WEAKLY TRIANGULATED GRAPHS

A Project Report Submitted

for the Course

CS498 Project I

by

Charanjit Singh Ghai

(Roll No. 11010178)

Rachuri Anirudh

(Roll No. 11010155)

Sparsh Kumar Sinha

(Roll No. 11010166)

to the

DEPARTMENT OF CSE

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

November 2014

CERTIFICATE

This is to certify that the work contained in this project report entitled "**Weakly Triangulated Graphs**" submitted by **Charanjit Singh Ghai** (**Roll No.: 11010178**), **Rachuri Anirudh** (**Roll No.: 11010155**) and **Sparsh Kumar Sinha** (**Roll No.: 11010166**) to Department of Computer Science and Engineering, Indian Institute of Technology Guwahati towards the requirement of the course **CS498 Project II** has been carried out by them under my supervision.

Guwahati - 781 039 (Dr. S. V. Rao)

November 2013 Project Supervisor

ABSTRACT

A graph G = (V, E) is said to be triangulated (also called chordal) if it has no chordless cycle of length four or more. Such a graph is said to be rigid if, for a *valid* assignment of edge lengths, it has a unique linear layout and non-rigid otherwise. An assignment of lengths to the edges of G is said to be valid if the distances between the adjacent vertices in a linear placement is consistent with the lengths assigned to the edges of G. A graph G = (V, E) is weakly triangulated if neither G nor its complement $\overline{G}$ contains a chordless cycle of five or more vertices. In this report, we shall be looking into the problem of generating random weakly triangulated graphs and finding linear layouts of chordal bipartite graphs which is a sub-class of weakly triangulated graphs.

Contents

List of Figures **vi**

1 Introduction **1**

1.1 Problem Definitions 2

1.1.1 Generate Random Weakly Triangulated Graphs 2

1.1.2 Chordal Bipartite Graphs 2

2 Generating Random Weakly Triangulated Graphs **3**

2.1 Related Work 3

2.2 Preliminaries 4

2.3 Restricted Random Weakly Triangulated Graph 5

2.3.1 Random Rigidity Tree 6

2.3.2 Proof of Correctness 7

2.3.3 Assigning Rigidity Labels 9

2.3.4 Generating Line Rigid Graph 10

2.3.5 Proof of Correctness 10

3 Linear layouts of Chordal Bipartite Graphs **14**

3.1 Preliminaries 14

3.2 Constructing Rigidity Tree 16

3.3 Computing Peripheral Edge Order 18

3.3.1 Proposed Algorithm 19
3.3.2 Counting three length paths efficiently 25

4 Future Work and Conclusion **30**
4.1 Conclusion 30
4.2 Future Work 31

Bibliography **32**

List of Figures

3.1 Perfect edge without vertex elimination ordering from 3.1a to 3.1e. 16

3.2 Different possibilities for e_i, newly added peripheral edge between x and y: (a)Single three length path (b)More than one three length path with no common edge (c)Two three length paths with a common edge (d)More than two three length paths with a common edge. 18

3.3 Different cases for x_iy_i, the newly added bisimplical edge. . . . 18

3.4 Perfect edge elimination ordering from 3.4a to 3.4d. 20

3.5 (a,c,e,g) xy is a bisimplicial edge. (b,d,f,h) After removing edge xy and edges incident on x and y from graphs in (a,c,e,g) respectively 23

3.6 Different cases of $ING_{x,y}$ with xy being a bisimplicial edge (a)$ING_{x,y} = K_{1,0}$ (b)$ING_{x,y} = K_{1,1}$ (c)$ING_{x,y} = K_{2,2}$ (d)$ING_{x,y} = K_{1,2}$ (e)$ING_{x,y} = K_{1,3}$ 27

3.7 Disjoint set data structure when components C_1, C_2, C_3, C_4 merged to form new component C_{new} 28

Chapter 1

Introduction

Point Placement Problem is a problem where we are given distances between all possible pairs of n nodes, and we have to find a possible placement of those nodes in space such that those distances are not violated. This problem holds significant importance in the field of molecular biology, where given the inter-atomic distances for all possible pairs of atoms in a molecule, we have to come up with an arrangement of these atoms in the three dimensional space so that the distance constraints are not violated. This problem can be visualised as a graph where there is a vertex corresponding to each node(atom) and an edge for a distance constraint with weight equal to the distance value. Here by absence of edge we signify absence of the distance constraint between the corresponding two atoms or vertices. It has been shown that the point placement problem in general (even in two dimensions) is NP-complete. Thus, a simplified and similar problem which is point placement for weakly triangulated graphs where the aim is to place these nodes linearly (i.e. in one dimension) is studied in detail. Also, finding valid edge assignments for getting linear layouts etc. are similar problems which are studied in the domain of weakly triangulated graphs. We discuss two problems in the following sections falling under the same domain.

1.1 Problem Definitions

1.1.1 Generate Random Weakly Triangulated Graphs

Owing to proposal of algorithms for computing linear layouts of weakly triangulated graphs, and other algorithms solving various problems related to weakly triangulated graphs, there is a need for generating these graphs randomly. Thus, we shall be looking into how to generate these graphs randomly. Precisely, the aim here is to come up with an algorithm which can generate random weakly triangulated graphs. Further, the probability of generation of each such graph should be non zero.

1.1.2 Chordal Bipartite Graphs

A chordal bipartite graph is one which is a bipartite graph as well as weakly triangulated. [3] showed how to compute all linear layouts of a weakly triangulated graph, for a valid assignment of lengths to the edges of the graph. Our aim is to extend this result to come up with an algorithm to solve this problem efficiently for chordal bipartite graphs, a sub-class of the weakly triangulated graphs using their properties.

Chapter 2

Generating Random Weakly Triangulated Graphs

2.1 Related Work

Although our work was related to graph generation, we did a survey on generating random trees. This choice would not need any justification after reading the Preliminaries section. The survey by **Sean Luke and Liviu Panait** [4] gave many algorithms for generating random trees constrained by depth of the tree. The main idea revolves around generating a node as a terminal(leaf) or a non terminal(internal node) of some degree and recursively calling the procedure of max depth-1 for each child of the non terminal. This tree in some sense can be related with the parse tree of mathematical expressions. Also, **Hitoshi IBA** [1] gave an algorithm to generate random trees by generating random strings of x and y. Generation of random string is followed by parsing the string from left to right.Each encountered x is pushed on to a stack (as a leaf node), while for each y with k consecutive leading x nodes, k nodes are popped from the stack and made the children of current (new) node which itself is finally pushed onto stack. The string randomly generated is constrained to have n x and $(n-1)$ y.

2.2 Preliminaries

In this section, we define terms and state results required for the work.

We define a weakly triangulated graph as follows:

Definition 2.2.1. A graph is said to be weakly triangulated if it does not have a chordless cycle of length 5 or more.

An edge of a graph is called a peripheral edge if it is not the middle edge of a P_4 (chordless path of four vertices). The following constructive characterization, analogous to the perfect elimination ordering of a triangulated graph, is central to our approach.

Theorem 2.2.2. *A graph is weakly triangulated if and only if it can be generated in the following manner:*

1. *Start with an empty graph G_0.*
2. *Repeatedly add an edge e_j to the graph G_{j-1} to create the graph G_j such that e_j is a peripheral edge of G_j.*

A total order of the m edges $(e_1, e_2, ..., e_m)$ of a graph G is a peripheral edge order if for $1 \leq j \leq m$, e_j is peripheral in the graph $G_j = (V, E_j)$, where $E_j = \{e_1, e_2, ..., e_j\}$. Thus the following theorem is equivalent to Theorem 1:

Theorem 2.2.3. *A graph is weakly triangulated if and only if it admits a peripheral edge order.*

A graph G is *minimally line rigid* if it has no proper induced *subgraph* that is line rigid. For example, the weakly triangulated graphs $K_{2,3}$ and K_3 are minimally line rigid. A subgraph of a graph G is *maximally line rigid* if it has no proper induced *supergraph* that is line rigid.

A component of G is a maximally connected subgraph. A biconnected component is a 2-connected component. When G is connected, it decomposes into a tree of biconnected components called the block tree of the graph. The blocks are joined to each other at shared vertices called cut vertices or *articulation points.* An edge $\{u, v\}$ of G is a *hinge edge* if removal of the vertices u and v and the edges incident on these disconnects G into three or more disjoint components.

The rigidity structure of G is characterized in the following results.

Lemma 2.2.4. *The only minimally line rigid subgraphs of a weakly triangulated graphs are the $K_{2,3}$ and K_3.*

Theorem 2.2.5. *The rigidity graph of a weakly triangulated graph without articulation points and hinge edges is a tree.*

2.3 Restricted Random Weakly Triangulated Graph

In this section, we describe an algorithm for generation of weakly triangulated graph without articulation point and hinge edge. The rigidity structure of weakly triangulated graph without articulation point and hinge edges can be represented as a tree with the following properties:

- Each node can be a quadrilateral or a maximally rigid sub-graph of G.
- Each node which corresponds to a quadrilateral can have degree ≤ 4.

• No two nodes both of which correspond to a maximally rigid sub-graph of G can be adjacent to each other in the rigidity tree.

Remark 2.3.1. If we leave aside the problem of labelling each node of the rigidity tree as to a quadrilateral or a maximally rigid sub-graph, the only constraint in the rigidity tree from a structural point of view is that no two nodes with degree ≥ 5 should be adjacent to each other. This constraint is enforced because as soon as the degree of two nodes becomes ≥ 5, both of them have to be labelled as a maximally rigid sub-graph and no two maximally rigid sub-graphs can be adjacent to each other. We refer this as the **Structural Constraint** of the Rigidity Tree.

Taking these constraints into account and the ideas that we learnt for generating random trees, we shall give an algorithm to generate a random n node rigidity tree.

2.3.1 Random Rigidity Tree

We denote by n, k, m respectively the number of nodes in the rigidity tree, number of nodes in the final random weakly triangulated graph and number of edges in the final random weakly triangulated graph. Note that the aim here is not to come up with an algorithm which generates all possible valid rigidity trees of n nodes with equal probability rather, the aim is to come up with an algorithm which generates all possible rigidity trees of n nodes with some non zero probabilities and does not generate anything else. We also thus expect an integer n as an argument to our algorithm.

Our algorithm generates a valid Rigidity tree of n nodes using Recursion. In base case, it returns a tree with only one node when argument n equals one. When n is greater than one, it recursively generates a valid Rigidity tree of $n - 1$ nodes and then builds a set of candidates adjacent to which a new node can be added to get a valid n node Rigidity Tree. The candidate set contains only those nodes which either don't have degree equal to four or those which have degree four but do not have any neighbour which has degree five. The reason such nodes are excluded is that if a node has degree greater than or equal to five, then it has to represent a maximally rigid component, and two nodes which correspond to maximally rigid components can not be adjacent to each other in the Rigidity tree.

Algorithm 1 Generate Random Rigidity Tree(n)

```
1: if $n = 1$ then
2:     $G_n \leftarrow$ Tree with only one node
3: else
4:     $G_{n-1} \leftarrow$ Generate Random Rigidity Tree(n-1)
5:     $L \leftarrow$ Set of degree 4 vertives having at least one adjacent vertex with
   degree $\geq 5$
6:     Add a node adjacent to a random node not in L
7:     $G_n \leftarrow$ final graph
8:
9: return $G_n$
```

We claim that the above procedure indeed generates a random rigidity tree with n nodes. We shall prove our claim by induction. But, Let us first consider the following Lemma.

2.3.2 Proof of Correctness

Lemma 2.3.2. *Consider generation process of a n node rigidity tree. Each intermediate rigidity tree in this generation process is also a valid rigidity tree.*

Proof. The correctness of the lemma is easy to see as addition of a node can only violate the structural constraint of the rigidity and not removal of a node. Thus, if we have a valid node rigidity tree, then tree obtained after removing any node of the tree must also be a valid rigidity tree because if it was not, then the originial tree will not be a valid rigidity tree thus contradicting the assumption itself. □

Coming back to proof of the correctness of the algorithm, the induction statement P_n is as follows:
P_n: The algorithm generates all rigidity trees of n vertices with some non zero probability and generates no tree which violates the constraints on the rigidity tree.

Proof. Base Case: Clearly P_1 is true since the only rigidity tree possible with one node is a tree containing only one node and no edges. Thus P_1 is trivially true.

Inductive Hypothesis: Let P_n be true for all $n \leq k$.

Inductive Step: When the algorithm is called with $(k+1)$ as parameter, it first calls itself recursively with k as parameter. Now assuming that the P_k is true, we can say that the Algorithm generates all possible rigidity trees with k nodes with some probability and no tree which does not follow the constraints is generated. Now consider any valid $(k+1)$ node rigidity tree and a k node sub-graph of this rigidity tree. Clearly the k node sub-graph is a rigidity tree of k node owing to structural property of the rigidity tree. Also, the left over node will be a leaf node which could either be adjacent to a node with degree $= 4$ which itself has no node with degree ≥ 5 adjacent to it or a node which does not follow these constraints. Since the algorithm considers both the cases, and the only violation to structural property can occur if 2 nodes with degree ≥ 5 become adjacent, we don't violate the structural property while considering all the possibilities for adding a node.

Thus, P_{k+1} is true.

Thus by Mathematical Induction we can say that P_n is true $\forall$ n $\geq$ 1.

$\square$

2.3.3 Assigning Rigidity Labels

Having generated a random Rigidity Tree, we now want to fix the type of each node that is determine which all nodes are Maximally Rigid Components and which all nodes are Quadrilaterals.

Algorithm 2 Assigning Labels to Nodes (Rigidity Tree rgt)

```
1: $S_1 \leftarrow$ Set of nodes in rgt with deg $\geq 5$
2: $S \leftarrow$ Set of all nodes in rgt
3: for node $n$ in $S_1$ do
4:     Label[n] $\leftarrow$ Maximally Rigid
5:     $S \leftarrow S - \{n\}$
6:     for neighbor $s$ of $n$ do
7:         Label[s] $\leftarrow$ Non-Rigid
8:         $S \leftarrow S - \{s\}$
9: while $S \neq \phi$ do
10:    Choose any random node $n$ from $S$
11:    Choose a random number $k$
12:    if $k$ is odd then
13:        Label[n] $\leftarrow$ Maximally Rigid
14:        $S = S - \{n\}$
15:        for neighbor $s$ of $n$ do
16:            Label[s] $\leftarrow$ Non-Rigid
17:            $S \leftarrow S - \{s\}$
18:    else
19:        Label[n] $\leftarrow$ Non-Rigid
20:        $S = S - \{n\}$
21: return Label
```

The above algorithm first fixes the label of the nodes which have to Maximally Line Rigid. Once done, it fixes the labels of the immediate neighbors as Quadrilaterals. This is what we term as the *one neighborhood expansion* rule, i.e. if a node is fixed to be Maximally Line Rigid, then fix the label of its' neighbors as Quadrilateral. For the nodes other than those for which the Label was fixed to be Maximally Line Rigid i.e. those which had degree not equal to five, the algorithm picks a random node and assigns a random label and accordingly applies the one neighborhood expansion rule.

2.3.4 Generating Line Rigid Graph

Having generated a random Rigidity tree and fixing the labels of each node randomly, we now wish to generate the random *Maximally Rigid Component* nodes of the rigidity tree. For this we need to figure out an algorithm which can generate random Line Rigid graphs. The Algorithm 3 presented on next page does that.

2.3.5 Proof of Correctness

Clearly, any graph generated by this algorithm is Line Rigid since we mainly try and apply 3 operations on an already generated Rigid Component which are:

1) Adding a new vertex adjacent to at least 2 vertices of the original component. Clearly, if the original component is Line Rigid, this new component will also be line rigid since at least 2 line constraints are applied on the new vertex, hence its' position would be fixed and we would have only one arrangement of vertices satisfying the edge weight constraints. 2) Joining 2 non adjacent vertices of the original component will also lead to a line rigid component if the original component was line rigid. This is easy to see, since the edge weight of this new edge is going to be fixed by the original component itself, because it was line rigid. 3) Superimposing 2 line rigid components. This will also lead to a line rigid component, since the relative

Algorithm 3 Generate Random Line Rigid Graph(initialComponent)

```
1: $final \leftarrow \phi$
2: if $initialComponent \neq \phi$ then
3:     Choose a random number $p$
4:     if $p\%2 = 0$ then
5:         $final \leftarrow initialComponent$
6: else
7:     Choose a random number $k$
8:     if $k\%2 = 0$ then
9:         $final \leftarrow$ Generate Random Line Rigid Graph($K_3$)
10:    else if $k\%2 = 1$ then
11:        $final \leftarrow$ Generate Random Line Rigid Graph($K_{2,3}$)
12: Choose a random number $t$
13: if $t\%3 = 0$ then
14:     Choose random number $r$ s.t. $2 \leq r \leq$ number of vertices in
                $initialComponent$
15:     Choose $r$ random vertices from $initialComponent$
16:     Add a new vertex v s.t. v is adjacent to all these r vertices
17:     $G \leftarrow$ final graph
18:     $final \leftarrow$ Generate Random Line Rigid Graph(G)
19: else if $t\%3 = 1$ then
20:     Choose two random vertices from initialComponent s.t. they don't
                already have an edge between them
21:     Add an edge between the two vertices
22:     $G \leftarrow$ final graph
23:     $final \leftarrow$ Generate Random Line Rigid Graph(G)
24: else
25:     $M \leftarrow$ Generate Random Line Rigid Graph($\phi$)
26:     Choose a random edge $e_1$ from initialComponent
27:     Choose a random edge $e_2$ from M
28:     Superimpose $e_1$ on $e_2$
29:     Let G be the final graph
30:     $final \leftarrow$ Generate Random Line Rigid Graph (G)
31: return $final$
```

positions of all the vertices except the vertices involved in superposition are fixed, so we will finaly get only one valid linear arrangement of vertices.

All these points rely on one point now that the initial original component must be line rigid. This, however is easy to see again because the only basic components returned in the algorithm are $K_{2,3}$ and K_3 which themselves are line rigid.

Thus, we have shown that any graph generated by our algorithm is line rigid. Now, we need to prove that all possible line rigid graphs can be generated by our algorithm.

Fact: Each line rigid graph must have a $K_{2,3}$ or a K_3 as a subgraph.

The above fact was proved by [3]. Now, we will prove that any Line Rigid graph which has a $K_{2,3}$ or a K_3 as a subgraph can be generated by our algorithm and thus complete the proof.

Without loss of generality, lets' consider a line rigid graph which has a K_3 as a subgraph. Now, consider a vertex which has at least 2 edges incident to any of the vertices of the K_3.

Case 1: No such vertex exists. Case 1a: The vertex is incident to no other vertex. In this case, we have a dangling edge and hence the original graph can not be line rigid. Case 1b: The connected component of the vertex apart from the K_3 has no vertex incident to the K_3 either. In this case, the vertex is an articulation point, and we are not considering those graphs which have an articulation point or hinge edges. Case 1c: The connected component of the vertex(s) apart from K_3 has at least one vertex incident to the K_3. In this case, the vertex from the connected component which has an edge incident to the K_3 must be adjacent to the vertex s otherwise, the graph will not be a weakly-triangulated graph. Now, this adjacent vertex (adj(s)) can either have an edge incident to the same vertex of K_3 to which s had an edge, or

to a different vertex.
Case 1ci:When it has an edge to the same vertex, this vertex of K_3 to which both s and adj(s) have an edge becomes articulation point.
Case 1cj: In case it has an edge to a different vertex, we have a quadrilateral in between, which would itself be a different node in the rigidity tree being adjacent to 2 line rigid components.

Thus, we have shown that there exists a vertex which is adjacent to at least 2 vertices of K_3. Now, this same proof will work for any line rigid graph i.e. at each point, if we want to expand the graph by one vertex, that vertex should be adjacent to at least 2 vertices of the original line rigid graph. Further, once we have added the relevant vertices, we can see the second way of expanding the lien rigid component i.e. adding edges between non-adjacent vertices can be used to obtain the entire graph.

Thus, our Algorithm for genration of line rigid graphs is both sound and complete.

Chapter 3

Linear layouts of Chordal Bipartite Graphs

3.1 Preliminaries

Definition 3.1.1. A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets X and Y such that every edge connects a vertex in X to one in Y. It can be represented as $G =$ *(X,Y,E)* where every edge $e \in E$ and $e = xy$ is such that $x \in X, y \in Y$.

Definition 3.1.2. A *complete bipartite graph* is a bipartite graph $G =$ *(X,Y,E)* in which every vertex in X is connected to every vertex in Y.

Definition 3.1.3. A bipartite graph is called *Chordal Bipartite* if it does not contain any chordless cycle of length greater than four.

Lemma 3.1.4. *If a bipartite graph does not contain a any chordless cycle of length greater than four, even its complement graph cannot contain any chordless cycle of length greater than four.*

Proof. Let the original graph be $G = (X, Y, E)$. Lets assume that the lemma is false and consider a chordless cycle of length l ($l > 4$) in the complement graph $\overline{G}$. In the complement graph, the vertex sets X and Y form two cliques (each vertex is connected to every other vertex). Since $l > 4$, at least three of l vertices of the cycle must belong to the same vertex set X or Y of the original bipartite graph. Suppose vertices $u, v, w \in X$ are part of the cycle and $\overline{G}$ has edges uv, vw, uw which form a triangle. The chordless cycle cannot not contain all of these three edges because $l > 4$ and the left out edge becomes a chord for the cycle. Hence, $\overline{G}$ cannot contain any chordless cycle of length greater than four. □

In other words, a bipartite graph which is also weakly triangulated is called Chordal Bipartite Graph. We use notation $N(x)$ to denote set of neighbors of vertex x and $ING_{x,y}$ to denote the subgraph induced by vertices of $N(x)$ and $N(y)$.

Definition 3.1.5. An edge $e = xy$ of a bipartite graph $G =$ *(X,Y,E)* is *bisimplicial* if $ING_{x,y}$ is a complete bipartite subgraph of G. A chordal bipartite graph can have more than one bisimplicial edge. The edges x_1y_1,x_4y_4 are a bisimplicial edge in Figure 3.1(a).

Definition 3.1.6. A set of graphs $G_0, G_1, ..., G_k$ where $G_0 = G$ and G_i is obtained from G_{i-1} by removing a bisimplical edge $e = xy$ and the edges incident on x and y and finally G_k is an empty graph. An ordering of edges $\sigma = [e_1, e_2, ..., e_k]$ is called *Perfect Edge without Vertex Elimination Ordering*(*PEWVE Ordering*) if each edge e_i in σ is a bisimplicial edge for the graph G_{i-1}.A graph is chordal bipartite if and only if it has at least a PEWVE Ordering.

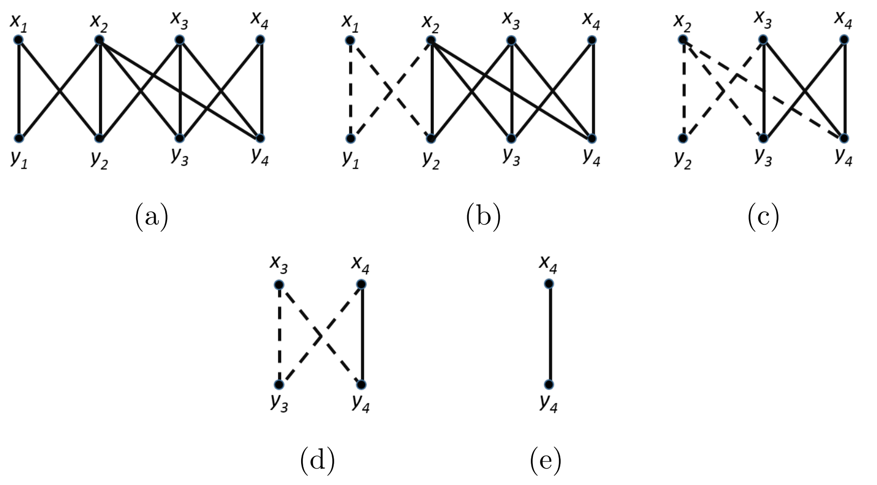


Figure 3.1: Perfect edge without vertex elimination ordering from 3.1a to 3.1e.

Consider the bipartite graph H as shown in Figure 3.1(a). For this graph H, x_1y_1 is a bisimplicial edge because $(N(x_1)=\{y_2\})$ + $(N(y_1)=\{x_2\})$ forms a complete bipartite subgraph and x_2y_2 is not a bisimplicial edge because $(N(x_2)=\{y_1, y_3, y_4\})$ + $(N(y_2)=\{x_1, x_3\})$ does not form a complete bipartite subgraph as H does not contain edge x_3y_1. From 3.7 we can see that $[x_1y_1, x_2y_2, x_3y_3, x_4y_4]$ is a perfect edge elimination ordering because each of them is bisimplicial edge at each stage of perfect edge elimination from Figure 3.1(a) to 3.1(e).

3.2 Constructing Rigidity Tree

Asish Mukhopadhyay *et al* [3] constructed Rigidity Tree based on reverse peripheral edge order of the given weakly triangulated graph. We are building the graph starting from an empty graph by adding peripheral edges one by one in the reverse order of Peripheral Edge Ordering of the original weakly triangulated graph and simultaneously update the changes in

Rigidity Tree. Rigidity Tree construction includes addition of new nodes like triangles, quadrilaterals and $K_{2,3}$. Throughout this chapter, consider e_i as the peripheral edge that is being newly added and is between vertices x and y. Construction of rigidity tree has a bottleneck of finding number of three length and two length paths between x and y which takes $O(n^2)$ time to find the formation of new nodes and transformation of non-rigid components into rigid ones.

In our case of chordal bipartite graphs being bipartite, we do not need to consider formation of the triangles and there won't be any two length paths between x and y since bipartite graphs do not contain odd cycles. So, our only concern lies in finding out number of three length paths between x and y more efficiently.

Depending on the number of three length paths existing between x and y there can be different changes in the rigidity tree.

Case 1 (*Number of three length paths = 0*):

This case is trivial. The newly added peripheral edge e_i will be a dangling edge and hence there won't be any effect on the rigidity tree.

Case 2 (*Number of three length paths = 1*):

In this case, a new non-rigid quadrilateral is formed and hence a new node will be added to rigidity tree. See Fig. 3.2a

Case 3 (*Number of three length paths $\geq$ 2 with no common edge among these paths*):

In this case, e_i is added to rigid component and there will be no change in rigidity tree structure. See Fig. 3.2b. Note that we need to have chords between x and y so as to keep the graph chordal bipartite.

Case 4 (*Number of three length paths = 2 with a common edge among these paths*):

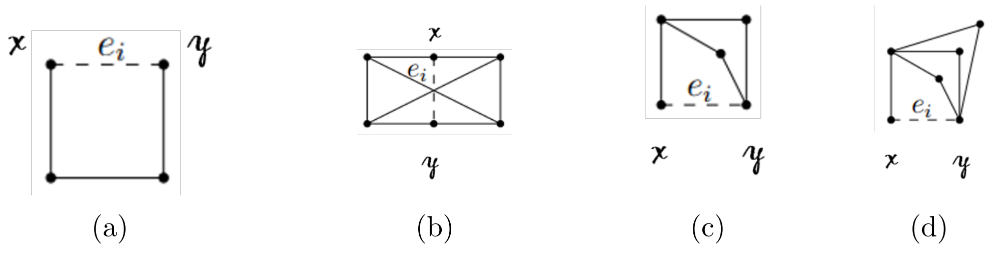


Figure 3.2: Different possibilities for e_i, newly added peripheral edge between x and y: (a)Single three length path (b)More than one three length path with no common edge (c)Two three length paths with a common edge (d)More than two three length paths with a common edge.

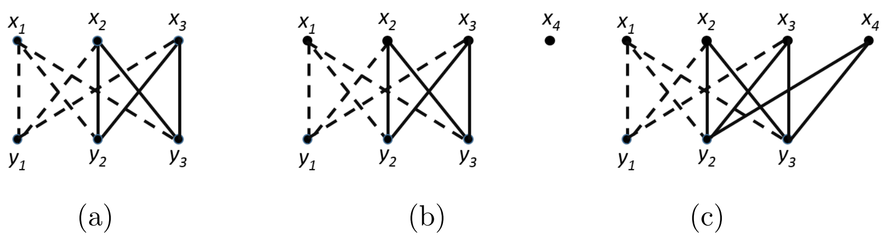


Figure 3.3: Different cases for x_iy_i, the newly added bisimplical edge.

In this case, quadrilateral, a dangling edge and e_i together forms new rigid $K_{2,3}$ component and this is added as a new node in rigidity tree. See Fig. 3.2c

Case 5 (*Number of three length paths $\geq$ 3 with a common edge among these paths*):

In this case, dangling edge becomes part of an old rigid component and hence no change in rigidity tree. See Fig. 3.2d

3.3 Computing Peripheral Edge Order

Lemma 3.3.1. *Let $x_iy_i$be bisimplical edge in the graph G. Each edge incident on x_i or y_i is a peripheral edge in $G - x_iy_i$ if the graph does not have some special kind of edges.*

Proof. The edges can be divided into three types of cases:-

Case 1

This is a bisimplical edge which is clearly chorded by the complete bipartite graph encompassed by its neighbours as shown by x_1y_1 in Figure 3.3(a).

Case 2

This is the edge of type x_1y_2 non bisimplical in Figure 3.3(b) wherein this edge is not the middle edge of a chordless P_4.

Case 3

This is the edge of type x_1y_2 non bisimplical in Figure 3.3(b) wherein this edge is the middle edge of a chordless P_4 $y_3x_1y_2x_4$. □

We will assume that the chordal bipartite graph does not have edges of type Case 3 from here on-wards and develop a linear time algorithm to calculate the linear layout.

3.3.1 Proposed Algorithm

Computing the PEWVE Ordering

According to Kloks and Kratsch[2] the Perfect Edge without Vertex Elimination ordering for a graph can easily be calculated from the doubly lexical ordering of the bipartite adjacency matrix in $O(min(m \log n, n^2))$ where m is the number of edges and n is the number of vertices of the graph. Ryuhei Uehara [5] showed that we can compute this ordering in $O(m + n)$ time.

We assume that the given Chordal Bipartite Graph does not contain hinge edges or articulation points and we will find out the number of linear layouts

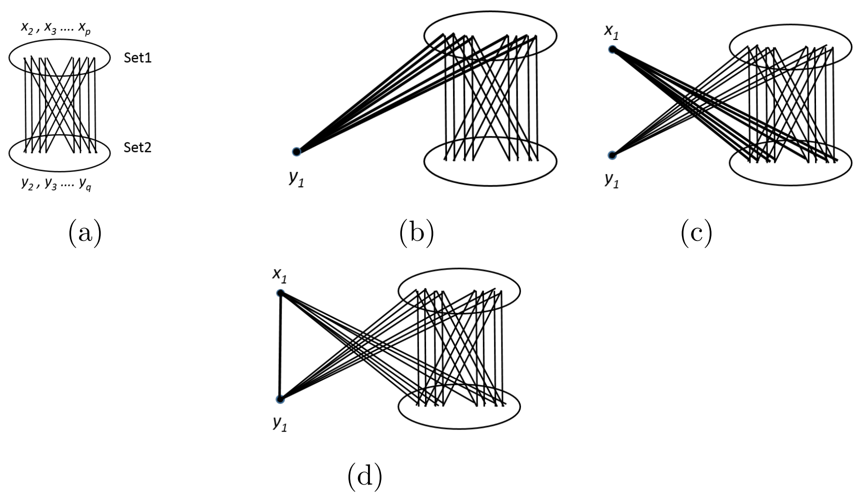


Figure 3.4: Perfect edge elimination ordering from 3.4a to 3.4d.

of the given graph more efficiently when compared to finding the same for weakly triangulated graphs as done by Asish Mukhopadhyay *et al* [3].

PEWVE Ordering to Peripheral Edge Order

The algorithm that will described below is applied for Chordal Bipartite graph $G = (X, Y, E)$ with n vertices and m edges.

Description

Add every edge in the reverse of the PEWVE Ordering after adding all the edges incident on the vertices of the added edge. Doing this produces a valid peripheral edge ordering . Let us remind you that a peripheral edge is an edge such that it is not the middle edge of a chordless P4.

Proof of Correctness

The proof of the correctness of the algorithm can be done through Mathematical Induction. The proof assumes the correctness upto adding of the k-1 th edge in the reverse PEWVE Ordering and extends that to the k th edge. It is as follows:-

Base Case-

In case of k=1 as shown in the figure it is a peripheral edge as it is the lone edge and hence cannot be a middle edge of a chordless P4..

General Case-

Assume without loss of generality that the algorithm is correct upto the addition of the (k-1)th edge in the reverse of the PEWVE Ordering. Then if we add the edges incident on the vertices of this edge these edges are peripheral as these are the dangling edges that cannot be a middle edge. Also note that the edge has not been added thus far hence removing a possibility of the edges to be a middle edge in the graph constructed so far.

Now add the k th edge in the reverse of the PEWVE Ordering to the graph thus far. This is also a peripheral edge since by the definition of the bisimplicial edge the neighbourhood of the vertices induce a complete bipartite graph which will serve as a chord for any potential P4 for which this edge is the middle edge. This has been explained further in the figure 3.4.

Complexity Analysis

The finding of the PEWVE order takes $O(m \log n, n^2)$. Getting the peripheral edges from the PEWVE ordering takes $O(m)$ extra time. Note that we need to maintain $ING_{x,y}$ for the bisimplicial edges .

Lemma 3.3.2. *Assuming the given graph doesn't contain any hinge edges or articulation points, even if we consider that the addition of a peripheral edge* *that is not bisimplicial during the reconstruction of the original graph doesn't change the structure of the Rigidity Tree, the final Rigidity Tree structure do not change.*

Proof. This lemma is important because there are only $O(n)$ number of bisimplicial edges whereas there are $O(m)$ peripheral edges which reduces our task of monitoring the formation of new $K_{2,3}$'s and quadrilaterals which are the only minimally rigid graphs possible in chordal bipartite graphs. There can be a problem in completing this task if any non-bisimplicial but peripheral edge completes the formation of a new quadrilateral or $K_{2,3}$ in which case it goes unnoticed. So, if a peripheral edge that is not bisimplicial is the last edge in reverse peripheral edge order that completes the formation of quadrilateral or $K_{2,3}$, it may not be detected.

According to the algorithm of finding out peripheral edge ordering that is described earlier, the only way in which an edge can be peripheral but not bisimplicial is when it is the incident edge of one of the vertices of a bisimplicial edge. Also, if the first bisimplicial edge that is chosen in PEWVE Ordering is among the edges of quadrilateral or $K_{2,3}$, it will be the last edge to be added to complete quadrilateral or $K_{2,3}$ while reconstructing the graph. So, the first bisimplicial edge should be outside quadrilateral or $K_{2,3}$ and to make any edge of quadrilateral or $K_{2,3}$ non-bisimplicial, the bisimplicial edge should be incident on one of the vertices of quadrilateral or $K_{2,3}$.

First consider the case of $K_{2,3}$. Let $e_i = xy$ be the bisimplical edge with x being one of the vertices of $K_{2,3}$. $N(x)$ has at least two vertices. y is not be part of $K_{2,3}$. If $N(y)$ is empty, then xy becomes a dangling edge and x will be an articulation point. But, we assumed that given chordal bipartite graph does not have any articulation points. So, $N(y)$ has at least one vertex. Consider when $N(x)$ has two vertices and $N(y)$ has one vertex

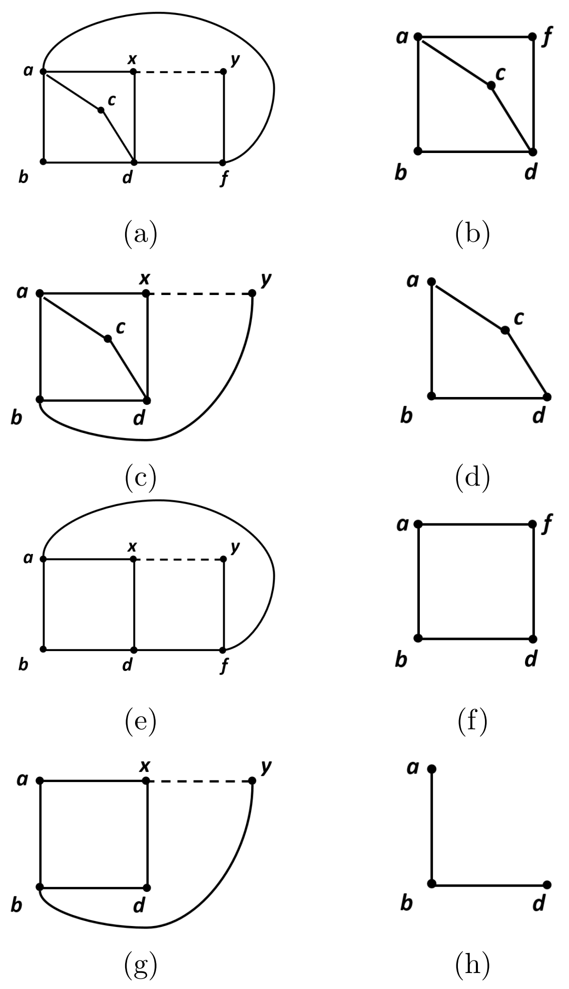


Figure 3.5: (a,c,e,g) xy is a bisimplicial edge. (b,d,f,h) After removing edge xy and edges incident on x and y from graphs in (a,c,e,g) respectively

then the vertex in $N(y)$ may or may not be part of $K_{2,3}$ as we can see in Fig.3.5a and Fig.3.5c. Suppose the vertex in $N(y)$ is not part of $K_{2,3}$, removing edges xy and edges in $ING_{x,y}$ gives $K_{2,3}$ as we can see in Fig.3.5b. So, when we construct the original graph in reverse peripheral edge order, $K_{2,3}$ is formed first which can be detected and $ING_{x,y}$ and edge xy are added to rigid component. So, rigidity tree will be constructed correctly in this case. Now, consider the vertex in $N(y)$ to be part of $K_{2,3}$ as shown in Fig.3.5c. Upon removing edges xy and edges in $ING_{x,y}$, we get a non-rigid quadrilateral. So, while reconstructing the original graph, first the rigidity tree contains component corresponding to non-rigid quadrilateral which then becomes rigid because it shares two of its edges with $K_{2,3}$ which is formed due to edges of $ING_{x,y}$, edge xy and edges induced on x and y. For example, in Fig.3.5c, vertices a,b,c,d form quadrilateral and vertices (a,d,y),(b,x) form $K_{2,3}$ and the quadrilateral and $K_{2,3}$ share two edges *ab* and *bd*. According to Asish Mukhopadhyay *et al*, there is no component cycle of length two in rigidity tree (lemma 4 of [3]). So, in both the cases for the single vertex in $N(y)$, rigidity tree is constructed correctly. Even if $ING_{x,y} \supset K_{1,2}$, rigidity tree is constructed correctly because removal of edge xy and edges induced on x and y gives a supergraph of $K_{2,3}$ which is line rigid.

Now, consider the case of quadrilateral. In the lines similar to the above argument, $N(y)$ should have at least one vertex. Consider $N(x)$ contains two vertices and $N(y)$ contains one vertex. When the vertex in $N(y)$ is not part of quadrilateral (shown in Fig.3.5f), it forms a quadrilateral after removing edge xy and edges incident on x and y which is formed earlier in reconstructing the original graph which will be later relabeled as rigid because it shares two edges with $K_{2,3}$ which is formed by vertices $\{\ a, d, x, y\}$. So in this case, even though formation of a different quadrilateral is recognized rather than the quadrilateral *abcd*, it is correctly labeled at the end as a rigid component. When the vertex in $N(y)$ is part of quadrilateral, it becomes exactly as $K_{2,3}$ without any additional edges. So, this case can be handled correctly. Even when $ING_{x,y} \supset K_{1,2}$, even though the formation of quadrilateral may not be detected correctly, the component is labeled correctly as rigid because edge xy, edges in $ING_{x,y}$, edges induced on x and y form a graph $\supset K_{2,3}$ and the quadrilateral shares two of its edges with it.

Hence, by the above argument we can conclude that even though if we do not consider that non-bisimplicial peripheral edges can change the rigidity tree, the final rigidity tree structure do not change and hence we only need to consider the addition of peripheral edges that are also bisimplicial to effect the changes in rigidity tree. □

3.3.2 Counting three length paths efficiently

According to Lemma 3.3.2, if the next peripheral edge e_i added is not a bisimplicial edge, then rigidity tree structure cannot change and hence we do not need to find number of three length paths between x and y. However, if e_i is bisimplicial edge, we need to count the number of three length paths. In Fig. 3.7, there are 5 different cases dependent on $ING_{x,y}$ each corresponding 5 different cases for number of three length paths between x and y.

Case 1:

$ING_{x,y} = K_{1,0}$ corresponding to number of three length paths = 0. For example see Fig. 3.7a.

Case 2:

$ING_{x,y} = K_{1,1}$ corresponding to number of three length paths = 1. A new non-rigid quadrilateral is formed. For example see Fig. 3.6b.

Case 3:

$ING_{x,y} = K_{2,2}$ corresponding to number of three length paths ≥ 2 with no common edge among these paths. For example see Fig. 3.6c.

Case 4:

$ING_{x,y} = K_{1,2}$ corresponding to number of three length paths $= 2$ with a common edge among these paths. A new rigid $K_{2,3}$ component is formed. For example see Fig. 3.6d.

Case 5: *Number of three length paths $\geq$ three with a common edge among these paths* $ING_{x,y} = K_{2,2}$ corresponding to number of three length paths $\geq$ 3 with a common edge among these paths. For example see Fig. 3.6e.

In brief, we can say that–

- if $ING_{x,y} \subset K_{1,1}$, e_i is a dangling edge.
- if $ING_{x,y} = K_{1,1}$, a non-rigid quadrilateral is formed.
- if $ING_{x,y} = K_{1,2}$, $K_{2,3}$ is formed.
- if $ING_{x,y} \supset K_{1,2}$, e_i becomes part of a rigid component.

Since, we know that edge e_i is a bisimplicial edge, we can find out $ING_{x,y}$ in constant time if we store $ING_{x,y}$ (in terms of $K_{i,j}$ where i,j are some non-negative integers) corresponding to each bisimplicial edge while calculating PEWVE Ordering itself and hence overall it takes $O(m)$ time for m peripheral edges. Thus, we could reduce the bottleneck of finding changes in rigidity tree and constructing it from $O(n^2m)$ time to $O(m)$ time for chordal bipartite graphs.

Using Union-Find for merging components

The addition of an edge can change the rigidity of a component from non-rigid to rigid. Since a rigid component may now be adjacent to one

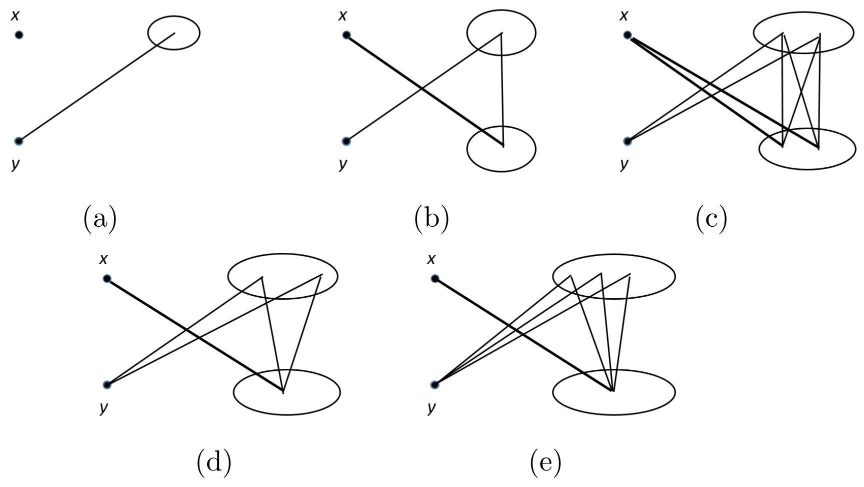
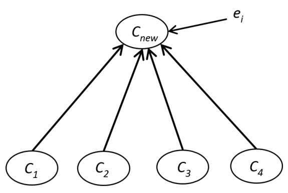


Figure 3.6: Different cases of $ING_{x,y}$ with xy being a bisimplicial edge (a)$ING_{x,y} = K_{1,0}$ (b)$ING_{x,y} = K_{1,1}$ (c)$ING_{x,y} = K_{2,2}$ (d)$ING_{x,y} = K_{1,2}$ (e)$ING_{x,y} = K_{1,3}$

or more rigid components in the rigidity tree each of the edges in those components must now me merged to a new component c_{new}. In worst case each edge in the graph might have to go through the changes and thus in Asish Mukhopadhyay *et al* [3] this used to take $O(m^2)$ time.We can now use the Union-Find data structure with Path Compression to massively reduce this complexity to linear time. For this we maintain a new tree which is the component tree (a tree formed out of the components of the rigidity tree). Now we do not need to change the components pointers of each of the edges separately.For example if in the i^{th} state edge e_i is added and a new component c_{new} is formed. Suppose it is adjacent to constant number (at max 4) since we are considering the absence of hinge edges.So in worst case let it be adjacent to 4 components c_1, c_2, c_3, and c_4. Since our component tree already has 4 edges corresponding to the already existing components we will add a new vertex c_{new} and draw and edge from each of the four components to it(Similar as in a Union_find data structure). Now to find



(a)

Figure 3.7: Disjoint set data structure when components C_1, C_2, C_3, C_4 merged to form new component C_{new}

the corresponding component of an edge consider its original component and traverse in the Union Find data structure to find the ultimate component to which it belongs.

Complexity Analysis

The finding of the PEWVE order takes $O(m \log n, n^2)$ according to Kloks and Kratsch and Ryuhei Uehara improved this computation and can be done in $O(m + n)$ time. Getting the peripheral edges from the PEWVE ordering takes $O(m)$ extra time. Note that we need to maintain $ING_{x,y}$ for the bisimplicial edges .The addition of peripheral edges that are not bisimplicial cannot be ignored simply because they change the neighborhood of the vertices which are the endpoints of bisimplicial edges.The calculation of whether they form a $K_{1,1}$, a subset of $K_{1,1}$, $K_{1,2}$, or if $K_{1,2}$ is its subset can be done in constant time. Thus there is no need to spend $O(n^2)$ time to calculate the number of three paths as earlier since this step is merely reduced to a constant time operation on the adjacency list of a graph. From Asish Mukhopadhyay *et al* [3] we had seen that the rigidity tree construction had an upperbound of $O(m^2)$ since the merging of a rigid component with other rigid component has $O(m)$ time complexity and there are a maximum of $O(m)$ such peripheral edges(Though this bound is very weak). However in our case the merging can only occur when we encounter a peripheral edge which is also bisimplicial. The upperbound on number of bisimplicial edges is $O(n)$ and each such merging takes $O(1)$ time thanks to the Union-Find Data Structure so the total time complexity is $O(n)$ in our case using a different data structure. So, we can find the number of linear layouts (2^q where q is the number of quadrilaterals in the given graph) for a given chordal bipartite graphs without any articulation points and hinge edges in $O(m + n)$ time.

Chapter 4

Future Work and Conclusion

4.1 Conclusion

In the problem of generating weakly triangulated graph randomly, we have seen how to generate random rigidity tree. Also, we have brought together few techniques which can be used to solve various other problems of generating constrained weakly triangulated graphs.We have also seen how to assign labels to the nodes in the rigidity tree and finally generated the complete Weakly Triangulated Graph.
The algorithm proposed by Asish Mukhopadhyay *et al* [3] for computing the linear layouts of a weakly triangulated graph with n vertices and m edges took $O(n^2m)$ time. We proposed an improved $O(m + n)$ time algorithm to solve the same problem for chordal bipartite graphs without articulation points and hinge edges.

4.2 Future Work

Further towards solving generation of random weakly triangulated graph problem, we would like to generate the Weakly Triangulated Graphs without any constraints since currently we considered generating only those weakly triangulated graphs which didn't have any hinge edges and articulation points. We proposed $O(m+n)$ time algorithm for computing the linear layouts of chordal bipartite graphs without articulation points and hinge edges. Now, we would like to extend this algorithm for general chordal bipartite graphs that may contain articulation points and hinge edges.

Bibliography

[1] Hitoshi IBA. Random tree generation for genetic programming. *International Conference on Evolutionary Computation*, 2005.

[2] Ton Kloks and Dieter Kratsch. Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph. *Information Processing Letters*, 55(1):11–16, 1995.

[3] Asish Mukhopadhyay, SV Rao, Sidharth Pardeshi, and Srinivas Gundlapalli. Linear layouts of weakly triangulated graphs. *Algorithms and Computation*, pages 322–336, 2014.

[4] Liviu Panait Sean Luke. A survey and comparison of tree generation algorithms. 2001.

[5] Ryuhei Uehara. Linear time algorithms on chordal bipartite and strongly chordal graphs. *Automata, languages and programming*, pages 993–1004, 2002.