7-7-2020

Weakly Chordal Graphs: An Experimental Study

Sudiksha Khanduja
University of Windsor

Weakly Chordal Graphs: An Experimental Study

by

Sudiksha Khanduja

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

2020

Weakly Chordal Graphs: An Experimental Study

by

Sudiksha Khanduja

APPROVED BY:

Y. Aneja
Odette School of Business

D. Wu
School of Computer Science

A. Mukhopadhyay
School of Computer Science

May 15, 2020

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis 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 anyone's 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 or Institution.

ABSTRACT

Graph theory is an important field that enables one to get general ideas about graphs and their properties. There are many situations (such as generating all linear layouts of weakly chordal graphs) where we want to generate instances to test algorithms for weakly chordal graphs. In my thesis we address the algorithmic problem of generating weakly chordal graphs. A graph $G=(V, E)$, where V is its vertices and E is its edges, is called weakly chordal graph, if neither G, nor its complement $\overline{G}$, contains an induced chordless cycle on five or more vertices.

Our work is in two parts. In the first part, we carry out a comparative study of two existing algorithms for generating weakly chordal graphs. The first algorithm for generating weakly chordal graphs, repeatedly finds a two-pair and adds an edge between them. The second generation algorithm starts by constructing a tree and then generates an orthogonal layout (also weakly chordal graph) based on this tree. Edges are then inserted into this orthogonal layout till there are m edges. The output graphs from these two methods are compared with respect to several parameters like number of four cycles, run times, chromatic number, the number of non two-pairs in the graphs generated by the second method.

In the second part, we propose an algorithm for generating weakly chordal graphs by edge deletions starting from an arbitrary input random graph. The algorithm starts with an arbitrary graph to be able to generate a weakly chordal graph by the basis of edge deletion. The algorithm iterates by maintaining weak chordality by preventing any hole or antihole configurations being formed for any successful deletion of an edge.

DEDICATION

To my teachers, loving family & friends who have supported me in every step of my life

ACKNOWLEDGMENTS

I express my sincere gratitude to Dr. Ashish Mukopadhyay, without whose patient guidance and constant supervision, I would not have come so far.

I offer my sincere appreciation to the committee members, Prof. Dan Wu and Prof. Yash Aneja for their useful critiques and advice.

My special thanks to Sarthak, Soumya, Aayushi, Lokesh, Saurav, Anjali, Akshat, Rahul, Jayanth & Jugal, who spent time in active discussions and gave moral support that helped me prepare for my presentations and help finish my work.

My grateful thanks is also extended to my colleagues cum friends for their invaluable help throughout my Master's degree. Finally & most of all to my loving parents & family for their unparalleled encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Graph theory is a field of study that helps supply numerical information for enumerative problems and to provide a source from which specimen graphs can be taken for use in real-life problems. It enables one to get general ideas about graphs and their properties. There are many situations (such as generating all linear layouts of weakly chordal graphs) where we would want to generate instances of these to test algorithms for the class of weakly chordal graphs. In this thesis, we address the algorithmic problem of generating weakly chordal graphs. Weakly chordal graphs are a class of perfect graphs introduced by Hayward in 1985 [13].

1.1 Graph Generation

Early work in the field of generating graphs used to focused mainly on creating catalogues of small sized graphs. The authors [8] published a catalogue of all graphs to exist on 10 vertices. The motivation to do this was that, such repositories were considered extremely useful for providing counterexamples to old conjectures and to come up with new ones. The focus then shifted to generating graphs of arbitrary size uniformly at random be it labeled or unlabeled. As a generation method like that, would require a solution for the counting problem, research was then focused to the classes of graphs for which the counting problem could be solved and yielded polynomial time generation algorithms. Such were graphs with

prescribed degree sequences, special classes of graphs such as outer planar graphs, maximal planar graphs, regular graphs. One can refer to [25] for a complete survey work prior to 1990.

As stated in [20], there are many situations where one would want to generate instances of these to test algorithms for weakly chordal graphs. For example, in [3] the authors have generated all the existing linear layouts of weakly chordal graphs. A generation mechanism could be used to obtain test instances for this algorithm. It can do the same for more optimization algorithms, like finding a maximum clique, maximum stable set, minimum clique cover, minimum coloring, for both weighted and unweighted versions, for weakly chordal graphs proposed in [21] and their improved versions in [22], [23].

1.2 Problem Statement

We focus on the problem of generating a weakly chordal graph. While there is a considerable amount of research done in order to generate algorithms to generate weakly chordal graph but the authors in [1] proposed an open problem of generating weakly chordal graph starting from an arbitrary graph. A solution to this open problem in [1] starting from an arbitrary graph is the main contribution of this thesis. An application of this generation algorithm would be to obtain test-instances for an algorithm for enumerating linear layouts of a weakly chordal graph proposed in [3].

1.3 Motivation

The motivation of the problem comes from the need to establish test-instances for an algorithm. When the distribution is unknown, the assumption of uniform distribution might still help. Otherwise, we might look upon a generation algorithm as providing test-instances for an algorithm to enumerate linear layouts of weakly

chordal graph. With this motive, an algorithm for generating weakly chordal graphs by adding edges incrementally was recently proposed in [20]. An applied application of this generation algorithm would be to obtain test-instances for an algorithm for enumerating linear layouts of a weakly chordal graph proposed in [3].

1.4 Thesis Organization

The list below presents the organization of the chapters which makes up this thesis. Also given is a brief description of the topics each chapter deals with.

- Chapter 2 gives a clear background knowledge on the class of weakly chordal graphs. The chapter outlines the theoretical characterizations about weakly chordal graphs along with a preliminaries section that has all definitions and notations used in the remainder of this thesis. It outlines summary from current literature while reviewing a comparison of two already existing methods to generate a weakly chordal graphs.

- Chapter 3 describes the proposed algorithm and its inner workings giving justification for the chosen approach at each step and also shows the experimental results after applying our algorithm.

- Chapter 4 concludes the work done in this thesis and suggests some possible future research directions.

- Bibliography declares a detailed list of references from which facts and numbers have been used as a guide for this thesis.

Chapter 2

Weakly Chordal Graphs

A simple, undirected graph $G = (V, E)$ is said to be weakly chordal if neither G nor its complement, $\overline{G}$, has an induced chordless cycle on five or more vertices (a hole). Figure 2.1 shows an example of a weakly chordal graph, G, and its complement, $\overline{G}$.

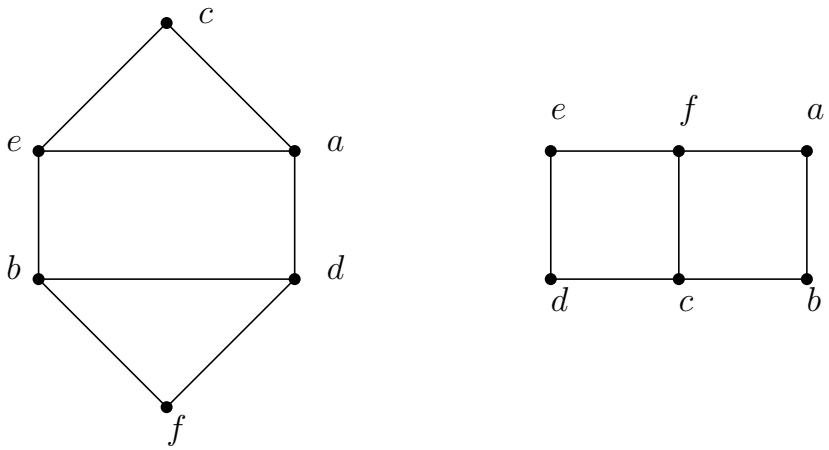


Figure 2.1: *Weakly chordal graph G and its complement $\overline{G}$*

A chord is namely an edge connecting two non-consecutive vertices on the cycle. Every chordal graph is also a weakly chordal graph. G is chordal if it has no induced chordless cycles of size four or more.

However, as Figure 2.2 shows, the complement of a chordal graph G can contain an induced chordless cycle of size four. The complement cannot contain a five cycle though, as the complement of a five cycle is also a five cycle (see Figure 2.3). The

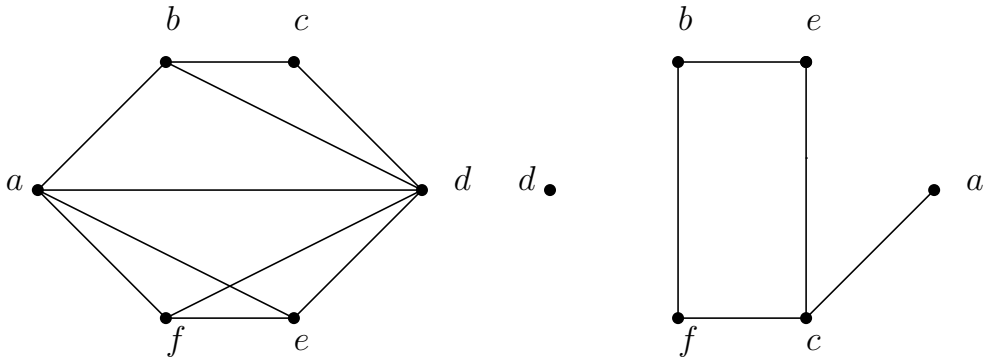
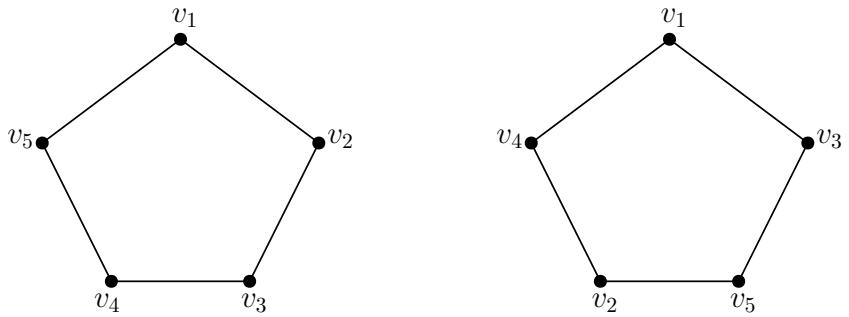


Figure 2.2: *Chordal graph G with it's complement $\overline{G}$ that has a chordless 4-cycle*

above example makes it clear why chordal graphs are also weakly chordal.

Weakly chordal graphs were introduced by Hayward in [13] as a generalization of



(a) G (b) $\overline{G}$

Figure 2.3: *Complement of a five cycle is also a five cycle*

chordal graphs, these graphs form a subclass of the perfect graphs. The authors [21] characterised weakly chordal graphs via the presence of a two-pair, namely a pair of nonadjacent vertices such that every induced path between them has exactly two edges. Their theorem is as follows.

Theorem 2.0.1 *[16] A graph is weakly chordal if and only if each induced subgraph either is a clique or contains a two-pair of the subgraph.*

A number of algorithms on weakly chordal graphs work by repeatedly finding a two-pair $\{x, y\}$ and modifying the neighborhoods of x and/or y. This is the basis of the previously best algorithms for recognizing weakly chordal graphs [24] and for solving a variety of optimization problems on weakly chordal graphs [[24], [16]] (namely, weighted and unweighted versions of maximum clique, minimum

vertex coloring, maximum independent set and minimum clique covering). The algorithm for finding a two-pair in a sparse graph is due to Arikati and Rangan [2] and runs in $O(mn)$ time.

An alternate definition that does not refer to the complement graph is that G does not contain a hole or an antihole, which is the complement of a hole. Berry et al. [6] gave a very different and interesting definition of a weakly chordal graph as one in which every edge is LB-simplicial. They also proposed the open problem of generating a weakly chordal graph from an arbitrary graph. A solution to this problem is the subject of this thesis.

The next sections in this chapter are organised as follows: Section 2.1 Preliminaries outlines the commonly used terms and notations used throughout this book. Section 2.2 is a literature survey on two of the existing algorithms to generate the class of weakly chordal graphs. (Section 2.2.1 covers the first generation algorithm by Arikati and Rangan [2], they use the notion of a two-pair to generate weakly chordal graphs. Section 2.2.2 covers the second existing algorithm to generate weakly chordal graphs is studied and explained. Section 2.2.3 outlines a few experiments conducted on the implementation of these 2 algorithms in order to gain some structural insights about the structure of the weakly chordal graphs produced by the two algorithms.)

2.1 Preliminaries

The following section gives a background details of the terms and notations used subsequently. We will assume that G is a graph on n vertices and m edges, that is, $|V| = n$ and $|E| = m$. The *neighborhood* $N(v)$ of a vertex v is the subset of vertices $\{u \in V \mid (u, v) \in E\}$ of V. The *degree* $\deg(v)$ of a vertex v is equal to $|N(v)|$. A vertex v of G is *simplicial* if the induced subgraph on $N(v)$ is complete (alternately, a *clique*). A *path* in a graph G is a sequence of vertices connected by

edges. We use $P_k(k \geq 3)$ to denote a chordless path, spanning k vertices of G. For instance, a path on 3 vertices is termed as a P_3 and, similarly, a path on 4 vertices is termed as a P_4. If a path starts and ends in the same vertex, the path is a cycle denoted by C_k, where k is the length of the cycle. A *chord* in a cycle is an edge between two non-consecutive vertices in the cycle.

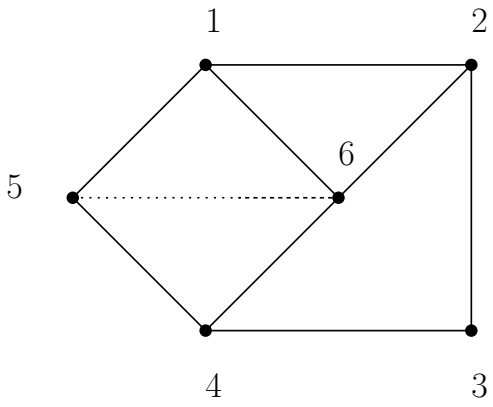


Figure 2.4: {5,6} is a 2-pair

The author [12] characterized weakly chordal graphs via the presence of a two-pair, namely a pair of nonadjacent vertices such that every induced path between them has exactly two edges. Their theorem is as follows. A graph is weakly chordal if and only if each induced subgraph either is a clique or contains a two-pair of the subgraph. A *clique* is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete. A *two-pair* is a pair of vertices $\{u, v\}$ in G, if the only chordless paths between the u and v are of length 2. For example, consider Figure 2.4, the paths that exist between {5,6} are

A vertex is *simplicial* if its neighborhood is a clique. For a set of vertices A, a *confluence point* is a vertex of A that sees all the vertices in $N(A)$.

In Figure 2.5, {a,b} is called as the minimal vertex separator for vertices $\{u, v\}$

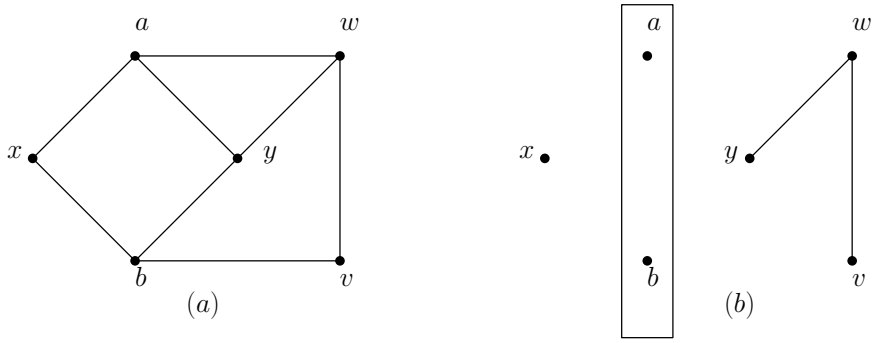


Figure 2.5: {a,b} is a minimal vertex separator for $\{u, v\}$

because it separates vertices into disjoint components. For $X \subseteq V$, $C(X)$ denotes the set of connected components of $G(V - X)$ (connected components are also vertex sets). $S \subset V$ is called a separator if $\|\mathrm{C(S)}\| \geq 2$, an ab-separator if a and b are in different connected components of $C(S)$, a minimal ab-separator if S is an ab-separator and no proper subset of S is an ab-separator, and a *minimal* separator if there is some pair $\{a, b\}$ such that S is a minimal ab-separator.

From [14], an S-saturating is defined as: Given a set S of vertices, an edge e of $G(\mathrm{V} - \mathrm{S})$ is said to be S-saturating if, for each component S_j of $G(S)$, at least one endpoint of e sees all vertices of S_j.

The authors in [7] define an LB-simplicial edge based on the role an edge plays in a weakly triangulated graph. An edge e of E is LB-simplicial if, for each minimal separator S included in the neighborhood of e, e is S-saturating. An edge e is LB-simplicial such that $e \cup N(e) = V$. According to Theorem 2 [7], the set of minimal separators included in the neighborhood of an edge e can be computed in the following fashion: for each component C of $C(e \cup N(e))$, compute $N(C)$.

In the figure 2.6 borrowed from [6] we conduct an LB-simplicality test for the edge bh. First we compute the neighbourhood of edge bh which is equal to $\{d, e, f, g\}$. Next we compute the closed neighbourhood of edge bh which is equal to $\{a, c\}$. The only minimal separator of G included in the neighborhood of bh is

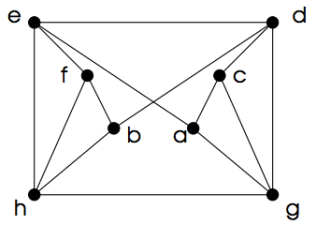


Figure 2.6: LB-Simpliciality test for edge bh

$N(\{a, c\}) = \{d, e, g\}$. Connected components of $G(\{d, e\ g\})$ are $\{d\}$ and $\{e, g\}$. Vertex h sees both vertices in $\{e, g\}$, and b sees d. Hence bh is $\{d, e, g\}$-saturating and thus LB-simplicial. Note that $de \cup N(de) = V$, thus edge de will generate no minimal separator. The total set of minimal separators is $S(G) = \{\{a, d, g\}, \{a, d, h\}, \{b, e, g\}, \{b, e, h\}, \{c, e, g\}, \{d, e, g\}, \{d, e, h\}, \{d, f, h\}\}$.

A *hole* is an induced cycle with five or more vertices and an *antihole* is the complement of a hole. A graph is weakly chordal (also called weakly triangulated) if it contains no holes and no antiholes. The class of weakly chordal graphs, introduced in [12], is a well-studied class of *perfect graphs*. A graph is called *perfect* if the chromatic number and the clique number have the same value for each of its induced subgraphs. The *chromatic number* of a graph G is the smallest number of colors needed to color the vertices of G so that no two adjacent vertices share the same color. Chromatic number gives information about how connected is the graph.

2.2 Literature review of existing algorithms

2.2.1 Two-pair method

To review the existing work done in the area of generation algorithms, this section summarizes an existing algorithm for generating weakly chordal graphs. This

method was proposed by the authors [2], this method uses the notion of a *two-pair* in a graph. A pair of vertices u, v in a graph G is termed as a two-pair if the only chordless paths between the u and v are of length 2.

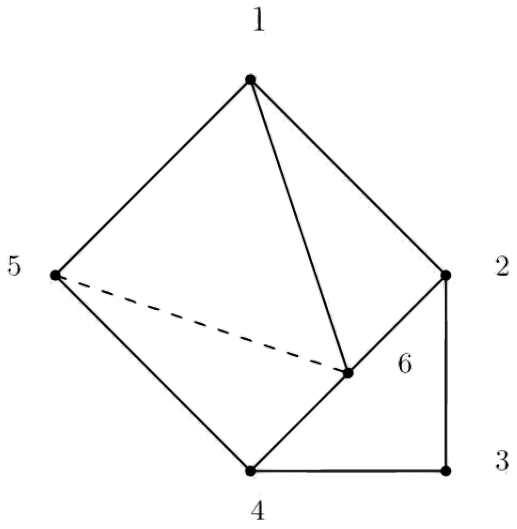


Figure 2.7: Two-pair in a graph

In the figure 2.7, vertex 5 and vertex 6 make a two-pair as the only chordless paths between them are of length 2 i.e. {5,4,6}, {5,1,6}. A weakly chordal graph that is not a *clique* has a two-pair [16]. A *clique* is a subset of vertices of an undirected graph G such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete. The generation algorithm proposed by the author starts by supposing, let $\{u, v\}$ be a two-pair in an arbitrary graph G. Then $G + \{u, v\}$ is weakly chordal iff G is weakly chordal [24]. The algorithm to generate a weakly chordal graph on n vertices, starts by making a tree which satisfies the definition of weak chordality and then repeatedly finds a two-pair $\{u, v\}$ and add to G the edge joining u to v. To find a two-pair we can use an $O(mn)$ algorithm proposed in this method by the authors, Arikati and Rangan [2].

2.2.1.1 Overview of Algorithm

Let $\{u; v\}$ be a two-pair in an arbitrary graph G. Then $G + \{u; v\}$ is weakly chordal iff G is weakly chordal [24]. We can then generate a weakly chordal graph

on n vertices by starting with a tree (as the complements of a five-cycle is also a five-cycle) and repeatedly find a two-pair $\{u; v\}$ and add to G the edge joining u to v. To find a two-pair we can use an $O(mn)$ algorithm proposed in this method by the authors, Arikati and Rangan [2].

This algorithm to find a two-pair [2] takes an input of any graph G that has V vertices and E edges, for which it outputs a pair of two vertices $\{u, v\}$ that make a two-pair as defined within the input graph G. It begins by choosing a random vertex out of all the present vertices in the given input graph, next, we compute the neighbourhood of the chosen vertex v to be $N(v)$. Let us assume that v is a fixed vertex in G and $N(v)$ =$\{u|u$ is adjacent to $v\}$. Next, we remove from the input graph G, the chosen vertex v and its neighbourhood $N(v)$ to compute the connected components present in graph $H_1, ..., H_k$ (after removal of v and $N(v)$). Let the connected components of H= G-$\{v\}$-$N(v)$ be $H_1, ..., H_k$. The algorithm proceeds by identifying a vertex in H that can form a two-pair with v. $N_i(v)$= $\{u|u \in N(v)$ and u is adjacent to some vertex in $H_i\}$.

Lemma 1 [2] : Let u be any vertex in some connected component, say H_i of H. Then, (u, v) is a two-pair if and only if u is adjacent to every vertex of $N_i(v)$.

Hence, for every vertex $u \in H$, first define $label(u)$ to be the index such that $u \in H\_label(u)$. Then, for every vertex $u \in H$, define $NV(u) = \{w|w$ adjacent to u and $w \in N(v)\}$. Lemma 2 from [2] states, "A pair (u, v) of vertices, where $u \in H$, form a two-pair if and only if $|N\_label(u)(v)| = |NV(u)|$. Let G be an arbitrary graph with two-pair $\{u; v\}$. Then the graph obtained by adding the edge $\{u; v\}$, G' is weakly triangulated if and only if G is weakly triangulated [15].

This algorithm is rewritten from [2].

Algorithm 2.1 Two-Pair

Input: A graph $G = (V, E)$, and adjacency lists denoted by $N(v)$, $v \in V$.
Output: A two-pair, if it exists

1: **for** all $v \in V$, **do** compute the connected components H_1, ..., H_k for $H = G$ - $\{v\}$ - $N(v)$;
2: **for** all $x \in N(v)$ **do** label$(x) = 0$;
3: **for** all $u \in H$ **do** compute label(u)
4: **for** all $u \in H$ **do** compute $|\mathrm{NV}(u)|$
5: **for** i=1 **to** k **do**
6: $N_i(V) = \emptyset$;
7: **for** all $x \in \mathrm{N}(v)$ **do**
8: **for** all $u \in \mathrm{N}(x)$ **do**
9: **if** $label(u) \neq 0$, **then** $\mathrm{N}_{\mathrm{label(u)}}(\mathrm{v}) := \mathrm{N}_{\mathrm{label(u)}}(\mathrm{v}) \cup \{x\}$
10: **end for**
11: **end for**
12: **end for**
13: **for** all $u \in H$ **do** compute $|N_{\mathrm{label(u)}}(\mathrm{v})|$;
14: **for** all u $\in H$ **do if**($|N_{\mathrm{label(u)}}(\mathrm{v})| = |NV(u)|$) **then** declare (u,v) is a two-pair and STOP;

2.2.1.2 Example

Consider the following example in figure 2.8 below to implement algorithm to find a two-pair in the given graph.

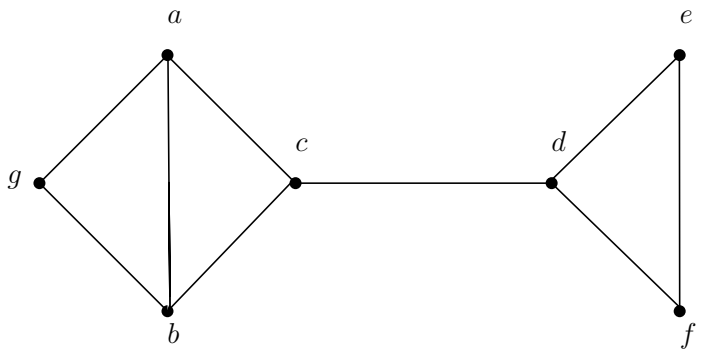


Figure 2.8: Example to find two-pair

Let, H = G - $\{v\}$ - $N(v)$ where v is the chosen vertex to find a two-pair with, $N(v)$ is the neighbourhood of the vertex v. We begin by randomly choosing any vertex as v. Let chosen vertex v be c from the figure 2.8. Hence the $N(v = c)$

= $\{a, b, d\}$. After substituting, the equation becomes, $H = G$ - $\{c\}$ - $\{a, b, d\}$. Next, we compute the connected components left in the graph G after $H = G$ - $\{c\}$ - $\{a, b, d\}$. After removal of vertex c and it's neighbours $\{a, b, d\}$, the graph G is broken into 2 connected components. These two components are $H_1 = \{g\}$ and $H_2 = \{e, f\}$. Next, we assign a Label = 0 to $\{v = c\}$ and $N(v = c)$ i.e. $\{c, a, b, d\}$ all get assigned a Label = 0. i.e. Label(a)=0, Label(b)=0, Label(c)=0, Label(d)=0. Next, we compute $NV(u)$: u is adjacent to some v in H, u is subset of vertices in neighbours $NV(u = g) = \{a, b\}$, $NV(u = f) = \{d\}$, $NV(u = e) = \{d\}$. Next, we compute $N_iV(u)$: label 0 neighbours $N_1V(u = \{g\}) = \{a, b\}$, $N_2V(u = \{e, f\}) = \{d\}$. The algorithm identifies a vertex in one of the components H_i,...,H_k that can form two-pair with v.

| $N_1V(u) = \{a,b\}| = |NV(u=g) = \{a,b\}|$

Hence, $\{c, g\}$ is a two-pair in figure 2.8.

In order to generate a weakly chordal graph using the method of two pairs, first an input of N vertices and E edges is given. It begins by generating an initial tree layout is generated. Since trees are weakly chordal graph, in the first phase it generates a tree with at least N vertices. In the next phase it calculates the number of edges e in the initial tree layout. To match the expected number of input edges E, subtract e from E (say m).

2.2.1.3 Results

After implementation of this algorithm in Python below are a few examples attached as results. For every example there are two graphs, first graph shows the initial layout showing the four cycles generated on at least V input vertices.n the next phase it calculates the number of edges e in the initial tree layout. To match the expected number of input edges E, subtract e from E (say m). Now the algorithm to find runs m times to find m two-pairs that are joined by an edge in every iteration maintaining weak chordality.

This example takes an input of 6 vertices and 12 edges. It first produces an initial layout 2.9 and the second figure 2.10 is the final graph on 6 vertices and 12 edges.

The two-pairs added are:

Two-pair 1- {10,8} Two-pair 2- {7,10} Two-pair 3- {8,7} Two-pair 4- {10,6} Two-pair 5- {9,11} Two-pair 6- {9,6}

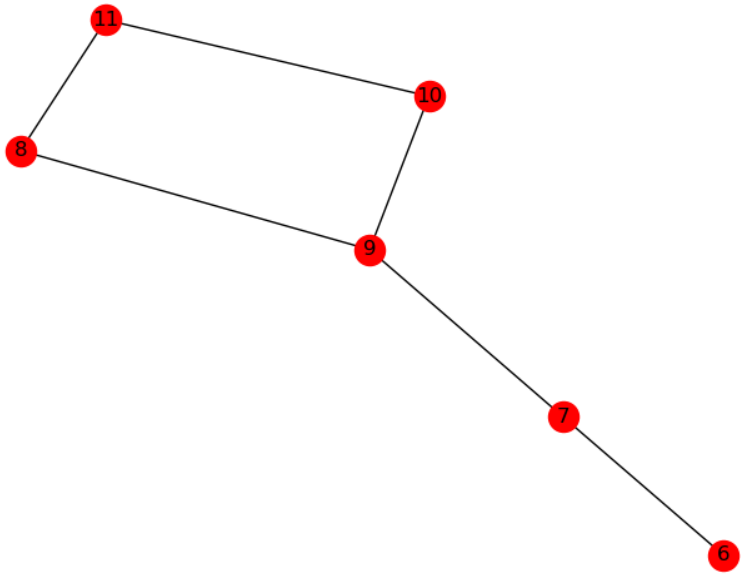


Figure 2.9: Two-pair Example 1.1 Initial Graph on 6 Vertices 6 Edges

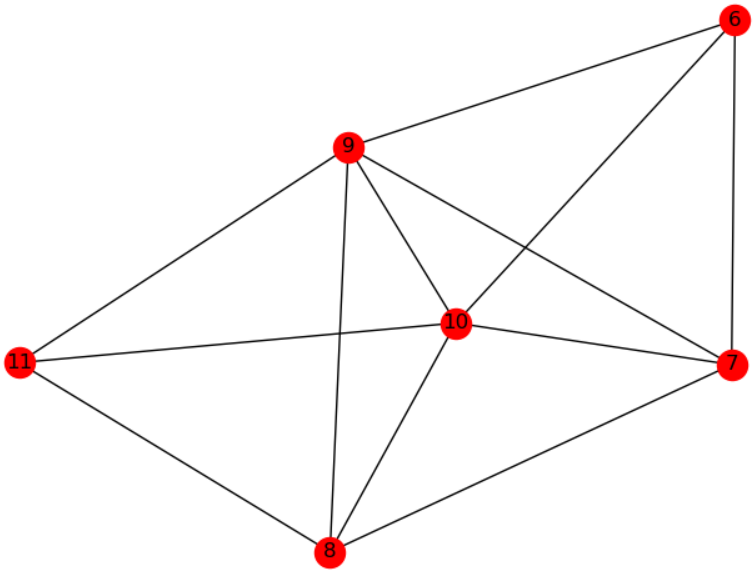


Figure 2.10: Two-pair Example 1.2 Final Weakly Chordal Graph for 6 Vertices 12 Edges

This example takes an input of 8 vertices and 15 edges. It first produces an initial layout 2.11 and the second figure 2.12 is the final graph on 8 vertices and 15 edges.

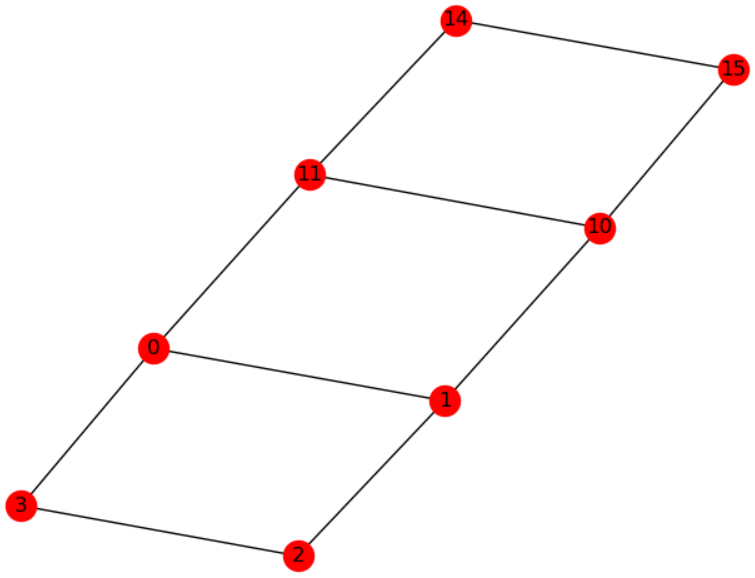


Figure 2.11: Two-pair example 2.1 Initial Graph on 8 Vertices 10 Edges

Two-pair 1- {11,1} Two-pair 2- {10,0} Two-pair 3- {14,10} Two-pair 4- {1,3} Two-pair 5- {14,1}

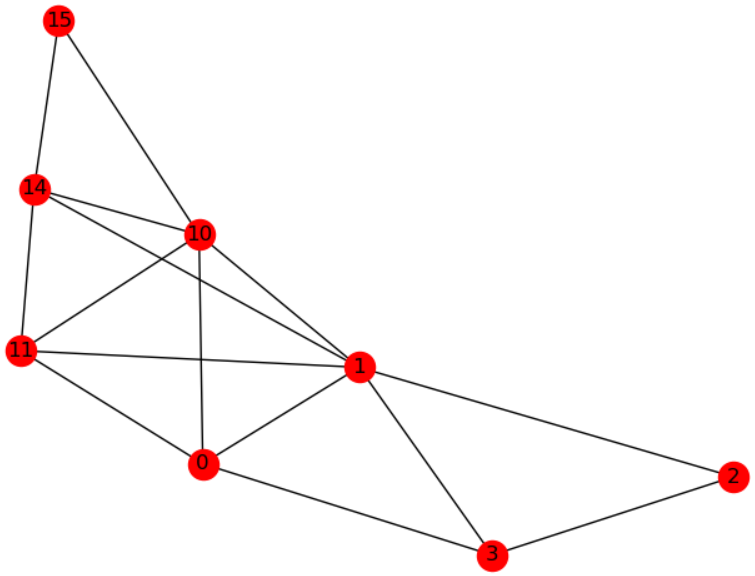


Figure 2.12: Two-pair example 2.2 Final Weakly Chordal Graph on 8 Vertices 15 Edges

This example takes an input of 10 vertices and 17 edges. It first produces an initial layout 2.13 and the second figure 2.14 is the final graph on 10 vertices and 17 edges.

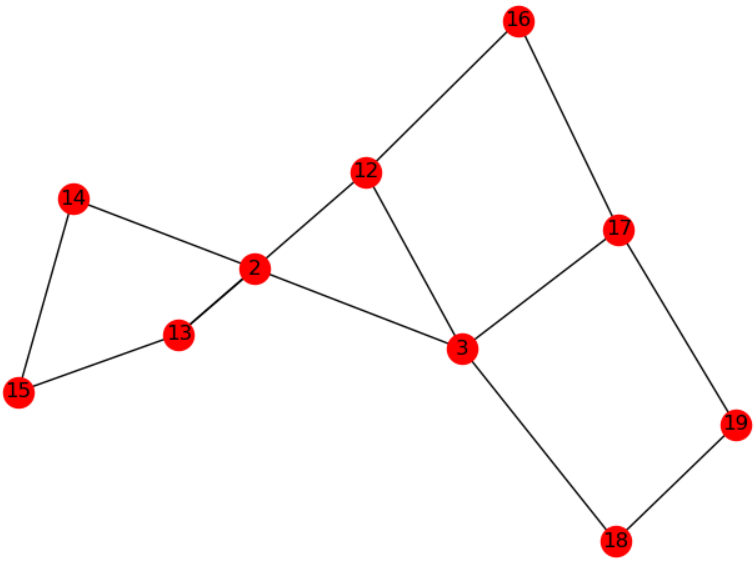


Figure 2.13: Two-pair example 3.1 Initial Graph on 10 Vertices 13 Edges

Two-pair 1- {3,16} Two-pair 2- {18,17} Two-pair 3- {19,3} Two-pair 4- {13,3}

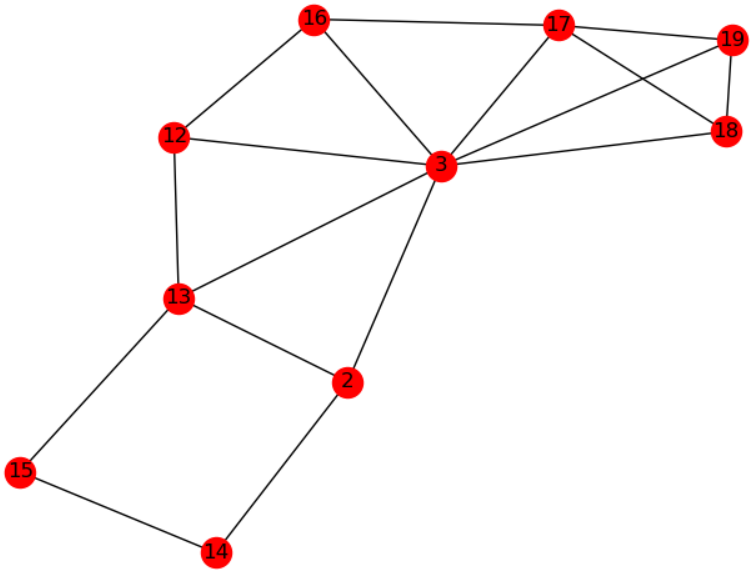


Figure 2.14: Two-pair example 3.2 Final Weakly Chordal Graph on 10 Vertices 17 Edges

2.2.1.4 Complexity

The tree generation in the initial phase can be constructed in time $O(n)$, where n is the number of nodes in a tree. For n nodes in the tree, we insert $3n+1$ edges in the layout and each edge insertion can be done in a constant time. So the initial tree layout can be generated in $O(n)$ time.

The algorithm two-pair by [2] finds a two-pair if it exists, in a given input graph G=(V, E), in time $O(nm)$. Hence, the overall complexity is $O(nm^2)$.

2.2.2 Separator-based method

This paper by the authors [20] outlines a proposed algorithm to generate weakly chordal graphs on a given set of input of n vertices and m edges. The base a graph G to be generated based on the concept of a separator based method that generalizes owing to Markenzon [19] for generating chordal graphs and inturn allows them to be able to exploit structural properties of a weakly chordal graph. The authors [20] in this method to generate weakly chordal graph, first begin by constructing a tree and then generate an orthogonal layout which is also a weakly

chordal graph on n vertices. Then authors [20] insert additional edges, as needed, for a total of m edges to match the m input. Their algorithm ensures that the graph remains weakly chordal after each edge is being added or inserted. The time complexity of an insertion query is $O(n^3)$ and an insertion takes constant time. The main advantages of this method are that it uses very simple data structures and exploits the basic structural properties of a weakly chordal graph. Another advantage of this generation scheme is that this algorithm is able to join an edge between two non-two pair vertices maintaining weak chordality property.

2.2.2.1 Overview of Algorithm

The inputs to this algorithm are the number of vertices, n, and the number of edges, m, of a weakly chordal graph to be generated. The algorithms is divided into three phases, it begins with Phase 1 comprising of trees as trees are weakly chordal graphs. In the first phase, the algorithm begins by generating a tree with at least n vertices. In the second phase, the algorithm uses the tree to generate an orthogonal layout. An orthogonal layout is also a weakly chordal graph on at least n vertices. The orthogonal layout is made up of 4-cycles and edges incident on the vertices of these 4-cycles. In the third and the final phase, vertices are removed from the graph if the count exceeds n and additional edges are introduced to bring up the edge tally to m, by maintaining weak chordality. Inserting an edge between two vertices u and v so that weak chordality is preserved requires meticulous consideration. Let $I_{\mathrm{u,v}}$ be the set of common neighbors of u and v. If $I_{\mathrm{u,v}} = \emptyset$, the alogrithm needs to check whether the removal of $\mathrm{I}_{u,v}$ separates u and v, that is put them in different components of $G[V$ - $I_{\mathrm{u,v}}]$. To check for this a breadth-first-search in $G[V$ - $I_{\mathrm{u,v}}]$ is done, starting at u to see if v is reachable. This search can be done in the induced graph of a reduced set of vertices called $AuxNodes$. If v is not reachable from u, the algorithm inserts an edge between u and v, else we search for shortest paths between u and v. We do not insert the edge $\{u, v\}$ if the length of a shortest path is greater than 3. Otherwise, the

algorithm has to check for other conditions, such as single or multiple shortest paths, forbidden configurations, alternate longer paths between u and v to decide whether the insertion of $\{u; v\}$ preserves weak chordality. If $\{u,v\} = \emptyset$, it proceeds in the same way as when $I_{\{u,v\}}$ does not separate u and v. The only difference is that here it has to consider the entire graph to search for shortest paths between u and v but the other details remain essentially the same.

2.2.2.2 Phase 1: Generation of Tree

The first phase begins by generating a tree, T, on $\frac{n}{2}$ nodes is generated such that each node has degree of at most four. Starting with a single node, the algorithm add new ones either by splitting an edge into two or joining a new node to an existing node, chosen at random. After $\frac{n}{2}$ nodes have been added, the algorithm traverses the tree to check if a pair of degree-4 nodes or a degree-3 node and a degree-4 node are adjacent. Each such pair is separated by inserting a new node adjacent to both. Let $k \geq \frac{n}{2}$ be the number of nodes in the resulting tree, T'.

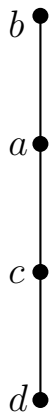
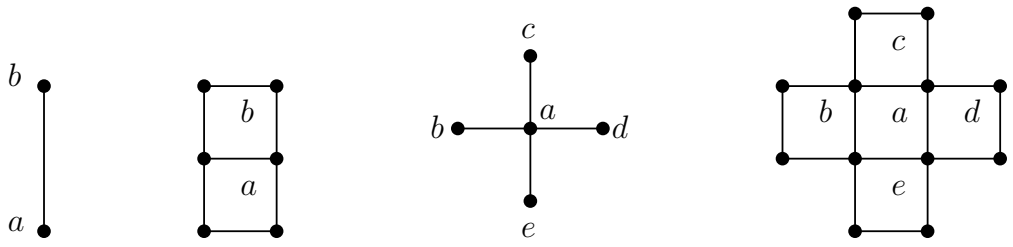


Figure 2.15: Tree generation (figure borrowed from [20])

2.2.2.3 Phase 2: Generation of Initial Layout

In the second phase, the algorithm proceeds by generate an orthogonal layout that corresponds to T' in the following way. For each node in T', the algorithm creates a 4-cycle. By keeping 4-cycles in the initial layout, the authors in [20] are trying

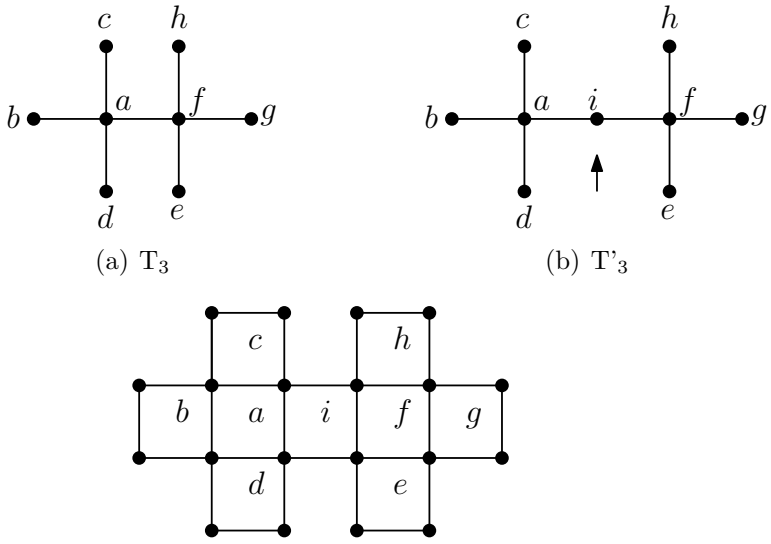
to ensure that the algorithm generates proper weakly chordal graphs, and not just chordal graphs. Two 4-cycles have an edge in common iff the corresponding tree nodes are adjacent. Figure 2.16(a) shows a tree T_1 with two nodes, a and b. The corresponding layout is shown in figure 2.16(b). It has two 4-cycles, corresponding to each of the nodes of T'; these share an edge in common as the tree nodes a and b are adjacent. The authors defined this as the initial layout. For the example tree of figure 2.16(c), the corresponding initial layout is shown in figure 2.16(d).



(a) *Tree* T_1 (b) *Initial layout* G_1 (c) T_2 (d) *Initial layout* G_2

Figure 2.16: *Tree to layout of four-cycles (figures borrowed from [20])*

After the tree is generated, if there exists any two degree-4 nodes or a degree-3 node and a degree-4 node that are adjacent, a new node is inserted between them to separate them (Figures 2.17(a), 2.17(b), 2.17(c)). This is essential because otherwise, the orthogonal layout will force two 4-cycles that do not correspond to adjacent tree nodes to share an edge. If the resulting orthogonal layout has more than n vertices, enough vertices are deleted from the 4-cycles to bring the count down to n. The orthogonal layout has $2 * k + 2 > n$ vertices where $k \geq (\frac{n}{2})$ is the number of vertices in the tree generated in Phase 1. Vertices that make as a candidates for deletion are the ones that have degree equal to two. Once the vertex-deletion is completed, if the number of edges m' in the resulting layout is m or more, the algorithm stops and returns the layout which is also a weakly chordal graph, as the output. Otherwise, the algorithm proceeds to the next phase.



(a) T_3 (b) T'_3 (c) *Initial layout* G_3

Figure 2.17: *Tree to layout of four-cycles (figures borrowed from [20])*

2.2.2.4 Phase 3: Generation of Weakly Chordal Graph

In the third and final phase, $(m - m')$ additional edges are required to be inserted into the initial layout while weakly chordality is preserved. Two cases can arise, according to the value of $I_{u,v}$. Either $I_{u,v} \neq \emptyset$; or $I_{u,v} = \emptyset$;

Case 1: $I_{u,v} \neq \emptyset$ or $I_{u,v}$ is non empty

Since the value of $I_{u,v}$ is non-empty, the algorithm needs to check whether the removal of $I_{u,v}$ separates u and v. This is achieved by checking for the existence of a path from u to v by using a breadth-first search in the induced graph $G[V - I_{u,v}]$. To make this breadth first search more efficient we perform this search in a smaller set than $V - I_{u,v}$. This set is called as *AuxNodes*. For the class of chordal graphs [19] defined this set to be $N(x) - I_{u,v}$, where x is any vertex in $I_{u,v}$. They define AuxNodes $= N(I_{u,v}) \cup N(N(I_{u,v}) \cup \{I_{u,v}\})$, which as an extended neighborhood of $I_{u,v}$.

Two subcases arrive (a) Exactly one P_4 or (b) More than one P_4 connects u to v.

Case 1.1: Exactly one P_4

Let SP be the set of vertices of the unique $P_4 = u{-}x{-}y{-}v$, where x and y are

internal vertices. Just as in our search for P_4-paths relative to $I_{u,v}$ we need to define the set AuxNodes for shortest paths relative to P_4. Shortest paths relative to P_4 can have vertices from the sets SP, $N(SP)$ and $N(N(SP))$. Thus we set AuxNodes in three different ways by removing either one of the internal vertices or both the

Case 1.2: More than one P_4 connects u to v

When multiple P_4's exist, we need to check for a forbidden configuration formed by a pair of P_4's as shown in Figure 2.18(a). Inserting an edge $\{u, v\}$ into this configuration does not create a chordless cycle of size five or more in G, but it creates a chordless six cycle in G as can be seen from the complement of the configuration in Figure 2.18(b). Since a graph G is weakly chordal if neither G nor its complement $\overline{G}$ contains a chordless cycle of size 5 or more, such an insertion is not permitted.

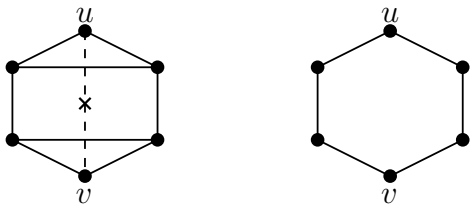


Figure 2.18: *Forbidden configuration formed by P_4's from [20])*

Having checked for forbidden configurations, the next step is to check if a chordless path longer than a P_4 exists between u and v. Let $P^1{}_4$, $P^2{}_4$,..., $P^k{}_4$ be k ($\geq$ 2) P4's from u to v, where$P^i{}_4$ = u - x_i - y_i - v and x_i, y_i are its internal vertices. Define all $SP = \{u, x_1, x_2,..., x_i, y_1, y_2,...,y_j, v\}$ to be the set of vertices on all the P_4's between u and v. As in case 1.2, the search for a chordless path longer than a P_4 can be restricted to the set of vertices $N(allSP) \cup N(N(allSP)$ $\cup$ allSP).

Case 2: $I_{u,v}$ is empty

In this case, there are no common neighbors of u and v but a path exists between u and v. This case can be solved in a similar way as for case 1.2. Here the AuxGraph is the entire graph because $I_{\mathrm{u,v}}$ is empty. If any of the paths is greater than P_4, we do not insert $\{u, v\}$ and choose another pair of vertices. Otherwise, we use the same algorithms as for case 1.2 to check if the addition of $\{u, v\}$ keeps the graph weakly chordal or not. The corresponding cases are referred to as case 2.1 and case 2.2.

2.2.2.5 Example

Consider generating a weakly chordal graph with $n = 8$ vertices and $m = 12$ edges. Figure 2.19 shows a tree on $\frac{n}{2}$ nodes. There are four nodes in the tree and on expanding each node to a 4-cycle, we get the orthogonal initial layout of Figure 2.19(a)) with $n = 10$ vertices. Then we removed two vertices 8 and 9 from the initial layout and we get a layout with $n = 8$ vertices and m' $= 10$ edges as shown in Figure 2.19(c)). Need to insert (m-m') $= 2$ more edges into this initial layout to generate a weakly chordal graph with the requisite number of vertices and edges.

Say we want to insert edge between the vertices 3 and 4. Since $I_{3,4}$ is non-empty we have Case 1. As the removal of $I_{3,4}$ leaves the vertices 3 and 4 in two different components, we can safely insert an edge between vertices 3 and 4 as per case 1.0 as shown in Figure 2.19(d). Next, we try to insert edge $\{3, 6\}$. The insertion of this edge corresponds to case 1.1 because the removal of their common neighbor, viz., $\{5\}$ does not separate 3 and 6. Now, $N(5)$ is $\{3, 4, 6\}$ and $N(N(I_{3,6}) \cup I_{3,6})$ is $\{0, 2, 7\}$; and therefore AuxNodes $= \{0, 2, 3, 4, 6, 7\}$. In G[AuxNodes] we search for paths from 3 to 6. There is a single shortest path $SP = \{3, 4, 7, 6\}$ and this corresponds to Case 1.1. Since $N(SP) = \{0, 2\}$ is not an empty set, we need to compute $\{N(N(SP) \cup SP\}$ which is empty. Thus AuxNodes $= N(SP)$ $\cup$ $N(N(SP) \cup SP)$ $SP = \{0, 2, 3, 4, 6, 7\}$ vertices. We create different induced graphs on G[AuxNodes] by removing both the internal vertices from SP or exactly

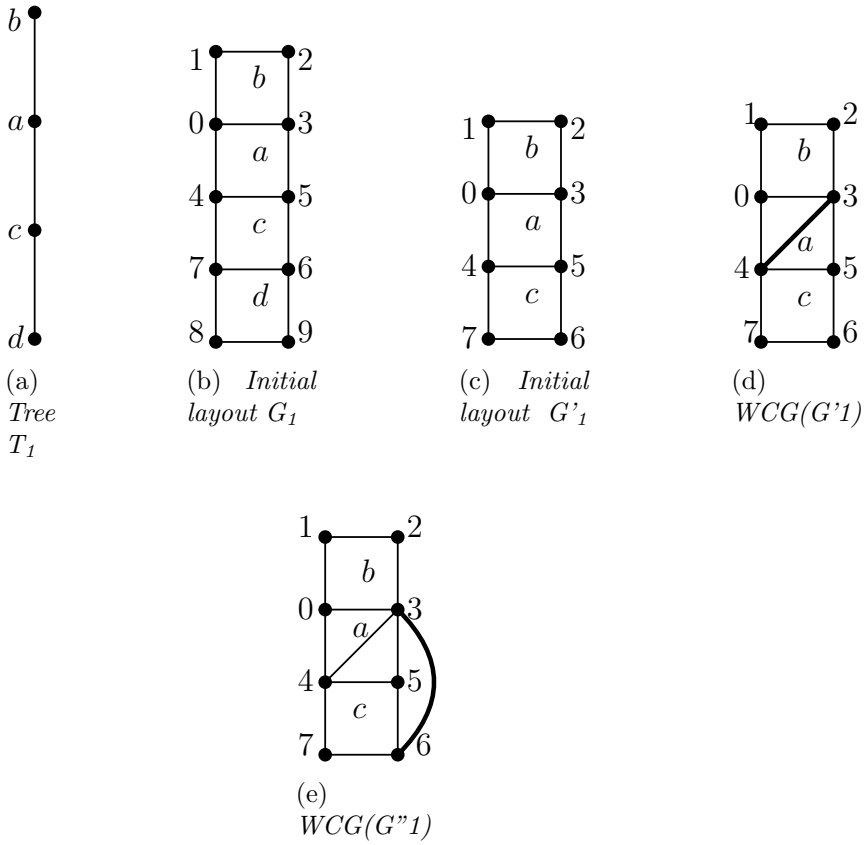


Figure 2.19: *Tree to weakly chrodal graph (figures borrowed from [20])*

one of them and observed that there is no chordless path between 3 and 6. Hence, we can insert an edge between 3 and 6.

2.2.2.6 Results

This example takes an input of 6 vertices and 12 edges. It first produces an initial layout 2.20 on 6 vertices and 7 edges then it produces the second figure 2.21 which is the final graph on 6 vertices and 12 edges.

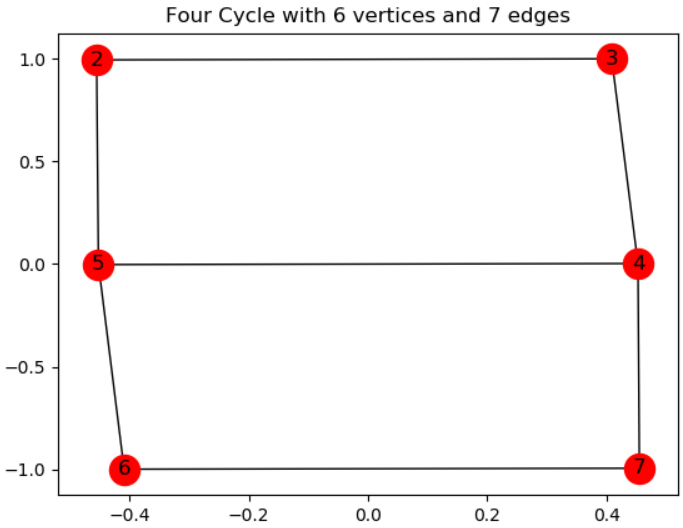


Figure 2.20: Initial graph for 6 vertices 7 edges

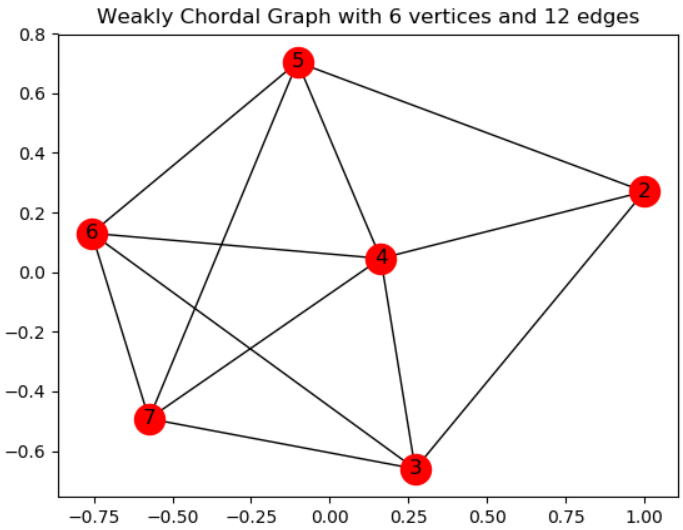


Figure 2.21: Final weakly chordal graph on 6 vertices 12 edges

This example takes an input of 8 vertices and 15 edges. It first produces an initial layout 2.22 and the second figure 2.23 is the final graph on 8 vertices and 15 edges.

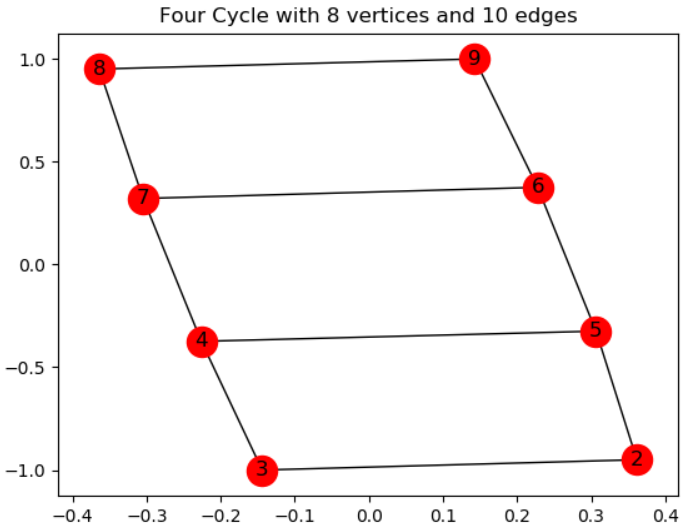


Figure 2.22: Initial graph for 8 vertices 10 edges

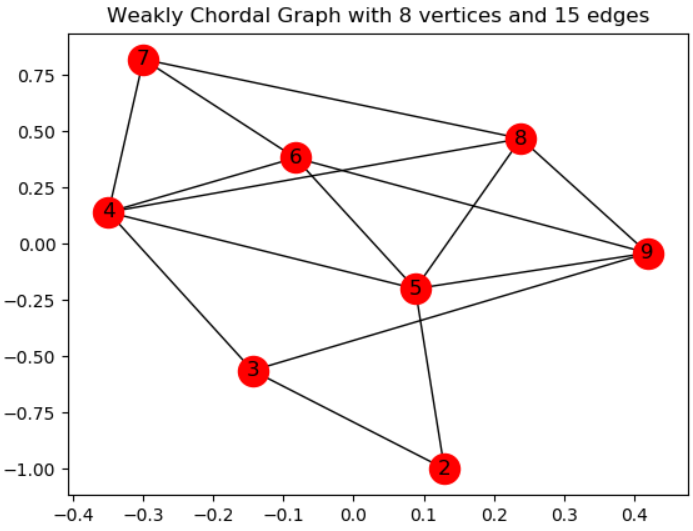


Figure 2.23: Final weakly chordal graph on 8 vertices 15 edges

This example takes an input of 10 vertices and 17 edges. It first produces an initial layout 2.24 and the second figure 2.25 is the final graph on 10 vertices and 17 edges.

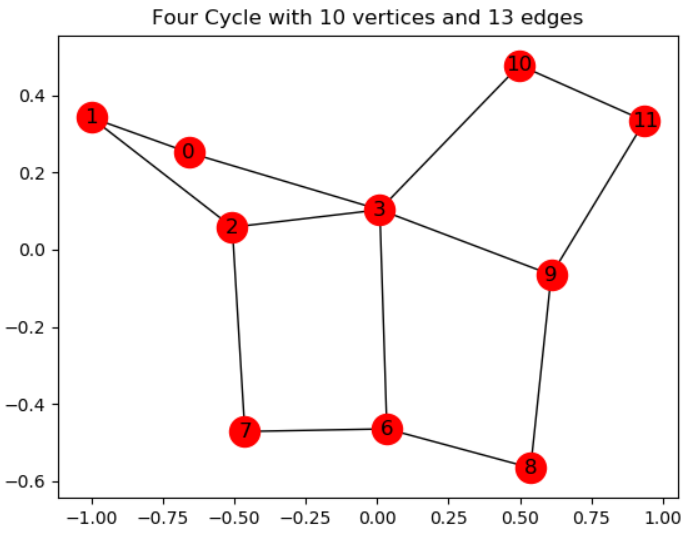


Figure 2.24: Initial graph for 10 vertices 13 edges

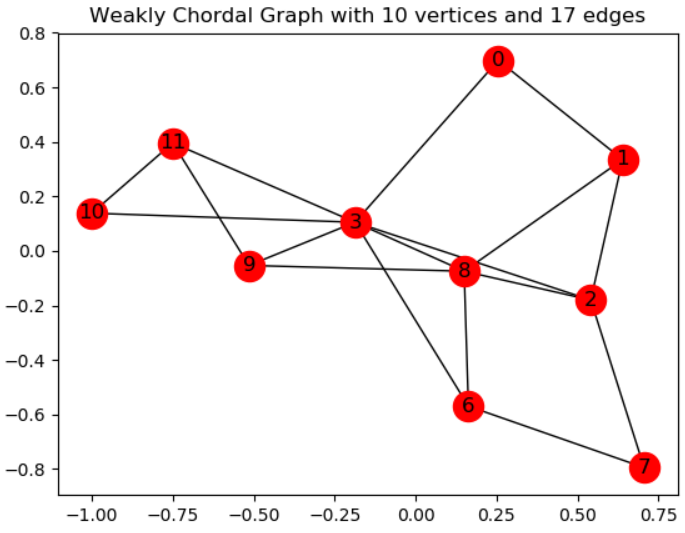


Figure 2.25: Final weakly chordal graph on 10 vertices 17 edges

2.2.2.7 Complexity

The tree generation in the initial phase can be constructed in time $O(n)$, where n is the number of nodes in a tree. For n nodes in the tree, we insert $3n+1$ edges in the layout and each edge insertion can be done in a constant time. So the initial tree layout can be generated in $O(n)$ time. In the third phase, a pair of vertices $\{u, v\}$ is chosen at random to insert an edge between them. Two types of failures may arise. One is that the pair of vertices $\{u, v\}$ corresponds to an existing edge and the other is that the addition of $\{u, v\}$ violates weak chordality property. To avoid the first type of failure one can either maintain a list of edges belonging to the complement graph or we can check the existence of an edge in constant time by maintaining the adjacency matrix representation of the current graph.

To bound the query complexity of adding an edge $\{u, v\}$ to the existing weakly chordal graph, we note that this is dominated by the case when there are multiple P_4's between u and v and we have to consider these in pairs and run breadth-first search on the entire graph (the case when $I_{\mathrm{u,v}} = \emptyset$ in Algorithm 3). An upper bound on the number of pairs P_4's can be estimated this way.

Assume G has n vertices. Let $\{v_1, v_2, \ldots, v_{\mathrm{l}}\}$ be the set of vertices adjacent to v that lie on the P_4 shortest paths between u and v. If d_{v} is the degree of v then $l < d_{\mathrm{v}}$. Likewise, let $\{v'_1, v'_2, v'_3, v'_4, \ldots, v'_{\mathrm{k}}\}$ be the set of vertices adjacent to u that lie on these shortest paths. Again $k \leq d_{\mathrm{u}}$, where d_{u} is the degree of u. Thus $d_{\mathrm{u}}d_{\mathrm{v}}$ is an upper bound of the number of P_4 paths between u and v.

Let d_{i} be the degree of v_{i} relative to the vertices $\{v'_1, v'_2, v'_3, v'_4, \ldots, v'_{\mathrm{k}}\}$ for $i = 1, \ldots, l$. Then an upper bound on the number of pairs of edge-disjoint P_4 paths between u and v is given by $PathCount = \sum_{i \neq j} d_{\mathrm{i}}\, d_{\mathrm{j}}$. Let, $\sum d_{\mathrm{i}} = t$. Now it follows from the equality $2l \sum d_{\mathrm{i}}\, d_{\mathrm{j}} = (l-1)\,(\sum d_i)^2 - \sum_{i \neq j} (\mathrm{d}_i - d_j)^2$ that $PathCount$ is maximum when all d_{i}'s are equal. Therefore an upperbound on $PathCount$ is $l^2\,(\frac{t}{l})^2 = \mathrm{t}^2$. Since $t = O(lk)$, we have $PathCount = \mathrm{O}(\mathrm{d}_u^2 d_v^2)$.

2.2.3 Comparative Studies

In this section we carry out experiments on the weakly chordal graphs generated by the two generation algorithms studied above. We provide same input i.e. same number of V vertices and E edges to yield weakly chordal graphs generated on same input by two different generation algorithm. The underlying motivation to conduct this comparative study is to be able to draw structural information about how different are these weakly chordal graphs generated on same input by two different algorithms. First in table 2.1 we compute the number of non two-pairs joined by an edge in the second algorithm to see presence of long chordless paths within the graphs. Next, we compare the outputs based on the chormatic number in table 2.2, in table 2.4 and 2.3 we compute the number of 3 cycles and number of chordless 4 cycles for each of the weakly chordal generated. Lastly, we compare the run times on same input graphs for two different generation methods for WCG in table 2.5, In the following sections, each parameters comparison is discussed.

2.2.3.1 Non Two-pairs

The first generation algorithm studied is based on joining only two-pairs [2], and the second generation algorithm is based on separator [20] which is also able to join non-two pairs along with two-pairs. Hence we decided to compute the number of non-two pair edges joined by the separator based generation algorithm for weakly chordal graphs to see how many non two-pair edges comprise the final graph. This experiment was conducted with the motivation to be able to mark the existence of long chordless paths. The column for non 2-pair edges added depicts the existense of long chordless paths within the graphs in Table 2.1. As shown, the results of the experiment are documented below in the Table 2.1 that outline the input total number of vertices and edges in the first column, edges added in initial layout in second column, non 2-pair edges added in third column and 2-pair edges added in final column.

{Vertices, Edges}	Intital layout edges	Non 2-pair edges	2-pair edges
{50,100}	72	20	8
{50,300}	71	156	73
{50,500}	70	210	220
{50,1000}	73	514	413
{100,200}	142	28	30
{100,500}	146	108	248
{100,1000}	145	359	496
{150,300}	218	58	24
{150,500}	222	185	93
{200,500}	294	144	62
{200,1000}	297	234	469

Table 2.1: Number of non two-pairs added in separator based method

2.2.3.2 Chromatic Number

The chromatic number of a graph G is the smallest number of colors needed to color the vertices of G so that no two adjacent vertices share the same color i.e., the smallest value of k possible to obtain a k-coloring. Empty graphs have chromatic number 1. For a graph G =(V, E) let the chromatic number be k. Then the ratio $\frac{k}{n}$ is related to the average path length (n=|1|). We compare $\frac{k}{n}$ with $\frac{n}{n}$, as for a complete graph, the average path length is 1, $\frac{k}{n} < \frac{n}{n}$. For a completely disjoint graph ratio is $\frac{1}{n}$, in this case the average path length is infinity.

The motivation to chose this characteristic of chromatic number to compare the two outputs yielded was to be able to see the sparsity of the graph and to be able to find a pattern between how connected are the graphs generated. The Table 2.2 below mentions the edges taken by the two generation methods to reach the target ratio between chormatic number (k) ratio and the number of vertices(n) for the weakly chordal graphs by inputs to both the algorithms. The ratio column depicts the target ratio (chormatic number (k) / number of vertices (v)) and the following columns represent the number of edges added on a graph with 50 vertices by each of the generation algorithms to reach that given ratio.

Ratio (k/50)	Number of Edges	Number of Edges	Number of Edges	Number of Edges
	(Two-Pair Based)		(Separator Based)	
0.08	80	95	51	75
0.10	100	130	80	100
0.14	155	190	130	180
0.18	310	430	245	200
0.20	450	475	265	295
0.22	510	540	320	355
0.24	570	585	380	420
0.30	620	635	555	570
0.40	710	735	685	705

Table 2.2: Number of edges added for a weakly chordal graph on 50 vertices by two-Pair method and separator based method for generating WCG to attain a fixed ratio of chromatic number (k) to number of vertices (n=50)

2.2.3.3 Number Of Cycles

In this section we summarize the experiment conducted on two of the generation algorithms to generate weakly chordal graphs. The first generation algorithm studied is based on joining two-pairs by the authors [2], and the second generation algorithm is based on separator by the authors [20]. The underlying motive to conduct this experiment was to be able to assess the structure of the graph. We compute the number of 3 cycles existing within an input graph as well as we compute the number of chordless 4 cycles. For the class of weakly chordal graphs, chordless four cycles are the only chordless cycles allowed, hence it was important to consider this as a factor to generalize about the structure of the graph. We computed the number of chordless four cycles in every weakly chordal graph from both the algorithms implemented. We then computed the number of three cycles within these graphs, in order to be able to gain information if the graphs being generated are more chordal than the other. The results of the experiments are documented in the Table 2.3 and Table 2.4 below showing the ratio of number of chordless four cycles to the number of three cycles for the two methods. For an input of given vertices and edges we generate different number of graphs and report the number of cycles for respective graphs. We then compute the average

of these cycles and depict the average ratio of number of 3 cycles to number of chordless 4 cycles.

Number of Runs	Input {10,15}	Input {10,20}	Input {20,50}	Input {20,60}	Input {30,60}	Input {40,60}	Input {40,80}
	(number of 3 cycles, number of chordless 4 cycles)						
1	(4,2)	(14,0)	(42,5)	(97,5)	(35,5)	(6,15)	(42,6)
2	(6,0)	(14,1)	(42,6)	(66,10)	(33,8)	(12,9)	(43,8)
3	(5,1)	(13,1)	(43,4)	(73,7)	(31,8)	(10,13)	(41,9)
4	(4,2)	(13,0)	(43,1)	(67,9)	(34,4)	(8,13)	(41,12)
5	(5,1)	(14,1)	(43,4)	(82,6)	(32,10)	(12,9)	(45,9)
6	(3,2)	(13,38)	(42,6)	(60,12)	(30,8)	(10,13)	(48,3)
7	(5,2)	(14,0)	(42,4)	(77,5)	(31,7)	(8,13)	(41,10)
8	(6,1)	(14,1)	(43,2)	(66,10)	(34,8)	(12,6)	(43,8)
Average Ratio	3.45	2.65	10.625	9.18	4.48	0.85	5.29

Table 2.3: Two-Pair Method: Ratio of 3 cycles to chordless 4 cycles reported for average ratio

Number of Runs	Input {10,15}	Input {10,20}	Input {20,50}	Input {20,60}	Input {30,60}	Input {40,60}	Input {40,80}
	(number of 3 cycles, number of chordless 4 cycles)						
1	(3,4)	(9,11)	(35,25)	(51,48)	(20,31)	(4,17)	(13,74)
2	(4,2)	(12,3)	(32,32)	(59,29)	(27,16)	(2,23)	(35,22)
3	(2,6)	(8,11)	(37,17)	(52,51)	(25,20)	(8,13)	(23,43)
4	(4,3)	(12,6)	(34,24)	(61,44)	(19,36)	(2,21)	(36,20)
5	(3,4)	(10,8)	(32,26)	(65,18)	(21,31)	(4,17)	(23,42)
6	(0,10)	(11,5)	(30,42)	(55,34)	(23,23)	(2,23)	(22,41)
7	(7,2)	(12,13)	(34,26)	(62,41)	(19,38)	(5,16)	(27,37)
8	(0,11)	(10,9)	(35,25)	(50,49)	(13,51)	(0,25)	(26,46)
Average Ratio	0.54	1.27	1.23	1.44	0.67	0.17	0.63

Table 2.4: Separator Method: Ratio of 3 cycles to chordless 4 cycles reported for average ratio

2.2.3.4 Run Times

In this section we summarize the experiment conducted on two of the generation algorithms to generate weakly chordal graphs on the basis of their average run times. The first generation algorithm studied is based on joining two-pairs by the authors [2], and the second generation algorithm is based on separator by the authors [20]. Both these algorithms were implemented in Python 2.7. Both these algorithms are given same input values, and recorded for 10 iterations, the time taken by each of the algorithm to generate a weakly chordal graph on the input n vertices and m edges. The times reported by the two pair based generation algorithm [2] are reported in seconds in the Table 2.5 below along with the average run time. The times reported by the separator based generation algorithm [20] are reported in seconds in the Table 2.6 below along with the average run time.

Number of Runs	Input {30,60}	Input {40,70}	Input {50,100}	Input {60,120}	Input {80,150}	Input {90,190}	Input {100,170}
1	34.99	92.16	41.90	26.67	13.88	10.87	7.53
2	33.57	89.27	41.68	34.36	24.68	5.41	5.03
3	36.76	102.78	55.31	28.84	22.13	3.05	5.06
4	33.15	95.53	37.91	28.32	13.06	7.84	6.66
5	33.63	98.28	53.57	36.37	18.39	4.96	4.01
6	31.63	91.03	43.24	22.54	21.43	3.52	6.32
7	34.25	96.74	52.34	23.12	11.54	6.13	4.23
8	37.23	93.32	47.42	29.23	15.34	7.43	6.32
9	32.68	101.54	59.23	33.42	18.43	4.43	7.12
10	31.72	97.43	41.43	31.69	21.43	3.54	7.43
Average Time	33.96	95.80	47.40	29.45	18.03	5.71	5.97

Table 2.5: Two-Pair Method for generating WCG: Average Run Times

Number of Runs	Input {30,60}	Input {40,70}	Input {50,100}	Input {60,120}	Input {80,150}	Input {90,190}	Input {100,170}
1	0.19	0.22	0.91	1.80	1.61	2.79	3.04
2	0.30	0.33	0.77	1.42	1.85	3.48	3.01
3	0.20	0.28	0.67	1.53	1.28	2.15	2.91
4	0.22	0.27	0.71	1.85	1.83	3.22	3.89
5	0.32	0.17	0.74	0.77	1.43	2.95	2.71
6	0.53	0.34	0.59	1.23	1.93	3.78	3.82
7	0.28	0.47	0.73	1.71	1.08	4.12	2.43
8	0.21	0.25	0.69	.67	1.23	4.00	3.49
9	0.23	0.33	0.55	1.10	1.22	2.32	2.94
10	0.31	0.41	0.68	1.09	1.56	3.69	2.83
Average Time	0.27	0.30	0.70	1.31	1.50	3.25	3.10

Table 2.6: Separator Based Method for generating WCG: Average Run Times

Chapter 3

Proposed method

In this chapter we explain the proposed approach to generate a weakly chordal graph while starting from an input of an arbitrary graph. This chapter begins with an overview of the complete approach and then outlines detailed explanation for each step followed in the process. This chapter is finally concluded with the proposed algorithm posted for publication and extremely detailed examples from its implementation in Python 2.7. In the proposed method, we propose a scheme to generate weakly chordal graphs by first beginning with an input of vertices and edges required for the final weakly chordal graph to be obtained. The algorithm initiates by generating a random graph on given input, let this be G. The algorithm then reduce G to a chordal graph G', using the minimum vertex degree heuristic. The fill edges that are added while converting an arbitrary input graph to a chordal graph, are marked as potential candidates for subsequent deletion. Since G' is necessarily a weakly chordal graph, we use an algorithm for deleting edges from a weakly chordal graph to remove fill edges, maintaining the weak chordality property. In order to delete as many fill edges as possible we create a queue of all the fill edges. A fill edge is removed from the front of the queue, which we then try to delete. If we don't succeed we put it at the back of the queue. We keep doing this until no more fill edges can be removed. Operationally, we implement this by defining a deletion round as one in which the edge at the

back of the queue is at the front. We stop when the size of the queue does not change over two successive deletion rounds. The subsequent sections elaborate this process in detail.

3.1 Overview of the method

The algorithm begins by generating a random graph G on n vertices and m edges. Before this we already check if G is weakly chordal, using the LB-simpliciality recognition algorithm by to [7]. If G is weakly chordal, we stop terminate to find another. Otherwise, we proceed as follows. We first convert random graph G to a chordal graph H by introducing additional edges, which are named as fill-edges using the minimum degree vertex (mdv, for short) heuristic [11].

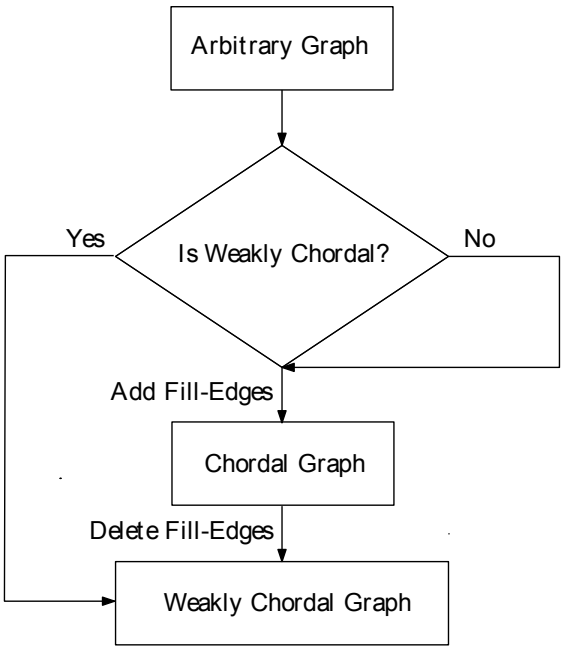


Figure 3.1: Overview of process (figure borrowed from [17])

The edges that are added to convert the random graph into a chordal graph is done on the basis of mdv heuristic that adds edges so that the minimum degree vertex in the current graph is simplicial. All the edges that are added which are named as fill-edges are entered into a queue, named as a fill-edge queue, FQ. These fill-edges are the only candidates for following the upcoming deletion process from

H. Since H is chordal, it is also a weakly chordal graph. We propose an algorithm that deletes edges from this weakly chordal graph to remove fill-edges, maintaining the weak chordality property. A fill-edge is deleted on the criteria that it does not create a hole or an antihole in the graph. We propose a criteria for detecting holes and anti-holes in a weakly chordal graph. Starting from the front of the queue, a fill-edge in taken to try for deletion. If deleting the fill-edge does not result in a hole or anti-hole configuration then we successfully delete this edge, else we put it at the back of the queue. We keep doing this until no more fill-edges can be deleted. We implement this by defining one deletion round as one in which the fill-edge at the back of the queue is at the front. We stop when the size of the queue does not change over two successive deletion rounds. Figure 3.1 is a flow chart for the flow of control followed subsequently.

3.2 Generating random arbitrary graphs

The first step is to generate a random graph on the given input. To generate a random graph, we use the algorithm by Keith M. Briggs, called 'dense_gnm_random_graph'. This algorithm is based on Knuth's Algorithm from the Selection sampling technique, of section 3.4.2 of [18]. It initiates by taking two input variables n and m, where n is the number of vertices and m is the number of edges to produce a random graph. For a given input n, we set m to a random value lying in the range between $n-1$ and $\frac{n(n-1)}{2}$. The output graph may or may not be disconnected. In case it is disconnected we connect the disjoint components using additional edges.

3.2.1 LB-simpliciality Test

A recognition algorithm propsed by the authors in [6] is used for the next step. In [7] Berry et al. proved the following result:

Theorem 3.2.1 *[7] A graph is weakly chordal if and only if every edge is LB-simplicial.*

We apply this recognition algorithm to the random graph generated by the previous step. We then continue with the next steps only in the case that the recognition algorithm fails. Otherwise, we return G.

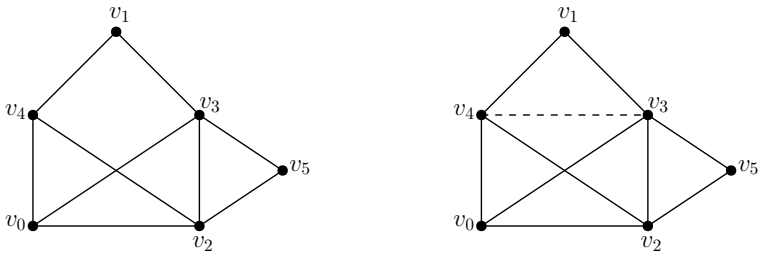
3.3 Arbitrary Graph to Chordal Graph

We started with a random arbitrary graph G and then converted it into a chordal graph H. The starting random arbitrary graph G is actually embedded within the chordal graph H. This process of addition of edges and the process is known as fill-in or triangulation. Triangulations in which a minimum or a minimal number of edges is added, are desirable. A triangulation $H = (V, E \cup F)$ of $G = (V, E)$ is minimal if $(V, E \cup F')$ is non-chordal for every proper subset F' of F. In a minimum triangulation the number of edges added is the fewest possible. Berry at al. [5] proposed an algorithm, known as LB-Triangulation, for the minimal fill-in problem. LB-Triangulation works on any ordering α of the vertices, and produces a fill that is provably exclusion-minimal. In the proposed algorithm, we have used the *mdv* heuristic [11], as the experiments conducted have shown that this adds fewer fill-edges as compared to LB-Triangulation. The next section briefly explains this heuristic.

3.3.1 The Minimum Degree Vertex Heuristic

Let $H = (V, E \cup F)$ be the graph obtained from $G = (V, E)$, where F is set of fill-edges. We start to prune from the graph G all the vertices that have a degree equal to 1 and assign it to H for the resulting graph G. From the vertices remaining of G, we choose v a vertex of minimum degree by breaking the ties arbitrarily and turn the neighborhood $N(v)$ of v into a clique by adding edges. These are fill-edges that we add to the edge set of H, as well as to the fill-edge queue, FQ. Finally, we then remove from G, the vertex v and all the edges that are incident on it. We continue to repeat this process until the graph G is empty. The graph

H is now a chordal graph and is identical with the initial graph G, sans degree 1 vertices, and with fill-edges added. This is illustrated with an example.



(a) *Arbitrary graph* (b) *Chordal Graph*

Figure 3.2: *Arbitrary graph to chordal graph (figure borrowed from [17])*

The initial graph G shown in Fig. 3.2(a) and the graph H with all fill-edges added shown in Fig. 3.2(b). In the initial graph G both v_1 and v_5 have a minimum degree. We break the tie in favour of v_5. Since the induced subgraph on $N(v_5)$ is already a clique no fill-edges are added and G is set to $G - \{v_5\}$. In the reduced graph G, v_1 is of minimum degree and the induced graph on $N(v_1)$ is turned into a clique by adding $\{v_3, v_4\}$ as a fill-edge, which is also added to H. Since the reduced graph $G - \{v_1\}$ is a clique, we can pick the vertices v_0, v_2, v_3, v_4 in an arbitrary fashion to reduce the graph G to an empty graph, without the need to be introducing any further fill-edges into H. The algorithm [17] for this process is described below:

Algorithm 3.1 ArbitraryToChordal

Input: An arbitrary graph $G = (V, E)$
Output: Returns a chordal graph $H = (V, E \cup F)$ and fill-edge queue FQ

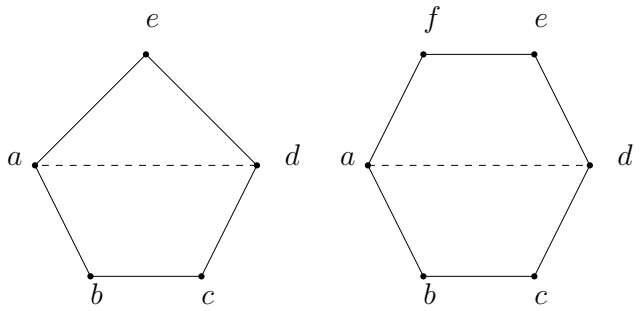
1: Delete all vertices of degree 1 from G and assign to H
2: Sort V in ascending order of degrees
3: Choose a vertex v of minimum degree
4: Turn $N(v)$ of v into a clique by adding edges, which are added to the edge set of H and to the fill-edge queue, FQ
5: Remove the vertex v from G and all the edges incident on it
6: Repeat steps 2 to 5 until G is empty

3.4 Chordal Graph to Weakly Chordal Graph

The chordal graph H is necessarily a weakly chordal, and hence, we apply an algorithm that will be based on deleting edges to H that preserves weak chordality. The edges that were added by the *mdv* heuristic are the edges that make candidate for deletion. The candidate edge each time is deleted temporarily from H, so we can check if the deletion of this edge results in a hole or an antihole configuration in H. If the deletion of this edge does not result in a hole or an antihole configuration, we delete this edge. This complete process is detailed in the sections to follow.

3.4.1 Fill-Edge Queue

We define a fill-edge as each edge that was added to convert an arbitrary input graph into chordal graph. To be able to delete as many fill-edges as possible, a queue of fill-edges FQ is maintained. From the front of the queue, a fill-edge is removed which we try to then delete from H. Incase this gives rise to a hole or an antihole configuration, we put it at the end of the queue. We keep repeating this until from FQ no more fill-edges can be removed.



(a) One P_4 and one P_3 $(b) Two P_4$

Figure 3.3: *Detecting Holes*

3.4.2 Detecting Holes

A hole in a graph G can be defined as an chordless cycle induced on five or more vertices. Since, a graph is weakly chordal if it is (hole, antihole)-free [10], it is important to detect if the deletion of an edge gives rise to a hole configuration. The class of weakly chordal graphs allows the biggest cycle of size four, the holes can hence be formed either by a pair of two P_4's or a by a pair of a P_3 and a P_4, as shown in Fig. 3.3.

We can detect a hole configuration in H by starting with an edge $\{u, v\}$ of H and start by deleting it temporarily. Next, we must make sure to check if the deletion of this edge has created a hole in H. To detect a hole, a breadth-first search in H is performed with u as the starting vertex and find all chordless P_3 and P_4 paths between the two vertices u and v. A hole can be created in two following ways: (a) by a disjoint pair of P_4, having six distinct vertices between them such that there exists no chord joining an internal vertex on one P_4 to an internal vertex on the other, this is termed as a hole on two P_4s; (b) by a disjoint pair of P_3 and P_4 between the two vertices u and v of the temporarily deleted edge, with five distinct vertices between them, such that there exist no chord joining an internal vertex on the P_4 to the internal vertex of the P_3; this is termed as a hole on a P_3 and a P_4.

For example in Figure 3.3, (a) has {a,b,c,d} as one P_4 and {a,e,d} as one P_3, which together join to make a cycle of size 5 which is not permitted for a weakly chordal graph. In Figure 3.3, (b) has {a,b,c,d} and {a,f,e,d} are two P_4's, which together join to make a cycle of size which is not permitted for a weakly chordal graph.

3.4.3 Antiholes

By definition an antihole in a graph is the complement of a hole [10]. An antihole configuration in a weakly chordal graphs has the structure shown in Fig. 3.4. This is an induced graph on six distinct vertices each of which is of degree three.

Carefully note the structure shown in Fig. 3.4. It is created by two P_3 paths, $\{a, c, d\}$, $\{a, f, d\}$, and one P_4 path, $\{a, b, e, d\}$, between a and d. Furthermore, there is an edge connecting b to c and another connecting e to f. All the vertices have a degree of 3 if {a,d} are deleted.

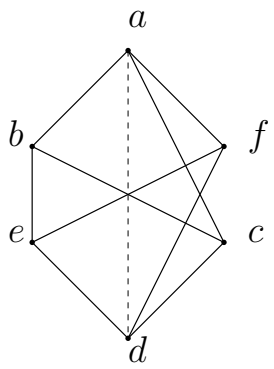


Figure 3.4: Antihole

3.4.4 Detecting Antiholes

For an antihole configuration to be detected in the graph, we pick an edge $\{a, d\}$ and delete it from the graph temporarily. We then check if deletion of the edge $\{a, d\}$ creates an antihole structure in the graph. To detect this, we first implement breadth-first search with a as the starting vertex to find all chordless P_3 and P_4 paths between a and d. An antihole structure is formed by a pair of two P_3 and one P_4 such that the induced graph on the six vertices, are uniformly all of the same degree equal to three and there exists a chord from the internal vertex of each P_3 to one of the internal vertices in the P_4. For example, in Fig. 3.4, $\{a, b, e, d\}$ is a P_4, $\{a, c, d\}$ and $\{a, f, d\}$ are two P_3 paths. There exists exactly one chord from b to c and exactly one from e to f and, in the induced graph on these six vertices,

every vertex has degree three, making it an antihole configuration.

3.5 Proposed Algorithm

This section outlines the proposed algorithm for deleting edges from a weakly chordal graph to remove fill edges while maintaining its weak chordality property. To be able to delete as many fill-edges as possible, we start by removing a fill-edge $\{u, v\}$ from the front of the fill-edge queue, which we then try to delete from H. If we do not succeed in deleting the fill-edge, we put it at the back of the queue. We keep repeating this until no more fill-edges can be removed from the fill-edge queue. In practise, we implement this by defining one deletion round as one in which the edge at the end of the queue is at the start. One deletion round comprises of picking an edge from the start of the queue and deleting it from H. Now we check if the deletion of $\{u, v\}$ creates a hole or an antihole configurations in H. If it creates a hole or an antihole configuration, we do not delete the edge $\{u, v\}$ and add it back to the fill-edge queue. Otherwise, we delete the edge from H and also remove it from the fill-edge queue FQ. When the size of the fill-edge queue FQ does not change over two successive deletion rounds, we stop.

A random arbitrary graph on 6 vertices and 8 edges is taken for example in Fig. 3.5. It is converted into a chordal graph by inserting two additional edges. These two additional edges added are put in the fill-edge queue $[\{b, d\}, \{a, d\}]$. We first maintain a temporary copy of chordal graph G in T. The deletion algorithm begins by picking first edge $\{b, d\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting $\{b, d\}$ does not give rise to any hole or antihole configurations, $\{b, d\}$ is permanently deleted from starting graph H which is now a weakly chordal graph. Now the updated fill-edge queue is $[\{a, d\}]$. The deletion algorithm now picks the first edge in $\{a, d\}$ in the fill-edge queue and temporarily deletes it from graph T to check for

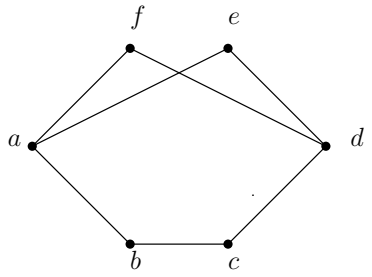
Algorithm 3.2 ChordalToWeaklyChordal

Input: A chordal graph $H = (V, E \cup F)$ with fill-edge queue FQ
Output: A weakly chordal graph G_w

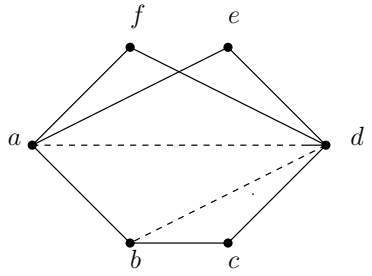
```
1: T ← H                                          ▷ Make a copy of H
2: FQ ← fill-edges of H
3: prevSize ← 0
4: newSize ← |FQ|
5: while (prevSize ≠ newSize && newSize ≠ 0) do   ▷ Check size of FQ over
   two deletion rounds
6:     prevSize ← newSize
7:     for (each edge{u, v} infill-edge queue, FQ) do
8:         Delete edge {u, v} from T
9:         if (Hole or Antihole Detected) then
10:            Do not delete edge from graph H, add edge back to temporary graph
   T, and to the back of the queue FQ
11:        else
12:            Delete edge {u, v} from graph H
13:        end if
14:    end for
15:    newSize ← |FQ|
16: end while
17: G_w ← H
18: return G_w
```

hole and antihole configurations. Since deleting $\{a, d\}$ gives rise to a hole configuration on one P_4 $\{a, b, c, d\}$ and one P_3 $\{a, f, d\}$, $\{a, d\}$ is not permanently deleted from H. Since the queue is now empty, the graph G_w returned by the algorithm is weakly chordal with a small subset of fill-edges added to the original graph G.

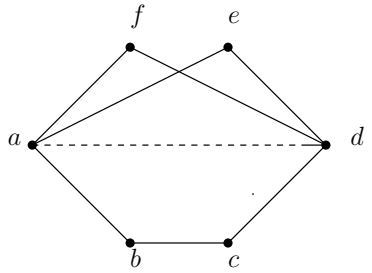
Consider Figure 3.6 for another example, a random arbitrary graph on 6 vertices and 9 edges is obtained. It is converted into a chordal graph H (see Figure 3.6) by adding three additional edges. These three additional edges added are put in the fill-edge queue $[\{a, c\}, \{b, d\}, \{a, d\}]$. Maintain a temporary copy of the chordal graph H in T. The deletion algorithm begins by picking first edge $\{a, c\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting $\{a, c\}$ does not give rise to any hole or antihole configurations, $\{a, c\}$ is permanently deleted from starting graph H which is now a weakly chordal graph shown in Figure 3.6. Now the updated fill-edge



(a) Arbitrary Graph



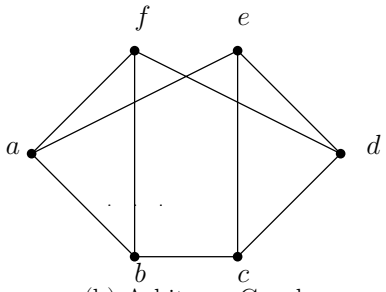
(b) Chordal Graph



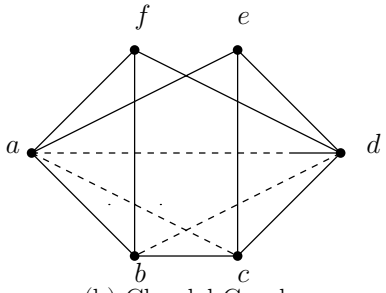
(c) Weakly Chordal Graph

Figure 3.5: *Arbitrary graph to weakly chordal graph*

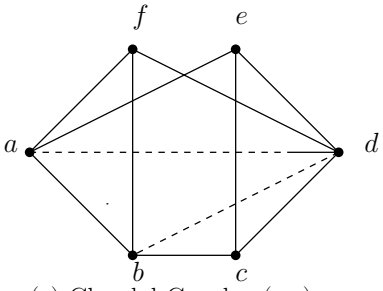
queue is $[\{b,d\},\{a,d\}]$. The deletion algorithm now picks the first edge in $\{b,d\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting $\{b,d\}$ does not give rise to any hole or antihole configuration, $\{b,d\}$ is permanently deleted from starting graph H, which is now a weakly chordal graph. Now the updated fill-edge queue is $[\{a,d\}]$. The deletion algorithm now picks the first and only edge $\{a,d\}$ in the fill-edge queue and temporarily deletes it from graph T to check for a hole or an antihole configuration. Since deleting $\{a,d\}$ gives rise to an antihole configuration on two P_3 paths $\{a,f,d\}$,$\{a,e,d\}$ and one P_4 $\{a,b,c,d\}$, the edge $\{a,d\}$ is not permanently deleted from starting graph H. Since the queue is now empty, the graph G_w returned by the algorithm is weakly chordal with a small subset of fill-edges



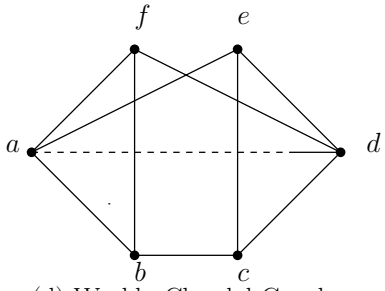
(b) Arbitrary Graph



(b) Chordal Graph



(c) Chordal Graph - (a,c)



(d) Weakly Chordal Graph

Figure 3.6: *Arbitrary graph to weakly chordal graph*

added to the original graph G as shown in Figure 3.6.

3.5.1 Results

3.5.1.1 Output 1

The following example takes an input of 10 nodes and 15 edges.

Consider Figure 3.7(a), a random arbitrary graph on 10 vertices and 15 edges is obtained.

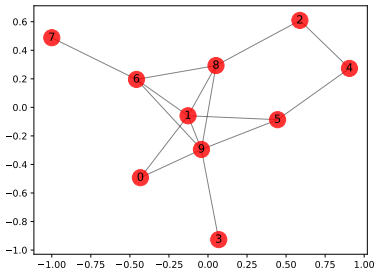
It is converted into a chordal graph H (see Figure 3.7(b)) by adding 2 additional edges.

These 2 additional edges added are put in the fill-edge queue $[\{8,4\},\{8,5\}]$. Maintain a temporary copy of the chordal graph H in T.

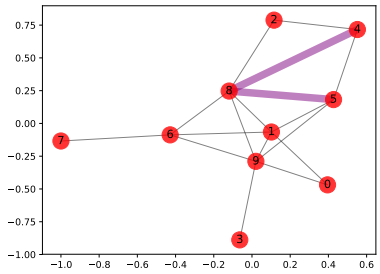
The deletion algorithm begins by picking first edge $\{8, 4\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting it does not give rise to any hole or antihole configurations, $\{8, 4\}$ is permanently deleted from starting graph H.

Now the updated fill-edge queue is $[, \{8, 5\}]$. The deletion algorithm now picks the first edge in $\{8, 5\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting $\{8, 5\}$ does not give rise to any hole or antihole configuration, $\{8, 5\}$ is permanently deleted from starting graph H, which is now a weakly chordal graph.

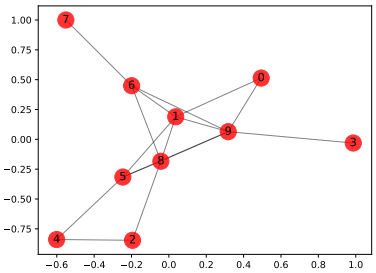
Since the queue is now empty, the graph G_w returned by the algorithm is weakly chordal with a small subset of fill-edges added to the original graph G as shown in Figure 3.7(c).



(a) *Arbitrary Graph (G)*



(b) *Chordal Graph (H)*



(c) *Weakly Chordal Graph* ($H = H - \{v_4, v_8\} = G_w$)

Figure 3.7: *Arbitary graph to a weakly chordal one*

3.5.1.2 Output 2

The following example takes an input of 12 nodes and 18 edges.

Consider Figure 3.9(a), a random arbitrary graph on 12 vertices and 18 edges is obtained.

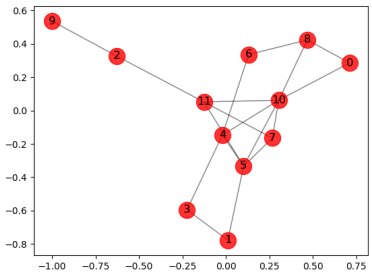
It is converted into a chordal graph H (see Figure 3.9(b)) by adding 2 additional edges.

These 2 additional edges added are put in the fill-edge queue $[\{8,4\},\{3,5\}]$. Maintain a temporary copy of the chordal graph H in T.

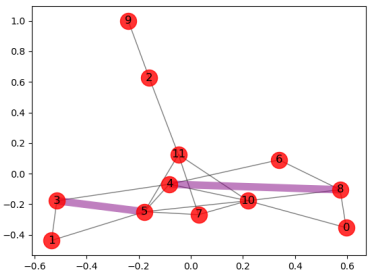
The deletion algorithm begins by picking first edge $\{8,4\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting it does not give rise to any hole or antihole configurations, $\{8,4\}$ is permanently deleted from starting graph H.

Now the updated fill-edge queue is $[,\{3,5\}]$. The deletion algorithm now picks the first edge in $\{3,5\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting $\{3,5\}$ does not give rise to any hole or antihole configuration, $\{3,5\}$ is permanently deleted from starting graph H, which is now a weakly chordal graph.

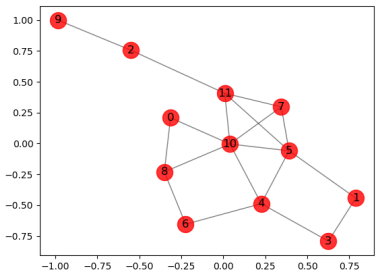
Since the queue is now empty, the graph G_w returned by the algorithm is weakly chordal with a small subset of fill-edges added to the original graph G as shown in Figure 3.9(c).



(a) *Arbitrary Graph (G)*



(b) *Chordal Graph (H)*



(c) *Weakly Chordal Graph*

Figure 3.8: *Arbitary graph to a weakly chordal one*

3.5.1.3 Output 3

The following example takes an input of 15 nodes and 20 edges.

Consider Figure 3.9(a), a random arbitrary graph on 15 vertices and 20 edges is obtained.

It is converted into a chordal graph H (see Figure 3.9(b)) by adding 3 additional edges.

These 3 additional edges added are put in the fill-edge queue $[\{8,6\},\{3,8\},\{10,14\}]$. Maintain a temporary copy of the chordal graph H in T.

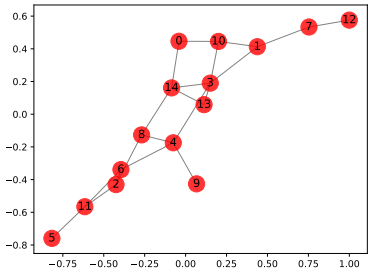
The deletion algorithm begins by picking first edge $\{8,6\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting it does not give rise to any hole or antihole configurations, $\{8,6\}$ is permanently deleted from starting graph H.

Now the updated fill-edge queue is $[\{3,8\},\{10,14\}]$. The deletion algorithm now picks the first edge in $\{3,8\}$ in the fill-edge queue and temporarily deletes it

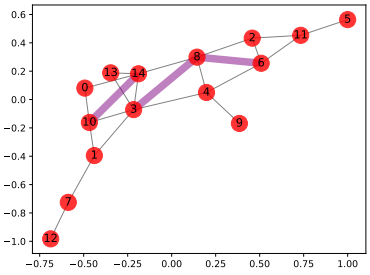
from graph T to check for hole and antihole configurations. Since deleting $\{3, 8\}$ does not give rise to any hole or antihole configuration, $\{3, 8\}$ is permanently deleted from starting graph H, which is now a weakly chordal graph.

Now the updated fill-edge queue is $[\{10, 14\}]$. The deletion algorithm now picks the first edge in $\{10, 14\}$ in the fill-edge queue and temporarily deletes it from graph T to check for hole and antihole configurations. Since deleting $\{10, 14\}$ does not give rise to any hole or antihole configuration, $\{10, 14\}$ is permanently deleted from starting graph H, which is now a weakly chordal graph.

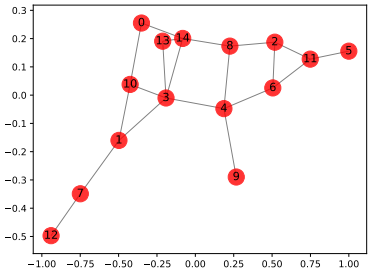
Since the queue is now empty, the graph G_w returned by the algorithm is weakly chordal with a small subset of fill-edges added to the original graph G as shown in Figure 3.9(c).



(a) *Arbitrary Graph (G)*



(b) *Chordal Graph (H)*

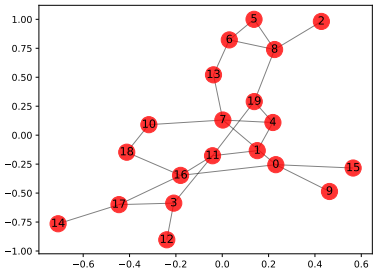


(c) *Weakly Chordal Graph*

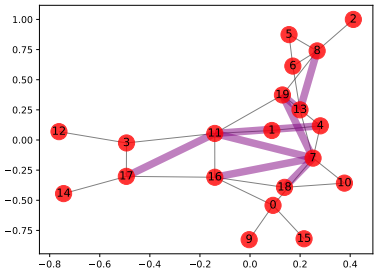
Figure 3.9: *Arbitary graph to a weakly chordal one*

3.5.1.4 Output 4

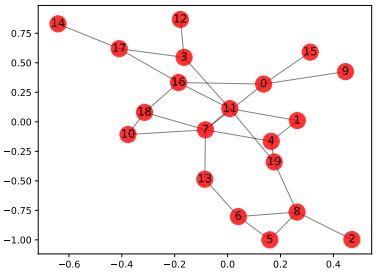
The following example takes an input of 20 nodes and 25 edges.



(a) *Arbitrary Graph (G)*



(b) *Chordal Graph (H)*



(c) *Weakly Chordal Graph*

Figure 3.10: *Arbitary graph to a weakly chordal one*

3.5.2 Experiments

To convert an arbitrary graph G to a chordal graph H we use a triangulation algorithm. In order to add as few edges as possible while converting arbitrary graph G to chordal graph H, and to be able to assess which triangulation algorithm adds fewer edges when converting arbitrary graph G to a chordal graph H, we conduct an experiment to compare the two existing triangulation algorithms, minimum degree vertex (explained in detail in section 3.3.1) and minimal triangulation. The author in [5] introduced an algorithm that provides minimal triangulation in O(nm) time, and can create any minimal triangulations of an arbitrary graph in any order of vertices. LB- Triangulation is an efficient algorithm to compute minimal triangulation using an arbitrary ordering on the vertices [5]. In this algorithm, any ordering on the vertices,produces minimal triangulation by adding only the necessary edges at each step, instead of making the current vertex simplicial.

In this experiment we start by generating an arbitrary graph G on the input number of vertices v and edges e. We supply the same arbitrary graph G to both minimum degree vertex and LB-triangulation method to report the number of edges added by each algorithm to convert arbitrary graph G to chordal graph H. We then proceed to convert both chordal graphs H generated by the two methods to make the final weakly chordal graph and report the final number of edges in the weakly chordal graph by the two methods.

Arbitrary Graph {V,E}	Fill Edges added by MDV	Final Edges left in WCG	Fill Edges added by LBT	Final Edges left in WCG
{10,30}	4	30	4	30
{10,30}	4	31	6	31
{10,30}	8	33	14	33
{10,30}	4	30	4	30
{10,30}	7	32	9	33
{10,30}	3	30	5	30

Table 3.1: Comparison for number of edges added by MDV and LB-Triangulation

Arbitrary Graph {V,E}	Fill Edges added by MDV	Final Edges left in WCG	Fill Edges added by LBT	Final Edges left in WCG
{15,20}	6	23	5	22
{15,20}	8	23	8	24
{15,20}	4	21	4	21
{15,20}	4	22	4	22
{15,20}	14	27	11	26

Table 3.2: Comparison for number of edges added by MDV and LB-Triangulation

Arbitrary Graph {V,E}	Fill Edges added by MDV	Final Edges left in WCG	Fill Edges added by LBT	Final Edges left in WCG
{12,15}	6	18	5	17
{12,15}	3	15	1	15
{12,15}	8	20	5	18
{12,15}	7	17	5	17
{12,15}	2	15	2	15

Table 3.3: Comparison for number of edges added by MDV and LB-Triangulation

Arbitrary Graph {V,E}	Fill Edges added by MDV	Final Edges left in WCG	Fill Edges added by LBT	Final Edges left in WCG
{12,20}	10	25	9	24
{12,20}	7	22	6	22
{12,20}	4	21	3	21
{12,20}	12	20	7	22
{12,20}	7	23	6	24

Table 3.4: Comparison for number of edges added by MDV and LB-Triangulation

Arbitrary Graph {V,E}	Fill Edges added by MDV	Final Edges left in WCG	Fill Edges added by LBT	Final Edges left in WCG
{20,25}	6	29	8	29
{20,25}	11	30	20	35
{20,25}	16	31	22	30
{20,25}	4	26	7	28
{20,25}	15	34	21	36

Table 3.5: Comparison for number of edges added by MDV and LB-Triangulation

On computing experiments for more graphs, it is evident the minimum degree vertex adds fewer or equal edges in comparison to LB-Triangulation to convert an arbitrary graph G to a chordal graph H. Hence, we chose to use minimum degree

vertex for converting an arbitrary graph G to a chordal graph H for the remainder of the proposed algorithm.

3.5.3 Complexity

To compute the complexity we first consider that the mdv heuristic can be implemented in $O(n^2m)$ time, while the time-complexity of the recognition algorithm based on LB-simpliciality is in $O(nm)$. Next to be able to bound the query complexity of deleting from the weakly chordal graph an edge $\{u, v\}$, we note that this is dominated by the task of finding multiple P_3 and P_4 paths between u and v and we have to consider these in pairs and run the breadth-first search. An upper bound on the number of pairs of P_3 and P_4 paths between u and v is $O(d_u^2 d_v^2)$, where d_u and d_v are the degrees of u and v respectively. For consider such a path from u to v (see Figure 3.11): x is one of the at most d_u vertices adjacent to u and y is one of the at most d_v vertices adjacent to v, so that we have at most $O(d_u d_v)$ P_4 paths from u to v and thus $O(d_u^2 d_v^2)$ disjoint pairs of P_4 paths from u to v.

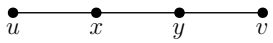


Figure 3.11: *A P_4-path from u to v*

Now, if If $|E|$ be the number of edges currently, in the weakly chordal graph, the complexity of running a breadth-first search is $O(n + |E|)$. Since m is the number of edges in the final weakly chordal graph, an upper bound on the query complexity is $O(d_u^2 d_v^2(n + m))$. The deletion of an edge take constant time since we maintain an adjacency matrix data structure to represent G.

Chapter 4

Conclusions

In this thesis, we focus on the problem of generating weakly chordal graphs. In the first part, we carry out a comparative study of two existing algorithms for generating weakly chordal graphs. Their outputs are compared with respect to several parameters to study how they differ structurally. Chapter 1 provides outline to the overview of this thesis, problem statement and the underlying motivation to carry out this research. Chapter 2 discusses two of the existing algorithms in the literature to generate weakly chordal graphs. These two algorithms are explained in detail and results from it's Python implementation are attached. The two implementations are compared in Chapter 2 in order to understand how the outputs differ structurally from the graphs being generated by two different generation algorithms.

In [7] the authors proposed an open problem of generating weakly chordal graph starting from an arbitrary graph. A solution to this open problem forms the second part of this thesis. Chapter 3 outlines the main contribution of this thesis providing a solution to the open problem by proposing an algorithm to generate weakly chordal graphs starting from an arbitrary input graph. While the two existing generation algorithms studied are based on the various structural properties of weakly chordal graphs, the first generation algorithm [2] is based on notion of a two-pair in a graph and the other [20] is based on separators in

a graph to generate weakly chordal graphs, the proposed algorithm to generate weakly chordal graph [17] uses the notion of holes and anti-holes. The proposed algorithm allows us to be able to generate dense weakly chordal graphs and it does not require any complex data structures. An application of this generation algorithm would be to obtain and discover the test-instances for an algorithm for enumerating linear layouts of a weakly chordal graph proposed in [3].

4.1 Future work

Further work can be done on several fronts. More work needs to be done in order to be able to understand how the outputs of the different algorithms to generate weakly chordal graphs are structurally different.

An interesting open problem will be to be able to prove minimality for the proposed method, to be able to establish if the proposed method to generate weakly chordal graph starting from an arbitrary graph, adds a minimal number of edges.

Another open problem to put forward is for counting the number of labeled weakly chordal graphs on n vertices and m edges, which will further help generate weakly chordal graphs uniformly at random.

BIBLIOGRAPHY

[1] Jean Paul Bordat Anne Berry and Pinar Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nord. J. Comput*, 7(3):164–177, 2000.

[2] Srinivasa Rao Arikati and C. Pandu Rangan. An efficient algorithm for finding a two-pair, and its applications. *Discrete Applied Mathematics*, 31(1):71–74, 1991.

[3] Sidharth Pardeshi Asish Mukhopadhyay, S. V. Rao and Srinivas Gundlapalli. Linear layouts of weakly triangulated graphs. *Discrete Math., Alg. and Appl.*, 8(3):1–21, 2016.

[4] Roger Baker and Kenneth Kuttler. *Linear algebra with applications*. World Scientific, 2014.

[5] Anne Berry. A wide-range efficient algorithm for minimal triangulation. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA.*, pages 860–861, 1999.

[6] Anne Berry, Jean Paul Bordat, and Pinar Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nord. J. Comput.*, 7(3):164–177, 2000.

[7] Anne Berry, Jean Paul Bordat, and Pinar Heggernes. Recognizing weakly triangulated graphs by edge separability. In *Algorithm Theory - SWAT 2000,*

7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings, pages 139–149, 2000.

[8] Sritharan R. Tang Y. Cameron, K. Finding a maximum induced matching in weakly chordal graphs. *Discrete Mathematics*, 266(1-3):133–142, 2003.

[9] Francis YL Chin, Henry CM Leung, Wing-Kin Sung, and Siu-Ming Yiu. The point placement problem on a line–improved bounds for pairwise distance queries. In *Algorithms in Bioinformatics*, pages 372–382. Springer, 2007.

[10] Carl Feghali and Jirí Fiala. Reconfiguration graph for vertex colourings of weakly chordal graphs. *CoRR*, abs/1902.08071, 2019.

[11] Alan George and Joseph W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[12] R. B. Hayward. Weakly triangulated graphs. *J. Combinatorial Theory*, Series B 39:200–209, 2019.

[13] Ryan B. Hayward. Weakly triangulated graphs. *J. Comb. Theory, Ser. B*, 39(3):200–208, 1985.

[14] Ryan B. Hayward. *Two classes of perfect graphs.* PhD thesis, McGill University, 1987.

[15] Ryan B. Hayward. Generating weakly triangulated graphs. *Journal of Graph Theory*, 21(1):67–69, 1996.

[16] Ryan B. Hayward, Chính T. Hoàng, and Frédéric Maffray. Optimizing weakly triangulated graphs. *Graphs and Combinatorics*, 5(1):339–349, 1989.

[17] Sudiksha Khanduja, Aayushi Srivastava, Asish Mukhopadhyay, and Md. Zamilur Rahman. Generating weakly chordal graphs from arbitrary graphs. *arXiv:2003.13786v1*, 2020.

[18] Donald E. Knuth. *The Art of Computer Programming, Volume 2/Seminumerical algorithms, Third Edition.* Addison-Wesley, 1997.

[19] Lilian Markenzon, Oswaldo Vernet, and Luiz Henrique Araujo. Two methods for the generation of chordal graphs. *Annals OR*, 157(1):47–60, 2008.

[20] Asish Mukhopadhyay Md. Zamilur Rahman and Yash P. Aneja. A separator-based method for generating weakly chordal graphs. *CoRR*, page abs/1906.01056, 2019.

[21] Chinh T. Hoang Ryan B. Hayward and Frederic Maffray. Optimizing weakly triangulated graphs. *Graphs and Combinatorics*, 5(1):339–349, 1989.

[22] Jeremy P. Spinrad Ryan B. Hayward and R. Sritharan. Improved algorithms for weakly triangulated graphs. *ACM Trans. Algorithms*, 3(2):1–14, 2007.

[23] Jeremy P. Spinrad and R. Sritharan. Algorithms for weakly triangulated graphs. *Discrete Applied Mathematics*, 59(2):181–191, 1995.

[24] Jeremy P. Spinrad and R. Sritharan. Algorithms for weakly triangulated graphs. *Discrete Applied Mathematics*, 59(2):181–191, 1995.

[25] G. Tinhofer. Generating graphs uniformly at random. *Springer Vienna*, pages 235–255, 1990.

VITA AUCTORIS

Name: Sudiksha Khanduja

Place of Birth: Bhilai, Chhattisgrah, India

Year: 1995

Education: Bachelor of Technology in Computer Sciences from SRM University, India (2014 - 2018)

Masters in Computer Science at the University of Windsor, Canada (Fall 2018 - Summer 2020)