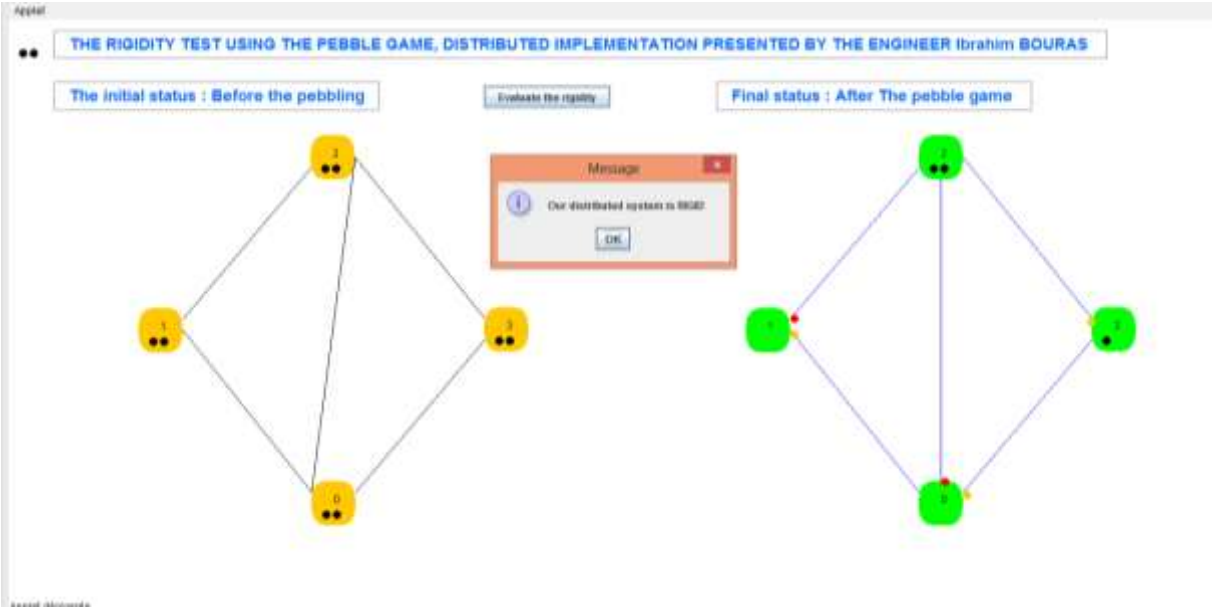
```
public static void main(String[] args) {
    Distributed_pg t = new Distributed_pg();
   Thread[] agent_thread = new Thread[n];
  // intialize the graph ;
    init_procedure(graph);
```

```
// create n similar agent , means n similar threads
for (int i = 0; i < n; i++) {
   agent_thread[i] = t.new Agent(i);
   agent_thread[i].start();
}
```

And that's all, the system will work alone.

3.5.3.Experimental demonstration

This is snapshot of our demonstration.



For the following graph :

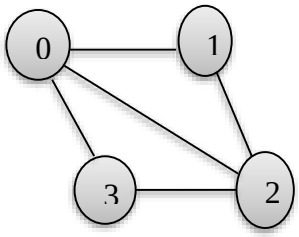


Figure 3.1: The used graph for test the implemenetation

```
run:
 Im Agent 0 at beginnig , I have 2 Free Pebbles
 Im Agent 1 at beginnig , I have 2 Free Pebbles
 Im Agent 2 at beginnig , I have 2 Free Pebbles
 Im Agent 3 at beginnig , I have 2 Free Pebbles
neighbor and not been leader yet of 0 are : 1
neighbor and not been leader yet of 0 are : 2
neighbor and not been leader yet of 0 are : 3
        ////-- 0 im the leader
```

Current edge is 0-->1

PebbleRequestMsg(0,1,0)
PebbleFoundMsg(1,0)
free pebbles [0] = 3Current edge is 0-->1

PebbleRequestMsg(0,1,1)
PebbleFoundMsg(1,0)
free pebbles [0] = 4Current edge is 0-->1

Total_covered_edges =1
freepebles[0]=2
freepebles[1]=1
freepebles[2]=2
freepebles[3]=2
So, Our Distributed System is NOT Rigid yet

Current edge is 0-->2

PebbleRequestMsg(0,2,0)
PebbleFoundMsg(2,0)
free pebbles [0] = 3Current edge is 0-->2

PebbleRequestMsg(0,2,1)
PebbleFoundMsg(2,0)
free pebbles [0] = 4Current edge is 0-->2

Total_covered_edges =2
freepebles[0]=2
freepebles[1]=1
freepebles[2]=1
freepebles[3]=2
So, Our Distributed System is NOT Rigid yet

Current edge is 0-->3

PebbleRequestMsg(0,3,0)
PebbleFoundMsg(3,0)
free pebbles [0] = 3Current edge is 0-->3

PebbleRequestMsg(0,3,1)
PebbleFoundMsg(3,0)
free pebbles [0] = 4Current edge is 0-->3

Total_covered_edges =3
freepebles[0]=2
freepebles[1]=1
freepebles[2]=1
freepebles[3]=1
So, Our Distributed System is NOT Rigid yet

```
-----new leader is elected---:
neighbor and not been leader yet of 1 are : 2
        ////-- 1 im the leader
Current edge is 1-->2

PebbleRequestMsg(1,0,0)
PebbleFoundMsg(0,1)
    free pebbles [1] = 2Current edge is 1-->2

PebbleRequestMsg(1,2,1)
PebbleFoundMsg(2,1)
    free pebbles [1] = 3Current edge is 1-->2

PebbleRequestMsg(1,2,2)
PebbleRequestMsg(2,0,2) ..
PebbleFoundMsg(0,2)
PebbleFoundMsg(2,1)
    free pebbles [1] = 4Current edge is 1-->2

Total_covered_edges =4
freepebles[0]=0
freepebles[1]=2
freepebles[2]=1
freepebles[3]=1
 So, Our Distributed System is NOT Rigid yet

-----new leader is elected---:
-----new leader is elected---:
-----new leader is elected---:
neighbor and not been leader yet of 2 are : 3
        ////-- 2 im the leader
Current edge is 2-->3

PebbleRequestMsg(2,1,0)
PebbleFoundMsg(1,2)
    free pebbles [2] = 2Current edge is 2-->3

PebbleRequestMsg(2,3,1)
PebbleFoundMsg(3,2)
    free pebbles [2] = 3Current edge is 2-->3

PebbleRequestMsg(2,3,2)
PebbleRequestMsg(3,0,2) ..
PebbleRequestMsg(0,1,2) ..
PebbleFoundMsg(1,0)
PebbleFoundMsg(0,3)
PebbleFoundMsg(3,2)
    free pebbles [2] = 4Current edge is 2-->3
```

Total_covered_edges =5
Our distributed system is RIGID
freepebles[0]=0
freepebles[1]=0
freepebles[2]=2
freepebles[3]=1

we remark that a graph of four (4) vertices and five (5) edges needs fourty six (46) messages for evaluating the rigidity in our distributed system. Three times leader election. Three remaining pebbles at the endwhich meadns the DOF.

In this chapter, we have seen the distributed implementation of the pebble game algorithm using java threads and the needed tools and components for achieving the goal. We modeled the agents as threads and the communication between them via IPC techniques, we've verified our algorithm and it worked.

CONCLUSION

In this dissertation, we have presented the famous Rigidity Theory and its importance, its applications, and the adequate algorithm for evaluating rigidity and flexibility.

The notion of Flexibility is as important as Rigidity in some areas such as protein, both notions are important, when evaluating rigidity, rigidity is evaluating simultaneously.

We showed the major application areas of this notion such as Biology, Architecture and various CAD applications. We shed light on some important applications.

For evaluating rigidity quickly in 2D, we have presented the $2|V|$-3 pebble game algorithm in its two versions, the centralized and the distributed ones, which counts $2|V|$-3 independent edges to be rigid. It was an important task to offer detailed illustrations of the basic pebble operation which is covering any edge in the structure to be studied.

The pebble game algorithm gives us one of the most important information which is the Degree of Freedom (DOF), that can be concluded from the number of free pebbles remained once the rigidity have been evaluated, i.e. at the end of the execution of this algorithm, the DOF equals the remaining Free Pebbles.

We have mentioned several important theorems which will increase our knowledge and understanding of the pebble game algorithm. It is very important to understand the details of the pebble game algorithm and know some additional properties and useful extensions for having a closer look, and be able to correct or propose versions.

The overall strategy and objective of this dissertation was to implement the distributed pebble game algorithm for testing the combinatorial rigidity theory, with the underlying motivation that these algorithms can be used to address important applications and problems in many fields.

The decentralization of the aforementioned algorithm (the centralized version of the pebble game for 2D proposed in 1997 by D. J. Jacobs and B. Hendrickson) proposed by Ryan K. Williams et al is based on asynchronous inter-agent message-passing and a distributed memory architecture, coupled with algorithm auction for electing leaders to lead during covering a specific edge.

In the auction algorithm we used, we proposed a combination of two politics, the first one, storing the neighbors of the current leader in FIFO order, then we elect the neighbor who has the maximum number of neighbors from this list, in fact it is logic, to be elected as leader you have to be the most supported.

The decentralization of the pebble game algorithm is based on seven algorithms: the auction algorithm, the initialization of the agent in the constructor of the thread, the leader run, the message request algorithm, the message found and the not found message algorithm.

Our implementation was done using Java threads, which is the ideal simulation of the distributed algorithms, car in one hand, fault tolerance and message alteration is null, in other hand, the communication between the threads was done using inter-communication processes politics in asynchronous messaging mode. i.e shared handler.

During this research, we proved that the pseudo algorithm published in the used article is not detailed enough to be implemented. It needs more efforts.

We found that this problem is one of the most curious domain, beside to its open problems such as the double banana in 3D. So, the gate is still open to the research and towards the application of this algorithm in many fields of research. Here is the entrance to another researcher.

References

[1] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts and P. Walter, Molecular Biology of the Cell, Garland Science, New York, Fourth Edition, 2002.

[2] M.E. Barnes and W. Whiteley, Preprint, York University, 2005.

[3] F. Bemporad, J. Gsponer, H.I. Hopearuoho, G. Plakoutsi1, G. Stati1, M. Stefani1, N. Taddei, M. Vendruscolo and F. Chiti, Biological function in a non-native partially folded state of a protein, The EMBO Journal, 27, 15251535, 2008.

[4] M.V. Chubynsky and M.F. Thorpe, Algorithms for three-dimensional rigidity analysis and a first-order percolation transition, Physical Review, 76, 041135.

[5] R. Connely, T. Jordan and W. Whiteley, Generic Global Rigidity of Body-Bar Frameworks, EGRES Technical Report No. 2009-13.

[6] R. Diestel, Graph Theory, Springer-Verlag, New York, Second Edition, 2000.

[7] H. Gluck, Almost all simply connected closed surfaces are rigid, In Geometric Topology, Lecture notes in Mathematics, Springer-Verlag, Berlin, No. 438, p.225-239, 1975.

[8] J. Graver, Counting on Frameworks: Mathematics to Aid the Design of Rigid Structures, The Mathematical Association of America, Washington, 2001.

[9] J. Graver, B. Servatius, and H. Servatius, Combinatorial Rigidity, Graduate Studies in Math., AMS, 1993.

[10] M. Grubler, Allgemeine Eigenschafter der Zwanglaugen ebenen kinematischen Ketten, Part I, Zivilingenieur 29, pp. 167-200, 1883.

[11] B. Hendrickson and D.J. Jacobs, An algorithm for two dimensional rigidity percolation: The pebble game, J. Comput. Phys., 137:346-365, 1997.

[12] B. Hendrickson, Conditions for unique graph realizations, SIAM J. Comput., 21, pp. 65-84, 1992.

[13] L. Henneberg, Die graphische Statik der starren Systeme. Leipzig, 1911; Johnson Reprint, 1968.

[14] B. Jackson and T. Jordan, The generic rank of body-bar-and-hinge frameworks, EGRES Technical Report No. 2007-06.

[15] D.J. Jacobs, Generic rigidity in three-dimensional bond-bending networks, Journal of Physics a-Mathematical and General, 31:6653-6668, 1998.

[16] Donald J. Jacobs, A.J. Rader, Leslie A. Kuhn, Protein Flexibility Predictions Using Graph Theory. 216.

[17] N. Katoh, and Tanigawa, S., A Proof of the Molecular Conjecture, Discrete & Computational Geometry 45: 647-700, 2011.

[18] L.A. Kuhn, D.J. Rader and M.F. Thorpe, Protein flexibility predictions using graph theory, Proteins, 44:150-65, 2001.

[19] G. Laman, On graphs and rigidity of plane skeletal structures, J. Eng. Mathematics, 4:331-340, 1970.

[20] A. Lee and I. Streinu. Pebble Game Algorihms and (k, l)-sparse graphs, 2005 European Conference on Combinatorics, Graph Theory and Applications (EuroComb 05), DMTCS Proceedings: 181-186, 2005.

[21] A. Lee and I. Streinu. Pebble game algorithms and sparse graphs, Discrete Mathematics, 308, 1425-1437, 2008.

[22] A. Mantler and J. Snoeynik, Banana Spiders: A study of connectivity in 3D combinatorial rigidity, CCCG, 44-47, 2004.

[23] J.C. Maxwell, On reciprocal figures and diagrams of forces, Phil. Mag., 27:250-261, 1864.

[24] A. Recski, Matroid Theory and its Applications, Springer-Verlag, Berlin, 1989.

[25] K. Ryan, Williams, Andrea Gasparri, Attilio Priolo, and Gaurav S. Sukhatme ,Decentralized Generic Rigidity Evaluation in Interconnected Systems,IEEE International Conference on Intelligent Robots and Systems, 2013.

[26] O. Shai, B. Servatius, and W. Whiteley, Combinatorial characterization of the assur graphs from engineering. European Journal of Combinatorics 31, (2010),1091-1104.

[27] A. Sljoka, Algorithms In Rigidity Theory With Applications To Protein Flexibility And Mechanical Linkages, Doctoral Thesis,Toronto, Ontario,2012.

[28] A. Sljoka, O. Shai. and W. Whiteley, Checking Mobility and Decomposition of Linkages via Pebble Game Algorithm, Proceedings for ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, IDETC/CIE, Aug 28-31, 2011, Washington, USA, 2011.

[29] D. Stewart, A Platform with Six Degrees of Freedom, Proc. Institution of Mechanical Engineers (UK), Vol 180, Pt 1, No 15, 1964.

[30] K. Sugihara, On redundant bracing in plane skeletal structures, Bull. Electrotech. Lab. 44, 376, 1980.

[31] Andrew S. Tanenbaum, Modern Operating Systems, Pearson Prentice Hall, 2008, pp.115-120.

[32] T.S. Tay and W. Whiteley, Generating Isostatic Frameworks, Structural Topology 11, 21-69, 1985.

[33] M.F.Thorpe, D.J. Jacobs, N.V. Chubynsky and A.J. Rader, Generic rigidity of network glasses. In: Thorpe MF, Duxbury PM, eds. Rigidity theory and applications. NewYork:Kluwer Academic/ Plenum Publishers, pp 239277, 1999.

[34] W. Whiteley, Counting out to the fexibility of molecules, Phys. Biol. 2, S116-S126, 2005.

[35] W. Whiteley, Rigidity and scene analysis, In J. Goodman and J. ORourke, editors, Handbook of Discrete and Computational Geometry, chapter 60, pages 1327-1354. Chapman Hall/CRC Press, Boca Raton, FL, 2nd edition, 2004.

[36] W. Whiteley, Rigidity of molecular structures: generic and geometric analysis, Rigidity theory and applications, M.F. Thorpe and P.M. Duxbury, Editors, Academic/Kluwer. p. 21-46, 1999.

[37] W.Whiteley, Some Matroids from discrete applied geometry. In J. Bonin, J. Oxley, and B.Servatius, editors, Matroid Theory, volume 197 of Contemp. Math.,pages 171-311. Amer. Math. Soc., Providence, 1996.

[38] W. Whiteley, The Equivalence of Molecular Rigidity Models as Geometric and Generic Graphs, Manuscript, 2004.

[39] W. Whiteley, Y. R. Yang,J. Aspnes, T. Eren, D.K. Goldenberg Student Member, IEEE, A. S. Morse Fellow, A Theory of Network Localization, 2005.

Websites

[40] http://2012.igem.org/Team:TU_Darmstadt/Modeling_GNM

[41] http://kinari.cs.umass.edu

[42]http://www.javamex.com/tutorials/synchronization_concurrency_synchronized1.shtml

[43] http://www.javaworld.com/article/2077138/java-concurrency/introduction-to-java-threads.html

[44] http://www.referenceforbusiness.com/management/Bun-Comp/Computer-Aided-Design-and-Manufacturing.html

[45] https://tensegrity.wikispaces.com/Tensegrity+Tower+by+Fagerstr%C3%B6m

ملخص

لقد تساءل جيمس كلارك ماكسويل ما إذا كان حساب عدد العقد وعدد الروابط في شبكة ما كافٍ لمعرفة تماسك البنية أم ليونتها. بعدها أجابه لامان وأكد ذلك بأن هذا المشكل في المستوي هو مشكلة حسابية محضة بمعنى الحساب كاف.

شرط حساب 2|v|-3 في المستوي يمكن حسابه بسرعة باستعمال خوارزم لعبة الحجيرات الذي اقترح له هاندريكسون وجايكوبس نسخته الممركزة وبعده اقترح مجموعة من الباحثين النسخة الموزعة للخوارزم . في هذه المذكرة طبقنا النسخة الموزعة لخوارزم لعبة الحجيرات باستعمال التواصل غير المتواقت للرسائل وباستعمال برنامج البرمجة جافا واستعملنا لذلك العمليات الجزئية وطرق التواصل بينها. لنصل في الأخير إلى الهدف المنشود.

الكلمات المفاتيح : Rigidity, Flexibility, Laman graph, pebble game algorithm.

Abstract

James Clark Maxwell wondered: counting vertices and edges in a given framework is enough to make predictions about its rigidity and fexibility ?

Laman in 1970 confirmed that the 2d generic bar and joint frameworks is completely combinatorial problem. i.e counting is enough to decide.

The 2|V|-3 counting condition for 2d structures verified by the fast pebble game algorithm which tracks quickly this count. Hendrickson and jacobs proposed the centralized pebble game version in 1997. In 2014, group of researchers proposed the decentralized version in IEEE.

In this thesis, we the implement the asynchronous, distributed algorithm of the pebble game for testing the rigidity in 2d framework, using java threads and interprocess communicatons protocols.

We have also introduced several rigidity theory applications, we will apply our algorithm on some 2d strucutres. The results obtained in this Master theisis showed that our distributed implementation is well done and worked.

Keywords: Rigidity, Flexibility, Laman graph, pebble game algorithm.

Résumé

James Clark Maxwell posa une question : Compter le nombre des arcs et des sommet de cetraine graph est suffisant pour tester sa rigidity ?

Laman en 1970 affirma que en 2d, generique bar-hinge structure est totalement un probleme combinatoriale i.e compter est suffisant pour evaluer.

La condition 2|v|-3 pour les structure 2d verifie par l'algorithme de pebble game qui est le plus rapide testeur. Hendrickson et Jacobs proposerent le version centralisé de cet algorithm en 1997. En 2014 un ensemble des chercheurs proposerent le version décentralisée.

En cet mémoire, nous implementons la version decentralisée et asynchrone de l'agorithme de pebble game pour tester la rigidity dans un système completement distribue en utilisant Java threads et IPC les protocoles de la communication entre-processus.

Nous avons aussi mentionné les differents domaines d'applications de la rigidity, et enfin nous avons réalisé nôtre objective.

Les mots cléts: Rigidity, Flexibility, Laman graph, pebble game algorithm.