10-19-2015

Experiments with Point Placement Algorithms and Recognition of Line Rigid Graphs

Pijus Kumar Sarker
University of Windsor

Experiments with Point Placement Algorithms

and

Recognition of Line Rigid Graphs

By

Pijus Kumar Sarker

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2015

Experiments with Point Placement Algorithms

and

Recognition of Line Rigid Graphs

by

Pijus Kumar Sarker

APPROVED BY:

Dr. Esaignani Selvarajah
Odette School of Business

Dr. Xiaobu Yuan
School of Computer Science

Dr. Asish Mukhopadhyay, Advisor
School of Computer Science

September 11, 2015

DECLARATION OF ORIGINALITY

I, Pijus Kumar Sarker, declare that I am the sole author of this thesis titled, 'Experiments with Point Placement Algorithms and Recognition of Line Rigid Graphs' and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyones copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix. I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University of Institution.

ABSTRACT

The point placement problem is to determine the position of n distinct points on a line, up to translation and reflection by fewest possible pairwise adversarial distance queries. This masters thesis focusses on two aspects of point placement problem. In one part we focusses on an experimental study of a number of deterministic point placement algorithms and an incremental randomized algorithm, with the goal of obtaining a greater insight into the behavior of these algorithms, particularly of the randomize algorithm.

The pairwise distance queries in the point placement problem creates a type of graph, called *point placement graph.* A point placement graph G is defined as line rigid graph if and only if the vertices of G has unique placement on a line. The other part of this thesis focusses on recognizing line rigid graph of certain class based on structural property of an arbitrarily given graph. Layer graph drawing and rectangular drawing are used as key idea in recognizing line rigid graphs.

DEDICATION

To my loving parents,

Ajay Kumar Sarker and Joiotsna Rani

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my advisor *Prof. Asish Mukhopadhyay*. Without his inspiration and supervision this thesis would not have been possible.

I would like to thank my committee members, *Prof. Xiaobu Yuan* and *Prof. Esaignani Selvarajah* for their, patience, objectivity and observations.

A number of people who motivated me and supported time to time, and credit goes to:
-*Ajay Kumar Sarker* for believing on my belief.
-*Joiotsna Rani* for being my first teacher.
-*Mousumi Saha* for her continuous support to make me strong and believe on myself in every-way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Recently researchers from computer science showed a great interest in biological problems. Researchers came up with a number of interesting problems from biology that can be analyzed or solved theoretically. One of the interesting problem is DNA mapping problem. In general the goal is the recover the whole DNA sequence for an organism. An approach is to use some known substrings which are called *markers* or *restriction site*. Pairwise distances between the *markers* can be measured with *flourescent in situ hybridization* (FISH) experiment. The problem is to find relative position of the markers in the DNA sequence. The researchers reduced the problem to a *point placement problem* and tried to solve it efficiently using graph theories and distance geometry approach. In the point placement problem the markers are represented as points(each of them are distinct) and the distances between pair of marker represents the distance between pair of points. The point placement problem is to find unique position of a set of distinct points on a line with minimum number of distance queries between the points, where relative distance between the points will remain the same if their positions is transformed or reflected. The Point placement problem has two flavour: exact model and inexact model. In exact model distance between all pair of points are known whereas in inexact model distance between all pair of points are not known and each distance is bounded by upper bound and lower bound. In the first part of this thesis we are focussing on the exact model where distance between all pair of points are known.

The distance queries in the point placement problem creates *point placement graph(ppg)* which is the input to point placement algorithm. In the point placement graph the each vertex corresponds to point, every edge corresponds to connection between two points and edge length correspond to the distance between pairs of points. If the vertices of point placement graph have unique placement on a line then the graph is called line rigid graph or rigid point placement graph. Thus the point placement problem reduces to construction of line rigid graph. Researchers have studied the problem to construct rigid point placement graphs [1], [2], [3], [4]. In this thesis, we have formulated a new problem. The problem is to recognize line rigid graphs based on structural charactarization. In another part of this thesis, we focus on the recognition problem.

1.2 Problem Statement

1.2.1 Experimental Study of existing Point Placement Algorithms

The point placement problem is to determine the position of n distinct points on a line, up to translations and reflections by the fewest possible pairwise (adversarial) distance queries. In this thesis we report on an experimental study of a number of deterministic point placement algorithms and an incremental randomized algorithm, with the goal of obtaining a greater insight into the behavior of these algorithms, particularly of the randomized one.

1.2.2 Recognizing Line Rigid Graphs

A graph G is line rigid if it has an unique linear layout, up to translation and reflection. Every line rigid graph is associated with a function l that assigns positive lengths to the edges and an assignment of edge length called *valid* if the graph has unique linear layout. The problem is to verify whether an arbitrarily given graph is line rigid or not. In this thesis we have presented a scheme to recognize line rigid graphs with some properties. The given graph G must be planer, *2-connected* and maximum degree of the vertices are three. A graph is planar if it can be embedded in the plane so that no two edges intersect geometrically except at a vertex to which they are both incident. And in *2-connected* graphs at least 2 vertices needed to remove to disconnect the graph.

1.3 Contributions

In this thesis we have worked on two aspects of point placement problem. In first part of this thesis we have done an experimental study of the various point placement algorithms in order to study the trade-off between query and time complexities. We have specially focus on the randomized point placement algorithm and compare with other deterministic point placement algorithm.

In the second part we have worked on a new problem, *Line Rigid Graph Recognition* which is motivated from point placement problem. The problem is to recognize line rigid graphs based on the structural characterization of a given graph. We have established an connection between line rigidity and graph drawing and proposed a scheme that recognizes line rigid graphs of a class, *2-3 planar graph*.

1.4 Chapter Outline

The list bellow presents the organization of the chapters with summary of the contents.

- Chapter 2 contains a detailed study of existing point placement algorithms and measure performance of the algorithms by observing the experimental results.
- Chapter 3 contains a proposed scheme to recognizes line rigid graphs with some constraints.
- Chapter 4 contains overall conclusion and future work.

Chapter 2

Experimental Study on Point Placement Algorithms

2.1 Introduction

The point placement problem: Let $P = \{p_1, p_2, ..., p_n\}$ be a set of n distinct points on a line L. The point location problem is to determine the locations of the points uniquely (up to translation and reflection) by making the fewest possible pairwise distance queries of an adversary. The queries can be made in one or more rounds and are modeled as a graph whose nodes represent the points and there is an edge connecting two points if the distance between the corresponding points is being queried. The distances between the pairs of points returned by the adversary are exact.

A special version of this problem is when a query graph is presented with assigned edge lengths and all possible placements of its vertices are to be determined. In , this problem was solved for weakly triangulated graphs.

A classical version of this problem is the construction of the coordinates of a set of n points, given exact distances between all pairs of points (see [5], [6]). Algorithms exist that not only determine the coordinates but also the minimum dimension in which the points can be embedded (see [7]).

2.2 Motivation

The motivation for studying this problem stems from the fact that it arises in diverse areas of research, to wit computational biology, learning theory, computational geometry, etc.

In learning theory this problem is one of learning a set of points on a line non-adaptively, when learning has to proceed based on a fixed set of given distances, or adaptively when learning proceeds in rounds, with the edges queried in one round depending on those queried in the previous rounds.

The version of this problem studied in Computational Geometry is known as the turnpike problem. The description is as follows. On an expressway stretching from town A to town B there are several gas exits; the distances between all pairs of exits are known. The problem is to determine the geometric locations of these exits. This problem was first studied by Skiena *et al.* [8] who proposed a practical heuristic for the reconstruction. A polynomial time algorithm was given by Daurat *et al.* [9].

In computational biology, it appears in the guise of the restriction site mapping problem. Biologists discovered that certain restriction enzymes cleave a DNA sequence at specific sites known as restriction sites. For example, it was discovered by Smith and Wilcox [10] that the restriction enzyme Hind II cleaves DNA sequences at the restriction sites GTGCAC or GTTAAC. In lab experiments, by means of fluorescent in situ hybridization (FISH experiments) biologists are able to measure the lengths of such cleaved DNA strings. Given the distances (measured by the number of intervening nucleotides) between all pairs of restriction sites, the task is to determine the exact locations of the restriction sites. The point location problem also has close ties with the probe location problem in computational biology (see [11])

The turnpike problem and the restriction mapping problem are identical, except for the unit of distance involved; in both of these we seek to fit a set of points to a given set of inter-point distances. As is well-known, the solution may not be unique and the running time is polynomial in the number of points. While the point placement problem, prima facie, bears a resemblance to these two problems it is different in its formulation - we are allowed to make pairwise distance queries among a distinct set of labeled points. It turns out that it is possible to determine a unique placement of the points up to translation and reflection in time that is linear in the number of points.

2.3 Overview of contents

In the next section we briefly review some of the well-known deterministic algorithms and the only known incremental randomized algorithm. In the following section we report on the experimental results obtained by careful implementations of several deterministic algorithms and the incremental randomized algorithm. This is followed by a detailed discussion of the results and we conclude in the next section.

2.4 Overview of some current point placement algorithms

Several algorithms are extant that work in one or more rounds. The current state of the art is summarized in Table 2.1.

TABLE 2.1: *The current state of the art*

Algorithm	Rounds	Query Complexity		Time Complexity
		Upper Bound	Lower Bound	
3-cycle	1	$2n-3$	$4n/3$	$O(n)$
4-cycle	2	$3n/2$	$9n/8$	$O(n)$
5-cycle	2	$4n/3+O(\sqrt{n})$	$9n/8$	$O(n)$
5:5 jewel	2	$10n/7+O(1)$	$9n/8$	$O(n)$
6:6 jewel	2	$4n/3+O(1)$	$9n/8$	$O(n)$
3-path	2	$9n/7$	$9n/8$	$O(n)$
randomized	2	$n+O(n/\log n)$	?	$O(n^2/\log n)$

Comment: The $9n/8$ lower bound on 2-round algorithms was proved in [4], improving the lower bound of $30n/29$ by Damaschke [12] and the subsequent improvement to $17n/16$ by [1] and the further improvement to $12n/11$ by [3]. As for the lower bound on 1-round algorithms, the following result was proved in [12].

Theorem 2.1. *The density of any line rigid graph is* $4/3$ *with the exception of the jewel,* $K_{2,3}, K_3$ *and* K_4^- *(shown in Fig 2.1).*

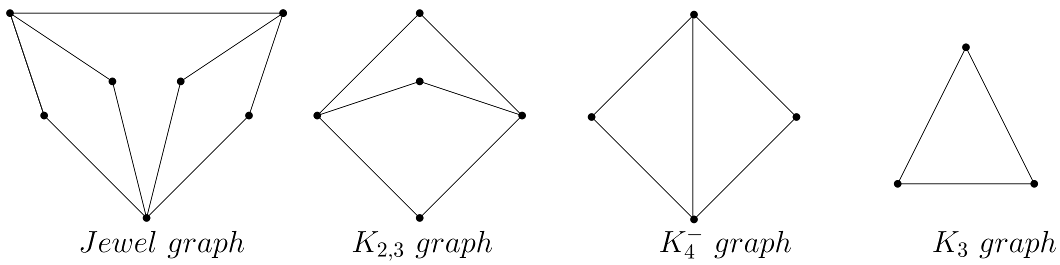


FIGURE 2.1: Graphs quoted in Theorem 2.1

The density, multiplied by n, gives the lower bound of $4n/3$.

The simplest of all, the 3-cycle 1-round algorithm, has the query graph shown in Fig. 2.2:

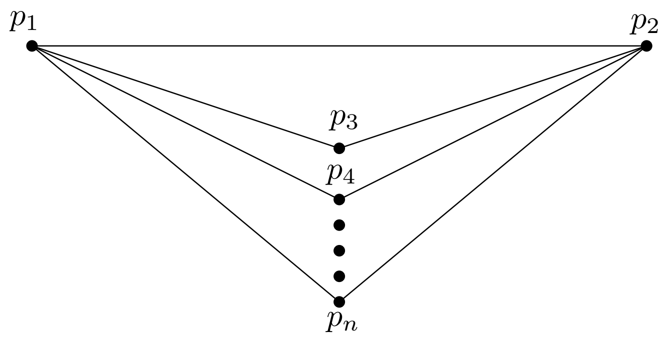


FIGURE 2.2: Query graph using triangles.

The query complexity of this algorithm is $2n - 3$ self-evident as this is the number of edges in the graph. The 4-cycle 2-round algorithm is typical of the other 2-round algorithms listed in Table 2.1 and thus merits a brief description.

If $G = (V, E)$ is a query graph, an assignment l of lengths to the edges of G is said to be valid if there is a placement of the nodes V on a line such that the distances between adjacent nodes are consistent with l. We express this by the notation (G, l). By definition (G, l) is said to be line rigid if there is a unique placement up to translation and reflection, while G is said to be line rigid if (G, l) is line rigid for every valid l. A 3-cycle (or triangle) graph is line rigid, which is why the 3-cycle algorithm needs only one round to fix the placement of all the points. A 4-cycle (or quadrilateral) is not line rigid, as there exists an assignment of lengths that makes it a parallelogram whose vertices have two different placements as in Fig. 2.3.

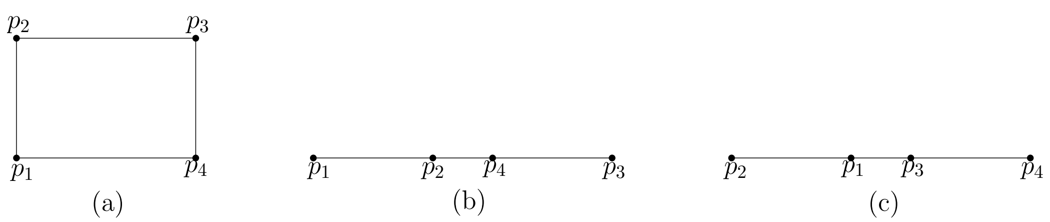


FIGURE 2.3: Two different placements of a parallelogram $p_1p_2p_3p_4$

2.4.1 4-cycle algorithm

For this algorithm, the query graph presented to the adversary in the first round has the structure shown in Fig. 2.4.

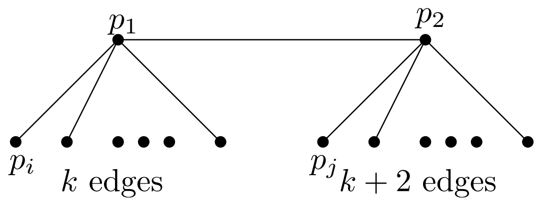


FIGURE 2.4: Query graph for first round in a 2-round algorithm using quadrilaterals.

Making use of the following simple but useful observation,

Observation 1. *At most two points can be at the same distance from a given point p on a line L*

In the second round we query edges connecting pairs of leaves, one from the group of size k and the other from the group of size $k+2$, making quadrilaterals that are not parallelograms (the rigidity condition $|p_1p_i| \neq |p_2p_j|$ ensures that the quadrilateral $p_1p_ip_jp_2$ is not a parallelogram).

2.4.2 5-cycle algorithm

In the 5-cycle algorithm [1], the query graph submitted to the adversary in the first round is shown in Fig. 2.5.

Each five cycle is completed by selecting edges to ensure that the following rigidity conditions are satisfied. For more details on this algorithm see [1].

1. $|p_iq_i| \neq |rs_j|$

2. $|p_iq_i| \neq |s_jt_{jk}|$

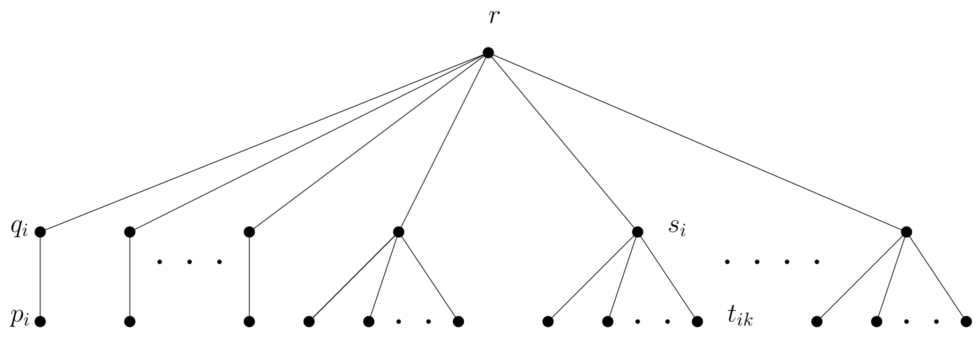


FIGURE 2.5: Query graph for the 5-cycle algorithm

3. $|p_iq_i| \neq ||rs_j| \pm |s_jt_{jk}||$

4. $|s_jt_{jk}| \neq |q_ir|$

5. $|s_jt_{jk}| \neq ||p_iq_i| \pm |q_ir||$

2.4.3 3-path algorithm

In the 3-path algorithm [13], the query graph submitted to the adversary in the first round is shown in Fig. 2.6.

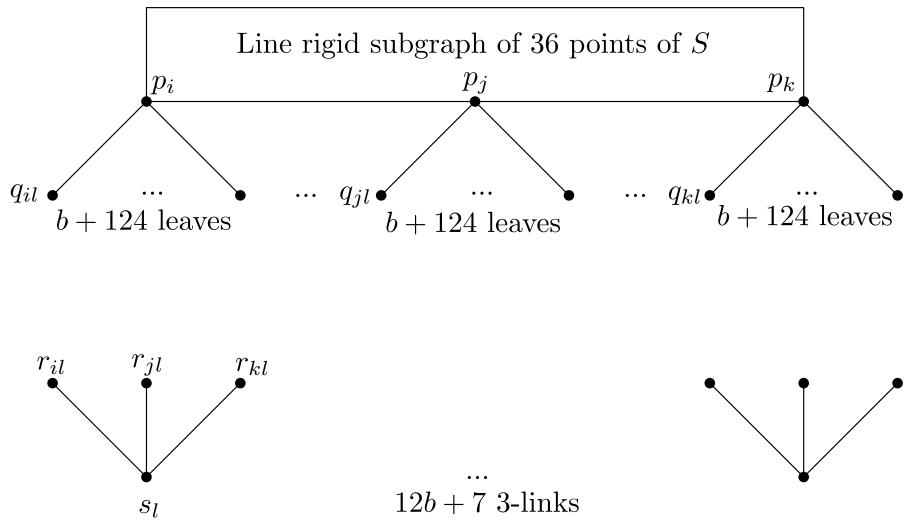


FIGURE 2.6: Query graph for the 3-path algorithm

In the second round, the algorithm select edges suitably to satisfy the following rigidity conditions.

1. $|p_1p_2| \notin \{|r_1s|, |r_2s|, ||r_1s| \pm |r_2s||\}$,

2. $|p_2p_3| \notin \{|r_2s|, |r_3s|, ||r_2s| \pm |r_3s||\}$,

3. $|p_3p_1| \notin \{|r_3s|, |r_1s|, ||r_3s| \pm |r_1s||\}$,

4. $|p_1q_1| \notin \{|r_1s|, |r_2s|, ||r_1s|\pm|r_2s||, ||p_1p_2|\pm|r_1s||, ||p_1p_2|\pm|r_2s||, ||p_1p_3|\pm|r_1s||, ||p_1p_3| \pm |r_3s||, ||p_1p_2| \pm |r_1s| \pm |r_2s||, ||p_1p_3| \pm |r_1s| \pm |r_3s||\}$,

5. $|p_2q_2| \notin \{|r_1s|, |r_2s|, |p_1q_1|, ||r_1s| \pm |r_2s||, ||p_1p_2| \pm |r_1s||, ||p_1p_2| \pm |r_2s||, ||p_2p_3| \pm |r_2s||, ||p_2p_3| \pm |r_3s||, ||p_1q_1| \pm |r_1s||, ||p_1q_1| \pm |r_2s||, ||p_1p_2| \pm |r_1s| \pm |r_2s||, ||p_2p_3| \pm |r_2s| \pm |r_3s||, ||p_1q_1| \pm |r_1s| \pm |r_2s||, ||p_1q_1| \pm |p_1p_2| \pm |r_1s||, ||p_1q_1| \pm |p_1p_2| \pm |r_2s||, ||p_1q_1| \pm |p_1p_2| \pm |r_1s| \pm |r_2s||\}$,

6. $|p_3q_3| \notin \{|r_1s|, |r_2s|, |r_3s|, |p_1q_1|, |p_2q_2|, ||r_2s| \pm |r_3s||, ||r_3s| \pm |r_1s||, ||p_1p_3| \pm |r_3s||, ||p_2p_3|\pm|r_3s||, ||p_1q_1|\pm|r_1s||, ||p_1q_1|\pm|r_3s||, ||p_2q_2|\pm|r_2s||, ||p_2q_2|\pm|r_3s||, ||p_1p_3|\pm|r_1s|\pm|r_3s||, ||p_2p_3|\pm|r_2s|\pm|r_3s||, ||p_1q_1|\pm|r_1s|\pm|r_3s||, ||p_2q_2|\pm|r_2s|\pm|r_3s||, ||p_1q_1|\pm|p_1p_3|\pm|r_3s||, ||p_2q_2|\pm|p_2p_3|\pm|r_3s||, ||p_1q_1|\pm|p_1p_3|\pm|r_1s|\pm|r_2s||, ||p_2q_2| \pm |p_2p_3| \pm |r_2s| \pm |r_3s||\}$.

on each 3-path component shown in Fig. 2.7. For more details on this algorithm see [13].

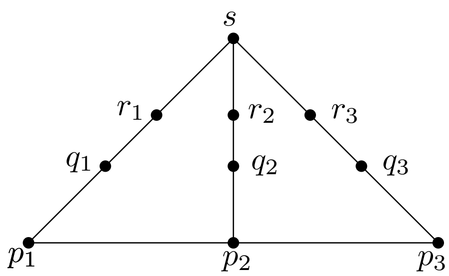


FIGURE 2.7: A 3-path component

2.4.4 Randomized algorithm

Damaschke [14] proposed an incremental randomized algorithm (for an introduction to randomized algorithms see [15]) that expands a set L of points whose positions have been fixed. The set L is initialized by picking an arbitrary point p_0 from S and setting it as the origin of the line on which the points lie. Relative to p_0 a random path $P = p_0p_1p_2...$ is incrementally constucted by choosing a point p_i at random from the set $S - L$, and measuring the distance $d(p_i, p_{i+1})$ for each $i = 0, 1, 2, ...$ Simultaneously, the algorithm maintains all possible signed sums $\pm d(p_0p_1) \pm d(p_1p_2) \pm \cdots \pm d(p_i, p_{i+1}) \cdots$, until for some p_{k+1} the signed sums are no longer all distinct.

If a signed sum that repeats is the actual distance of p_{k+1} from p_0, then the placement of p_k relative to p_{k+1} becomes ambiguous. We stop at this point, query the distance $d(p_0, p_k)$ and use the signed sum equal to this distance to fix the placements on L of all the points on the path from p_1 to p_k (in Damaschke's description the position of p_k is fixed relative to two points in L and the signed sum corresponding to this position is chosen to fix the placements of the other points on the path constructed thus far). Resetting p_k as the new p_0 and p_{k+1} as the new p_1, the algorithm repeats until $L = S$.

Damaschke proved the following result.

Theorem 2.2. *The above randomized algorithm for the point location problem has, for any instance, performance ratio 1 + O(1/ log n) with high probability.*

The term performance ratio is the number of distance queries divided by the number of points.

It is straightforward to turn this into a 2-round algorithm. Fix the placement of 2 points p_0 and p_1 and choose a random path $P = p_1p_2 \dots p_n$ on all the remaining points to be placed and submit this query graph to the adversary. As before, we compute signed sums, stopping when two signed sums are equal when we have reached the point p_{k+1} on P. We resolve the ambiguity in the placement of p_{k+1} by adding edges from p_{k+1} to p_0 and p_1, whose lengths we will query in the second round. Continue as in the incremental algorithm from p_{k+1} on.

2.5 Experimental Results

We implemented all the four deterministic algorithms and the 2-round version of the incremental randomized algorithm, discussed in the previous section. The control parameters used for comparing their performances are: query complexity and time complexity. The results of the experiments for the deterministic algorithms are shown in the graphs below. In our experiments, we simulated an adversary by creating a linear layout and checking the placements of the points by the algorithms against this. This also solved the problem of ensuring a valid assignement of lengths to the queried edges. We will have more to say about this in the next section.

Predictably enough, the above chart shows that the behavior of the algorithms with respect to query complexity is consistent with the upper bounds for these algorithms shown in Table 2.1. Each of these algorithms were run on points sets of different sizes, up to 50000 points.

Clearly, 3-cycle is consistently the fastest; but despite its complex structure the 3-path algorithm does well as compared to the 4-cycle and the 5-cycle algorithms. We have not included the performance of the randomized algorithm in the above graphs as it is incredibly slow and we ran it for point sets of size up to 16,000. Table 2.2 below shows its performance details.

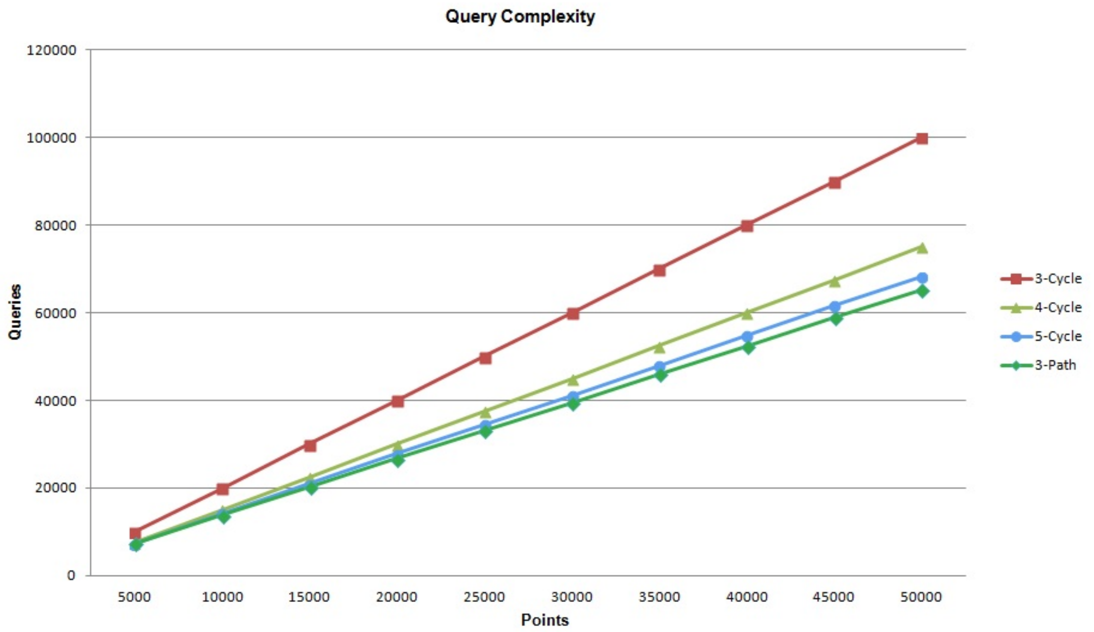


FIGURE 2.8: *Query Complexity Graph*

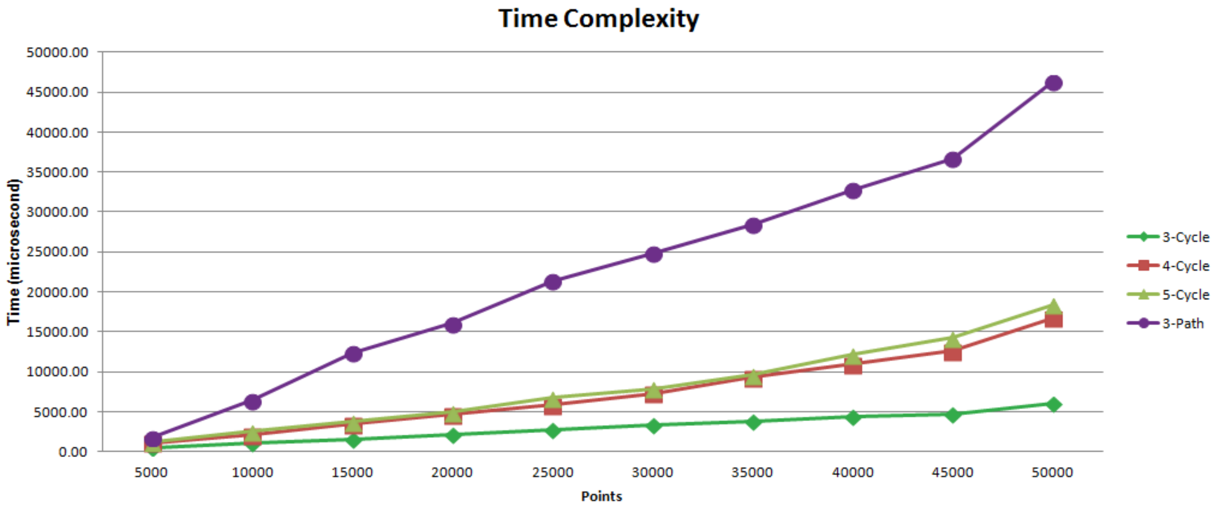


FIGURE 2.9: *Time Complexity Graph*

2.6 Discussion

The behaviour of the deterministic algorithms with respect to time complexity is opposite to their behaviour with respect to query complexity. The growth-rate of the running time versus the size of the input point-set is also near-linear. Both results are as expected.

Number of points	Number of Distance Queries	Running time (hrs:mins:secs)
2000	2382	0:10:41
4000	4712	0:57:32
6000	7048	2:25:38
8000	9348	5:27:53
10000	11668	8:38:34
12000	13999	13:25:24
14000	16282	18:34:58
16000	18625	23:19:40

Table 2.2: Performance of 2-round randomized algorithm

As reported, in none of the deterministic algorithms it was explicitly stated how to obtain an actual layout from the rigid graph constructed on the input point set. In our implementations we devised a signed-sum technique to generate a layout.

The assumption that an assignment of lengths is valid is a strong one and, as mentioned earlier, we circumvented this problem by creating a layout and reporting quried lengths based on this. The correctness of the placements of the points by an algorithm is verified by checking that it generates a layout identical to the one used to report queried lengths.

An algorithmic approach to the solution of this problem is based on constructing the Cayley-Menger matrix out of the squared distances of a query graph.

For a query graph with n vertices, the pre-distance matrix $D = [D_{ij}]$ is a symmetric matrix such that $D_{ij} = d_{ij}^2$, where d_{ij} is the distance between the vertices (points) i and j of the query graph. The Cayley-Menger matrix, $C = [C_{ij}]$ is a symmetric $(n+1) \times (n+1)$ matrix such $C_{0i} = C_{i0} = 1$ for $0 < i \leq n$, $C[0, 0] = 0$ and $C_{ij} = D_{ij}$ for $1 \leq i, j \leq n$ [16], [5].

The vertices of the query graph has a valid linear placement provided the rank of the matrix B is at most 3 (this is a special case of the result that there exists a d-dimensional embedding of the query graph if the rank of B is at most $d+2$; our claim follows by setting $d = 1$) [5].

It's interesting to check this out for the query graph in Fig. 2.10 on 3 points.

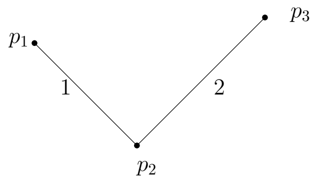


FIGURE 2.10: *A query graph on 3 vertices*

The Cayley-Menger matrix B for the above query graph is:

$$B = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & x^2 \\ 1 & 1 & 0 & 4 \\ 1 & x^2 & 4 & 0 \end{bmatrix}$$

,

where $x = d_{13}$, the unknown distance bewteen the points p_1 and p_3.

By the above result, the 4×4 minor, det(B) = 0. This leads to the equation

$$x^4 - 10x^2 + 9 = 0$$

which has two solutions $x = 3$ and $x = 1$, corresponding to the two possible placements (embeddings) of the points p_1, p_2 and p_3. Assuming p_2 is placed to the right of p_1, in one of these placements p_3 is to the right of both p_1 and p_2; in the other, to the left of them both.

2.6.1 Deterministic versus Randomized

Table 2.2 lends credence to the claim by Damaschke [14] that the number of distance queries of the incremental randomized algorithm is bounded above by $O(n(1 + 1/\log n))$ in the worst case. Unfortunately, it is too slow to be run with very large inputs.

We suspect that the number of times signed sums become equal is intimately connected with the distribution of the points that we generate by pretending to be the adversary. To test this we generated the layout by picking a point at random in a fixed size interval, and picking the next random point in the same fixed-size interval whose left end point is the last point selected. In our experiments we varied this fixed interval from 5 units to 500000 units and reported the number of times we got equal signed sums for points sets of sizes varying from 20 to 1000. Interestingly enough, as can be seen from Table 2.3 below that the numbers decrease as the interval-size increases.

	Range									
# of points	1-5	1-10	1-20	1-50	1-100	$1-10^3$	$1-10^4$	$1-5*10^4$	$1-10^5$	$1-5*10^5$
20	7	7	6	5	4	3	3	2	2	1
50	16	13	11	10	9	7	6	6	5	4
100	25	23	20	19	15	11	9	8	8	7
200	45	39	35	33	29	22	18	17	16	
400	78	70	61	56	49	41	39	34		
1000	167	149	140	123	111	94	82	76		

Table 2.3: Performance of incremental randomized algorithm for nearly uniform distributions

The incremental randomized algorithm is often held up as an example of simplicity in comparison to deterministic algorithms, like the 3-path one, for example. The above experiments paint a completely different picture. From a practical point of view, it is completely ineffective as it is essentially a brute-force algorithm. The 3-path algorithm, on the other hand, scores high on both parameters - low query complexity and low time complexity.

Chapter 3

Recognizing Line Rigid Graphs

3.1 Introduction

Graph recognition problem is very old and researchers have studied various graph recognition problems, like interval graph [17], laman graph ([18], [19]), clique graph [20], cographs [21] , circle graphs [22] and many more. But line rigid graph recognition problem is very new. As far our knowledge no prior work has been done on this problem.

A graph is called line rigid if all the vertices of the graph has unique position on a line up to translation and reflection; and distance between any two point consistent with the graph. Consider a graph $G = (V, E)$ consisting with n vertices and m edges and each edge associated with weights. Now the problem is to assign coordinates to each vertex so that the Euclidean distance between any two vertices is equal to the weight associated with that edge. This is the *graph realization problem* [23]. The *graph realization problem* is about computing relative location of a set of vertices placed in Euclidean space, relying only on inter-vertex distance measurement. *Unique graph realization* refers to exactly one realization of a graph in Euclidean space. A graph that has unique realization must be rigid [23].

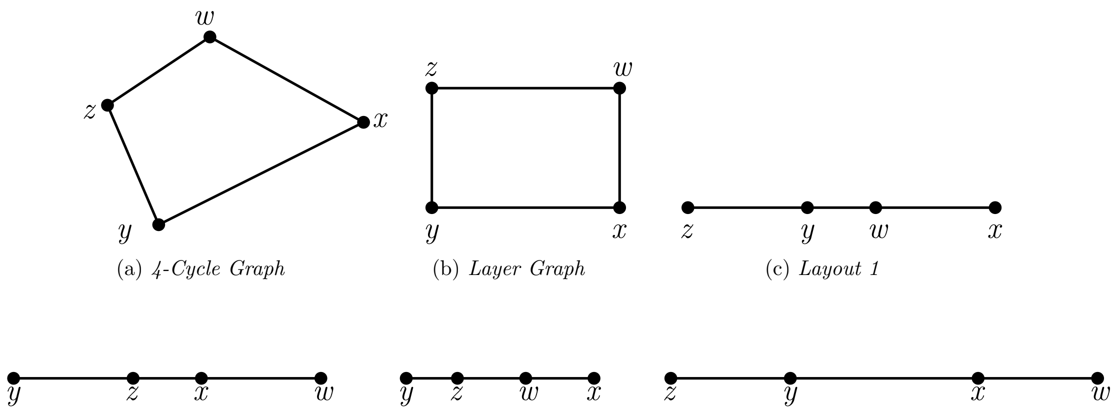
Graphs with unique graph realization in one dimension refers to line rigid graphs. *Hendrickson et. al.* [23] proposed some conditions for unique graph realization in any dimension.

Consider a graph G. If G is line rigid then the vertices of G must have unique placement on a line such that the distances between adjacent nodes are consistent with the corresponding edges of G. The uniqueness of a layout refers to the layouts obtained after translation or reflection is same. According to *Chin et. al.* [1], a graph is line rigid if and only if it can't be drawn as *layer graph*. In *layer graph* representation of a graph G must hold the following conditions,

- all the edges are drawn as horizontal(parallel to x) or vertical(parallel to y) line in *xy-plane*. *i.e.* $\mid v_1 - v_2 \mid = (v_1 - v_2).x$ or $(v_1 - v_2).y$
- length of edge $\mid v_1 v_2 \mid$, is same as the weight of the edge.
- there are two vertices v_1 and v_2 with different *x-coordinates* and *y-coordinates*. *i.e.* $(v_1 - v_2).x \neq \mid v_1 - v_2 \mid$ and $(v_1 - v_2).y \neq \mid v_1 - v_2 \mid$
- when the angle between x and y tends to 0^o or 180^o, no vertices will fall on top of another.

A graph G has two layout if G has *layer graph* drawing. Fig. 3.1 shows four layout of a graph that has *layer graph* representation. Consider xy, yz, xw edges are given for a 4-cycle and zw edge is free. There are four possible placements of node w and z

1. w can be left of x and right of y
2. w can be right of x and y
3. z can be left of y and x



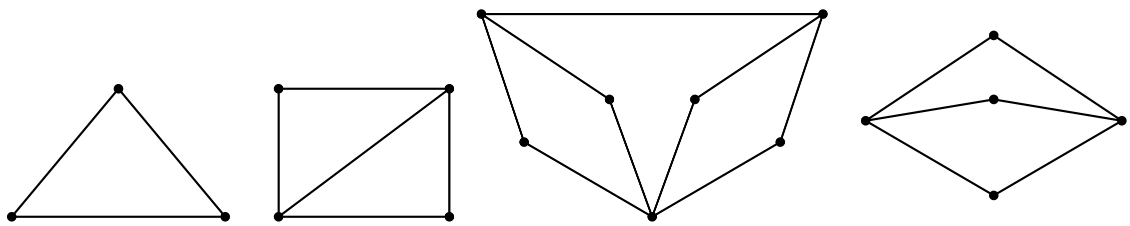
(a) *4-Cycle Graph* (b) *Layer Graph* (c) *Layout 1*

(d) *Layout 2* (e) *Layout 3* (f) *Layout 4*

FIGURE 3.1: Layout of a 4-cycle graph

4. z can be right of y and left of x

Line rigid graph doesn't have layer graph representation. Some line rigid graphs are shown in Fig. 3.2. We can't draw these graphs as layer graph. Now the line rigid graph recognition problem is reduced to layer graph drawing problem. In this chapter our goal is to find out whether a given graph has layer graph drawing or not. If the graph doesn't have layer graph drawing, then the graph recognized as a line rigid graph.



(a) Triangle *(b)* K_4^- *(c) Jewel* *(d)* $K_{2,3}$

FIGURE 3.2: Line rigid graphs

3.2 Motivation

This problem is motivated from one-dimensional rigidity problem known as point placement problem. Line rigid graph is widely used in the point placement problem. The point placement algorithm finds the position of n distinct point on a line, up to translation and reflection with fewest possible pairwise distance queries. The point placement algorithm creates a point placement graph after making pairwise distance queries. A point placement graph is line rigid for some valid assignment of edge lengths. Few interesting question arises from the point placement problem,

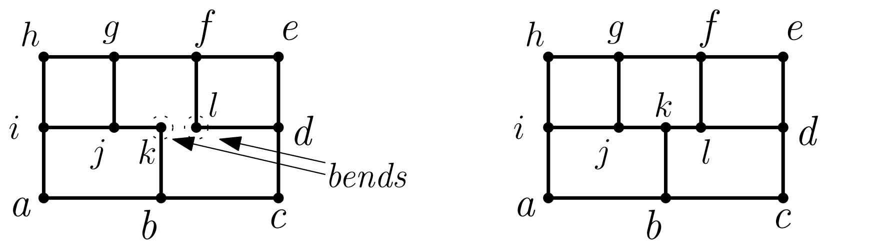
1. How to generate arbitrary line rigid graphs ?
2. How to verify line rigid graphs based on the structural property ?

In this thesis we are concentrating on the verification problem.

3.3 Main Idea

Our main idea is based on the layer graph representation of line rigid graph presented by *Chin et. al.* [1]. Layer graph drawing of a graph is same as the rectangular grid drawing. In rectangular grid drawing the faces of a graph are drawn as rectangular shape where each vertex is located on a grid point and each edge is drawn as a horizontal or vertical line segment. There are two difference between rectangular drawing and layer graph drawing. In rectangular drawing there are no bends in the inner subgraph but in layer graph bends are allowed. Furthermore in layer graph no vertices will fall on top of another one when the angle between the vertical and horizontal edges are 0^o or 180^o. But there are no such restriction on rectangular drawing. In Fig. 3.3 shows that difference. Our intension is to find a rectangular drawing of arbitrary graph and ensure that no vertex will fall on top of another vertex when the angle between *x-coordinate* and *y-coordinate* is 0^o or 180^o

Our proposed scheme recognizes arbitrary line rigid graphs with some property: the graph must be *2-connected*, planar and maximum degree of vertices are three. A connected graph is *2-connected* if the graph remains connected after



(a) *valid layer graph but not rectangular drawing*

(b) *valid layer graph and rectangular drawing*

FIGURE 3.3: Difference between layer graph drawing and rectangular drawing

removing at most one vertex. On the other way we can say that at least two vertices needed to remove to disconnect the graph. At first we will create an planar embedding of the given graph G. A graph may have exponential number of plane embedding and all the embeddings may not have rectangular drawing. If any one of the embedding has a rectangular drawing then we consider G has rectangular drawing. It is inefficient to check all the embeddings. To solve this issue three more embeddings are created wisely from the first embedding so that we can decided whether the graph has rectangular drawing or not by checking these four embedding only. After creating the embeddings four vertices are chosen appropriately from the outer face of an embedding. These four vertices represents four corner of a rectangle and all other edges and vertices are either on the boundary or inside of the rectangle. Then the subgraph inside the rectangle drawn as rectangles faces.

3.4 Recognition Scheme

We consider the arbitrary graph G is 2-connected, planar and maximum degree of any vertex is three. The adjacency list of G is known. Now we will propose a scheme that recognizes whether G is line rigid or not. The scheme is divided into four phases.

Phase 1: *Initial Pruning*

Phase 2: *Generate a planar Embedding(* Γ*) of G*

Phase 3: *Rectangular Drawing of* Γ

Phase 4: *Layer Graph representation of Rectangular Drawing*

The overview of the whole process is visualized in Fig. 3.4. Initially the adjacency list of a graph G with 31 vertices is given as shown in Fig. 3.4(a). After that we create an planar embedding Γ of G shown in Fig. 3.4(b). There can be many embeddings of G. So we create four possible embeddings wisely based on some structural characterization of Γ. Then we pick each embeddings one by one and try to obtain rectangular drawing. To obtain rectangular drawing we need to chose four vertices in the outer face of Γ as shown in Fig. 3.4(c). These four vertices will be the corners of rectangle. Considering those four vertices as four corners of a rectangle we obtain the rectangular drawing of inner subgraph as shown in Fig. 3.4(d). After obtaining the rectangular drawing we adjust the horizontal edges to ensure that no vertices will fall on top of each other when the angle between horizontal lines and vertical lines are 0^o or 180^o. Now the modified rectangular drawing satisfies all the characteristics of layer graph drawing. The layer graph drawing of G is shown in Fig. 3.4(e).

3.4.1 Initial Pruning

In this phase we will check some structural property for which there no layer graph drawing is possible. The graph has to be connected. Triangulated graphs can't be drawn as a layer graph as they are line rigid. So if the given graph is triangulated then we can say that the graph G is line rigid. Some triangulated graphs are shown in Fig. 3.5.

Triangles are intrinsically line rigid and layer graph drawing is not possible. If all the faces are triangular in a graph then we can't draw it as layer graph.

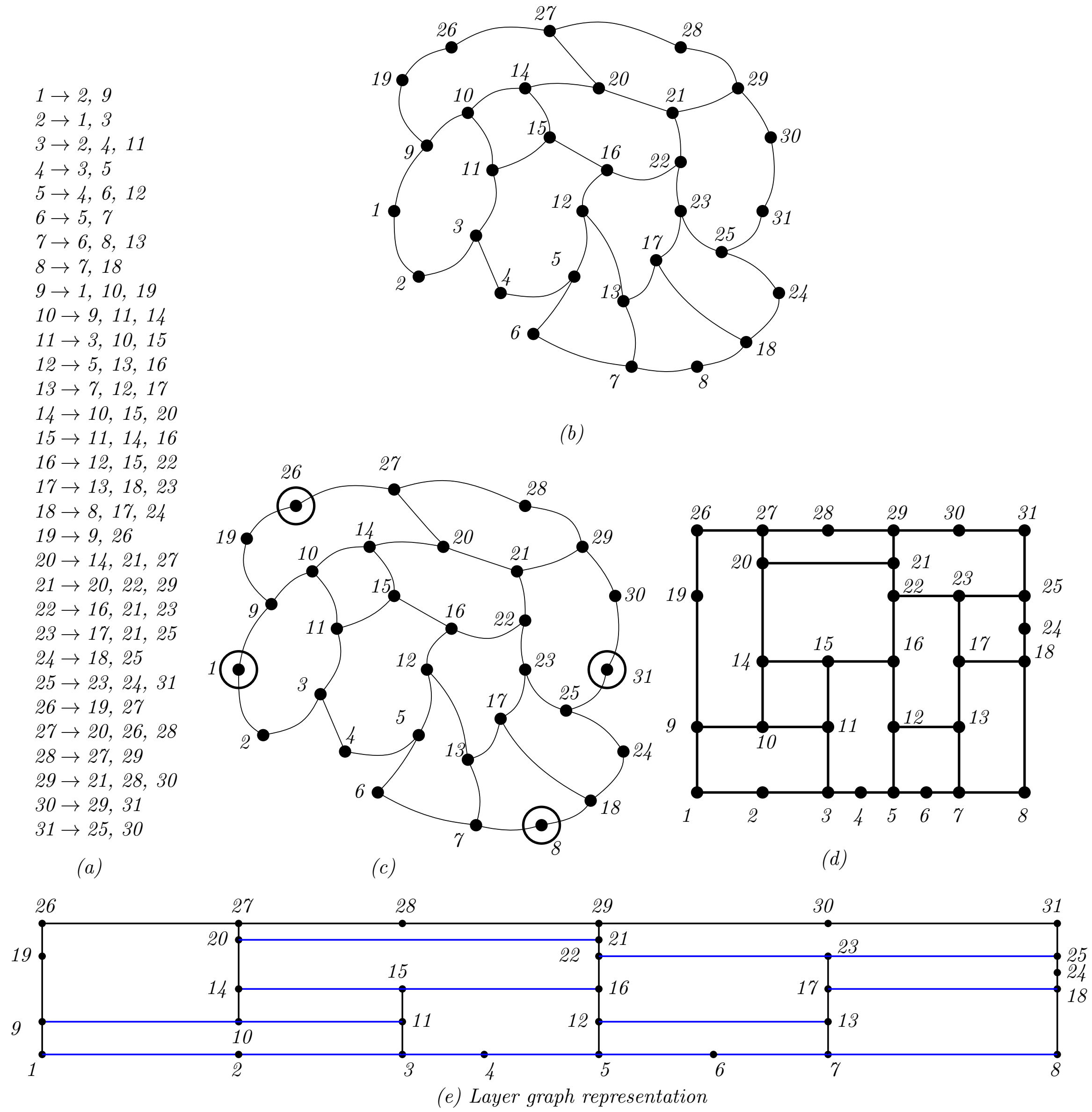


Figure 3.4: (a) Adjacency list of G (b) Embedding of G (c) Appropriate four vertices are chosen as corners (d) Rectangular drawing (e) Layer graph representation

Depth-first search can be used to check for any cycle with length three. The back edges in the depth-first search represent cycle. Our intension of check cycles with length three. For each back edge we will look for common adjacent vertex from the vertices of the back edge. A common vertex from end vertices of a back edge represents a cycle with length three. There can be more than one common vertex if the back edge is part of more than one cycles of length three. Two common

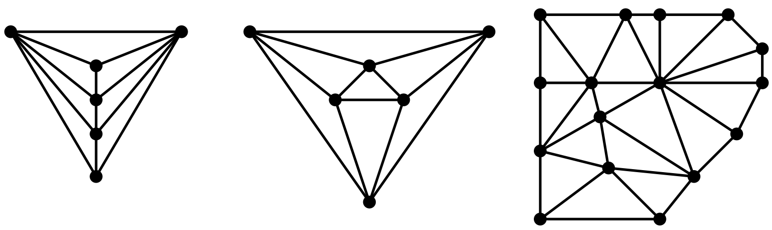
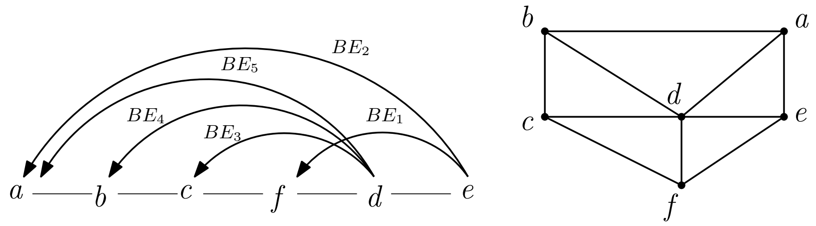


FIGURE 3.5: Triangulated graphs

vertex of a back edge ensures existence of K_4^-. Fig. 3.6 shows clear representation of the process. Fig 3.6 (a) shows the adjacency list of the given graph, (b)

$a \to b,\ d,\ e$
$b \to a,\ c,\ d$
$c \to b,\ d,\ f$
$d \to a,\ b,\ c,\ e,\ f$
$e \to a,\ d,\ f$
$f \to c,\ d,\ e$

(a) Adjacency List of G



(b) Depth-first Search *(c) Graph G*

FIGURE 3.6: Check 3-cycles in G

represents an random Depth-First search tree where the bold edges with arrow are back edges. There are five back edges BE_1, BE_2, BE_3, BE_4 *and* BE_5. The end vertices f and e of back edge BE_1 has a common vertex d. Hence there is 3-cycle, def associated with BE_1. The same way back edge BE_2 is associated with 3-cycles, ade. Here all the back edges associated with at least one 3-cycle which shows that all the faces of G are triangular. Hence G is a triangulated graph.

3.4.2 Generate an Planar Embedding

An embedding of a graph G on a surface is a representation of G where the points are associated to vertices and arcs are associated to edges. An embedding is planar if no two arcs intersect geometrically. Graph G_1 in Fig. 3.7(a) has a planar embedding but G_2 in Fig. 3.7(b) does not have a planar embedding.

In this phase we will create an planer embedding of an arbitrary graph G using the algorithms given by *Hopcroft et. al.* [24] and *Mehlhorn et. al.* [25]. Hopcroft presented a efficient algorithm to testing planarity of an given graph and also provided idea to create an embedding if exists. Later in 1996 K. Mehlorn and P. Mutzel provided a detailed description of the embedding phase of Hopcroft and Tarjan planarity testing algorithm. Hopcroft used the original idea provided by *Bergs et. al.* [26] and *Auslander et. al.* [27]. Bergs provided a characterization of planar graph: a graph is planar if and only if all of its biconnected components are planar. Auslander and Parter has given a idea of planar graph embedding. First, find a cycle in a graph. After removing the cycle the graph falls into several pieces. Then an algorithm will recursively embed each piece in the plane with the original cycle. Lastly all the embeddings of the pieces are merged. If merging is possible then the embedding of the graph is found otherwise embedding is not possible.

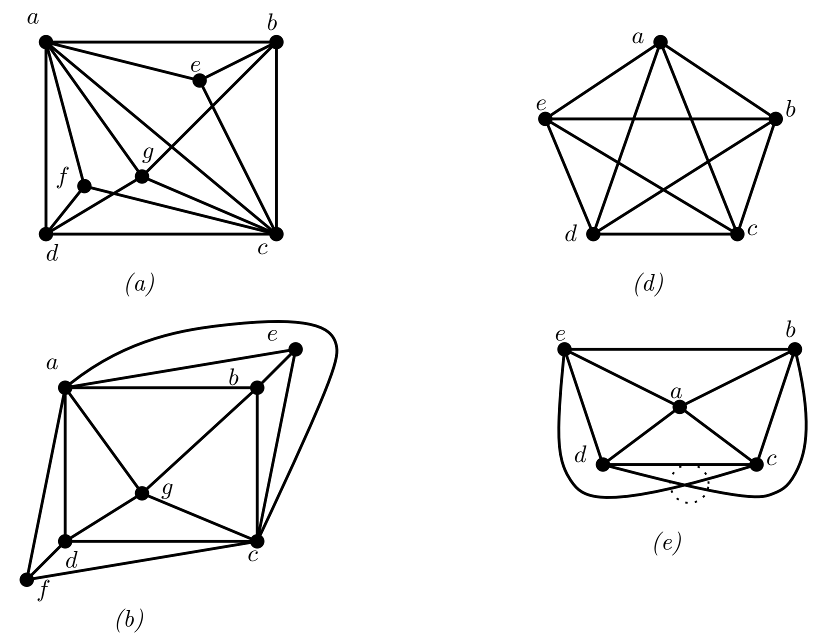


FIGURE 3.7: (a) Graph G_1 (b) Planar embedding of G_1 (c) Graph G_2 (d) Planar embedding is not possible for G_2

In the first step we will check that the number of edges in given graph $G(V, E)$ does not exceed the edge restriction of planar graph. The number of edges in a planar graph must be $\leq 3|V| - 3$. If there are more edges in G then we decide that G is nonplanar. In the next step, G will be divided into biconnected components. Then test planarity of each biconnected components and embed them.

A graph $G = (V, E)$ with a set of finite number of vertices V and a set of finite number of edges E. A graph $G = (V_1, E_1)$ is a subgraph of $G = (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. There is a path from vertex v_i to v_j if there exists a sequence of vertices v_x, $i \leq x \leq j$, and edges e_y, $i \leq y \leq j$, such that $e_i = (v_i, v_{i+1})$. A path from a vertex to itself is a closed path. A closed path from vertex v_i to v_i with one or more edges is a *cycle* if all of its edges are distinct and only the vertex v_i repeated twice.

The edges of depth-first tree are classified into four types: *tree edge*, *back edge*, *forward edge* and *cross edge*. During a DFS execution, the classification of edge (u, v), the edge from vertex u to vertex v, depends on whether we have visited v before in the DFS and if so, the relationship between u and v.

1. If v is visited for the first time as we traverse the edge (u, v), then the edge is a *tree edge*.
2. Else, v has already been visited:
   (a) If v is an ancestor of u, then edge (u, v) is a *back edge*.
   (b) Else, if v is a descendant of u, then edge (u, v) is a *forward edge*.
   (c) Else, if v is neither an ancestor or descendant of u, then edge (u, v) is a *cross edge*.

In Fig. 3.8 *tree edges* are in bold, *back edges* are E_{eb}, E_{hc}, *forward edge* E_{eh}, and *cross edge* E_{ae}. For our flexibility we will consider two type of edges, *tree*

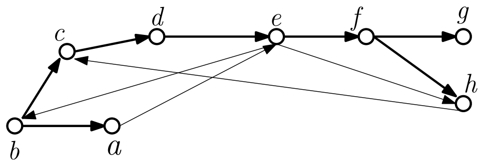
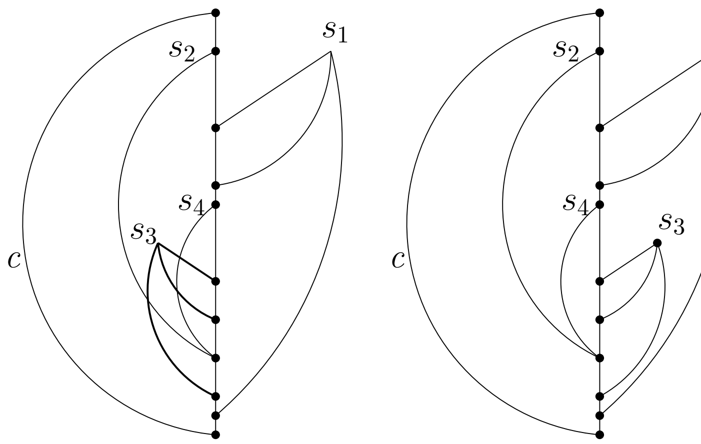


FIGURE 3.8: Edges of a DFS tree

edges and *back edges*. Here we define *back edge* as an edge going from currently visited vertex to any previously visited vertex.

A cycle in DFS tree consists of *tree path* and *back edge*. Tree path is a path between two vertices that uses *tree edges* only. Some cycle will consist of simple path of edges not in previously found cycles, plus a simple path edges in old cycles. Now consider the first cycle c. It will consist of a sequence of tree edges followed by one back edge in path P. The numbering of vertices along the cycle are ordered. Each piece not part of a cycle will consists either of a single *back edge* (v, w) or of a *tree edge* (v, w) plus a subtree with root w, plus all *back edges* which lead from the subtree. Then each piece will be processed and add them to a planar representation in decreasing order of v. Each piece can be either *inside* or *outside* of cycle c. We continue to add new pieces and move old pieces if necessary until either a piece cannot be added or the entire graph is embedded in the plane. Fig. 3.9 shows the conflict between piece S_2, S_3 and S_4. Two ways to resolve the conflict between S_2 and S_3: (1) we can move S_2 outside c (2) or we can move S_3 outside c. And two ways to resolve conflict between S_3 and S_4: (1) move S_4 outside c (2) move S_3 outside c. In both case moving S_3 outside c resolve the conflicts. Fig. 3.9(b) shows the embedding after removing the conflict.

At first a depth-first tree will be generated. Depth-first tree can be achieve by traversing the graph using depth-first tree traversal algorithm. A sequence of number are assigned to each vertex as they are visited. The numbers must be in incremental sequence and no numbers will be repeated. Directions are added to the edges to represents the traversal sequence. Fig. 3.10 (a) shows the adjacency



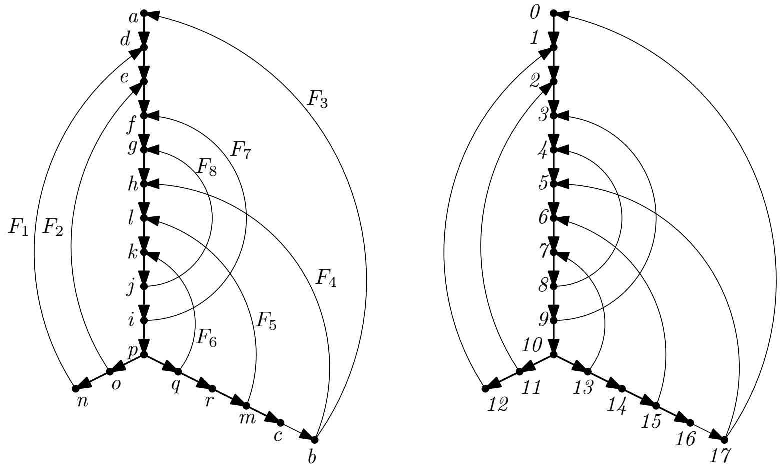
(a) *(b)*

FIGURE 3.9: Conflicts in DFS tree

list of a graph G, Fig. 3.10 (b) represents the *DFS* tree. In the *DFS* tree the *back edges* are F_1, F_2, F_3, F_4, F_5, F_6, F_7 and F_8 and the *tree edges* are in bold. Fig. 3.10(c) represents the vertex numbering in the *DFS* tree.

$a \rightarrow b,\ d$
$b \rightarrow a,\ c,\ h$
$c \rightarrow b,\ m$
$d \rightarrow a,\ e,\ n$
$e \rightarrow d,\ f,\ o$
$f \rightarrow e,\ g,\ i$
$g \rightarrow f,\ h,\ j$
$h \rightarrow b,\ g,\ l$
$i \rightarrow f,\ p,\ j$
$j \rightarrow g,\ i,\ k$
$k \rightarrow j,\ l,\ q$
$l \rightarrow h,\ k,\ m$
$m \rightarrow c,\ l,\ r$
$n \rightarrow d,\ o$
$o \rightarrow e,\ n,\ p$
$p \rightarrow o,\ i,\ q$
$q \rightarrow p,\ k,\ r$
$r \rightarrow q,\ m$

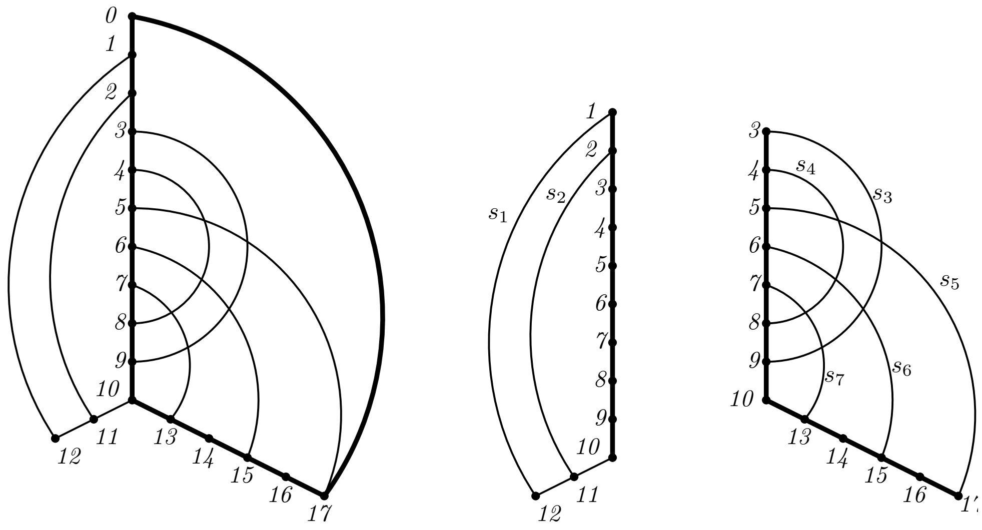
(a) Adjacency list of G



(b) DFS Tree of G *(c) Vertex numbering in DFS Tree*

FIGURE 3.10: DFS tree of a graph

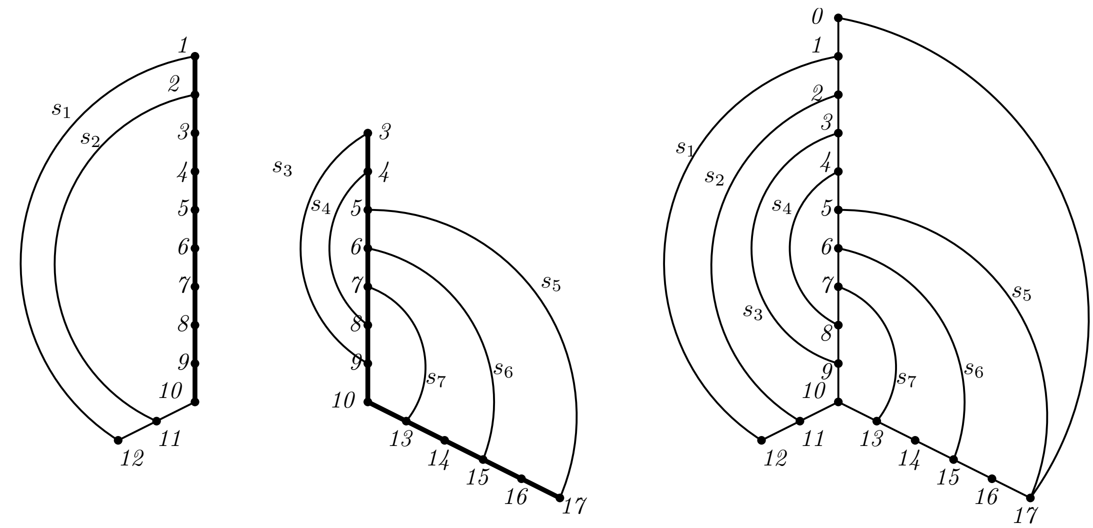
Now the largest cycle is choosen and remove from the *DFS* tree. In Fig. 3.11(a) shows the largest cycle in bold lines. After removing the cycle divides the *DFS* tree in two subtrees, T_1 and T_2, shown in Fig. 3.11(b) and Fig. 3.11(c). For each subtree we will follow the same process: choose the largest cycle, remove it and embed the pieces or subtrees. In subtree T_1 there are two pieces s_1 and s_2 and in subtree T_2 there are five pieces, s_3, s_4, s_5, s_6, s_7.



(a) Largest cycle is chosen *(b) Subtree T_1* *(b) Subtree T_2*

FIGURE 3.11: Subgraphs after removing largest cycle from a graph

In subtree T_1 there is no conflict between the pieces. But in T_2 there are conflict between s_3, s_4 and s_5, s_6, s_7. Fig. 3.12 (a) and (b) shows the embedding of the subtrees and Fig. 3.12(c) is obtained by combining the embedding of each subtree to the largest cycle.



(a) Embedding of T_1 *(b) Embedding of T_2* *(c) Combined the embedding of T_1 and T_2*

FIGURE 3.12: Embedding of subtrees and combine the embedding to get entire embedding of G

3.4.3 Rectangular Drawing of an Embedding

In this phase we will generate rectangular drawing of G from an embedding using the algorithm given by *M.S Rahman et. al.* [28]. Before starting the drawing algorithm we will go through two preprocessing steps: *(a) analyze embedding*[29] and *(b) choose four corner vertices*[30]. After these two steps we will engage the drawing algorithm with a fixed embedding and four corner vertices in the outer face of the embedding.

3.4.3.1 Analyze Embedding

First we will define few structural property of a graph which are necessary to understand this phase. Suppose we have given a graph, $G(V, E)$ where V is the set of vertices and E is the set of edges. The degree of a vertex, v is represented as $d(v)$. In a *cubic* graph all the vertices are degree three vertices. A graph G is *k-connected* if deletion of at least k vertices makes the graph disconnected. In a *2-connected* graph a pair of vertices can disconnect a graph and such pair of vertices called *separation pair* in G. There is no *separation pair* in *3-connected* graph.

Consider a path, $P = v_0, v_1, v_2, \ldots\ldots v_{k+1}, k \geq 1$, in G. In P, $d(v_0) \geq 3$, $d(v_2) = d(v_3) = \ldots = d(v_k) = 2$, $d(v_{k+1}) \geq 3$. The subpath $P' = v_1, v_2, \ldots\ldots v_k$ is called a *chain* of G and the vertices v_0 and v_{k+1} are the *supports* of the chain P'. The graph shown in Fig 3.13(c) has four chains: P_1, P_2, P_3 and P_4.

Subdividing an edge is to add one or more vertices of degree two between the end vertices the edge and remove the old edge. Subdivision of a graph is shown in Fig. 3.13(d). Consider we have added a set of vertices, $S = \{w_0, w_1, ...w_k\}$. Then there is a path from u to v that pass through all the vertices S and $d(w_i) = 2$,

$0 \leq i \leq k$. A graph is called *cyclically 4-edge-connected* if the removal of any three or fewer edges leaves a graph such that exactly one of the connected components has a cycle. Fig. 3.13(a) is cyclically 4-edge connected but not Fig. 3.13(b).

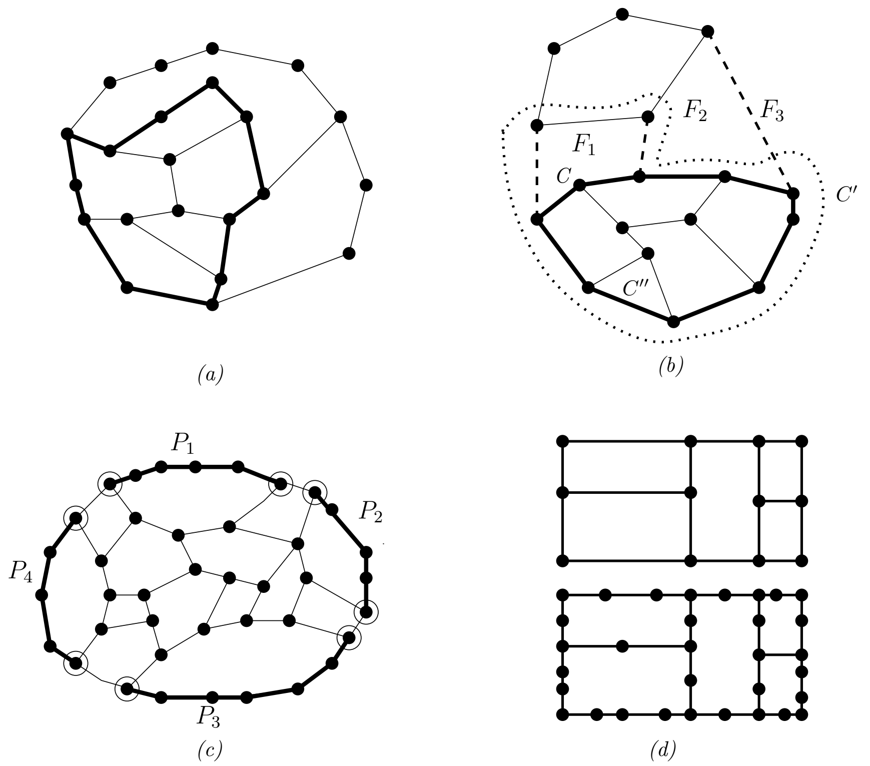
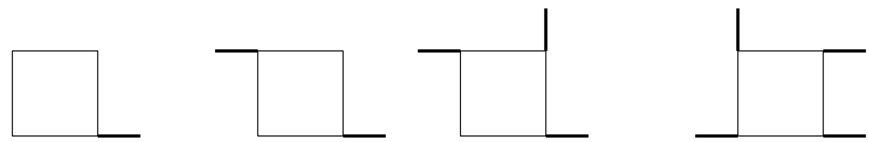


FIGURE 3.13: (a) Cyclically 4-edge connected graph (b) Not cyclically 4-edge connected (c) Four chains in a graph (d) Subdivision of a graph

Consider a planar biconnected graph G and Γ is the embedding of G. The contour of a face in Γ is a cycle of G and called a $face$. The outer $face$ of Γ is represented by $F_0(\Gamma)$. Let C be a cycle in Γ. The plane subgraph of Γ inside C (including C) is denoted by *inner subgraph* $\Gamma_I(C)$ for C and the plane subgraph of Γ outside C(including C) is denoted by *outer subgraph* $\Gamma_O(C)$ for C. An edge connects to exactly one vertex of a cycle C and other end of the edge located outside C is called a *leg* of C. The vertex of C to which a *leg* is connected is called *leg-vertex* of C. A cycle C is called *k-legged cycle* of Γ if C has exactly k *legs* and there is no edge which joins two vertices on C. Fig. 3.14 the *legs* are in bold. In Fig. 3.13(a) and (b) a *3-legged* cycle is marked with thick solid lines.



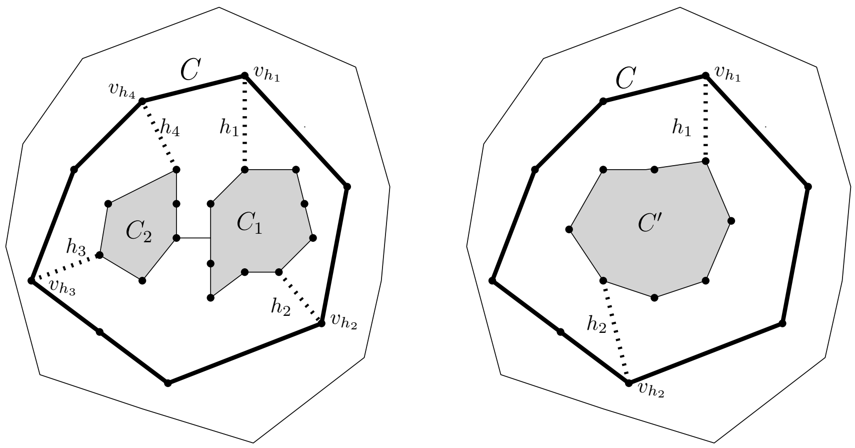
(a) *1-legged cycle* (b) *2-legged cycle* (c) *3-legged cycle* (d) *4-legged cycle*

FIGURE 3.14: Legged Cycles

A face F of Γ is called *peripheral face* for a 3-legged cycle C in Γ if F is in $\Gamma_O(C)$ and the contour of F contains an edge on C. In any embedding Γ for any *3-legged* cycle will have three peripheral faces. In Fig. 3.13(b) there are three peripheral faces for a *3-legged cycle* drawn in thick solid lines. A *k-legged* cycle C is called a *minimal k-legged cycle* if $G_I(C)$ does not contain any other *k-legged* cycle of G. Cycle C in Fig. 3.13(b) is not *minimal 3-legged cycle.* But C'' is *minimal 3-legged cycle.* Consider two cycles C and C' in Γ are called independent if there is no common vertex among their *Inner subgraphs.* A cycle C in Γ is called *regular* if the plane graph $\Gamma - \Gamma_I(C)$ has a cycle. In the plane graph depicted in Fig. 3.13(b), the cycle C drawn by thick solid lines is a *regular 3-legged cycle*, while the cycle C' indicated by thin dotted lines is not a *regular 3-legged cycle.* The 2-legged cycle represented by a thin dotted line in Fig. 3.13(a) is not regular. A *2-legged* cycle C in Γ is not regular if and only if $\Gamma - \Gamma_I(C)$ is a chain of G and a *3-legged* cycle C is not regular if and only if $\Gamma - \Gamma_I(C)$ contains exactly one vertex that has degree 3 in G.

An edge $e_{u,v}$ of an embedding Γ which connects to exactly one vertex in the contour of a cycle C in Γ and located inside C is called an *hand* of C and the vertex of C to which *hand* connects is called *hand-vertex* of C. In Fig 3.15(a) cycle C represented in thick solid line has four hands, h_1, h_2, h_3 and h_4 and the hand vertices are v_{h_1}, v_{h_2}, v_{h_3} and v_{h_4}.

In a *2-handed* cycle C there will be exactly two hands in Γ and there will be no edge which joins two vertices on C and located inside C. A *2-handed* cycle C called a *regular 2-handed* cycle if $\Gamma - \Gamma_O(C)$ contains a cycle. Every *2-handed*



(a) *(b)*

FIGURE 3.15: (a)A cycle C with four *hands* (b) *Regular-2-handed* cycle C and *regular-2-legged* cycle C'

cycle is associated with a *2-legged* cycle. In Fig. 3.15(b) cycle C drawn by thick solid line is a *2-handed* cycle and there is a *2-legged* cycle C' inside C.

So far we have discussed about all the necessary terms which will be used to analyze the given embedding. The goal to analyze the embedding to show its structural property for which there is no rectangular drawing is possible and if we don't find those property then we will move to next preprocessing phase.

According to *T. Nishizeki et. al.* [31] if G is a subdivision of a 3-connected planar graph then their is exactly one embedding of G for each face embedded as the outer face. He also mentioned that for any two plane embedding of a graph, any face cycle in one embedding will also represent a face cycle in another embedding. Then multiple embeddings of G are possible by choosing each face as outer face. *M.S Rahman* [29] provided following necessary and sufficient condition for a graph to have a rectangular drawing.

Theorem 3.1 (*M.S Rahman et. al. [29]:*)**.** *Consider a G is a subdivision of a planar 3-connected cubic graph and Γ is an arbitrary plane embedding of G.*

1. *Let, G is cyclically 4-edge-connected, that is, Γ has no regular 3-legged cycle. Then the planar graph G has a rectangular drawing if and only if Γ has a*

face F such that

(a) F contains at least four vertices of degree 2.

(b) there are at least two chains on F, and

(c) if there are exactly two chains on F, then they are not adjacent and each of them contains at least two vertices.

2. *Let, G is not cyclically 4-edge connected. Then the embedding Γ of G has a regular 3-legged cycle C. Let F_1, F_2 and F_3 are three peripheral faces for C, and let Γ_1, Γ_2, and Γ_3 be the plane embeddings of G taking F_1, F_2, and F_3, respectively, as the outer face. Then the planar graph G has a rectangular drawing if and only if at least one of the three embeddings Γ_1, Γ_2, and Γ_3 has a rectangular drawing.*

Algorithm: Theorem 3.1 leads to an algorithm to determine if an embedding of a graph has a rectangular drawing. We will start with determining whether the given graph G is a subdivision of a planar 3-connected cubic graph. Given a graph G and one of its arbitrary embedding Γ. First we will find a regular *2-legged* cycle in Γ. If there is no regular *2-legged* cycle in Γ then G is a subdivision of a planar 3-connected cubic graph. Then we call another subroutine *subdivision drawing* to check existence of rectangular drawing of G and if rectangular drawing exists, then we move to next preprocessing phase, *choose four corner vertices.*

Subdivision drawing: Let Γ be any embedding on a graph G. First, we will check if there is any regular *3-legged* cycle in Γ. If Γ contains a regular *3-legged* cycle, then G is cyclically 4-edge-connected. If G is cyclically 4-edge-connected, then find a face F in Γ that satisfies conditions (a), (b) and (c) in Theorem 3.1-(1). If such face F is not found then G has no rectangular drawing. If Γ has such face F then create another embedding Γ' of G whose outer face is F. Then we move to next preprocessing phase *choose four corner vertices* with graph G and the embedding Γ. Now consider that G is not cyclically 4-edge-connected and Γ has a regular *3-legged* cycle, C. Next find three peripheral faces F_1, F_2 and F_3 of C. For each perifheral face create a new embedding of G where the face is the outer face in G. Lets Γ_1, Γ_2 and Γ_3 are three new embeddings. Then for each new embedding we go throught next preprocessing phase, *choose four corner vertices.* If none of the embeddings pass through next phase then we conclude that G has no rectangular drawing.

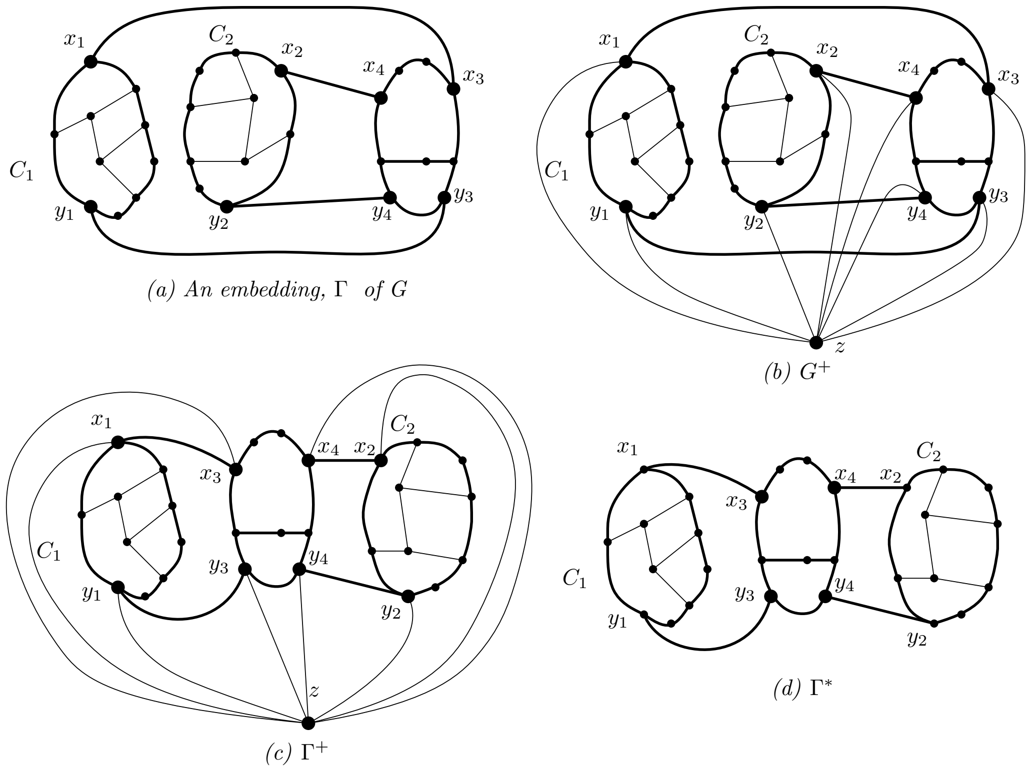


FIGURE 3.16: (a) Γ, Embedding of a graph G (b) Add dummy vertex z and dummy edges (x_i, z) and (y_i, z) (c) z is embedded on outer face of G^+ (d) Removed dummy vertex and dummy edges

If there are regular *2-legged* cycle in Γ then G is not a subdivision of a planar 2-connected cubic graph. Suppose total l of regular *2-legged* cycles and the pair of *leg-vertices* of regular *2-legged* cycles or *hand-vertices* of regular *2-handed* cycles are (x_i, y_i), $1 \leq i \leq l$. Then create a new graph G^+ from G by adding a dummy vertex z and dummy edges (x_i, z) and (y_i, z); $1 \leq i \leq l$. Then we will test planarity of the of G^+. If G^+ is not planar, then we conclude that G has no rectangular drawing. Otherwise find an embedding Γ^+ of G^+ such that z is embedded on the outer face. Create new embedding Γ^* by deleting the dummy vertex z and dummy edges (x_i, z) and (y_i, z). If there are three or more independent *2-legged* cycles in Γ^* then we conclude that G has no rectangular drawing. Otherwise consider that Γ^* has two minimal regular *2-legged* cycles, C_1 and C_2. Then create four plane embeddings Γ_1, Γ_2, Γ_3 and Γ_4 from Γ^* by flipping $\Gamma^*_I(C_1)$ and $\Gamma^*_I(C_2)$ around the *leg-vertices*. Now for each embedding we move to next preprocessing step, *choose four corner vertices*. If none of the embedding passes through the preprocessing step then we conclude that G has no rectangular drawing.

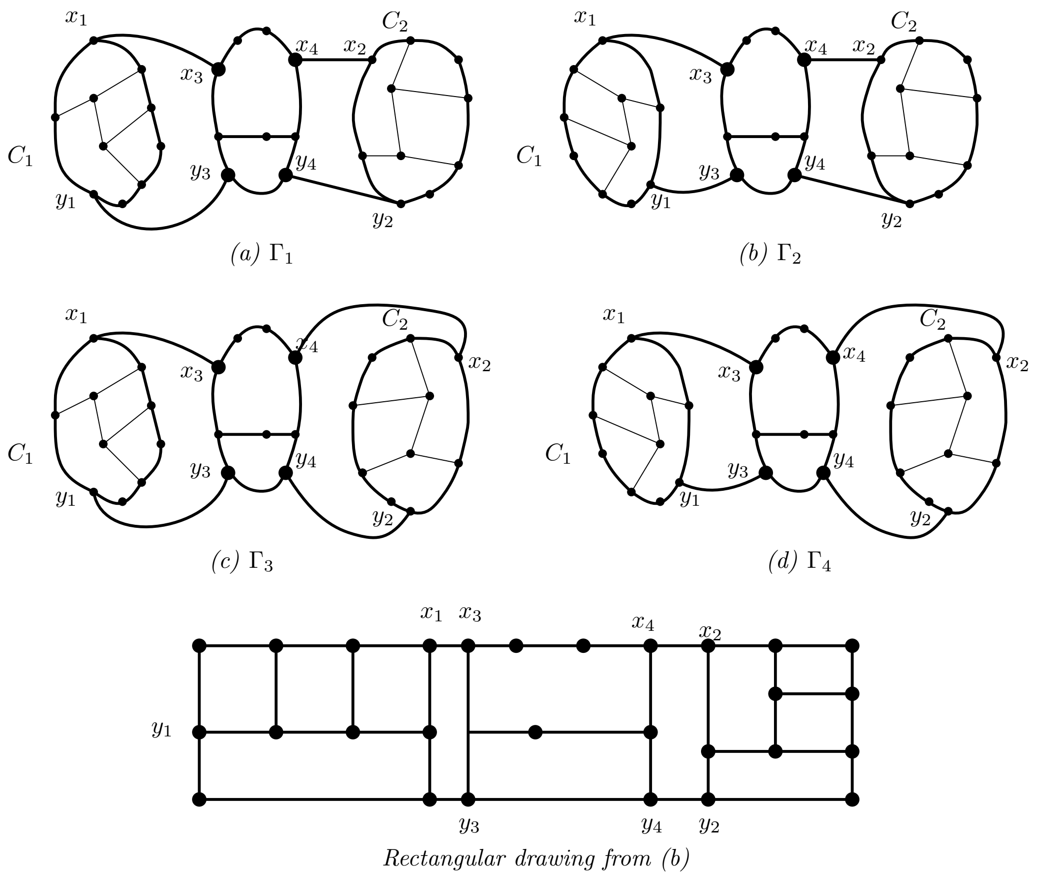


FIGURE 3.17: (a)-(d) Four embedding created by flipping C_1 and C_2 around leg vertices (e) Rectangular drawing obtained from Γ_2

3.4.3.2 Choose four corner vertices

In this preprocessing phase we will use the algorithm given by *M.S Rahman et. al.*[30] to find appropriate corner vertex. The algorithm works on a special kind of graph called *2-3 plane graph*. A *2-3 plane graph* is a plane graph and each vertex has degree 3 except the vertices on outer face which have degree 2 or 3. The idea behind choosing four corner vertex prior to drawing is to check if we can draw the outer face of a graph as a rectangle. It is important to choose right vertices as corners otherwise it is not possible to draw G in rectangular shape. In Fig 3.18(b) corner vertices are chosen correctly but in Fig 3.18(c) the corner vertices are inappropriate and rectangular drawing is not possible.

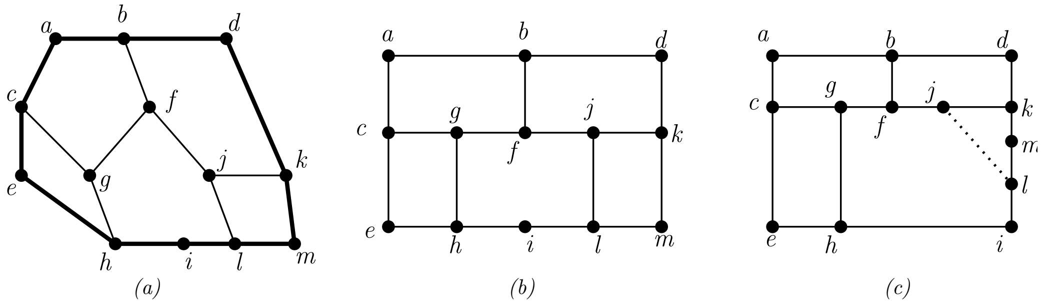


FIGURE 3.18: (a) Γ, Embedding of a graph G (b) Rectangular drawing of G (c) Wrong corner vertices

A *contour* of a face F is defined by taking a clockwise cycle formed by the edges on the boundary of the face. The *contour* of the outer face of a graph G by $C_0(G)$. Any vertex which is not on $C_0(G)$ is called *inner vertex* of G. In Fig 3.18(a) $C_0(G)$ is drawn as thick solid line and the *inner* vertices are f, g and j. In a rectangular drawing of a graph G, each faces are represented as a rectangular shape. The contour $C_0(G)$ of the outer face of G should be rectangular and has four convex corners. But there will be no convex corners or bend in the inner faces. Since rectangular drawing D of G has no bend in the inner faces, only the vertices of degree 2 of G can be drawn as a corner in D and should be on $C_0(G)$. Therefore, at least four vertices of degree 2 are needed on $C_0(G)$ to obtain rectangular drawing of G. A $C_0 - component$ is a subgraph obtained from G after removing $C_0(G)$. Alternatively we can say that if H is a single $C_0-component$ of a graph G, then $G = H \cup C_0(G)$. The graph G in Fig 3.19 has four $C_0-components$: F_1, F_2, F_3 and F_4.

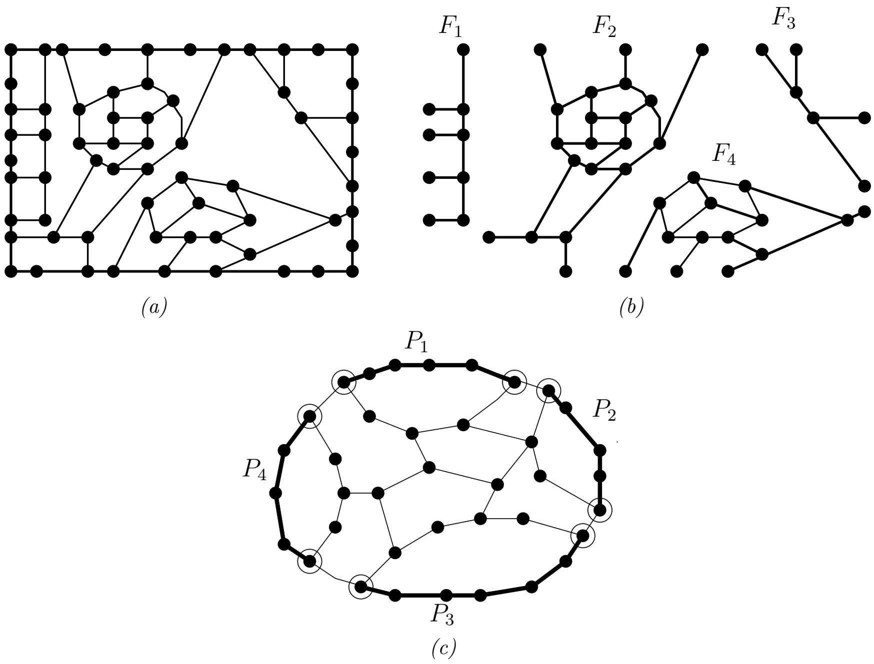


FIGURE 3.19: (a) A graph G (b) C_0-*components* of G (c) Four chains in a graph

Theorem 3.2 (*M.S Rahman et. al. [30]*)**.** *A 2-3 plane graph has a rectangular drawing if and only if it satisfies the following conditions*

(a) G has no 1-legged cycle

(b) Every 2-legged cycle in G contain at least two degree 2 vertex.

(c) Every 3-legged cycle in G contain at least one degree 2 vertex.

(d) If G contains c_2 independent 2-legged cycle and c_3 independent 3-legged cycle then, $2c_2 + c_3 \leq 4$.

From Theorem 3.2 (d) it is visible that if a graph G has a rectangular drawing then G can have at most two independent 2-legged cycles and at most four independent 3-legged cycles. Fig. 3.20 shows maximum number of independent cycles in rectangular drawing.

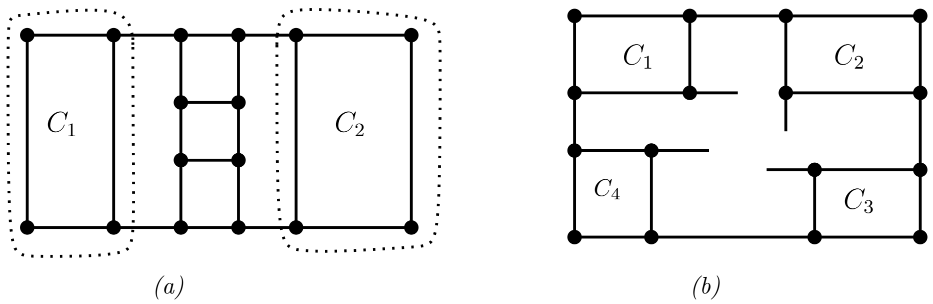


FIGURE 3.20: (a) Two Independent 2-legged cycles (b) Four Independent 3-legged cycles

Algorithm: Consider that a graph G satisfies the conditions in Theorem 4. This algorithm will help us to choose corner vertices appropriately. The graph G and one of its embedding Γ is known a priori. Four corner vertices will be chosen based on the structural property of G and its embedding.

Case 1: Consider G contains at most three degree 3 vertices on $C_0(G)$. First consider that there are only two degree 3 vertex on $C_0(G)$. G will have only one C_0-component. These two degree 3 vertex along with degree two vertices in $C_0(G)$ create two chains. Furthermore, there are two *2-legged* cycles C_1 and C_2 in G. In Fig 3.21 (a) the cycles are shown as dotted lines. Since G satisfies condition (b) is Theorem 3.2, each *2-legged* cycle must contain at least two degree 2 vertices. Then we can chose any two degree 2 vertices from the chains of each cycle as corners. Fig 3.21 (a) and (b) shows the process to chose four corner vertices if G contains only two degree 3 vertices. Furthermore, G will have exactly one C_0-component if there are exactly three degree 3 vertices in $C_0(G)$. The C_0-component has no cycles and G has three *3-legged* cycles C_1, C_2 and C_3 identified by dotted lines in Fig 3.21(c). There are three chains associated with three *3-legged* cycles. Since G satisfies condition (c) in Theorem 3.2, each *3-legged* cycle contains at least one degree 2 vertex. Then we select one degree 2 vertex from chains of each *3-legged* cycles and the remaining one can be chosen arbitrarily.

Case 2: Now consider that G contains four or more vertices of degree 3 on $C_0(G)$. First we need to find all *2-legged* cycles and *3-legged* cycles in G.

Subcase 1: G will have exactly one $C_0(G)$-component if G does not contain pair of independent *2-legged* cycles. If G has no pair of independent *3-legged* cycles the we can choose four vertices of degree 2 as corners in a way that each chain contains at most two vertices of degree 2 and each pair of consecutive chains contains at most three. In Fig 3.22(a) show that four corners are chosen from two non-consecutive chains indicated with dotted regions and the graph in in Fig 3.22(b) contains three chain chains.

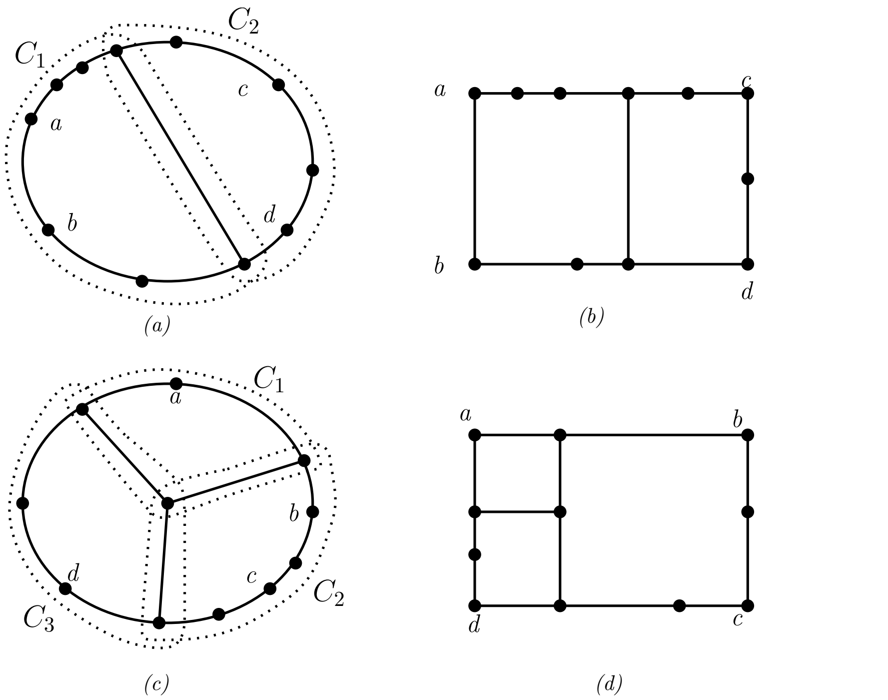


Figure 3.21: (a) A graph G_1 with two chains (b) Rectangular drawing of G_1 (c) A graph G_2 with three chain (d) Rectangular drawing of G_2

If G contains a pair of independent *3-legged* cycles then G has at most four independent *3-legged* cycles. In this case first we need to find all independent *minimal 3-legged* cycles in G. Lets say there are k independent *3-legged* cycles in G. Each *minimal 3-legged* cycle contains at least one vertex of degree 2 on $C_0(G)$. Then we arbitrarily choose a vertex of degree 2 from each *minimal 3-legged* cycles. If the number of independent *3-legged* cycles less than four then we choose $4 - k$

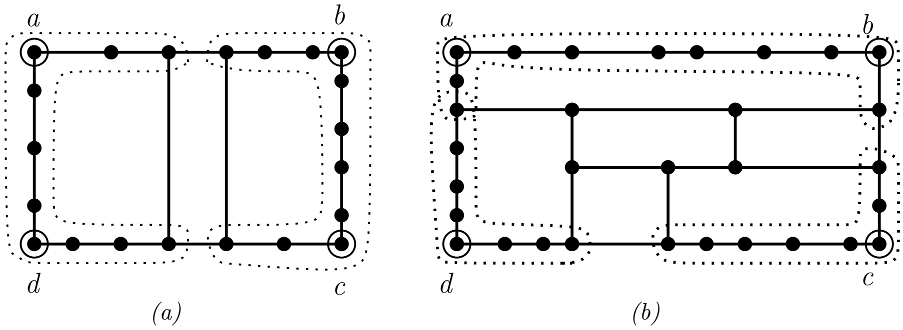


FIGURE 3.22: (a) A graph with two chains (b) A graph with three chains

vertices of degree 2 on $C_0(G)$ which are not chosen so far in a way that each chain contains at most two of the four chosen vertices. In Fig 3.23(b) four degree 2 vertices a, b, c and d are chosen as designated corners where vertices a, b and c are chosen from three independent minimal *3-legged* cycles and vertex d is chosen arbitrarily from rest of degree 2 vertices on $C_0(G)$.

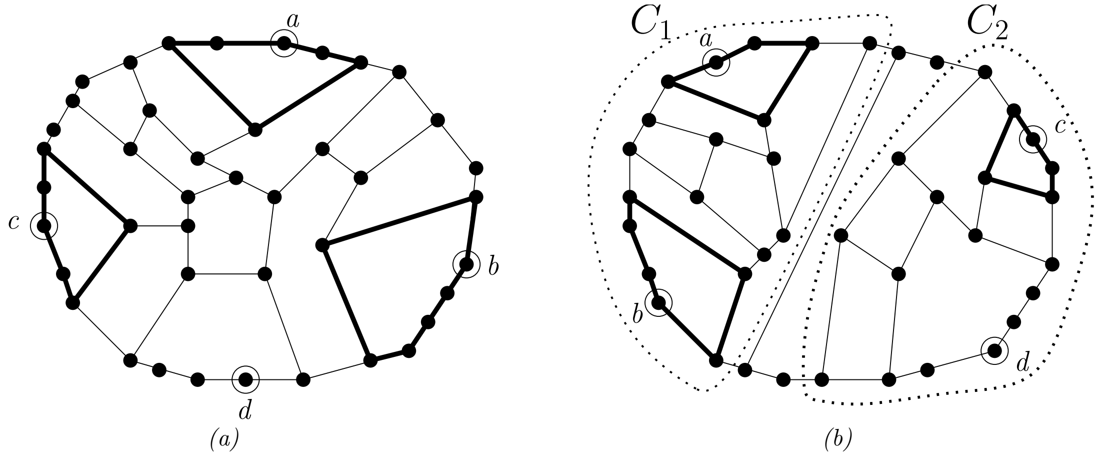


FIGURE 3.23: (a) Three independent *3-legged* cycles in G (b) Two independent *2-legged* cycles in G

Subcase 2: Consider G has two or more $C_0(G)$-components.Then G has a pair of independent 2-legged cycles. Let C_1 and C_2 be two independent *minimal 2-legged* cycles. We can assume that both C_1 and C_2 are minimal *2-legged* cycles. Now find the minimal *3-legged* cycles in $G(C_i)$; $1 \leq i \leq 2$. Let k_j is the number of minimal 3-legged cycles in G. For each minimal *3-legged* cycles we choose arbitrarily exactly one vertex of degree 2 on the 3-legged cycles. If the number of independent *3-legged* cycles less than two then we choose arbitrarily $2 - k_j$

vertices of degree 2 on $V(C_i)$ which are not chosen so far. In Fig 3.23(b) C_1 and C_2 are two independent *2-legged* cycles. Cycle C_1 has two independent minimal 3-legged cycles and C_2 has one independent minimal *3-legged* cycle. Among four corner vertices a, b, c and d; vertices a and b are chosen from two independent *3-legged* cycles in C_1; and vertex c is chosen from one independent *3-legged* cycle, and vertex d is chosen arbitrarily from the remaining vertices of degree 2 in C_2.

3.4.3.3 Rectangular Drawing

In this section we will obtain rectangular drawing of a given graph G. We will use the rectangular grid drawing algorithm of *Md. Rahman et. al.*[28] to achieve the rectangular drawing. So far we have an embedding Γ of G and appropriate four corner vertices a, b, c and d. So we can easily draw the outer face as rectangle by joining the four corner vertices. Now we will discuss on drawing all the inner faces as rectangle.

A plane graph divides the plane into connected regions called *faces*. The *contour* of a face is a *clockwise* cycle formed by the edges on the boundary of the face. In a graph G the contour of the outer face of G is denoted by $C_0(G)$ or C_0.

Theorem 3.3 (*Md. Rahman et. al. [28]*). *G has no rectangular drawing if*

(a) the C_0-component has a cycle with less than four legs.

(b) G has a critical cycle C attached to path P_N, P_E, P_S or P_W, except the outer cycle C_0.

(c) G has a cycle with $n_{cc}(C)=0$ attached to path $P = P_N+P_E, P_E+P_S, P_S+P_W$ or $P_W + P_N$, except the outer cycle C_0

A path P is attached to a cycle C in G if P does not contain any vertices in the proper inside of C and the intersection of C and P is a single subpath of P.

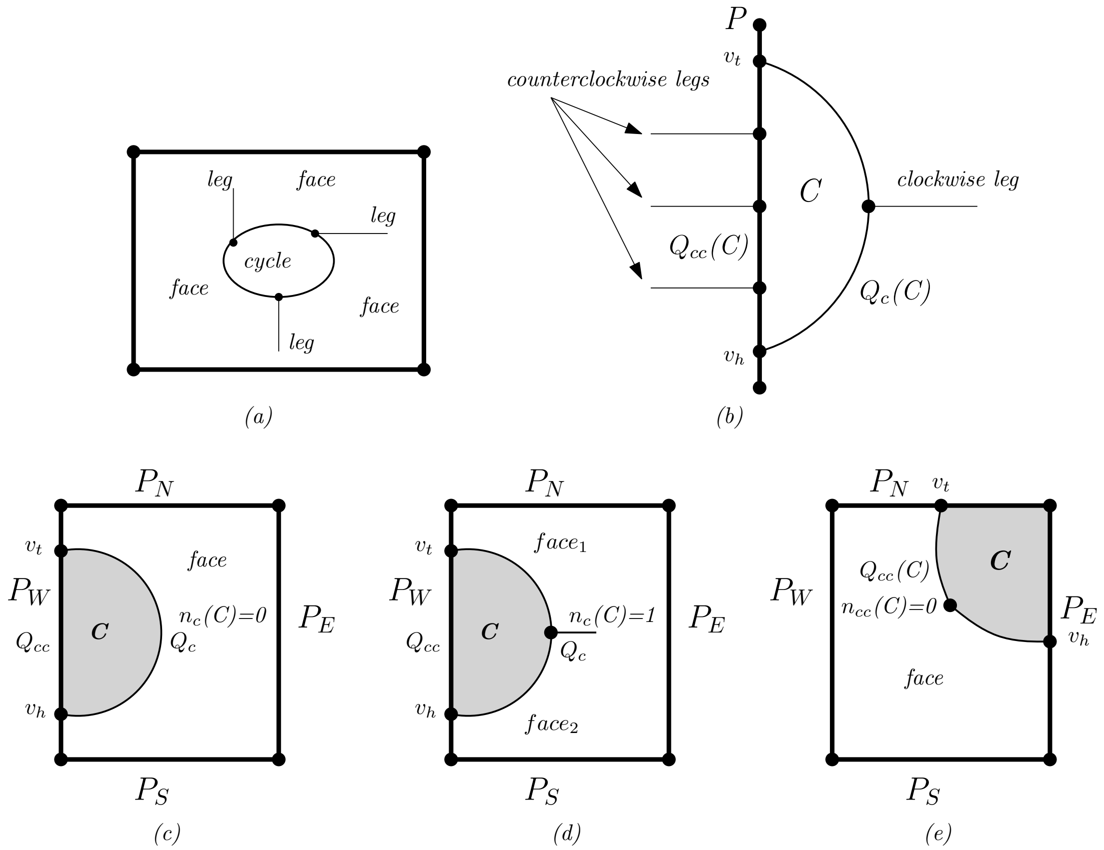


FIGURE 3.24: (a) Faces of a *3-legged* cycle (b) Clockwise critical cycle C attached to path P (c) Critical cycle attached to P_W when $n_c(C)$=0 (d) Critical cycle attached to P_W when $n_c(C)$=1 (e) A cycle with $n_{cc}(C)=0$ attached to P_N+P_E path

Consider v_t is the starting vertex or *tail vertex* and v_h is the ending vertex or *head vertex* of P on cycle C. $Q_c(C)$ represents a path on C turning clockwise around C from v_t to v_h and $Q_{cc}(C)$ represents a path on C turning clockwise around C from v_t to v_h. A leg l of a cycle C is called *clockwise leg* l_c for P if it is incident to a vertex in $Q_c(C)$ except v_t and v_h. The number of clockwise legs of C for a P is denoted by $n_c(C)$. On the same way we define counterclockwise leg l_{cc} of a C for P and the number of counterclockwise legs are indicated by $n_{cc}(C)$. A cycle C attached to path P is called a *clockwise cycle* if $Q_{cc}(C)$ is a subpath of P. Similarly, a cycle C is called *counterclockwise cycle* if $Q_c(C)$ is a subpath of P. A cycle C is called *critical cycle* if either C is a clockwise cycle and $n_c(C) \leq 1$ or C is a counterclockwise cycle and $n_{cc}(C) \leq 1$. Fig 3.24(b) shows a critical cycle attached to a path P.

A graph G has no rectangular drawing if the C_0-*component* has a cycle with less than four legs. It is impossible of obtain rectangular drawing of all inner faces of G that are outside C and whose contours contain edges of the cycle if a cycle in C_0-*component* has less than four legs. Fig 3.24(a) illustrates a cycle with three legs and the faces.

A rectangle has four sides and opposite sides are parallel to each other. Each side may contain more than two vertices. To identify the sides we denote upper side as *north-path* or P_N, bottom side as *south-path* or P_S, right side as *east-path* or P_E and left side as *west-path* or P_W. A graph G has no rectangular drawing if G has at least one critical cycle attached to path P_N, P_E, P_S or P_W, except the outer cycle C_0. In Fig 3.24(c) a critical cycle C with $n_c(C)=0$ attached to P_W and Fig 3.24(d) shows a critical cycle C with $n_c(C)=1$ attached to P_W. G has no rectangular drawing if G has a cycle with $n_{cc}(C)=0$ attached to path $P = P_N + P_E, P_E + P_S, P_S + P_W$ or $P_W + P_N$, except the outer cycle C_0. In Fig 3.24(e) a cycle is attached to $P_N + P_E$ path. It is impossible to draw the inner face, which is outside C and whose contour contains $Q_{cc}(C)$, as a rectangle. A C_0-component is called a *bad component* if it satisfies one of the conditions in Theorem 3.3. The C_0-component mentioned in Theorem 3.3(c) is called a *bad corner*.

Theorem 3.4 (*Md. Rahman et. al. [28]*)**.** *Let G be a plane 2-3 graph and four corner vertices of degree 2 divides the outer cycle C_0 into four paths P_N, P_E, P_S and P_W. Then G has a rectangular drawing if and only if G has no bad component.*

Consider path P_N and P_S has set of vertices $\{v_0, v_1, ..., v_{p-1}, v_p\}$ and $\{u_{q-1}, u_q, ..., u_1, u_0\}$ respectively. A path is called *NS-path* if it starts at any one vertex v_i on P_N and ends at any one vertex u_j on P_S without passing through any other vertex on

C_0. Similarly we can define *EW-path*. If C_0 is the outer face of a rectangle then *NS-path* or *EW-path* are the paths that joins between two opposite side of a rectangle. An *NS-path* P divides G in to two subgraphs G_E^P and G_W^P, where G_E^P is the east part of G including P, and G_W^P is the west part of G including P. Drawing P as straight line, we fix the embedding of $C_0(G_W^P)$ as a rectangle with the north path $P_N^{'}$=$v_0v_1, v_1v_2, ...v_{i_1}v_i$, east path $P_E^{'} = P$, the south path $P_S^{'}$=$u_ju_{j-1}, u_{j+1}u_{j+2}......u_{q-1}u_q$, and the west path $P_W^{'}$=P_W. Similarly we can fix the embedding of $C_0(G_E^P)$. Now we define path P is an *NS-partitioning path* if neither G_W^P nor G_E^P has a bad component. Similarly *SN*-, *WE*- and *EW-partitioning path* can be defined. If G has a partitioning path P then we can obtain a rectangular drawing of G by recursively finding the rectangular drawings of G_E^P and G_W^P or G_N^P and G_S^P, and patching them together with P.

A *boundary face* is an inner face of G and its contour contains at least one edge of C_0. A *boundary path* is a maximal (directed) path on the contour of a boundary face connecting two vertices on C_0 without passing through any edge on C_0. The direction of a boundary path is same as the contour of the face. For $X, Y \in \{N, E, S, W\}$, a *boundary XY-path* is a boundary path starting at a vertex on P_X and ending at a vertex on P_Y.

Lemma 3.5 (*Md. Rahman et. al. [28]*)**.** *Any boundary* $NS-$, $SN-$, $EW-$, $WE-path$ P *of* G *is a partitioning path.*

Now consider that G has no boundary $NS-$, $SN-$, $EW-$ and $WE-paths$. That means there is no single partitioning path. Then the C_0 component has at least one vertex on each of the paths P_N, P_E, P_S and P_W. In this case we can find a pair of partitioning paths P_c and P_{cc}. These pair of partitioning paths will divide G into two or more subgraphs having no bad components. Both P_c and P_{cc} are *NS-paths* which have the same ends and do not cross each other in

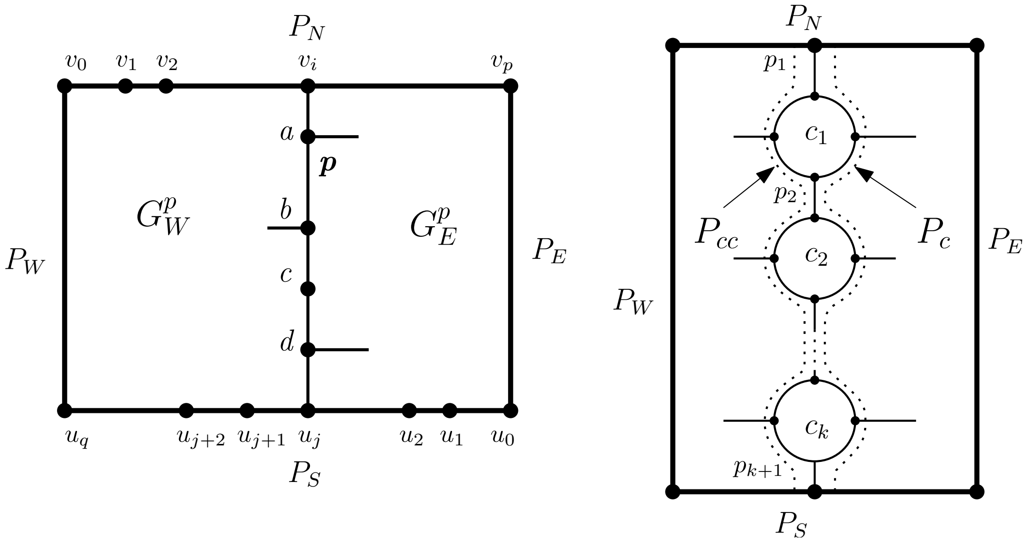


FIGURE 3.25: (a) Faces of a *3-legged* cycle (b) Clockwise critical cycle C attached to path P

the place but they may share several edges. If P_c and P_{cc} are not equal then several cycles will be created by these to paths. The edges $E(P_c) \oplus E(P_{cc}) = E(P_c) \cup E(P_{cc}) - E(P_c) \cap E(P_{cc})$ will create vertex-disjoint cycles C_1, C_2, ... C_k, $k \geq 1$. Fig 3.25(b) shows such cycles between P_c and P_{cc}. If there are k cycles then P_c and P_{cc} share $k+1$ maximal subpaths, P_1, P_2, ... P_{k+1}. As P_c and P_{cc} does not cross each other we can assume that P_c turns around cycles C_1, C_2, ... C_k clockwise and P_{cc} truns around them counterclockwise. P_c and P_{cc} are chosen such a way that each cycle C_i has exactly four legs; assuming clockwise order. The first leg is in P_i, the second one is a clockwise leg, the third one is contained in P_{i+1} and the fourth leg is a counterclockwise leg. Thus G is divided into $G_W^{P_{cc}}$, $G_E^{P_c}$, $G(C_1)$, $G(C_2)$, ... $G(C_k)$. In Fig 3.26(a) P_c and P_{cc} are shown as dotted lines and in Fig 3.26(b) shows five subgraphs, $G_W^{P_{cc}}$, $G_E^{P_c}$, GC_1, GC_2 and GC_3, obtained by spliting G.

Lemma 3.6 (*Md. Rahman et. al. [28]*)**.** *Assume that G has no bad component and that a cycle C in C_0-component has exactly four legs dividing C into four paths P_N', P_E', P_S' and P_W'. Then the subgraph $G(C)$ of G inside C has no bad component for any rectangular embedding of C fixed by P_N', P_E', P_S' and P_W'.*

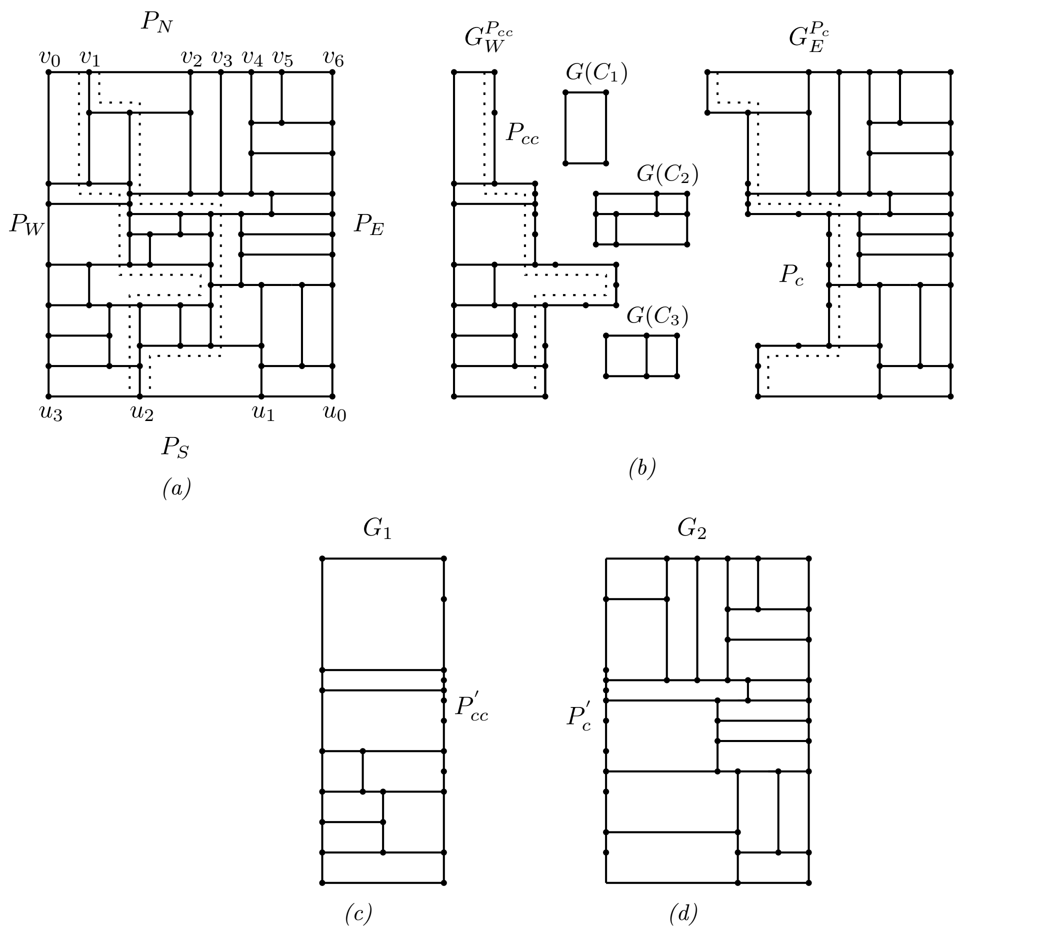


FIGURE 3.26: (a) P_c and P_{cc} in G (b) Subgraphs of G (c) Subgraph formed by $P_{cc}^{'}$ (d) Subgraph formed by $P_c^{'}$

For any fixed rectangular embeddings of C_1, C_2, ... C_k we can assume that $G(C_1)$, $G(C_2)$, ... $G(C_k)$ has a bad component. Two rectangular embeddings are possible for each cycle C_i, $1 \leq i \leq k$ as shown in Fig 3.27(a). For cycles C_1, C_2, ... C_k there are 2^k differenct embeddings are possible where P_c and P_{cc} are embedded as alternating sequences of horizontal and vertical line segment as shown in Fig 3.27(b). From graph $G_W^{P_{cc}}$ we contract all the edges of P_{cc} which are on the horizontal side of rectangular embeddings of C_1, C_2, ... C_k. Let G_1 and G_2 two graph obtained from $G_W^{P_{cc}}$ and $G_E^{P_c}$ by contracting edges from P_{cc} and P_c. In the new graph we denote the updated P_{cc} path as $P_{cc}^{'}$ and P_c as $P_c^{'}$ as shown in Fig 3.26(c) and Fig 3.26(d). If G_2 has a rectangular drawing with fixed $P_c^{'}$ as a straight line then the rectangular drawing of G_2 can be modified to achieve rectangular drawing of $G_E^{P_c}$ where P_c is drawn as an alternating sequence of horizontal and vertical lines. Similarly we can obtain rectangular drawings of $G_W^{P_{cc}}$, $G(C_1)$, $G(C_2)$, ... $G(C_k)$ and patch them to get a rectangular drawing of G. The paths P_c and P_{cc} are defined as *pair of partitioning paths* or a *partition-path.* If $P_c = P_{cc}$ then it is a single partitioning path.

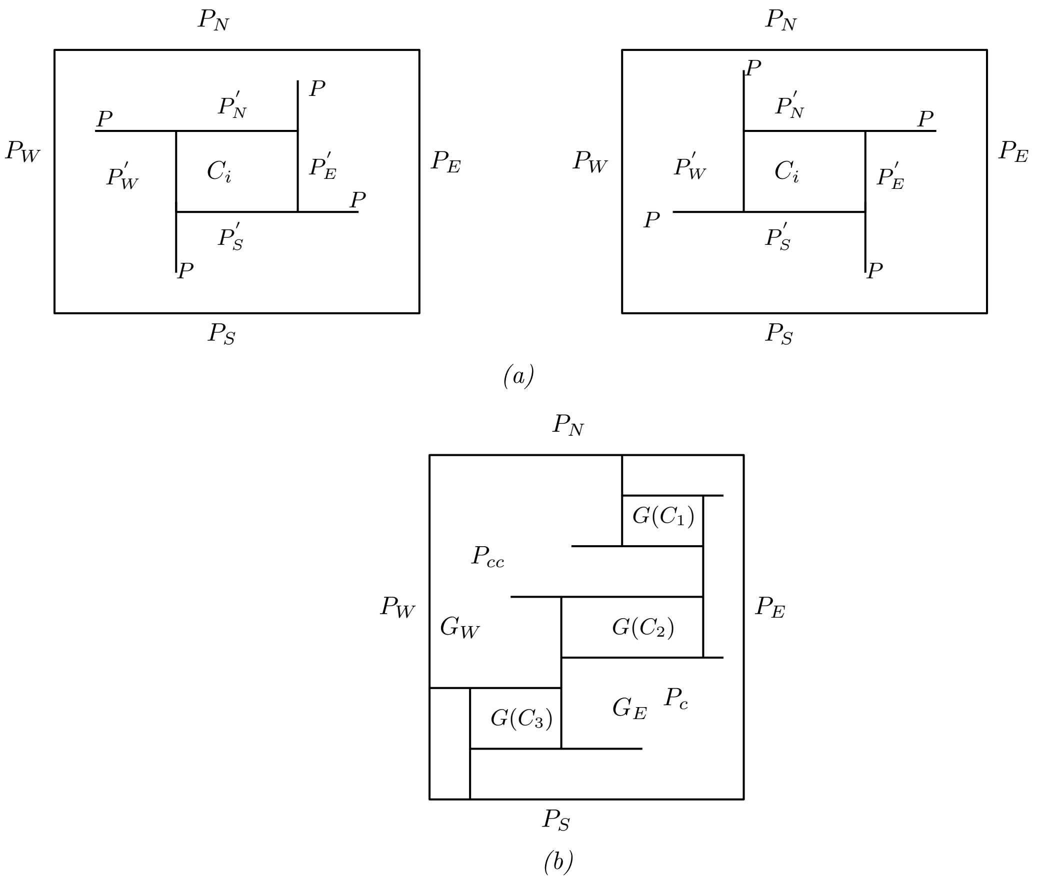


FIGURE 3.27: (a) Two alternative embeddings of C_i (b) Embedding of a partition-pair

Algorithm: So far we have a graph G, an embedding Γ of G and four designated corner vertices a, b, c, and d from the previous steps. The corner vertices will be on the contour $C_0(G)$ of G. Now we draw the contour $C_0(G)$ of the outer face of G as rectangle by two horizontal line P_N and P_S and two vertical line P_E and P_W by joining a, b, c and d. Then we find all the C_0-components H_1, H_2, ... H_p. Our intension is to achieve rectangular drawing of each C_0-components. We will call a subroutine *draw_rectangle* for drawing all C_0-components H_i. We initiate the subroutine with a C_0-components H_i and a graph G_i where $G_i = C_0 \cup H_i$.

Subroutine ***draw_rectangle(H,G)***: Here H is the C_0-component of G. First we will find partitioning paths P in G.

Case 1: G has a partitioning path P. Consider P is one of the boundary $NS-$, $SN-$, $EW-$ or $WE-paths$. Let, P =*NS-path*. Then we will draw all the edges of P on a vertical line. Now if the number of edges in $P \geq 2$ then we will find all the C_0-components of G_E^P for the fixed rectangular embedding of cycle $C_0(G_E^P)$. Let, F_1, F_2, ... F_q are the C_0-components of $C_0(G_E^P)$. Then we will obtain rectangular drawing of each C_0-component F_i by calling *draw_rectangle(*F_i,$C_0(G_E^P) \cup F_i$*)* subroutine. Similarly we can obtain the rectangular drawing of G_W^P. Finally patch all the drawings together to get rectangular drawing of G.

Case 2: G has no partitioning paths or no boundary $NS-$, $SN-$, $EW-$ or $WE-paths$. In this case we will find the westmost $NS-path$ P. From path P we will get a partition-pair P_c and P_{cc}.

Subcase 1: If $P_c = P_{cc}$, then we will draw all the edges of P_c on the vertical line. Now consider P_c divides G into two subgraphs, $G_1 = G_W^{P_c}$ and G_2=$G_E^{P_c}$. Then our aim is to obtain rectangular drawing of each graph G_i, $i = \{1, 2\}$. Now for each G_i we will find the C_0-components, F_1, F_2, ... F_q. Then we achieve rectangular drawing of each C_0-components F_j by calling *draw_rectangle(*F_i,$C_0(G_i) \cup F_j$*)* subroutine. Then finally patch all the drawings to obtain rectangular drawing of G.

Subcase 2: If $P_c \neq P_{cc}$, then we will draw all the edges of P_c and P_{cc} on alternating sequence of horizontal and vertical lines as shown in Fig 3.27. Let G_1 and G_2 graphs are obtained by contracting the edges of P_c and P_{cc} which are on the horizontal side of C_1, C_2, ... C_k respectively, and $G_3 = G(C_1), G(C_2), ...G(C_k)$. Now for each subgraphs G_i we find the C_0-components, F_1, F_2, ... F_q of G_i and obtain their rectangular drawing by calling *draw_rectangle(*F_j,$C_0(G_i) \cup F_j$*)*

subroutine. Then finally patch all the drawings together to achieve the rectangular drawing of G.

3.4.4 Layer Graph representation of Rectangular Drawing

In Layer graph representation all the edges are either vertical or horizontal, each cross point will represents a vertex and no vertices will fall on top of each other when angle between vertical and horizontal edges are 0^o or 180^o. We have already meet other criteria in previous sections excepts the last one. In this section we will ensure that no vertex the vertices of rectangular drawing will fall on top of each other when the angle between vertical and horizontal edges are 0^o or 180^o. Rectangular drawing of graph G in Fig. 3.28 (a) does not meet all the characteristics of layer graph drawing. Fig. 3.28 (b) and (c) shows the position of vertices of G when bended to rigid(angle between horizontal and vertical edges are 0^o) and left(angle between horizontal and vertical edges are 180^o) respectively. In both layouts there are many vertices fall on top of each other.

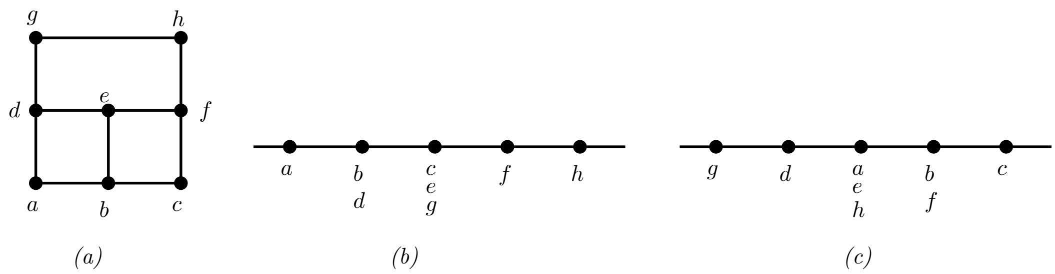


Figure 3.28: (a) Rectangular drawing of a graph G (b)-(c) Two different layout of G, vertices fall on top of each other

In the last section we obtained rectangular drawing of G where each face is drawn as rectangle and each edges are drawn as vertical or horizontal line segment. According to rectangular drawing so far no vertices are at same position in the drawing.

The original idea to transform rectangular drawing to layer graph drawing is to create enough horizontal space between each vertical line segment. By creating enough horizontal space between each vertical line segment ensures that the vertices on each vertical line segment will not fall on top of any other vertices when the angle between horizontal and vertical edges are 0^o or 180^o.

First we will draw the P_S path horizontally and P_W path vertically starting from the leftmost vertex of P_S and we will draw P_N later. Here we consider all horizontal edges are elastic and all the vertical edge lengths are fixed which will help us to represent the rectangular drawing as layer graph. After that we will find several vertical lines connected to P_S. Now we will consider two cases based on the edges of *NS-Path*.

Case 1: Consider all the vertical line segment connected to P_S are *NS-paths* and all the edges in *NS-path* are vertically drawn in rectangular drawing.

Consider there are n vertical lines or NS-Paths VL_0, VL_1, ... VL_n, connected to P_S. Now we place the first vertical line VL_0 at the left most position in P_S. Then we have to fix the position of next vertical line VL_1 so that when we bend the VL_0 to right or bend VL_1 to left then the vertices on VL_0 and VL_1 should not fall on top of each other. To ensure this we should have enough horizontal space for both vertical lines. So the position of VL_1, $X_{VL_1} = |VL_0| + |VL_1| + X + 1$, where X is the number of degree two vertices between VL_0 and VL_1. Similarly we can calculate X_{VL_2}=$|VL_1| + |VL_2| + X + 1$. After fixing the placement of each vertical lines we need to add the horizontal lines between the vertices on vertical lines. Fig 3.29 illustrates the process. From the Fig 3.29 you can see that if we place the horizontal lines such a way then there is no possibility that the vertices will fall on top of each other.

Case 2: Consider all the vertical line segment connected to P_S are not *NS-paths* where all the edges in *NS-path* are vertically drawn in rectangular drawing. We are considering about the horizontal edges in P_c or P_{cc}.

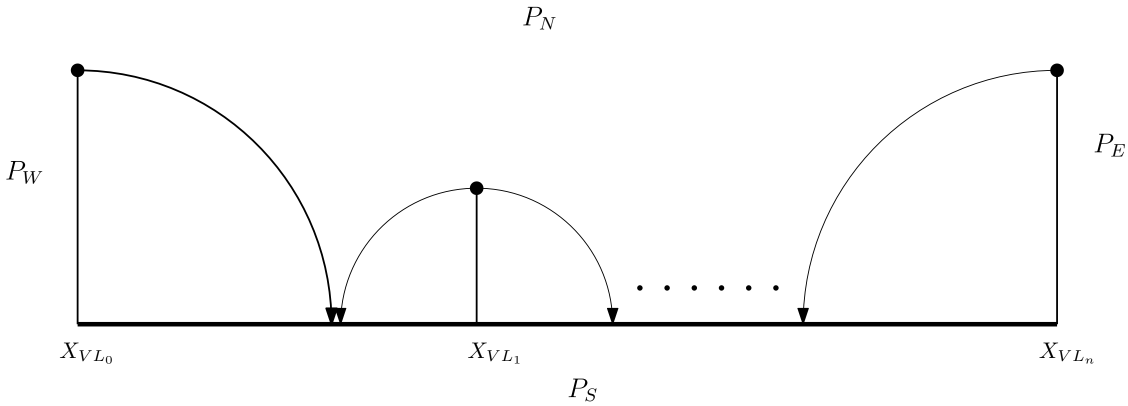


FIGURE 3.29: Layer graph representation from rectangular drawing of G

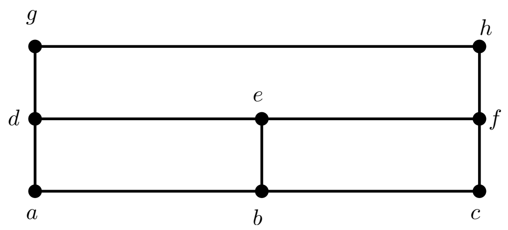


FIGURE 3.30: Layer graph representation of rectangular drawing of G in Fig. 3.28

First we will draw the left most path or *NS-Path* or P_W and place it at X_{VL_0} on a horizontal line. Here the horizontal line will be an edge or a subpath of P_S. Now moving to right on P_S we will find the next vertical line VL_1 connected to P_S. VL_1 can be one of the following:

(a) VL_1 is a *NS-Path* as described in case 1.

(b) VL_1 can be an vertical edge or subpath with consecutive vertical edges of P_c or P_{cc}.

Let VL_1 is an vertical line of type (b). Now we consider a virtual vertical path P_v containing all horizontal edges pass through VL_1. Then we will calculate the appropriate length of each horizontal edges between VL_0 and P_v. Keeping the vertical line segments fixed and adjust the horizontal line segments.

Consider the Fig. 3.31. We denote distance between any two vertex a and b by d_{ab}. In the Fig. 3.31 d_{af} can be expressed as summation of two horizontal edge distances, d_{gh} and d_{hi}. Again d_{hi} can be expressed as summation of another two horizontal edge distances, d_{lm} and d_{mn}. After that there is no option to express any of the previously mentioned horizontal edges to summation of some new horizontal edges. So to determine correct value of d_{af} we need to calculate d_{gh}, d_{lm} and d_{mn}. For each level of horizontal lines we add the vertical lengths to calculated distances.

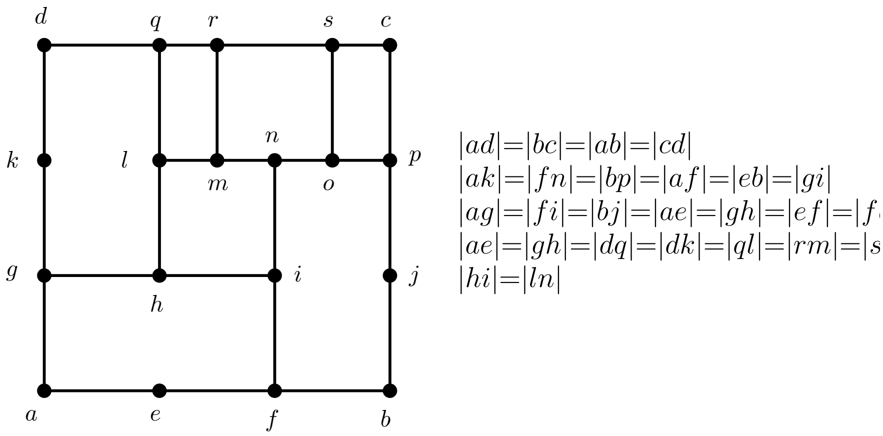


FIGURE 3.31: Rectangular drawing of a graph

So the horizontal distances between vertical lines in Fig. 3.31 are:

$$|gh| \geq [|gd| + |hq| + 1]$$

$$|hi| \geq [|lq| + |mr| + 1] + [|mr| + 1] + |lh|$$

$$|lm| \geq [|lq| + |mr| + 1]$$

$$|mn| \geq [|mr| + 1]$$

$$|no| \geq [|os| + 1]$$

$$|op| \geq [|os| + |pc| + 1]$$

$$|fb| \geq [|os| + 1] + [|os| + |pc| + 1] + |bp| + 1$$

$$\begin{aligned}|af| &\geq |gh| + |hi| + |fi| + 1\\ &\geq |gh| + |lm| + |mn| + |hl| + |fi| + 1\\ &\geq [|gd| + |hq| + 1] + [|lq| + |mr| + 1] + [|mr| + 1] + |fi| + 1\end{aligned}$$

Fig. 3.32 represents the layer graph drawing of rectangular drawing in Fig. 3.31. The layer graph in Fig. 3.32 created by srinking all the edges 33.33% of Fig. 3.31.

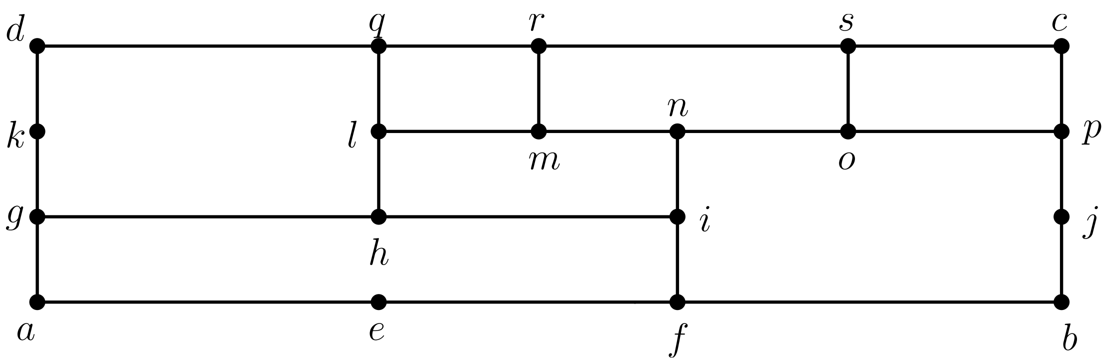


FIGURE 3.32: Layer graph drawing of Fig. 3.31

Lemma 3.7. *Consider a rectangular drawing of a graph G where the edges are elastic. Then the rectangular drawing can be transformed to a layer graph drawing by changing the horizontal edges lengths.*

By elastic we considered that the edge lengths of the graph in rectangular drawing can be increase or decrease the edge lengths. As the edges of G are elastic, we can create enough space between each pair of consecutive vertical lines by increasing or decreasing lengths of horizontal line segments. Creating enough space between each of consecutive vertical lines will ensure that no vertices will fall on top of each other when angle between vertical and horizontal lines are 0^o or 180^o.

Theorem 3.8. *A graph G is not line rigid if one of the embedding of G has a rectangular drawing.*

According to the transformation process to layer graph drawing it is clear that every rectangular drawing can be transformed to layer graph representation. If there exists an rectangular drawing of a graph then a layer graph drawing also exists for that graph. *Chin et. al.* [1] proved that a graph is line rigid if and only if it has no layer graph drawing.

Chapter 4

Conclusion

4.1 Experimental Study on existing Point Placement Algorithm

All algorithms have been implemented in C on a Apple Mac laptop with the following configuration: Intel(R) Xeon(R) CPU, X7460 @ 2.66GHz OS: Ubuntu 12.04.5, Architecture: i686.

Further work can be done on several fronts. Particularly worthwhile is to conduct further experiments into the behavior of the randomized algorithm, specifically the influence of floating point arithmetic on keeping signed sums unequal. On the theoretical side, it might be interesting to come up with a completely different randomized algorithm - one that does not depend on maintaining an exponential number of signed sums.

4.2 Recognizing Line Rigid Graphs

Although the proposed scheme considers planar 2-connected graph with maximum degree 3 but the phases in the scheme obtain layer graph drawing of a *2-3 graph*. So our proposed schema can recognize whether planar 2-connected *2-3 graph* is line rigid or not.

The proposed scheme can't recognize line rigid graph that has maximum degree more than 3, and bends in the inner subgraphs. But a in layer graph there can be bends and maximum degree can be more than 3. It will be interesting to obtain layer graph drawing of a graph that has multiple bend and maximum degree is more than 3. Furthermore, line rigid graphs can be non-planar. Another interesting problem will be to find line rigidity of non-planar graphs.

Bibliography

[1] Francis Y. L. Chin, Henry C. M. Leung, Wing-Kin Sung, and Siu-Ming Yiu. The point placement problem on a line - improved bounds for pairwise distance queries. In *Proceedings of the Workshop on Algorithms in Bioinformatics*, volume 4645 of *LNCS*, pages 372–382, 2007.

[2] Md. Safiul Alam, Asish Mukhopadhyay, and Animesh Sarker. Generalized jewels and the point placement problem. In *CCCG '09: Proceedings of the 21st Canadian Conference on Computational Geometry*, pages 45–48, 2009.

[3] Md. Shafiul Alam and Asish Mukhopadhyay. More on generalized jewels and the point placement problem. *J. Graph Algorithms Appl.*, 18(1):133–173, 2014.

[4] Md. Shafiul Alam and Asish Mukhopadhyay. Three paths to point placement. In *Proceedings of the Conference on Algorithms and Discrete Applied Mathematics*, volume 8959 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 2015.

[5] Gale Young and Alston S Householder. Discussion of a set of points in terms of their mutual distances. *Psychometrika*, 3(1):19–22, 1938.

[6] L. M. Blumenthal. *Theory and applications of distance geometry.* New York : Chelsea, 2nd edition, 1970.

[7] G.M. Crippen and T.F. Havel. *Distance geometry and molecular conformation*, volume 15. Research Studies Press, Taunton, Somerset, England, 1988.

[8] Steven S. Skiena, Warren D. Smith, and Paul Lemke. Reconstructing sets from interpoint distances (extended abstract). In *SCG '90: Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 332–339, New York, NY, USA, 1990. ACM. ISBN 0-89791-362-0. doi: http://doi.acm.org/10.1145/98524.98598.

[9] Alain Daurat, Yan Gérard, and Maurice Nivat. The chords' problem. *Theor. Comput. Sci.*, 282(2):319–336, 2002. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(01)00073-1.

[10] H.O. Smith and K.W.Wilcox. A restriction enzyme from hemophilus influenzae. i. purification and general properties. *Journal of Molecular Biology*, 51: 379–391, 1970.

[11] Joshua Redstone and Walter L Ruzzo. Algorithms for a simple point placement problem. In *Algorithms and Complexity*, pages 32–43. Springer, 2000.

[12] Peter Damaschke. Point placement on the line by distance data. *Discrete Applied Mathematics*, 127(1):53–62, 2003.

[13] Md. Shafiul Alam and Asish Mukhopadhyay. Improved upper and lower bounds for the point placement problem. *CoRR*, abs/1210.3833, 2012.

[14] Peter Damaschke. Randomized vs. deterministic distance query strategies for point location on the line. *Discrete Applied Mathematics*, 154(3):478–484, 2006.

[15] Prabhakar Raghavan and Rajeev Motwani. Randomized algorithms, 1995.

[16] Ioannis Z. Emiris and Ioannis D. Psarros. Counting euclidean embeddings of rigid graphs. *CoRR*, abs/1402.1484, 2014.

[17] Dieter Kratsch, Ross M McConnell, Kurt Mehlhorn, and Jeremy P Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *SIAM Journal on Computing*, 36(2):326–353, 2006.

[18] Sergey Bereg. Certifying and constructing minimally rigid graphs in the plane. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 73–80. ACM, 2005.

[19] Ruth Haas, David Orden, Günter Rote, Francisco Santos, Brigitte Servatius, Hermann Servatius, Diane Souvaine, Ileana Streinu, and Walter Whiteley. Planar minimally rigid graphs and pseudo-triangulations. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 154–163. ACM, 2003.

[20] Liliana Alcón, Luerbio Faria, Celina MH de Figueiredo, and Marisa Gutierrez. The complexity of clique graph recognition. *Theoretical Computer Science*, 410(21):2072–2083, 2009.

[21] Derek G. Corneil, Yehoshua Perl, and Lorna K Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985.

[22] Jeremy Spinrad. Recognition of circle graphs. *Journal of Algorithms*, 16(2): 264–282, 1994.

[23] Bruce Hendrickson. Conditions for unique graph realizations. *SIAM Journal on Computing*, 21(1):65–84, 1992.

[24] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568, 1974.

[25] Kurt Mehlhorn and Petra Mutzel. On the embedding phase of the hopcroft and tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[26] Claude Berge. The theory of graphs and its applications, john willey and sons, 1964.

[27] L. Auslander and S. V. Parter. On embedding graphs in the sphere. *Journal of Mathematics and Mechanics*, 10(3):517–523, 1961.

[28] Md Saidur Rahman, Shin-Ichi Nakano, and Takao Nishizeki. Rectangular grid drawings of plane graphs. *Computational Geometry*, 10(3):203–220, 1998.

[29] Md Saidur Rahman, Takao Nishizeki, and Shubhashis Ghosh. Rectangular drawings of planar graphs. *Journal of Algorithms*, 50(1):62–78, 2004.

[30] Md Saidur Rahman, Shin-ichi Nakano, and Takao Nishizeki. Rectangular drawings of plane graphs without designated corners. *Computational Geometry*, 21(3):121–138, 2002.

[31] Takao Nishizeki and Norishige Chiba. *Planar graphs: Theory and algorithms.* Elsevier, 1988.

[32] Asish Mukhopadhyay, S.V. Rao, S. Pardeshi, and S. Gundlapalli. Linear layouts of weakly triangulated graphs. In *Proceedings of the Workshop onAlgorithms and Computation*, volume 8344 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2014.

[33] Md. Shafiul Alam and Asish Mukhopadhyay. A new algorithm and improved lower bound for point placement on a line in two rounds. In *CCCG '10: Proceedings of the 22nd Canadian Conference on Computational Geometry*, pages 229–232, 2010.

[34] Bill Jackson and Tibor Jordán. Connected rigidity matroids and unique realizations of graphs. *Journal of Combinatorial Theory, Series B*, 94(1):1–29, 2005.

[35] Tolga Eren, David Kiyoshi Goldenberg, Walter Whiteley, Yang Richard Yang, A Stephen Morse, Brian DO Anderson, and PN Belhumeur. Rigidity, computation, and randomization in network localization. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2673–2684. IEEE, 2004.

[36] Brendan Mumey. Probe location in the presence of errors: a problem from DNA mapping. *Discrete Applied Mathematics*, 104(1-3):187–201, 2000.

[37] Gian Carlo Bongiovanni, Giorgio Gambosi, and Rossella Petreschi, editors. *Algorithms and Complexity, 4th Italian Conference, CIAC 2000, Rome, Italy, March 2000, Proceedings*, volume 1767 of *Lecture Notes in Computer Science*, 2000. Springer. ISBN 3-540-67159-5.

[38] Md Saidur Rahman, Shin-ichi Nakano, and Takao Nishizeki. Box-rectangular drawings of plane graphs. *Journal of Algorithms*, 37(2):363–398, 2000.

[39] R. Connelly. Rigidity. In Gruber P. M. and Wills J. M., editors, *Handbook of Convex Geometry*, pages 223–271. Unknown, 1993.

[40] T. Havel. *Distance Geometry: Theory, Applications and Chemical Applications*. Unknown, 1988.

[41] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *Journal of computer and system sciences*, 30(1):54–76, 1985.

[42] James B Saxe. *Embeddability of weighted graphs in k-space is strongly NP-hard*. Carnegie-Mellon University, Department of Computer Science, 1980.

[43] L. Asimow and B. Roth. The rigidity of graphs. *Trans. Amer. Math. Soc.*, 245:279–289, 1978.

VITA AUCTORIS

NAME:	Pijus Kumar Sarker
PLACE OF BIRTH:	Gaibandha, Bangladesh
YEAR OF BIRTH:	13 November 1986
EDUCATION:	Gaibandha Govt. Boys. High School, Gaibandha, Bangladesh 1996 - 2002
	Gaibandha Govt. College, Gaibandha, Bangladesh 2002 - 2004
	Rajshahi University of Engineering and Technology, Rajshahi, Bangladesh 2005 - 2009 B.Sc
	University of Windsor, Windsor, Ontario, Canada 2013 - 2015 M.Sc.