7-7-2020

Chordal Graphs and Their Relatives: Algorithms and Applications

Md Zamilur Rahman
University of Windsor

Chordal Graphs and Their Relatives: Algorithms and Applications

by

Md Zamilur Rahman

A Dissertation

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

Windsor, Ontario, Canada

2020

Chordal Graphs and Their Relatives: Algorithms and Applications

by

Md Zamilur Rahman

APPROVED BY:

A. Maheshwari, External Examiner
Carleton University

F. Baki
Odette School of Business

M. Kargar
School of Computer Science

D. Wu
School of Computer Science

Y. Aneja, Co-Advisor
Odette School of Business

A. Mukhopadhyay, Advisor
School of Computer Science

April 20, 2020

Declaration of Co-Authorship/ Previous Publication

I. Co-Authorship

I hereby certify that this dissertation incorporates material that is the result of my research conducted under the supervision of my advisors Prof. Dr. A. Mukhopadhayay and Prof. Dr. Y. Aneja. In all cases, the key ideas, primary contributions, experimental designs, data analysis, interpretation, and writing were performed by the author. In chapter 2, Tajinder Dhillon and Mitchell Fujs helped in the implementation of a modified version of k-chromatic chordal graph generation.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publications

This thesis includes some materials from 2 original papers that have been previously published/submitted for publication in peer reviewed conferences/journals, as follows:

Thesis Chapter	Publication Title/Full Citation	Publication Status
Chapter 3	Md Zamilur Rahman, Asish Mukhopadhyay, and Yash P. Aneja. A separator-based method for generating weakly chordal graphs. CoRR, abs/1906.01056, 2019.	arXiv.org
	Md Zamilur Rahman, Asish Mukhopadhyay, and Yash P. Aneja. "A separator-based method for generating weakly chordal graphs" In Discrete Mathematics, Algorithms and Applications, 2019-2020.	in press
Chapter 4	Md Zamilur Rahman, Asish Mukhopadhyay, and Yash P. Aneja. An algorithm for generating strongly chordal graphs. CoRR, abs/1804.09019, 2018.	arXiv.org
	Md. Zamilur Rahman and Asish Mukhopadhyay. Strongly chordal graph generation using intersection graph characterization. CoRR, abs/1909.02545, 2019.	arXiv.org
Chapter 5	Md Zamilur Rahman and Asish Mukhopadhyay. Semi-dynamic Algorithms for Strongly Chordal Graphs. CoRR, abs/2002.07207, 2020.	arXiv.org
Chapter 6	Md Zamilur Rahman, Asish Mukhopadhyay, Yash P. Aneja, and Cory Jeane. "A distance matrix completion approach to 1-round algorithms for point placement in the plane". In International Conference on Computational Science and Its Applications, pages 494-508. Springer, 2017.	published (Cory Jeane helped in writing program for generating chordal graphs but that is not included in this dissertation)
	Md Zamilur Rahman, Udayamoorthy Navaneetha Krishnan, Cory Jeane, Asish Mukhopadhyay, and Yash P. Aneja. "A distance matrix completion approach to 1-round algorithms for point placement in the plane." In Transactions on Computational Science XXXIII, pages 97-114. Springer, Berlin, Heidelberg, 2018.	published (Cory Jeane's contributions remain the same but it is published with additional work in collaboration with Udayamoorthy Navaneetha Krishnan and the additional work is not included in this dissertation)

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my dissertation. I certify that the above material describes work completed during my registration as a graduate student at the University of Windsor.

III. General

I declare that, to the best of my knowledge, my dissertation 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 dissertation, 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 dissertation.

I declare that this is a true copy of my dissertation, including any final revisions, as approved by my dissertation committee and the Graduate Studies office, and that this dissertation has not been submitted for a higher degree to any other University or Institution.

Abstract

While the problem of generating random graphs has received much attention, the problem of generating graphs for specific classes has not been studied much. In this dissertation, we propose schemes for generating chordal graphs, weakly chordal graphs, and strongly chordal graphs. We also present semi-dynamic algorithms for chordal graphs and strongly chordal graphs. As an application of a completion technique for chordal graphs, we also discuss a 1-round algorithm for approximate point placement in the plane in an adversarial model where the distance query graph presented to the adversary is chordal.

The proposed generation algorithms take the number of vertices, n, and the number of edges, m, as input and produces a graph in a given class as output. The generation method either starts with a tree or a complete graph. We then insert additional edges in the tree or delete edges from the complete graph. Our algorithm ensures that the graph properties are preserved after each edge is inserted or deleted. We have also proposed algorithms to generate weakly chordal graphs and strongly chordal graphs from an arbitrary graph as input. In this case, we ensure the graph properties will be achieved on the termination of the conversion process.

We have also proposed a semi-dynamic algorithm for edge-deletion in a chordal graph. To the best of our knowledge, no study has been done for the problem of dynamic algorithms for strongly chordal graphs. To address this gap, we have also proposed a semi-dynamic algorithm for edge-deletions and a semi-dynamic algorithm for edge-insertions in strongly chordal graphs.

Dedication

To Zikra and Zayna

Acknowledgements

All praise is due to Allah, the Lord of the Worlds. I would like to thank the Almighty for giving me the opportunity, strength, determination, and patience to do my research. This work would not have been possible without His blessing.

I would like to take the opportunity to express my deepest gratitude and sincere appreciation to my supervisors, Prof. Dr. Asish Mukhopadhyay and Prof. Dr. Yash Aneja, for their continuous support, encouragement, patience, and invaluable guidance throughout my Ph.D. program. This work could not have been achieved without their continuous effort, suggestions, and cooperation. Words cannot thank them enough.

I would like to thank my committee members Prof. Dr. Fazle Baki, Dr. Dan Wu, and Dr. Mehdi Kargar for their time, constructive comments, and suggestions during my research. I would like to extend my thanks to Prof. Dr. Anil Maheshwari for his kind acceptance to be the examiner of my dissertation defense.

I would like to express my gratitude for the fund I have received through the University of Windsor graduate student support. I would also like to thank my lab mates for their help and inspiration during the program. I owe special thanks to the staff of the School of

Computer Science for their great support.

I would like to thank my parents, brother, wife, and relatives for their constant support, love, cooperation, sacrifice, and encouragement.

I also place on record my sense of gratitude to one and all who, directly or indirectly, have lent their helping hand in this venture.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A graph is a structure that represents the pairwise relationships between objects. The objects are represented by the set of vertices of the graph and a pairwise relationship between two objects is represented by an edge between them. We use graphs to model a large variety of real-world problems. These problems can arise in many practical and relevant fields, such as biology, social networks, circuit design, scheduling, telecommunication, and data analysis. By using a graph-theoretic approach, one constructs a graph that represents a mathematical model of the real-world aspects of a problem.

Graphs are categorized into different classes based on their properties. Among them, chordal (also known as a triangulated, a rigid circuit) graphs are one of the most extensively studied classes of graphs. Chordal graphs arise in many practical and relevant fields such as computing the solutions of systems of linear equations, in database management systems, VLSI, biology, and so on [7, 11, 12, 35, 43]. A large number of algorithms have been developed for solving different problems for the class of chordal graphs. There are many situations where we would like to generate input instances to test these algorithms for chordal graphs. For instance, we proposed a 1-round algorithm for approximate point placement in the plane in an adversarial model where the distance query graph presented to the adversary is chordal (see chapter 6). Also, in one of our generation methods for strongly chordal graphs, we generate chordal graphs as an intermediate step (see chapter 4). Chapter 2 deals with generating methods for chordal graphs.

Weakly chordal graphs were introduced by Hayward [26] in 1985. The class of weakly chordal graphs contains the class of chordal graphs and the class of complements of weakly chordal graphs. A graph is chordal if it has no chordless cycles of size 4 or more, whereas a graph G is weakly chordal (or weakly triangulated) if neither G nor its complement $\overline{G}$ contains a chordless cycle of size 5 or more. As was done for chordal graphs, in [39], the authors showed how to generate all linear layouts of a weakly chordal graph. A generation mechanism for weakly chordal graphs can be used to obtain test instances for this latter algorithm. However, until the work reported in this dissertation, no algorithm was known for generating weakly chordal graphs, exploiting their structural properties. In chapter 3, we present algorithms for generating weakly chordal graphs.

Strongly chordal graphs, introduced by Farber [22], are a proper subclass of the class of chordal graphs. As such, among many definitions of a strongly chordal graph the following has a more intuitive connection with the parent class of chordal graphs that have no induced cycle of size greater than 3. A graph G is strongly chordal if it is chordal and every even cycle of length 6 or more has a strong chord, that is, a chord that divides such an even cycle into two paths of odd-length. The interest in this subclass stems from the fact that many algorithmic problems are NP-complete (such as INDEPENDENT SET, CLIQUE COLORING, CLIQUE COVER, DOMINATING SET, and STEINER TREE, etc.) for chordal graphs are solvable in polynomial time for this subclass. For example, the k-tuple domination problem for strongly chordal graphs can be solved in linear-time if a strong ordering is provided [34]. While there are algorithms to recognize strongly chordal graphs, we did not find any in the literature that can generate these. In chapter 4, we propose algorithms for generating strongly chordal graphs. Next, in chapter 5, we present semi-dynamic algorithms for strongly chordal graphs under deletions and insertions of edges.

In chapter 6, we discuss the point placement problem in the plane, which is a special case of the graph embedding problem of Saxe [49]: For a given incomplete edge-weighted graph G and a parameter k, the problem is to decide if there is a mapping of the vertices of G to points in a Euclidean k-space such that any two vertices of G, connected by an edge, are mapped to points, whose Euclidean distance is equal to the weight of the edge. Saxe showed that the problem is strongly NP-complete, even when $k = 1$. In our version of the problem, for a given set of points, a distance query graph is generated and submitted to the adversary where the adversary is the source of true distances. When the distance query graph is chordal, then there exists a sequence of chordal graphs such that each intermediate graph is obtained by adding exactly one new edge to the immediately previous graph by using a distance matrix completion algorithm. After completion of the distance matrix, we can compute the locations of the points.

In the next section, we collect in one place some common graph-theoretic terminology used throughout the dissertation. In section 1.2, we describe all the obtained results of this dissertation.

1.1 Preliminaries

Let $G = (V, E)$ be an undirected graph with $n(= |V|)$ vertices and $m(= |E|)$ edges. The neighborhood $N(v)$ of a vertex v is the subset of vertices $\{u \in V \mid \{u, v\} \in E\}$ of V. The closed neighborhood $N[v]$ of a vertex v is the set $N(v) \cup \{v\}$. Figure 1.1 shows a graph where $n = 5$, $m = 7$, and the neighbors of v_3 is $N(v_3) = \{v_1, v_4\}$, while the closed neighborhood of v_3 is $N[v_3] = \{v_1, v_4, v_3\}$. For any vertex set $S \subseteq V$ and the edge set $E(S) \subseteq E$ where

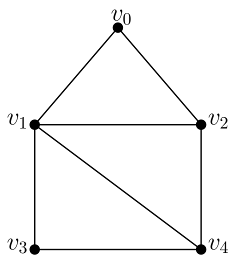


Figure 1.1: *A graph with* 5 *vertices and* 7 *edges*

$E(S) = \{\{u, v\} \in E \mid u \in S$ and $v \in S\}$, let $G[S]$ denote the subgraph of G induced by S, namely the subgraph $(S, E(S))$. In other words, an induced subgraph is a subset of the vertices of a graph G together with any edges whose endpoints are both in this subset. In Fig. 1.1, if $S = \{v_1, v_3, v_4\}$, then $G[S]$ is the induced subgraph containing the edges $E(S) = \{\{v_1, v_3\}, \{v_1, v_4\}, \{v_3, v_4\}\}$.

A path in a graph G is a sequence of vertices $[v_i, v_{i+1}, \ldots, v_k]$, where $\{v_j, v_{j+1}\}$ for $j = i, i+1, \ldots, k-1$, is an edge of G. The first vertex is known as the start vertex, the last vertex is called the end vertex, and the remaining vertices in the path are known as internal vertices. A cycle is a closed path where the start vertex and the end vertex coincide. The size of a cycle is the number of edges in it. A clique in G is a subset of vertices ($S \subseteq V$) of G, where its induced subgraph $G[S]$ is complete. A maximal clique is a clique that cannot be extended by including one more adjacent vertex. A graph on n vertices that forms a clique on n vertices is called a complete graph.

1.2 Obtained Results of this Dissertation

In this section, we describe the results obtained in this dissertation.

Chordal Graph Generation and Maintenance. We propose unified methods (Unified-Deletion and Unified-Insertion) for the generation of chordal graphs. The unified methods take the number of vertices and the number of edges as input and produce chordal graphs by maintaining a clique tree. The algorithms unify the insertion or deletion of edges in a chordal graph by maintaining the clique tree following a dynamic algorithm for chordal graphs by Ibarra [30]. The main advantage of the proposed methods that they can generate a connected chordal graph for the exact number of vertices and edges. Unified-Insertion is suitable to generate sparse chordal graphs and Unified-Deletion is suitable to generate dense chordal graphs. We propose a method for generating a chordal graph from an arbitrary graph based on a theorem by Dirac [18]. This generation method exploits the fact that chordal graphs can be generated by taking their unions. We present a semi-dynamic algorithm for chordal graphs under edge-deletions. This algorithm only requires maintenance of the basic adjacency matrix data structure.

Weakly Chordal Graph Generation. We describe a separator based method for generating weakly chordal graphs. The proposed method allows us to exploit the structural properties of a weakly chordal graph. An additional feature of our algorithm is that an added edge can be a non-two-pair edge. This result has been accepted to publish in *Discrete Mathematics, Algorithms and Applications* [45]. We present an algorithm for generating a weakly chordal graph from an arbitrary input graph. The algorithm is based on the theorem for recognizing weakly chordal graphs due to Berry et al. [9]. This problem was listed as an open problem in [9].

Strongly Chordal Graph Generation and Maintenance. We discuss three strongly chordal graph generation algorithms based on three different characterizations namely, totally balanced matrices [22], forbidden subgraph [22], and intersection graph [21] characterization. To the best of knowledge, these are completely new results. We propose semi-dynamic algorithms for deletions and insertions of edges into a strongly chordal graph. The proposed semi-dynamic algorithms are based on two different characterizations of strongly chordal graphs. The deletion algorithm is based on a strong chord characterization, while the insertion algorithm is based on a totally balanced matrix characterization.

Chordal Graphs and Point Placement in the Plane. As an application of chordal graphs, we propose a 1-round algorithm for approximate point placement in the plane in an adversarial model where the distance query graph presented to the adversary is chordal. The remaining distances are determined using a distance matrix completion algorithm for chordal graphs, based on a result by Bakonyi and Johnson [6]. The layout of the points is determined from the complete distance matrix using the traditional Young-Householder approach [57]. We show how the fact that chordal graphs form a completion class can be used to solve this point placement problem. To the best of our knowledge, such a connection has not been exploited before. This result have been published in Proceedings of the 17th *International Conference on Computational Science and Its Applications (ICCSA)* [46]. An extended version of this work (the extended part is not included in this dissertation) has been published in *Transactions on Computational Science* [44].

1.3 Organization of this Dissertation

The remainder of this dissertation is organized as follows.

In chapter 2, we discuss methods for generating chordal graphs and a semi-dynamic algorithm under for chordal graphs under edge-deletions. Chapter 3 describes generation methods for weakly chordal graphs. In chapter 4, we present methods for generating strongly chordal graphs. Chapter 5 focuses on semi-dynamic algorithms for deletions and insertions of edges into a strongly chordal graph. In chapter 6, we introduce a 1-round algorithm for point placement in the plane in an adversarial model where the distance query graph presented to the adversary is chordal. Finally, chapter 7 highlights the contributions of this dissertation with discussions and provides direction for possible future work.

1.4 Summary

In this chapter, we stated and motivated the graph generation problems for different classes of graphs. This chapter also presented some common graph-theoretic terminologies.

Chapter 2

Chordal Graph Generation and Maintenance

Chordal graphs are one of the most studied classes of graphs. We start this chapter by defining chordal graphs with the well-known standard characterizations of chordal graphs. In section 2.2 of this chapter, we propose unified methods for the generation of chordal graphs. The unified methods take the number of vertices and the number of edges as input and produce chordal graphs by maintaining a clique tree. A third method for generating chordal graphs from an arbitrary graph is based on a modified version of an algorithm by Dirac [18]. The modified version introduces fewer edges compared to the algorithm by Dirac. This appears in section 2.3. In section 2.4, we propose a semi-dynamic algorithm for chordal graphs under edge-deletions. This algorithm only requires maintenance of the basic adjacency matrix data structure.

2.1 Preliminaries

In this section, we review different characterizations of chordal graphs and existing algorithms for generating them.

2.1.1 Chordal Graphs

A *chord* of a cycle in a graph $G = (V, E)$ is an edge joining two non-consecutive vertices. For instance, in the graph of Fig. 2.1, the edge between v_1 and v_4 is a chord of the cycle $\langle v_1, v_2, v_4, v_3 \rangle$. A graph G is said to be *chordal* if it has no induced chordless cycles of size 4 or more (see Fig. 2.1 for an example).

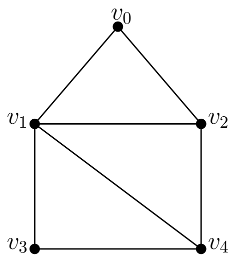


Figure 2.1: *A chordal graph*

2.1.2 Perfect Elimination Ordering

An *elimination ordering*, α of the vertices of G is a map $\alpha : \{1, 2, \ldots, n\} \rightarrow V$. Thus, $\alpha(i)$ is the i^{th} vertex in the elimination ordering and $\alpha^{-1}(v_i)$ is the index of v_i in α. A vertex v is said to be *simplicial* if $N(v)$ is a clique, that is, a complete subgraph on $N(v)$. The vertex v_3 is simplicial in Fig. 2.1 because $N(v_3) = \{v_1, v_4\}$ is a complete subgraph. The ordering α is a *perfect elimination ordering* (or *simplicial ordering*) if for $1 \leq i \leq n$, the vertex v_i is simplicial in the induced graph on the vertex set $\{v_i, v_{i+1}, \ldots, v_n\}$. For the chordal graph of Fig. 2.1, a perfect elimination ordering is: $\alpha(1) = v_3, \alpha(2) = v_4, \alpha(3) = v_2, \alpha(4) = v_1$ and $\alpha(5) = v_0$, obtained from the following label lists of the vertices over five steps using Lexicographic breadth-first search (Lex-BFS) [48] algorithm. The following

	v_0	v_1	v_2	v_3	v_4
Step 0	()	()	()	()	()
Step 1	()	(5)	(5)	()	()
Step 2	()	(5)	(5,4)	(4)	(4)
Step 3	()	(5)	(5,4)	(4)	(4,3)
Step 4	()	(5)	(5,4)	(4,2)	(4,3)

Lex-BFS algorithm is due to Rose et al. [48]. As shown by Dirac [18], every non-trivial

Algorithm 2.1 *Lex-BFS*

Input: A graph G

Output: A perfect elimination ordering α

```
1: Assign the empty label list, (), to each vertex in $V$
2: for $i \leftarrow n$ to 1 do
3:     Pick an unnumbered vertex $v \in V$ with the lexicographically largest label list
4:     Set $\alpha(i) \leftarrow v$
5:     For each unnumbered vertex $w$ adjacent to $v$, add $i$ to the label list of $w$
6: end for
7: return $\alpha$
```

chordal graph has at least two non-adjacent simplicial vertices. The following theorem due to Fulkerson and Gross states that chordal graphs are characterized by the existence of a perfect elimination ordering of its vertices.

Theorem 2.1 [23] *A graph G is chordal if and only if there exists a perfect elimination ordering of its vertices.*

2.1.3 Minimal Vertex Separators

A subset of vertices $S \subset V$ is a vertex separator of G if it separates the graph into two or more distinct connected components of $G[V - S]$. If S separates u and v into two components, then S is said to be a $u - v$ separator. If no proper subset of S is an $u - v$ separator, then S is a minimal $u - v$ separator. For the graph shown in Fig. 2.1, $\{v_1, v_4\}$ is a minimal $v_2 - v_3$ separator. The following theorem by Dirac [18] characterizes chordal graphs in terms of their minimal separators.

Theorem 2.2 [18] *A graph G is chordal if and only if every minimal vertex separator of G is complete in G.*

2.1.4 Weighted Clique Intersection Graphs

Let K_G denotes the set of all maximal cliques in G. The weighted clique intersection graph, denoted by W_{K_G}, has K_G as its vertex set. Two distinct maximal cliques K and K' are joined by an edge if $K \cap K' \neq \emptyset$ and the weight of this edge is set to $|K \cap K'|$. Figure 2.2(b) is the weighted clique intersection graph of Fig. 2.2(a). There are three maximal cliques $v_0v_1v_2$, $v_1v_2v_4$, and $v_1v_3v_4$ in G. The weight of each edge is also set appropriately. Note the W_{K_G} of a complete graph consists of a single vertex only because the only maximal clique is the graph itself.

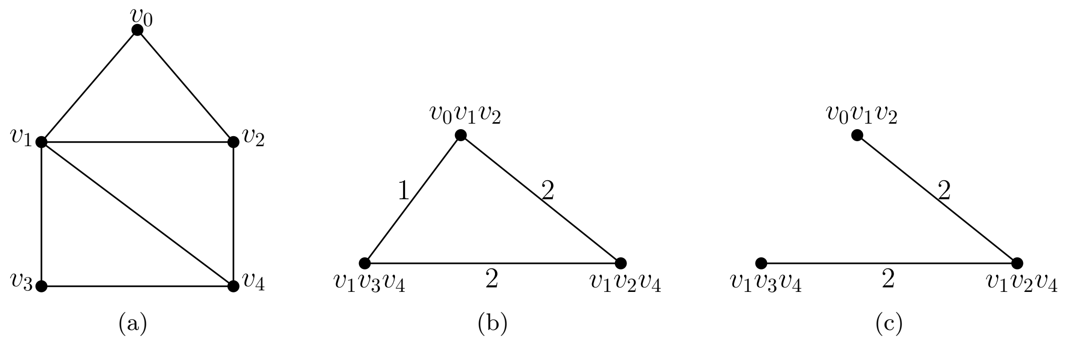


Figure 2.2: *(a) Chordal graph (G), (b) Weighted clique intersection graph (W_{K_G}), (c) Clique tree (T)*

2.1.5 Clique Trees

A clique tree T of a graph G is a spanning tree of W_{K_G} that satisfy the following clique-intersection property:

For every pair of distinct cliques $K, K' \in K_G$, the set $K \cap K'$ is contained in every clique on the path connecting K and K' in the tree.

Indeed, G is a chordal graph if and only if it has a clique tree. A greedy spanning tree algorithm (e.g., Kruskal's algorithm) can be used to construct a clique tree.

Note the clique tree T of W_{K_G} of a complete graph is a singleton node only. Figure 2.2(c) is a clique tree of the weighted clique intersection graph of Fig. 2.2(b). It is easy to verify that it satisfies the clique-intersection property.

Yannakakis and Tarjan [53] designed the well-known Maximum Cardinality Search (MCS) algorithm for ordering the vertices of a graph G that turns out to be the reverse of a Perfect Elimination Ordering (PEO) if the graph is chordal. At each step, the algorithm picks as the next vertex to number an unnumbered vertex adjacent to the largest number

```
$prev\_card \leftarrow 0$
$\mathcal{L}_{n+1} \leftarrow 0$
$s \leftarrow 0$
$\mathcal{E}_T \leftarrow 0$
for $i \leftarrow n$ to 1 do
    Choose a vertex $v \in V - \mathcal{L}_{i+1}$ for which $|adj(v) \cap \mathcal{L}_{i+1}|$ is maximum
    $\alpha(v) \leftarrow i$
    $new\_card \leftarrow |adj(v) \cap \mathcal{L}_{i+1}|$
    if $new\_card \leq prev\_card$ then
        $s \leftarrow s + 1$
        $K_s \leftarrow adj(v) \cap \mathcal{L}_{i+1}$
        if $new\_card \neq 0$ then
            $k \leftarrow \min\{j | v_j \in K_s\}$
            $p \leftarrow clique(v_k)$
            $\mathcal{E}_T \leftarrow \mathcal{E}_T \cup \{K_s, K_p\}$
        end if
    end if
    $clique(v_i) \leftarrow s$
    $K_s \leftarrow K_s \cup \{v_i\}$
    $\mathcal{L}_i \leftarrow \mathcal{L}_{i+1} \cup \{v_i\}$
    $prev\_card \leftarrow new\_card$
end for
```

Figure 2.3: *Expanded-MCS [11]*

of numbered vertices, with ties broken arbitrarily. The MCS algorithm runs in $O(n + m)$ time.

Blair and Peyton [11] present an efficient algorithm for computing a clique tree, which is an extension of the MCS algorithm. We used this expanded version of the MCS algorithm to obtain a clique tree from G in $O(n + m)$ time. This algorithm computes a clique tree T directly from G without generating W_{K_G} as an intermediate step. The algorithm is reproduced below.

2.1.6 Prior Work

In [3], the authors proposed an algorithm for generating chordal graphs by using a PEO. In each iteration, a vertex is chosen from the PEO, and the vertices adjacent to it in G are turned into a clique by inserting additional edges as needed.

In [36], Markenzon et al. proposed two methods for the generation of chordal graphs. The first method adds edges incrementally while maintaining chordality. The method is simple, dispensing with the need for any auxiliary data structure. The second method adds vertices incrementally while maintaining a perfect elimination ordering of the vertices and also a clique tree representation of the graph. The first method generates sparse graphs, while the second method generates dense ones. The first method makes crucial use of the following theorem.

Theorem 2.3 [36] *Let $G = (V, E)$ be a connected chordal graph and $u, v \in V$, $\{u, v\} \notin E$. The augmented graph $G + \{u, v\}$ is chordal if and only if $G[V - I_{u,v}]$ is not connected, where $I_{u,v} = N(u) \cap N(v)$.*

The number of vertices, n, and the number of edges, m, are the two inputs to this method. This method starts with the generation of a tree with the given number of vertices n. After the tree generation phase, $m - n + 1$ more edges are added. The algorithm adds a new edge, provided chordality is preserved.

The second method is based on the following lemma:

Lemma 2.4 [36] *Let $G = (V, E)$ be a chordal graph, $v \notin V$ a new vertex and $Q \subseteq V$ a clique of G. The graph $JOIN(G, Q, v) = (V \cup \{v\}, E \cup \{(v, x) | x \in Q\})$ is also chordal.*

This method takes the number of vertices and an upper bound of the number of edges as input. This algorithm makes the crucial use of a clique tree of a chordal graph. The generation method starts with a single vertex. Then, in every iteration, a new vertex is added and joined it to a clique. To increase the number of edges, the algorithm randomly picks maximal cliques and merges these to attain the specified upper bound on the number of edges.

Seker et al. [51] proposed an algorithm for the generation of "random" chordal graphs. The algorithm is based on the characterization that a graph G is chordal if and only if G is the intersection graph of subtrees of a tree. The algorithm starts by generating a random tree T on n nodes [47]. Next, it creates average k-sized n random subtrees of T. Then, G is produced as the intersection graph of the chosen subtrees. The algorithm takes n and k (a random integer) as input and produces a chordal graph G on n vertices. But the number of edges, m, in the resulting graph is not fixed for the same n and k. The authors claim that their generation algorithm can generate a "random" chordal graph, but no proof is given to support this claim other than an experimental analysis of the distribution of maximal cliques.

In section 2.2, we present unified methods for the generation of chordal graphs. In section 2.3, we propose a modified version of an algorithm by Dirac [18] to generate chordal graphs from arbitrary graphs. Section 2.4 explains the semi-dynamic algorithm for a chordal graph under edge-deletions.

2.2 Unified Chordal Graph Generation

We propose two algorithms for generating chordal graphs. Both algorithms take the number of vertices ($|V| = n$) and the number of edges ($|E| = m$) as input and produce a chordal graph as an output. One algorithm deletes edges from the initial graph (a complete graph) and the other algorithm adds edges to the initial graph (a tree). The first algorithm (Unified-Deletion) generates a complete graph on n vertices and then deletes edges from this complete graph until m edges are left. On the other hand, the second algorithm (Unified-Insertion) generates a random connected tree on n vertices and then adds edges till there are m edges. For both algorithms, we maintain a clique tree to enable the insertion or deletion of edges so that chordality is preserved. Finally, a chordal graph is generated from a clique tree. The algorithms unify the insertion or deletion of edges in a chordal graph by maintaining the clique tree following a dynamic algorithm for chordal graphs by Ibarra [30]. In sections 2.2.2 and 2.2.3, we present both methods (Unified-Deletion and Unified-Insertion) with algorithms and examples. One of the main advantages of the proposed unified algorithms is that chordal graphs can be generated for the exact number of vertices and edges. We also ran some experiments where we generated chordal graphs for different sets of vertices and edges and analyzed these with respect to different attributes (such as the number of max cliques, min/max/mean clique size, etc.).

The rest of the section is organized as follows. Section 2.2.1 explains dynamic deletion and insertion algorithms for chordal graphs with examples. In sections 2.2.2 and 2.2.3, we present unified methods for the generation of chordal graphs. Section 2.2.4 discusses the complexity with experimental results. Finally, section 2.2.5 contains concluding remarks and open problems.

2.2.1 Dynamic Maintenance of Chordal Graphs

The algorithm due to Ibarra [30] maintains a clique tree representation of a chordal graph G which it queries to determine if deleting ($G - \{u, v\}$) from or inserting ($G + \{u, v\}$) into G an arbitrary edge $\{u, v\}$ preserves chordality or not.

Deletion of an Edge from a Chordal Graph:

The deletion of an edge $\{u, v\}$ is decided based on the following theorem:

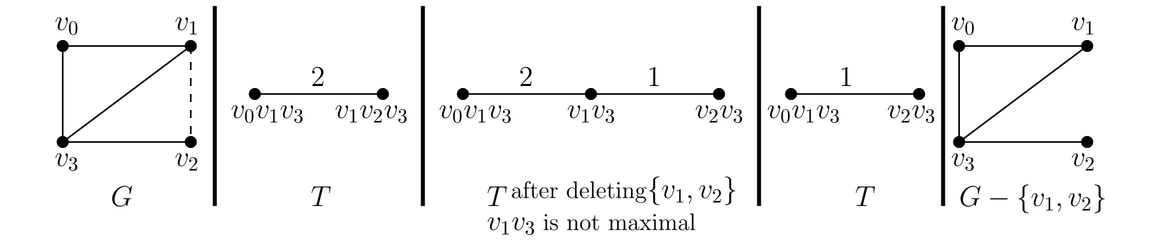
Theorem 2.5 [30] *Let G be a chordal graph with edge $\{u, v\}$. Then $G - \{u, v\}$ is chordal if and only if G has exactly one maximal clique containing $\{u, v\}$.*

Algorithm 2.2 *Delete-Query [30]*

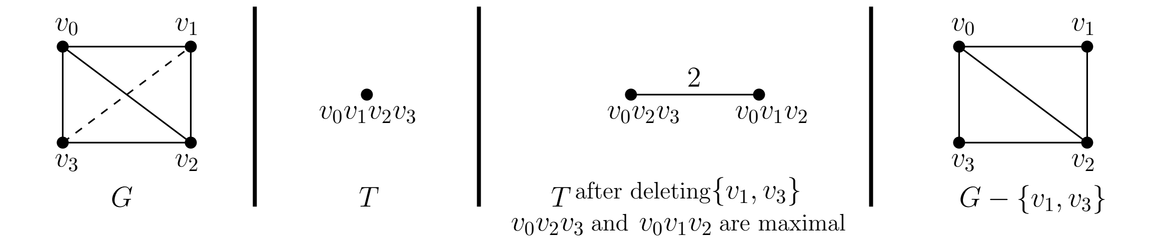
Input: A clique tree T of a chordal graph G and an edge $\{u, v\}$ to be deleted
Output: Return True or False

```
1: $canBeDeleted \leftarrow$ False
2: if the edge $\{u, v\}$ belongs to exactly one node ($x$) in $T$ then
3:     $canBeDeleted \leftarrow$ True
4:     return $canBeDeleted$ and the node $x$
5: else
6:     return $canBeDeleted$
7: end if
```

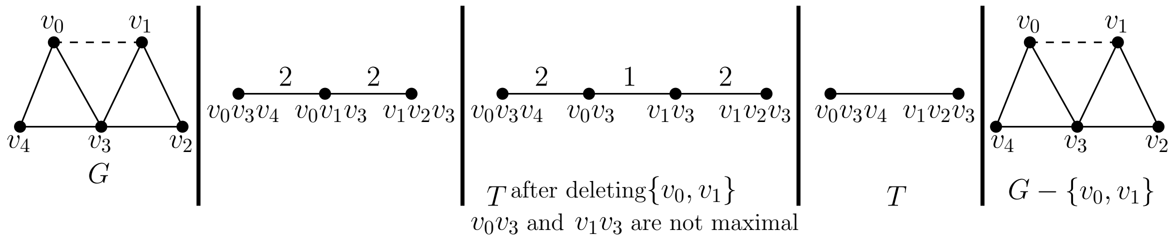
That is, an edge $\{u, v\}$ can be deleted if and only if a unique node in the clique tree containing the edge $\{u, v\}$. Then the delete operation is performed and the clique tree is updated. The node containing the edge $\{u, v\}$ can be replaced with 0, 1, or 2 nodes [30]. These situations are explained with examples in Fig. 2.4. Algorithm *Delete* deletes the edge $\{u, v\}$, and updates the clique tree.



(a)



(b)



v_0 v_1 v_4 v_3 v_2
G
2 2
$v_0v_3v_4$ $v_0v_1v_3$ $v_1v_2v_3$
2 1 2
$v_0v_3v_4$ v_0v_3 v_1v_3 $v_1v_2v_3$
T after deleting $\{v_0, v_1\}$
v_0v_3 and v_1v_3 are not maximal
$v_0v_3v_4$ $v_1v_2v_3$
T
v_0 v_1 v_4 v_3 v_2
$G - \{v_0, v_1\}$

(c)

Figure 2.4: *Different scenarios illustrated that the clique tree node containing the edge $\{u, v\}$ can be replaced with 0, 1, or 2 nodes.*

Insertion of an Edge into a Chordal Graph:

Whether an edge $\{u, v\}$ can be inserted or not is decided based on the following two theorems:

Algorithm 2.3 *Delete [30]*

Input: A clique tree T of a chordal graph G and an edge $\{u, v\}$ to be deleted
Output: An updated Clique tree after deleting $\{u, v\}$

```
 1: $deleted \leftarrow$ False
 2: if $Delete - Query(T, u, v)$ returns True then
 3:     For every $y \in N(x)$, test whether $u \in K_y$ or $v \in K_y$ and whether $w(x, y) = k - 1$ ▷ $w(x, y) = |K_x \cap K_y|$
 4:     Replace node $x$ with new nodes $x_1$ and $x_2$ respectively representing $K_x^u$ and $K_x^v$ and add edge $\{x_1, x_2\}$ with $w(x_1, x_2) = k - 2$.    ▷ $K_x^u = K_x - \{v\}$ and $K_x^v = K_x - \{u\}$
 5:     if $y \in N_u$ then    ▷ $N_u = \{y \in N(x)|u \in K_y\}$
 6:         replace $\{x, y\}$ with $\{x_1, y\}$
 7:     end if
 8:     if $z \in N_v$ then    ▷ $N_v = \{z \in N(x)|v \in K_z\}$ and $K_z$ is a maximal clique
 9:         replace $\{x, z\}$ with $\{x_2, z\}$
10:     end if
11:     if $w \in N_{\overline{uv}}$ then    ▷ $N_{\overline{uv}} = \{w \in N(x)|u, v \notin K_w\}$ and $K_w$ is a maximal clique
12:         replace $\{x, w\}$ with $\{x_1, w\}$ or $\{x_2, w\}$ (chosen arbitrarily)
13:     end if
14:     if $K_x^u$ and $K_x^v$ are both maximal in $G - \{u, v\}$ then
15:         return $deleted$ and the updated clique tree $T$
16:     end if
17:     if $K_x^u$ is not maximal because $K_x^u \subset K_{y_i}$ for some $y_i \in N_u$ then
18:         choose one such $y_i$ arbitrarily, contract $\{x_1, y_i\}$, and replace $x_1$ with $y_i$
19:     end if
20:     if $K_x^v$ is not maximal because $K_x^v \subset K_{z_i}$ for some $z_i \in N_v$ then
21:         choose one such $z_i$ arbitrarily, contract $\{x_2, z_i\}$, and replace $x_2$ with $z_i$
22:     end if
23:     $deleted \leftarrow$ True
24:     return $deleted$ and the updated clique tree $T$
25: end if
```

Theorem 2.6 [30] *Let G be a chordal graph without edge $\{u, v\}$. Then $G + \{u, v\}$ is chordal if and only if there exists a clique tree T of G such that $u \in K_x$, $v \in K_y$ for some $\{x, y\} \in T$, where K_x and K_y are two maximal cliques, x and y are two nodes in T, $\{x, y\} \in T$ represents an edge between x and y in T.*

Theorem 2.7 [30] *Let G be a chordal graph without edge $\{u, v\}$. Let T be a clique tree of G and let x, y be the closest nodes in T such that $u \in K_x$, $v \in K_y$. Assume $\{x, y\} \notin T$. There exists a clique tree T' of G with $u \in K_{x'}$, $v \in K_{y'}$ and $\{x', y'\} \in T'$ if and only if the* *minimum-weight edge e on the $x - y$ path in T satisfies $w(e) = w(x, y)$.*

That is, an edge $\{u, v\}$ can be inserted if $\{x, y\} \in T$ or there is a path between x and y in the clique tree such that the minimum-weight of an edge on the $x - y$ path is equal to the overlap of K_x and K_y. If one of these conditions is satisfied, then the edge $\{u, v\}$ can be inserted and the clique tree is updated. Theorem 2.7 is illustrated with an example in Fig. 2.5. Assume we want to insert an edge $\{v_3, v_5\}$, where the vertex v_3 is in maximal clique $v_2v_3v_6$ (say x) and the vertex v_5 is in maximal clique $v_5v_6v_7$ (say y). There is no edge between x and y in T but the minimum-weight edge e on the $x - y$ path in T is 1 which satisfies $w(e) = w(x, y)$. The updated clique tree and the resulting graph after inserting $\{v_3, v_5\}$ is shown in Figs. 2.5(c) and 2.5(d), respectively. In general, the nodes x and y can

Algorithm 2.4 *Insert-Query*

```
Input: A clique tree $T$ of a chordal graph $G$ without the edge $\{u, v\}$ to be inserted
Output: Return True or False
 1: $canBeInserted \leftarrow$ False
 2: Find the closest nodes $x, y \in T$ such that $u \in K_x$, $v \in K_y$
 3: if $\{x, y\} \in T$ then
 4:     $canBeInserted \leftarrow$ True
 5:     return canBeInserted and the nodes $x, y$
 6: else
 7:     Find the minimum weight edge $e$ on the $x - y$ path in $T$
 8:     if $w(e) == w(x, y)$ then
 9:         $canBeInserted \leftarrow$ True
10:         return $canBeInserted$ and the nodes $x, y$
11:     else if $w(e) > w(x, y)$ then
12:         $canBeInserted \leftarrow$ False
13:         return $canBeInserted$
14:     end if
15:     return $canBeInserted$
16: end if
```

be replaced with 1, 2, or 3 nodes [30]. These situations are explained with examples in

Fig. 2.6. Algorithm *Insert* inserts the edge $\{u, v\}$ and updates the clique tree.

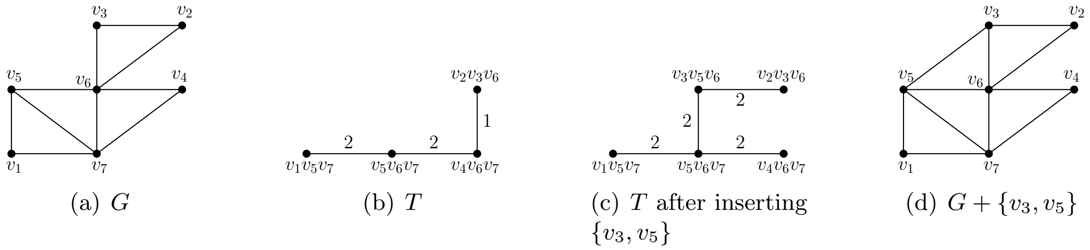


Figure 2.5: *Illustration of theorem 2.7*

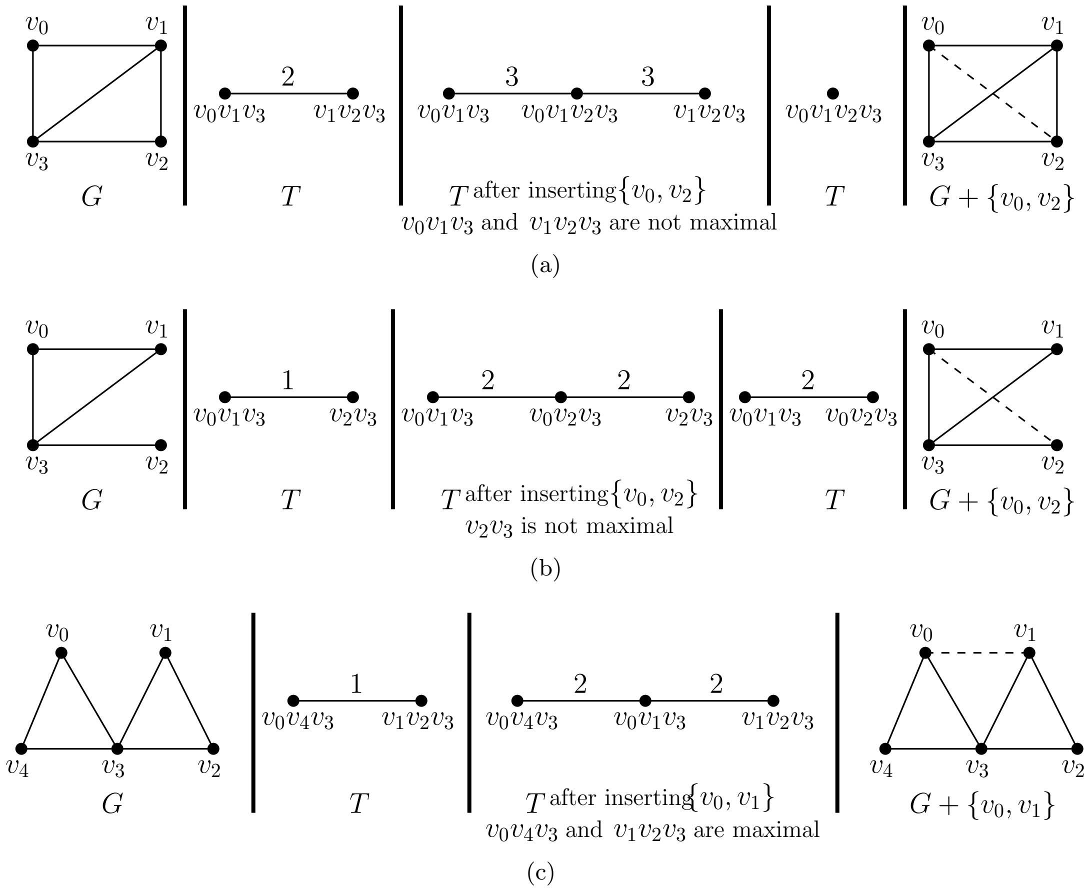


Figure 2.6: *Different scenarios illustrated that the nodes x and y can be replaced with* 1, 2, *or* 3 *nodes.*

The query operations for checking an edge can be deleted or inserted run in $O(n)$ time, and the deletion and insertion operations also run in $O(n)$ time, where n is the number of vertices.

Algorithm 2.5 *Insert [30]*

```
Input: A clique tree $T$ of a chordal graph $G$ and an edge $\{u, v\}$ to be inserted
Output: An updated Clique tree after inserting $\{u, v\}$
1: $inserted \leftarrow$ False
2: if $Insert - Query(T, u, v)$ returns "True" then
3:     Replace edge $\{x, y\}$ in $T$ with new node $z$ representing $K_z = I \cup \{u, v\}$ and add edges $\{x, z\}$, $\{y, z\}$, each with weight $|I| + 1$ where $I = K_x \cap K_y$
4:     Determine whether $K_x$, $K_y$ are maximal in $G + \{u, v\}$ by comparing $|K_x|$, $|K_y|$, and $w(x, y)$
5:     if $K_x^u$ and $K_x^v$ are both maximal in $G + \{u, v\}$ then
6:         return $inserted$ and the updated clique tree $T$
7:     end if
8:     if $K_x$ is not maximal then
9:         contract $\{x, z\}$ and replace $x$ with $z$
10:    end if
11:    if $K_y$ is not maximal then
12:        contract $\{y, z\}$ and replace $y$ with $z$
13:    end if
14:    $inserted \leftarrow$ True
15:    return $inserted$ and the updated clique tree $T$
16: end if
```

2.2.2 First (Unified-Deletion) Method

The Unified-Deletion algorithm starts by generating a complete graph on n vertices. The next step is to generate a clique tree (T) from the complete graph. Initially, the clique tree (T) consists of a single node. We update T after every deletion. The algorithm iterates $m' - m$ times (where $m' = \frac{n(n-1)}{2}$ is the number of edges in a complete graph) and deletes edges until m edges are left. Whether an edge $\{u, v\}$ can be deleted or not is decided based on the *Delete-Query* algorithm, discussed in the previous section. If *Delete-Query* algorithm returns "True" and the node x from the clique tree, then the *Delete* algorithm deletes the edge $\{u, v\}$ and updates the clique tree. In the final step, we construct a chordal graph (G) from the clique tree (T).

Algorithm 2.6 *Unified-Deletion*

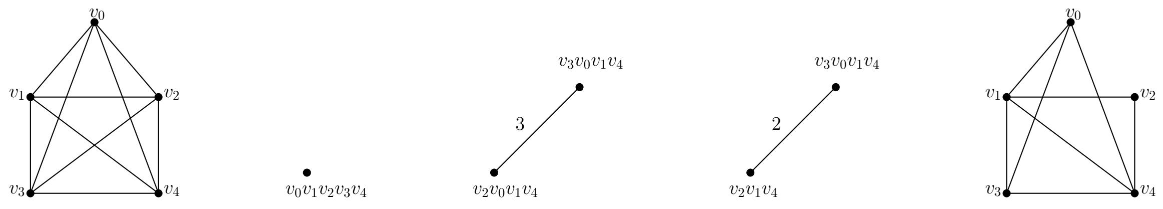
```
Input: The number of vertices $n$ and the number of edges $m$
Output: A chordal graph ($G$)
 1: Generates a complete graph ($G$) on the given $n$
 2: Generates a clique tree ($T$) from $G$
 3: $p \leftarrow 0$; $q \leftarrow m' - m$          $\triangleright$ $m' = \frac{n(n-1)}{2}$ (no. of edges in the complete graph)
 4: while $p < q$ do
 5:     Choose a pair of vertices $u$ and $v$ at random
 6:     if the edge $\{u, v\}$ does not exist then
 7:         continue
 8:     else
 9:         if $Delete(T, u, v)$ returns True then
10:             $p \leftarrow p + 1$
11:         end if
12:     end if
13: end while
14: Construct a chordal graph ($G$) from the clique tree ($T$)
```

When we choose an edge $\{u, v\}$ to delete from G, a failure may occur. That is, a random edge is picked for deletion but the deletion of this edge violates the chordality property and the algorithm *Delete-Query* returns "False". Another edge $\{u, v\}$ is chosen at random until an edge is found that can be deleted. This may require several unsuccessful trials. To avoid picking an edge that is already deleted, we maintain two different lists. One list contains the candidate edges for deletion and the other list includes the edges which have been previously deleted.

Since we are interested in generating connected chordal graphs, if there is a maximal clique containing one edge only, this edge is not deleted even if the conditions for deletion are satisfied. This is another reason for an edge to be not picked for deletion. The *Unified-Deletion* method is suitable for generating dense chordal graphs. We use the running intersection property [11] to obtain the connected chordal graph from the clique tree.

Running Intersection Property [11] *A total ordering of the cliques in K_G, say $K_1, K_2, \ldots K_m$, has the running intersection property (RIP) if for each clique K_j, $2 \leq j \leq m$, there exists a clique K_i, $1 \leq i \leq j-1$, such that $K_j \cap (K_1 \cup K_2 \cup \cdots \cup K_{j-1}) \subset K_i$.*

Markenzon et al. [36] showed how to construct the adjacency list for G in $O(m)$ time by iterating over all the nodes in the clique tree (T), where m is the number of edges in the generated final chordal graph. When processing a node in the clique tree, we partition the vertices into two sublists: a list with old vertices and another list with new vertices. The new list contains all the new vertices and a complete subgraph is created on this set. Another set of edges is introduced from the vertices of an old list to the vertices of a new list.



(a) G (b) T (c) T' (d) T'' (e) G

Figure 2.7: *Chordal graph generation: Unified-Deletion*

Figure 2.7 shows an example of generating a chordal graph with 5 vertices and 8 edges. The first two steps of the *Unified-Deletion* algorithm are shown in the first two Figs. 2.7(a) and 2.7(b). The clique trees after deleting $\{v_2, v_3\}$ and $\{v_2, v_0\}$ are shown in Figs. 2.7(c) and 2.7(d), respectively. Figure 2.7(e) shows the generated chordal graph with 5 vertices and 8 edges.

2.2.3 Second (Unified-Insertion) Method

The *Unified-Insertion* algorithm works in a similar way as the *Unified-Deletion* algorithm. The main differences here we start with a tree and then add edges to the tree. A tree on n vertices can be generated in $O(n)$ time using Prüfer coding [42] or by starting with a tree containing a single node and then add a new node by making it adjacent to one of the existing nodes in the tree, chosen at random [47]. Here we generate tree on n nodes as follows: starting with a single node, we add new ones either by splitting an edge into two or joining a new node to an existing node, chosen at random. The second step of the algorithm generates a clique tree (T). In the next step, the *Insert* algorithm adds an edge if the *Insert-Query* algorithm returns "True". The *while* loop iterates $m-n+1$ times, adding as many edges.

Algorithm 2.7 *Unified-Insertion*

```
Input: The number of vertices $n$ and the number of edges $m$
Output: A chordal graph ($G$)
 1: Generates a tree graph ($G$) on the given $n$
 2: Generates a clique tree ($T$) from $G$
 3: $p \leftarrow 0$; $q \leftarrow m - m'$          ▷ $m' = n - 1$ (the no. of edges in the tree graph ($G$))
 4: while $p < q$ do
 5:     Choose a pair of vertices $u$ and $v$ at random
 6:     if the edge $\{u, v\}$ already exists then
 7:         continue
 8:     else
 9:         if $Insert(T, u, v)$ returns True then
10:             $p \leftarrow p + 1$
11:         end if
12:     end if
13: end while
14: Construct a chordal graph ($G$) from the clique tree ($T$)
```

When we choose an edge for insertion in G, a failure may occur. That is, a random

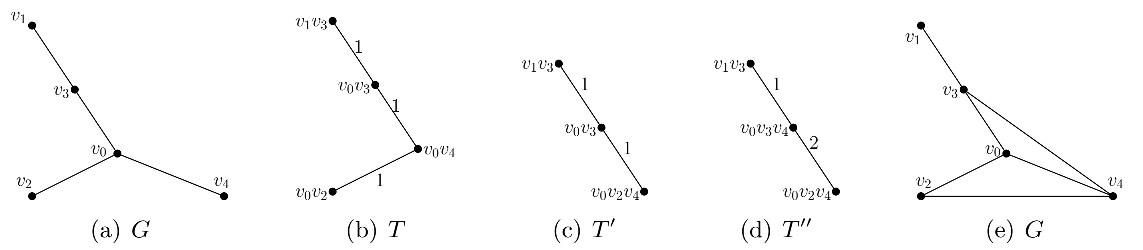


Figure 2.8: *Chordal graph generations: Unified-Insertion*

edge is picked for insertion but the insertion of this new edge violates chordality and *Insert-Query* returns "False". Another edge $\{u, v\}$ is chosen at random until there is an edge to be inserted. This may require many trials to pick an edge for insertion. To avoid picking an edge that already exists, we compute $K_x \setminus \{K_x \cap K_y\}$ and $K_y \setminus \{K_x \cap K_y\}$ and then choose a vertex from each set for insertion. It is evident that we will always find an edge to insert. The last step is to construct a chordal graph (G) from the clique tree (T) by applying the running intersection property [11] as stated before.

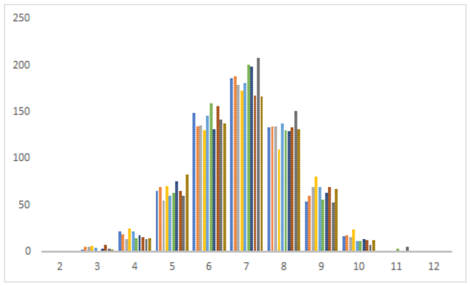
Figure 2.8 illustrates the *Unified-Insertion* method, adding two edges ($\{v_2, v_4\}$ and $\{v_3, v_4\}$), in that order, into the graph. The resulting chordal graph is shown in 2.8(e).

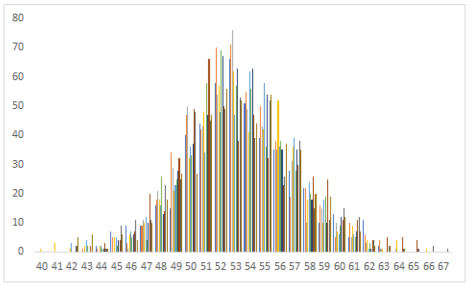
2.2.4 Complexity and Experimental Results

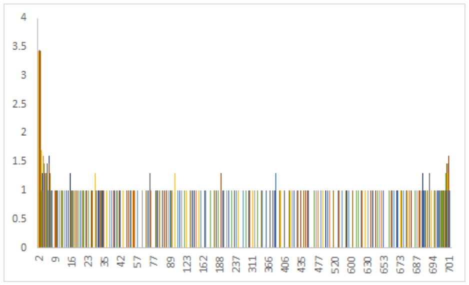
For the *Unified-Insertion* approach, the complexity of generating a tree is $O(n)$. The computation of building a clique tree requires $O(n+m')$ time, where n is the number of vertices and m' is the number of edges in the tree. Each *Insert-Query*, *Insert*, *Delete-Query*, and *Delete* operations can be performed in $O(n)$ time. Since we pick edge in random, the insertion/deletion of an edge may violate the chordality. To insert $m - m'$ edges, we may query $m - m' + k$ times, where $m' = n - 1$ and k is the number of unsuccessful trials. An upper bound on the *Insert-Query* operations is $(m - m' + k)O(n)$. Similarly, an upper bound on the *Delete-Query* operations is $(m' - m + k)O(n)$, where $m' = \binom{n}{2}$ and k is the number of unsuccessful trials. The construction of a chordal graph from a clique tree takes $O(m)$ time, where m is the number of edges in the generated final chordal graph.

Table 2.1: *Experimental Results of Unified Methods*

n	m	Method	# Conn. Comp.s	# Maximal Cliques	# Min Clique Size	# Max Clique Size	# Mean Clique Size	# Sd of Clique Sizes
1000	5647	Insert	1	627	2	10	6.93	1.36
	50375	Insert	1	484	37	65	53.29	4.39
	252238	Delete	1	302	2	699	30.75	125.45
	399907	Delete	1	117	2	884	90.77	236.22
2500	35290	Insert	1	1442	9	22	15.53	2.21
	322434	Insert	1	1220	116	151	134.11	5.80
	1572067	Delete	1	739	2	1762	31.29	202.97
	2509819	Delete	1	272	2	2229	107.40	430.21


(a) $n = 1000, m = 5647$


(b) $n = 1000, m = 50375$


(c) $n = 1000, m = 252238$

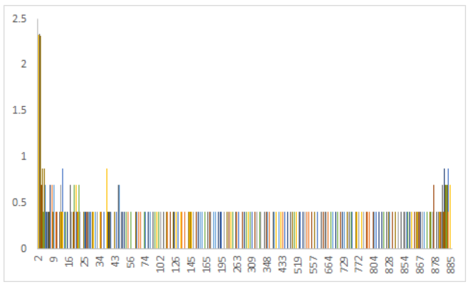

(d) $n = 1000, m = 399907$

Figure 2.9: *Maximal clique distributions*

Table 2.1 shows experimental results of the unified chordal graph generation methods where the first two columns represent the number of vertices and the number of edges, respectively. The third column represents the selected method. Column 4 represents the number of connected components in the resulting chordal graph and column 5 represents the number of maximal cliques. Column 6-8 represents the minimum, maximum, and mean clique size. The last column shows the standard deviation of clique sizes. We ran experiments on two different sets of vertices with a different number of edges. We choose a method, based on the minimal number of edges needed to insert or delete to reach m.

Seker et al. [51] studied the distribution of maximal cliques to show the varieties of the generated chordal graphs. We also tried to do a similar study here to understand whether the distribution of cliques tells anything about the randomness of the generated chordal graph. Figure 2.9 shows the distribution of maximal cliques, where the maximal clique distributions in Figs. 2.9(a) and 2.9(b) demonstrate output graphs contain maximal cliques of many different sizes. We observe that the medium-size maximal cliques become visible relative to the small and large maximal cliques. In the other two cases (delete), it seems that we have more small maximal cliques with few large maximal cliques than the other maximal cliques. An interesting open problem would be to generate chordal graphs uniformly at random. There are published works on the uniform generation of random regular graphs ([37, 55]). A polynomial-time algorithm is presented for the fast uniform generation of regular graphs by Jerrum and Sinclair [31].

2.2.5 Discussion

We proposed unified methods to generate chordal graphs. These methods used the clique tree data structure for chordal graphs. The first method (*Unified-Deletion*) starts with a complete graph and is more suitable for generating dense chordal graphs. On the other hand, the second method (*Unified-Insertion*) is more suitable for generating sparse chordal graphs. One of the advantages of the proposed methods that they can generate a connected chordal graph for the exact number of vertices and edges.

2.3 k-chromatic Chordal Graph Generation

In this section, we present a modified version of an algorithm by Dirac [18] for generating k-chromatic chordal graphs. The parameter k is bounded below by the clique size and upper bounded by $|V| - |M| + 1$ of the generated graph, where $|V|$ is the number of vertices in the graph and $|M|$ is the size of a maximal set of independent vertices.

The following subsection defines some terminologies. In subsection 2.3.2, we present our algorithm. Finally, section 2.3.3 contains concluding remarks.

2.3.1 Definitions

A graph $H = (V, E + F)$ is called a triangulation of $G = (V, E)$ if H is chordal. A triangulation $H = (V, E + F)$ of $G = (V, E)$ is minimal if and only if the removal of any single fill edge from H results in a non-chordal graph [48]. An independent vertex set of a graph G is a subset of vertices that have no edges between them. An independent vertex set is maximal if it is not a subset of any other independent vertex set. A graph coloring is an assignment of colors to the vertices of a graph G such that no two adjacent vertices have the same color. If a coloring uses at most k-colors, it is known as a k-coloring and the minimum number of colors needed to color the vertices in G is the chromatic number of the graph. A graph that can be assigned a k-coloring is k-colorable, and it is k-chromatic if its chromatic number is exactly k.

2.3.2 The Algorithm

In this section, we present an algorithm for the construction of k-chromatic chordal graphs. The algorithm is based on the following theorem due to Dirac:

Theorem 2.8 [18] *From any graph with n vertices which contains α mutually independent vertices, it is always possible to obtain a $(n-\alpha+1)$-colorable rigid circuit graph by adding edges.*

Algorithm 2.8 *k-chromaticChordalGraph*

```
Input: An arbitrary graph $G = (V, E)$
Output: A chordal graph $G$
1: Find a maximal set of $\alpha$ mutually independent vertices ($M$) from arbitrary graph
2: $n' = n \setminus M$        ▷ $n'$ represents the set of not mutually independent vertices
3: Create complete subgraph on $n'$
```

Algorithm 2.8 takes an arbitrary graph as input and turns it into a chordal graph. To generate an arbitrary graph, we used the algorithm called 'dense_gnm_random_graph' method by Keith M. Briggs, which is inspired by Knuth's Algorithm S (Selection sampling technique), in section 3.4.2 of [32]. This random graph generation method takes the number of vertices (n) and the number of edges (m) as input and produces a random graph. For the given n, we compute m as a random value lying in the range between $n-1$ and $\frac{n(n-1)}{2}$. Then we pass n and m to the 'dense_gnm_random_graph' method. The resulting graph may not be connected and is considered as input to the Algorithm 2.8.

The next step is to find a mutually independent set of vertices (M) of the arbitrary graph. We used the approximation algorithm by Boppana and Halldórsson [13] to find M. We get n' by subtracting M from n. Now we create a complete subgraph on n' by making it into a clique. The resulting graph is chordal because of the following theorem due to Dirac:

Theorem 2.9 [18] *If n_1 and n_2 are rigid circuit graphs and $n_1 \cap n_2$ is a clique or empty, then $n_1 \cup n_2$ is a rigid circuit graph.*

Algorithm 2.9 *LB-Triang [8]*

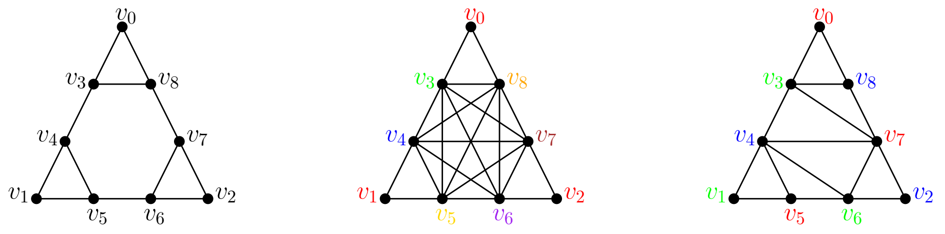
```
Input: An arbitrary graph $G = (V, E)$
Output: A minimal fill-in $F$ of $G$, A minimal triangulation $H = (V, E + F)$ of $G$.
1: Choose an arbitrary order $\sigma$ of $V$
2: for each vertex $x$ in $V$ taken in the order $\sigma$ do
3:     Compute $N[x]$
4:     if $N[x] \neq V$ then
5:         Compute the set of connected components $\mathcal{C}_G(N_H[x])$
6:         for each connected component $C$ in $\mathcal{C}_G(N_H[x])$ do
7:             Create complete subgraph on $N_G(C)$
8:         end for
9:     end if
10: end for
```

Dirac's method introduces more edges than required to turn the graph induced by n' into a chordal graph. To obtain a sparser chordal graph, we apply the LB-Triang algorithm proposed by Anne Berry in [8] to the graph induced by n'. Before using the LB-Triang algorithm on n', we make the neighborhood of each vertex in M into a clique. Note that

Dirac's theorem still holds as the neighborhood of each vertex in the independent set being an induced subgraph of a chordal graph is also chordal. The LB-Triang algorithm described below produces a minimal triangulation on n'.

Algorithm 2.10 *k-chromaticChordalGraphModified*

```
Input: An arbitrary graph $G = (V, E)$
Output: A chordal graph $G$
1: Find a maximal set of $\alpha$ mutually independent vertices ($M$) from arbitrary graph
2: for each vertex $\alpha$ in $M$ do
3:     Compute $N[\alpha]$
4:     Make the $N[\alpha]$ a clique
5: end for
6: $n' = n \setminus M$          ▷ $n'$ represents the set of not mutually independent vertices
7: Call LB-Triang on the graph induced by $n'$
```



(a) Given graph (G)

(b) Chordal graph (G) (using algorithm 2.8)

(c) Chordal graph (G) (using algorithm 2.10)

Figure 2.10: *Construction of k-chromatic chordal graphs*

The modified version of the Algorithm 2.8 is given in Algorithm 2.10, where we made the neighborhood of each α a clique and then we applied LB-Triang on the graph induced by n'. Figure 2.10 shows an example of a chordal graph generation using both algorithm 2.8 and algorithm 2.10. Assume $\{v_0, v_1, v_2\}$ are the mutually independent set of vertices. Algorithm 2.8 creates a complete subgraph on the set of vertices $\{v_3, v_4, v_5, v_6, v_7, v_8\}$. On the other hand, Algorithm 2.10 introduces three edges $\{v_3, v_7\}$, $\{v_4, v_6\}$, and $\{v_4, v_7\}$ only by applying LB-Triang algorithm 2.9 on the same set of vertices (n'). The neighborhood of

$\{v_0, v_1, v_2\}$ is already a clique. We can color the mutual set of vertices using the same color but we need different colors for the non-mutual set of vertices if we apply algorithm 2.8 because the non-mutual set of vertices forms a clique. Thus the chordal graph produced by algorithm 2.8 is $(n - \alpha + 1)$ colorable. The chordal graph obtained by algorithm 2.10 is also $(n - \alpha + 1)$ colorable but we may need fewer colors if the induced graph on the set of non-mutual vertices is not complete. In the worst case, we can find the maximum independent set in $O(n/(\log n)^2)$ time [13]. The time requires for the set difference is linear in the sizes of the two sets and is thus bounded by $O(n)$. The complexity of LB-Triang is $O(nm)$ [8].

2.3.3 Discussion

We modified the k-chromatic chordal graph generation algorithm proposed by Dirac. For the same set of vertices, we can generate chordal graphs by introducing fewer edges. This is achieved by making the neighborhoods of a mutually independent set of vertices into cliques and applying the LB-Triang algorithm on the non-mutual set of vertices of the given graph G. The resulting chordal graph is still k-colorable but may require fewer colors. An interesting open problem would be to compare the varieties of the chordal graphs generated from the original method and the modified method and also to find the exact number of colors required to color the resulting chordal graphs.

2.4 Semi-dynamic Algorithm for Chordal Graphs

In this section, we propose a semi-dynamic algorithm for edge-deletions in a chordal graph. For a given chordal graph (G), the method takes an edge $\{u, v\}$ to be deleted as an input. In [30], the author proposed a fully dynamic algorithm for chordal graphs by using a clique tree as an auxiliary data structure. But the semi-dynamic algorithm proposed here for edge-deletions does not require us to maintain any additional data structure besides an adjacency matrix representation of G. In the following subsections, we describe our algorithm and support it with examples.

2.4.1 Semi-dynamic Algorithm for Deletions

Let $G = (V, E)$ be a chordal graph and $e = \{u, v\}$ be an arbitrary edge of G. To reiterate, a graph G is said to be chordal if it has no induced chordless cycles of size 4 or more. The edge e can be deleted if and only it is not the only chord of a 4-cycle. Since the addition of a chord splits a cycle of size 4 into two P_3-paths (each spanning three vertices), it is sufficient to check for the presence of a chord in every cycle of length 4.

A potential 4-cycle of which $e = \{u, v\}$ is a chord is formed by disjoint pairs of chordless P_3-paths that go from u to v. Thus we determine all such P_3-paths and for every disjoint pair of these, we check whether $\{u, v\}$ is the only chord or not. If there is a chord other than $\{u, v\}$ in every disjoint pair of P_3-paths, then the edge $\{u, v\}$ can be deleted. On the other hand, if $\{u, v\}$ is the only chord for any disjoint pair of P_3-paths, then the edge $\{u, v\}$

cannot be deleted. To find all P_3-paths from u to v, we compute $N(u) \cap N(v)$.

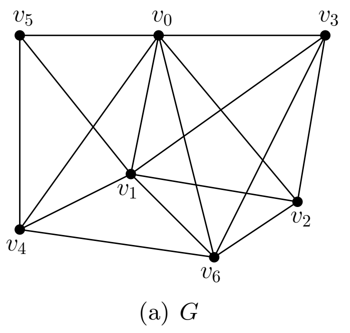
Algorithm 2.11 *Delete*

```
Input: A chordal graph $G$ and an edge $\{u, v\}$ to be deleted
Output: A chordal graph $G - \{u, v\}$
1: if $Delete - Query(G, u, v)$ returns "True" then
2:     Delete the edge $\{u, v\}$ from $G$
3: end if
```

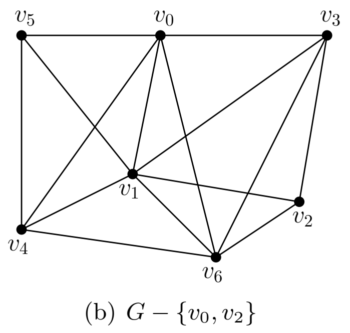
Algorithm 2.12 *Delete-Query*

```
Input: A chordal graph $G$ and an edge $\{u, v\}$ to be deleted
Output: Return True or False
1: $canBeDeleted \leftarrow$ False
2: if the edge $\{u, v\}$ does not exist then
3:     return $canBeDeleted$
4: else
5:     if $\{u, v\}$ is not the only chord then
6:         $canBeDeleted \leftarrow$ True
7:         return $canBeDeleted$
8:     else
9:         return $canBeDeleted$
10:    end if
11: end if
```

Algorithm 2.12 returns "True", if $\{u, v\}$ can be deleted from G. When the algorithm returns "True", we perform the delete operation (see algorithm 2.11).



(a) G



(b) $G - \{v_0, v_2\}$

Figure 2.11: *An example*

Consider the chordal graph G shown in Fig. 2.11. Can we delete the edge $\{v_1, v_6\}$? Now we check the deletion of $\{v_1, v_6\}$ preserves the chordality property or not. To do that, we compute the neighborhood of v_1 and v_6 and then compute $N(v_1) \cap N(v_6) = \{v_0, v_2, v_3, v_4, v_5, v_6\} \cap \{v_0, v_1, v_2, v_3, v_4\} = \{v_0, v_2, v_3, v_4\}$. There are four P_3-paths ($[v_1, v_0, v_6]$, $[v_1, v_2, v_6]$, $[v_1, v_3, v_6]$, and $[v_1, v_4, v_6]$) between v_1 and v_6 in the graph G. We notice that $\{v_1, v_6\}$ is the only chord in the induced graph created with pairs of $[v_1, v_2, v_6]$, $[v_1, v_4, v_6]$ and $[v_1, v_3, v_6]$, $[v_1, v_4, v_6]$. Hence the deletion of $\{v_1, v_6\}$ is not allowed. Were the chord $\{v_2, v_4\}$ also present in the induced graph formed by pairs of $[v_1, v_2, v_6]$, $[v_1, v_4, v_6]$ and $[v_1, v_3, v_6]$, $[v_1, v_4, v_6]$ we could have delete $\{v_1, v_6\}$.

The edge $\{v_0, v_2\}$, on the other hand, can be deleted from the graph. Since $N(v_0) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$ and $N(v_2) = \{v_0, v_1, v_3, v_6\}$ then $N(v_0) \cap N(v_2) = \{v_1, v_3, v_6\}$. There are three P_3-paths ($[v_0, v_1, v_2]$, $[v_0, v_3, v_2]$, and $[v_0, v_6, v_2]$) between v_0 and v_2. We are allowed to delete the edge $\{v_0, v_2\}$ because there is another chord present in the induced graph created with a pair of P_3-paths (shown in Fig. 2.11(b)).

2.4.2 Complexity of Deletions

The neighborhood computations take linear time. To determine if $\{u, v\}$ is the only chord of G, we have to find all P_3 paths between u and v. The number of such paths is bounded above by $O(d_u d_v)$, where d_u and d_v are the degrees of the vertices u and v respectively. The actual paths can also be found out in $O(d_u d_v)$ time by computing the intersection of u and v. Clearly, we have a P_3 path between u and v for each entry in the common neighborhood set. Thus the number of paths and the time complexity of finding these are both bounded by $O(d_u d_v)$. The next step is to find out if $\{u, v\}$ is not the only chord of each 4-cycle determined by disjoint pairs of these P_3 paths, a task that can be accomplished in $O(d_u^2 d_v^2)$ time. Thus the query complexity of this step is in $O(d_u^2 d_v^2)$.

The deletion of an edge takes constant time since we maintain an adjacency matrix data structure to represent G.

2.4.3 Discussion

In this section, we have presented a semi-dynamic algorithm for edge-deletions in chordal graphs. The proposed semi-dynamic algorithm is based on the characterization of chordal graphs that prohibit induced chordless cycles of size 4 or more. The proposed method is straightforward and does not require us to maintain the complex clique tree representation of a chordal graph. We also avoid the reconstruction of a chordal graph from the final clique tree.

2.5 Summary

In this chapter, we have presented two different chordal graph generation methods and also a semi-dynamic algorithm for edge-deletions. The Unified methods generate chordal graphs for a given number of vertices and a given number of edges. The k-chromatic chordal graph generation method turns an arbitrary graph into a chordal graph by adding a minimal number of edges.

Chapter 3

Weakly Chordal Graph Generation

Though a chordal graph G cannot contain an induced chordless cycle of size four or more, it can be seen from Fig. 3.1 that its complement can contain an induced chordless cycle of size four. However, the complement cannot contain a 5-cycle, as the complement of a 5-cycle is also a 5-cycle.

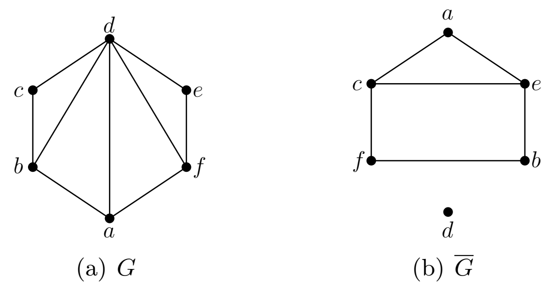


Figure 3.1: *Complement of a chordal graph with a chordless 4-cycle*

This suggests the generalization of chordal graphs to weakly chordal (or weakly triangulated) graphs as those graphs G such that neither G nor its complement $\overline{G}$ contains induced chordless cycles of size five or more [26]. From the symmetry of the definition, it follows that $\overline{G}$ is also weakly chordal. Figure 3.2 shows an example of a weakly chordal graph, G, and its complement, $\overline{G}$.

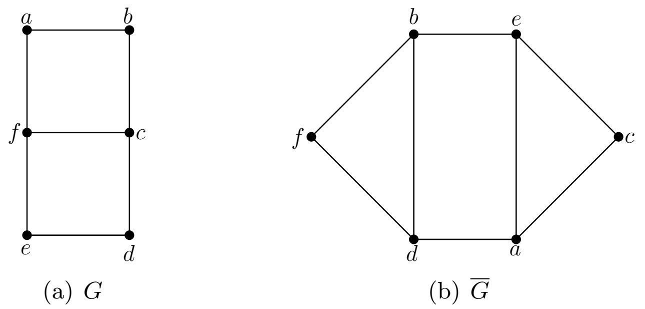


Figure 3.2: *Weakly chordal graph*

In this chapter, we address the algorithmic problem of generating weakly chordal graphs. While the problem of generating graphs uniformly at random has received much attention (see [37, 54, 55, 56]), little is known about this problem. In fact, the only prior work we are aware of is described later in this section. There are many situations where we would like to generate instances of these to test algorithms for weakly chordal graphs. For instance, in [39], the authors generate all linear layouts of weakly chordal graphs. A generation mechanism can be used to obtain test instances for this algorithm. A number of optimization algorithms, like finding a maximum clique, maximum stable set, minimum clique cover, minimum coloring, for both weighted and unweighted versions, of a weakly chordal graph have been studied in [28] and versions with improved time complexities have appeared in [29, 52]. It would be interesting and useful to carry out experimental studies of these algorithms using our generation algorithm to create a variety of input instances. In another direction, in the line of the work of Seker et al. [51] for chordal graphs, it would be useful to initiate an experimental study of the extent to which our algorithm generates random weakly chordal graphs by studying the distribution of maximal cliques.

In this chapter, we propose two methods for the generation of weakly chordal graphs. The first method takes the number of vertices (n) and the number of edges (m) as input. We first construct a tree and then generate an orthogonal layout (which is a weakly chordal graph on the n vertices) based on this tree. We then insert additional edges, if needed, for a total of m edges. On the other hand, the second method turns an arbitrary graph into a weakly chordal graph by adding edges in the graph. This method checks the LB-simpliciality of every edge in the graph including the newly added edge.

3.1 Weakly Chordal Graph Generation

Let $P_k (k \geq 3)$ denote a chordless path, spanning k vertices of G. An edge e in G is peripheral if it is *not* the middle edge of a P_4 of G. In [27], Hayward proposed the following constructive characterization of weakly chordal graphs, based on the notion of a peripheral edge.

Theorem 3.1 [27] *A graph is weakly chordal if and only if it can be generated in the following manner:*

1. *Start with an empty graph G_0.*

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

This is analogous to a similar characterization for chordal graphs by Fulkerson and Gross [23]. No details were provided as to how to decide if an edge is peripheral and the complexity of the generation method. The proof of this theorem uses the notion of a two-pair and a generation method can be devised based on this. A pair of vertices $\{u, v\}$ in G is a two-pair if the only chordless paths between u and v are of length 2. Interestingly enough, a weakly chordal graph that is not a clique has a two-pair [28]. Furthermore, let $\{u, v\}$ be a two-pair in an arbitrary graph G. Then $G + \{u, v\}$ is weakly chordal if and only if G is weakly chordal [52]. We can then generate a weakly chordal graph on n vertices by starting with a tree (as the complement of a 5-cycle is also a 5-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 $O(mn)$ (where n is the number of vertices and m is the number of edges in the current graph) algorithm due to Arikati and Rangan [4]. Unfortunately, this does not allow us to exploit the structural properties of a weakly chordal graph, nor does it allow us to add an edge between a pair of vertices, which is not necessarily a two-pair. Thus in the following section, we propose a separator-based strategy that generalizes an algorithm due to Markenzon [36] for generating chordal graphs and allows us to exploit the structural properties of a weakly chordal graph, with the additional feature of being able to join non two-pair vertices that keep the graph weakly chordal (the dashed line in Fig. 3.3).

3.1.1 Separator-based Weakly Chordal Graph Generation

In this section, we propose a scheme for generating a weakly chordal graph on n vertices with m edges. In this method, we first construct a tree and then generate an orthogonal layout (which is a weakly chordal graph on the n vertices) based on this tree. We then insert

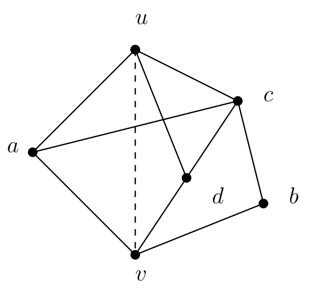


Figure 3.3: *G is weakly chordal and so is $G + \{u, v\}$*

additional edges, if needed, for a total of m edges. Our algorithm ensures that the graph remains weakly chordal after each edge is inserted. The time complexity of an insertion query is $O(d_u^2 d_v^2(n + m))$, where d_u and d_v are the degrees of the vertices u and v we want to join with an edge and an insertion takes constant time. The advantages of this method are that it uses very simple data structures and exploits the basic structural properties of a weakly chordal graph.

The rest of the section is organized thus. In the next section, we add a brief review of Markenzon's incremental method [36] for generating chordal graphs. The following subsection contains details of our algorithm, beginning with a brief overview. The final subsection contains some concluding remarks and suggestions for further research.

Preliminaries

For any vertex set $S \subseteq V$, the open neighborhood, $N(S)$, of S is defined to be $N(S) = \{x \in V - S \mid \exists y \in S,$ such that $\{x, y\} \in E\}$.

Given n and m, Markenzon's method [36] starts by generating a labeled tree on n vertices using Prüfer's scheme [42]. Next, an edge is inserted in each iteration to reach m.

The algorithm picks a pair of vertices u and v and checks whether v is reachable from u in the induced graph $G[S]$, where $S = V - I_{u,v}$ and $I_{u,v} = N(u) \cap N(v)$. If $G[S]$ is connected, then a path exists between u and v and thus $G + \{u, v\}$ is not chordal; otherwise, the augmented graph $G + \{u, v\}$ is chordal. To make the search for a path efficient, the method reduces the set S from $V - I_{u,v}$ to $N(x) - I_{u,v}$, for any $x \in I_{u,v}$. The correctness of the algorithm was established by proving the following theorem.

Theorem 3.2 [36] *Let $G = (V, E)$ be a connected chordal graph and $u, v \in V, \{u, v\} \notin E$. The augmented graph $G + \{u, v\}$ is chordal if and only if $G[V - I_{u,v}]$ is not connected.*

Overview of the Method

As in Markenzon's method for generating chordal graphs, the inputs to our algorithm are the number of vertices, n, and the number of edges, m, of a weakly chordal graph to be generated. The algorithm has three phases. As trees are weakly chordal graphs, in the first phase, it generates a tree with at least n vertices. In the next phase, it uses this tree to generate an orthogonal layout (which is also a weakly chordal graph on at least n vertices) made up of 4-cycles and edges incident on the vertices of these 4-cycles. In the third and final phase, vertices are removed if the count exceeds n and additional edges are introduced to bring up the edge tally to m, maintaining weak chordality.

Inserting an edge between two vertices u and v so that weak chordality is preserved requires careful consideration. Let $I_{u,v}$ be the set of common neighbors of u and v. If $I_{u,v} \neq \emptyset$, we check whether the removal of $I_{u,v}$ separates u and v, that is put them in different components of $G[V - I_{u,v}]$. To check for this we can do a breadth-first search in $G[V - I_{u,v}]$, starting at u to see if v is reachable. As we shall see, this search can be done in the induced graph of a reduced set of vertices called *AuxNodes*. If v is not reachable from u, we insert 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, we have 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 $I_{u,v} = \emptyset$, we proceed in the same way as when $I_{u,v}$ does not separate u and v (see previous paragraph). The only difference is that here we have to consider the entire graph to search for shortest paths between u and v but the other details remain essentially the same.

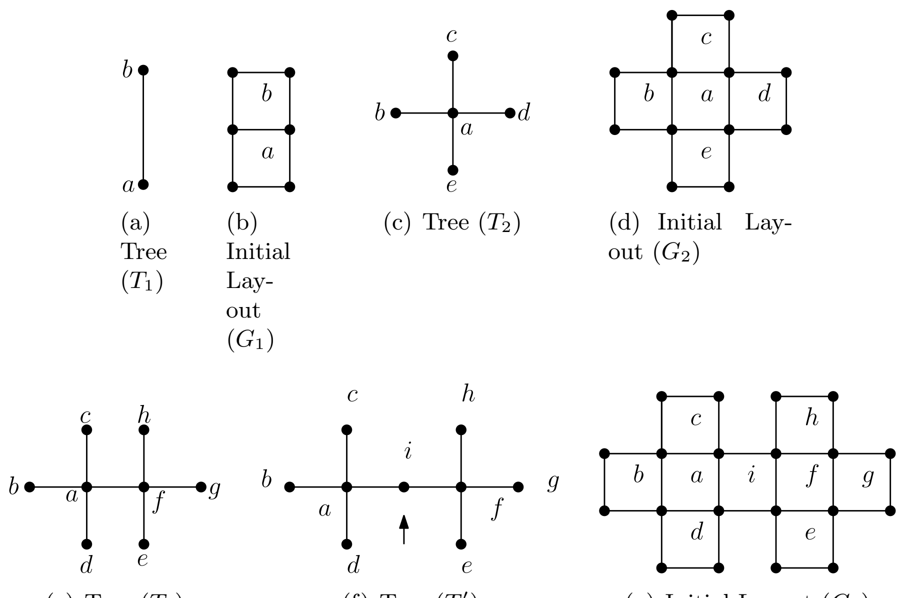
The three phases are explained in full details in the next subsections.

Phase 1: Generation of Tree

We generate as follows a tree, T, on $\lceil \frac{n}{2} \rceil$ nodes such that each node has degree at most four. Starting with a single node, we add new ones either by splitting an edge into two or joining a new node to an existing node, chosen at random. After $\lceil \frac{n}{2} \rceil$ nodes have been added, we traverse 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 (we explain this in the description of the next phase). Let $k(\geq \lceil \frac{n}{2} \rceil)$ be the number of nodes in the resulting tree, T'.

Phase 2: Generation of Initial Layout

In this phase, we generate an orthogonal layout that corresponds to T' in the following way. For each node in T', we create a 4-cycle. By having 4-cycles in our layout, we ensure that the algorithm generates proper weakly chordal graphs and not just chordal graphs. Two 4-cycles have an edge in common if and only if the corresponding tree nodes are adjacent. Fig. 3.4(a) shows a tree T' with two nodes, a and b. The corresponding layout is shown in Fig. 3.4(b). It has two 4-cycles, each corresponding to a node of T'; these share an edge in common as the tree nodes a and b are adjacent. We define this as the initial layout. For the example tree of Fig. 3.4(c), the corresponding initial layout is shown in Fig. 3.4(d).



(a) Tree (T_1)
(b) Initial Layout (G_1)
(c) Tree (T_2)
(d) Initial Layout (G_2)
(e) Tree (T_3)
(f) Tree (T'_3)
(g) Initial Layout (G_3)

Figure 3.4: *Tree to layout (4-cycles)*

As explained in the first phase, after generating a tree, if two degree-4 nodes or a degree-

3 node and a degree-4 node are adjacent, we insert a new node between them to separate them (see Figs. 3.4(e), 3.4(f), and 3.4(g)). 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, we delete enough vertices from the 4-cycles to bring the count down to n. Note that the orthogonal layout has $2k + 2 > n$ vertices where $k(\geq \lceil \frac{n}{2} \rceil)$ is the number of vertices in the tree generated in Phase 1. Vertices that are candidates for deletion are those that have degree two. Post vertex-deletion, if the number of edges m' in the resulting layout is m or more, the algorithm stops and returns the layout (which is a weakly chordal graph) as our output. Otherwise, we proceed to the next phase.

Phase 3: Generation of Weakly Chordal Graph

In this phase, we insert $(m - m')$ additional edges into the initial layout, preserving weakly chordality. Two cases arise, according as $I_{u,v} \neq \emptyset$ and otherwise. We discuss them in this order.

Case 1: $I_{u,v}$ is non-empty Since $I_{u,v}$ is non-empty, we need to check whether the removal of $I_{u,v}$ separates u and v. This can be settled by checking for the existence of a path from u to v by a breadth-first search in the induced graph $G[V - I_{u,v}]$. To make this search more efficient, we perform this search in a smaller set than $V - I_{u,v}$. Call this set *AuxNodes*. For chordal graphs Markenzon et al. [36] defined this set to be $N(x) - I_{u,v}$, where x is any vertex in $I_{u,v}$. The correctness of this choice was shown by appealing to the fact that in a chordal graph minimal separators are cliques. For weakly chordal graphs, we have to be more careful. We define $AuxNodes = N(I_{u,v}) \cup N(N(I_{u,v}) \cup \{I_{u,v}\})$, which as an extended neighborhood of $I_{u,v}$. In the next paragraph, we substantiate this definition with the help of an example.

Refer to the graph in Fig. 3.5. We have $I_{u,v} = \{a\}$ and $N(I_{u,v}) = \{u, v, b, d\}$. If we define $AuxNodes = \{u, v, b, d\}$, it is clear that there is no path from u to v in $G[AuxNodes]$. However, if we redefine $AuxNodes$ as an extended neighborhood of $I_{u,v}$, viz., $AuxNodes = \{u, v, b, c, d, e\}$ then there are two chordless paths, $[u, c, d, e, v]$ and $[u, c, d, b, v]$, between u and v in $G[AuxNodes]$. Each of these paths prevents the addition of the edge $\{u, v\}$ to G as this creates a chordless 5-cycle.

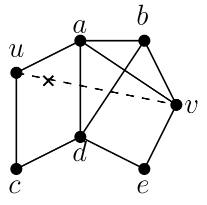


Figure 3.5: *Why neighbors of neighbors?*

Setting u as the source vertex, we now perform breadth-first search in $G[AuxNodes]$ for a path from u to v. If *no path exists*, then adding $\{u, v\}$ to G keeps it weakly chordal. If a path does exist and its length is *longer than 3*, then $\{u, v\}$ cannot be added to G without violating weak chordality. If there is a path of length three (a P_4 as we shall say) from u to v, we cannot yet add the edge $\{u, v\}$ to G as there may exist chordless paths of length greater than three between u and v, which precludes this addition. We need to check for this. We consider two subcases: (1) exactly one P_4 connects u to v; (2) more than one P_4 connects u to v.

Case 1.1: 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. As can be inferred from the graphs in Figs. 3.6(a), 3.6(b) 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 internal vertices from P_4. By setting *AuxNodes* to $N(SP) \cup N(N(SP) \cup SP) \cup (SP - \{x\})$, or to $N(SP) \cup N(N(SP) \cup SP) \cup (SP - \{y\})$ or to $N(SP) \cup N(N(SP) \cup SP) \cup (SP - \{x, y\})$ we can capture all potential shortest paths from u to v that are longer than P_4. The first two cases are used to find longer paths via one of the internal nodes of P_4 and the third case is to find longer paths disjoint from P_4. Breadth-first search from u now takes place in the induced graph $G[AuxNodes]$. A formal description in Algorithm 3.1 has all the details.

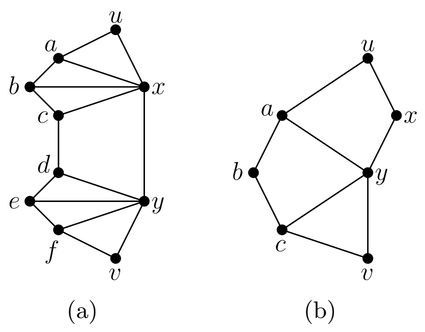


Figure 3.6: *Neighbors of a path*

Case 1.2: When multiple P_4's exist, we need to check for a forbidden configuration

Algorithm 3.1 *SingleShortestPathSubCase*

```
Input: Single shortest path ($SP \leftarrow \{u, x, y, v\}$, $u$, and $v$)
Output: Returns True if $\{u, v\}$ can be inserted in $G$ otherwise False
 1: canBeInserted ← False
 2: Compute the vertex set $N(SP)$
 3: if $N(SP) == \emptyset$ then
 4:     canBeInserted ← True
 5:     return canBeInserted
 6: else
 7:     Compute the vertex set $N(N(SP) \cup SP)$
 8:     Compute the vertex set $AuxNodes \leftarrow N(SP) \cup N(N(SP) \cup SP)$
 9:     Create $AuxGraph$ on $AuxNodes' \leftarrow N(SP) \cup N(N(SP) \cup SP) \cup (SP - \{x\})$ or
        $AuxNodes' \leftarrow N(SP) \cup N(N(SP) \cup SP) \cup (SP - \{y\})$ or $AuxNodes' \leftarrow N(SP) \cup$
        $N(N(SP) \cup SP) \cup (SP - \{x, y\})$
10:     Perform breadth-first search $BFS(u, AuxGraph)$ till all the vertices of $AuxNodes' -$
        $\{u\}$ have been visited or $v$ has been reached
11:     if no chordless longer path exists between $u$ and $v$ in $AuxGraph$ then
12:         canBeInserted ← True
13:         return canBeInserted
14:     end if
15: end if
16: return canBeInserted
```

formed by a pair of P_4's as shown in Fig. 3.7(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 6-cycle in $\overline{G}$ as can be seen from the complement of the configuration in Fig. 3.7(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 3.8 shows some other allowed configurations formed by pairs of P_4's, where adding $\{u, v\}$ to G does not violate its weak chordality.

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 $P_4^1, P_4^2, \ldots, P_4^k$ be $k(\geq 2)$ P_4's from u to v, where $P_4^i = [u, x_i, y_i, v]$ and x_i, y_i are its internal vertices. Define $allSP =$

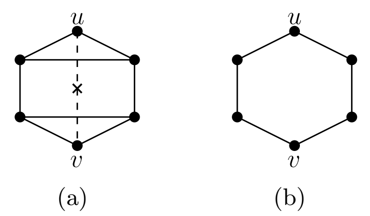


Figure 3.7: *(a) Forbidden configuration formed by a pair of P_4's; (b) its complement*

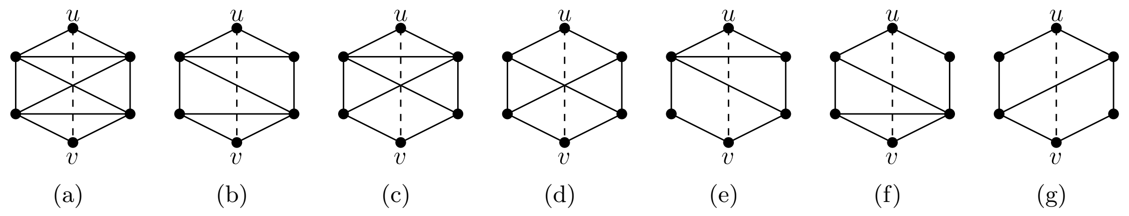


Figure 3.8: *Some other permitted configurations formed by a pair of P_4's*

$\{u, x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j, v\}$ 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 $N(allSP) \cup N(N(allSP) \cup allSP)$. Figures 3.9(a)-3.9(g) show all the different ways such a longer path from u to v can contain internal vertices of the P_4's. In these figures, the black dots (●) represent the set of vertices that are in the $AuxNodes$ set and lie on a path longer than P_4 from u to v.

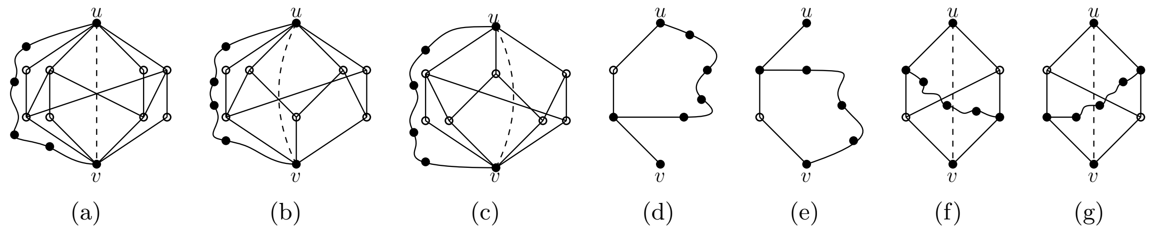


Figure 3.9: *Case 1.2: Paths longer than P_4 between u and v, (a)-(c) completely disjoint path, (d)-(g) shared path*

To find a path longer than P_4 from u to v, completely disjoint from all the P_4's, we set $AuxNodes = \{N(allSP) \cup N(N(allSP) \cup allSP) \cup \{u, v\}\} - \{x_1, x_2, \ldots, x_n\} -$

Algorithm 3.2 *MultipleShortestPathSubCase*

```
Input: Multiple shortest paths ($allSP \leftarrow \{u, x_1, x_2, \ldots, x_i, y_1, y_2, \ldots, y_j, v\}$, $u$, and $v$)
Output: Returns True if $\{u, v\}$ can be inserted in $G$ otherwise False
1: canBeInserted ← False
2: if no forbidden configuration then
3:     Compute the vertex set $N(allSP)$
4:     Compute the vertex set $N(N(allSP) \cup allSP)$
5:     Compute the vertex set $AuxNodes \leftarrow N(allSP) \cup N(N(allSP) \cup allSP)$
6:     Create $AuxGraph$ on $AuxNodes' \leftarrow AuxNodes \cup \{u, v\} - \{x_1, x_2, \ldots, x_n\} - \{y_1, y_2, \ldots, y_n\}$    ▷ see Figs. 3.9(a)-3.9(c)
7:     Perform breadth-first search $BFS(u, AuxGraph)$ till all the vertices of $AuxNodes' - \{u\}$ have been visited or $v$ has been reached
8:     if no chordless longer path exists between $u$ and $v$ in $AuxGraph$ then
9:         Create $AuxGraph$s on the candidate vertex sets $AuxNodes' \cup \{x_i\}$ and $AuxNodes' \cup \{y_i\}$    ▷ see Figs. 3.9(d) and 3.9(e)
10:        Perform breadth-first search $BFS(u, AuxGraph)$ till all the vertices of the candidate vertex sets have been visited or $v$ has been reached
11:        if no chordless longer path exists between $u$ and $v$ in any of the $AuxGraph$s then
12:            for each disjoint pair of $P_4^i$ and $P_4^j$, create $AuxGraph$s on the candidate vertex sets $AuxNodes' \cup \{x_i, y_j\}$ or $AuxNodes' \cup \{x_j, y_i\}$    ▷ see Figs. 3.9(f) and 3.9(g)
13:            Perform breadth-first search $BFS(u, AuxGraph)$ till all the vertices of the candidate vertex sets have been visited or $v$ has been reached
14:            if no chordless longer path exists between $u$ and $v$ in any of the $AuxGraph$s then
15:                canBeInserted ← True
16:                return canBeInserted
17:            end if
18:        end if
19:    end if
20: end if
21: return canBeInserted
```

$\{y_1, y_2, \ldots, y_n\}$. Each of the Figs. 3.9(a), 3.9(b), and 3.9(c) illustrates this situation.

The second possibility that must be considered is that such a longer path between u and v, passes through one internal vertex of a $P_4^i = u - x_i - y_i - v$ and thus shares an edge with it. Note that we do not preclude the possibility that this shared edge is a part of other P_4's. In this case, we set the $AuxNodes$ in turn to $\{N(allSP) \cup N(N(allSP) \cup allSP) \cup \{u, v\} \cup \{x_i\}\}$ and $\{N(allSP) \cup N(N(allSP) \cup allSP) \cup \{u, v\} \cup \{y_j\}\}$ respectively while searching for such a path. Figures 3.9(d) and 3.9(e) illustrate the two different ways this can happen. This search is repeated for each of the k P_4's.

The third possibility is that such a longer path contains one internal vertex from each of two disjoint P_4's. The search for a longer path is now done by setting $AuxNodes$ in turn to $\{N(allSP) \cup N(N(allSP) \cup allSP) \cup \{u,v\} \cup \{x_i, y_j\}\}$ and $\{N(allSP) \cup N(N(allSP) \cup allSP) \cup \{u,v\} \cup \{x_j, y_i\}\}$ respectively. The two scenarios are illustrated in Figs. 3.9(f) and 3.9(g). This search is repeated for each pair of P_4's.

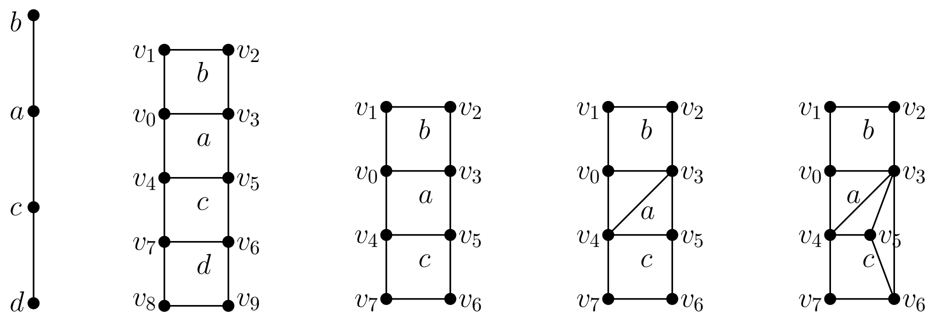
Algorithm 3.3 *InitLayoutToWCG*

```
Input: An initial layout $G = (V, E)$
Output: A weakly chordal graph $G + \{u, v\}$
 1: $p \leftarrow 0$; $q \leftarrow m - m'$            ▷ $m'$ is the no. of edges in the initial layout
 2: while $p < q$ do
 3:     Choose a pair of vertices $u$ and $v$ at random
 4:     if the edge $\{u, v\}$ already exists then
 5:         continue
 6:     else
 7:         Compute $N(u)$ and $N(v)$
 8:         Compute $I_{u,v} \leftarrow N(u) \cap N(v)$
 9:         if $I_{u,v}$ is non-empty then                  ▷ case 1: $I_{u,v} \neq \emptyset$
10:             Compute the vertex set $N(I_{u,v})$
11:             Compute the vertex set $N(I_{u,v}) \cup \{I_{u,v}\}$
12:             Compute the vertex set $AuxNodes \leftarrow N(I_{u,v}) \cup N(N(I_{u,v}) \cup \{I_{u,v}\})$
13:             Create $AuxGraph$ on $AuxNodes$
14:             Perform breadth-first search $BFS(u, AuxGraph)$ till all the vertices of
    $AuxNodes - \{u\}$ have been visited or $v$ has been reached
15:             if $v$ has not been reached from $u$ in $AuxGraph$ then
16:                 insert edge $\{u, v\}$
17:                 $p \leftarrow p + 1$
18:             else
19:                 if the shortest path is not longer than a $P_4$ then
20:                     if there is a single $P_4$ between $u$ and $v$ then     ▷ case 1.1 (single $P_4$)
21:                         if $singleShortestPathSubCase(P_4, u, v)$ returns True then
22:                             insert edge $\{u, v\}$
23:                             $p \leftarrow p + 1$
24:                         end if
```

```
                else                                                  ▷ case 1.2 (multiple $P_4$'s)
                    if $multipleShortestPathSubCase(P_4^1, P_4^2, \ldots, P_4^k, u, v)$ returns True then
                        insert edge $\{u, v\}$
                        $p \leftarrow p + 1$
                    end if
                end if
            end if
        end if
    else                                                              ▷ case 2: $I_{u,v} = \emptyset$
        if the shortest path is not longer than a $P_4$ then
            if there is a single $P_4$ between $u$ and $v$ then       ▷ case 2.1 (single $P_4$)
                if $singleShortestPathSubCase(P_4, u, v)$ returns True then
                    insert edge $\{u, v\}$
                    $p \leftarrow p + 1$
                end if
            else                                                      ▷ case 2.2 (multiple $P_4$'s)
                if $multipleShortestPathSubCase(P_4^1, P_4^2, \ldots, P_4^k, u, v)$ returns True then
                    insert edge $\{u, v\}$
                    $p \leftarrow p + 1$
                end if
            end if
        end if
    end if
  end if
end while
```

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 1.1 and 1.2. Here the *AuxGraph* is the entire graph because $I_{u,v}$ 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 1.1 and 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. The details of these cases are formally described in algorithm 3.3.

An Example Consider generating a weakly chordal graph with $n = 8$ vertices and $m = 12$ edges. All the phases, from left to right, are shown in Fig. 3.10. Figure 3.10(a) shows a tree on $\lceil \frac{n}{2} \rceil$ nodes. There are four nodes in the tree and on expanding each node to a 4-cycle, we get the initial orthogonal layout of Fig. 3.10(b) 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 Fig. 3.10(c). We 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.



(a) Tree (T_1) (b) Initial Layout (G_1) (c) Initial Layout (G_1') (d) WCG(G_1') (e) WCG(G_1'')

Figure 3.10: *Tree to weakly chordal graph*

Say we want to insert edge between the vertices v_3 and v_4. Since I_{v_3,v_4} is non-empty we have Case 1. As the removal of I_{v_3,v_4} leaves the vertices v_3 and v_4 in two different components, we can safely insert an edge between vertices v_3 and v_4 as per case 1 as shown in Fig. 3.10(d). Next, let us try to insert edge $\{v_3, v_6\}$. The insertion of this edge corresponds to case 1.1 because the removal of their common neighbor, viz., $\{v_5\}$ does not separate v_3 and v_6. Now, $N(v_5)$ is $\{v_3, v_4, v_6\}$ and $N(N(I_{v_3,v_6}) \cup I_{v_3,v_6})$ is $\{v_0, v_2, v_7\}$ and therefore

$AuxNodes = \{v_3, v_4, v_6, v_0, v_2, v_7\}$. In $G'_1[AuxNodes]$ we search for paths from v_3 and v_6. There is a single shortest path $SP = \{v_3, v_4, v_7, v_6\}$ and this corresponds to Case 1.1. Since $N(SP) = \{v_0, v_2\}$ is not an empty set, we need to compute $\{N(N(SP) \cup SP)\}$ which is empty. Thus $AuxNodes = N(SP) \cup N(N(SP) \cup SP) \cup SP = \{v_3, v_4, v_6, v_0, v_2, v_7\}$ vertices. We create different induced graphs on $G'_1[AuxNodes]$ by removing both the internal vertices from SP or exactly one of them and observed that there is no chordless path between v_3 and v_6. Hence, we can insert an edge between v_3 and v_6.

Complexity

The tree generation is the first phase in the proposed approach and can be constructed in time $O(k)$, where k is the number of nodes in a tree. For k nodes in the tree, we insert $3k + 1$ edges in the layout and each edge insertion can be done in constant time. So the initial layout can be generated in $O(k)$ 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, we 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 $I_{u,v} = \emptyset$ in Algorithm 3). An upper bound on the number of pairs of P_4's can be estimated this way.

Assume G has n vertices. Let $\{v_1, v_2, \ldots, v_l\}$ 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 $l \leq d_v$. Likewise, let $\{v'_1, v'_2, v'_3, v'_4 \ldots v'_k\}$ be the set of vertices adjacent to u that lie on these shortest paths. Again $k \leq d_u$, where d_u is the degree of u. Thus, $d_u d_v$ 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 $\{v'_1, v'_2, v'_3, v'_4 \ldots v'_k\}$, 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 $PathCount = \sum_{i \neq j} d_i d_j$. Let $\sum d_i = t$. Now, it follows from the equality $2l \sum d_i d_j = (l-1)(\sum d_i)^2 - \sum_{i \neq j}(d_i - d_j)^2$ that $PathCount$ is maximum when all d_i's are equal. Therefore, an upperbound on $PathCount$ is $l^2.(\frac{t}{l})^2 = t^2$. Since $t = O(lk)$, we have $PathCount = O(d_u^2 d_v^2)$.

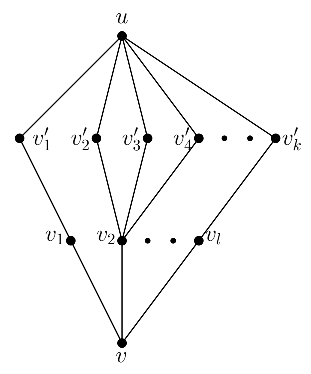


Figure 3.11: *P_4-paths between u and v*

If $|E|$ be the number of edges in the current weakly chordal graph, the time 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 $O(d_u^2 d_v^2(n + m))$.

Conclusions

We have proposed a method for the generation of weakly chordal graphs. We have implemented the proposed algorithm in Python. An interesting open problem is to investigate how to generate weakly chordal graphs uniformly at random. This requires coming up with a scheme for counting the number of labeled weakly chordal graphs on n vertices having m edges. If we could also determine the probability distribution underlying our algorithm for generating weakly chordal graphs, then we could compute the relative entropy between the two distributions to estimate how close our algorithm is to generating weakly chordal graphs uniformly at random. The term *relative entropy* (also known as the Kullback-Leibler 'distance') is a measure of the similarity between two probability distributions. For a given two probability distributions P and Q, the relative entropy given by $H(P||Q)$ is defined as follows [19]:

$$H(P||Q) = \sum_i P(x_i) \log \frac{P(x_i)}{Q(x_i)}$$

The relative entropy is always nonnegative and equal to zero if and only if $P(x_i) = Q(x_i)$.

3.1.2 Arbitrary Graph to Weakly Chordal Graph

In this section, we describe an algorithm for generating a weakly chordal graph from an arbitrary input graph. This problem has been listed as an open problem in [9]. Here and below, the abbreviation LB will stand for the initials of the authors Lekerkerker and Boland of the paper [33].

A vertex v of a graph G is said to be LB-simplicial if all the separators of G contained in the neighborhood $N(v)$ of v are cliques. In [33], Lekerkerker and Boland gave the following alternate characterization of chordal (triangulated) graphs.

Theorem 3.3 [33] *A graph is triangulated iff every vertex is LB-simplicial.*

This implies a recognition algorithm for chordal graphs. In [9], Berry et al. extended the above recognition algorithm to weakly chordal graphs, based on the notion of LB-simpliciality for edges in a graph G.

Let $e = \{u, v\}$ be an edge of G and S is a separator contained in its neighborhood, $N(e)$. For each component S_j of $\overline{G}(S)$, if at least one endpoint of e sees all vertices of S_j, then e is said to be S-saturating.

An edge e is LB-simplicial if it satisfies one of the following two conditions [9]:

- e is S-saturating for each minimal separator S included in the neighborhood of e
- $N[e] = V$

Our generation algorithm is based on a scheme for recognizing weakly chordal graphs that is based on the following result due to Berry et al. [9].

Theorem 3.4 [9] *A graph $G = (V, E)$ is weakly triangulated iff every edge of E is LB-simplicial.*

We run the recognition algorithm on an arbitrary input graph G. If an edge e fails the LB-simpliciality test, we add the necessary edges to make it LB-simplicial. We iterate over every edge (including the newly added ones) in the graph G and make it LB-simplicial. Before explaining this method in more detail, in the next subsection, we introduce some necessary notations and definitions.

Notations

The graph obtained after each iteration is denoted by $H = (V, E + F)$, F be the set of added edges. $N[C] = \cup_{x \in C} N[x]$ denotes the closed neighborhood of C (note that it also contains C). For $X \subseteq V$, $\mathcal{C}(X)$ is the set of connected components of $G(V - X)$. S is called a separator if $\mathcal{C}(S) \geq 2$, an xy-separator if x and y are in two different connected components. In the following subsections, we present the algorithm and explain each of the steps.

The Generation Method

This method generates a weakly chordal graph from an arbitrary graph. Thus, the first step is to generate an arbitrary graph. For this, we use an algorithm called 'dense_gnm_random_graph' method by Keith M. Briggs, which is inspired by Knuth's Algorithm S (Selection sampling technique) that appears in section 3.4.2 of [32]. The method generates an arbitrary graph on n vertices with m edges. The arbitrary graph may have more than one component. If so, we connect the components by introducing an edge between each component to obtain a connected graph. Then we use the resulting graph as an input to Algorithm 3.4, described below.

Algorithm 3.4 *ArbitraryToWCG*

```
Input: An arbitrary graph $G = (V, E)$
Output: A weakly chordal graph $H = (V, E + F)$ of $G$
 1: while the graph turns into weakly chordal do
 2:     for each edge $e \in E$ do    ▷ $e$ represents an edge between a pair of vertices $u$ and $v$
 3:         Compute $N[\{u, v\}]$
 4:         if $N[\{u, v\}] \neq V$ then
 5:             Compute the set of connected components $\mathcal{C}(N[\{u, v\}])$
 6:             Compute the minimal separators $S$ contained in the neighborhood of $\{u, v\}$
 7:             for each minimal separator $S$ in the list do
 8:                 Compute the set of connected components of $\overline{G}(S)$
 9:                 if all the components are visible from one of the end points of $e$ then
10:                     The edge $e$ is LB-simplicial
11:                 else
12:                     Add new edges and turns the component into a clique
13:                 end if
14:             end for
15:         end if
16:     end for
17: end while
```

Let k be the number of rounds. In the first round ($k = 1$), the algorithm iterates every edges in the arbitrary graph G and adds new edges to turn every edge LB-simplicial. The edges have already turned LB-simplicial may not remain LB-simplicial after introducing other edges in the graph. Thus if any new edges added in the first round, the algorithm takes the modified graph as an input for the next round ($k = 2$), iterates every edge and adds new edges to turn every edge LB-simplicial, unless it is already so. The algorithm stops when there are no new edges added and every edge becomes LB-simplicial.

Checking for LB-simpliciality of an edge e requires a number of different computations, including computing the neighborhoods, the set of connected components in G and $\overline{G}(S)$. The first step is to compute the closed neighbors of both the endpoints of an edge e. The next step is to compute the minimal separators contained in the neighborhood of e. To get the minimal separators, we can compute $N[C] \setminus C$ for each component C of $\mathcal{C}(N[e])$. For each minimal separator S in the list, we compute the set of connected components of $\overline{G}(S)$.

The algorithm iterates and verifies whether every edge is LB-simplicial or not. Figure 3.12 shows an example of generating a weakly chordal graph (Fig. (b)) from an arbitrary graph (Fig. (a)). The closed neighborhood of $N[\{a,b\}]$ is $\{a,b,f,c\}$. Now $\mathcal{C}(N[\{a,b\}]) =$

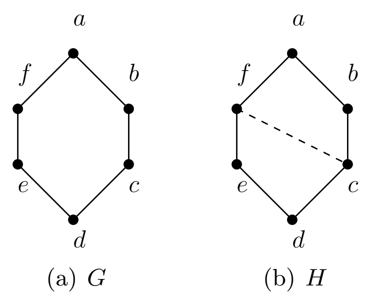


Figure 3.12: *Arbitrary graph to weakly chordal graph*

$\{d,e\}$. The only separator included in the neighborhood of $N(\{d,e\})$ is $\{c,f\}$. Now the connected components of $\overline{G}(\{c,f\})$ is $\{c,f\}$. Vertex a can see f only but not c and vertex b can see c only but not f. Now if we introduce an edge between c and f, then the edge $\{a,b\}$ becomes LB-simplicial. Next choose another edge say, $\{b,c\}$. The closed neighborhood of

$N[\{b, c\}]$ is $\{b, a, c, d, f\}$. Similarly, we can compute $\mathcal{C}(N[\{b, c\}])$ which is $\{a, d, f\}$. The only separator included in the neighborhood of $N(\{e\})$ is $\{d, f\}$. Vertex b neither see d nor f but vertex c can see both d and f. So the edge $\{b, c\}$ is LB-simplicial. In a similar way, we can check the LB-simpliciality of the other edges and because of the addition of only one new edge $\{c, f\}$, the arbitrary graph G turns into a weakly chordal graph H. In the second round, the algorithm takes H as an input, iterates every edge, and observe every edge is LB-simplicial. Thus no new edges are added and the arbitrary graph turns into a weakly chordal graph.

Discussion

The proposed algorithm can turn an arbitrary graph into a weakly chordal graph. The method turns a component into a clique when the component is not visible from one of the endpoints of an edge. By doing so, perhaps we introduce more edges than required. We also need to check if each edge, including the newly added edges, is LB-simplicial or not. During the checking of LB-simpliciality of an edge e, we introduce edges to ensure that e is LB-simplicial. The newly added edges are also LB-simplicial. But the addition of other edges in the later iterations, the edges added previously, may not remain LB-simplicial. Thus we need to check the LB-simpliciality of all the edges in more than one round. An interesting open problem is to find a way to turn an arbitrary graph into a weakly chordal graph by introducing the minimal number of edges while checking the LB-simpliciality of every edge in the given arbitrary graph.

3.2 Summary

In this chapter, we have described a scheme for generating weakly chordal graphs on n vertices and m edges. We also proposed an algorithm to generate a weakly chordal graph from an arbitrary graph.

Chapter 4

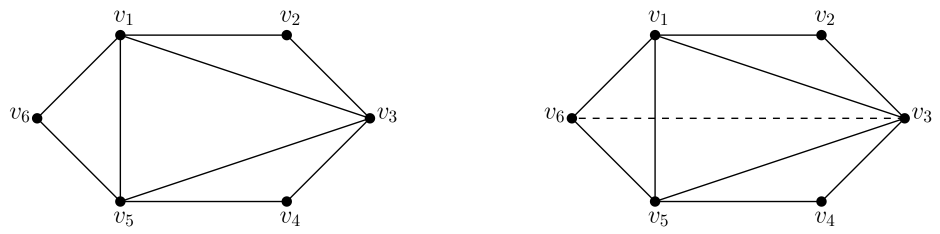
Strongly Chordal Graph Generation

Strongly chordal graphs are a subclass of the well-studied class of chordal graphs. The interest in this class stems from the fact that many hard problems are solvable in polynomial time for this class of graphs. In this chapter, we explore a number of different methods for generating strongly chordal graphs. This would be of interest if we were to test, for example, an implementation of a polynomial-time algorithm for k-tuple dominating sets for strongly chordal graphs [34]. To the best of our knowledge, there does not seem to exist any such generation algorithm in the literature. However, a number of different characterizations of strongly chordal graphs are known.

Farber [22] established a number of different characterizations that include one based on totally balanced matrices, another that is based on a class of forbidden induced subgraphs called trampolines, and a third based on the notion of a strong chord. These also include an intersection graph characterization [21] that is analogous to a similar characterization for chordal graphs [24]. In this chapter, we propose generation algorithms based on three of the characterizations listed above. In the first method, our algorithm first generates chordal graphs, using an available algorithm (see chapter 2) and then adds enough edges to make it strongly chordal, unless it is already so. The edge additions rely on the characterization that a certain neighborhood matrix of a strongly chordal graph is a totally balanced matrix (this is when the neighborhood matrix does not have $\left[\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ as a submatrix). The second generation method is based on the forbidden subgraph characterization of strongly chordal graphs. Our proposed algorithm starts with generating a trampoline and converts it into a strongly chordal graph by adding a minimum number of edges. Seker et al. [51] exploited the intersection graph characterization of chordal graphs to obtain an algorithm for generating them. Here, we propose an algorithm to show that strongly chordal graphs can also be generated using their intersection graph characterization. This is our third method for generating strongly chordal graphs. A characterization that is intuitive and close to the definition of a chordal graph as having no chordless 4-cycles is one that the based on the notion of a strong chord. A strong chord partitions the boundary of an even cycle of size 6 or more into two odd length paths. A graph $G = (V, E)$, is strongly chordal if and only if it is chordal and every even cycle of size 6 or more has a strong chord.

4.1 Definitions

A graph G is chordal if and only if there exists a perfect elimination ordering (or simplicial ordering, defined in section 2.1.2) of its vertices [23]. For instance, $v_2, v_6, v_4, v_1, v_3, v_5$ is a perfect elimination ordering of the vertices of the chordal graphs shown in Fig. 4.1. We



(a) Chordal graph but not strongly chordal graph (G_1)

(b) Strongly chordal graph(G_2)

Figure 4.1: *An example*

have an analogous characterization for a strongly chordal graph G. A vertex v of G is said to be *simple* if the sets in $\{N[u] : u \in N[v]\}$ can be linearly ordered by inclusion. An alternate definition is this. Vertices u and v of G are said to be *compatible* if $N[u] \subseteq N[v]$ or $N[v] \subseteq N[u]$ [22]. Then, a vertex v is simple if the vertices in $N[v]$ are pairwise compatible. None of the vertices of the graph in Fig. 4.1(a) is simple. Thus, v_2 is not simple as the vertices v_1 and v_3 in $N[v_2] = \{v_1, v_2, v_3\}$ are not pairwise compatible.

A *strong elimination ordering* of a graph $G = (V, E)$ is an ordering $v_1, v_2, \ldots, v_n$ of V such that the following condition holds: for each i, j, k, and l, if $i < j$, $k < l$ and $v_k, v_l \in N[v_i]$, and $v_k \in N[v_j]$, then $v_l \in N[v_j]$ [22]. Thus a graph G is strongly chordal if it admits a strong elimination ordering, which is a generalization of the notion of perfect elimination ordering used to define chordal graphs. Since the graph shown in Fig. 4.1(a) is not strongly chordal, it has no strong elimination ordering. On the other hand, $v_2, v_6, v_4, v_1, v_3, v_5$ is a strong elimination ordering of the vertices of the graph shown in Fig. 4.1(b).

The strong chord characterization also helps us distinguish Fig. 4.1(a) from Fig. 4.1(b). The graph in Fig. 4.1(a) is chordal but not strongly chordal because it has no strong chord (in the literature, this graph is known as the Hajos graph). On the other hand, the graph in Fig. 4.1(b) is strongly chordal, since $\{v_1, v_4\}$ (shown as a dashed line segment) is a strong chord.

The rest of the chapter is organized as follows. In the next section, we turn a chordal graph into a strongly chordal graph by adding edges. Next, in section 4.3, we propose a method based on the forbidden subgraph characterization to generate strongly chordal graphs. Section 4.4 presents an algorithm for the generation of strongly chordal graphs based on the intersection graph characterization.

4.2 First Method

Overview: We first generate a chordal graph on n vertices and m edges by using an existing algorithm (see chapter 2). Next, a perfect elimination ordering for this graph is computed using the lexicographic breadth-first search (Lex BFS) algorithm (discussed in section 2.1.2), proposed by Rose et al. in [48]. This perfect elimination ordering is used in the generation of strongly chordal graphs. Algorithm 4.1 transforms the input chordal graph into a strongly chordal graph and the perfect elimination ordering of the input chordal graph into a strong elimination ordering of the resulting strongly chordal graph.

4.2.1 Details

The *neighborhood matrix*, $M(G)$, of a graph G is an $n \times n$ matrix whose rows and columns are labeled by the vertices $v_1, v_2, \ldots, v_n$ and its (i, j)-th entry is 1 if $v_i \in N[v_j]$ and 0 otherwise. The ordering of the vertices $v_1, v_2, \ldots, v_n$ of G is a strong elimination ordering if and only if the matrix

$$\Delta = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

is not a submatrix of the neighborhood matrix, $M(G)$.

Our generation algorithm is based on the following observation.

Observation 4.1 [22] *The row (and column) labels of $M(G)$ correspond to a strong elimination ordering if and only if the matrix M does not contain Δ as a submatrix.*

Now, consider the following definition from Farber [22].

Definition 4.2 *A $(0, 1)$ matrix is said to be totally balanced if it does not contain as a submatrix the (edge-vertex) incidence matrix of a cycle of length at least three.*

The absence of Δ in M implies that M is totally balanced and the theorem below allows us to claim that G is strongly chordal.

Theorem 4.3 [22] *A graph G is strongly chordal if and only if $M(G)$ is totally balanced.*

Algorithm 4.1 *StronglyChordalGraphGeneration*

```
Input: A chordal graph $G$
Output: A strongly chordal graph $G$
 1: Call Lex BFS($G$) to generate a perfect elimination ordering
 2: Generate the neighborhood matrix $M(G)$ of $G$ from the perfect elimination ordering
 3: for $i \leftarrow 2$ to $n$ do                                   ▷ skipped first row
 4:     for $j \leftarrow 2$ to $n$ do                               ▷ skipped first column
 5:         if $M[i][j] == 0$ then
 6:             if ($M[i-1][j-1] == 1$ and $M[i-1][j] == 1$ and $M[i][j-1] == 1$) then
 7:                 $M[i][j] \leftarrow 1$                            ▷ switch 0 to 1
 8:                 $M[j][i] \leftarrow 1$                            ▷ symmetric
 9:             end if
10:         end if
11:     end for
12: end for
```

Algorithm 4.1 searches for the occurrences of Δ in $M(G)$, adding new edges to the graph whenever the 0 entry of a Δ-matrix is changed to a 1. The iteration continues until there is no $\left[\begin{smallmatrix}1 & 1\\ 1 & 0\end{smallmatrix}\right]$ submatrix in $M(G)$. On termination, the chordal graph turns into a strongly chordal graph and the perfect elimination ordering into a strong elimination ordering.

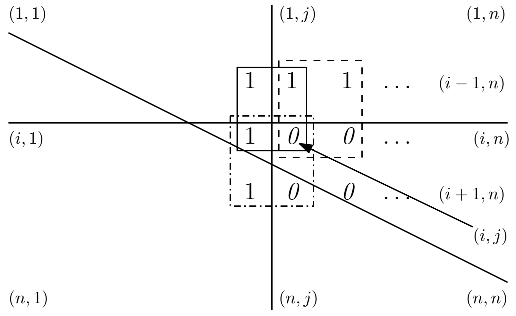
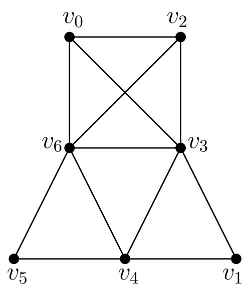
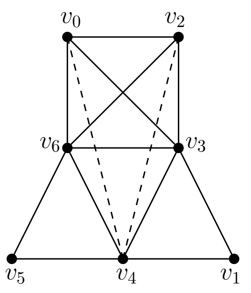


Figure 4.2: *Neighborhood matrix $M(G)$*

Figure 4.2 illustrates how the proposed algorithm works. If there is a 0 in the i^{th} row and j^{th} column and if ($M[i-1][j-1] == 1$ and $M[i-1][j] == 1$ and $M[i][j-1] == 1$), then we change the entry from 0 to 1 (which corresponds to the insertion of an edge in the graph). Say, $M[i][j+1] = 0$ as well then on changing $M[i][j]$ to 1, a $\left[\begin{smallmatrix}1 & 1\\ 1 & 0\end{smallmatrix}\right]$ submatrix is created in $M(G)$. Thus we set $M[i][j+1] == 1$. Since $M(G)$ must be symmetric, both the times we change 0 to 1 in symmetric positions.



(a) Chordal graph but not strongly chordal graph



(b) Strongly chordal graph

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 0 & 1 \\ \mathbf{1} & \boldsymbol{\mathit{0}} & \boldsymbol{\mathit{0}} & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(c) Neighborhood matrix $M(G)$ of the chordal graph is shown in Fig. (a)

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & \boldsymbol{\mathit{1}} & 0 & 1 \\ 0 & 1 & 1 & 1 & \boldsymbol{\mathit{1}} & 0 & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 0 & 1 \\ \mathbf{1} & \boldsymbol{\mathit{1}} & \boldsymbol{\mathit{1}} & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(d) Neighborhood matrix $M(G)$ of the strongly chordal graph is shown in Fig. (b)

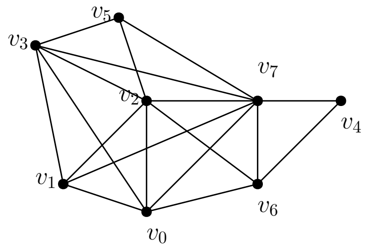
Figure 4.3: *An example of a chordal and a strongly chordal graph*

4.2.2 An Example

Figure 4.3 shows an example of a chordal graph and the strongly chordal graph generated by the above algorithm from this chordal graph. The graph shown in Fig. 4.3(a) is chordal but not strongly chordal as there is no strong chord (no edge $\{v_0, v_4\}$ and $\{v_2, v_4\}$) in the 6-cycles $\langle v_0, v_3, v_1, v_4, v_5, v_6, v_0\rangle$ and $\langle v_2, v_6, v_5, v_4, v_1, v_3, v_2\rangle$. Two occurrences of the $\left[\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ submatrix can be detected in the neighborhood matrix $M(G)$. According to the algorithm 4.1, by resetting the 0 entry to 1, in each of these occurrences, the submatrix does not occur any more in $M(G)$. This is equivalent to introducing two new edges $\{v_0, v_4\}$

and $\{v_2, v_4\}$ in the input graph. Now the updated graph has strong chords in the 6-cycles $\langle v_0, v_3, v_1, v_4, v_5, v_6, v_0\rangle$ and $\langle v_2, v_6, v_5, v_4, v_1, v_3, v_2\rangle$. The resulting graph is now strongly chordal and the perfect elimination ordering $v_1, v_0, v_2, v_3, v_4, v_5, v_6$ has turned into a strong elimination ordering.

Figure 4.4(a) shows another example of a chordal graph and the perfect elimination ordering is $v_5, v_1, v_3, v_0, v_2, v_4, v_6, v_7$. The graph also happens to be strongly chordal as there are no $\left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right]$ submatrices in its neighborhood matrix.



(a) Chordal graph and also strongly chordal graph.

$$\begin{bmatrix}1&0&1&0&1&0&0&1\\0&1&1&1&1&0&0&1\\1&1&1&1&1&0&0&1\\0&1&1&1&1&0&1&1\\1&1&1&1&1&0&1&1\\0&0&0&0&0&1&1&1\\0&0&0&1&1&1&1&1\\1&1&1&1&1&1&1&1\end{bmatrix}$$

(b) Neighborhood matrix $M(G)$ of the strongly chordal graph is shown in Fig. (a)

Figure 4.4: *An example of a chordal and a strongly chordal graph*

Theorem 4.4 *Algorithm 4.1 generates a strongly chordal graph, along with a strong elimination ordering.*

We know the ordering of the vertices $v_1, v_2, \ldots, v_n$ of a graph G is a strong elimination ordering if and only if the matrix $\left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right]$ is not a submatrix of the neighborhood matrix, $M(G)$ and the algorithm makes sure that no such submatrices are present in $M(G)$. Thus, the resulting graph is a strongly chordal.

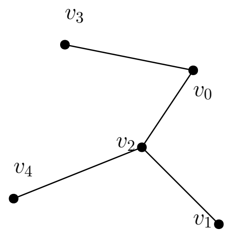
4.2.3 Complexity

The time complexity of Lex BFS is linear in the number of vertices, n, and the number of edges, m. The time complexity of algorithm 4.1 depends on the number of 0's in the matrix, $M(G)$. We can process these 0's in row-major order from left to right. For each 0, we check upwards in the column in which this 0 lies and to the left in the row in which it lies. If the row and column indices of a 0 are i and j, then this complexity is dominated by the sum of $i \times j$ taken over all the 0's in $M(G)$. Thus $O(n^4)$ is a rough upper bound on the time complexity of Algorithm 4.1.

4.2.4 Remarks

The proposed algorithm takes n vertices and m edges as input to generate strongly chordal graphs. It is interesting to note that if we start with a tree as the initial chordal graph, we cannot add new edges as a tree is also strongly chordal and thus, there is no Δ submatrix present in the neighborhood matrix of a tree. Hence we would get very sparse strongly chordal graphs. Figure 4.5(a) shows an example of a tree with a perfect elimination ordering v_3, v_0, v_1, v_2, v_4. The neighborhood matrix is shown in Fig. 4.5(b), from which we can see that there is no Δ submatrix present in its $M(G)$. An interesting challenge is to generate a strongly chordal graph without generating a chordal graph as an intermediate step.

For testing purposes, at the end of the strongly chordal graph generation process, the recognition algorithm 4.6 is applied to verify that the graph generated by the algorithm 4.1 is strongly chordal. A formal description of the recognition algorithm is given below:



(a) Tree and also strongly chordal graph.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

(b) Neighborhood matrix $M(G)$ of the tree is shown in Fig. (a).

Figure 4.5: *An example of a tree with neighborhood matrix*

```
Input: A graph $G = (V, E)$
Output: A strong elimination ordering
1: Set $n \leftarrow |V|$.
2: Let $V_0 = V$ and let $(V_0, <_0)$ be the partial ordering on $V_0$ in which $v <_0 u$ if and only
   if $v = u$. Let $V_1 = V$, and set $i \leftarrow 1$.
3: Let $G_i$ be the subgraph of $G$ induced by $V_i$. If $G_i$ has no simple vertex then output $G_i$
   and stop. Otherwise, define an ordering on $V_i$ by $v <_i u$ if $v <_{i-1} u$ or $N_i[v] \subset N_i[u]$.
4: Choose a vertex $v_i$ which is simple in $G_i$ and minimal in $(V_i, <_i)$. Let $V_{i+1} = V_i - \{v_i\}$.
   If $i = n$ then output the ordering $v_1, v_2, \ldots, v_n$ of $V$ and stop. Otherwise, set $i \leftarrow i + 1$
   and go to step 2.
```

Figure 4.6: *Strong elimination ordering [22]*

4.2.5 Arbitrary Graph to Strongly Chordal Graph

In section 2.3, we discussed algorithms to turn an arbitrary graph into a chordal graph and in the previous section, we generated chordal graphs as an intermediate step to generate strongly chordal graphs. By combining these two approaches, we have an algorithm to generate strongly chordal graphs from arbitrary graphs. Figure 4.7 shows the phases involved

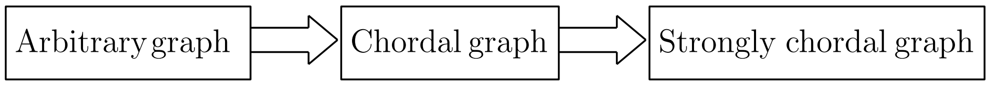
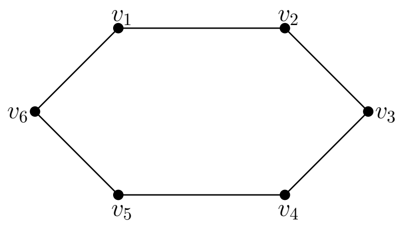


Figure 4.7: *Phases of the algorithm*

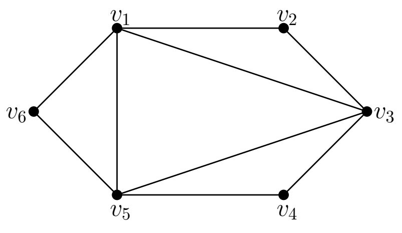
in the algorithm. A formal description of the algorithm is given below:

Algorithm 4.2 *ArbitraryToSCG*

```
Input: An arbitrary graph $G = (V, E)$ on $n$ vertices
Output: A strongly chordal graph $G$
1: Turn an arbitrary graph into a chordal graph by applying algorithm 2.10 (see chapter 2)
2: Apply algorithm 4.1 to turn the chordal graph into a strongly chordal graph
```



(a) Arbitrary graph (G)



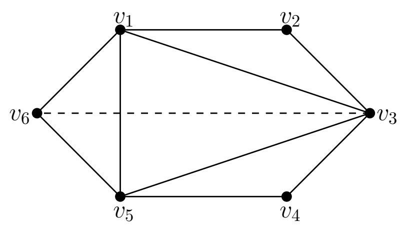
(b) Chordal graph (G')

$$\begin{bmatrix} 1 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 \\ 0 & 1 & 0 & \mathbf{1} & \boldsymbol{\mathit{0}} & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ \mathbf{1} & \mathbf{1} & 0 & 1 & 1 & 1 \\ \mathbf{1} & \boldsymbol{\mathit{0}} & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(c) Neighborhood matrix $M(G')$.

$$\begin{bmatrix} 1 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 \\ 0 & 1 & 0 & \mathbf{1} & \boldsymbol{\mathit{1}} & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ \mathbf{1} & \mathbf{1} & 0 & 1 & 1 & 1 \\ \mathbf{1} & \boldsymbol{\mathit{1}} & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(d) Neighborhood matrix $M(G'')$.



(e) Strongly chordal graph (G'')

Figure 4.8: *Arbitrary graph to strongly chordal graph*

The following example illustrates the generation of a strongly chordal graph from an arbitrary graph. All the phases, from left to right, are shown in Fig. 4.8. By applying algorithm 2.10, we turn the arbitrary graph (Fig. (a)) into a chordal graph (Fig. (b)). The corresponding $M(G')$ is shown in Fig. (c). The graph shown in Fig. 4.8(b) is chordal but not strongly chordal as there is no strong chord in the 6-cycle. Two occurrences of the $\left[\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ submatrix can be detected in the neighborhood matrix $M(G')$. According to the algorithm 4.1 (second step in algorithm 4.2), by resetting the 0 entry to 1, in each of these occurrences, the submatrix does not occur anymore in $M(G')$. This is equivalent to introducing an edge $\{v_3, v_6\}$ in the chordal graph (G'). Now the resulting graph has a strong chord in the 6-cycle. The resulting graph (G'') is now strongly chordal.

4.3 Second Algorithm

Overview: A trampoline is a chordal graph G on $2n$ vertices, for some $n \geq 3$, whose vertex set can be partitioned into two sets, $W = \{w_1, w_2, \dots, w_n\}$ and $U = \{u_1, u_2, \dots, u_n\}$, so that W is independent and for each i and j, w_i is adjacent to u_j if and only if $i = j$ or $i = j + 1($ mod $n)$ and $G[U]$ is a complete graph [22]. Fig. 4.9(a) and 4.10(a) show trampolines on 8 and 10 vertices, respectively.

The algorithm of this section is based on the following forbidden subgraph characterization of strongly chordal graphs by Farber:

Theorem 4.5 [22] *A chordal graph is strongly chordal if and only if it contains no induced trampoline.*

Since a trampoline is provably chordal (any permutation of the vertices in W, followed by any permutation of the vertices in U is a perfect elimination ordering), we exploit Theorem 4.5 to generate a strongly chordal graph from a trampoline and then introduce additional edges by applying the completion algorithm. In section 4.3.3, we extend the scope of this method to generate dense strongly chordal graphs from a network of trampolines.

4.3.1 Details

We first explain how to turn a trampoline on $2n$ vertices into a strongly chordal graph by adding edges strategically.

Algorithm 4.3 *TrampolineToSCG*

```
Input: A trampoline $G = (V, E)$ on $n$ vertices
Output: A strongly chordal graph $G$
1: Choose an independent vertex $w_i$ from set $W$
2: Find the neighbors of $w_i$ in the set $U$
3: Find the vertices in the set $U$ those are not in the neighborhood of $w_i$ and add edges
   from $w_i$ to those vertices
```

From an alternate characterization of strongly chordal graphs, we know that if every even cycle of length at least 6 in a chordal graph G has a strong chord, then G is necessarily strongly chordal and hence cannot contain an induced trampoline. Thus the strategy is to introduce strong chords in even cycles of size 6 and larger. Consider the trampoline shown in Fig. 4.9(a). There is no strong chord for the even cycle of length 8 $\langle w_1, u_1, w_2, u_2, w_3, u_3, w_0, u_0, w_1 \rangle$. Algorithm 4.3 adds edges by joining one of the independent vertices w_i to its non-neighbor vertices in the set U. In this example, two edges $\{w_1, u_3\}$ and $\{w_1, u_2\}$ are added to turn the trampoline into a strongly chordal graph (see Fig. 4.9(b)). While $\{u_0, u_2\}$ or $\{u_3, u_1\}$ is a strong chord for every even cycle of length 6, the outer 8-cycle has no strong chord. Joining w_1 and u_3 splits this 8-cycle into a 4-cycle $\langle w_1, u_0, w_0, u_3, w_1 \rangle$ and a 6-cycle $\langle w_1, u_1, w_2, u_2, w_3, u_3, w_1 \rangle$. However, there is no strong chord in the newly created 6-cycle. This is rectified by adding an edge between w_1 and u_2. This approach can be extended to any trampoline of size $n \geq 3$ and ensures a strong

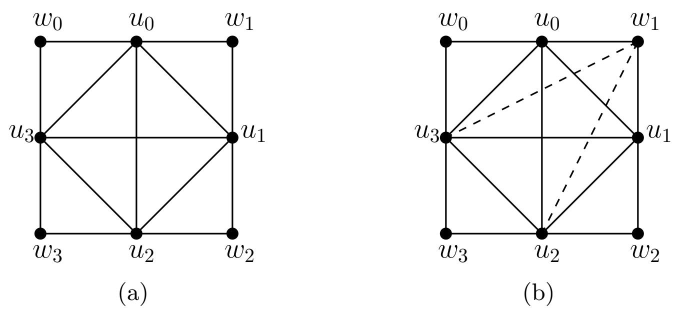


Figure 4.9: *(a) Trampoline(G) (b) Strongly chordal graph(G′)*

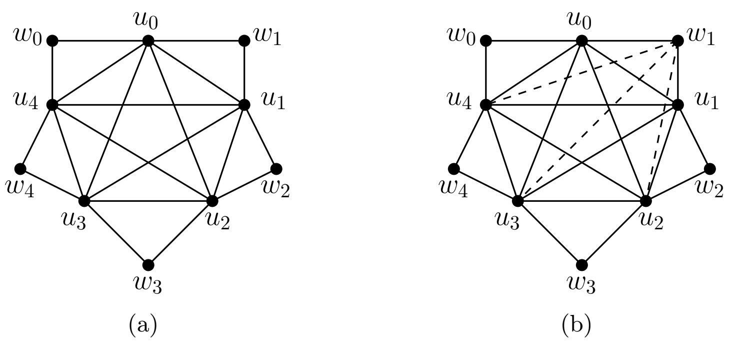


Figure 4.10: *(a) Trampoline(G) (b) Strongly chordal graph(G′)*

chord in every even cycle of length 6 and more. Clearly, given a trampoline of size $2n$, by adding $n-2$ strong chords, we can turn it into a strongly chordal graph. Another example is shown in Fig. 4.10, where a trampoline of size 10 is turned into a strongly chordal graph by adding 3 strong chords. In [41], Odom showed that strongly chordal graphs, in addition to other graph classes like chordal graphs, constitute a completion class. This allows us to add an edge at a time to a strongly chordal graph to reach the complete graph, remaining in the class all throughout. The completion Algorithm 4.11 is based on the theorem below.

Theorem 4.6 [41] *Let $G = (V, E)$ be a connected graph of order n and size m. Let $G_0 = G$, and define the sequence of graphs $G_0, G_1, \ldots, G_s$ using Algorithm 4.11. If α is a strong elimination ordering for G, then α is a strong elimination ordering for each G_i,*

$i = 1, 2, \ldots, s$.

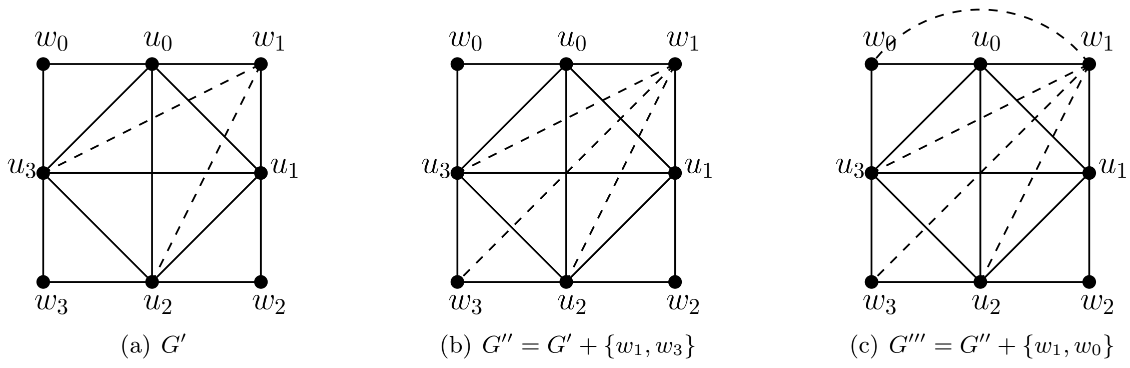
```
**Input:** A strongly chordal graph $G$, a strong elimination ordering $\alpha$, and the number of
  edges $m$
**Output:** A strongly chordal graph $G$
  $G_0 \leftarrow G$
  $E_0 \leftarrow E$
  $s \leftarrow m - m'$     $\triangleright$ $m'$ is the no. of edges in $G$ (before applying the completion algorithm)
  **for** $i \leftarrow 1$ to $s$ **do**
      $k_i \leftarrow \max\{j | \deg(v_j) < n - 1\}$
      $m_i \leftarrow \max\{l | v_{k_i}, v_l \notin E_{i-1}\}$
      $e_i \leftarrow v_{k_i}, v_{m_i}$
      $E_i \leftarrow E_{i-1} \cup \{e_i\}$
      $G_i \leftarrow (V, E_i)$
  **end for**
```

Figure 4.11: *SCGCompletion [41]*

The completion algorithm takes a strongly chordal graph, a strong elimination ordering, and the number of edges (m) as input and produces a strongly chordal graph. We use the recognition algorithm by Farber, described in section 3.1.2, to generate a strong elimination ordering. Based on the strong elimination ordering, we choose a pair of vertices k_i and m_i according to the conditions mentioned in the algorithm and introduce edges between them. We iterate s times to add s additional edges to meet the target of m edges.

4.3.2 An Example

We illustrate the completion algorithm by means of an example. Consider the strongly chordal graph (G') shown in Fig. 4.9(b) where $m' = 16$. Let $m = 18$. To add two ($s = 2$) more edges we use the completion algorithm. From Farber's recognition algorithm, we obtain the following strong elimination ordering: w_2, w_0, w_3, u_2, u_1, u_0, u_3, w_1. We choose a pair of vertices from the ordering with no edge between them. In the first iteration, we introduce an edge between u_0 and w_3 and in the next iteration, we introduce an edge between u_0 and w_2. This gives a strongly chordal graph G''' for the given n and m.



(a) G' (b) $G'' = G' + \{w_1, w_3\}$ (c) $G''' = G'' + \{w_1, w_0\}$

Figure 4.12: *Strongly chordal graph generation*

4.3.3 Network of Trampolines

To generate a greater variety of strongly chordal graphs, we construct a trampoline network and apply Algorithm 4.3 as a subroutine to turn each trampoline of this network into a strongly chordal graph. A network of trampolines and a strongly chordal graph derived from it are shown in Fig. 4.13.

4.3.4 Complexity

To create a trampoline of size $2n$ takes $O(n^2)$ time, this being the size of an adjacency list to represent this graph. The complexity of the completion procedure is in $O(n^3)$, this being the complexity of generating a strong elimination ordering using Farber's algorithm. Thus the time complexity of this method is in $O(n^3)$ for generating a strongly chordal graph from a single trampoline. The time complexity of generating a strongly chordal graph from a trampoline network is in $O(k + \Sigma_{i=1}^{k} n_i{}^3)$, where k is the number of nodes in the trampoline

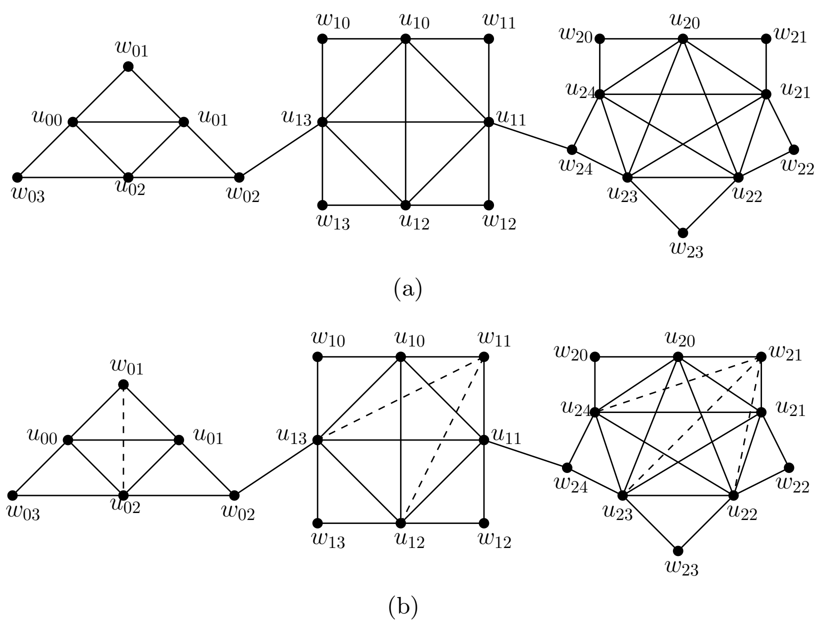


Figure 4.13: *(a) Network of Trampoline(G) (b) Strongly chordal graph(G′)*

network and n_i is the size of the i-th trampoline.

4.4 Third Algorithm

Preamble: Farber's intersection graph characterization [21] of strongly chordal graphs is analogous to a similar characterization for chordal graphs by Gavril [24]. Seker et al. [51] exploited this characterization of chordal graphs to obtain an algorithm for generating them. In this section, we propose an algorithm to show that strongly chordal graphs can also be generated using their intersection graph characterization.

The following essential definitions from Farber [21] underlie this characterization. Let r be the root of an edge-weighted tree T. The edge-weights are positive numbers that can be conveniently interpreted as the lengths of the edges.

Definition 4.7 [21] *The weighted distance from a node u to a node v in T, denoted by*

$d^*_T(u,v)$, is the sum of the lengths of the edges of the (unique) path from u to v.

Definition 4.8 [21] *Let T_1 and T_2 be two subtrees of T. Subtree T_1 is full with respect to T_2, denoted by $T_1 > T_2$, if for any two vertices $u, v \in T_2$ such that $d^*_T(r,u) \leq d^*_T(r,v)$, $v \in V(T_1)$ implies that $u \in V(T_1)$.*

Definition 4.9 [21] *A collection of subtrees $\{T_1, T_2, \ldots, T_n\}$ of T is compatible if for each pair of subtrees T_i and T_j either $T_i > T_j$ or $T_j > T_i$.*

Using the definitions above, Farber established the following intersection graph characterization for strongly chordal graphs.

Theorem 4.10 [21] *A graph is strongly chordal if and only if it is the intersection graph of a compatible collection of subtrees of a rooted, weighted tree, T.*

4.4.1 The Algorithm

Let S be an adjacency matrix whose rows correspond to a compatible collection of subtrees, $\{T_1, T_2, \ldots, T_k\}$ of a rooted, weighted tree T as in Theorem 4.10 and columns correspond to the vertices $\{v_1, v_2, \ldots, v_n\}$ of T, arranged from left to right in order of non-decreasing distance from the root, r. Our main observation is that Definition 4.8 can be re-interpreted to imply that the matrix S cannot have $\Delta_1 = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ as a sub-matrix. More precisely, if i and j are two rows of S, corresponding to compatible subtrees T_i and T_j of T, then there cannot exist columns k and l that intersect these two rows to create Δ_1. Thus S belongs to the class of 0-1 matrices that do not have Δ_1 as a submatrix. Note that this is only a necessary condition. If we can generate a 0-1 matrix that satisfies this necessary condition, we have to ensure that each row corresponds to a subtree of a weighted tree T. The details of how this can be achieved are described in the algorithm below that is built atop our observation of the forbidden sub-matrix property of S.

Each of the entries of the first row and the first column are randomly set to 0 or 1. The entries of the submatrix $[2 \ldots k, 2 \ldots n]$ are carefully set to 0 or 1 so as not to have Δ_1 as a submatrix. This is done in row-major order. While setting the entry of the i-th row and j-th column we check the entries exhaustively in the columns to the left of the j-th columns and the entries above the i-th row to make sure that no 2×2 submatrix is equal to Δ_1. To have a compatible collection of subtrees $\{T_1, T_2, \ldots, T_n\}$ of a given tree T, we also do not want $\Delta_2 = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ as a submatrix. Such a submatrix can create cycles in the tree T we wish to construct from the rows of our matrix representing a collection of compatible subtrees. Algorithm 4.4 generates a $0-1$ matrix S without Δ_1 or Δ_2 as a submatrix.

In the next phase, we prune some of the rows of S. First, we remove rows with all 0's. Then we remove duplicate rows (if any) because they produce identical subtrees and denote the matrix as S'. In the next step, we generate a strongly chordal graph from the matrix S'. Each subtree (row) represents a vertex in the strongly chordal graph, and there is an edge between two vertices in the strongly chordal graph if $T_i \cap T_j \neq \emptyset$. Algorithm 4.5 takes the number of columns (nodes) n and the number of rows (subtrees) k $(k \leq n)$ as inputs and outputs a strongly chordal graph G.

Algorithm 4.4 $0-1$*MatrixGeneration*

Input: The number of rows (subtrees) r and the number of columns (nodes) c

Output: A matrix S

```
1:  for $i \leftarrow 1$ to $n$ do
2:      for $j \leftarrow 1$ to $n$ do
3:          $randomEntry \leftarrow random(0,1)$            ▷ randomly chosen either 0 or 1
4:          if $i == 1$ or $j == 1$ then
5:              $S[i][j] \leftarrow randomEntry$
6:          else
7:              if $randomEntry == 1$ then
8:                  if ($S[i-1][j-1] == 1$ and $S[i-1][j] == 1$ and $S[i][j-1] == 0$) or
                    ($S[i-1][j-1] == 0$ and $S[i-1][j] == 1$ and $S[i][j-1] == 1$) then
9:                      $randomEntry \leftarrow 0$            ▷ switch 1 to 0
10:                     $S[i][j] \leftarrow randomEntry$
11:                 else
12:                     $S[i][j] \leftarrow randomEntry$            ▷ keep 1 as a valid entry
13:                 end if
14:             else
15:                 $S[i][j] \leftarrow randomEntry$            ▷ keep 0 as a valid entry
16:             end if
17:         end if
18:     end for
19: end for
```

Algorithm 4.5 *StronglyChordalGraphGenerationFromSubtrees*

Input: The number of columns (nodes) n and the number of rows (subtrees) k

Output: A strongly chordal graph G

```
1: Call $0-1$ matrix generation algorithm to generate a matrix without the sub matrix
   $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$
2: Remove any rows with zeros only
3: Remove any duplicate rows
4: Generate a strongly chordal graph from rows by computing the intersection of each row
```

In the following paragraphs, we explain on an example, the strongly chordal graph generation process step-by-step.

4.4.2 An Example

Algorithm 4.4 generates the following $0-1$ matrix S with $k = 12$ and $n = 12$.

$$S = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

After removing 6 rows (row index: 5^{th}, 7^{th}, 9^{th}, 10^{th}, 11^{th}, 12^{th}) that have only zero entries, we get the following matrix S':

$$S' = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

From the matrix S', we can see there are six subtrees. The subtrees are shown in Fig. 4.14. The strongly chordal graph shown in Fig. 4.15 is generated by representing each T_i of Fig. 4.14 as a node v_{T_i}. There is an edge between v_{T_i} and v_{T_j} if the intersection of T_i and T_j is non-empty.

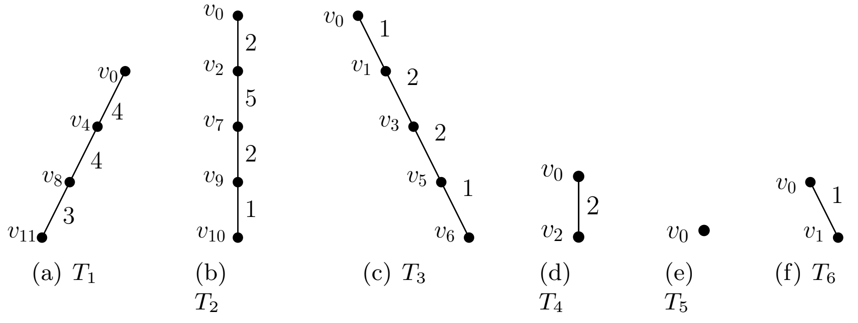


Figure 4.14: *Subtrees* $\{T_1, T_2, T_3, T_4, T_5, T_6\}$

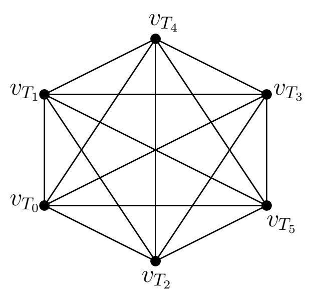


Figure 4.15: *A strongly chordal graph generated from subtrees* $\{T_1, T_2, T_3, T_4, T_5, T_6\}$

4.4.3 Complexity

Algorithm 4.4 takes $O(n^4)$ time to generate matrix S, ensuring it does not have Δ_1 or Δ_2 as a submatrix. The intersection of two subtrees $(T_i \cap T_j)$ can be computed in $O(n)$ time. Each subtree represents a vertex in a strongly chordal graph and if $T_i \cap T_j \neq \emptyset$, then there is an edge between two vertices in a strongly chordal graph. The insertion of an edge can be done in $O(1)$ time.

4.4.4 Discussion

In this chapter, we have presented three novel algorithms for the strongly chordal graph generation based on the three different characterizations. It would be interesting to improve on the time complexities of these algorithms or find more efficient ways of generating strongly chordal graphs. We implemented all the proposed algorithms in Python. Apropos the third algorithm, it is interesting to point out that we tested a large number of intersection graphs generated from Δ_1-free matrices. Without exception, all of these passed the recognition algorithm test for strong chordality. It would be worthwhile to investigate this further.

4.5 Summary

In this chapter, we have presented three different methods for the strongly chordal graph generation based on the three different characterizations.

Chapter 5

Semi-dynamic Algorithms for Strongly Chordal Graphs

There is an extensive literature on dynamic algorithms for a large number of graph-theoretic problems, particularly for all varieties of shortest path problems. Germane to this chapter are a number of fully dynamic algorithms that are known for chordal graphs [30, 38]. However, to the best of our knowledge, no study has been done for the problem of dynamic algorithms for strongly chordal graphs. To address this gap, in this chapter, we propose a semi-dynamic algorithm for edge-deletions and a semi-dynamic algorithm for edge-insertions in a strongly chordal graph.

The rest of the chapter is structured as follows. In the next section, we explain the design of a semi-dynamic algorithm for deletions and in section 5.2, we discuss the design of a semi-dynamic algorithm for insertions. Finally, section 5.3 contains concluding remarks and open problems.

5.1 Semi-dynamic Algorithm for Deletions

Let $G = (V, E)$ be a strongly chordal graph and C an even cycle of size six or greater in G. A chord $\{u, v\}$ of C is a strong chord if a shortest distance between u and v along C, $d_C(u, v)$, is odd. The deletion algorithm is based on the following characterization of a strongly chordal graph.

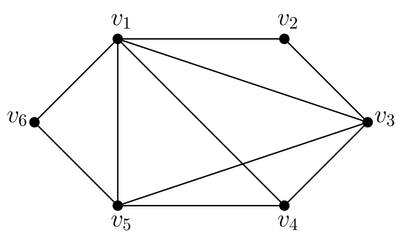
Theorem 5.1 [22] *A graph G is strongly chordal if and only if it is chordal and every even cycle of length at least 6 in G has a strong chord.*

Let $e = \{u, v\}$ be an arbitrary edge of G. Then e can be deleted from G provided $G - e$ remains chordal and it is not the only strong chord of a 6-cycle. The check for chordality exploits the following theorem.

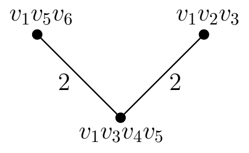
Theorem 5.2 [30] *Let e be an edge of a chordal graph G. Then $G - e$ remains chordal if and only if G has exactly one maximal clique containing e.*

Since strongly chordal graphs are a subclass of chordal graphs, a clique tree data structure, T, representing G, is used to check for the chordality condition.

Consider the chordal graph shown in Fig. 5.1(a) that has three maximal cliques $v_1v_2v_3$, $v_1v_3v_4v_5$, and $v_1v_5v_6$. Each maximal clique is represented by a node in the tree T and the weight of each edge is the size of the overlap of the two maximal cliques that it joins. To obtain a clique tree T from G, we use an expanded version of the Maximum Cardinality Search (MCS) algorithm (see section 2.1.5) by Blair and Peyton [11]. If an edge e is present



(a) A chordal graph (G)



(b) A clique tree (T)

Figure 5.1: *An example*

in two or more clique tree nodes, then we cannot delete e as its deletion will violate the chordality property of G. For instance, we cannot delete the edge $\{v_1, v_5\}$ or the edge $\{v_1, v_3\}$ from G (see Fig. 5.1) because either deletion will make G non-chordal.

Thus, by maintaining the clique tree T, we can determine if an edge can be deleted without violating chordality. The details are as follows. For each node in T, we compute the intersection of the node (a maximal clique contains two or more vertices of G) with the edge e. If we find T has exactly one node containing the end-points of e, then we continue and check if the deletion preserves strong chordality as well.

As explained earlier, an edge e can be deleted if and only it is not the only strong chord of a 6-cycle. For instance, consider the strongly chordal graph shown in Fig. 5.2. The edge $\{v_0, v_5\}$ (shown as a dashed line segment) splits the 8-cycle $\langle v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_0\rangle$ into a 4-cycle, $\langle v_0, v_5, v_6, v_7, v_0\rangle$ and a 6-cycle, $\langle v_0, v_5, v_4, v_3, v_2, v_1, v_0\rangle$. The addition of a strong chord $\{v_1, v_4\}$ (shown as a dashed line segment) splits the 6-cycle $\langle v_0, v_5, v_4, v_3, v_2, v_1, v_0\rangle$ into two 4-cycles, $\langle v_0, v_5, v_4, v_1, v_0\rangle$ and $\langle v_1, v_4, v_3, v_2, v_1\rangle$. Alternatively, we could interpret $\{v_0, v_5\}$ as a strong chord of the 6-cycle $\langle v_0, v_7, v_6, v_5, v_4, v_1, v_0\rangle$, post the introduction of $\{v_1, v_4\}$ as a strong chord of the initial 8-cycle.

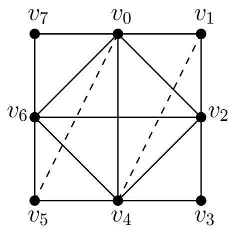


Figure 5.2: *A strongly chordal graph*

Since the addition of a strong chord splits an even cycle of size 6 or greater into two odd-length paths, for the next part of the test, it is sufficient to check for the presence of a strong chord in every even cycle of length 6. The formal proof of this is given below where we justify why it is enough to check if an edge e that is a candidate for deletion is a strong chord of a 6-cycle.

Definition 5.3 *Let C_k denote a cycle with k edges.*

Definition 5.4 *An ensemble $\mathcal{E}$ of strong chords of an even cycle with $2n$ edges, C_{2n}, is a set of $n-2$ strong chords that are pairwise disjoint, except for common endpoints.*

Two different ensembles of strong chords are shown in Fig. 5.3 for an 8-cycle, C_8.

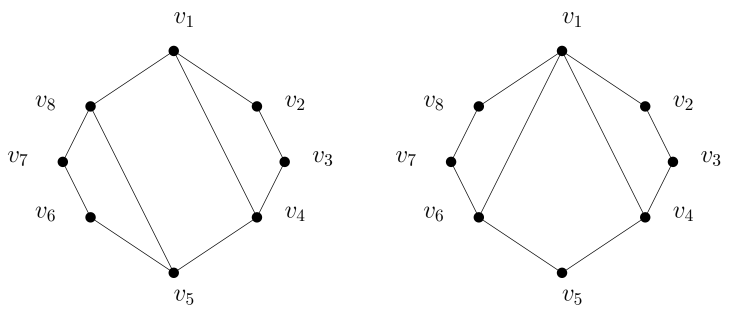


Figure 5.3: *Two different ensembles of strong chords of an 8-cycle*

Lemma 5.5 *A strong chord of a cycle C_{2n} belongs to an ensemble of $n-2$ strong chords, $\mathcal{E}$.*

Proof: Let $\{v_1, v_k\}$, for $k \geq 4$ and even, be a strong chord of an even cycle $C_{2n} = \langle v_1, v_2, \ldots, v_{2n} \rangle$ of length $2n$, where $n \geq 3$. The proof is by induction on n. Clearly as a single strong chord splits a 6-cycle into two cycles of length 4, the claim is true for $n = 3$. Assume that the claim is true for a cycle of length $2(n-1)$. Since $\{v_1, v_k\}$ splits C_{2n} into two cycles C_k and $C_{k'}$ of even lengths k and $k' = 2n - k + 2$, by the inductive hypothesis, there exists a disjoint ensemble of $k/2 - 2$ strong chords that partition C_k and a disjoint ensemble of $(2n-k+2)/2 - 2$ strong chords that partition C_{2n-k+2}. Since the strong chord $\{v_1, v_k\}$ is not counted, the total number of strong chords that partition C_{2n} is: $k/2 - 2 + (2n-k+2)/2 - 2 + 1 = n - 2$. This proves the assertion.

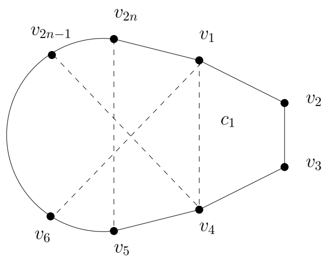


Figure 5.4: *Every strong chord is a strong chord of a 6-cycle*

Theorem 5.6 *Each of the strong chords in an ensemble $\mathcal{E}$ of strong chords of C_{2n} is a strong chord of a 6-cycle.*

Proof: Once again, the proof is by induction on n. This is true for C_6, as the ensemble $\mathcal{E}$ has only one strong chord. Assume the claim holds for an even cycle of smaller length. Among the ensemble $\mathcal{E}$ of strong chords of C_{2n} there is one, say c_1, that forms a C_4 with three boundary edges (see Fig. 5.4). By the inductive hypothesis, in the even cycle formed by the rest of the $2n-3$ edges and c_1 each of the strong chords of the residual ensemble $\mathcal{E}-\{c_1\}$ is a strong chord of a 6-cycle, C_6. To show that c_1 is also a strong chord of a 6-cycle, we observe from Fig. 5.4, that c_1 is a strong chord of the 6-cycle, $\langle v_{2n}, v_1, v_2, v_3, v_4, v_5\rangle$, or of the 6-cycle $\langle v_{2n}, v_1, v_2, v_3, v_4, v_5\rangle$ or of the 6-cycle $\langle v_{2n}, v_1, v_2, v_3, v_4, v_5\rangle$ and $\{v_{2n}, v_5\}$ or $\{v_1, v_6\}$ or $\{v_{2n-1}, v_4\}$ is a strong chord in $\mathcal{E}-\{c_1\}$. This completes the proof.

A potential 6-cycle of which $e=\{u,v\}$ is a strong chord, is formed by disjoint pairs of P_4-paths that go from u to v. Thus we determine all P_4-paths and for every disjoint pair of these, we check whether $\{u,v\}$ is the only strong chord or not. If there is a strong chord other than $\{u,v\}$ in every disjoint pair of P_4-paths, then the edge $\{u,v\}$ can be deleted. On the other hand, if $\{u,v\}$ is the only strong chord for any disjoint pair of P_4-paths, then the edge $\{u,v\}$ cannot be deleted. To find all P_4 paths between u and v, we compute the adjacency matrix of a bipartite graph, one part consisting of the neighbors of u and the other part consisting of the neighbors of v (see Fig. 5.6).

Algorithm 5.1 *Delete*

```
Input: A strongly chordal graph $G$ and an edge $\{u,v\}$ to be deleted
Output: A strongly chordal graph $G-\{u,v\}$
1: if $Delete-Query(G,T,u,v)$ returns "True" then
2:     Delete the edge $\{u,v\}$ from $G$
3:     Call UpdateCliqueTreeAfterDeletion
4: end if
```

Algorithm 5.2 returns “True” and the node x, if $\{u,v\}$ can be deleted from G, in which case we perform the delete operation, described in algorithm 5.1. After performing the delete operation, we update both the clique tree and the graph. The clique tree node that

Algorithm 5.2 *Delete-Query*

```
Input: A strongly chordal graph $G$, a clique tree $T$ of $G$, and an edge $\{u, v\}$ to be deleted
Output: Return True or False
1: $canBeDeleted \leftarrow$ False
2: if the edge $\{u, v\}$ does not exist then
3:     return $canBeDeleted$
4: else
5:     if the edge $\{u, v\}$ belongs to exactly one node ($x$) in $T$ then
6:         if $\{u, v\}$ is not a strong chord then
7:             $canBeDeleted \leftarrow$ True
8:             return $canBeDeleted$ and the node $x$
9:         else
10:            return $canBeDeleted$
11:        end if
12:    else
13:        return $canBeDeleted$
14:    end if
15: end if
```

contains the edge $\{u, v\}$ can be replaced with 0, 1, or 2 nodes. The algorithm *UpdateCliqueTreeAfterDeletion* due to Ibarra [30] deletes the edge $\{u, v\}$ and updates T.

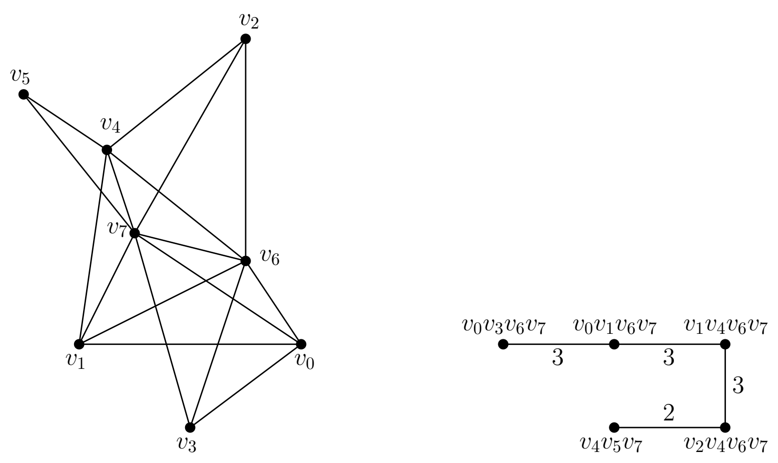
5.1.1 An Example

Consider deleting the edge $\{v_1, v_4\}$ from the strongly chordal graph G of Fig. 5.1(a). First, we check if chordality is preserved. From the clique tree T (see Fig. 5.1(b)), we observe there is only one node that contains the edge $\{v_1, v_4\}$. This satisfies the chordality condition and now we check for the strong chordality condition. For this, we compute the neighborhoods of v_1 and v_4 where $N(v_1) = \{v_2, v_3, v_4, v_5, v_6\}$ and $N(v_4) = \{v_1, v_3, v_5\}$. It is now easy to see that there are three P_4-paths, $[v_1, v_2, v_3, v_4]$, $[v_1, v_5, v_3, v_4]$, and $[v_1, v_6, v_5, v_4]$ between v_1 and v_4. We note that $\{v_1, v_4\}$ is the only strong chord in the graph. Hence the deletion of $\{v_1, v_4\}$ is not allowed. But were any of the other two strong chords, $\{v_2, v_5\}$ or $\{v_3, v_6\}$, been present in the graph we could delete $\{v_1, v_4\}$.

Algorithm 5.3 *UpdateCliqueTreeAfterDeletion* [30]

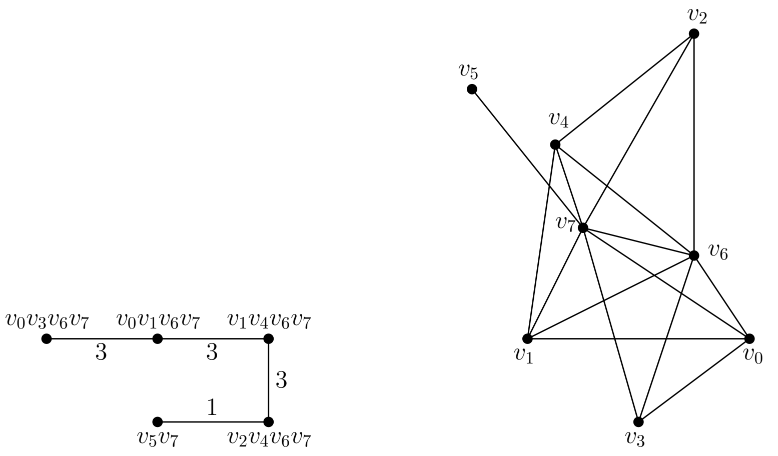
```
Input: A clique tree $T$ and an edge $\{u, v\}$ to be deleted
Output: An updated clique tree $T$
 1: For every $y \in N(x)$, test whether $u \in K_y$ or $v \in K_y$ and whether $w(x, y) = k - 1$ ▷ $K_x$
    and $K_y$ are maximal cliques and $w(x, y) = |K_x \cap K_y|$
 2: Replace node $x$ with new nodes $x_1$ and $x_2$ respectively representing $K_x^u$ and $K_x^v$ and
    add edge $\{x_1, x_2\}$ with $w(x_1, x_2) = k - 2$          ▷ $K_x^u = K_x - \{v\}$ and $K_x^v = K_x - \{u\}$
 3: if $y \in N_u$ then                                   ▷ $N_u = \{y \in N(x) \mid u \in K_y\}$
 4:     replace $\{x, y\}$ with $\{x_1, y\}$
 5: end if
 6: if $z \in N_v$ then          ▷ $N_v = \{z \in N(x) \mid v \in K_z\}$ and $K_z$ is a maximal clique
 7:     replace $\{x, z\}$ with $\{x_2, z\}$
 8: end if
 9: if $w \in N_{\overline{uv}}$ then      ▷ $N_{\overline{uv}} = \{w \in N(x) \mid u, v \notin K_w\}$ and $K_w$ is a maximal clique
10:     replace $\{x, w\}$ with $\{x_1, w\}$ or $\{x_2, w\}$ (chosen arbitrarily)
11: end if
12: if $K_x^u$ and $K_x^v$ are both maximal in $G - \{u, v\}$ then
13:     return the updated clique tree $T$
14: end if
15: if $K_x^u$ is not maximal because $K_x^u \subset K_{y_i}$ for some $y_i \in N_u$ then
16:     choose one such $y_i$ arbitrarily, contract $\{x_1, y_i\}$, and replace $x_1$ with $y_i$
17: end if
18: if $K_x^v$ is not maximal because $K_x^v \subset K_{z_i}$ for some $z_i \in N_v$ then
19:     choose one such $z_i$ arbitrarily, contract $\{x_2, z_i\}$, and replace $x_2$ with $z_i$
20: end if
21: return the updated clique tree $T$
```

For another example, consider the graph, shown in Fig. 5.5(a). Here, chordality is preserved if we delete the edge $\{v_7, v_2\}$. However, as it is the only strong chord for the 6-cycles $\langle v_7, v_0, v_6, v_2, v_4, v_5, v_7\rangle$ and $\langle v_7, v_3, v_6, v_2, v_4, v_5, v_7\rangle$, it cannot be deleted. The edge $\{v_5, v_4\}$, though, can be deleted from the graph. A check of the clique tree shows that only a single node contains the edge $\{v_5, v_4\}$. Proceeding to check for the strong chordality condition, we note that $\{v_5, v_4\}$ is not a strong chord and thus can be deleted. The updated clique tree T and the updated graph G are shown in Fig. 5.5(c) and Fig. 5.5(d), respectively.



(a) A strongly chordal graph (G)

(b) A clique tree (T) of G



(c) T after deleting $\{v_4, v_5\}$

(d) $G - \{v_4, v_5\}$

Figure 5.5: *An example*

5.1.2 Complexity of Deletions

Using the expanded version of the MCS algorithm, the preprocessing time required to construct a clique tree T is in $O(n + m)$. To check if an edge $\{u, v\}$ belongs to exactly one maximal clique in T, we perform a set intersection operation with the vertices of a maximal clique and the vertices of the edge $\{u, v\}$. The time for this is linear in the sizes of the two sets and is thus bounded by $O(n)$. Thus the query complexity for this part is in $O(n)$.

To determine if $\{u, v\}$ is a strong chord of G, we have to find all P_4 paths between u and

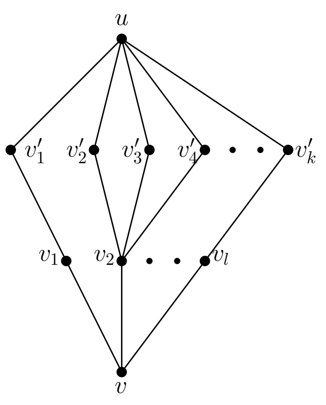


Figure 5.6: *Estimating the number of P_4 paths from u to v*

v. It can be seen from Fig. 5.6 that the number of such paths is bounded above by $O(d_u d_v)$, where d_u and d_v are the degrees of the vertices u and v respectively. The actual paths can also be found out in $O(d_u d_v)$ time by computing the adjacency matrix of a bipartite graph, one part consisting of the neighbors of u and the other part consisting of the neighbors of v, whose adjacencies are determined from G. Clearly, we have a P_4 path for each entry 1 in the adjacency matrix. Thus the number of paths and the time complexity of finding these are both bounded by $O(d_u d_v)$. The next step is to find out if $\{u, v\}$ is not the only strong chord of each 6-cycle determined by disjoint pairs of these P_4 paths, a task that can be accomplished in $O(d_u^2 d_v^2)$ time. Thus the query complexity of this step is in $O(d_u^2 d_v^2)$.

Using an implementation of a fully dynamic algorithm by Ibarra [30] for maintaining chordal graphs, the clique tree can be updated in $O(n)$ time. As we also maintain an adjacency matrix representation of G for the strongly chordal graph query, an update to this structure can be done in $O(1)$ time. Thus the overall query update time is in $O(n)$.

5.2 Semi-dynamic Algorithm for insertions

Let G be a strongly chordal graph and $\alpha = v_1, v_2, \ldots, v_n$ be a strong elimination ordering (defined in section 4.1) of its vertices, V. The existence of such an ordering is characteristic of G. The neighborhood matrix $M(G)$ of G, based on α, is an $n \times n$ matrix whose (i, j)-th entry is 1 if $v_i \in N[v_j]$ and is 0 otherwise. Let Δ be the submatrix:

$$\Delta = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Our dynamic insertion algorithm is based on the following observation.

Observation 5.7 [22] *The row (and column) labels of $M(G)$ correspond to a strong elimination ordering if and only if the matrix M does not contain Δ as a submatrix.*

Farber [22] defined a 0-1 matrix M to be *totally balanced* if it does not contain a submatrix that can be interpreted to be the edge-vertex incidence matrix of a cycle of size 3 or greater.

The absence of Δ in $M(G)$ implies that $M(G)$ is totally balanced and the theorem below allows us to claim that G is strongly chordal.

Theorem 5.8 [22] *A graph G is strongly chordal if and only if $M(G)$ is totally balanced.*

Thus if G is strongly chordal, then $G + \{u, v\}$ remains so if inserting the edge $\{u, v\}$ into G does not create any Δ submatrix in $M(G + \{u, v\})$.

To insert an edge $\{u, v\}$ into G, we first check if it is already present in G. If not, we insert it into G, provided no submatrix Δ is created in $M(G)$.

The initialization process consists of computing a strong elimination ordering α of a strongly chordal input graph, G. For this, we use a recognition algorithm for strongly chordal graphs due to Farber [22].

Algorithm 5.4 *Insert*

```
Input: A strongly chordal graph $G$ and an edge $\{u, v\}$ to be inserted
Output: A strongly chordal graph $G + \{u, v\}$
1: if $Insert - Query(G, u, v)$ returns “True” then
2:     Insert edge $\{u, v\}$ into $G$
3:     Update neighborhood matrix $M(G)$
4: end if
```

Having obtained a strong elimination ordering α, we create the neighborhood matrix $M(G)$ of G. We also identify the relative order of u and v in the ordering α. Now we check if the insertion of an edge $\{u, v\}$ creates any Δ submatrix or not.

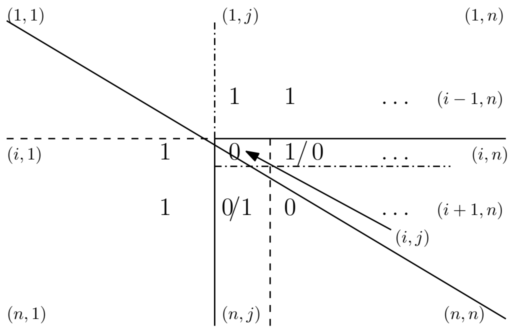


Figure 5.7: *Algorithm to find Δ submatrix*

The searching strategy can be explained with the help of Fig. 5.7. Assume there is a 0 in the i^{th} row and j^{th} column that we want to change into a 1 (which corresponds to the insertion of an edge in the graph). For this, we need to test if, as a result, Δ appears as a submatrix in $M(G)$. We check in three different directions from the (i, j)-th position: upward to $(1, n)$, downward to (n, n), and leftward to $(n, 1)$. If no Δ submatrix is found,

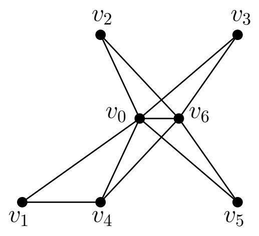
Algorithm 5.5 *Insert-Query*

```
Input: A strongly chordal graph $G$ and an edge $\{u, v\}$ to be inserted
Output: Return True or False
 1: $canBeInserted \leftarrow$ True
 2: if $\{u, v\}$ is an edge of $G$ then
 3:     $canBeInserted \leftarrow$ False
 4:     return $canBeInserted$
 5: else
 6:     for $l \leftarrow j$ to $n$ do                                    ▷ downward
 7:         for $k \leftarrow i$ to $n$ do
 8:             if $(M[i][l+1] == 1$ and $M[k+1][j] == 1$ and $M[k+1][l+1] == 0)$ then
 9:                 $canBeInserted \leftarrow$ False
10:                 return $canBeInserted$
11:             end if
12:         end for
13:     end for
14:     for $l \leftarrow j$ to $n$ do                                    ▷ upward
15:         for $k \leftarrow i$ to 0 do
16:             if $(M[k-1][j] == 1$ and $M[k-1][l+1] == 1$ and $M[i][l+1] == 0)$ then
17:                 $canBeInserted \leftarrow$ False
18:                 return $canBeInserted$
19:             end if
20:         end for
21:     end for
22:     for $l \leftarrow j$ to 0 do                                      ▷ leftward
23:         for $k \leftarrow i$ to $n$ do
24:             if $(M[i][l-1] == 1$ and $M[k+1][l-1] == 1$ and $M[k+1][j] == 0)$ then
25:                 $canBeInserted \leftarrow$ False
26:                 return $canBeInserted$
27:             end if
28:         end for
29:     end for
30:     return $canBeInserted$
31: end if
```

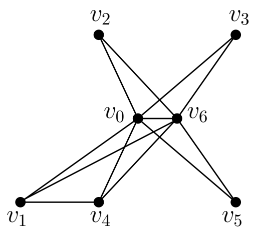
we change the (i, j)-th entry to 1. Since $M(G)$ is symmetric, simultaneously we change the (j, i)-th entry to 1. The formal details of the above search strategy is given in Algorithm 5.5. If it returns "True", Algorithm 5.4 inserts the edge $\{u, v\}$ into G.

5.2.1 An Example

Consider the strongly chordal graph shown in Fig. 5.8(a). After finding a strong elimination ordering $v_1, v_2, v_3, v_4, v_6, v_5, v_0$ of G, we create the neighborhood matrix $M(G)$. Now, say we want to insert an edge $\{v_2, v_5\}$ into G. However, this insertion creates a Δ submatrix in G and this cannot be done. Next, suppose we want to insert the edge $\{v_1, v_6\}$ into G. This is possible as its insertion does not create any Δ submatrix in G. Since $M(G)$ is a symmetric matrix, we change 0 to 1 in both symmetric positions (which corresponds to the insertion of $\{v_1, v_6\}$ in G).



(a) A strongly chordal graph G



(b) $G + \{v_1, v_6\}$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(c) A neighborhood matrix $M(G)$ of the strongly chordal graph shown in Fig. (a)

$$\begin{bmatrix} 1 & 0 & 0 & 1 & \boldsymbol{1} & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ \boldsymbol{1} & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(d) A neighborhood matrix $M(G)$ of the strongly chordal graph shown in Fig. (b)

Figure 5.8: *An example of insertion*

5.2.2 Complexity of Insertions

Computing a strong elimination ordering using Farber's algorithm takes $O(n^3)$ time, and it takes $O(n^2)$ time to initialize the neighborhood matrix $M(G)$. Thus the preprocessing time complexity is $O(n^3)$. The upper bound on searching for a Δ submatrix in $M(G)$ is $O(n^2)$. Thus the time complexity of an insert-query is $O(n^2)$.

The insertion of an edge takes constant time since we maintain a neighborhood matrix data structure to represent G.

With a more significant amount of preprocessing, here is an input-sensitive solution to the search for a Δ-submatrix in $M(G)$.

With each entry of the $M(G)$ matrix we associate four integer values, v_u, v_d, h_l, h_r, that record the runs of that entry vertically up and down, horizontally left and right. This is illustrated in Fig. 5.9.

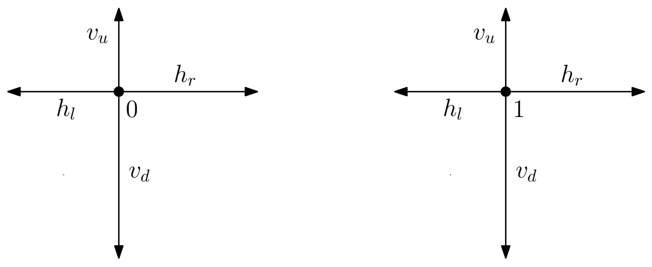


Figure 5.9: *Runs relative to an entry*

In addition, we record for each row and column of $M(G)$, the range of 0's and 1's into which it can be decomposed (see Fig. 5.10).

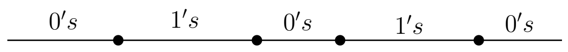


Figure 5.10: *Range decomposition of a row or column*

We explain how the search for a Δ-submatrix is done in the upper right quadrant relative to a 0 as the (i, j) entry of $M(G)$ that we would like to replace by a 1. For this, we intersect a 0-range to the right of this entry on the i-th row and a 1-range vertically above this entry on the j-th column (see Fig. 5.11).

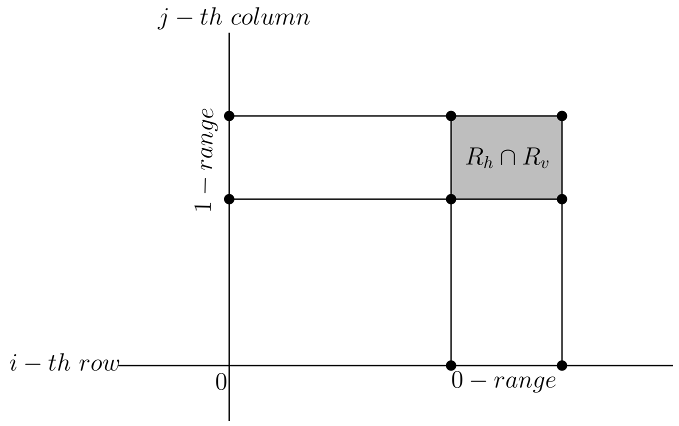


Figure 5.11: *Searching the intersection of ranges*

We want to find a 1 in this intersected range. For this, we traverse the boundary of this range, looking for a 1 or probe its interior relative to a 0 for a 1 (using the recorded information on horizontal and vertical runs). We terminate the traversal as soon as we find a 1 or report that there is no 1 in this range.

The complexity of this search is: $\Sigma(|R_h| + |R_v|)$, where $|R_h|$ and $|R_v|$ are the sizes of the horizontal and vertical ranges and the sum is taken over all pairs, consisting of a 1-range vertically above the (i, j)-th entry and a 0-range to the left of this entry. This shows that the search-complexity is sensitive to the distribution of 0's and 1's in the matrix $M(G)$ and, except for the worst-case scenario when all the 0 and 1 ranges are of size 1, has time complexity that is of lower order than n^2.

A similar search is carried out for a 1 in the lower left quadrant and a 0 in the lower right quadrant. In case all three searches fail, the 0 entry is changed into a 1. We also update the ranges of 0's and 1's on the i-th and j-th columns and update the neighborhood information of the 0's in v_u, v_d, h_l and h_r, adjoining the (i, j)-th entry. All this work takes linear time.

5.3 Discussion

The proposed semi-dynamic algorithms are based on two different characterizations of strongly chordal graphs. The deletion algorithm is based on a strong chord characterization, while the insertion algorithm is based on a totally balanced matrix characterization. An interesting and challenging open problem is to come up with an efficient fully dynamic algorithm for this class of graphs.

5.4 Summary

In this chapter, we have presented semi-dynamic algorithms for deletions and insertions of edges into a strongly chordal graph.

Chapter 6

Chordal Graphs and Point Placement in the Plane

In this chapter, we discuss an application of chordal graphs to the problem of designing a 1-round algorithm for approximate point placement in the plane in an adversarial model. The distance query graph presented to the adversary is chordal. The remaining distances are determined using a distance matrix completion algorithm for chordal graphs, based on a result by Bakonyi and Johnson [6]. The layout of the points is determined from the complete distance matrix using the traditional Young and Householder approach [57].

6.1 Introduction

The problem of locating n distinct points on a line, up to translation and reflection, in an adversarial setting has been extensively studied [1, 15, 16, 17]. The best known 2-round algorithm that makes $9n/7$ queries and has a query lower bound of $9n/8$ queries is due to Alam and Mukhopadhyay [2]. In this chapter, we propose a 1-round algorithm for the same problem in the plane. To the best of our knowledge, there is no prior work extant on this problem. A practical motivation for this study is the extensively researched and closely related sensor network localization problem [5, 10].

6.2 Preliminaries

Let $D = [d_{ij}]$ be an $n \times n$ *symmetric matrix* (a square matrix that is equal to its transpose, e.g., $D = D^T$), whose diagonal entries are 0 and the off-diagonal entries are positive. It is said to be an *Euclidean Distance Matrix* if there exists points $p_1, p_2, \ldots, p_n$ in some k-dimensional Euclidean space such that $d_{ij} = d(p_i, p_j)^2$, where $d(p_i, p_j)$ is the Euclidean distance between the points p_i and p_j. A set of necessary and sufficient conditions for this was given by Schoenberg [50], as well as Young and Householder [57]. A *partial distance matrix* is one in which some entries are missing. If $x \neq 0$ is a $n \times 1$ vector and λ is a scalar such that $Dx = \lambda x$, then x is an *eigenvector* of D with *eigenvalue* λ. D is *positive semi-definite* if $x^T Dx \geq 0$ for all non-zero column vector x in $\mathbb{R}^n$ and x^T denotes the transpose of x. The *rank* of a matrix D corresponds to the maximal number of linearly independent columns of D and is denoted as $rankD$.

The *distance graph* of an $n \times n$ distance matrix, is a graph on n vertices with an edge connecting two vertices v_i and v_j if there is an entry greater than zero in i-th row and j-th column of the distance matrix.

6.3 Point Placement On a Line: A Quick Review

To provide a context and motivation for the results of this chapter, we provide a quick review of the main ideas underlying point placement algorithms for points on a line, with reference to a state-of-the-art algorithm [2].

Let $P = \{p_1, p_2, \ldots, p_n\}$ be n distinct points on a line. A distance graph on n vertices (corresponding to the n points in P) has edges joining pairs of points whose distances on the line are sought of an adversary. An assignment of lengths to the edges of this graph by an adversary is assumed to be valid if there exists a linear layout consistent with these lengths. The distance graph is said to be *line-rigid* if a consistent layout exists for all valid, adversarial assignments of lengths. All the distance graphs shown in Fig. 6.1 are line-rigid. However, a 4-cycle is not line-rigid as there exists an assignment of lengths that makes it a parallelogram, whose vertices have two distinct linear layouts.

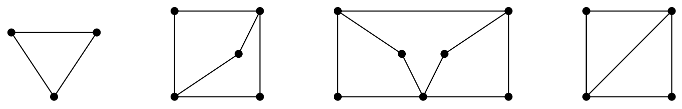

(a) K_3 (b) $K_{2,3}$ (c) Jewel (d) K_4^-

Figure 6.1: *Some examples of line rigid graphs*

We define a k-round ($k \geq 1$) algorithm as one in which the adversarial distance queries are done in k batches in order to resolve the placement of the points.

A prototypical 1-round algorithm [16] uses the line-rigid 3-cycle (or triangle) graph as the core structure and constructs the following distance graph on n points (see Fig. 6.2).

As the figure shows, the graph has $n-2$ triangles hanging from a common strut. The number of distance queries made $2n-3$, where the number of distance queries made for the $((n-1)-2)$ triangles is $2(n-3)$ and the number of distance queries for the remainder triangle is 3.

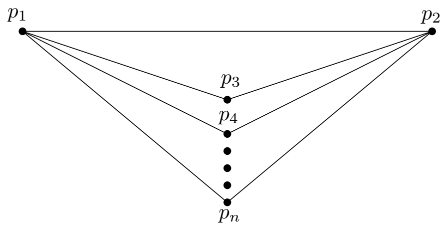


Figure 6.2: *Distance graph for a 1-round algorithm*

A prototypical 2-round algorithm [16] uses the 4-cycle (or triangle) graph as the core structure and constructs the following distance graph on n points (see Fig. 6.3). As the figure shows, the graph has b and $b+2$ edges, hanging from the left and right end-points respectively of a fixed edge. The explanation is that a 4-cycle is not line-rigid and the rigidity condition that at least one pair of opposite edges are not equal can be satisfied over two rounds. The discrepancy of 2 in the number of edges hanging from the two end-points allows us to pair edges which are not equal and thus meet the line rigidity condition for a 4-cycle. The number of distance queries made is $3n/2-2$. Thus by increasing the number of rounds and constructing a more complex query graph, we reduce the number of distance queries by a constant factor. The goal is to minimize the number of adversarial queries we ask of an adversary.

The best known 2-round algorithm to-date [2], builds a distance query graph using the

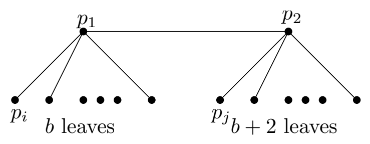


Figure 6.3: *Distance graph for a 2-round algorithm*

3-path graph of Fig. 6.4. Its query complexity is $9n/7 + O(1)$. This comes at the expense of 55 rigidity conditions that must be satisfied over two rounds.

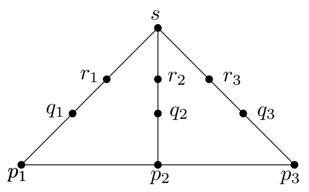


Figure 6.4: *The 3-path graph*

The main tool for obtaining these rigidity conditions is the concept of a layer graph, introduced in [15]. A layer graph is an orthogonal re-drawing (if possible) of the distance query graph that must satisfy the following conditions:

P1. Each edge e of G is parallel to one of the two orthogonal directions $\mathbf{x}$ and $\mathbf{y}$.

P2. The length of an edge e is the distance between the corresponding points on L.

P3. Not all edges are along the same direction (thus, a layer graph has a two-dimensional extent).

P4. When the layer graph is folded onto a line, by a rotation either to the left or to the right about an edge of the layer graph lying on this line, no two vertices coincide.

Chin et al. [15] showed that a given distance query graph is not line rigid if and only if it has a layer graph drawing. The different rigidity conditions are derived from a painstaking enumeration of all possible layer graph drawings of the given distance query graph. This is a challenging task.

Our experience with the implementation of 2-round algorithms (see [40]) has shown that the rigidity conditions are easy to verify when exact arithmetic is used; indeed, we simulated an adversary by generating layouts with integral coordinates. However, if pairwise distances are not integral, the rounding errors introduced in finite-precision calculations can make checking the rigidity conditions difficult. This is an unavoidable issue for point-placement in the plane.

Another difficulty of generalizing this approach to two and higher dimensions is that of obtaining a suitable generalization of the layer graph concept and the associated theorem. This motivates the approach taken in this chapter. The advantage of this approach is that it is susceptible to generalization to higher dimensions.

6.4 Overview of our results

We first discuss a reductionist approach to this problem: reducing point placement in the plane to point placement on a line. We consider the case when the points lie on a circle, using stereographic projection to reduce this to a 1-dimensional point placement problem. For points lying on an integer grid, we reduce the problem to two 1-dimensional point placement problems.

The algorithms for point placement on a line require testing a very large number of constraints involving edge lengths of a distance graph. Our experiments have shown that these work well when the points on a line have integral coordinates. To circumvent this problem, we consider a matrix distance completion approach, when the distance graph is chordal. In our adversarial setting, we seek the lengths of this chordal graph from an adversary (an adversary can be thought of as a source of correct distance measurements).

Once the adversary has returned edge lengths for the chordal distance graph, we solve a matrix distance completion problem. Bakonyi and Johnson [6] showed that if the distance graph corresponding to a partial distance matrix is chordal, there exists a completion of this partial distance matrix. Precisely, they proved the following result.

Theorem 6.1 [6] *Every partial distance matrix in R^k, the graph, G, of whose specified entries is chordal, admits a completion to a distance matrix in R^k.*

Finally, we compute the planar coordinates of the vertices of this complete distance graph using an algorithm based on a result of Young and Householder [57].

6.5 Point placement in the plane

When the points $p_1, p_2, \ldots, p_n$ lie on an integer grid, we can solve the problem by solving two 1-dimensional point placement problems by projecting them on the x and y-axes. We assume that no two points lie on the same vertical or horizontal line of the grid (see Fig. 6.5).

When the points lie on a circle, we can solve the problem by a stereographic projection of the points on a line and then applying a 1-dimensional point location algorithm to the

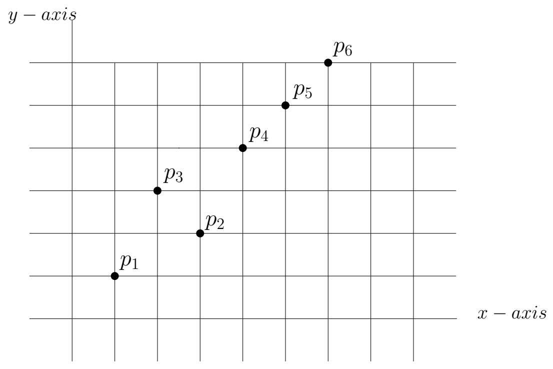


Figure 6.5: *Points on a two-dimensional integer grid*

projected points (see Fig. 6.6).

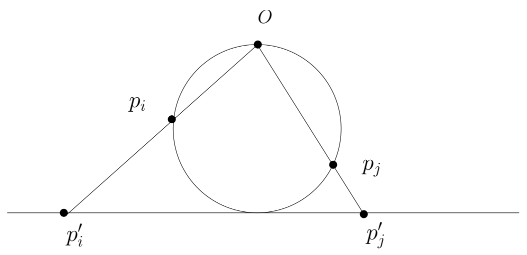


Figure 6.6: *Stereographic projection of points on a circle*

When the distance query graph is complete, we can compute the locations of the points using an algorithm, based on the following result due to Young and Householder [57].

Theorem 6.2 [57] *A necessary and sufficient condition for a set of numbers $d_{ij} = d_{ji}$ to be the mutual distances of a real set of points in Euclidean space is that the matrix $B = [d_{1i}^2 + d_{1j}^2 - d_{ij}^2]$ be positive semi-definite; and in this case the set of points is unique apart from a Euclidean transformation.*

In this case, there exists an orthogonal matrix σ such that

$$B = \sigma L^2 \sigma^t$$

where $L^2 = [\lambda_1^2, \lambda_2^2, \ldots, \lambda_r^2, 0, \ldots, 0]$, λ's are the eigenvalues, σ^t is the transpose of σ, and r is the embedding dimension (which is the minimum dimension, into which the points can be mapped). Thus, we have

$$B = (\sigma L)(\sigma L)^t$$

Since $B = AA^t$, where the rows of the matrix A are the coordinates of the points $p_1, p_2, \ldots, p_n$ in some r dimensional Euclidean space, the coordinates of the points are determined by solving the system of linear equations.

$$A = \sigma L$$

When the (distance) graph of the partial distance matrix is chordal, we use a distance matrix completion algorithm, the major components of which are discussed below.

6.5.1 Computing a perfect elimination ordering, α, of G

A perfect elimination ordering can be found by a breadth-first search of G, combined with lexicographic labeling of its vertices. The Lex BFS algorithm, due to Rose et al. [48] described in chapter 2 (see section 2.1.2). We used the Lex-BFS algorithm to compute a perfect elimination ordering.

6.5.2 Computing a chordal graph sequence

An algorithm for generating the sequence of chordal graphs depends on the following results, proved in [25].

Theorem 6.3 [25] *G has no minimal cycles of length exactly 4 if and only if the following holds: For any pair of vertices u and v with $u \neq v$, $\{u, v\} \notin E$, the graph $G + \{u, v\}$ has a unique maximal clique which contains both u and v. (That is: if K and K' are both cliques in $G + \{u, v\}$ which contain u and v, then so is $K \cup K'$.)*

In particular, Theorem 6.3 holds for chordal graphs. The next theorem suggests an iterative algorithm for solving the distance matrix completion problem.

Theorem 6.4 [25] *Let $G = (V, E)$ be chordal. Then there exists a sequence of chordal graphs $G_i = (V, E_i)$, $i = 0, 1, \ldots, k$, such that $G = G_0, G_1, G_2, \ldots, G_k$ is the complete graph and G_i is obtained by adding to G_{i-1} an edge $\{u, v\}$ as in Theorem 6.3.*

Such an edge $\{u, v\}$ is selected using the following scheme described in [25]. Assume that a perfect elimination ordering α of the vertices of the input chordal graph G is available. Let v_k be the vertex $\alpha^{-1}(k)$. Set $k_i = max\{k \mid \{v_k, v_m\} \notin E_i$ for some $m\}$ and $r_i = max\{r \mid \{v_r, v_{k_i}\} \notin E_i\}$. Then the edge to be added is $\{u, v\} = \{u_{k_i}, v_{r_i}\}$. In the next section we discuss an algorithm for selecting a maximal clique, containing this edge.

6.5.3 Computing a maximal clique containing a given edge

An interesting algorithm due to Bron-Kerbosch [14] computes all maximal cliques, from which we can select the maximal clique that contains this edge. The Bron-Kerbosch algorithm is a recursive backtracking algorithm, and a version based on choosing a pivot is described thus. The algorithm maintains three sets R, P, and X, reporting the set R as the vertices of a maximum clique when at any level of the recursive calls, the sets P and X become empty.

Algorithm 6.1 *ComputingCliques [14]*

```
1: BronKerbosch2($R, P, X$):
2: if ($P$ and $X$ are both empty) then
3:     report $R$ as a maximal clique
4: end if
5: choose a pivot vertex $u \in P \cup X$
6: for each vertex $v \in P \setminus N(u)$ do
7:     BronKerbosch2($R \cup \{v\}, P \cap N(v), X \cap N(v)$)
8:     $P := P \setminus \{v\}$
9:     $X := X \cup \{v\}$
10: end for
```

We have implemented a simple algorithm that starts with the edge of interest and grows this into a maximal clique. In greater details, start with a clique containing two vertices of the given edge, and grow the current clique one vertex at a time by looping through the graph's remaining vertices. For each vertex v examined, add v to the clique if it is adjacent to every vertex that is already in the clique; otherwise, discard v.

6.5.4 Distance matrix completion of a clique

The distance matrix of a clique with the distance of one edge missing can be formulated as the problem of completing a partial distance matrix with one missing entry. The following lemma proposes a solution to this problem.

Theorem 6.5 [6] *The partial distance matrix*

$$\begin{pmatrix} 0 & D_{12} & x \\ D_{12}^t & D_{22} & D_{23} \\ x & D_{23}^t & 0 \end{pmatrix}$$

admits at least one completion to a distance matrix F. Moreover, if

$$\begin{pmatrix} 0 & D_{12} \\ D_{12}^t & D_{22} \end{pmatrix}$$

and

$$\begin{pmatrix} D_{22} & D_{23} \\ D_{23}^t & 0 \end{pmatrix}$$

are distance matrices with embedding dimensions p and q then x can be chosen so that the embedding dimension of F is $s = max\{p, q\}$.

This is equivalent to finding completions of the partial distance matrix:

$$\begin{pmatrix} 0 & 1 & 1 & e^t & 1 \\ 1 & 0 & d_{12} & \overline{D}_{13} & d_{14} \\ 1 & d_{12} & 0 & \overline{D}_{23} & x \\ e & \overline{D}_{13}^t & \overline{D}_{23}^t & \overline{D}_{33} & \overline{D}_{34} \\ 1 & d_{14} & x & \overline{D}_{34}^t & 0 \end{pmatrix}$$

to a matrix in which the Schur complement

$$\begin{pmatrix} a & B & x - d_{12} - d_{14} \\ B^t & C & D \\ x - d_{12} - d_{14} & D^t & f \end{pmatrix}$$

of the upper left 2×2 principal submatrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

has a positive semidefinite completion of rank s. This provides a solution for x that follows from the following result.

Theorem 6.6 [20] *Let*

$$R = \begin{pmatrix} a & B & x \\ B^t & C & D \\ x & D^t & f \end{pmatrix}$$

be a real partial positive semidefinite matrix in which rank $\begin{pmatrix} a & B \\ B^t & C \end{pmatrix} = p$ *and rank* $\begin{pmatrix} C & D \\ D^t & f \end{pmatrix} =$ q*. Then there is real positive semidefinite completion* F *of* R *such that the rank of* F *is* $max\{p, q\}$*. This completion is unique iff* $rankC = p$ *or* $rankC = q$*.*

In the next section, we discuss the last stage of the point placement problem: this is to determine the coordinates of a layout from a completed distance matrix.

6.5.5 Experimental results

We have implemented the above algorithm in Python on a laptop with an Intel Core i7 processor with 16GB RAM running under Windows 10. The software includes a module for the generation of chordal graphs to be used as input. The chordal graph generation uses one of the algorithms discussed in chapter 2.

The result of an experiment is described below for a chordal graph. The following partial distance matrix, where the off-diagonal 0's represent unknown distances,

$$\begin{pmatrix} 0 & 9 & 0 & 0 & 0 & 0 & 5 & 20 \\ 9 & 0 & 2 & 25 & 40 & 34 & 20 & 17 \\ 0 & 2 & 0 & 17 & 0 & 0 & 0 & 13 \\ 0 & 25 & 17 & 0 & 5 & 0 & 0 & 2 \\ 0 & 40 & 0 & 5 & 0 & 2 & 0 & 5 \\ 0 & 34 & 0 & 0 & 2 & 0 & 10 & 5 \\ 5 & 20 & 0 & 0 & 0 & 10 & 0 & 13 \\ 20 & 17 & 13 & 2 & 5 & 5 & 13 & 0 \end{pmatrix}$$

is obtained from the distances returned by the adversary based on the layout of a chordal graph, G, shown in Fig. 6.7.

The matrix distance completion algorithm outputs the following matrix,

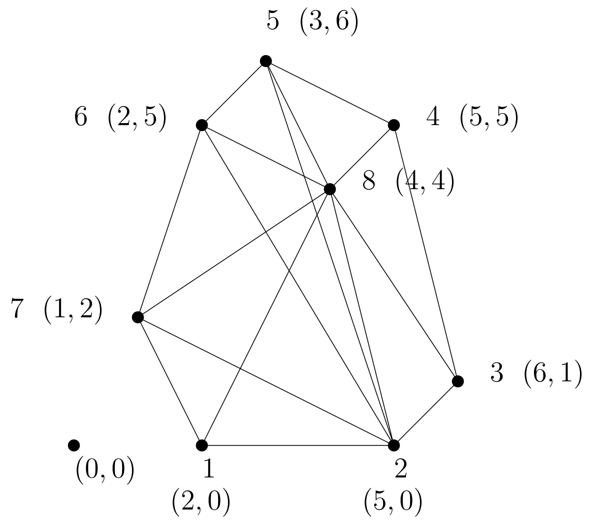


Figure 6.7: *A chordal graph on 8 vertices*

$$\begin{pmatrix} 0 & 9 & 17.0 & 34.0 & 37.0 & 25.0 & 5 & 20 \\ 9 & 0 & 2 & 25 & 40 & 34 & 20 & 17 \\ 17.0 & 2 & 0 & 17 & 34.0 & 32.0 & 26.0 & 13 \\ 34.0 & 25 & 17 & 0 & 5 & 9.0 & 25.0 & 2 \\ 37.0 & 40 & 34.0 & 5 & 0 & 2 & 20.0 & 5 \\ 25.0 & 34 & 32.0 & 9.0 & 2 & 0 & 10 & 5 \\ 5 & 20 & 26.0 & 25.0 & 20.0 & 10 & 0 & 13 \\ 20 & 17 & 13 & 2 & 5 & 5 & 13 & 0 \end{pmatrix}$$

where the computed entries are shown with a decimal point, followed by a single 0. The correctness of the computed entries can be checked against Fig. 6.7.

We ran the above complete distance matrix through our implementation of the Young and Householder algorithm. A plot of the output is shown in Fig. 6.8. As can be seen that, apart from scale and orientation and missing edges, it is the same as the plot of the original graph shown in Fig. 6.7.

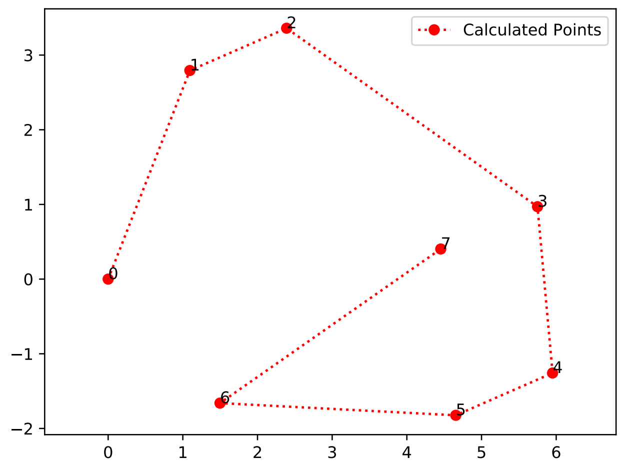


Figure 6.8: *A plot of the output of the Young and Householder algorithm*

6.6 Computational Complexity

The computational complexity of the algorithm can be parametrized with respect to three different measures: (a) number of rounds, which is 1 in our case; (2) query complexity, which is the number of distance queries posed to the adversary and is a function of the number of rounds; (c) the time complexity of the algorithm.

The query complexity is the number of edges in the initial chordal graph. We have tried to make it as sparse as possible. It is always a tree on n vertices, with a few more edges added, to meet the requirements of the distance matrix completion algorithm. Thus the query complexity is $O(n)$. The time complexity of the algorithm is dominated by the number of times we have to perform distance matrix completion of a clique. This is $O(n^2 f(n))$, as we go from a chordal graph, which is nearly a tree to a complete (chordal) graph. This explains the n^2 term. The factor $f(n)$ is the complexity of the distance matrix completion algorithm. A loose upper bound is $O(n^3)$ [6]. Thus the time complexity of our algorithm is in $O(n^5)$.

6.7 Discussion

In this chapter, we have proposed a 1-round algorithm for point placement in the plane in an adversarial setting, taking advantage of an existing infrastructure for completing partial distance matrices whose distance graphs are chordal. The locations of the points in the plane are recovered from the complete distance matrix, using an algorithm based on a result in [57].

Much more work remains to be done. The most interesting open question is this: for what kind of chordal graphs do we have a unique distance matrix completion? Bakonyi and Johnson [6] proved the following:

Theorem 6.7 [6] *Let R be a partial distance matrix in R^k, the graph $G = (V, E)$ of whose specified entries is chordal and let $\mathcal{S}$ be the set of all minimal vertex separators of G. Then R admits a unique completion to a distance matrix if and only if*

$$\begin{pmatrix} 0 & e^T \\ e & R(S) \end{pmatrix}$$

has rank $k + 2$ for any $S \in \mathcal{S}$.

The characterization is interesting but computationally very expensive. It would be interesting to know if simpler characterizations exist that could be used to generate chordal graphs having unique completions.

Other problems of interest are the extensions of the algorithm to other classes of graphs than chordal graphs and the design of 2-round algorithms.

6.8 Summary

At the beginning of this chapter, we briefly discussed the point placement on a line problem. Then we proposed a 1-round algorithm for the same problem in the plane in an adversarial model.

Chapter 7

Conclusions and Open Problems

In this thesis, we have proposed algorithms for generating chordal graphs, weakly chordal graphs, and strongly chordal graphs by exploiting different characterizations of each class of graphs. The methods either take the number of vertices (n) and the number of edges (m) as input or start from an arbitrary graph and then turn the arbitrary graph into a graph in the target class. The generation methods that take n and m as input start with either a tree or a complete graph. When we start with a tree, we insert more edges to reach the target m and when we start with a complete graph, we delete edges to meet the target m. In this case, we maintain the relevant graph properties (chordal, weakly chordal, or strongly chordal) after every insertion or deletion. On the other hand, the relevant graph property is achieved at the end of the process when we generate these graphs, starting from an arbitrary graph. In this case, we stop the process after meeting certain criteria or the number of edges in the resulting graph reaches m. Note that in both types of generation methods, the number of vertices remains the same. We have also proposed semi-dynamic algorithms for chordal graphs and strongly chordal graphs by maintaining simple data structures.

At first, we proposed methods for generating chordal graphs and a semi-dynamic algorithm for edge-deletions. The proposed unified methods for generating chordal graphs maintain a clique tree. The unified methods are suitable for generating both sparse and dense graphs. We modified a result by Dirac [18] to generate k-chromatic chordal graphs with the addition of fewer edges. The method is straightforward and does not require the maintenance of the clique tree. This method turns an arbitrary graph into a chordal graph. The semi-dynamic algorithm for chordal graphs starts with a non-trivial chordal graph and maintains chordality after the deletion of every edge. This algorithm maintains only a simple adjacency matrix data structure. There has been some work on the uniform generation of trees and regular graphs. An interesting open problem would be to generate chordal graphs uniformly at random. Work has also been done on the generation of regular graphs from prescribed degree sequences. To the best of our knowledge, the problem of generating chordal graphs from prescribed degree sequences has not been studied and therefore, merits serious attention.

Second, we have also proposed two different methods for generating weakly chordal graphs. The first method maintains weak chordality after the insertion of every edge. An interesting open problem is to investigate how to generate weakly chordal graphs uniformly at random. The issue of generating weakly chordal graphs from prescribed degree sequences also merits attention. We have also proposed an algorithm that turns an arbitrary graph into a weakly chordal graph. An interesting open problem is to come up with an efficient algorithm that turns an arbitrary graph into a weakly chordal graph. Also, the problem of resolving whether the class of weakly chordal graphs is a completion class needs to be resolved.

Third, we have proposed three different generation methods and also semi-dynamic algorithms for strongly chordal graphs. The generation methods are based on three different characterizations of strongly chordal graphs. The first two methods generate chordal graphs as an intermediate step and then convert these chordal graphs into strongly chordal graphs. To the best of our knowledge, there seems to be no prior work for the problem of dynamic algorithms for strongly chordal graphs. To address this gap, we proposed a semi-dynamic algorithm for edge-deletions and a semi-dynamic algorithm for edge-insertions in strongly chordal graphs. An exciting and challenging open problem is to come up with an efficient fully dynamic algorithm for this class of graphs.

Finally, as an application of chordal graphs, we have proposed a 1-round algorithm for approximate point placement in the plane in an adversarial model where the distance query graph presented to the adversary is chordal. The proposed method determines the remaining distances using a distance matrix completion algorithm. An interesting problem would be to extend this result to other classes of graphs.

Bibliography

[1] Alam, M. S., and Mukhopadhyay, A. More on generalized jewels and the point placement problem. *Journal of Graph Algorithms and Applications 18*, 1 (2014), 133–173.

[2] Alam, M. S., and Mukhopadhyay, A. Three paths to point placement. In *Algorithms and Discrete Applied Mathematics* (Cham, 2015), S. Ganguly and R. Krishnamurti, Eds., Springer International Publishing, pp. 33–44.

[3] Andreou, M. I., Papadopoulou, V. G., Spirakis, P. G., Theodorides, B., and Xeros, A. Generating and radiocoloring families of perfect graphs. In *Experimental and Efficient Algorithms* (Berlin, Heidelberg, 2005), S. E. Nikoletseas, Ed., Springer Berlin Heidelberg, pp. 302–314.

[4] Arikati, S. R., and Rangan, C. P. An efficient algorithm for finding a two-pair, and its applications. *Discrete Applied Mathematics 31*, 1 (1991), 71–74.

[5] Aspnes, J., Eren, T., Goldenberg, D. K., Morse, A. S., Whiteley, W., Yang, Y. R., Anderson, B. D. O., and Belhumeur, P. N. A theory of network

localization. *IEEE Transactions on Mobile Computing 5*, 12 (2006), 1663–1678.

[6] Bakonyi, M., and Johnson, C. R. The Euclidian distance matrix completion problem. *SIAM Journal on Matrix Analysis and Applications 16*, 2 (1995), 646–654.

[7] Beeri, C., Fagin, R., Maier, D., and Yannakakis, M. On the desirability of acyclic database schemes. *J. ACM 30*, 3 (July 1983), 479–513.

[8] Berry, A. 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.* (1999), pp. 860–861.

[9] Berry, A., Bordat, J. P., and Heggernes, P. Recognizing weakly triangulated graphs by edge separability. *Nord. J. Comput. 7*, 3 (2000), 164–177.

[10] Biswas, P., Lian, T.-C., Wang, T.-C., and Ye, Y. Semidefinite programming based algorithms for sensor network localization. *ACM Trans. Sen. Netw. 2*, 2 (May 2006), 188–220.

[11] Blair, J. R. S., and Peyton, B. An introduction to chordal graphs and clique trees. In *Graph Theory and Sparse Matrix Computation* (New York, NY, 1993), A. George, J. R. Gilbert, and J. W. H. Liu, Eds., Springer New York, pp. 1–29.

[12] Bodlaender, H. L. A tourist guide through treewidth. *Acta Cybern. 11* (1993), 1–21.

[13] Boppana, R. B., and Halldórsson, M. M. Approximating maximum independent sets by excluding subgraphs. *BIT 32*, 2 (1992), 180–196.

[14] Bron, C., and Kerbosch, J. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM 16*, 9 (Sept. 1973), 575–577.

[15] Chin, F. Y. L., Leung, H. C. M., Sung, W.-K., and Yiu, S.-M. The point placement problem on a line - improved bounds for pairwise distance queries. In *Proceedings of the Workshop on Algorithms in Bioinformatics* (2007), vol. 4645 of *LNCS*, pp. 372–382.

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

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

[18] Dirac, G. A. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg 25*, 1 (Apr 1961), 71–76.

[19] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. *Biological Sequence Analysis (Probabilistic models of proteins and nucleic acids).* Cambridge University Press, The Pitt Building. Trumpington Street, Cambridge, United Kingdom, 2002.

[20] Ellis, R., and Lay, D. Rank-preserving extensions of band matrices. *Linear Multilinear Algebra 26* (1990), 147–179.

[21] Farber, M. *Applications of 1.p. duality to problems involving independence and domination.* PhD thesis, Rutgers University, 1982.

[22] Farber, M. Characterizations of strongly chordal graphs. *Discrete Mathematics 43*, 2-3 (1983), 173–189.

[23] Fulkerson, D. R., and Gross, O. A. Incidence matrices and interval graphs. *Pacific J. Math. 15*, 3 (1965), 835–855.

[24] Gavril, F. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B 16*, 1 (1974), 47 – 56.

[25] Grone, R., Johnson, C. R., Sa, E. M., and Wolkowicz, H. Positive definite completions of partial hermitian matrices. *Linear Algebra and its Applications 58* (1984), 109–124.

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

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

[28] Hayward, R. B., Hoàng, C. T., and Maffray, F. Optimizing weakly triangulated graphs. *Graphs and Combinatorics 5*, 1 (1989), 339–349.

[29] Hayward, R. B., Spinrad, J. P., and Sritharan, R. Improved algorithms for weakly chordal graphs. *ACM Trans. Algorithms 3*, 2 (2007), 14.

[30] Ibarra, L. Fully dynamic algorithms for chordal graphs and split graphs. *ACM Trans. Algorithms 4*, 4 (2008), 40:1–40:20.

[31] Jerrum, M., and Sinclair, A. Fast uniform generation of regular graphs. *Theor. Comput. Sci. 73*, 1 (1990), 91–100.

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

[33] Lekkeikerker, C., B. J. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae 51*, 1 (1962), 45–64.

[34] Liao, C., and Chang, G. J. k-tuple domination in graphs. *Inf. Process. Lett. 87*, 1 (2003), 45–50.

[35] Lovasz, L. On the shannon capacity of a graph. *IEEE Transactions on Information Theory 25*, 1 (1979), 1–7.

[36] Markenzon, L., Vernet, O., and Araujo, L. H. Two methods for the generation of chordal graphs. *Annals OR 157*, 1 (2008), 47–60.

[37] McKay, B. D., and Wormald, N. C. Uniform generation of random regular graphs of moderate degree. *J. Algorithms 11*, 1 (1990), 52–67.

[38] Mezzini, M. Fully dynamic algorithm for chordal graphs with O(1) query-time and o(n^2) update-time. *Theor. Comput. Sci. 445* (2012), 82–92.

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

[40] Mukhopadhyay, A., Sarker, P. K., and Kannan, K. K. V. Point placement algorithms: An experimental study. *International Journal of Experimental Algorithms 6*, 1 (2016), 1–13.

[41] ODOM, R. M. Edge completion sequences for classes of chordal graphs. Master's thesis, Naval Postgraduate School, 1995.

[42] Prüfer, H. Neuer beweis eines satzes über permutationen. *Arch. Math. Phys. 27* (1918), 742–744.

[43] Przytycka, T. M. Chordal graphs in computational biology – new insights and applications. In *Computational Science – ICCS 2006* (Berlin, Heidelberg, 2006), V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., Springer Berlin Heidelberg, pp. 620–621.

[44] Rahman, M. Z., Krishnan, U. N., Jeane, C., Mukhopadhyay, A., and Aneja, Y. P. A distance matrix completion approach to 1-round algorithms for point placement in the plane. *Trans. Computational Science 33* (2018), 97–114.

[45] Rahman, M. Z., Mukhopadhyay, A., and Aneja, Y. P. A separator-based method for generating weakly chordal graphs. *Accepted in Discrete Mathematics, Algorithms and Applications*.

[46] Rahman, M. Z., Mukhopadhyay, A., Aneja, Y. P., and Jean, C. A distance matrix completion approach to 1-round algorithms for point placement in the plane. In *Computational Science and Its Applications - ICCSA 2017 - 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part II* (2017), pp. 494–508.

[47] Rodionov, A. S., and Choo, H. On generating random network structures: Trees. In *Computational Science - ICCS 2003, International Conference, Melbourne, Australia and St. Petersburg, Russia, June 2-4, 2003. Proceedings, Part II* (2003), pp. 879–887.

[48] Rose, D. J., Tarjan, R. E., and Lueker, G. S. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing 5*, 2 (1976), 266–283.

[49] Saxe, J. Two papers on graph embedding problems. Tech. Rep. CMU-CS-80-102, Dept of Comp. Science, Carnegie Mellon University, 1980.

[50] Schoenberg, I. J. Remarks to murice fretchet's article "sur la definition axiomatique d'une classe d'espace distancis vectoriellement applicable sur l'espace de hilbert". *Annals of Mathematics 36*, 3 (1935), 724–731.

[51] Seker, O., Heggernes, P., Ekim, T., and Taskin, Z. C. Generation of random chordal graphs using subtrees of a tree. *CoRR abs/1810.13326* (2018).

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

[53] Tarjan, R. E., and Yannakakis, M. Addendum: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput. 14*, 1 (1985), 254–255.

[54] Tinhofer, G. Generating graphs uniformly at random. *Computational Graph Theory* (1990), 235–255.

[55] Wormald, N. C. Generating random regular graphs. *J. Algorithms 5*, 2 (1984), 247–280.

[56] Wormald, N. C. Generating random unlabelled graphs. *SIAM J. Comput. 16*, 4 (1987), 717–727.

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

Vita Auctoris

NAME:	Md Zamilur Rahman
PLACE OF BIRTH:	Kushtia, Bangladesh
YEAR OF BIRTH:	1984
EDUCATION:	University of Windsor, Ph.D. in Computer Science, Windsor, ON, Canada, 2020
	University of Lethbridge, M.Sc. (co-op) in Computer Science, Lethbridge, AB, Canada, 2015
	Jahangirnagar University, M.S. in Computer Science and Engineering, Dhaka, Bangladesh, 2007
	Jahangirnagar University, B.Sc. (Hons.) in Computer Science and Engineering, Dhaka, Bangladesh, 2005