A DECOMPOSITION APPROACH TO SOLVE THE SELECTIVE GRAPH COLORING PROBLEM

by

Oylum Şeker

B.S., Industrial Engineering, Boğaziçi University, 2009

M.S., Industrial Engineering, Boğaziçi University, 2012

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Industrial Engineering

Boğaziçi University

2018

ACKNOWLEDGEMENTS

The very first two people to whom I would like to express my profound gratitude are my supervisors Tınaz Ekim Aşıcı and Z. Caner Taşkın. I am deeply grateful to Tınaz Ekim Aşıcı for consistently and proactively creating new opportunities for me, for her willingness to share her vast knowledge, for being an inspiringly enthusiastic, productive and wonderful academic, and for her literally never-ending support and truly valuable guidance. I am immensely thankful to Z. Caner Taşkın for always pointing out practical solutions that prevent me from getting lost, for being ready to help at all times, for explaining any topic clearly and simply, for being an inspirational and excellent academic, and for his patient and precious guidance and support.

I am grateful to Pınar Heggernes, who provided her warm guidance in every aspect, kindly welcomed me in Bergen, and taught me many things in little time. I also appreciate her valuable contribution as a member of my dissertation committee.

I would like to thank Kuban Altınel, Didem Gözüpek, and Emre Alper Yıldırım for taking part in my dissertation committee, and for their constructive comments and helpful suggestions. I also would like to extend my thankfulness to Taner Bilgiç, former committee member, for his uniquely valuable remarks and suggestions.

I reserve a very special gratitude to Yaman Barlas, who has always cordially welcome my endless questions in any topic and spared his precious time for wisely answering them. I salute him.

A bundle of cozy thanks goes to SESDYN members İpek Dursun, Berk Görgülü, Nefel Tellioğlu, and Gizem Aktaş. Thanks to İpek for being proactively helpful and for her kind-heartedness, to Nefel for her warm, balanced and considerate friendliness, and to Berk for his cheerful friendship and kind efforts in helping me on many issues otherwise unsolvable. A big thanks to my dearest friend Gizem for her sincerity and

frankness, for the particularly valuable end-of-day chats that I will certainly long for, and for being there whenever I needed.

My heartfelt thanks go to Pınar Dursun for her warmth, openness, and for the extremely pleasant daily and scientific chats that we always failed to end timely. Very special thanks to Cemil Dibek for his true and sincere companionship, for the unlimitedly delightful conversations we had in limited times, and for his highly generous hospitality.

Finally, I greatly acknowledge the support of Boğaziçi University Research Fund (grant number 11765), which made it easier to undertake this research and gave me the opportunity to attend scientific conferences.

Cheers!

ABSTRACT

A DECOMPOSITION APPROACH TO SOLVE THE SELECTIVE GRAPH COLORING PROBLEM

Graph coloring is the problem of assigning a minimum number of colors to all vertices of a graph such that no two vertices that are linked by an edge receive the same color. The selective graph coloring problem is a generalization of the standard graph coloring problem; given a graph with a partition of its vertex set into clusters, the aim is to pick exactly one vertex per cluster so that, among all possible selections, the number of colors needed to color the vertices in the selection is minimum. In this study, we focus on a decomposition based exact solution framework for selective coloring, and apply it first to some special graph families, and then to general graphs with no particular structure. The special classes of graphs that we consider are perfect graphs and some special subclasses of perfect graphs, which are permutation, generalized split, and chordal graphs. In order to test the performance of our solution approach, we need graph instances from these graph classes, which led us to concentrate on the generation of random graphs from the graph classes under consideration in the second part of this study. We then test the decomposition method on graphs with different sizes and densities that we have produced with our generation methods, present computational results and compare them with an integer programming formulation of the problem solved by CPLEX, and a state-of-the-art algorithm from the literature. Our computational experiments indicate that our decomposition approach significantly improves the solution performance, especially in low-density graphs in permutation and generalized split graphs, and regardless of the edge-density in the class of chordal graphs. For perfect graphs in their general form, our approach outperforms both of the other two methods. In the case of general graphs, however, further improvements are needed to make our method competitive with the alternative methods we compare with.

ÖZET

SEÇMELİ ÇİZGE BOYAMA PROBLEMİ İÇİN BİR AYRIŞTIRMA YAKLAŞIMI

Çizge boyama problemi, bir çizgedeki tüm köşeleri birbirine komşu iki köşe aynı rengi almayacak şekilde boyama problemidir. Seçmeli çizge boyama problemi ise bilinen çizge boyama probleminin genelleştirilmiş bir hâlidir; bir çizge ve onun köşelerinin oluşturduğu kümenin bir bölmelemesi verilmişken, her bölmeden yalnızca bir tane köşeyi, sonuçta gerekecek renk sayısı en az olacak şekilde seçme problemidir. Bu çalışma, seçmeli çizge boyama problemi için ayrıştırmaya dayalı bir çözüm yöntemini, önce birtakım özel çizge sınıflarına ve daha sonra da belirli bir yapıya sahip olmayan genel çizgelere uygulamayı amaçlamaktadır. Çalışmanın ele aldığı özel çizge sınıfları, kusursuz çizge sınıfı ve onun devşirim çizgeleri, genelleştirilmiş iki parçalı çizgeler ve kirişli çizgelerden oluşan özel altsınıflarıdır. Kullanılan ayrıştırma yönteminin başarımını sınayabilmek için bu çizge sınıflarından somut örneklere duyulan gereksinim, çalışmanın ikinci kısmının ele alınan çizge sınıflarından rastgele örnek üretmek için yöntemler geliştirmeye odaklanmasını sağlamıştır. Ardından, ayrıştırma yöntemi, çalışmanın ikinci kısmında üretilen farklı boyut ve yoğunluktaki çizge örnekleri üzerinde sınanmış ve sonuçlar, bir tamsayı programlama modelinin CPLEX ile çözdürülmesinden ve literatürde bilinen en iyi yöntemden elde edilen sonuçlarla karşılaştırılmıştır. Yapılan deneylerin sonucu göstermiştir ki ayrıştırma yöntemimiz, çözüm bulmadaki başarımı, devşirim ve genelleştirilmiş iki parçalı çizge sınıflarında özellikle düşük yoğunluklu çizgelerde, kirişli çizgelerde ise yoğunluktan bağımsız olarak önemli ölçüde geliştirmiştir. Kusursuz çizgelerin genel biçiminde yapılan deneylerde ise ayrıştırma yönteminin, karşılaştırma yapılan diğer iki yönteme de genel olarak üstün sonuçlar sağladığı görülmüştür. Öte yandan, genel çizgelerde, ayrıştırma yönteminin karşılaştırılan yöntemlerle rekabet edebilir hale gelmesi için daha fazla geliştirilmesi gerektiği görülmüştür.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$E(G)$	Edge set of graph G
$G[V']$	The subgraph of G induced by V'
$\bar{G}$	Complement of graph G
$N(v)$	Neighborhood of vertex v
$V(G)$	Vertex set of graph G
$\mathcal{V}$	Partition of the vertex set V of graph G
$\alpha(G)$	Stability number of graph G
$\chi(G)$	Chromatic number of graph G
$\chi_{SEL}(G)$	Selective chromatic number of graph G
$\omega(G)$	Clique number of graph G
$\vartheta(G)$	Theta function of graph G

LIST OF ACRONYMS/ABBREVIATIONS

BFS	Breadth First Search
IP	Integer Programming
LP	Linear Programming
PEO	Perfect Elimination Ordering
SDP	Semidefinite Programming
Sel-Col	The Selective Graph Coloring Problem

1. INTRODUCTION

Graph coloring is the problem of assigning a minimum number of colors to all vertices of a graph such that no two adjacent vertices, i.e., vertices that are linked by an edge, receive the same color. The problem arises in a variety of areas including scheduling [3], register allocation used in compiler optimization [4], sudoku puzzles [5], and many more. In the most general terms, a problem relevant to graph coloring is comprised of entities, and incompatibilities among them. Entities are represented by vertices, and incompatibilities between pairs of entities by edges, so that no two entities posing incompatibility can result in a conflict when colored.

To embody the formal definition of the problem, a scheduling problem in its simplest form can be considered. Assume that a number of jobs, some of which share common resources, are to be assigned to certain time slots. Denoting jobs with vertices, and pairs of jobs that share a common resource with edges, the problem can be expressed in the domain of graphs. When the vertices of such a graph are colored, each color corresponds to a distinct time slot and hence the minimum number of colors needed to color the vertices of such a graph gives the least number of slots necessary to finish all the jobs [3].

The selective graph coloring problem, Sel-Col, is a generalization of the classical graph coloring problem. Given a graph and a partition of its vertex set into clusters, Sel-Col aims to choose one vertex per cluster so that, among all possible selections, the number of colors necessary to color the vertices in the selection is minimum (see Figure 1.1). In a graph where each cluster consists of a single vertex, Sel-Col becomes equivalent to the classical graph coloring problem [6].

Sel-Col, or *partition coloring* as it is alternatively called in the literature, is motivated by the wavelength routing and assignment problem, and was introduced in [7]. There, a telecommunication network, which is a collection of terminal nodes

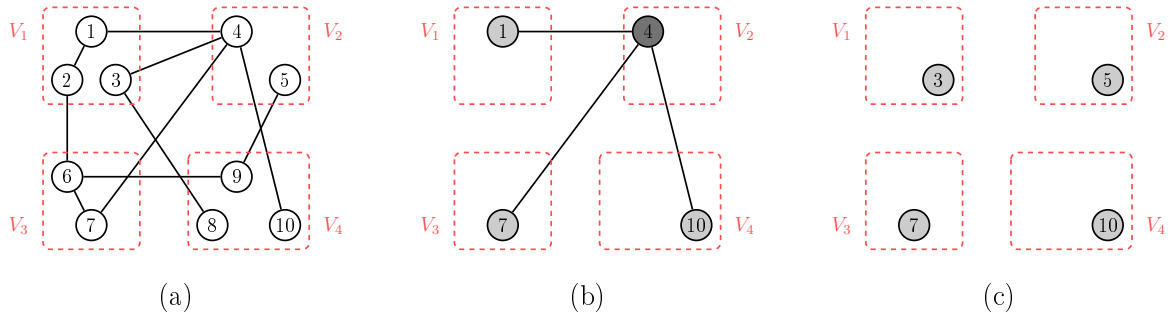


Figure 1.1. (a) A graph G with a partition of its vertex set into four clusters $V_1, \ldots, V_4$ shown in dashed rectangles, (b) a feasible selection in G with an optimal coloring of it, (c) an optimal selection in G with an optimal coloring of it.

linked by optical fibers capable of carrying a certain number of wavelengths, and a set of source-destination node pairs are given. The problem is to find the minimum number of distinct wavelengths to assign to each route, i.e., paths connecting the given source-destination pairs, such that no two routes that have a common link are assigned the same wavelength. In the graph model, the set of all possible routes and pairs of routes possessing a common link correspond to vertices and edges respectively, and groups of routes connecting a given source-destination pair constitute the clusters. Then, selection of one vertex per cluster achieves the goal of finding a route between each pair of terminals, and coloring of the selection delivers a proper wavelength to each route.

A typical example provided in [1] for which selective framework is suitable is timetabling problems, where each job has its own set of available time slots to be scheduled to. In this context, the authors mention the task of scheduling talks at a conference, where each speaker specifies her available time periods to give her talk. There, the organizer needs to assign rooms to speakers given the constraint that two talks with overlapping time intervals cannot be held in the same room. The aim of the organizer in this case is to select a time period for each speaker from her own set of available slots, so that, among all possible choices, the scheduling of the talks would require a minimum number of rooms. In this example, the set of available time slots

for a speaker would correspond to a cluster in the host graph, and two vertices would be adjacent if the associated time slots intersect. Then, selecting one vertex per cluster will set the time slots of the talks for each speaker, and each color used in the selection will correspond to a distinct room.

In the standard graph coloring problem, assignments (colors) on the entities (vertices) are done without any regard to alternative choices for them. However, as the example applications reveal, there are cases where entities have their own set of feasible options (clusters) and thus should be allocated one among those alternatives. In such cases, where the basic graph coloring framework fails to suffice, Sel-Col bridges the gap by offering the required flexibility.

The classical graph coloring problem is known to be NP-hard [8]. The fact that Sel-Col is a generalization of the graph coloring problem makes it an NP-hard problem, too. The difficulty of Sel-Col is twofold, as the authors in [1] point out. It may be caused by the existence of an exponential number of possible selections and/or by the hardness of optimally coloring the graph induced by the selection, even though a selection yielding an optimal solution can be obtained trivially (for instance because there is only one selection as in the case of classical graph coloring problem).

Sel-Col has a wide range of applications including wavelength assignment [7], frequency assignment, various types of scheduling problems, and more [1]. In many real life problems, the application domain yields host graphs that admit characteristics of certain graph families. Problems that are modeled as Sel-Col in specific graph classes include the multiple stacks travelling salesman problem in permutation graphs, several scheduling and timetabling applications in interval graphs, the antenna positioning and the frequency assignment problems in disk and unit disk graphs, etc. [1].

In this dissertation, we present a decomposition-based exact solution approach for Sel-Col, which was originally introduced in [9], and apply it first to some special graph families, and then to graphs with no particular structure. The special graph classes

that we consider in this study are the class of perfect graphs and some subclasses of it; namely, permutation, generalized split, and chordal graphs [10, 11]. Even when we confine ourselves to specific families of graphs, the hardness of the problem may well persist, as it does in our case. It is well known that although the classical graph coloring problem is NP-hard even in several restricted cases [12], it can be solved in polynomial time when restricted to perfect graphs and efficient combinatorial algorithms exist for many of its subclasses [13]. Nevertheless, SEL-COL being at least as difficult as the classical coloring problem from a computational point of view, remains NP-hard in many special cases. In particular, SEL-COL is proven to be NP-hard in permutation graphs, in split graphs, and in interval graphs [1]. The fact that chordal graphs and generalized split graphs contain the class of split graphs makes the problem NP-hard for these two classes, and since the problem is NP-hard in many subclasses of perfect graphs, it remains so in the general class of perfect graphs, too.

To test the performance of our solution approach, we need graph instances from the graph classes under consideration, which, to the best of our knowledge, are not readily available in the literature. Although there exist studies that propose methods to generate instances from some of these classes, they usually did not allow us to control the size and the density of the output graph, or were generally not able to produce varied outputs. Also, with an increasing number of studies that propose algorithms custom-tailored for specific graph classes, which are proven to be efficient in theory, it has become more important to provide a medium for observing how the performance of such algorithms manifests in practice (for a few of such studies, see [14–18]). As a natural consequence, the second part of this study is concentrated on the generation of random graphs from the special graph classes under consideration; namely, permutation, generalized split, chordal, and perfect graphs [10, 11, 19, 20].

The decomposition approach we utilize in this study separates the task of selection and the coloring of the selected vertices, which naturally inclines us to be interested in graph classes where the classical coloring problem can be solved efficiently. The class of perfect graphs is one such class where the graph coloring problem is polynomially solv-

able, which has a practical significance toward facilitating our solution procedure. In the subclasses of perfect graphs under consideration, there exist algorithms to solve the coloring problem that are not only polynomial-time but also of combinatorial nature, which is one main reason for us to treat SEL-COL separately in those graph families, where we combine integer programming techniques and combinatorial algorithms in our decomposition procedure. Thus, in addition to the significance that some of the aforementioned subclasses of perfect graphs have in real-world applications, perfect graphs in general have the potential to be more tractable from the computational aspect, too, as will be confirmed by our experimental results later.

The rest of the presentation of this dissertation is organized as follows. In the next two chapters, we provide some graph-theoretical definitions in order to prepare the ground for the forthcoming chapters, and a synopsis of relevant studies on SEL-COL from the literature. We then explain the solution approaches in Chapter 4 and 5, and continue with the methods we designed for random graph generation in Chapter 6.In Chapter 7, we present an extensive set of computational results for the decomposition approach in comparison to those of an integer programming formulation and a state-of-the-art algorithm from the literature. Finally, Chapter 8 concludes with a brief summary and points at possible future research directions.

2. DEFINITIONS

A *graph* is an ordered pair $G = (V, E)$ with V being the set of *vertices* (or *nodes*) and E being the set of *edges*, which are pairs of elements of V. It provides a representation of a set of objects and their interrelations, where objects refer to vertices, and the predefined relations between pairs of objects correspond to edges in the graph.

An *undirected graph* is a graph where elements comprising $E(G)$ are defined as unordered pairs. In other words, edges in an undirected graph have no orientation. A *simple graph* is an undirected graph in which an edge with both endpoints on the same vertex and multiple edges with endpoints on the same vertex pair are not allowed. In this thesis, whenever the term "graph" is used, simple graphs will be meant (see Figure 2.1).

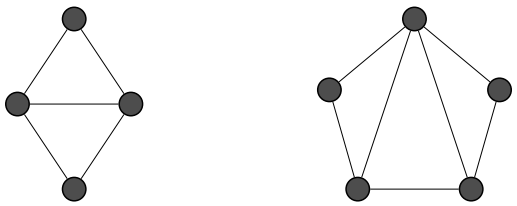


Figure 2.1. Two graphs $G_1 = (V_1, E_1)$ with $|V_1| = 4$ and $|E_1| = 5$, and $G_2 = (V_2, E_2)$ with $|V_2| = 5$ and $|E_2| = 7$.

Two vertices in a graph are called *adjacent* if they are connected by an edge. A vertex u is called a *neighbor* of another vertex v if there exists an edge $\{u, v\}$. The *neighborhood* of a vertex v is the set of all vertices that are adjacent to it, and is denoted by $N(v)$. The *degree* of a vertex, $d(v)$, is defined as the number of edges incident to it (see Figure 2.2).s

The *complement* of a graph $G = (V, E)$, denoted as $\bar{G}$, is a graph on the same vertex set V and such that two distinct vertices of $\bar{G}$ are adjacent if and only if they are

Figure 2.2. Two graphs on which neighborhoods $N(v)$ and $N(w)$ of vertices v and w are shown in dashed ellipses, where $d(v) = 3$ and $d(w) = 2$.

not adjacent in G. An *induced subgraph* is a graph formed by a subset $V^{'}$ of $V(G)$ and all edges connecting the pairs of vertices in $V^{'}$. For a graph $G = (V, E)$ and $V' \subseteq V$, the subgraph induced by V' will be shown as $G[V']$ (see Figure 2.3).

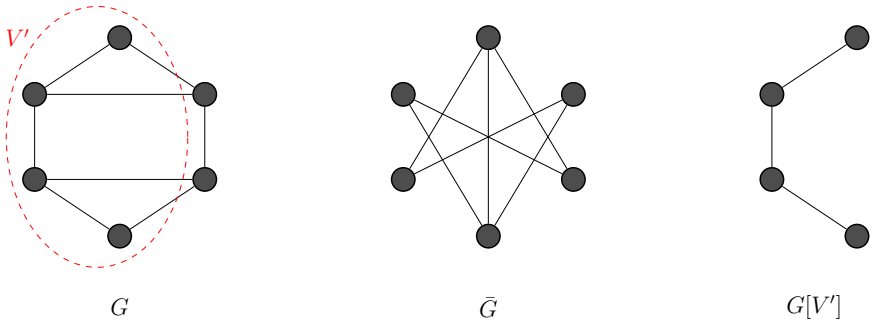


Figure 2.3. A graph G, its complement $\bar{G}$, and the graph $G[V']$ induced by V' (from left to right).

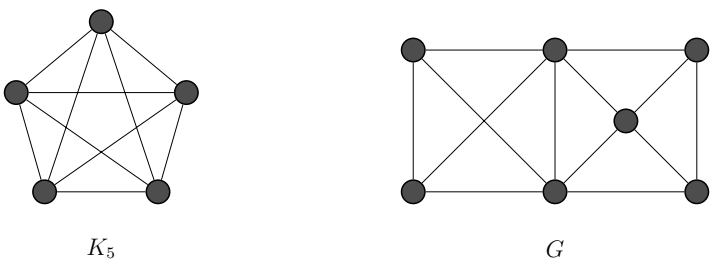


Figure 2.4. A clique on five vertices (left), and a graph G with $\omega(G) = 4$ (right).

A *clique* in a graph is a subset of vertices such that every distinct pair of vertices in the subset is adjacent. In a graph, a given clique is *maximal* if its size cannot be extended with inclusion of some other vertex; in other words, if it is not a proper subset

of another clique. The *clique number* of a graph G is the size of a largest clique in G and is denoted by $\omega(G)$. If a graph on n vertices contains all possible edges among its vertices, i.e., the graph itself is a clique of size n, then it is called a *complete graph* and shown by K_n (see Figure 2.4).

A *stable set*, or equivalently an *independent set*, is a set of vertices in a given graph in which no two vertices are adjacent. The stability number of a graph G, shown by $\alpha(G)$, is the size of the largest stable set.

A (simple) *cycle* is comprised of a sequence of consecutively adjacent vertices that starts and ends at the same vertex with no repetitions of vertices and edges. A *chord* is an edge linking two non-consecutive vertices in a cycle. A *tree* is a graph in which any two vertices are connected by exactly one path, where a *path* is a sequence of edges that connects a series of vertices in a graph (see Figure 2.5).

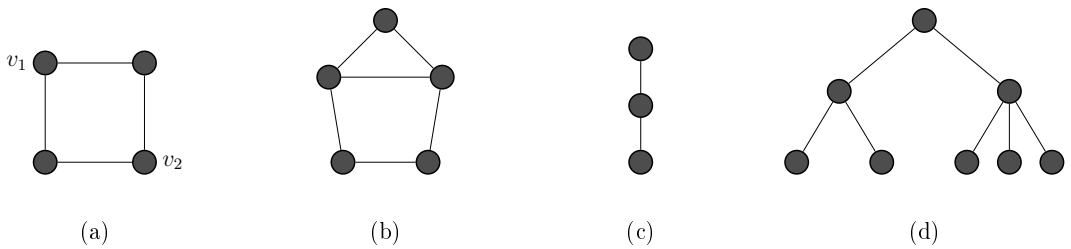
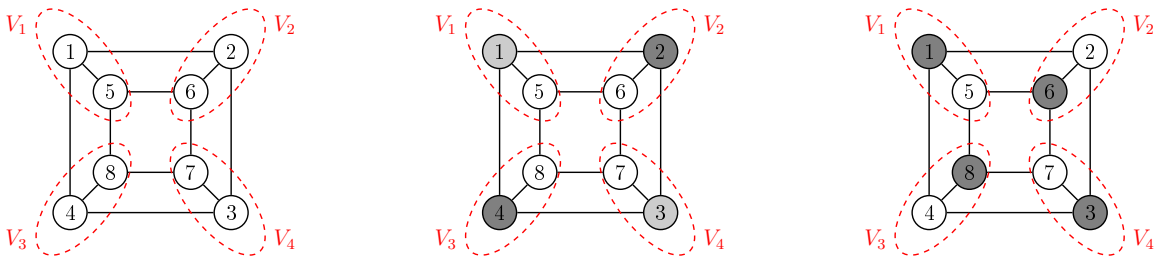


Figure 2.5. (a) A cycle with four vertices where v_1 and v_2 form a stable set, (b) a cycle with five vertices with a chord on it, (c) a path on three vertices, (d) a tree having eight vertices.

A coloring of a graph using at most k colors is called a (proper) *k-coloring*. A graph is called *k-colorable* if its vertices can be assigned a k-coloring. The *chromatic number* of a graph G, denoted by $\chi(G)$, is the minimum number of colors necessary to color all vertices of the graph. Note that a graph G is k-colorable for all $k \geq \chi(G)$ (see Figure 2.6).

Figure 2.6. A proper 3-coloring of of a graph G (left), and an optimal coloring of it with $\chi(G) = 2$ (right).

Given a graph $G = (V, E)$ with a partition $\mathcal{V} = \{V_1, ..., V_P\}$ of its vertex set into P clusters, a *selection* is a subset of vertices of G that contains exactly one vertex from each cluster in the partition; i.e., $V' \subseteq V$ such that $|V' \cap V_p| = 1$ for all $p \in \{1, \ldots, P\}$. A *selective k-coloring* of G is defined by a selection V' and a k-coloring of $G[V']$. The smallest integer k for which G admits a selective k-coloring is called the *selective chromatic number* of G and is denoted by $\chi_{SEL}(G, \mathcal{V})$ [1]. A *selective clique* of G is a clique in the graph induced by a selection. A maximal clique in the graph induced by a selection is called a *maximal selective clique*, and a maximal selective clique of maximum size is called a *maximum selective clique* of G (see Figure 2.7).



(a) (b) (c)

Figure 2.7. (a) A graph G with a partition of its vertex set into four clusters $\mathcal{V} = \{V_1, \ldots, V_4\}$ shown in dashed ellipses, (b) an optimally colored feasible selection $\{1, 2, 3, 4\}$ in G, which contains a maximum selective clique $\{1, 2\}$, (c) an optimal selection $\{1, 6, 3, 8\}$ in G with an optimal coloring of it, yielding $\chi_{SEL}(G, \mathcal{V}) = 1$.

A *permutation graph* is a graph whose vertices represent the elements of a permutation of numbers and edges the pairs of elements that are reversed by the permutation (see Figure 6.1).

A graph G is a *k-partite graph* if its vertex set can be partitioned into k different stable sets. In a *complete k-partite graph*, all possible edges between each pair of k stable sets are existent. A graph $G = (V, E)$ is a *generalized split graph* if there exists a partition $V = V_1 \cup V_2$ of the vertex set such that either $G[V_1]$ is a union of pairwise disjoint cliques and V_2 induces a clique in G, or $\bar{G}[V_1]$ is a union of pairwise disjoint cliques (in other words $G[V_1]$ is a complete multi-partite graph) and V_2 is a stable set in G. In the first case, the graph is called *unipolar*, and in the latter case it is called *co-unipolar* (see Figure 6.3). If the graph induced by V_1 is a clique and V_2 is a stable set, then G is just a *split graph*.

A graph is called *chordal* (or *triangulated*) if every cycle of length ≥ 4 contains a chord (see Figure 6.5).

A graph G is *perfect* if every induced subgraph $G' \subseteq G$ satisfies $\chi(G') = \omega(G')$. Permutation graphs, generalized split graphs and chordal graphs are subclasses of perfect graphs [13, 21] that we consider in this study. These three graph classes are *hereditary*; that is, induced subgraphs of a graph from any of those classes belong to the same class as the original graph does. Also, there is no containment relation among these three graph classes, but there are graphs that belong to more than one of these three classes. Figure 2.8 depicts the interrelations among them by providing example graphs for each possible case.

A graph G is called an *intersection graph* of F if there is a bijection between the set of vertices $\{v_1, v_2, \ldots, v_n\}$ of G and the sets in F such that v_i and v_j are adjacent if and only if $S_i \cap S_j \neq \emptyset$, for $1 \leq i, j \leq n$.

An *interval graph* is the intersection graph of a family of intervals on the real line, where each interval is represented by a vertex.

A *unit disk graph* is the intersection graph of a collection of disks with the same radius in the Euclidean plane, where each disk corresponds to a set in F.

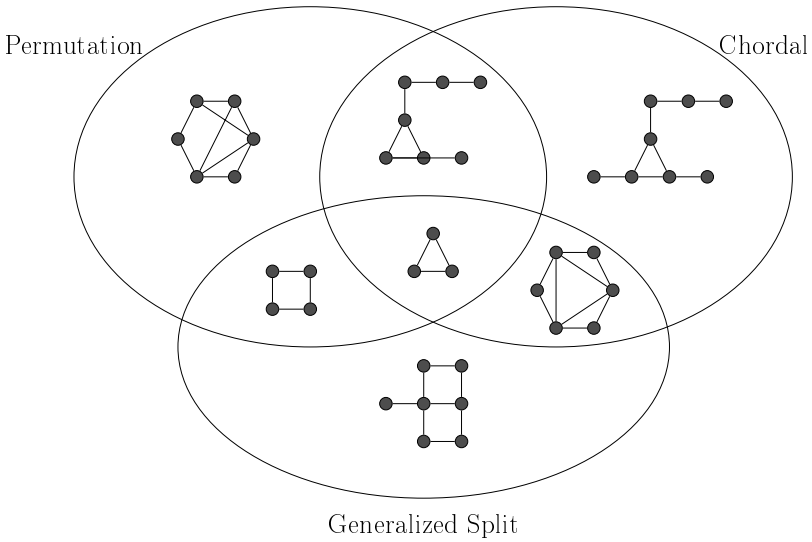


Figure 2.8. A Venn diagram illustrating the interrelations among permutation, chordal, and generalized split graphs.

3. LITERATURE REVIEW

This chapter summarizes the existing studies from the literature that are directed towards possible solution strategies for SEL-COL. We mention the relevant studies from the literature on random graph generation in Chapter 6, as needed.

The selective graph coloring problem, or partition coloring problem has emerged as an alternative model to solve the path coloring problem in wavelength assignment context [7]. Given a list of pairs of nodes in a network, the objective in the path coloring problem is to find a set of paths between the given source-destination pairs and to choose one path between every pair of source-destination in such a way that the number of colors to color the paths is minimized, where two paths sharing a common edge cannot be assigned the same color. In the literature, there exist some studies concerning the path coloring problem in its original version; that is, in which vertices and edges in the graph model serve as representatives of the nodes and links in the actual network (see for instance [22–24]). In order to handle a path coloring problem as a selective coloring problem, a two-phase approach is adopted. The first phase consists in finding a set of routes between the source-destination pairs in the original network. In the second phase, a graph model is constructed first by adding a vertex for each route between the given source-destination couples and placing an edge for every pair of routes sharing a common link in the original network [7]. The selective graph coloring problem is then solved on the graph model constructed, where the set of vertices associated with the set of routes connecting a given source-destination pair forms a cluster. Selection of vertices in the graph model corresponds to choosing a route for each source-destination pair, and coloring the vertex selection corresponds to assigning proper wavelengths to each route. Figure 3.1 shows a small illustrative network and the corresponding selective graph coloring model. The node pairs (a, b), (a, c), and (d, e) in the original Wavelength Division Multiplexing (WDM) optical network (left) stand for three source-destination couples, and are presented as clusters in the selective model (right). A best selection in this example is to select vertices 2, 4, and 6 for which only

a single color suffices.

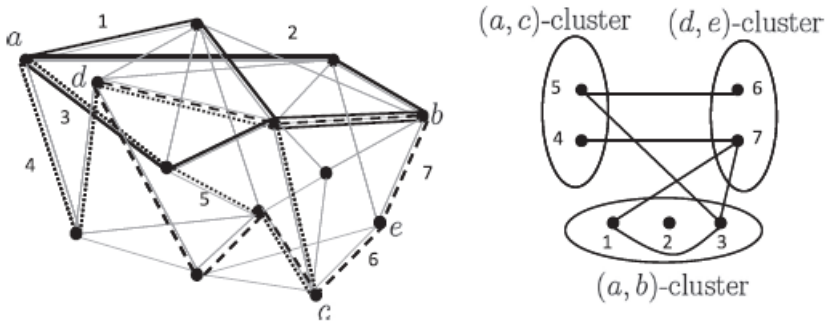


Figure 3.1. An example WDM optical network of routing and its selective graph coloring model (taken from [1]).

Besides wavelength assignment context, the selective framework can be adapted to a wide range of application areas. The authors in [1] elaborate on the selective graph coloring problem by reviewing some models from literature and proposing some new ones from various contexts including wavelength assignment (which is originally introduced in [7]), frequency assignment, scheduling, and more. For each domain of application, they discuss the related model and the complexity of selective coloring problem in classes of graphs motivated by that area. Also, analogies between models under certain cases as well as new solution approaches and open research directions are pointed out throughout the paper.

The computational complexity of the selective coloring problem in various graph classes is investigated by Demange *et al.* [6]. Many graph classes including split graphs, interval graphs, bipartite graphs, complete q-partite graphs and more are examined, some with specific constraints on the sizes of clusters. For each graph class under consideration, corresponding complexity statuses are proven and reported. Another study by Bonomo *et al.* [25] introduces the notion of selective-perfect graphs, studies perfectness structurally in selective framework, and gives a complete characterization of selective-perfect graphs.

We now mention the studies from the literature, which concentrate on solution strategies for the selective coloring problem, or partition coloring problem as it is alternatively called. We start from the ones that focus on heuristic approaches and then move to studies that propose exact solution approaches. The first such study is by Li and Simha [7], who introduce the partition coloring problem as an alternative representation of the path coloring problem in the context of wavelength routing and assignment problem in WDM optical networks. They show the complexity of the partition coloring problem via reduction from standard graph coloring. As mentioned above, the authors adopt a two-phase approach. First, they find one or more routes in the original optical network, and then construct the graph model for partition coloring. They review the three best known approximation algorithms for standard graph coloring problem and extend them to the partition coloring problem. The methods, which are constructive heuristics, are tested and reported on random instances as well as on networks from wavelength routing and assignment.

A later study by Noronha and Ribeiro [26] focuses on the problem of routing and wavelength assignment in optical networks, as Li and Simha do in [7]. The authors first compute a set of possible routes for each lightpath in the original optical network via heuristic methods for finding edge-disjoint paths, and then solve the partition coloring problem in the graph model with their proposed tabu search heuristic.

A recent thesis study by Çalık [9] presents exact and heuristic algorithms for the selective coloring problem. First, two alternative integer programming formulations are presented. Then, a compact integer programming formulation, and a reformulation of it for interval graphs are introduced. All the cliques of the input interval graph are generated, corresponding constraints are added to the compact formulation at the beginning, and the resulting program is solved to optimality in one go. Then, Çalık's thesis study applies this approach to unit disk graphs to find maximum selective cliques, which ultimately yield a lower bound on the selective chromatic number. As opposed to interval graphs, clique constraints are not added at once in the case of unit disk graphs; the problem is divided into a master problem and a subproblem, and constraints

are added step by step. The decomposition strategy we present in this dissertation is inspired from the approach proposed in Çalık's thesis study. We generalize the approach to more general graph types and solve the problem efficiently by adding constraints dynamically. In the second part of Çalık's thesis study, heuristic methods for the selective coloring problem are proposed. Initially, three different construction heuristics, i.e., methods that are used for obtaining an initial solution, are given. Then, three tabu search algorithms that use the output of the best-performing construction heuristic as a starting point are introduced, and two of them are reported to outperform those by Noronha and Ribeiro given in [26].

To the best of our knowledge, there are three studies that primarily focus on exact solution methods for the selective graph coloring problem. In the first one [27], Frota *et al.* present a branch-and-cut algorithm for the partition coloring problem. Their integer programming formulation picks one vertex to be the representative of vertices having the same color, instead of coloring all vertices in the selection. After some preprocessing operations to reduce the graph size, they implement a branch-and-cut algorithm with a branching rule custom-tailored for the partition coloring problem. The rule is a modification of the classical one used in [28] that branches on two non-adjacent vertices for the graph coloring problem. Also, a tabu search procedure is used to obtain an upper bound on each node of the branch-and-cut tree.

Another study by Hoshino *et al.* [29] proposes a new integer programming model and branch-and-price algorithm to solve the partition coloring problem. Their formulation combines the representative vertex concept introduced by [27] and the new idea that any solution to the partition coloring problem can be expressed as a collection of disjoint independent sets each having at most one vertex per cluster. In all instance classes tested, their algorithm is said to demonstrate superior performance than the one by Frota *et al.* [27].

A recent study by Furini *et al.* [30] proposes a new formulation for Sel-Col with an exponential number of variables and designs a branch-and-price algorithm to solve

it, where pricing phase is based on a single pricing problem, as opposed to the work by Hoshino *et al.* which requires to solve several pricing problems. Computational results indicate that the proposed branch-and-price framework improves on the previous state-of-the-art exact approaches from the literature. We use the open-source implementation of this algorithm to compare our results to.

4. INTEGER PROGRAMMING FORMULATIONS FOR SEL-COL

One can make use of integer programming (IP) formulations to solve SEL-COL. Given a graph $G = (V, E)$ with a partition of its vertex set into P clusters $V_1, ..., V_P$; i.e., $\bigcup_{p=1}^{P} V_p = V$ where $V_p \subset V$, $V_p \cap V_q = \emptyset$, $V_p \neq \emptyset \ \ \forall\ p, q \in \{1, ..., P\}$ and $p \neq q$, two alternative IP formulations will be provided here, as given in [9].

Model 1:

$$\min \sum_{s \in S} x_s \tag{4.1}$$

s.t.

$$\sum_{\{s \in S | s \cap V_p \neq \emptyset\}} x_s = 1 \quad \forall p \in \{1, ..., P\} \tag{4.2}$$

$$x_s \in \{0, 1\} \quad \forall s \in S, \tag{4.3}$$

where S is the set of all selective stable sets, i.e., stable sets that contain no more than one vertex per cluster, and x_s is a binary variable that takes the value 1 if stable set s is selected and zero otherwise. A feasible solution of the above model selects a number of disjoint stable sets, each representing a color class, such that exactly one vertex per cluster is selected, as secured by constraint (4.2). The optimal objective function value gives the minimum number of colors needed to color exactly one vertex from each cluster. One should note that a graph contains exponentially many selective stable sets in general, and hence there is an exponential number of variables in this formulation.

Model 2:

$$\min \sum_{k=1}^{P} y_k \tag{4.4}$$

s.t.

$$x_{ik} \leq y_k \quad \forall (i,k) \in V \times \{1,...,P\} \tag{4.5}$$

$$x_{ik} + x_{jk} \leq 1 \quad \forall\ (i,j) \in E,\ k \in \{1,...,P\} \tag{4.6}$$

$$\sum_{i \in V_p} \sum_{k=1}^{P} x_{ik} = 1 \quad \forall p \in \{1,...,P\} \tag{4.7}$$

$$y_k \in \{0,1\} \quad \forall k \in \{1,...,P\} \tag{4.8}$$

$$x_{ik} \in \{0,1\} \quad \forall (i,k) \in V \times \{1,...,P\}, \tag{4.9}$$

where x_{ik} is a binary variable taking value 1 if vertex i is colored with color k and 0 otherwise, y_k is another binary variable having value 1 if color k is used and 0 otherwise.

Constraint set (4.5) forces y_k to be 1 if color k is used by some vertex, and (4.6) ensures that two adjacent vertices do not use the same color. Finally, the requirement that exactly one vertex must be chosen and colored from each cluster is achieved via constraint set (4.7). The fact that only one vertex must be selected from each cluster makes the number of clusters P in the partition a natural upper bound on the number of necessary colors: in the worst case the vertices in a selection would form a clique and all P colors would be needed. The objective function gives the total number of colors used by summing over all y_k variables.

One important issue about this formulation is that, due to the way x_{ik} variables are defined, a feasible solution can be equivalently expressed in a number of alternative ways by simply reindexing the colors used in that solution. Specifically, a feasible n-coloring of a selection will have $n!$ equivalent representations obtained by merely permuting the indices of those n colors. Moreover, any n-subset of P colors can equally be used in an n-coloring, and there are $\binom{P}{n}$ different ways to select them.

Then, each feasible solution using n of the P available colors has $\frac{P!}{(P-n)!}$ equivalent alternative solutions. For instance, if we were to have a 3-vertex complete graph (K_3) with each vertex comprising a cluster on its own, the only feasible choice would be to use three colors. However, that solution can be equivalently represented in 3! possible ways by merely permuting the indices of the three colors. This symmetry inherent in the formulation, or existence of "clones" in feasible space, poses additional burdens to branch-and-bound/cut procedures by causing them to explore and fathom these isomorphic copies of solutions during the search [31]. Therefore, we add the constraint set (4.10) to Model 2 in order to reduce symmetry.

$$y_k \geq y_{k-1} \quad \forall k \in \{2, ..., P\} \tag{4.10}$$

Constraints (4.10) place a hierarchy on available colors (similar to symmetry breaking constraints in [31]) by enforcing the program to use the colors in increasing order of their indices; that is, by only allowing the lowest numbered n colors to be used in an n-colorable feasible solution. This way, the subset of clone solutions that would arise from selecting alternative combinations of the P available colors is eliminated.

Note that y-variables can be relaxed as continuous since constraints (4.5) ensure that y variables will take on binary values in an optimal solution. However, Model 2 still contains $O(|V| \times P)$ binary variables and $O(|E| \times P)$ constraints. Its solution time increases exponentially with the input size, and may become intractable in relatively small graphs. We test the performance of Model 2 in our computational experiments presented in Chapter 7.

The next chapter offers a decomposition framework as a promising alternative to these IP formulations.

5. DECOMPOSITION APPROACH FOR SEL-COL

In this chapter, we present a formulation, first suggested in [9], to serve our decomposition purposes. As in the previous chapter, we assume that we are given a graph $G = (V, E)$ with a partition $\mathcal{V} = \{V_1, ..., V_P\}$ of V into P clusters. Let t denote an estimate of the number of colors needed. SEL-COL can alternatively be formulated as follows:

Model 3:

$$\min \quad t \tag{5.1}$$

s.t.

$$\sum_{i \in V_p} x_i = 1 \quad \forall p \in \{1, ..., P\} \tag{5.2}$$

$$t \geq \chi(G[x]) \tag{5.3}$$

$$t \geq 0 \tag{5.4}$$

$$x_i \in \{0, 1\} \quad \forall i \in V, \tag{5.5}$$

where x_i is a binary variable taking value one if vertex i is selected and zero otherwise, and $G[x]$ is the graph induced by the selection given by variable vector $x = (x_1, \ldots, x_n)$.

The nonnegative variable t is forced to be no smaller than the chromatic number of an optimal selection by constraint set (5.3), which means that the minimum value it takes is exactly the selective chromatic number of the input graph G, $\chi_{SEL}(G, \mathcal{V})$. The constraint set (5.2) ensures that exactly one vertex is chosen from each of P clusters. The model is not well defined in its current form because of the $\chi(.)$ operator in constraint set (5.3). In order to re-express (5.3) in linear form, we need to embed a set of linear inequalities to carry out the coloring task. This inclines us to separate the problem into two parts where selection task is handled in one and coloring of the

given selection in the other.

Let us first relax constraint set (5.3). Removal of (5.3) from Model 3 leaves an integer programming formulation, which yields a feasible vertex selection for G. This relaxed version of the model constitutes the initial master problem. At each step, the master problem is solved to optimality and the vertex selection found by it is passed to a subproblem, which computes the minimum number of colors needed to color that selection. If the optimal objective value of the subproblem is higher than the optimal t-value of the master, it means that the master problem has underestimated the chromatic number of the current vertex selection. In that case, we add a constraint to the master that forces t to be at least as large as this chromatic number, unless the particular selection is altered. One such constraint can be expressed as follows:

$$t \geq \chi(G[x^{(j)}]) \left(1 - \sum_{\{i \in V | x_i^{(j)} = 1\}} (1 - x_i) \right) \tag{5.6}$$

where $G[x^{(j)}]$ denotes the graph induced by the selection found at iteration j given by variable vector $x^{(j)}$, and $\chi(G[x^{(j)}])$ the minimum number of colors needed to color it.

The summation term in inequality (5.6) becomes zero when $x = x^{(j)}$; i.e., when exactly the same vertices as in $x^{(j)}$ are picked, and it increases by one with each vertex removed from the selection given by $x^{(j)}$. Then, the term in parentheses takes value one only when $x = x^{(j)}$, and decreases by the number of vertices a selection differs from $x^{(j)}$. Addition of cuts as defined in (5.6) to the master forces the program to search for selections that haven't been considered so far, because all yet unexplored selections lead to an objective value of zero, whereas others produce their own chromatic numbers being positive values. Since the chromatic number must be higher than zero for any selection, a new cut of type (5.6) should be generated for each selection considered by the master. So, optimality cannot be proven until all possible selections, which will in general be exponentially many, have been examined.

One minor improvement to the above explained scheme is to define variable t to be greater than or equal to one. We know that only a single color suffices to color a stable set. Then, if there exists a selection that is a stable set and if the master problem happens to find one such solution before all possible solutions are considered, objective values of both master and subproblem will take value one and the process will terminate. However, this can only help in cases when such selections are actually present in the input graph.

Cuts (5.6) are quite weak in the sense that they provide lower bounds on t only for the selection they have been generated from and become redundant for all others. We define an alternative cut as follows:

$$t \geq \chi(G[x^{(j)}]) - \sum_{\{i \in V | x_i^{(j)} = 1\}} (1 - x_i) \tag{5.7}$$

This constraint depends on the fact that the chromatic number of a graph induced by a selection can decrease by at most one for each vertex switch. To see this, take any subset $V' \subset V$ in some graph $G = (V, E)$ and suppose that we replace vertex $v \in V'$ with another vertex $w \in V \setminus V'$ to get V''. We first note that $\chi(G[V'' \cup \{v\}]) \leq \chi(G[V'']) + 1$ because an optimal coloring of $G[V'']$ can be extended to a proper coloring of $G[V'' \cup \{v\}]$ by assigning a new color to v. If the chromatic number decreases by more than one, without loss of generality say two, then we would have $\chi(G[V'']) + 1 = \chi(G[V']) - 1 \geq \chi(G[V'' \cup \{v\}])$, which is a contradiction since $V' \subset (V'' \cup \{v\})$ and therefore $\chi(G[V']) \leq \chi(G[V'' \cup \{v\}])$. Note also that it is possible to have $\chi(G[V']) - \chi(G[V'']) = 1$. For instance, suppose that v is adjacent to all vertices in $G[V' \setminus \{v\}]$ and w is not adjacent to any vertex in $G[V' \setminus \{v\}]$. In this case, we know that the color of v must be different than all colors used to optimally color $G[V' \setminus \{v\}]$, and exclusion of v decreases the minimum number of colors needed by one; i.e., $\chi(G[V' \setminus \{v\}]) = \chi(G[V']) - 1$. This yields $\chi(G[V'']) = \chi(G[V']) - 1$ because w can simply be assigned any of the colors used to optimally color $G[V' \setminus \{v\}]$.

As in constraint (5.6), the summation term in (5.7) takes value zero if $x = x^{(j)}$, and increases by the number of vertices a selection x differs from $x^{(j)}$. Then, the right hand side reduces by one for each vertex differing from the selection given by $x^{(j)}$, whereas that of constraint (5.6) decreases by $\chi(G[x^{(j)}])$, which makes (5.7) stronger than (5.6).

A natural lower bound on the chromatic number $\chi(G)$ of a graph G is the size $\omega(G)$ of a maximum clique in it. If G contains a clique of size k, we need k colors to color that clique, because each vertex is adjacent to all other vertices and therefore receives a distinct color. Since this argument is valid for any clique in the graph, we need at least $\omega(G)$ colors to color G. We can translate this relationship into an inequality as given in (5.8), and use it as an additional cut within our decomposition procedure.

$$t \geq \sum_{i \in K^{(j)}} x_i \tag{5.8}$$

where $K^{(j)}$ is a maximum clique of $G[x^{(j)}]$.

Constraint (5.8) sums over the variables that are contained in a maximum clique of a given selection. If exactly the same set of vertices as in $x^{(j)}$ is selected, then t is forced to be at least as large as the size of a maximum clique in it. For any other selection x, the cardinality of the intersection of x with $K^{(j)}$ becomes a lower bound on t, as subset of a clique is again a clique.

In the next section, we present how we can apply our decomposition approach to perfect graphs.

5.1. Decomposition Approach in Perfect Graphs

As mentioned before, the difficulty of SEL-COL is twofold: it may be due to the existence of exponentially many selections and/or due to the hardness of optimally

coloring the graph induced by the selection. The decomposition approach we present here separates the task of vertex selection and the coloring of the selected set of vertices. So, a natural attempt is to look into families of graphs where coloring problem is polynomial-time solvable, and the class of perfect graphs is one such class.

Recall that a graph G is perfect if the size of a maximum clique is equal to the chromatic number for every induced subgraph. Since the subproblem of our decomposition procedure deals with the coloring of subgraphs induced by a selection, we can utilize this equality relation between $\omega(G)$ and $\chi(G)$ by replacing constraint (5.7) with (5.8). We prefer constraint (5.8) over (5.7) because it is stronger for perfect graphs, which we show in the following.

Proposition 5.1. *Constraint (5.8) is stronger than (5.7) for perfect graphs.*

Proof. Given a graph G and a partition $\mathcal{V} = \{V_1, ..., V_P\}$ of its vertex set, let us first define the two polyhedra $\mathcal{P}_{5.7}$ and $\mathcal{P}_{5.8}$ as follows:

$$\begin{aligned}
\mathcal{P}_{5.7} \coloneqq \Bigg\{ & x \in [0,1]^{|V|},\ t \in \mathbb{R}_{\geq 0} \colon \sum_{i \in V_p} x_i = 1 \quad \forall p \in \{1, ..., P\}, \\
& t \geq \chi(G[\hat{x}]) - \sum_{\{i \in V \mid \hat{x}_i = 1\}} (1 - x_i) \quad \forall \hat{x} \in \{0,1\}^{|V|} \text{ s.t. } \sum_{i \in V_p} \hat{x}_i = 1 \\
& \forall p \in \{1, ..., P\} \Bigg\} \\
\mathcal{P}_{5.8} \coloneqq \Bigg\{ & x \in [0,1]^{|V|},\ t \in \mathbb{R}_{\geq 0} \colon \sum_{i \in V_p} x_i = 1 \quad \forall p \in \{1, ..., P\}, \\
& t \geq \sum_{i \in \hat{K}} x_i \quad \forall \hat{x} \in \{0,1\}^{|V|} \text{ s.t. } \sum_{i \in V_p} \hat{x}_i = 1 \quad \forall p \in \{1, ..., P\} \text{ and} \\
& \hat{K} \text{ is a maximum clique of } G[\hat{x}] \Bigg\}
\end{aligned}$$

In other words, if we let $\mathcal{P}$ be the LP relaxation of the polyhedron defined by the constraint set of our initial master problem, $\mathcal{P}_{5.7}$ and $\mathcal{P}_{5.8}$ are constructed by further constraining $\mathcal{P}$ respectively with constraints (5.7) and (5.8) defined for each one of all possible vertex selections. We want to prove that $\mathcal{P}_{5.8} \subseteq \mathcal{P}_{5.7}$. To do this, we first show

that for any $\{\bar{t}, \bar{x}\} \in \mathcal{P}_{5.8}$, $\{\bar{t}, \bar{x}\} \in \mathcal{P}_{5.7}$ holds. Since vertex selection constraints are common on both $\mathcal{P}_{5.7}$ and $\mathcal{P}_{5.8}$, $\sum_{i \in V_p} \bar{x}_i = 1 \quad \forall p \in \{1, ..., P\}$ holds by construction. Now, let $\hat{x} \in \{0,1\}^{|V|}$ such that $\sum_{i \in V_p} \hat{x}_i = 1 \quad \forall p \in \{1, ..., P\}$ and $\hat{K}$ is a maximum clique of $G[\hat{x}]$. Note that we can write $\sum_{i \in \hat{K}} \bar{x}_i = |\hat{K}| - \sum_{i \in \hat{K}} (1 - \bar{x}_i)$. Since $\hat{K} \subseteq V(G[\hat{x}])$, we have $\sum_{\{i \in V | \hat{x}_i = 1\}} (1 - \bar{x}_i) \geq \sum_{i \in \hat{K}} (1 - \bar{x}_i) \geq 0$. As $\chi(G[\hat{x}]) = |\hat{K}|$ by the perfectness of G, we have $\bar{t} \geq \sum_{i \in \hat{K}} \bar{x}_i = |\hat{K}| - \sum_{i \in \hat{K}} (1 - \bar{x}_i) \geq \chi(G[\hat{x}]) - \sum_{\{i \in V | \hat{x}_i = 1\}} (1 - \bar{x}_i)$ and hence $\{\bar{t}, \bar{x}\} \in \mathcal{P}_{5.7}$.

Next, we show that this containment can be strict; i.e., there exists a perfect graph G for which at least one point in $\mathcal{P}_{5.7}$ is not contained in $\mathcal{P}_{5.8}$. To this end, consider the graph $G = (V, E)$ with $\mathcal{V} = \{V_1, V_2, V_3\}$, where $V = \{1, 2, 3, 4\}$, $E = \emptyset$, $V_1 = \{1, 2\}$, $V_2 = \{3\}$, and $V_3 = \{4\}$. There are two possible selections for this graph, which are $\hat{x}^{(1)} = (1, 0, 1, 1)$ and $\hat{x}^{(2)} = (0, 1, 1, 1)$. The constraints of type (5.7) associated with selections $\hat{x}^{(1)}$ and $\hat{x}^{(2)}$ are respectively $c_1 : t \geq 1 - (3 - (x_1 + x_3 + x_4))$ and $c_2 : t \geq 1 - (3 - (x_2 + x_3 + x_4))$. Now, take the point $(\bar{t}, \bar{x}_1, \ldots, \bar{x}_4) = (0.5, 0.5, 0.5, 1, 1)$. This point is contained in $\mathcal{P}_{5.7}$, because it satisfies the selection constraints as well as c_1 and c_2. A maximum clique of $G[\hat{x}^{(1)}]$ is $\{3\}$. The corresponding constraint of type (5.8), $t \geq x_3$ is violated by the given point, as $\bar{t} = 0.5$ and $\bar{x_3} = 1$. Hence, $(0.5, 0.5, 0.5, 1, 1) \notin \mathcal{P}_{5.8}$, and $\mathcal{P}_{5.8} \subset \mathcal{P}_{5.7}$. □

We note that constraints (5.8) can be interpreted from a combinatorial point of view. In particular, each selective clique found in earlier iterations has a corresponding constraint (5.8) added to the master problem. Each such constraint forces the value of the t-variable to be greater than or equal to the number of vertices in the intersection of its corresponding selective clique and the current vertex selection (represented by the x-variables). Since the master problem minimizes t, it seeks a selection that yields a smallest intersection with previously generated selective cliques, and the subproblem finds a maximum clique in the induced subgraph identified by the vertex selection. If the subproblem agrees on the same objective value with the master problem, then it means that the union of selective cliques for which cuts (5.8) have been added to the master problem contains a maximum clique of the current selection. In this case,

we conclude that a better selection cannot be achieved and the process should be terminated. Pseudo-code of our decomposition algorithm for perfect graphs is given in Figure 5.1.

Input: A perfect graph $G = (V, E)$, and a partition $\mathcal{V}$ of V

Output: An optimal selection x^* with $\chi_{SEL}(G, \mathcal{V}) = z^*$

$j \leftarrow 0$, $t^{(j)} \leftarrow 0$, $z_{sp}^{(j)} \leftarrow \infty$

while true **do**

  $j \leftarrow j + 1$

  Solve the master problem to optimality, find an optimal selection $x^{(j)}$ having optimal objective value $t^{(j)}$

  Find a maximum clique $K^{(j)}$ of $G[x^{(j)}]$ in the subproblem

  $z_{sp}^{(j)} \leftarrow |K^{(j)}|$

  if $z_{sp}^{(j)} > t^{(j)}$ **then**

    Add (5.8) to the master problem

  else

    break

  end if

end while

$x^* \leftarrow x^{(j)}$, $z^* \leftarrow t^{(j)}$

Figure 5.1. Decomposition Algorithm for Perfect Graphs.

The decomposition algorithm for perfect graphs continues to search for a selection as long as the objective value of the subproblem exceeds that of the master problem. At each step j, the master problem is solved to optimality to obtain a selection $x^{(j)}$ with associated optimal objective value $t^{(j)}$. If the subproblem finds a maximum clique in $G[x^{(j)}]$ of size $z_{sp}^{(j)} = t^{(j)}$, then the process is terminated. At this point, the incumbent solution x^* and its corresponding objective value t^* are optimal.

Even though there is a polynomial-time algorithm to solve the coloring problem in perfect graphs, this algorithm does not perform very well in practice, as we will see in Section 7.4. Thus, we first focus on families of perfect graphs where well-performing polynomial-time combinatorial algorithms exist to solve the coloring, or equivalently maximum clique problem. The class of perfect graphs and its subclasses permutation, generalized split, and chordal graphs are *hereditary* graph classes; that is, induced subgraphs of a graph from any of those classes belong to the same class as the original graph does. Therefore, the maximum clique algorithms tailored for each of these classes also work on the subgraphs induced by selections and thus can be employed in the subroutine of our decomposition procedure.

In subsections 5.1.1–5.1.3, we describe how the decomposition approach is applied to the selected families of perfect graphs; namely, permutation, generalized split, and chordal graphs. This part of our work also appears in [10]. Then, we apply our algorithm to perfect graphs in its general form in 5.1.4 [11].

5.1.1. Permutation Graphs

A permutation graph is a graph where each vertex represents an element in a given permutation, and each edge represents a pair of elements that are reversed by the permutation, as mentioned in Chapter 2. Permutation graphs have many real world applications from flight altitude assignment to the memory reallocation problem [13]. One specific domain of application that motivates the study of the selective coloring problem in the class of permutation graphs is given by Demange *et al.* in [1]. The application is about allocating items to tracks from their own pick-up points to their own delivery spots, where both pick-up and delivery points have a predetermined sequence to be visited. Each track has a first-in last-out structure and swapping of items on tracks is not allowed. When each item has different possibilities of pick-up and/or delivery points, the problem consists in selecting one pick-up and one delivery point per item in order to minimize the total number of tracks required, which is shown to be equivalent to Sel-Col in permutation graphs by representing each item with a

set of properly defined intervals on the real line [1].

In order to apply the decomposition idea presented previously, we need to find maximum clique(s) in the graphs induced by the selections identified by the master problem. Since each induced subgraph of a permutation graph is again a permutation graph, we will make use of a general polynomial-time algorithm given in [13] to find maximum cliques in permutation graphs.

Given a permutation π, let $\pi(i)$ denote the number in position i of π, and $\pi^{-1}(i)$ denote the position of number i in π. Recall that a graph is a permutation graph if there is a permutation π of its n vertices in such a way that v_i and v_j are adjacent if and only if $i < j$ and $\pi^{-1}(v_i) > \pi^{-1}(v_j)$ (or equivalently $i > j$ and $\pi^{-1}(v_i) < \pi^{-1}(v_j)$). It follows from the definition of permutation graphs that vertices corresponding to a decreasing subsequence in a given permutation form a clique. Then, a maximum clique in a permutation graph can be found by identifying a longest decreasing subsequence in the corresponding permutation. For this purpose, each number is considered in the order they appear in π, and is enqueued to the end of the first available set. A set is available for some number $\pi(i)$ if the last number in the set, say $\pi(j)$ for some $j < i$, is less than $\pi(i)$. Otherwise, $\pi(j)$ becomes the predecessor of $\pi(i)$. The sets will be complete after considering all numbers in π. It can be observed that each set corresponds to an increasing subsequence in π and thus is a stable set of the related graph. One should note that the predecessor of a number must be greater than itself. Moreover, it is known that such a partition of vertices into stable sets yields a minimum coloring of G. Consequently, we can identify a maximum clique (equivalently a longest decreasing subsequence) by starting from a number in the last stable set and tracing back the predecessors all the way up to the first stable set. Thus, the size of a maximum clique will be equal to the number of stable sets constructed.

It is also possible to generate all maximum cliques in the graph. For this purpose, we can list all predecessors of each number and trace back through each one to reveal all maximum cliques in the graph. To list all predecessors of a number $\pi(i)$, after setting

the first predecessor of it, we can simply search back from the index of first predecessor in that stable set, and include all numbers that are greater than $\pi(i)$ [13]. The number of maximum cliques in a permutation graph can be exponential. For instance, consider a graph G comprised of the disjoint union of n edges, which can be represented by the permutation $\pi = (2, 1, 4, 3, 6, 5, \ldots, 2n, 2n-1)$. The complement of a permutation graph being a permutation graph, if we take $\bar{G}$ as the input graph, we can search for all the maximum stable sets in G to find all the maximum cliques in $\bar{G}$. A maximum stable set of G has size n and must contain exactly one vertex from each one of the n edges. Since there are two ways to select a vertex from each edge, we have 2^n distinct maximum stable sets in G. Although the number of maximum cliques in a permutation graph can be exponential in general, as this example reveals, the algorithm to find all maximum cliques turned out to perform quite efficiently in practice. However, in our preliminary experiments, we observed that generating a single maximum clique and adding related cuts each time the subproblem is called yields better results. Therefore, each time the subproblem is called, we generate one maximum clique in the graph induced by that selection.

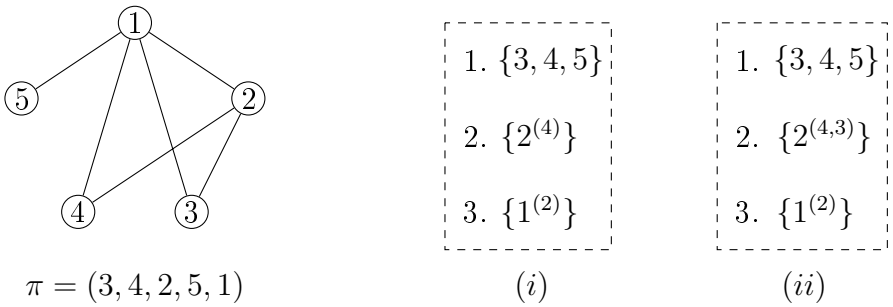


Figure 5.2. An example permutation graph together with two alternative groups of sets that would produce the maximum clique(s) in it.

In order to illustrate how the maximum clique algorithm works, in Figure 5.2 we provide a permutation π, its corresponding graph, and two groups of sets that would be produced if the algorithms for finding a single maximum clique and all maximum cliques are run respectively. Each row of numbers in two alternative implementations (i) and (ii) lists the stable sets, and the superscript of each number contains the predecessor(s)

of each number. If we backtrack the predecessors in version (i), we obtain maximum clique $\{1, 2, 4\}$; and if we do the same in version (ii), we get all maximum cliques $\{\{1, 2, 4\}, \{1, 2, 3\}\}$.

5.1.2. Generalized Split Graphs

A graph $G = (V, E)$ is a generalized split graph if there exists a partition of its vertex set V into V_1 and V_2 of the vertex set such that either $G[V_1]$ is a union of pairwise disjoint cliques and V_2 is a clique in G, or $\bar{G}[V_1]$ is a union of pairwise disjoint cliques and V_2 is a stable set in G, as mentioned in Chapter 2 (see Figure 6.3 for generic visual representations of generalized split graphs). An important characteristic of generalized split graphs is that they are shown to comprise almost all perfect graphs [32]. Since it is such a large subclass of perfect graphs, it is important to test the performance of our decomposition approach on them.

Let $G = (V, E)$ be a generalized split graph with $V = V_1 \cup V_2$ such that $V_1 = \bigcup_{i=1}^{k} H_i$ and $V_2 = H$. If G is unipolar, V_1 is the union of k disjoint cliques $H_1, \ldots, H_k$ and V_2 is a clique; if it is co-unipolar, V_1 induces a complete k-partite graph with k-partition $H_1, \ldots, H_k$ and V_2 is a stable set. In order to find maximum cliques in a generalized split graph, we first need to know whether it is unipolar or co-unipolar. When generating random generalized split graphs (the generation method is explained in Section 6.1.2), we label the output graph as unipolar or co-unipolar and also record its partition to make use of them later in our solution procedure.

In the unipolar case, we use the polynomial-time method that Eschen and Wang utilize in [33]. The first observation is that H_i's for $i = 1, \ldots, k$ being disjoint, a clique cannot contain vertices from distinct H_i components. Hence, one should identify a maximum clique in each one of the subgraphs $G[H \cup H_i]$ and find the largest among the k cliques. In order to find a maximum clique in each one of these k induced subgraphs easily, the authors in [33] remark that the complement of $G[H \cup H_i]$ forms a bipartite graph with bipartition H, H_i. The well-known König–Egerváry Theorem [34,35] states

Input: A generalized split graph G and its partition $H, H_1, \ldots, H_k$

Output: K as a maximum clique of G

$K \leftarrow \emptyset$

if G is unipolar **then**

  for $i = 1$ to n **do**

    Take the complement of $G[H \cup H_i]$ and find a maximum stable set S in it

    if $|S| > |K|$ **then**

      $K \leftarrow S$

    end if

  end for

else

  $max \leftarrow 0$

  for all $v \in H$ **do**

    Let $N'(v)$ be a set of neighbors of v from distinct H_i components

    if $|N'(v)| > max$ **then**

      $max = |N'(v)|$, $v^* \leftarrow v$, $N^* \leftarrow N'(v)$

    end if

  end for

  if $k < max$ **then**

    $K \leftarrow N^* \cup \{v^*\}$

  else

    for $i = 1$ to k **do**

      Arbitrarily select one vertex v_i from H_i

      $K \leftarrow K \cup \{v_i\}$

    end for

  end if

end if

Figure 5.3. Maximum clique algorithm for generalized split graphs.

that in a bipartite graph the size of a maximum matching equals the size of a minimum vertex cover. Moreover, if V' is a minimum vertex cover in $G = (V, E)$, then $V \setminus V'$ is a maximum stable set of G. So, in order to find a maximum clique in a unipolar graph, we seek maximum stable sets in the complement of each $G[H \cup H_i]$ to find a maximum clique in the entire graph because a stable set in the complement corresponds to a clique in the original graph.

To find a maximum clique in a co-unipolar graph, we first explore a vertex v in H that is connected to the maximum number of distinct H_i's. Since H_i's form a complete k-partite graph, the neighbors in distinct H_i's together with v form a clique. If the size of that clique is greater than k, then we are done. Otherwise, we can arbitrarily select a single vertex from each H_i for $i = 1, \ldots, k$ and output it as a maximum clique. This process takes polynomial time; in particular $O(|V| + |E|)$.

As in permutation graphs, it is possible to list all maximum cliques in a given generalized split graph. However, especially in cases where the graph is co-unipolar and the maximum cliques turn out to be all combinations of vertices from each H_i, the number of possible maximum cliques can be exponentially many. In our preliminary experiments, we observed that it is not practical to enumerate all maximum cliques. Hence, we confine ourselves to finding a single maximum clique each time the subproblem is called. Furthermore, in the case of unipolar graphs, we extend that single maximum clique of the selection to a maximal clique in the whole graph to lift our cut (5.8).

5.1.3. Chordal Graphs

A graph that contains no induced cycle of length four or more is called a chordal graph, as we referred to in Chapter 2. A chordal graph on n vertices has at most n maximal cliques, with equality if and only if the graph contains no edge [36]. Since the upper bound on the number of maximal cliques is reasonably small and there exists a linear-time algorithm to generate all maximal cliques in a given chordal graph [37],

we generate all maximal cliques of G beforehand and construct the associated cut set that is mathematically expressed in (5.9).

$$t \geq \sum_{i \in K} x_i \quad \forall K \in \mathcal{K} \tag{5.9}$$

where $\mathcal{K}$ denotes the set of all maximal cliques for the given chordal graph G.

We tailored our decomposition procedure for the case of chordal graphs in a slightly different (but equivalent) manner: The master problem starts with finding an optimal selection without any clique constraints imposed. Given an optimal selection identified by the master problem, the subproblem checks whether any of the constraints from the set (5.9) are violated and incorporates such constraints, if any, to the master. In other words, instead of generating the clique cuts one by one at each step, we construct them beforehand and add as needed, because the number of constraints to go through is linear in the size of the vertex set of the input graph in the case of chordal graphs. If, at some iteration, the master finds a selection which does not violate any of the constraints in (5.9), it means that the current state of the master is able to produce an optimal solution that satisfies all the clique constraints and hence an optimal solution for the entire problem is reached. It should be noted that the constraint set (5.9) is defined for the maximal cliques on the entire graph, not on selections. However, the formulation of the master already involves constraints making the x-variables to take value one only for vertices in the selection, which guarantees that the t-variable, equivalently the objective value, is forced to be at least as large as the cardinality of the intersection of a maximal clique with the selected set of vertices. Moreover, a maximal clique of a selection can always be extended to a maximal clique in the whole graph, which means that the set of all selective maximal cliques are covered by the set of all maximal cliques of the whole graph. Since the constraint set (5.9) is defined for each one of the maximal cliques in the graph, the minimum value that variable t ultimately takes on for some given selection will be exactly the size of a maximum clique in that selection.

5.1.4. General Perfect Graphs

In the case of perfect graphs in its general form, coloring, or equivalently maximum clique problem is polynomially solvable via semidefinite programming (SDP) [18]. Finding the size of a maximum clique in a perfect graph necessitates solving an SDP only once. However, extracting the maximum clique itself involves solving a series of SDP problems on successively smaller graphs for at most n times, where n is the number of vertices in the input graph.

Lovász introduced a function ϑ, known as *Lovász's theta number* or *theta function*, which is polynomial-time computable [38]. Given a graph G, Lovász's theta number $\vartheta(G)$ satisfies the following inequality:

$$\alpha(G) \leq \vartheta(G) \leq \chi(\bar{G}) \tag{5.10}$$

One should note that $\alpha(G) = \omega(\bar{G})$ for any graph, and $\chi(\bar{G}) = \omega(\bar{G})$ for perfect graphs. Then, $\vartheta(G) = \omega(\bar{G})$ holds for perfect graphs. In order to find the clique number $\omega(G)$ of a perfect graph G, we need to use the complement of the graph $\bar{G}$, which is again perfect by Theorem 6.16. The theta number $\vartheta(G)$ can be computed by several equivalent formulations [39–41]. We provide one of these formulations in (5.11)–(5.15), which is an SDP problem.

Let us introduce a few notations first.For two matrices $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times n}$, the trace inner product is denoted by $A \bullet B = \text{trace}(A^T B) = \text{trace}(BA^T) = \sum_{i,j} A_{ij}B_{ij}$. A symmetric real matrix A is said to be *positive semidefinite* if the scalar $zA^T z \geq 0$ for every $z \in \mathbb{R}^n$. For an $n \times n$ real symmetric matrix A, we use $A \succeq 0$ to indicate that A is positive semidefinite. Finally, we use $\mathcal{S}^{n\times n}$ to denote the space of $n \times n$ symmetric matrices.

$$\max \quad J \bullet X \tag{5.11}$$

s.t.

$$I \bullet X = 1 \tag{5.12}$$

$$X_{ij} = 0 \quad \forall \{i,j\} \in E \tag{5.13}$$

$$X \succeq 0 \tag{5.14}$$

$$X \in \mathcal{S}^{n \times n}, \tag{5.15}$$

where I is the identity matrix, J is a matrix of all ones, and E is the edge set of the input graph.

The optimal objective value of the SDP model provided in (5.11)–(5.15) gives the stability number $\alpha(G)$ of a perfect graph G. However, we cannot directly obtain a maximum stable set itself by solving this model once. A study by Grötschel *et al.* [18] proposes a method to extract a maximum stable set in perfect graphs by repeatedly computing the stability number in smaller induced subgraphs of the input graph. The main idea of this method is to remove vertices from the input graph until only the vertices of one maximum stable set remains. It works as follows: First, we find the stability number $\alpha(G)$ of the original input perfect graph G. Then, we mark all vertices of $G = (V, E)$ unlabeled. At each step, we select an unlabeled vertex $v \in V(G)$ and tentatively remove it from G. Note that $G' = G \setminus \{v\}$ is an induced subgraph of G, and hence is perfect, too. We then calculate $\alpha(G')$. If $\alpha(G') = \alpha(G)$, we set $G = G'$, because it means that v is not contained in all maximum stable sets of G and its removal will leave at least one maximum stable set intact. If $\alpha(G') < \alpha(G)$, then it means that v intersects with all of the maximum stable sets in the current graph and cannot be eliminated. Therefore, we label v in this case and keep it in our vertex set. This process continues until there is no unlabeled vertex, in which case the set of all remaining (labeled) vertices form a maximum stable set of the original graph. Since we either label or remove a vertex at each step, each vertex is considered once in this method. It outputs a maximum stable set after n iterations, with n being the number of vertices of the original input graph. It is also possible to find other maximum stable

sets, if any, by changing the order of vertices to be considered.

This method is the first polynomial-time algorithm for the maximum stable set problem in perfect graphs. Since we are interested in finding a maximum clique, which corresponds to a maximum stable set in the complement of the graph, we simply give the complement of the original graph as input. At each step of this method, we make use of the SDP model provided in (5.11)–(5.15) to find a maximum stable set.

We made a minor modification to Grötschel *et al.*'s algorithm in order to possibly avoid unnecessary computations. As we compute the size of a maximum stable set at the beginning of the algorithm, we continue iterating until the number of labeled vertices in the graph equals maximum stable set size, instead of waiting for all vertices to be considered.

Although SDP is a polynomial-time method (up to any desired accuracy) in theory, it does not perform very well in practice, as will be revealed by the results of our computational experiments presented in Section 7.4. As a promising alternative to solve the maximum clique problem in perfect graphs, we first considered a cutting plane algorithm proposed in a recent master thesis by Düzgün [42]. The algorithm makes use of the LP relaxation of a maximum stable set model, which is given in (5.16)–(5.18).

$$\max \sum_{i \in V} w_i \tag{5.16}$$

s.t.

$$\sum_{i \in C} w_i \leq 1 \quad \forall C \in \mathcal{C} \tag{5.17}$$

$$w_i \geq 0 \quad \forall i \in V, \tag{5.18}$$

where $\mathcal{C}$ is the set of all maximal cliques in the input graph.

Note that (5.17) may consist of an exponential number of constraints, which can be handled by cutting plane algorithm. The cutting plane algorithm starts with the

linear programming (LP) relaxation of a maximum stable set model with only a subset of its original set of maximal clique constraints included. Initially, with the help of heuristic methods, some maximal cliques are generated and added to the model. The algorithm reaches a maximum stable set iteratively, by adding violated inequalities and solving the current model anew each time. At the beginning of each iteration, current LP model is solved to optimality. If the solution vector is integral and no two adjacent vertices are included in the solution, i.e., if the solution actually is a stable set, then the current solution is optimal and the algorithm stops. If the solution vector contains fractional values, the algorithm identifies a most violated maximal clique constraint by formulating a separate IP model, solves it, and adds cutting planes to the master obtained from the violated clique.

Düzgün's algorithm is designed to find a maximum stable set. Nevertheless, since a maximum stable set of a graph corresponds to a maximum clique in its complement and the complement of a perfect graph is again perfect by Theorem 6.16, we can safely utilize this algorithm by giving the complement of the original graph as input to the algorithm.

In addition to Düzgün's algorithm that is tailored for perfect graphs, we also tested the performance of our decomposition approach on perfect graphs by incorporating a maximum clique algorithm by Tomita *et al.* [43] that works on any type of graph. It turned out that, although Düzgün's algorithm benefits from perfect graphs' structural properties, Tomita *et al.*'s general algorithm performed better, as we will show in Section 7.4. We will provide some more details about Tomita *et al.*'s algorithm in the next subsection.

5.2. Decomposition Approach in General Graphs

When the aim is to solve Sel-Col in perfect graphs, it suffices to find a maximum clique in the subproblem in our decomposition approach, because the size of a maximum clique is equal to chromatic number in perfect graphs. In a graph with no particular

structure, however, maximum clique size is just a lower bound to the chromatic number. Hence, addition of maximum clique cuts to the master does not merely suffice to lead us to the selective chromatic number of the input graph; we also need to incorporate the chromatic number information in our solution procedure.

One of the perks of using clique cuts as given in (5.8) is that it not only provides a bound for the selection it has been generated from, but does so for all possible selections by the number of vertices that intersect with the formerly found cliques. So, it is still beneficial to use clique cuts even when dealing with general graphs. However, at any iteration, having an optimum selection x with a master objective value equal to the maximum clique size in $G[x]$ does not guarantee optimality for the overall problem, as opposed to the case of perfect graphs; here, we should check if the optimum objective value of the master problem is equal to the chromatic number of the graph induced by that selection. To include the chromatic number information in our solution procedure, we make use of constraint (5.7). We prioritize constraints (5.8) as long as they are violated, because solving the coloring problem takes much more time than solving the maximum clique problem in practice. We modified our decomposition method so that, if the current selection found by the master yields an objective value less than the size of the maximum clique in it, then we add the related clique constraint to master and continue iterating; otherwise, we find the minimum number of colors to color the vertices in the selection and add constraint (5.7) if it is violated by the current solution. Pseudo-code of our decomposition method for general graphs is given in Figure 5.4.

Our algorithm first checks the clique constraint, and if it is violated, it avoids solving a coloring problem. Hence, in the first stage, the algorithm will only generate clique constraints, until the master problem can correctly estimate the size of a maximum clique in the best possible selection, where goodness of a selection is measured in terms of its maximum clique size in the first stage. If, at some iteration, the objective value of the master problem turns out to be equal to the size of the maximum clique of the current selection, then it means that the union of maximum cliques whose corresponding constraints have been added to the master problem already contains a

Input: A graph $G = (V, E)$, and a partition $\mathcal{V}$ of V

Output: An optimal selection x^* with $\chi_{SEL}(G, \mathcal{V}) = z^*$.

$j \leftarrow 0$, $t^{(j)} \leftarrow 0$

while true **do**

  $j \leftarrow j + 1$

  Solve the master problem to optimality, find an optimal selection $x^{(j)}$ having optimal objective value $t^{(j)}$

  Find a maximum clique $K^{(j)}$ of $G[x^{(j)}]$ in the subproblem via MCS Algorithm [43]

  if $|K^{(j)}| > t^{(j)}$ **then**

    Add (5.8) to the master problem

  else

    Find $\chi(G[x^{(j)}])$ in the subproblem

    if $\chi(G[x^{(j)}]) > t^{(j)}$ **then**

      Add (5.7) to the master problem

    else

      break

    end if

  end if

  $x^* \leftarrow x^{(j)}$, $z^* \leftarrow t^{(j)}$

end while

Figure 5.4. Decomposition algorithm for general graphs.

maximum clique of the best selection with respect to maximum clique size. In that case, we need to check if the chromatic number of the graph induced by this selection exceeds the objective value from the master problem. In a sense, coloring is the last resort in the subproblem when the master problem fails to find any selection that reveals a violated clique cut. Note that a best selection with respect to maximum clique size need not be the best with respect to the chromatic number. Therefore, we can still

generate cuts (5.8) in addition to (5.7) during the second stage of the algorithm. When the master problem yields an objective value that is equal to the chromatic number of the graph induced by that selection, then we know that no better selection is possible and we have reached an optimal solution.

In order to apply our decomposition approach to general graphs, we need both a maximum clique and a minimum coloring algorithm that can work on any type of graphs, which we discuss in the sequel.

A recent comprehensive review on both exact and heuristic algorithms for maximum clique problem by Wu and Hao [44] provides computational performance comparison of ten state-of-the-art exact algorithms on a set of popular DIMACS instances. One of the best-performing algorithms is that of Tomita *et al.* [43], which is a branch-and-bound algorithm that the authors call *MCS*. MCS is based on a previous maximum clique algorithm *MCR* by Tomita and Kameda [45] and shows considerably improved performance compared to the previous with the help of newly introduced techniques that reduce the search space.

MCR [45] is a basic branch-and-bound algorithm that begins with a small clique and continues searching for larger and larger cliques until it finds one that can be confirmed to be of maximum size. At every step, it starts from a single vertex and tries to expand it by adding new vertices. In order to avoid unnecessary searching, the algorithm makes use of a greedy coloring of the set R of common neighbors of vertices in the current clique Q. Greedy coloring assigns a minimum possible (integer) label to each vertex in R, which simply implies that the size of a maximum clique in R, $\omega(R)$, can be at most the maximum label used in greedy coloring. Then, current clique Q can be extended by at most $\omega(R)$ vertices. So, if sum of $|Q|$ and maximum label from greedy coloring does not exceed the size of a clique of maximum size found so far, $|Q^*|$, then there is no need to continue searching for vertices to be included in Q because it is simply not possible to obtain a larger clique on that branch.

In the improved maximum clique algorithm MCS [43], which we utilize in our decomposition algorithm, the authors focus on reducing the search space further by incorporating a recoloring routine. This routine aims to improve the coloring obtained from the greedy coloring procedure by recoloring vertices with the largest color label into a smaller one. The basic idea of this strategy is that if a vertex v having a color label greater than a threshold label $k_{th} = |Q^*| - |Q|$ is transferred to a color class with color label $k \leq k_{th}$, then it is no longer necessary to search from vertex v, because a larger clique cannot be obtained on the branch hanging from v. To save time, the authors apply this procedure only to vertices with the largest color label.

According to a survey on graph coloring problems by Malaguti and Toth [46], two best performing exact algorithms for graph coloring are a column generation approach by Mehrotra and Trick [28] and a branch-and-cut algorithm by Méndez-Díaz and Zabala [47]. On instances that are solved by both algorithms, the method based on column generation has larger computing time. This is the main reason for us to prefer the branch-and-cut algorithm in [47] and implement it to solve the coloring problem in our decomposition procedure. The authors in [47] first present a set of facet-defining and valid inequalities for graph coloring problem to be used in the branch-and-cut algorithm. Since the number of variables and inequalities in the model may become far too large even for moderately sized coloring problems, they use preprocessing techniques to possibly eliminate unnecessary variables and constraints. Additionally, the authors make use of a heuristic method to obtain an upper bound, improve the LP relaxation of the model, and propose a branching strategy that proved to be quite effective among several others that are tested.

In our preliminary experiments, we observed that our decomposition approach performs poorly as compared to the other two methods, i.e., the integer programming formulation Model 2 and the branch-and-price method by Furini *et al.* [30], and majority of the solution time is spent in coloring algorithm. For that reason, we decided to incorporate another promising coloring algorithm from the literature. One such algorithm, called *exactcolors*, is the one by Held *et al.* [48], which is an implementation

of the Mehrotra-Trick column-generation approach for graph coloring [28] and is available online. As we will see in Section 7.5, this algorithm did not help ameliorate the performance of our method either.

6. DATA GENERATION

In order to test the performance of the solution techniques we consider, we need problem instances. The decomposition approach we present in this study, whose details are discussed in Chapter 5, separates the task of vertex selection and coloring of the selected vertex set. Therefore, a natural focus of this study is on the families of graphs that are hereditary and in which coloring problem is polynomially solvable. The class of perfect graphs has this property and thus it constitutes the primary focus of this study. We consider the class of perfect graphs in its general form, and also subclasses of it, which possess additional desirable properties that we can benefit from in our solution procedure.

Considering the wide range of aforementioned practical application areas in which the graph classes we focus on naturally arise, and the fact that many studies from the literature deal with designing efficient algorithms that exploit properties of various graph classes (see for instance [14–18]), the need for availability of general-purpose problem instances from special graph classes arises as a natural consequence. We make all of our problem instances available online (specific URLs are provided in the related sections that follow) [10, 11, 19, 20]. To the best of our knowledge, our study is the first in making a large suite of randomly generated instances from the class of perfect graphs and some of its special subclasses online accessible.

A complete problem instance for Sel-Col consists of a graph $G = (V, E)$ and a partition $\mathcal{V}$ of its vertex set V. In the rest of this chapter, we describe algorithms to generate random graphs and vertex set partitions, and provide some relevant definitions and characterizations that we utilize in this context.

6.1. Random Graph Generation

In this section, we present our random graph generators for each graph class under consideration. The first three are permutation graphs, generalized split graphs, and chordal graphs, which are subclasses of perfect graphs, and the last one is the class of perfect graphs in its general form.

We want to control the number of vertices and the edge density of our problem instances in order to be able to investigate the potential differences in the performance of our solution strategy with respect to these parameter changes. A considerable part of our effort is spent on applying our decomposition strategy to specific classes of graphs, although we also employ it to graphs with no particular structure. A nontrivial part of generating graphs from specific graph families is to achieve a desired number of vertices and an (approximate) edge density, while satisfying the structural properties of the class under consideration and providing a diverse range of graphs as possible.

The next four subsections introduce the methods we designed to generate permutation, generalized split, chordal, and perfect graphs.

6.1.1. Permutation Graph Generation

Let π be a permutation of n numbers, $\pi(i)$ denote the number in position i of π, and $\pi^{-1}(i)$ the position of number i in π. As mentioned in Chapter 2, a graph is a permutation graph if there is a permutation π of its n vertices in such a way that v_i and v_j are adjacent if and only if $i < j$ and $\pi^{-1}(v_i) > \pi^{-1}(v_j)$ (or equivalently $i > j$ and $\pi^{-1}(v_i) < \pi^{-1}(v_j)$). Figure 6.1 shows two permutation graphs. Note that edge $\{2, 4\}$ exists in both graphs since position number of 4 is smaller than 2 in both permutations. Similarly, there is no edge between vertices 2 and 5 in either graph since 2 is positioned earlier than 5 in both permutations.

$\pi = (3, 4, 2, 5, 1)$ $\pi = (4, 6, 2, 1, 5, 3)$

Figure 6.1. Two example permutation graphs.

To the best of our knowledge, only the study by Andreou *et al.* [2] presents methods for random permutation graph generation. The first of the two proposed methods initially sets a uniformly random permutation of numbers from one to n, and then produces the corresponding graph using the definition of permutation graphs. The authors claim that, with high probability, graphs produced by this generator have maximum degree close to n. The second method they introduce places the numbers between $n/2$ and n in the second half of the permutation with probability p and in the first half otherwise, whereas the rest of the numbers are placed in the first half of the permutation with probability p and in the second half otherwise. When p is selected as 0.5, the second algorithm reduces to the first one. It should be noted that these algorithms are not meant for obtaining general permutation graphs, but rather permutation graphs having a known bound on some parameter. In this dissertation, however, our aim is to randomly produce permutation graphs without any such restriction.

Our algorithm to generate permutation graphs, which we call PermGraphGen and use in [10], implements the definition of permutation graphs. Given n as the number of vertices, p as a probability value to manipulate the edge density of the output graph, and K as an additional means to control the edge density, the algorithm first generates a random ordering π of numbers $1, \ldots, n$. To produce a random ordering, each number $\pi(i)$ is considered to be swapped with an element at some randomly picked index later than i. If the number at the randomly chosen index is greater than $\pi(i)$, in which case a swap would lead to a rise in the number of edges, the swap takes place with

probability p; otherwise, we interchange places of the two with probability $1-p$. Thus, as p gets larger, the density of the output graph tends to get higher. This procedure is repeated K times. Finally, the algorithm makes two vertices v_i and v_j adjacent only if $i < j$ and $\pi^{-1}(i) > \pi^{-1}(j)$. Pseudo-code of this algorithm is provided in Figure 6.2.

Input: An integer n, a probability value p, and an integer K

Output: A permutation graph $G = (V, E)$ on n vertices

Let π be an array of size n with $\pi(i) = i$ for $i = 1, \ldots, n$
for $k = 1$ to K **do**
  for $i = 1$ to n **do**
    Pick a random index $r > i$
    if $\pi(r) > \pi(i)$ **then**
      Swap the two elements with probability p
    else
      Swap with probability $1 - p$
    end if
  end for
end for
Let $G = (V, E)$ be an edgeless graph with $V = \{v_1, \ldots, v_n\}$
for $i = 1$ to n **do**
  for $j = i + 1$ to n **do**
    if $\pi^{-1}(i) > \pi^{-1}(j)$ **then**
      Add an edge between v_i and v_j
    end if
  end for
end for

Figure 6.2. Algorithm PermGraphGen.

In a previous version of this algorithm, only n and p were taken as input parameters. However, in our preliminary experiments, we observed that sometimes we cannot

achieve high edge densities in one pass of swapping the elements in π, even for very large p. So, we decided to increase the flexibility of our algorithm by considering K as an additional parameter, which enables us to apply the swap procedure multiple times. We generated our test instances by performing some initial experiments and accordingly tuning the input parameters to achieve the desired edge densities. We made the instances we have generated and used available online at [49].

6.1.2. Generalized Split Graph Generation

A generalized split graph is a graph such that either itself or its complement is a unipolar graph, as mentioned in Chapter 2. Figure 6.3 depicts the two general forms of generalized split graphs, unipolar and co-unipolar.

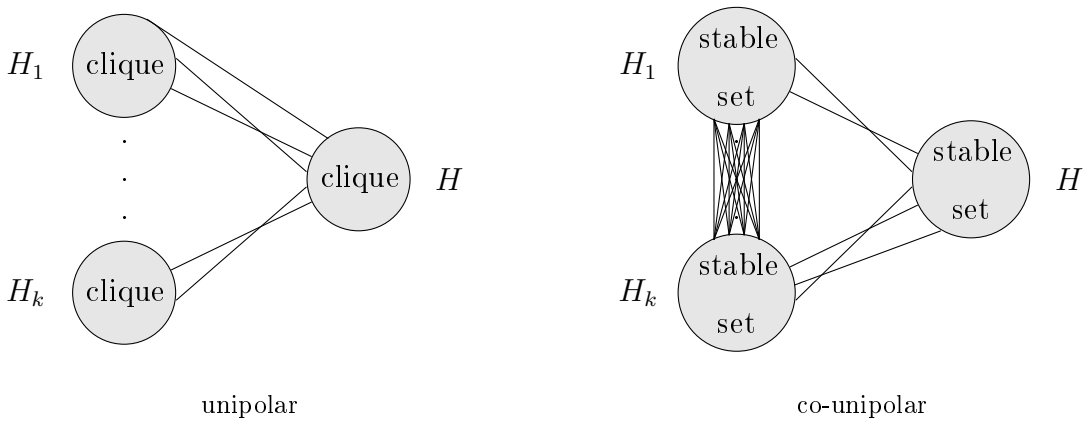


Figure 6.3. Two alternative forms of generalized split graphs.

To the best of our knowledge, there is one study on the generation of generalized split graphs in the literature. McDiarmid and Yolov [50] present an algorithm to produce generalized split graphs on n vertices almost uniformly at random, where n is an input parameter denoting the number of vertices. However, their method has no mechanism to control the edge density of the output graph. Our algorithm GSGGen, which we also present in [10], takes the number of vertices n, an upper bound u on the number of vertices in V_1, a desired edge density value ρ, and a binary parameter

Input: Two integers n and u, a real number ρ between 0 and 1, and a binary variable *unipolar*

Output: A generalized split graph G on n vertices with (approximate) edge density ρ

Create a random ordering σ of the vertex set $V = \{v_1, \ldots, v_n\}$

Let n_1 be a random integer between l and u where $l = f(n, u)$

$H \leftarrow \{\sigma_{n_1+1}, \ldots, \sigma_n\}$

Let k be a random integer set as a function of ρ and n_1

Randomly divide $\{\sigma_1, \ldots, \sigma_{n_1}\}$ into k disjoint sets $H_1, \ldots, H_k$

if *unipolar* **then**

Add edges to make $G[H]$ and $G[H_i]$ $\forall i \in \{1, \ldots k\}$ cliques each

else

Add edges to make $G\big[\bigcup_{i=1}^{k} H_i\big]$ complete k-partite

end if

Add random edges between $G[H]$ and $G[\bigcup_{i=1}^{k} H_i]$ to (approximately) achieve the desired edge density ρ

Figure 6.4. Algorithm GSGGen.

for the form of the graph to be unipolar or co-unipolar as input. In the first part, it creates a random partition of the vertex set into $H, H_1, \ldots, H_k$. To this end, it creates a random ordering σ of the vertices, and reserves the first n_1 vertices in σ for components $H_1, \ldots, H_k$, where n_1 is a random integer determined as a function of n and u. It assigns the rest of the vertices, i.e., the last $n - n_1 - 1$ vertices in σ to component H. In the second part, the algorithm adds necessary edges to make the structure of components $H, H_1, \ldots, H_k$ suitable to unipolarity or co-unipolarity. As a final step, the algorithm calculates its remaining "budget" for edge addition, and adds random edges between V_1 and $V_2 = H$ until (approximately) reaching the desired density. Given the other parameters, we tuned the value of parameter u based on our observations from the preliminary experiments we made. All the instances we have generated and used

are accessible at [49].

6.1.3. Chordal Graph Generation

A graph is called chordal if every cycle of length four or more contains a chord, where a chord is an edge between two non-consecutive vertices of a cycle. Chordal graphs can alternatively be defined as graphs not containing induced cycles of length at least four, as mentioned in Chapter 2. An example of chordal and a non-chordal graph is given in Figure 6.5.

Figure 6.5. (a) A chordal graph on 5 vertices, (b) A non-chordal graph due to the induced cycle 2-3-4-5-2.

As mentioned in Chapter 2, an intersection graph is formed from a collection of sets F by adding one vertex v_i for each $f_i \in S$ and placing an edge between two vertices v_i and v_j if the corresponding sets f_i and f_j intersect. In the special case where there is a tree T such that each set in F corresponds to the vertex set of a subtree of T, then G is called the *intersection graph of subtrees of a tree*. An ordering $(v_1, v_2, \ldots, v_n)$ of the vertices of a graph is a *perfect elimination order (PEO)* if the set of higher numbered neighbors of each vertex forms a clique. Let K be the set of maximal cliques of a graph G. A tree T with a bijection between its vertex set and the cliques in K is called a *clique tree* of G if for every vertex v of G, the set of vertices of T that correspond to the cliques containing v induce a connected subtree of T. A chordal graph on n vertices has at most n maximal cliques [51].

Chordal graphs have many different characterizations. For our purposes, the following will be sufficient.

Theorem 6.1 ([36, 52–54]). *Let G be a graph. The following are equivalent.*

(i) *G is chordal.*
(ii) *G has a perfect elimination order.*
(iii) *G is the intersection graph of subtrees of a tree.*
(iv) *G has a clique tree.*

Especially the last two points of Theorem 6.1 are crucial for our algorithm and its implementation. Our algorithm is based on the characterization that a graph is chordal if and only if it is the intersection graph of subtrees of a tree [54]. To the best of our knowledge, this characterization does not seem to have been directly used for random chordal graph generation earlier.

Rose *et al.* [55] gave an algorithm called Maximal Cardinality Search (MCS) that creates a perfect elimination order of a chordal graph in time $O(n + m)$. Blair and Peyton [37] gave a modification of MCS to list all the maximal cliques of a chordal graph in time $O(n+m)$. Implicit in their proofs is the following well-known fact, which can be characterized as folklore.

Lemma 6.2 ([37, 55]). *The sum of the sizes of the maximal cliques of a chordal graph is $O(n + m)$.*

Next, we briefly mention the algorithms for generating chordal graphs from the works of Andreou *et al.* [2], Pemmaraju *et al.* [56], and Markenzon *et al.* [57]. Some of these algorithms create very limited chordal graphs, which is either mentioned by the authors or clear from the algorithm. Thus, in the following we only mention the algorithms that are general enough to be interesting in our context.

It should be noted that the purpose of Andreou *et al.* [2] is not to obtain general chordal graphs, but rather chordal graphs with a known bound on some parameter. One of the algorithms they propose starts from an arbitrary graph and adds edges to obtain a chordal graph. The way the edges are added is not given in detail; however, it should be noted that there are many algorithms for generating a chordal graph from a given graph by adding a minimal set of edges and their running time is usually $O(nm)$ [58]. Andreou *et al.* [2] do not report on the quality of chordal graphs obtained by this method.

We highlight below the algorithms that are the most promising with respect to generating random chordal graphs. In addition to these, there is an $O(n^2)$-time algorithm by Markenzon *et al.* [57] that generates a random tree and adds edges to this tree until a chordal graph with desired edge density is obtained. However, no test results about the quality of the generated graphs are given.

Alg 1 [2]. The algorithm constructs a chordal graph by using a PEO. It considers the vertices in the reverse order they will appear in the PEO. At every iteration, a new vertex v is added and made adjacent to a random selection of already existing vertices, where existing vertices are in fact the vertices that come later than v in the PEO. Then necessary edges are added to turn the neighborhood of the new vertex into a clique such that a given maximum degree is not exceeded. No test results are given in the paper about the quality of the chordal graphs this algorithm produces. We implemented this algorithm without imposing a limit on the maximum degree of the output graph, because no detail was given about how the method avoids exceeding a given maximum degree. We compare the resulting graphs to those generated by our algorithm in Section 6.1.3.4.

Alg 2 [56, 57]. The algorithm starts from a single vertex. At each subsequent step, a clique C in the existing graph is chosen at random, and a new vertex is added and made adjacent to the vertices of C. The inverse of the order in which the vertices are added is a PEO of the final graph. It is observed by the authors of both papers that

this procedure results in chordal graphs with approximately $2n$ edges experimentally. They propose the following modifications:

Alg 2a [57] modifies the above generated graph by randomly choosing maximal cliques that are adjacent according to the clique tree and merging these until the desired edge density is obtained. Some test results about graphs generated by Alg 2a are provided in [57]. Although these tests are not as comprehensive as the ones we give on our algorithms in Section 6.1.3.4, we compare our results to those of [57] as best we can. The running time of Alg 2a is $O(m + n\alpha(2n, n))$.

Alg 2b [56] is a modification of Alg 2 in a different way: instead of randomly choosing a clique, a maximum clique is chosen and a random subset of it is made adjacent to the new vertex. Although test results for Alg 2b are provided in [56], the authors acknowledge that the produced graphs still have very few large maximal cliques and many very small maximal cliques. For this reason, we do not include Alg 2b in our comparisons.

To the best of our knowledge, the intersection graph of subtrees of a tree characterization of chordal graphs has not been used directly for their generation. Indeed, since all characterizations of a chordal graph are equivalent, even the existing algorithms mentioned above could be interpreted as based on any of these characterizations. Especially the algorithms based on clique trees can easily be translated to generate subtrees of a tree. However, none of these algorithms actually generates random subtrees of a randomly generated tree to produce the resulting chordal graph. One reason could be that this does not give a direct way to decide the number of edges in the generated graph. We will see that the edge density can be regulated by adjusting the sizes of the generated subtrees.

Our main algorithm for generation of chordal graphs on n vertices, which we also present in [19, 20], is given in Figure 6.6. The algorithm starts by constructing a random host tree. Then it generates subtrees on the host tree, and finally outputs a

chordal graph as the intersection graph of these subtrees.

Input: n and one or two other numbers to guide the number of resulting edges

Output: A chordal graph G on n vertices

1. Generate a tree T on n nodes uniformly at random
2. Generate n non-empty subtrees of $T : \{T_1, \ldots, T_n\}$
3. Output as G the intersection graph of $\{V(T_1), \ldots, V(T_n)\}$

Figure 6.6. Algorithm ChordalGen.

By Theorem 6.1, any graph generated by Algorithm ChordalGen is chordal. For the generation of a random host tree T on n nodes, we use the method by Rodionov and Choo [59], which can easily be implemented in $O(n)$ time. The method starts with a tree T that contains only one node. Then, at each step, it selects a random node x of T and adds a new node adjacent to it, until T has n nodes. This algorithm creates any one of n-node trees (isomorphic ones included) with equal probability, in particular with probability $\frac{1}{n^{n-1}}$ [59]. In the following subsections, we describe three different methods for generating the n subtrees, each of which is meant for giving different neighborhood and density properties in the output graph. Each method will take one or two parameters to guide the average size of the generated subtrees, with the purpose of controlling the resulting number of edges in G. Our algorithm is flexible in the sense that additional ways to generate the subtrees can be suggested and tested later.

We know from [54] that the intersection graph of any set of subtrees of a tree gives a chordal graph, and for any chordal graph G we can find a tree and a set of subtrees on it such that their intersection graph yields G. It should be noted that Algorithm ChordalGen fixes the size of the host tree and the number of subtrees on it as n, where n is an integer input to the program to be the size of the vertex set of the output graph. Lemma 6.3 shows that any chordal graph can be generated this way. To make sure that there is no confusion between the vertices of G and the vertices of

a tree or a clique tree, we will refer to vertices of a tree as *nodes* from now on.

Lemma 6.3. *For every chordal graph G on n vertices, there is a tree T on n nodes, and a set of n subtrees of T, such that G is the intersection graph of these subtrees.*

Proof. Let G be a chordal graph on n vertices and let T' be a clique tree of G. Let us call the vertices of G: $v_1, v_2, \ldots, v_n$. Define subtree T'_i to be the subtree of T' that corresponds to the nodes (maximal cliques) that contain vertex v_i, for $1 \leq i \leq n$. By the definition of a clique tree, T' has at most n nodes and each T'_i is a connected subgraph of T'. If T' has fewer than n nodes, we can add new nodes adjacent to arbitrary nodes of T' until we get a new tree T with n nodes. The subtrees stay the same. As two vertices are adjacent in G if and only if they appear together in a clique, G is the intersection graph of subtrees $T'_1, \ldots, T'_n$ of T. □

The graph isomorphism problem is the problem of deciding whether there is a bijective mapping (a permutation) from the vertices of one graph to the vertices of the second graph such that the edge connections are preserved. The problem is still unsolved in the sense that neither a polynomial-time algorithm has been found yet nor it has been proven to be NP-complete [60]. However, it is shown that the problem is as hard on chordal graphs as on general graphs [61], which adds to the difficulty of producing chordal graphs uniformly at random. Since the random tree generation method by Rodionov and Choo [59] can produce any tree on n nodes, and any set of n subtrees can be generated with positive probability with the subtree generation methods we present here, as shown by Claim 6.6 – Claim 6.12 in Section 6.1.3.1 – Section 6.1.3.3, Algorithm ChordalGen is able to generate every chordal graph with a strictly positive probability by Lemma 6.3.

We need to evaluate the order of $\sum_{i=1}^{n} |V(T_i)|$ to analyze the running time of Algorithm ChordalGen. In this dissertation, we present a lower bound. $\sum_{i=1}^{n} |V(T_i)|$ is $\Omega(mn^{1/4})$ as shown in what follows.

Lemma 6.4. *Let T be a tree on n nodes, and let $T_1, \ldots, T_n$ be n subtrees of T whose intersection graph is a chordal graph on n vertices and m edges. Then $\sum_{i=1}^{n} |V(T_i)|$ is $\Omega(mn^{1/4})$.*

Proof. Let G be a chordal graph that is the intersection graph of $\{V(T_1), \ldots, V(T_n)\}$, such that every subtree T_i is associated with a vertex v_i of G. Choose an integer $p \geq 2$. Let $n = p^4, n' = p^2 + 1, s = p^2 - p$, and $s_{n'} = p^3$. Note that $n > n'$. Let T be a path on n nodes $x_1, \ldots x_n$. Let T_i be the subtree of T consisting of the single node x_i, for $i = 1, \ldots, n' - 1$. Let $T_{n'}$ be the subpath of T consisting of the nodes from n' to n. The representation contains s copies of T_i for every $i = 1, \ldots, n' - 1$ and also $s_{n'}$ copies of $T_{n'}$. Therefore, the number of subtrees is

$$s(n' - 1) + s_{n'} = (p^2 - p)p^2 + p^3 = p^4 = n.$$

The corresponding graph is the disjoint union of a $K_{s_{n'}}$ and $n' - 1$ copies of K_s. The number m of edges of this graph is

$$m = (n' - 1)\frac{s(s-1)}{2} + \frac{s_{n'}(s_{n'} - 1)}{2} = p^2\frac{(p^2 - p)(p^2 - p - 1)}{2} + \frac{p^3(p^3 - 1)}{2} = p^6 - p^5.$$

On the other hand, the total size of the subtrees is

$$s(n' - 1) + s_{n'}(n - n' + 1) = (p^2 - p)p^2 + p^3(p^4 - p^2) = p^7 - p^5 + p^4 - p^3 \geq p^7 - p^5.$$

Therefore,

$$\frac{\sum_{i=1}^{n} |V(T_i)|}{m} \geq \frac{p^7 - p^5}{p^6 - p^5} = p + 1 > n^{1/4}.$$

Since the inequality holds for infinitely many values of p, we conclude the proof. □

Now, let us state the main result about Algorithm ChordalGen.

Theorem 6.5. *Algorithm ChordalGen generates a chordal graph with n vertices in time $O(S + \sum_{i=1}^{n} |V(T_i)|)$, where $O(S)$ is the time required to generate n subtrees of a host tree on n nodes.*

Proof. By Theorem 6.1, graph G generated by Algorithm ChordalGen is chordal. Clearly G has n vertices. Let m be the number of edges of G. Let us go through the steps of the algorithm to establish the running time.

Step 1. For the generation of a random host tree T on n vertices, we use the following method by Rodionov and Choo [59], which can easily be implemented in $O(n)$ time: start with a tree T that contains only one node. Then repeat $n-1$ times the following: pick a random vertex x of T and add a new vertex adjacent to it.

Step 2. We generate n subtrees $T_1, \ldots, T_n$ of T using a subtree generator. According to the premises of the theorem, this adds $O(S)$ to the overall time.

Step 3. The sum of the sizes of the generated subtrees is $\sum_{i=1}^{n} |V(T_i)|$. Let us now explain how we can obtain in time $O(n+m)$ the chordal graph that is the intersection graph of these subtrees. For every node x of T, let us define the following list: $C_x = \{v_j \mid T_j \text{ contains } x\}$, i.e., vertices of G whose corresponding subtrees contain node x of T. Let $\mathcal{C} = \{C_x \mid x \in V(T)\}$. Observe that every set in $\mathcal{C}$ is a clique of G, and $\mathcal{C}$ contains all maximal cliques of G. However, some of the cliques in $\mathcal{C}$ may not be maximal. If we blow up every node x of T to the set C_x, we get a tree which is almost a clique-tree of G; this procedure can clearly be done in $O(n+m)$ time. In the resulting tree, non-maximal cliques simply need to be deleted or merged into the maximal ones. By the methods described by Blair and Peyton [37] it is possible to turn T into a proper clique tree for G in time $O(n+m)$. Thus, in total $O(\sum_{i=1}^{n} |V(T_i)| + n + m)$ time we both have a representation of our output graph G and a list of maximal cliques of it. It could, however, be desirable to output an adjacency list representation for G. Markenzon *et al.* [57], using the methods of Blair and Peyton [37], explain how this can be done in $O(n+m)$ time. By Lemma 6.4, the overall running time of this step is

$O(\sum_{i=1}^{n} |V(T_i)|)$. □

Now we are ready to present the details of how the subtrees in the second step of Algorithm ChordalGen are generated. In Section 6.1.3.1 – Section 6.1.3.3, we describe the three subtree generation methods we use. Then in Section 6.1.3.4, we test our algorithm with these methods and compare the results with each other, as well as with Alg 1 and Alg 2a.

6.1.3.1. Algorithm GrowingSubtree. This algorithm takes as input a tree T on n nodes, and an integer k, and generates n subtrees of T of average size $\frac{k+1}{2}$. In our test results, we give both k and the resulting number of edges, m, to give an indication of how k affects the density of the generated graph.

For each subtree T_i, the algorithm picks a size k_i randomly from $[1, k]$. Then a random node of T is chosen as the single node of T_i to start with. In each of the subsequent $k_i - 1$ iterations, we pick a random node of T in the neighborhood of T_i and add it to T_i. Pseudo-code of this algorithm is provided in Figure 6.7.

Input: A tree T on n nodes and a positive integer $k \leq n$
Output: A set of n non-empty subtrees of T of average size $\frac{k+1}{2}$: $\{T_1, \ldots, T_n\}$

for $i = 1$ to n **do**
  Select a random node x of T and set $T_i = \{x\}$
  Select a random integer k_i between 1 and k
  for $j = 1$ to $k_i - 1$ **do**
    Select a random node $y \in T_i$ that has neighbors in T outside of T_i
    Select a random neighbor z of y outside of T_i and set $T_i = T_i \cup \{z\}$
  end for
end for

Figure 6.7. Algorithm GrowingSubtree.

Next, we show that for a given tree T, Algorithm GrowingSubtree can produce any set of n subtrees on T.

Claim 6.6. *Given a tree T on n nodes and a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of n subtrees on it, there exists a positive integer parameter $k \leq n$ such that Algorithm GrowingSubtree produces $\mathcal{S}$ with strictly positive probability.*

Proof. Suppose that we are at i^{th} iteration of Algorithm GrowingSubtree to produce subtree T_i, and let $S_i \in \mathcal{S}$ be a given subtree on T having k_i nodes. Furthermore, let $k = n$ to allow any possible subtree size from 1 to n. We will show that at any given iteration i, the probability that T_i is exactly S_i is strictly positive. The probability of selecting a random subtree size equal to k_i is $\frac{1}{k} > 0$. Since the algorithm picks the initial node of any subtree uniformly at random, the probability of selecting the initial node from S_i is $p_1 = \frac{k_i}{n} > 0$. If $k_i = 1$, then we have shown that at any given step our algorithm can produce S_i with nonzero probability (in particular, with probability $\frac{k_i}{n} \cdot \frac{1}{k} = \frac{k_i}{n^2} = \frac{1}{n^2} > 0$). So assume $k_i > 1$, and let x_j be the j^{th} node added to T_i. At this point, it is important to note that the algorithm grows the subtree by randomly picking one node from the neighborhood of the current subtree. Then, given $x_1 \in S_i$ as the initial node of T_i, the probability of selecting the second node x_2 from S_i is proportional to $p_2 = \frac{|N_{S_i}(x_1)|}{|N(x_1)|} > 0$, where $N_{S_i}(x_1)$ denotes the set of neighbors of x_1 that are in S_i, which is a subset of $N(x_1)$ and is guaranteed to be nonempty because otherwise S_i would consist of a single node contradicting our assumption that $k_i > 1$. To generalize, given $\{x_1, \ldots, x_{j-1}\} \in S_i$ as the first $j-1$ nodes of T_i, the probability of selecting the j^{th} node x_j from S_i is proportional to $p_j = \frac{|N_{S_i}(\{x_1,\ldots,x_{j-1}\})|}{|N(\{x_1,\ldots,x_{j-1}\})|} > 0$, assuming that $k_i > j-1$. Then, if we choose $k = n$, at any iteration, the algorithm can generate a given subtree S_i with a probability proportional to $\frac{1}{k}\prod_1^{k_i} p_i$, which is strictly positive because each term in the finite product is strictly positive.

Now suppose that we are given a chordal graph G together with a set of n subtrees $\mathcal{S} = \{S_1, \ldots, S_n\}$ on a tree T whose intersection graph gives G. By the above arguments, we know that any element of $\mathcal{S}$ can be generated with a nonzero probability at some given iteration of the algorithm. The number of permutations

of an n-set being $n!$, given T and $k = n$, the probability of producing the set $\mathcal{S}$ is proportional to $\frac{n!}{k} \prod_1^{k_i} p_i$, which is strictly positive. □

Next, we present a running time analysis of Algorithm GrowingSubtree.

Lemma 6.7. *The running time of Algorithm GrowingSubtree is* $O(\sum_{i=1}^{n} |V(T_i)|)$.

Proof. Observe first that each subtree T_i can be represented simply a list of nodes of T. We show that after an initial $O(n)$ preprocessing time, each subtree T_i can be generated in time $O(|V(T_i)|)$. For this, we need to be able to add a new node to T_i in constant time, at each of the $k_i - 1$ steps.

As selecting random elements in constant time is easier when accessing the elements of an array directly by indices, we start with copying the nodes of T into an array A of size n, and copying the adjacency list of each node x into an array A_x of size $d(x)$. This can clearly be done in total time $O(n)$ since T is a tree.

In general, selecting an unselected element of a set at random can be done easily in constant time if the set is represented with an array. Let us say we have an array S of t elements. We keep a separation index s that separates the selected elements from the not selected ones. At the beginning s is 1. At each step, we generate a random integer r between s and t. $S[r]$ is our randomly selected element. Then we swap the elements $S[s]$ and $S[r]$ and increase s by 1.

We can use this method both for selecting a node y of T_i that still has neighbors outside and for selecting a neighbor z of y that has not yet been selected. For the latter, whenever we select a neighbor z of y, we move z to the first part of the array A_x using swap. When the separation index reaches the degree of y then we know that y should not be selected to grow the subtree T_i at later steps. Representing T_i with an array of size k_i, we can use the same trick to move y to a part of the array that we will not select from. Also, when z is added, we can check whether it is a leaf in

T in constant time, and immediately move it to the irrelevant part of the array for T_i if so, since z can then not be used for growing T_i at later steps. It is sufficient to check that z is a leaf of T, because otherwise it must have neighbors outside of T_i, since T is a tree and we cannot have cycles. When the generation of T_i is finished, the separation indices of each of its nodes should be reset before we start generating T_{i+1}. The adjacency arrays need not be reorganized, as we will anyway be selecting neighbors at random. Note that we do not need this trick to select an initial node x of each subtree T_i, because we should indeed be able to select the same node several times (and grow another subtree from it perhaps in a different way). With the described method, each step of Algorithm GrowingSubtree takes $O(1)$ time, in addition to initial $O(n)$ time to copy the information into appropriate arrays. Thus the total running time is $O(\sum_{i=1}^{n} |V(T_i)|)$. □

Lemma 6.7 together with Theorem 6.5 gives the following.

Corollary 6.8. *Algorithm ChordalGen, using the GrowingSubtree method, runs in time* $O(\sum_{i=1}^{n} |V(T_i)|)$.

We will see in Section 6.1.3.4 that this method generates chordal graphs with the most even distribution of maximal clique sizes.

6.1.3.2. Algorithm ConnectingNodes. This algorithm takes as input a tree T on n nodes, and a positive real number λ. The purpose of the parameter λ is to guide the desired number of resulting edges in the graph. To generate each subtree T_i, we first select k_i nodes of T, where k_i is a random integer generated by making use of λ. T_i is then generated to be the minimal subtree that contains the selected k_i nodes. In other words, T_i is created by taking the union of unique paths that connect the selected set of k_i nodes. This implies that a subtree will most likely have many more nodes than those selected initially, and this must be taken into consideration when choosing λ. In our test results presented in Section 6.1.3.4, we give both λ and the resulting number of edges, m, to give an indication of how λ affects the density of the generated graph.

In the following algorithm, we will make use of the standard Breadth First Search (BFS) algorithm from an arbitrary node r of T. We will then treat T as a rooted tree with root r, and speak about parent-child relation in the standard way, with respect to root r. The BFS level of a vertex is simply the distance of that vertex from r.

For each subtree T_i, we first generate a random integer k_i by making use of Poisson distribution with mean λ. Poisson distribution is a discrete probability distribution widely used to model the number of occurrences of an event over a specified domain such as time, space etc. In our case, the domain is the host tree, and an event is selection of a node from the host tree. The parameter of this distribution is λ and it is the average rate of event occurrences, which implies that k_i values will tend to increase on the average as λ increases. The set of possible values a Poisson random variable can take is nonnegative integers, regardless of the value of λ. However, the minimum and maximum number of nodes that a subtree of an n-node host tree can contain are 1 and n respectively. Therefore, we equate k_i to 1 if it is zero, and to n if it is greater than n. In this method, we make use of Poisson distribution because we were not able to achieve a good precision for edge density by picking a random integer uniformly at random from a given interval.

To generate the minimal subtree that contains the k_i selected nodes, we make use of the nodes' parent and level information retrieved during BFS. Our key observation is that the minimal subtree must contain the parents of all selected nodes at some level d if there are other selected nodes at levels less than or equal to d. Using this idea, we add a node to T_i only when an edge incident to it has to be in T_i to join a node to the others. We start from the highest level (highest distance from r) and proceed by moving toward the root until all nodes in the selection become connected. The set L keeps the unprocessed nodes yet to be connected to form the subtree T_i. At each step, we consider the nodes in L that are at the same level, which are going to be joined as we move through the levels. Once the parents of those nodes are identified, we are done with level d and there is no need to reconsider nodes at level d any further. Therefore, we replace those nodes with their parents in L, which are to be considered at the next

Input: A tree T on n nodes and a positive real number λ

Output: A set of n subtrees of T: $\{T_1, \ldots, T_n\}$

Let r be an arbitrary node of T

Perform BFS from r, and identify the parent $P(x)$ and the BFS level $l(x)$ of each node x

$L \leftarrow \emptyset$

for $i = 1$ to n **do**

  $T_i \leftarrow \emptyset$

  Select a random integer k_i from Poisson distribution with mean λ

  if $k_i = 0$ **then**

    $k_i \leftarrow 1$

  else

    if $k_i > n$ **then**

      $k_i \leftarrow n$

    end if

  end if

  Pick k_i random nodes from T to form $T_i = \{x_1, \ldots, x_{k_i}\}$ and $L = \{x_1, \ldots, x_{k_i}\}$

  $d \leftarrow \max_{x \in T_i} l(x)$

  while $|L| > 1$ **do**

    for all $x \in L$ such that $l(x) = d$ **do**

      $T_i \leftarrow T_i \cup P(x)$

      $L \leftarrow (L \setminus \{x\}) \cup P(x)$

    end for

    $d \leftarrow d - 1$

  end while

end for

Figure 6.8. Algorithm ConnectingNodes.

step. Parent nodes are also added to T_i because they lie on the paths that connect the node selection. Afterwards, we move to the next level and apply the same procedure. This process continues until a single node is left in L, which simply means that we have a node set that has been linked at a single node already and that T_i includes all nodes of the subtree that minimally connects the randomly selected node set. Pseudo-code of this algorithm is provided in Figure 6.8.

Now, we will show that given a tree T, this algorithm is able to produce any n-set of subtrees on T.

Claim 6.9. *Given a tree T on n nodes and a set $\mathcal{S} = \{S_1, \ldots, S_n\}$ of n subtrees on it, Algorithm ConnectingNodes produces $\mathcal{S}$ with strictly positive probability.*

Proof. We first note that, at any iteration i, k_i can take any value in $\{1, \ldots, n\}$ with nonzero probability, simply because Poisson distribution can produce any nonnegative integer with nonzero probability regardless of the value of parameter λ, and values outside the desired domain $\{1, \ldots, n\}$ are taken as one or n. Suppose that we are at i^{th} iteration to produce subtree T_i, and let $S_i \in \mathcal{S}$ be a given subtree of T having n_i nodes. We will show that, at any given step, the probability that T_i is exactly S_i is strictly positive. First, note that the set of nodes whose minimal connection will result in some subtree S_i is not unique in general. Furthermore, for a given k_i, any subset of k_i nodes of T can be selected with strictly positive probability, because each node is selected with a strictly positive probability. Therefore, there is a nonzero probability that the algorithm selects a set of nodes on T whose minimal connection is exactly S_i. Then, if we are given a chordal graph G together with a set of n subtrees $\mathcal{S} = \{S_1, \ldots, S_n\}$ on a tree T whose intersection graph gives G, we know that any element of $\mathcal{S}$ can be generated with a nonzero probability at some given iteration of the algorithm, which implies that the probability of producing the set $\mathcal{S}$ is strictly positive. □

We now give a running time analysis of Algorithm ConnectingNodes.

Lemma 6.10. *The running time of Algorithm ConnectingNodes is* $O(\sum_{i=1}^{n} |V(T_i)|)$.

Proof. Rooting T from an arbitrary node, and determining the parent $P(x)$ of each node x in T as well as its level $l(x)$ with respect to the root node, takes $O(n)$ time in total for all nodes using BFS. The set L is represented by an array of lists with length equal to $\max_{x \in T} l(x)$. Each index d of L represents a list of unprocessed nodes having $l(x) = d$. The lists in L will be empty initially. If a node x at level d is selected, we add x to the list at index d of L in constant time. We also need to keep an n-dimensional boolean array B, which will be comprised of zeros at first, in order to check whether a node already exists in L. If a node x is chosen, the element at index x of B will be set to one in constant time. Note that B is a different representation of the set of nodes in L. Both L and B are initialized only once at the start of the algorithm after performing the BFS. Since the number of levels is at most n, initialization of L and B can be done in time $O(n)$. We represent T_i as a list of nodes, which can be initialized in $O(1)$. For each subtree, we generate a random integer k_i by making use of Poisson distribution with mean λ. Generation of some random integer x from Poisson distribution normally takes $O(x)$ time; starting from zero, the value of x is incremented one by one until the stopping condition is met [62]. However, since we need k_i, which is the number nodes to be selected, to lie between 1 and n, we terminate the process if we reach n before stopping condition is met, and we set k_i to 1 if the process returns a zero value. This way, we only spend $O(k_i)$ time to generate k_i. Then, at each iteration, k_i random nodes are chosen. To do this in time $O(k_i)$, we can copy the nodes of T into an array, which is done at the beginning only once and hence takes $O(n)$ time in total, and keep a separation index s that separates the selected elements from the ones that have not been selected yet, as explained in the proof of Lemma 6.7. Adding a chosen node to T_i, L and B can be done in $O(1)$. At the end of n such iterations, a total of $\sum_{i=1}^{n} k_i$ random choices are made and this is clearly less than or equal to $\sum_{i=1}^{n} |V(T_i)|$.

Now, it remains to show that generation of n subtrees can also be done in time $O(\sum_{i=1}^{n} |V(T_i)|)$ once T_i, L and B are populated with initial randomly selected nodes. Our aim is to construct the minimal subtree of T connecting all the nodes in

$\{x_1, \ldots, x_{k_i}\}$. To this end, at every iteration, we add the parents of all nodes of highest level to subtree T_i and replace these nodes by their parents in L. The way we store the nodes in L enables us to access unprocessed nodes at a given level in constant time. However, to be able to start with the highest level in constant time initially, we need to know the highest level of the randomly selected k_i nodes, which can be found in $O(k_i)$. While processing some node x at level d, we first investigate whether its parent node $P(x)$ has already been included in L by checking index $P(x)$ of B in constant time. If it is one, it means that the parent node has already been included in L and T_i, and we do not do anything. Otherwise, we append $P(x)$ to the list at index $(d-1)$ of L and set the corresponding index of B to one. When done with x, we remove it from L in $O(1)$ since x is an element of a list, and set index x of B to zero, which is again $O(1)$. Recall that since T_i is represented as a list, adding an element to T_i can be done in $O(1)$. Thus, we perform constant-time operations for each node under consideration in the inner for loop.

At the beginning of the while loop L has k_i isolated nodes. Whenever two nodes in L have a common parent, the cardinality of L decreases by one at the next step. Noting that $|L|$ indicates the number of currently existing connected components, which are to be attached together to reveal the subtree, the while loop to add new nodes to T_i terminates when $|L| = 1$; that is, as soon as the minimal subtree has been found. Now, it is enough to notice that during the generation of each subtree T_i using this method, we only consider and add the nodes of T which are in T_i, and iterate only through the levels that are contained in T_i. In other words, $|L|$ becomes 1 and the while loop stops after exactly when $|V(T_i)|$ nodes are considered. Because we spend constant time for each of the $|V(T_i)|$ nodes, the overall complexity of the operations within the while loop becomes $O(|V(T_i)|)$ for each subtree T_i. In order to obtain $O(|V(T_i)|)$ for the entire loop, we need to ensure that termination condition of the while loop can be checked in constant time. For this purpose, we keep the number of nodes in L as a variable, incrementing whenever a new node is added and decrementing upon removal of a node, which takes at most $O(|V(T_i)|)$ time. Finally, the arrays L and B should be reset before being passed to the next subtree. We know that L will contain a single node at the end

of the while loop, and equivalently a single nonzero element will exist in B. When the loop terminates, we will know at which index (level) of L we were finally at. So, we can simply access the final node, set the element in B corresponding to that node to zero, and delete it from L, all in constant time. This way, the algorithm from while loop on to the next subtree completes in time $O(|V(T_i)|)$. In total, these operations add to the running time of Algorithm ConnectingNodes a term of $O(\sum_{i=1}^{n} |V(T_i)|)$. □

Lemma 6.10, together with Theorem 6.5, gives the following:

Corollary 6.11. *Algorithm ChordalGen, using the ConnectingNode method, runs in time* $O(\sum_{i=1}^{n} |V(T_i)|)$.

6.1.3.3. Algorithm PrunedTree. The input to this algorithm consists of a tree T on n nodes, and an edge deletion fraction f and a selection barrier s that are real numbers between zero and one. To generate subtree T_i, we randomly select a fraction f of the edges on the tree and remove them. The number of edges to delete, say l, is calculated as $\lfloor (n-1)f \rfloor$, which will leave $l+1$ subtrees in total. We then determine the sizes of the $l+1$ subtrees and store the distinct values. We pick a random size k_i from the set of largest $100(1-s)\%$ of distinct values, and randomly choose a subtree with size k_i. We repeat this n times to generate all the subtrees.

One should note that we could simply select one connected component (subtree) at random without any preferential treatment; however, parameter s makes it easier to increase the density of the chordal graph by favoring larger components as the value of s advises. So, parameter s is an additional means to tune the edge density of the chordal graph; as its value increases, the size of the subtree to be selected tends to increase too. Increasing the edge deletion fraction f, however, tends to decrease the average size of subtrees emerging from deletion of edges. Pseudo-code of this algorithm is presented in Figure 6.9. We show in the sequel that this algorithm, like the previous two, can generate any n-set of subtrees on a given tree T.

Input: A tree T on n nodes, edge deletion fraction f, and selection barrier s

Output: A set of n non-empty subtrees of T: $\{T_1, \ldots, T_n\}$

for $i = 1$ to n **do**

Create a copy T' of T

Select randomly $\lfloor (n-1)f \rfloor$ edges of T' and delete them from T'

Determine connected components of T' and their sizes

Select randomly a subtree size k_i from the highest $100(1-s)\%$ subtree sizes

Select a random component of size k_i and choose it as T_i

end for

Figure 6.9. Algorithm PrunedTree.

Claim 6.12. *Given a tree T on n nodes and a set $\mathcal{S}$ of n subtrees on it, there exist real numbers f and s between zero and one such that Algorithm PrunedTree produces $\mathcal{S}$ with a strictly positive probability.*

Proof. Suppose that we are at i^{th} iteration of the algorithm to produce subtree T_i. Let $S_i \in \mathcal{S}$ be a given subtree of T on n_i nodes and E_i be the set of edges that disconnects S_i from T. Since the algorithm considers each edge for deletion with a nonzero probability, any subset of $\lfloor (n-1)f \rfloor$ edges can be selected with strictly positive probability. If we select f to be equal to $\frac{|E_i|}{n-1}$, then we ensure that there is a nonzero probability that all edges in E_i are deleted (note here that there may well exist $f > \frac{|E_i|}{n-1}$ for which there is a positive probability that all edges in E_i are deleted and those in $E(S_i)$ remain intact). Note that E_i is nonempty unless S_i is T. If $E_i = \emptyset$, then setting $f = 0$ will give us the desired subtree as T regardless of the value of parameter s. Otherwise, by choosing s to be zero, selection of any connected component of T' may occur with positive probability, which, together with the previous observations, imply that at a given iteration, the algorithm can produce any subtree with nonzero probability using suitably chosen parameter values. Having this, if we are given a chordal graph G together with a set of n subtrees $\mathcal{S} = \{S_1, \ldots, S_n\}$ on a tree T whose intersection

graph gives G, we conclude that the probability of producing the set $\mathcal{S}$ altogether is strictly positive. □

Now we give a running time analysis of Algorithm PrunedTree.

Lemma 6.13. *The running time of Algorithm PrunedTree is* $O(n^2)$.

Proof. Creating a copy of T, deleting a subset of its edges, and computing the resulting connected components takes $O(n)$ time by standard BFS. Now, we create an array A of size n, where each element in A is a linked list. For each connected component of T' of size t, we add this component at the end of the list in $A[t]$. Clearly, initializing A, and adding all subtrees to appropriate lists takes $O(n)$ time. We also make an additional array B which simply stores the sizes of all subtrees, in sorted order. B can be created in time $O(n)$, using A. We use B to find the highest $100(1-s)\%$ subtree sizes, by simply using the corresponding last portion of B. Random selection of a subtree of size k_i is simply done by picking a subtree from the list $A[k_i]$ in $O(1)$ time. Thus every subtree requires $O(n)$ time to generate. Repeating this n times, the overall complexity of PrunedTree algorithm amounts to $O(n^2)$. □

We now obtain the following result using Theorem 6.5 together with Lemma 6.13.

Corollary 6.14. *Algorithm ChordalGen, using the PrunedTree method, runs in time* $O(n^2 + \sum_{i=1}^{n} |V(T_i)|)$.

6.1.3.4. Comparison. In this section, we give extensive test results to show what kind of chordal graphs are generated by Algorithm ChordalGen. In Tables 6.1-6.3 we give the experimental results of our presented methods GrowingSubtree, ConnectingNodes and PrunedTree, respectively. We show how the selection of the input parameters affects the number of resulting edges m and connected components ("# conn. comp.s"). We also present the number of maximal cliques ("# maximal cliques"), and the minimum, maximum, and mean size for the maximal cliques ("Min clique size", "Max clique size",

"Mean clique size"), along with their standard deviation ("Sd of clique sizes"). For each parameter combination, we performed ten independent runs and report the average values across those ten runs. For each n, we tuned the parameter values in order to approximately achieve some selected average edge density values of 0.01, 0.1, 0.5, and 0.8, where edge density is defined as $\frac{m}{n(n-1)/2}$. All instances on 1000 to 10000 vertices that we present here together with an additional broad collection of relatively small-sized chordal graphs on 50 to 500 vertices with varying edge densities are available at [63]. The chordal graph instances that we have used to test our decomposition approach are available at [49].

Table 6.1. Experimental results of Algorithm ChordalGen with GrowingSubtree method.

n	Max subtree size (k)	Density	m	# conn. comp.s	# maximal cliques	Min clique size	Max clique size	Mean clique size	Sd of clique sizes
1000	7	0.011	5551.4	16.7	357.1	1.0	21.6	6.1	3.4
	33	0.104	51768.5	1.0	173.0	4.8	141.5	30.7	20.4
	139	0.497	248033.5	1.0	81.3	30.6	474.3	137.9	89.2
	324	0.803	400918.7	1.0	47.5	66.8	717.4	312.2	159.5
2500	13	0.011	34605.4	2.7	678.3	1.2	54.5	11.5	6.8
	63	0.104	326287.0	1.0	300.4	9.7	349.9	61.7	45.0
	269	0.505	1577474.1	1.0	134.3	50.2	1177.4	292.8	207.7
	635	0.806	2518595.5	1.0	83.3	132.6	1861.8	673.1	397.4
5000	20	0.010	129763.8	1.6	1104.6	1.6	96.5	18.1	11.4
	100	0.104	1296493.4	1.0	474.0	15.9	695.0	103.0	80.0
	450	0.498	6226843.9	1.0	205.7	74.8	2390.8	501.6	374.2
	1097	0.804	10053952.1	1.0	124.1	202.0	3656.7	1220.6	741.9
10000	31	0.010	502155.4	1.0	1754.4	3.4	199.2	29.1	19.8
	169	0.107	5362219.2	1.0	709.8	22.2	1376.0	181.6	149.6
	751	0.506	25298684.2	1.0	304.9	104.9	4687.5	894.0	711.5
	1855	0.802	40103196.8	1.0	184.1	278.3	7445.0	2141.8	1459.7

Algorithm ChordalGen together with GrowingSubtree is able to output connected chordal graphs unless density is too low, as the results in Table 6.1 show. In fact, for an average edge density of 0.01, as n increases, the average number of connected components converges to one. If we examine the "Min clique size" column, we see

that it is usually one for cases where the average number of connected components is greater than one, suggesting that the reason for obtaining disconnected chordal graphs is largely due to a few isolated vertices and that the dominating rest of the graph is comprised of a connected body. The fact that the starting point of the subtrees is selected uniformly at random and we can directly control the maximum size of them leaves little chance for a set of subtrees not to intersect with any other, leading to a separate connected component, unless the maximum subtree size k is very small.

Table 6.2. Experimental results of Algorithm ChordalGen with ConnectingNodes method.

n	λ	Density	m	# conn. comp.s	# maximal cliques	Min clique size	Max clique size	Mean clique size	Sd of clique sizes
1000	0.5	0.011	5455.4	349.0	597.0	1.0	75.8	3.0	5.5
	1.2	0.100	49805.1	121.4	495.3	1.0	266.5	8.0	23.1
	2.7	0.507	253074.6	8.6	238.7	1.0	627.0	30.7	87.4
	4.1	0.804	401708.6	1.8	96.3	1.6	835.4	81.5	183.9
2500	0.6	0.010	31559.8	849.2	1475.8	1.0	194.6	3.5	10.1
	1.2	0.101	314818.0	298.5	1215.3	1.0	657.5	9.6	40.7
	2.7	0.503	1571946.8	27.9	594.0	1.0	1620.6	36.0	142.1
	4.1	0.800	2498034.2	3.1	226.5	1.2	2074.8	102.8	315.3
5000	0.6	0.010	127700.5	1693.5	2960.1	1.0	395.8	3.8	15.2
	1.2	0.103	1290089.2	578.3	2409.8	1.0	1396.8	10.6	57.9
	2.7	0.505	6308093.4	44.4	1148.6	1.0	3217.5	41.0	211.0
	4.1	0.805	10060406.5	4.7	435.7	1.0	4261.4	119.4	460.6
10000	0.6	0.010	501760.2	3365.6	5901.9	1.0	806.2	4.0	21.8
	1.2	0.100	5022899.1	1180.4	4794.0	1.0	2703.4	11.7	83.8
	2.7	0.502	25114409.8	97.3	2300.6	1.0	6355.1	44.0	291.5
	4.1	0.803	40154270.6	9.3	852.9	1.0	8484.9	136.2	682.4

Table 6.2 reports the outputs of Algorithm ChordalGen using ConnectingNodes for generating subtrees. As the experimental results given in Table 6.2 reveal, Algorithm ConnectingNodes should be input very small λ values in order to achieve even quite dense graphs. Since even few number of selected nodes may result in large subtrees, which increases the chances of potential intersections with other subtrees and hence the number of edges in the output graph, the number of selected nodes has to be restricted via low values for λ, which is the main ingredient in setting the cardinality of

node selection. Because of this, the selected node set, and hence the subtree, commonly is comprised of a single node, especially in graphs with low density. Therefore, when there are many single-node subtrees, intersections are not very likely. Thus, we observe many isolated vertices in the generated chordal graphs, as implied by the high number of connected components and minimum size of one in maximal cliques (see "min clique size" column) in ConnectingNodes method.

Table 6.3 presents the experimental results of Algorithm ChordalGen when used with PrunedTree method. The two columns "Edge del. fr. (f)" and "Selection barrier (s)" correspond to input parameters that PrunedTree takes as input, whose roles are explained in Section 6.1.3.3. Here, we observe that for density values of 0.1, 0.5, and 0.8, the output graphs are predominantly connected. As in the previous two methods, minimum size of maximal cliques in case of 0.01 density is one, implying that the main cause of the number of connected components is probably a small group of isolated vertices.

Table 6.3. Experimental results of Algorithm ChordalGen with PrunedTree method.

n	Edge del. fr. (f)	Selection barrier (s)	Density	m	# conn. comp.s	# maximal cliques	Min clique size	Max clique size	Mean clique size	Sd of clique sizes
1000	0.950	0.35	0.011	5619.3	45.8	324.5	1.0	30.4	5.5	4.1
	0.700	0.60	0.104	51765.9	1.0	99.9	4.2	133.6	35.9	25.2
	0.140	0.85	0.497	248172.1	1.0	50.2	193.0	337.1	278.3	35.2
	0.100	0.95	0.806	402349.8	1.0	36.5	397.7	621.6	530.8	53.6
2500	0.950	0.70	0.011	34013.9	28.3	542.5	1.0	72.3	9.5	8.8
	0.700	0.70	0.101	316270.6	1.0	150.4	5.5	335.0	70.8	58.0
	0.120	0.90	0.507	1584225.2	1.0	66.2	492.0	844.4	703.2	84.4
	0.077	0.95	0.801	2500840.1	1.0	56.5	996.2	1530.1	1304.7	123.0
5000	0.950	0.77	0.010	130970.2	21.7	833.8	1.0	177.7	14.0	15.2
	0.700	0.75	0.097	1216527.9	1.0	202.6	4.8	655.2	117.0	106.5
	0.080	0.95	0.495	6182739.6	1.0	101.0	1083.4	1571.7	1395.7	109.0
	0.045	0.95	0.801	10004264.5	1.0	99.9	2269.3	2955.8	2672.9	146.1
10000	0.900	0.50	0.010	479501.2	22.5	1394.8	1.0	286.6	20.6	24.5
	0.700	0.81	0.102	5076707.1	1.0	260.0	7.2	1415.8	204.5	206.0
	0.060	0.95	0.507	25357868.2	1.0	143.5	2359.9	3176.8	2882.4	177.7
	0.031	0.95	0.793	39653114.8	1.0	157.7	4705.0	5709.5	5319.0	198.9

We want to compare our results to the results showing the kind of chordal graphs that are generated by Alg 2a [57]. Note, however that, the results given by [57] only contain graphs on 10000 vertices, with varying number of edges. Most metrics presented in [57] are about the number of edges. When it comes to the maximal cliques, they present only the average maximum clique size over the generated graphs for each edge density. Comparing these to our numbers we see that graphs corresponding to edge densities 0.01, 0.1, 0.5, and 0.8 of Alg 2a have average maximum clique sizes of 727, 2847, 6875, and 8760, respectively. As can be seen from Tables 6.1-6.3, these numbers are significantly higher than the corresponding numbers for the graphs generated by Algorithm ChordalGen. In fact, studying the numbers more carefully, we can conclude that the maximum clique of a graph generated by Alg 2a contains almost all the edges of the graph. In the case of density 0.01, such a clique contains more than half of the edges, whereas in the case of higher densities, the largest clique contains more than 80, 94, and 95 percent of the edges, respectively. Thus, there does not seem to be an even distribution of the sizes of maximal cliques of graphs generated by Alg 2a.

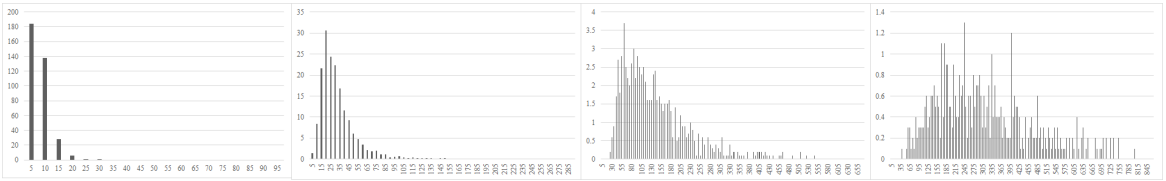
We also implemented Alg 1 [2], but without imposing a limit on the maximum degree of the output graph, because no detail was given about how the method avoids exceeding a given maximum degree in [2]. In Table 6.4 we give results of Alg 1 analogous to Tables 6.1–6.3 for 1000, 2500, and 5000 vertices. In order to obtain results for Table 6.4 comparable to those given in Tables 6.1-6.3, we aimed to have approximately the same edge density values. For this purpose, when determining the number of new neighbors of a vertex at each step in Alg 1, we multiplied the total number of candidate vertices with a coefficient between 0 and 1, which we call *upper bound coefficient*. A running time analysis for this algorithm has not been given in [2]. With our implementation, this algorithm turned out to be too slow to allow testing graphs on 10000 vertices in a reasonable amount of time. However, already from the obtained numbers, we can reach a conclusion for Alg 1 similar to that on Alg 2a. We observe that the maximum clique sizes obtained for 5000 vertices by Alg 1, are comparable to the maximum clique sizes obtained for 10000 vertices by Algorithm ChordalGen. Hence, like Alg 2a, also Alg 1 seems to generate graphs with a few large maximal cliques.

Table 6.4. Experimental results of our implementation of Alg 1 [2]

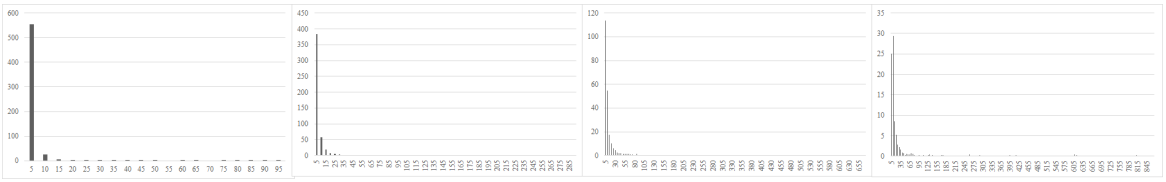
n	Upper bound coef.	Density	m	# conn. comp.s	# maximal cliques	Min clique size	Max clique size	Mean clique size	Sd of clique sizes
1000	0.00130	0.011	5659.3	1.0	933.3	2.0	58.0	4.9	9.4
	0.00300	0.100	49717.1	1.0	753.3	2.0	219.3	28.4	60.0
	0.01100	0.506	252864.4	1.0	401.5	2.0	562.5	190.1	233.6
	0.03500	0.805	401945.6	1.0	191.6	2.4	788.6	399.4	342.6
2500	0.00053	0.011	33201.8	1.0	2320.8	2.0	154.5	8.8	26.1
	0.00120	0.100	313001.8	1.0	1882.3	2.0	548.8	69.3	159.3
	0.00440	0.502	1568857.4	1.0	1006.5	2.0	1400.6	462.7	593.3
	0.01400	0.799	2495447.1	1.0	469.8	2.0	1975.9	936.8	888.2
5000	0.00027	0.011	133829.0	1.0	4629.0	2.0	313.9	15.9	56.2
	0.00062	0.107	1339169.7	1.0	3717.9	2.0	1136.5	147.0	342.7
	0.00220	0.494	6179872.9	1.0	2032.5	2.0	2774.2	897.7	1180.8
	0.00700	0.801	10011146.3	1.0	939.0	2.0	3950.0	1901.6	1794.5

As can be seen in Table 6.4, Alg 1 outputs connected chordal graphs for the selected set of average edge density values and number of vertices. The minimum size of the maximal cliques did not show much variation throughout our experiments and almost always turned out to be two. The consistency in this measure may be an indication of the lack of potential to produce a diverse range of maximal clique sizes.

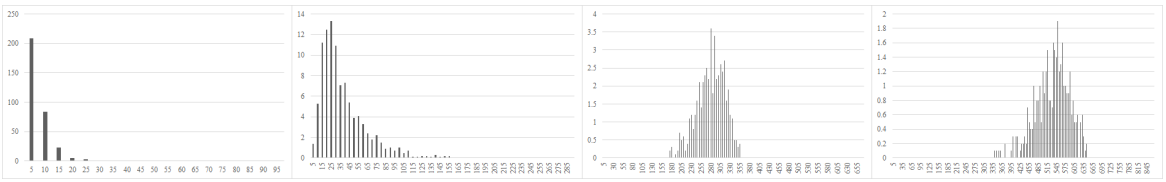
In our next set of experimental results we investigate how the sizes of the maximal cliques are distributed. We measure the variety of the produced graphs using the maximal cliques of the generated graph, noticing that maximal cliques play an important role in various characterizations of chordal graphs and that it has been done so in previous work [56]. We give extensive computational tests to demonstrate the kind of chordal graphs that our algorithm generates using each of the different subtree generation methods. We compare our methods with one another and with Alg 1 and Alg 2a. Figures 6.10-6.13 show the average number of maximal cliques across ten independent runs in intervals of width five, for 1000, 2500, 5000, and 10000 vertices and varying edge densities. These figures consist of four subfigures, except Figure 6.13, which contains only the first three, and each subfigure is comprised of four histograms corresponding to four different average edge density values. The first three subfigures



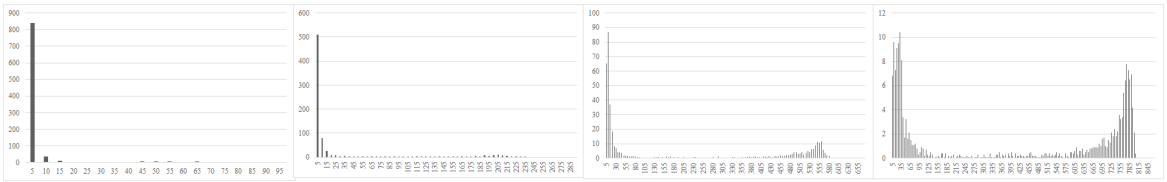
(a) Results from Algorithm ChordalGen with GrowingSubtree method



(b) Results from Algorithm ChordalGen with ConnectingNodes method



(c) Results from Algorithm ChordalGen with PrunedTree method

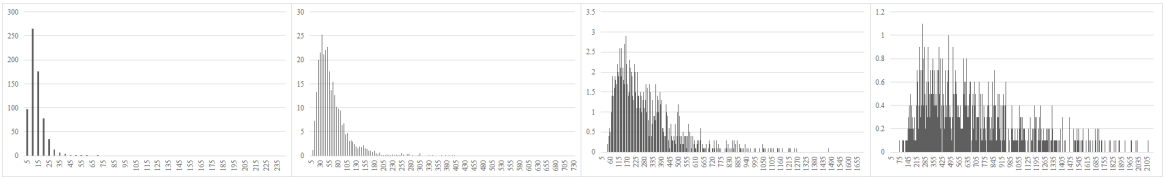


(d) Results from our implementation of Alg 1 [2]

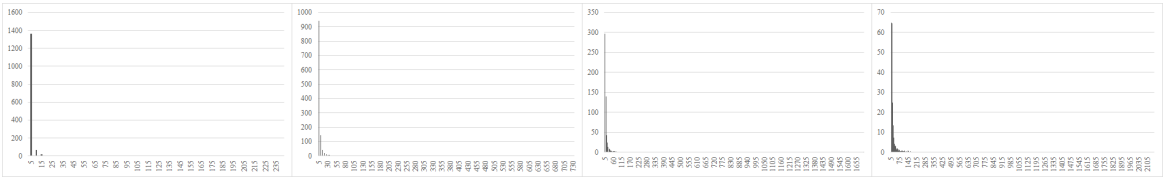
Figure 6.10. Histograms of maximal clique sizes for $n = 1000$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).

on the top row show the results from Algorithm ChordalGen combined with each one of the three subtree generation methods presented, and the bottom row shows results of our implementation of Alg 1 [2]. For a given n and average density value, the ranges of x-axes are kept the same in order to render the histograms comparable. The y-axes, however, have different ranges because maximum frequencies in histograms may vary drastically.

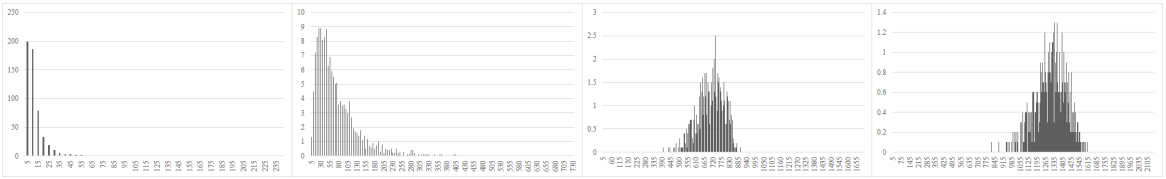
The sizes of maximal cliques of graphs produced by GrowingSubtree method are dispersed fairly over the range, which becomes more noticeable with the increase in



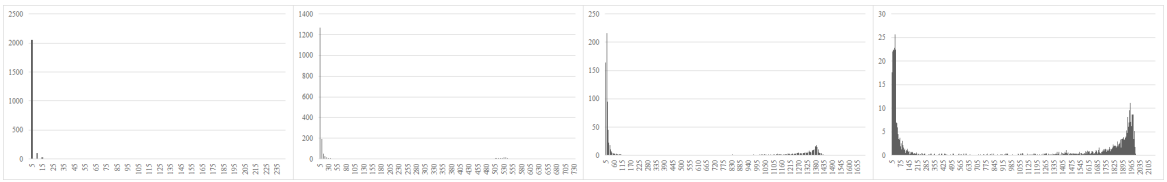
(a) Results from Algorithm ChordalGen with GrowingSubtree method



(b) Results from Algorithm ChordalGen with ConnectingNodes method



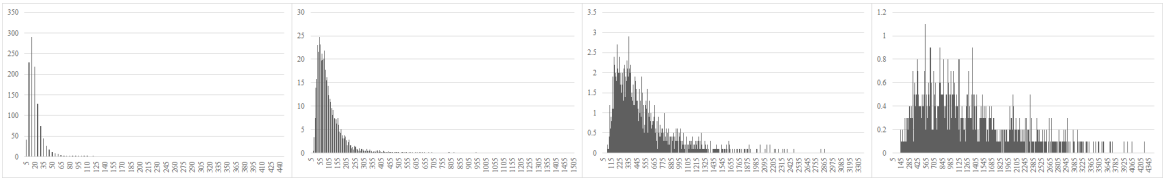
(c) Results from Algorithm ChordalGen with PrunedTree method



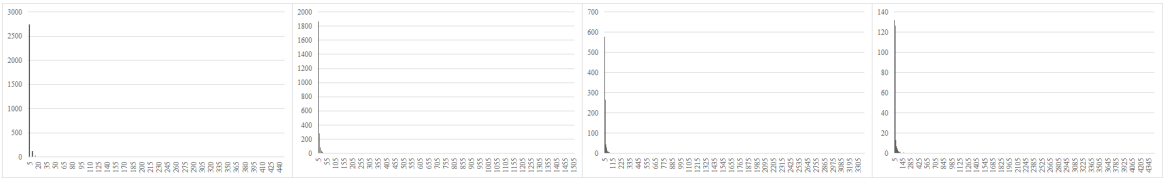
(d) Results from our implementation of Alg 1 [2]

Figure 6.11. Histograms of maximal clique sizes for $n = 2500$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).

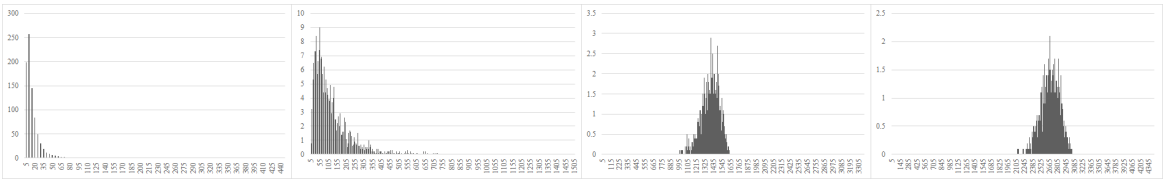
edge densities (as we proceed to the right). Frequencies do not show any obvious bias toward some portion of its domain, which may be considered as an indicator of the diversity of the chordal graphs produced, which is a desired characteristic of a random chordal graph generator. In ConnectingNodes method, however, the vast majority of cliques have size ten or less. The frequencies of larger cliques are barely noticeable compared to the dominant small-sized set. As the graphs become denser, frequencies of relatively larger cliques start to become visible too, but general behaviour remains the same. So, if chordal graphs with predominantly very small clique sizes are sought, ConnectingNodes method can be preferred. In PrunedTree method, the mode of the



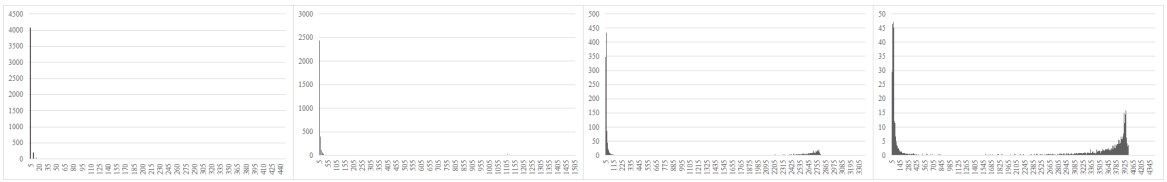
(a) Results from Algorithm ChordalGen with GrowingSubtree method



(b) Results from Algorithm ChordalGen with ConnectingNodes method



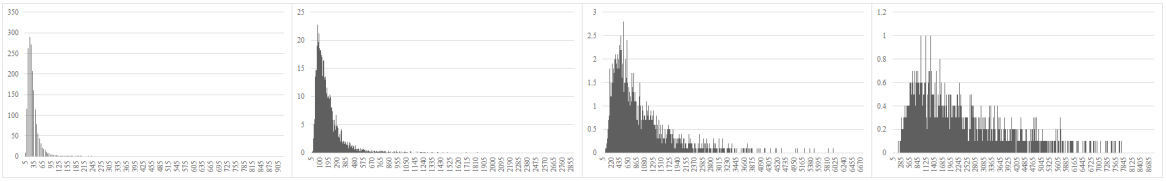
(c) Results from Algorithm ChordalGen with PrunedTree method



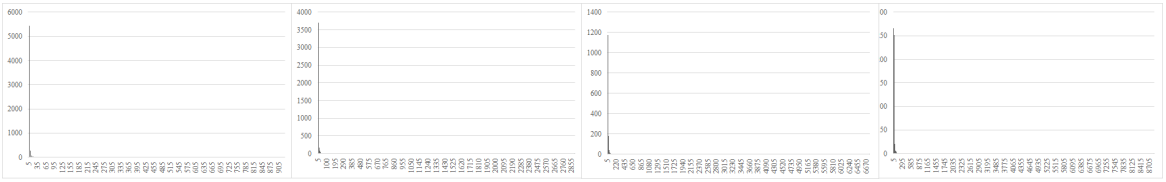
(d) Results from our implementation of Alg 1 [2]

Figure 6.12. Histograms of maximal clique sizes for $n = 5000$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).

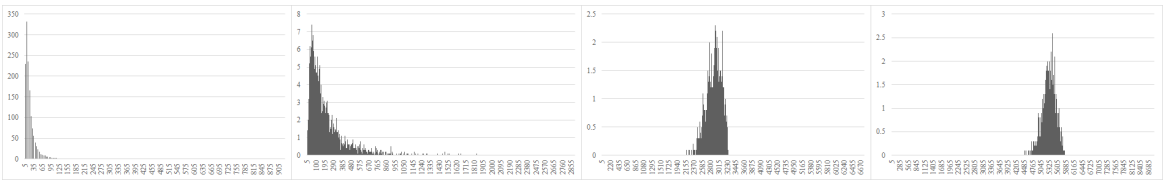
distribution shifts with the increase in edge densities and the sizes of cliques become clustered around some moderate value over the given range. As for Alg 1, the vast majority of maximal cliques of its output graphs have sizes of 2 to 15 for graphs with low densities of 0.01 and 0.1. With the increase in edge densities, frequencies of large-size maximal cliques become visible relative to the dominant small clique frequencies; however, all but the extremes of the range is barely used regardless of selection of n and edge density.



(a) Results from Algorithm ChordalGen with GrowingSubtree method



(b) Results from Algorithm ChordalGen with ConnectingNodes method

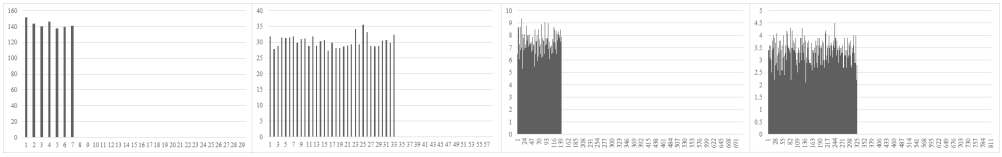


(c) Results from Algorithm ChordalGen with PrunedTree method

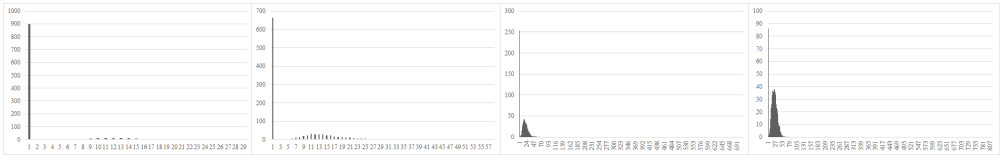
Figure 6.13. Histograms of maximal clique sizes for $n = 10000$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).

We also want to give an indication of how the sizes of the subtrees in our three subtree generation methods are distributed. To this end, we provide histograms of subtree sizes in Figures 6.14–6.17 for 1000, 2500, 5000, and 10000 vertices with varying edge densities. These figures are comprised of three subfigures, and each subfigure contains four histograms corresponding to the four different average edge density values we consider. The x-axes of the histograms show the number of nodes in subtrees, and the y-axes show the average number of subtrees across ten independent runs. For a given n and average density value, we kept the ranges of x-axes the same to be able to compare the shapes of histograms on an equal footing. The y-axes, however, have different ranges since maximum frequencies differ from each other considerably.

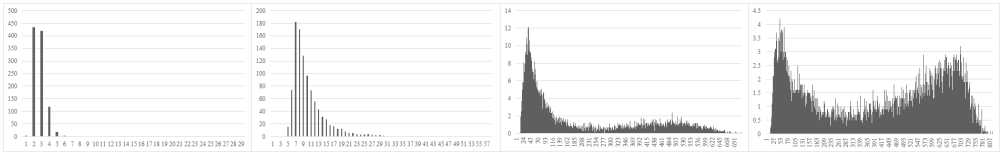
Our first observation from Figures 6.14–6.17 is that, for a given subtree generation method and average edge density, the shapes of the histograms remain roughly the same for different values of n we consider here. As mentioned before, GrowingSubtree method



(a) Results from Algorithm ChordalGen with GrowingSubtree method

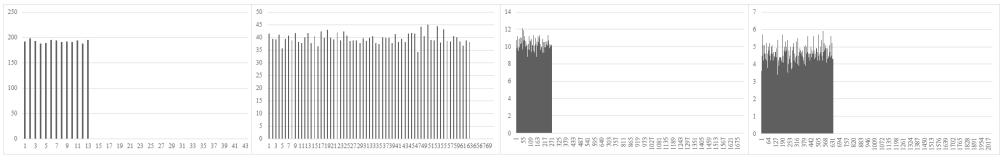


(b) Results from Algorithm ChordalGen with ConnectingNodes method

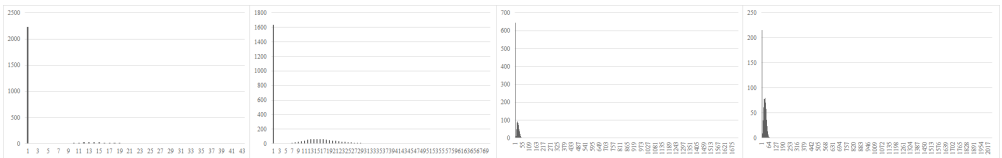


(c) Results from Algorithm ChordalGen with PrunedTree method

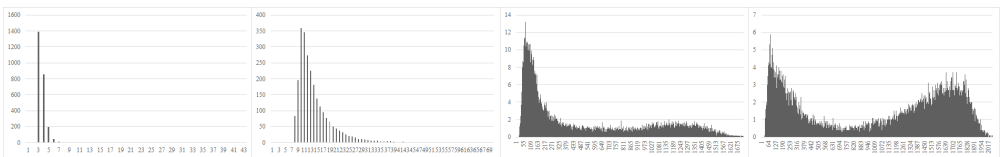
Figure 6.14. Histograms of subtree sizes for $n = 1000$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).



(a) Results from Algorithm ChordalGen with GrowingSubtree method



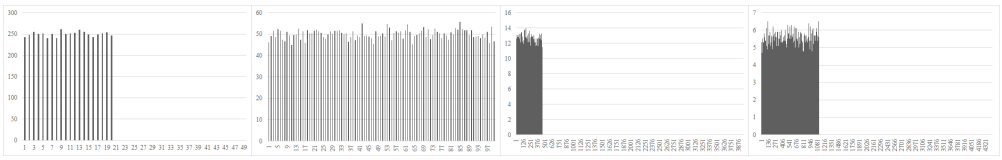
(b) Results from Algorithm ChordalGen with ConnectingNodes method



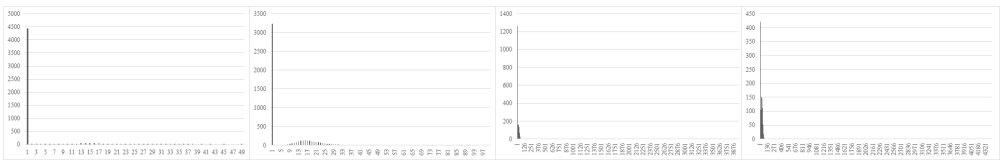
(c) Results from Algorithm ChordalGen with PrunedTree method

Figure 6.15. Histograms of subtree sizes for $n = 2500$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).

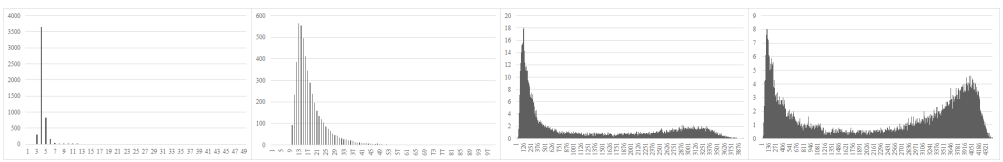
takes an upper bound k on the sizes of subtrees and determines the sizes of subtrees uniformly at random between 1 and k, which results in an expected even distribution of subtree sizes as we observe from Figures 6.13a–6.16a. In ConnectingNodes method, the biggest percentage in subtree sizes is one, which means a considerable portion of subtrees are comprised of a single node, as we observe from Figures 6.13b–6.16b. This is expected because the value of parameter λ needs to be very small in order to obtain even quite dense graphs, and this results is in a high number of single-node subtrees. Finally, we see that either none or just a few of the subtrees contain only a single node, and the relative frequency of moderately-sized subtrees are usually low in PrunedTree method. When edge densities are 0.01 and 0.1, almost all of the subtrees are small-sized. When we increase edge density, Algorithm PrunedTree increases the frequency of large-sized subtrees more than others and exhibits a double-hunched distribution.



(a) Results from Algorithm ChordalGen with GrowingSubtree method



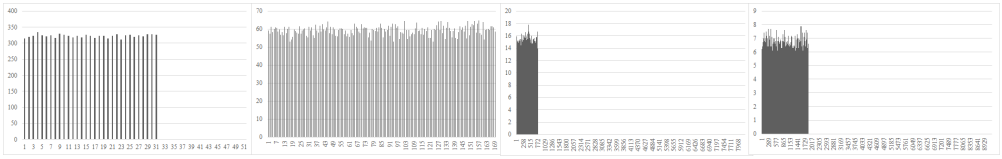
(b) Results from Algorithm ChordalGen with ConnectingNodes method



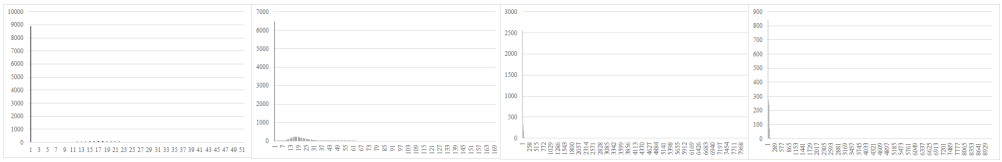
(c) Results from Algorithm ChordalGen with PrunedTree method

Figure 6.16. Histograms of subtree sizes for $n = 5000$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).

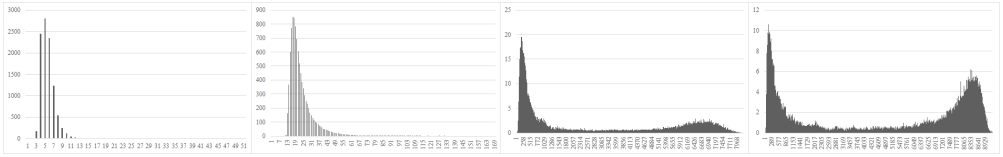
In addition to the collection of instances whose descriptive data have been summarized above, we have generated another set of chordal graph instances with smaller n-values to be used in testing the performance of our decomposition algorithm. We gen-



(a) Results from Algorithm ChordalGen with GrowingSubtree method



(b) Results from Algorithm ChordalGen with ConnectingNodes method



(c) Results from Algorithm ChordalGen with PrunedTree method

Figure 6.17. Histograms of subtree sizes for $n = 10000$ and average edge densities 0.01, 0.1, 0.5, and 0.8 (from left to right).

erated this new set chordal graphs using Algorithm ChordalGen together with GrowingSubtree method since we have observed that this combination produces most varied random graphs in terms of the distribution of maximal cliques. The number of vertices in this set varies from 100 to 1000, and the average edge density values are 0.1, 0.3, 0.5, and 0.7. As before, we tuned the maximum subtree size parameter k to achieve the desired edge density values on average, and produced ten random graphs for each n and average edge density pair. For brevity, we selected three representative n-values, which are 200, 500, and 800, and present their descriptive data in Table 6.5 and in Figure 6.18.

Our observations from Table 6.5 are similar to those for graphs with larger values of n. We obtain connected chordal graphs unless the edge density is low and n is small. When $n = 200$ and average edge density is 0.1 or 0.3, GrowingSubtree method occasionally results in disconnected chordal graphs, and the disconnectedness may result from isolated vertices, as the first row of the table with an average minimum

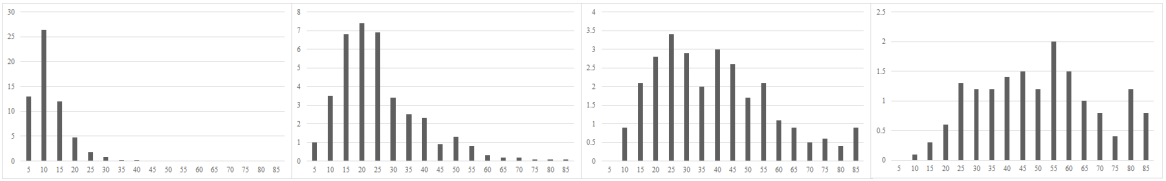
clique size value less than two reveals.

Table 6.5. Experimental results of Algorithm ChordalGen with GrowingSubtree method for relatively small n-values.

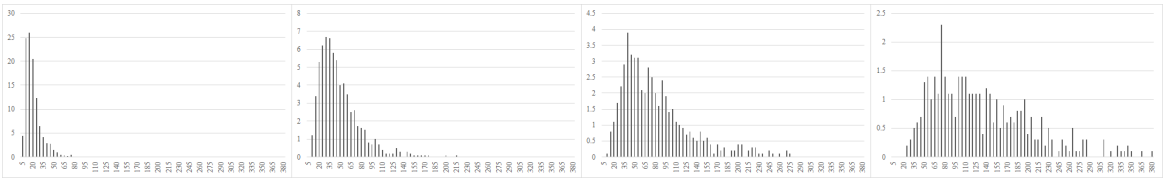
n	Max subtree size (k)	Density	m	# conn. comp.s	# maximal cliques	Min clique size	Max clique size	Mean clique size	Sd of clique sizes
200	11.0	0.106	2107.7	1.3	59.2	1.9	29.1	9.6	5.6
	26.0	0.306	6087.2	1.1	37.8	5.5	63.0	23.7	13.4
	43.0	0.495	9851.4	1.0	28.9	9.9	92.7	39.1	21.4
	70.0	0.695	13835.8	1.0	22.0	19.9	127.8	63.6	30.8
500	20.0	0.101	12637.6	1.0	107.9	3.1	66.3	18.2	11.5
	49.0	0.300	37460.1	1.0	68.3	9.6	160.1	46.8	29.3
	82.0	0.503	62809.4	1.0	54.1	16.2	245.4	78.9	49.7
	134.0	0.705	87934.8	1.0	40.6	32.4	326.0	130.4	71.0
800	29.0	0.103	32823.0	1.0	147.4	4.7	104.6	26.4	17.0
	69.0	0.304	97071.6	1.0	90.6	15.9	253.1	66.3	44.9
	115.0	0.497	158801.0	1.0	67.8	26.4	386.6	116.6	74.5
	199.0	0.698	222978.1	1.0	50.9	48.2	511.4	194.6	109.7

In order to provide visual insight as to what kind of chordal graphs our algorithm with GrowingSubtree method produces, we again provide histograms of maximal clique sizes in Figure 6.18. The figure consists of three subfigures and each subfigure contains four histograms corresponding to four different average edge density values. We again kept the ranges of x-axes the same for a given pair of n and average edge density, to make them comparable. As in the histograms of chordal graphs with larger values of n, the sizes of maximal cliques are distributed fairly over the range, especially when the edge density increases. For $n = 200$, distribution of sizes in graphs with approximate density of 0.5 and 0.7 are not much skewed to the right, as opposed to cases where n is higher.

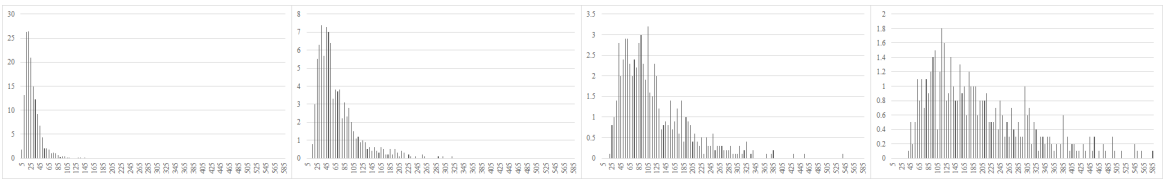
To summarize, the three different subtree generation methods presented here each offer output graphs of different structures. As far as the distribution of maximal clique sizes are concerned, ConnectingNodes and PrunedTree methods yield graphs of somewhat more specific structure in the sense that the sizes of maximal cliques



(a) $n = 200$



(b) $n = 500$



(c) $n = 800$

Figure 6.18. Histograms of maximal clique sizes of instances produced using Algorithm ChordalGen with GrowingSubtree for $n = 200$, $n = 500$, and $n = 800$, and average edge densities 0.1, 0.3, 0.5, and 0.7 (from left to right in each row).

are always clustered in the very initial portion in ConnectingNodes method, and in PrunedTree method in the initial part of the range for low densities, in middle portions for moderate to high densities. GrowingSubtree method, though, generates the most varied chordal graphs compared to both existing methods and to other two of our suggested methods. Depending on the context and structural needs for the output graph, Algorithm ChordalGen can be used with one of the three subroutines chosen suitably in order to produce chordal graphs of varying size and density.

Last but not least, our work gives rise to an open question about chordal graphs which has not been addressed to date to the best of our knowledge: what is the worst case time complexity of a chordal graph generation algorithm which produces the entire set of nodes of all subtrees in a subtree intersection model? In the current paper, we give a lower bound, namely $\Omega(mn^{1/4})$, on the time complexity. However, the exact

worst case time complexity remains unknown.

6.1.4. General Perfect Graph Generation

A graph G is *perfect* if, for every induced subgraph, the size of a maximum clique is equal to the minimum number of colors required to color it; i.e., if for every induced subgraph $G' \subseteq G$, $\chi(G') = \omega(G')$ holds. To the best of our knowledge, there have been only theoretical studies on perfect graph generation in its general form. In his survey [64], Chvátal raises the question of whether all perfect graphs might be constructible from some "primitive" perfect graphs using perfection-preserving operations, while exemplifying some classes all elements of which can be set up through this idea. Though that question still remains to be answered, there are operations proven to preserve perfection that can seemingly serve well to the purpose of generating perfect graphs. Our perfect graph generation algorithm, which we call Algorithm PerfectGen, is based on this idea. We take a diverse set of small-sized perfect graphs and reach to an end-graph by combining randomly selected ones via perfection-preserving operations. The size and range of the set of small-sized graphs being an important ingredient in the diversity of the perfect graph to be produced, we aimed to keep it as broad as possible. For this purpose, we made use of the set of all non-isomorphic connected graphs up to nine vertices, offered by McKay [65]. By making use of the two well-known theorems presented below, we filtered out the ones that are not perfect, and used the remaining collection to build larger perfect graphs.

Theorem 6.15. *(Strong Perfect Graph Theorem, [21]) A graph G is perfect if and only if neither G nor $\bar{G}$ contains an odd cycle of length at least five.*

Theorem 6.16. *(Weak Perfect Graph Theorem, [66, 67]) A graph G is perfect if and only if its complement $\bar{G}$ is perfect.*

In order to check whether a given graph is perfect, we use the well-known characterization of perfect graphs given in Theorem 6.15. We check the cycles of the input graph. If we find an odd hole of size five or more; i.e., if we detect a cycle with size

five or more that is comprised of an odd number of vertices and has no chord in it, we conclude that the graph is not perfect. Otherwise (if no odd hole is present in the input graph), we take the complement of the graph and do the same check. If an odd hole of size five or more exists in the complement, we conclude that the original input graph is not perfect; else, it is perfect. This procedure is applied to all connected non-isomorphic graphs having at most nine vertices, which are offered in [65], and those passing the check are added to the collection, say $\mathcal{P}$, to be used for perfect graph generation. The number of connected non-isomorphic graphs having one to nine vertices offered by McKay [65] are respectively 1, 1, 2, 6, 21, 112, 853, 11117, and 261080, and the number of perfect graphs in this collection turned out to be 1, 1, 2, 6, 20, 105, 724, 7805, and 126777, respectively.

Algorithm PerfectGen works as follows: We input a desired number of vertices n and a desired edge density ρ to the algorithm. Initially, we randomly choose a perfect graph from collection $\mathcal{P}$, which is to be extended into a final perfect graph on n vertices. Then, at each step, we first pick a random perfection-preserving operation *op* among the six such operations we selected from the literature, whose details we are going to provide in the sequel. If the selected operation *op* necessitates a perfect graph other than the current perfect graph G that is being extended, then we randomly pick a graph G' from $\mathcal{P}$ and combine G and G' via operation *op*. Otherwise, we simply apply operation *op* to G. This routine continues until G has n vertices in total.

The first part of the algorithm explained above has no mechanism to control the number of edges in G. In fact, it is not very easy to directly control the number of edges because the change in the number of edges as well as in the number of vertices cannot usually be foreseen before starting to apply the operation. Moreover, the change in the number of edges is not monotonic throughout the iterations in general; i.e., it can increase, decrease (only possible if we take the complement of the graph), or stay the same. Thus, we first build a perfect graph G on n vertices and then check its edge density d. If d is within some predetermined ϵ-distance from the desired edge density ρ, then we accept G and terminate the algorithm. Otherwise, if we can achieve the

Input: An integer n and a real number ρ between 0 and 1

Output: A perfect graph G on n vertices with (approximate) edge density ρ

$\epsilon \leftarrow 0.025$

$d \leftarrow 0$

while $d < \rho - \epsilon$ **or** $d > \rho + \epsilon$ **do**

  Let $G = (V, E)$ be a graph selected randomly from the collection of small-sized perfect graphs $\mathcal{P}$ such that $|V| \leq n$

  while $|V| < n$ **do**

    Select a perfection-preserving operation *op* randomly

    if *op* requires another input graph **then**

      Select a random graph $G' = (V', E')$ from $\mathcal{P}$ with $|V'| \leq n - |V|$

      Attach G' to G via operation *op*

    else

      Modify G with operation *op*

    end if

  end while

  $m \leftarrow |E|$

  $d \leftarrow \frac{m}{\frac{n(n-1)}{2}}$

  if $\rho - \epsilon < 1 - d < \rho + \epsilon$ **then**

    $G \leftarrow \bar{G}$, where $\bar{G}$ is the complement of G

    $d \leftarrow 1 - d$

  end if

end while

Figure 6.19. Algorithm PerfectGen.

desired density by taking the complement of G, then we deliver $\bar{G}$ as the output graph; else, we simply discard G and start to construct a new perfect graph from scratch. Pseudo-code of the algorithm is presented in Figure 6.19.

Now we present the set of six perfection-preserving operations that we have used in our perfect graph generation algorithm.

- *Clique identification* [68]:

  Let G_1, G_2 be disjoint graphs, and K_i be a nonempty clique in G_i satisfying $|K_1| = |K_2|$. Define a one-to-one correspondence between vertices of K_1 and K_2; i.e., choose a bijective map $f : K_1 \to K_2$. A graph obtained by unifying each vertex v in K_1 with vertex $f(v)$ in K_2 is said to arise from G_1 and G_2 by clique identification. A graph G obtained from two perfect graphs via clique identification is perfect.

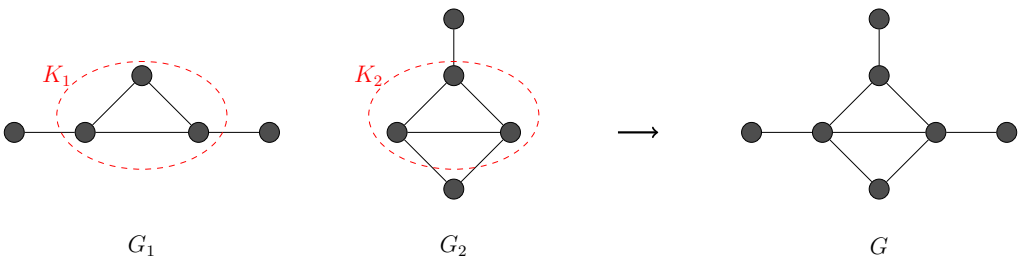


Figure 6.20. Two perfect graphs combined by clique identification operation.

- *Disjoint union*:

  Let G_1, G_2 be two disjoint graphs. The disjoint union of G_1 and G_2 is simply $G = G_1 \cup G_2$ with $V(G) = V(G_1) \cup V(G_2)$ and $E(G) = E(G_1) \cup E(G_2)$. Disjoint union of two perfect graphs is again perfect (obvious from the definition of perfect graphs).

- *Substitution* [66]:

  Let G_1, G_2 be disjoint graphs, v be a vertex of G_1, and N be the set of all neighbors of v in G_1. Removing v from G_1 and linking each vertex in G_2 to those in N results in a graph that arises from G_1 and G_2 by substitution. If G_1 and G_2 are perfect, a graph G derived via substitution of the two is perfect, too. We note that this operation is also known as *Replication Lemma* in the literature and it played an important role in the proof of the Weak Perfect Graph Theorem (Theorem 6.16).

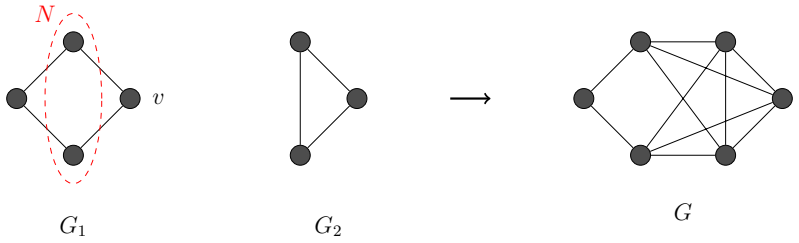


Figure 6.21. Two perfect graphs combined by substitution operation.

- *"Composition" by Bixby* [69, 70]:

  Let G_1, G_2 be disjoint graphs each with at least three vertices, v_i be a vertex of G_i, $N(v_i)$ the set of all neighbors of v_i. The composition of G_1 and G_2 is obtained from $G_1 \setminus \{v_1\}$ and $G_2 \setminus \{v_2\}$ by connecting all vertices in $N(v_1)$ to those in $N(v_2)$. A graph obtained from two perfect graphs via composition operation is again perfect.

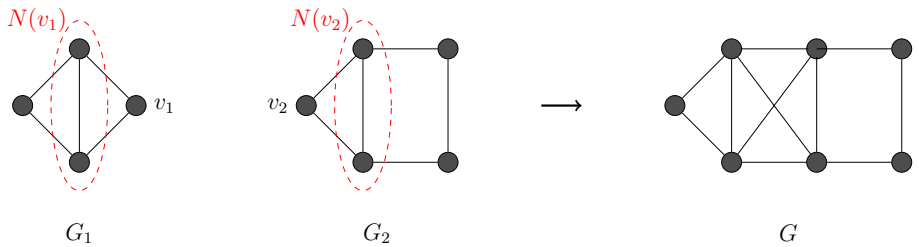


Figure 6.22. Two perfect graphs combined by composition operation.

- *Complement*:

  By Theorem 6.16, the complement of a perfect graph is again perfect.

- *Join*:

  Let G_1, G_2 be disjoint graphs. The join of G_1 and G_2, say G, is obtained by connecting all vertices in G_1 to all those in G_2. A graph obtained from two perfect graphs via join operation is perfect. To show that this operation indeed preserves perfection, assume that G_1 and G_2 are perfect. Consider $\bar{G}$ which is simply $\bar{G_1} \cup \bar{G_2}$. G_1 and G_2 being perfect, $\bar{G_1}$ and $\bar{G_2}$ are so, too, by Theorem 6.16. As disjoint union of two perfect graphs is perfect, $\bar{G} = \bar{G_1} \cup \bar{G_2}$ and therefore G is perfect.

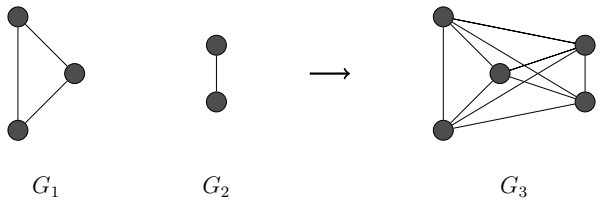


Figure 6.23. Two perfect graphs combined by join operation.

All of the perfect graph instances that we have generated with the presented method can be accessed online at [71].

6.2. Partition Generation

The algorithm we designed to generate a random partition of a given vertex set into clusters takes a pair of integers to be respectively the lower and upper bounds on the sizes of clusters as input. The first phase of the algorithm initially creates a random ordering σ of vertices. Then, at each step, the size r of the cluster under construction is set uniformly random between the lower and upper bounds input to the algorithm, and a separator is placed r-many elements ahead of the previous cluster's last vertex in σ. The set of vertices between two consecutive points the separator is placed serves as one cluster. This procedure continues until all vertices in V belong to some cluster.

In the second phase of this procedure, we make sure that the number of vertices in each cluster is between the lower and upper bounds input to the algorithm. At some step of the first phase, if the separator does not attempt to go beyond the last vertex in σ, then the number of vertices in that cluster will simply range between the lower and upper bounds, as desired. Otherwise, that is, if the predetermined random size r of a cluster tries to push the separator beyond the last vertex in σ, then this last cluster is going to contain fewer than r vertices. If the resultant size is still between the lower and upper bounds, then there is no need for further action; the partitioning process is finished. Otherwise, we distribute the vertices of this last cluster to other clusters that still have room for vertex addition. Although not very probable, redistribution of the

Input: A set V of vertices, and a pair of positive integers $\{lb, ub\}$ with $lb \geq 1$ and $ub \geq lb$

Output: A partition $\mathcal{V}$ of V

Create a random ordering σ of the vertices in V, and $sep \leftarrow 0$, $i \leftarrow 1$, $t \leftarrow 1$

while $sep \leq n$ **do**

  Generate a random integer r between lb and ub, $l \leftarrow sep + r$, and $V_i \leftarrow \emptyset$

  if $l > n$ **then**

    $l \leftarrow n$

  end if

  $V_i \leftarrow \{\sigma(sep+1), \ldots, \sigma(l)\}$, $sep \leftarrow sep + r$, $t \leftarrow i$, $i \leftarrow i + 1$

end while

$\mathcal{V} \leftarrow \{V_1, \ldots, V_t\}$

if $|V_t| < lb$ **then**

  for all $v \in V_t$ **do**

    $i \leftarrow 1$

    while $i \leq t - 1$ **do**

      if $|V_i| < ub$ **then**

        $V_i \leftarrow V_i \cup \{v\}$, $V_t \leftarrow V_t \setminus \{v\}$, $i \leftarrow t$

      end if

      $i \leftarrow i + 1$

    end while

  end for

end if

if $V_t \neq \emptyset$ **then**

  Create a partition $\mathcal{V}$ from scratch

else

  $\mathcal{V} \leftarrow \mathcal{V} \setminus V_t$

end if

Figure 6.24. Algorithm PartitionGen.

vertices in the last cluster to others may not succeed, simply because other clusters do not happen to have enough room. In such cases, we build the partition starting from scratch. Pseudo-code of this algorithm is provided in Figure 6.24. The collection of random partitions of the vertex sets for the graph instances we have generated are available online in the addresses provided in previous sections.

Having presented the methods we designed to generate random graph instances together with the partitions of the vertex sets, we present the computational results we obtained for each one of the solution methods we consider for SEL-COL in the following chapter.

7. COMPUTATIONAL STUDY

In this chapter, we present the results of a series of experiments we conducted to evaluate the performance of our decomposition procedure by comparing it with that of the integer programming formulation Model 2 and the branch-and-price algorithm by Furini *et al.* [30].

We implemented the algorithms described in the previous chapter in C++, and executed them on a computer with 2.00-GHz Intel Xeon CPU. Throughout all the experiments, we used CPLEX version 12.8. In the case of chordal graphs, we utilized the lazy constraint feature of CPLEX, and in all other classes of graphs, we used the callback mechanism of CPLEX. To solve the SDP formulations for perfect graphs in its general form, we used MOSEK version 8.1.0.24.

We randomly generated our test instances for different n values ranging from 50 to 1000, and four different average edge density values 0.1, 0.3, 0.5, and 0.7, where edge density of a graph is defined as $\frac{m}{\frac{n(n-1)}{2}}$ with m denoting the number of edges.

When an instance could not be solved to optimality by any of the methods we consider, we report the optimality gap percentage, which is calculated as $\frac{UB-LB}{UB} \times 100$ with UB and LB denoting the upper and lower bounds respectively, to give an indication of how far a feasible solution is away from an optimal solution.

For each one of the three methods, we set a time limit of 1200 seconds throughout all the experiments. When an instance could not be solved optimally within the limit, the solution time of that instance is taken as 1200 seconds. In our experiments, the B&P algorithm by Furini *et al.* failed to report optimality gaps for instances that could not be solved optimally within the time limit. For such cases, we take the optimality gap as 100%.

To improve the performance of our decomposition approach for permutation graphs and generalized split graphs, we applied some initial treatment before starting the solution procedure. In particular, we initially searched for maximal cliques in the entire graph by using the algorithms explained in Sections 5.1.1-5.1.2 and added them as clique constraints to the master program beforehand. Since the intersection of a maximal clique in the entire graph with a selection gives a selective clique (not necessarily maximal), clique constraints generated on the entire graph serve as valid inequalities. We also applied a heuristic method to find a selection, found a maximum clique in it and used this as an initial solution to be used as upper bound. Our heuristic method chooses a vertex from each cluster that has the fewest number of neighbors in other clusters. These initial treatments are also adapted to the IP formulation Model 2 in order to compare it to our decomposition method on equal terms.

7.1. Experimental Results for Permutation Graphs

In this section, we test the performance of the IP formulation, the B&P algorithm by Furini *et al.* [30] and our decomposition approach on permutation graphs. Table 7.1 summarizes the computational results for permutation graph instances with cluster sizes varying between two and five. The first three columns in this table present the number of vertices ("n"), average edge density ("Avg density"), and average number of clusters ("Avg # clust") across ten random instances. In the next three groups of columns, we report the results of our experiments for the three algorithms under "IP formulation", "B&P", and "Decomposition" headings, respectively. For the IP formulation, columns 4–8 show the number of instances that could be optimally solved among ten ("# opt"), average optimality gap percentages over instances that could not be solved to optimality within the given time limit of 1200 seconds ("Avg % gap in nonopt") and over all instances ("Avg % gap overall"), average solution time in seconds over instances that are optimally solved ("Avg time in opt") and over all instances ("Avg time overall"). Columns 9–13 and 14–18 list the same set of results respectively for B&P of Furini *et al.* and our decomposition method. Finally, the rightmost column shows the average time spent in the subproblem of our decomposition algorithm in

seconds across ten instances ("Avg time in subpr"). In each row of this table, we report average values across ten independent runs. The upper half of the table shows the results for low density values 0.1 and 0.3, and the lower half for high density values 0.5 and 0.7. At the end of each half, total number of instances solved to optimality, and average time and optimality gaps are reported for each one of the three methods.

Our observation from the results listed in Table 7.1 is that in permutation graphs with low density, i.e., 0.1 and 0.3, the decomposition algorithm clearly outperforms the IP formulation and B&P method in terms of the number of instances solved to optimality, average optimality gap, and average amount of time spent on instances that are solved to optimality and in general. In permutation graphs with high density, i.e., 0.5 and 0.7, performance of the IP formulation and our method worsen in general, whereas that of B&P improves. As n grows, the performance of all three methods deteriorate in general. Increasing edge density, however, results in improved performance for B&P method, while deteriorating that of the IP formulation and the decomposition method. The results in the lower half of Table 7.1 indicate that the decomposition method outperforms the other two in terms of average optimality gap and average time spent in instances that could be solved optimally. However, although the decomposition method outperforms the IP formulation in every aspect, B&P is able to solve the highest number of instances to optimality.

We conducted two additional sets of experiments on permutation graphs in order to test the effect of cluster sizes. Tables 7.2–7.3 show the results obtained on the same set of graphs as before but with cluster sizes between 4–7 and 6–9, respectively. Comparing the results in Table 7.1 to those in Tables 7.2–7.3, we observe considerable improvement in the performance of all methods. For a given n value, as average size of clusters increase, the total number of clusters and hence the number of variables and constraints in Model 2 reduce. This reduction in the size of the IP formulation leads to improved performance. Comparative performance of the three methods is similar to the case with small clusters; the decomposition method outperforms the other two in all respects for low density graphs, while B&P manages to yield the highest number of

Table 7.1. Experimental results for permutation graph instances with small clusters.

n	Avg density	Avg # clust	IP formulation					B&P					Decomposition					
			# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg time in subpr
100	0.103	28.7	10		0.00	0.63	0.63	10		0.00	0.77	0.77	10		0.00	0.23	0.23	0.03
	0.300	27.2	10		0.00	4.83	4.83	10		0.00	35.24	35.24	10		0.00	0.32	0.32	0.05
200	0.102	56.3	10		0.00	8.23	8.23	9	100.00	10.00	0.48	120.43	10		0.00	0.16	0.16	0.03
	0.304	56.5	10		0.00	207.08	207.08	10		0.00	260.33	260.33	10		0.00	0.64	0.64	0.14
300	0.103	85.7	10		0.00	36.11	36.11	9	100.00	10.00	380.17	462.15	10		0.00	0.18	0.18	0.05
	0.309	85.0	0	90.45	90.45		1200.00	3	100.00	70.00	1071.58	1161.47	10		0.00	7.05	7.05	0.65
400	0.105	115.2	10		0.00	328.36	328.36	2	100.00	80.00	3.91	960.78	10		0.00	0.30	0.30	0.12
	0.302	112.7	0	96.25	96.25		1200.00	0	100.00	100.00		1200.00	10		0.00	33.32	33.32	2.51
500	0.101	141.1	7	80.46	24.14	383.44	628.41	1	100.00	90.00	14.39	1081.44	10		0.00	0.26	0.26	0.12
	0.306	140.4	0	99.05	99.05		1200.00	0	100.00	100.00		1200.00	8	20.00	4.00	236.39	429.11	9.12
			67	**93.90**	**30.99**	**138.39**	**481.37**	**54**	**100.00**	**46.00**	**220.86**	**648.26**	**98**	**20.00**	**0.40**	**27.89**	**47.16**	**1.28**
100	0.500	29.4	10		0.00	14.08	14.08	10		0.00	19.92	19.92	10		0.00	0.56	0.56	0.07
	0.707	28.1	10		0.00	51.46	51.46	10		0.00	18.35	18.35	10		0.00	5.97	5.97	0.13
200	0.497	55.9	7	76.19	22.86	624.69	797.29	10		0.00	169.24	169.24	10		0.00	100.30	100.30	1.57
	0.703	57.5	0	86.91	86.91		1200.00	10		0.00	116.00	116.00	3	14.29	10.00	276.96	923.09	3.35
300	0.499	85.5	0	98.04	98.04		1200.00	10		0.00	489.77	489.77	4	20.56	12.33	136.91	774.77	7.55
	0.701	85.2	0	100.00	100.00		1200.00	10		0.00	360.35	360.35	0	40.95	40.95		1200.00	29.32
400	0.497	114.9	0	100.00	100.00		1200.00	4	100.00	60.00	1028.71	1131.48	0	24.38	24.38		1200.00	22.11
	0.698	114.9	0	100.00	100.00		1200.00	10		0.00	799.98	799.98	0	48.84	48.84		1200.00	24.80
500	0.502	141.9	0	100.00	100.00		1200.00	0	100.00	100.00		1200.00	0	34.64	34.64		1200.00	31.75
	0.700	142.8	0	100.00	100.00		1200.00	0	100.00	100.00		1200.00	0	62.01	62.01		1200.00	33.56
			27	**96.96**	**70.78**	**230.08**	**926.28**	**74**	**100.00**	**26.00**	**375.29**	**550.51**	**37**	**37.01**	**23.32**	**104.14**	**780.47**	**15.42**

Table 7.2. Experimental results for permutation graph instances with medium-sized clusters.

			IP formulation					B&P					Decomposition					
n	Avg density	Avg # clust	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg time in subpr
100	0.103	17.8	10		0.00	0.42	0.42	10		0.00	0.03	0.03	10		0.00	0.24	0.24	0.03
	0.300	18.1	10		0.00	1.70	1.70	10		0.00	1.93	1.93	10		0.00	0.25	0.25	0.04
200	0.102	36.6	10		0.00	5.21	5.21	10		0.00	0.21	0.21	10		0.00	0.22	0.22	0.04
	0.304	36.3	10		0.00	37.99	37.99	10		0.00	45.15	45.15	10		0.00	0.50	0.50	0.10
300	0.103	54.9	10		0.00	15.37	15.37	10		0.00	1.23	1.23	10		0.00	0.15	0.15	0.03
	0.309	54.3	6	70.83	28.33	416.25	729.75	10		0.00	289.44	289.44	10		0.00	1.63	1.63	0.25
400	0.105	72.6	10		0.00	56.19	56.19	10		0.00	3.02	3.02	10		0.00	0.22	0.22	0.09
	0.302	72.3	0	84.07	84.07		1200.00	8	100.00	20.00	967.45	1013.96	10		0.00	4.12	4.12	0.59
500	0.101	90.3	10		0.00	117.40	117.40	10		0.00	12.43	12.43	10		0.00	0.28	0.28	0.13
	0.306	89.6	0	93.71	93.71		1200.00	0	100.00	100.00		1200.00	10		0.00	30.47	30.47	1.95
			76	**85.88**	**20.61**	**81.32**	**336.40**	**88**	**100.00**	**12.00**	**146.76**	**256.74**	**100**	**-**	**0.00**	**3.81**	**3.81**	**0.32**
100	0.500	18.4	10		0.00	6.04	6.04	10		0.00	8.76	8.76	10		0.00	0.39	0.39	0.06
	0.707	18.1	10		0.00	21.97	21.97	10		0.00	11.49	11.49	10		0.00	3.50	3.50	0.10
200	0.497	35.6	9	100.00	10.00	194.64	295.18	10		0.00	98.17	98.17	10		0.00	19.54	19.54	0.59
	0.703	35.8	7	75.00	22.50	565.28	755.70	10		0.00	80.88	80.88	2	25.00	20.00	600.30	1080.06	2.90
300	0.499	54.3	0	97.89	97.89		1200.00	10		0.00	376.98	376.98	7	33.33	10.00	331.44	592.01	3.04
	0.701	54.2	0	100.00	100.00		1200.00	10		0.00	277.88	277.88	0	51.38	51.38		1200.00	26.10
400	0.497	72.5	0	100.00	100.00		1200.00	10		0.00	981.12	981.12	3	45.48	31.83	669.48	1040.84	17.85
	0.698	72.1	0	100.00	100.00		1200.00	10		0.00	658.14	658.14	0	65.82	65.82		1200.00	32.29
500	0.502	89.2	0	98.71	98.71		1200.00	0	100.00	100.00		1200.00	0	46.00	46.00		1200.00	18.31
	0.700	90.4	0	100.00	100.00		1200.00	0	100.00	100.00		1200.00	0	72.79	72.79		1200.00	37.94
			36	**98.30**	**62.91**	**196.98**	**827.89**	**80**	**100.00**	**20.00**	**311.68**	**489.34**	**42**	**51.35**	**29.78**	**270.77**	**753.63**	**13.92**

Table 7.3. Experimental results for permutation graph instances with large clusters.

n	Avg density	Avg # clust	IP formulation					B&P					Decomposition					
			# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg time in subpr
100	0.103	13.4	10		0.00	0.17	0.17	10		0.00	0.03	0.03	10		0.00	0.10	0.10	0.01
	0.300	12.8	10		0.00	0.60	0.60	10		0.00	0.04	0.04	10		0.00	0.12	0.12	0.02
200	0.102	26.3	10		0.00	2.73	2.73	10		0.00	0.09	0.09	10		0.00	0.12	0.12	0.02
	0.304	26.5	10		0.00	7.04	7.04	10		0.00	0.28	0.28	10		0.00	0.24	0.24	0.06
300	0.103	39.7	10		0.00	14.10	14.10	10		0.00	0.82	0.82	10		0.00	0.13	0.13	0.03
	0.309	40.0	10		0.00	217.55	217.55	10		0.00	17.20	17.20	10		0.00	0.72	0.72	0.26
400	0.105	53.6	10		0.00	33.01	33.01	10		0.00	2.40	2.40	10		0.00	0.25	0.25	0.10
	0.302	52.8	8	100.00	20.00	217.56	414.05	10		0.00	242.04	242.04	10		0.00	6.53	6.53	0.98
500	0.101	66.4	10		0.00	61.27	61.27	10		0.00	5.38	5.38	10		0.00	0.25	0.25	0.10
	0.306	65.9	7	77.78	23.33	837.36	946.15	9	100.00	10.00	287.90	379.11	10		0.00	2.35	2.35	0.82
			95	**86.67**	**4.33**	**139.14**	**169.67**	**99**	**100.00**	**1.00**	**55.62**	**64.74**	**100**	**-**	**0.00**	**1.08**	**1.08**	**0.24**
100	0.500	13.0	10		0.00	1.94	1.94	10		0.00	0.30	0.30	10		0.00	0.17	0.17	0.03
	0.707	13.1	10		0.00	10.36	10.36	10		0.00	6.49	6.49	10		0.00	1.13	1.13	0.07
200	0.497	26.4	9	100.00	10.00	76.90	189.21	10		0.00	49.18	49.18	10		0.00	10.36	10.36	0.34
	0.703	26.4	8	83.33	16.67	313.96	491.17	10		0.00	45.06	45.06	7	38.89	11.67	414.58	650.21	1.94
300	0.499	39.0	3	86.90	60.83	830.53	1089.16	10		0.00	196.23	196.23	10		0.00	104.04	104.04	3.22
	0.701	39.6	0	100.00	100.00		1200.00	10		0.00	166.37	166.37	0	54.33	54.33		1200.00	22.88
400	0.497	52.9	0	95.76	95.76		1200.00	10		0.00	536.61	536.61	5	40.00	20.00	222.86	711.43	15.96
	0.698	53.9	0	100.00	100.00		1200.00	10		0.00	459.61	459.61	0	68.88	68.88		1200.00	29.59
500	0.502	66.9	0	97.50	97.50		1200.00	5	100.00	50.00	1151.27	1175.64	0	39.17	39.17		1200.00	15.73
	0.700	66.3	0	100.00	100.00		1200.00	10		0.00	794.16	794.16	0	75.42	75.42		1200.00	56.99
			40	**96.79**	**58.08**	**246.74**	**778.18**	**95**	**100.00**	**5.00**	**340.53**	**342.97**	**52**	**56.14**	**26.95**	**125.52**	**627.73**	**14.68**

optimally solved instances. The average time that the decomposition method spends for optimally solved instances is again significantly lower than those of the other two, regardless of edge density.

Before finalizing this section, we provide a brief summary of the results reported in Tables 7.1–7.3. The first two columns of Table 7.4 list the cluster sizes ("Sizes of clusters") as small, medium, or large, and the edge density of graphs ("Density") as low or high, where 0.1 and 0.3 are referred to as low densities, and 0.5 and 0.7 as high densities as before. The next three sets of columns report the total number of instances solved optimally ("# opt"), overall average of percentage optimality gap ("Avg % gap"), and overall average of solution time ("Avg time") for each one of the three methods, respectively, considering all of the 600 problem instances. Finally, the bottom row of the table lists the total number of instances solved to optimality and the averages of % gap and solution time for each method over a total of 600 instances.

Table 7.4. Summary of experimental results for all permutation graph instances.

		IP formulation			**B&P**			**Decomposition**		
Sizes of clusters	**Density**	**# opt**	**Avg % gap**	**Avg time**	**# opt**	**Avg % gap**	**Avg time**	**# opt**	**Avg % gap**	**Avg time**
small	low	67	30.99	481.37	54	46.00	648.26	98	0.40	47.16
	high	27	70.78	926.28	74	26.00	550.51	37	23.32	780.47
	all	**94**	**50.88**	**703.82**	**128**	**36.00**	**599.39**	**135**	**11.86**	**413.81**
medium	low	76	20.61	336.40	88	12.00	256.74	100	0.00	3.81
	high	36	62.91	827.89	80	20.00	489.34	42	29.78	753.63
	all	**112**	**41.76**	**582.15**	**168**	**16.00**	**373.04**	**142**	**14.89**	**378.72**
large	low	95	4.33	169.67	99	1.00	64.74	100	0.00	1.08
	high	40	58.08	778.18	95	5.00	342.97	52	26.95	627.73
	all	**135**	**31.20**	**473.93**	**194**	**3.00**	**203.85**	**152**	**13.47**	**314.41**
		341	**41.28**	**586.63**	**490**	**18.33**	**392.09**	**429**	**13.41**	**368.98**

As far as the percentage of optimally solved instances is concerned, the B&P method can solve about 82% of all instances, while it respectively drops to 72% and 57% for the decomposition method and the IP formulation. In terms of the average optimality gap and solution time, however, the decomposition algorithm outperforms the other two methods. Our algorithm yields the best results in all respects in low-density instances.

7.2. Experimental Results for Generalized Split Graphs

The second family of perfect graphs we conduct experiments on is generalized split graphs. We present the results of our computational experiments in Tables 7.5–7.7 in the same order and structure as in the case of permutation graphs.

We observe from the upper half of Table 7.5 that for relatively low-density graphs, the IP formulation was able to solve a slightly larger percentage of test instances to optimality as compared to the decomposition method. However, the decomposition method yields better results in terms of all other measures compared to both methods. As for the results obtained from high density instances shown in the lower half of the table, B&P is able to solve slightly higher number of instances optimally; but, decomposition approach is able to result in better average optimality gap percentages and average times.

The performance of all three methods deteriorate in general as n increases. As in the case of permutation graphs, we conducted two additional sets of experiments to see how cluster sizes affect the performance of the algorithms. When we compare the results in Table 7.5 to those in Tables 7.6–7.7, we observe enhanced performance in the IP formulation with the increase in cluster sizes due to reduction in the size of the IP formulation. As far as the number of instances solved to optimality is concerned, the decomposition method produces more robust results. As cluster sizes increase, performance of B&P also improves in low-density instances, but we cannot observe a monotonic trend in high-density instances.

As in the case of permutation graphs, we want to conclude this section with a synopsis of our experimental results. Table 7.8 puts together a brief summary of results presented in Tables 7.5–7.7, and has the same structure as Table 7.4 for permutation graphs. The B&P method is able to yield a highest number of optimally solved instances in total; however, the decomposition method yields the best results in terms of the average optimality gap and solution time, as in the case of permuta-

Table 7.5. Experimental results for generalized split graph instances with small clusters.

			IP formulation					B&P					Decomposition					
n	Avg density	Avg # clust	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg time in subpr
100	0.101	29.9	10		0.00	0.74	0.74	10		0.00	15.53	15.53	10		0.00	0.30	0.30	0.03
	0.299	28.7	10		0.00	4.09	4.09	10		0.00	22.38	22.38	9	14.29	1.43	0.31	120.28	0.21
200	0.101	58.4	10		0.00	4.42	4.42	10		0.00	119.30	119.30	7	25.00	7.50	4.78	363.34	1.13
	0.300	58.3	10		0.00	78.86	78.86	10		0.00	173.60	173.60	7	13.10	3.93	0.46	360.32	33.17
300	0.102	86.1	10		0.00	38.95	38.95	9	100.00	10.00	432.26	509.03	7	23.33	7.00	100.29	430.21	8.60
	0.300	85.3	8	87.50	17.50	298.75	479.00	8	100.00	20.00	475.71	620.57	6	24.79	9.92	1.13	480.68	29.86
400	0.101	115.9	10		0.00	163.67	163.67	6	100.00	40.00	705.57	903.34	8	22.50	4.50	36.83	269.46	4.27
	0.300	114.2	2	60.21	48.17	739.80	1107.96	5	100.00	50.00	1067.11	1133.55	7	22.22	6.67	4.45	363.11	27.49
500	0.101	143.5	9	40.00	4.00	363.05	446.75	2	100.00	80.00	944.11	1148.82	7	27.50	8.25	0.77	360.54	51.78
	0.300	144.0	1	94.60	85.14	1153.10	1195.31	0	100.00	100.00		1200.00	7	26.01	7.80	3.68	362.57	42.60
			80	**77.40**	**15.48**	**284.54**	**351.98**	**70**	**100.00**	**30.00**	**439.51**	**584.61**	**75**	**22.80**	**5.70**	**15.30**	**311.08**	**19.91**
100	0.497	28.8	10		0.00	7.30	7.30	10		0.00	17.04	17.04	9	25.00	2.50	2.07	121.86	9.56
	0.701	29.8	10		0.00	17.78	17.78	10		0.00	17.48	17.48	9	21.43	2.14	1.00	120.90	5.92
200	0.504	58.0	8	83.33	16.67	172.99	378.39	10		0.00	157.56	157.56	8	22.18	4.44	0.52	240.42	13.85
	0.700	57.5	7	52.78	15.83	355.95	609.17	10		0.00	109.51	109.51	9	21.43	2.14	32.52	149.27	13.04
300	0.501	87.5	6	72.78	29.11	368.70	701.22	10		0.00	338.01	338.01	9	25.00	2.50	140.84	246.76	11.50
	0.695	85.2	5	100.00	50.00	152.55	676.28	10		0.00	377.56	377.56	5	19.63	9.82	18.81	609.41	10.00
400	0.500	114.9	0	69.22	69.22		1200.00	9	100.00	10.00	739.04	785.13	7	21.13	6.34	11.85	368.30	13.59
	0.699	115.8	1	79.94	71.95	192.23	1099.22	8	100.00	20.00	689.02	791.21	3	18.55	12.98	0.47	840.14	28.21
500	0.500	142.4	0	96.85	96.85		1200.00	5	100.00	50.00	1120.67	1160.33	5	24.75	12.37	24.47	612.23	43.72
	0.700	143.8	1	82.41	74.17	949.35	1174.94	4	100.00	60.00	834.08	1053.63	2	21.51	17.21	0.79	960.16	139.41
			48	**81.50**	**42.38**	**277.11**	**706.43**	**86**	**100.00**	**14.00**	**440.00**	**480.75**	**66**	**21.31**	**7.24**	**23.34**	**426.94**	**28.88**

Table 7.6. Experimental results for generalized split graph instances with medium-sized clusters.

n	Avg density	Avg # clust	IP formulation					B&P					Decomposition					
			# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg time in subpr
100	0.101	18.2	10		0.00	0.25	0.25	10		0.00	2.84	2.84	10		0.00	1.95	1.95	0.03
	0.299	18.2	10		0.00	1.35	1.35	10		0.00	3.84	3.84	9	20.00	2.00	1.53	121.38	3.17
200	0.101	36.4	10		0.00	3.46	3.46	10		0.00	28.07	28.07	10		0.00	2.25	2.25	0.08
	0.300	36.5	10		0.00	16.96	16.96	10		0.00	77.64	77.64	8	19.64	3.93	74.15	299.32	7.78
300	0.102	54.6	10		0.00	16.57	16.57	10		0.00	94.93	94.93	6	41.67	16.67	5.11	483.07	10.35
	0.300	54.3	9	100.00	10.00	140.88	246.79	10		0.00	449.18	449.18	7	44.29	13.29	129.46	450.62	18.90
400	0.101	73.4	10		0.00	77.94	77.94	10		0.00	294.48	294.48	8	33.33	6.67	45.20	276.16	6.87
	0.300	73.3	8	75.00	15.00	456.11	604.89	4	100.00	60.00	757.85	1023.14	7	33.33	10.00	5.62	363.93	14.77
500	0.101	90.5	10		0.00	231.34	231.34	6	100.00	40.00	79.93	527.96	7	47.78	14.33	19.10	373.37	19.61
	0.300	90.5	3	56.62	39.63	496.33	988.90	1	100.00	90.00	607.83	1140.78	6	38.87	15.55	2.16	481.30	37.40
			90	**64.63**	**6.46**	**144.12**	**218.85**	**81**	**100.00**	**19.00**	**239.66**	**364.28**	**78**	**37.47**	**8.24**	**28.65**	**285.34**	**11.90**
100	0.497	18.2	10		0.00	4.61	4.61	10		0.00	7.24	7.24	9	40.00	4.00	14.14	132.73	7.82
	0.701	17.7	10		0.00	5.01	5.01	10		0.00	28.99	28.99	9	33.33	3.33	0.59	120.53	13.03
200	0.601	36.8	10		0.00	117.66	117.66	10		0.00	89.12	89.12	8	40.87	8.17	120.36	336.29	12.67
	0.700	36.8	10		0.00	167.16	167.16	10		0.00	147.37	147.37	4	5.56	3.33	78.69	751.48	7.32
300	0.501	54.5	10		0.00	166.49	166.49	10		0.00	263.47	263.47	6	31.88	12.75	24.38	494.63	19.65
	0.695	54.5	5		0.00	53.06	626.53	6	100.00	40.00	766.78	940.07	7	21.43	6.43	41.96	389.37	3.23
400	0.500	72.4	5	90.00	45.00	545.79	872.90	7	100.00	30.00	823.50	936.45	6	37.80	15.12	69.57	521.74	20.63
	0.699	71.9	3	89.66	62.76	439.65	971.89	1	100.00	90.00	797.84	1159.78	3	26.39	18.47	27.63	848.29	20.18
500	0.500	91.3	0	97.62	97.62		1200.00	2	100.00	80.00	1022.11	1164.42	6	40.16	16.06	88.63	533.18	40.06
	0.700	90.5	2	87.50	70.00	536.96	1067.39	1	100.00	90.00	883.85	1168.38	1	32.00	28.80	1.22	1080.12	43.82
			65	**78.68**	**27.54**	**226.27**	**519.97**	**67**	**100.00**	**33.00**	**483.03**	**590.53**	**59**	**28.41**	**11.65**	**46.72**	**520.84**	**18.84**

Table 7.7. Experimental results for generalized split graph instances with large clusters.

n	Avg density	Avg # clust	IP formulation					B&P					Decomposition					
			# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg time in subpr
100	0.101	12.7	10		0.00	0.25	0.25	10		0.00	0.04	0.04	10		0.00	0.13	0.13	0.01
	0.299	13.2	10		0.00	0.71	0.71	10		0.00	5.02	5.02	9	25.00	2.50	0.77	120.69	0.78
200	0.101	26.1	10		0.00	1.08	1.08	10		0.00	24.56	24.56	8	50.00	10.00	25.81	260.65	0.18
	0.300	26.3	10		0.00	5.65	5.65	10		0.00	56.00	56.00	9	20.00	2.00	101.72	211.55	1.80
300	0.102	39.8	10		0.00	8.53	8.53	10		0.00	149.38	149.38	7	38.89	11.67	23.90	376.73	6.53
	0.300	39.6	10		0.00	40.41	40.41	10		0.00	205.34	205.34	7	38.89	11.67	11.87	368.31	12.72
400	0.101	53.4	10		0.00	29.49	29.49	9	100.00	10.00	466.75	540.08	6	50.00	20.00	104.94	542.96	1.24
	0.300	53.4	10		0.00	216.67	216.67	7	100.00	30.00	610.20	787.14	5	40.00	20.00	5.34	602.67	9.37
500	0.101	66.8	10		0.00	125.22	125.22	9	100.00	10.00	362.52	446.27	6	50.00	20.00	4.86	482.92	19.58
	0.300	65.9	9	50.00	5.00	418.26	496.43	9	100.00	10.00	368.12	451.31	6	48.33	19.33	17.41	490.45	26.89
			99	**50.00**	**0.50**	**84.63**	**92.44**	**94**	**100.00**	**6.00**	**224.79**	**266.51**	**73**	**43.40**	**11.72**	**29.67**	**345.70**	**7.91**
100	0.497	13.2	10		0.00	1.33	1.33	10		0.00	7.77	7.77	9	25.00	2.50	14.59	133.14	2.55
	0.701	13.1	10		0.00	2.52	2.52	10		0.00	15.33	15.33	9	28.57	2.86	0.45	120.41	14.15
200	0.504	26.6	10		0.00	12.68	12.68	10		0.00	86.15	86.15	8	45.00	9.00	1.15	240.95	9.50
	0.700	26.7	10		0.00	41.60	41.60	10		0.00	156.95	156.95	8	41.43	8.29	140.78	352.63	12.28
300	0.501	39.6	10		0.00	73.63	73.63	10		0.00	373.93	373.93	6	37.50	15.00	0.61	480.37	10.21
	0.695	39.6	5	60.00	30.00	59.51	629.75	10		0.00	674.24	674.24	6	25.42	10.17	95.40	537.24	7.04
400	0.500	53.1	7	61.11	18.33	295.64	566.95	5	100.00	50.00	956.08	1078.04	6	40.00	16.00	14.42	488.65	14.85
	0.699	52.8	4	85.09	51.06	449.67	899.87	10		0.00	497.87	497.87	5	35.00	17.50	257.89	728.94	22.04
500	0.500	67.4	1	82.91	74.62	621.18	1142.12	4	100.00	60.00	741.31	1016.52	7	55.71	16.71	24.72	377.30	34.67
	0.700	66.7	3	85.71	60.00	370.23	951.07	6	100.00	40.00	565.94	819.56	1	37.26	33.54	0.07	1080.01	32.82
			70	**78.00**	**23.40**	**192.80**	**432.15**	**85**	**100.00**	**15.00**	**407.56**	**472.64**	**65**	**37.59**	**13.16**	**55.01**	**453.96**	**16.01**

Table 7.8. Summary of experimental results for all generalized split graph instances.

Sizes of clusters	Density	IP formulation # opt	IP formulation Avg % gap	IP formulation Avg time	B&P # opt	B&P Avg % gap	B&P Avg time	Decomposition # opt	Decomposition Avg % gap	Decomposition Avg time
small	low	80	15.48	351.98	70	30.00	584.61	76	5.50	310.33
	high	48	42.38	706.43	86	14.00	480.75	66	7.58	427.44
	all	**128**	**28.93**	**529.20**	**156**	**22.00**	**532.68**	**142**	**6.54**	**368.88**
medium	low	90	6.46	218.85	81	19.00	364.28	77	9.03	287.05
	high	65	27.54	519.97	67	33.00	590.53	64	11.63	454.85
	all	**155**	**17.00**	**369.41**	**148**	**26.00**	**477.41**	**141**	**10.33**	**370.95**
large	low	99	0.50	92.44	94	6.00	266.51	72	12.55	364.49
	high	70	23.40	432.15	85	15.00	472.64	66	12.66	431.26
	all	**169**	**11.95**	**262.30**	**179**	**10.50**	**369.58**	**138**	**12.60**	**397.88**
		452	**19.29**	**386.97**	**483**	**19.50**	**459.89**	**421**	**9.82**	**379.24**

tion graphs. Average optimality gap of the decomposition method is about half of the average optimality gaps that the other two methods yield.

7.3. Experimental Results for Chordal Graphs

The class of chordal graphs is the last subclass of perfect graphs we experiment with. Tables 7.9–7.14 show the results for small, medium and large cluster sizes and for low and high edge densities, respectively. The general structure of these tables is the same as Tables 7.1–7.7, except that we additionally include the average number of maximal cliques across ten graph instances ("Avg # cliques"), which is an important indicator of how well the algorithm performs, and exclude the column for time spent in subproblem.

All the chordal graph instances are solved to optimality under one second by our decomposition approach, as the results in Tables 7.9–7.14 reveal. Here, we experimented with graphs up to 1000 vertices in order to be able to clearly demonstrate the gap between performances of the three algorithms. For a given n, the solution times of our algorithm decrease as edge density increases due to a parallel decline in the number of maximal cliques. The IP formulation can also solve a large percentage of chordal graph instances to optimality within the permitted time limit; yet, our method clearly outperformed it time-wise. When the clusters in the partition are small-sized, IP failed

Table 7.9. Experimental results for chordal graph instances of density 0.1 and 0.3 with small clusters.

n	Avg density	Avg # clust	Avg # cliques	IP formulation # opt	IP formulation Avg % gap in nonopt	IP formulation Avg % gap overall	IP formulation Avg time in opt	IP formulation Avg time overall	B&P # opt	B&P Avg % gap in nonopt	B&P Avg % gap overall	B&P Avg time in opt	B&P Avg time overall	Decomposition # opt	Decomposition Avg % gap in nonopt	Decomposition Avg % gap overall	Decomposition Avg time in opt	Decomposition Avg time overall
100	0.103	29.3	38.2	10		0.00	0.95	0.95	10		0.00	19.22	19.22	10		0.00	0.11	0.11
	0.299	28.3	25.0	10		0.00	1.26	1.26	10		0.00	15.75	15.75	10		0.00	0.15	0.15
200	0.106	56.6	59.2	10		0.00	4.35	4.35	10		0.00	156.28	156.28	10		0.00	0.16	0.16
	0.306	58.0	37.8	10		0.00	4.00	4.00	10		0.00	83.80	83.80	10		0.00	0.13	0.13
300	0.108	86.4	75.8	10		0.00	11.45	11.45	10		0.00	482.17	482.17	10		0.00	0.22	0.22
	0.298	86.1	48.3	10		0.00	18.57	18.57	10		0.00	252.91	252.91	10		0.00	0.18	0.18
400	0.101	113.8	96.3	10		0.00	28.19	28.19	8	100.00	20.00	1052.36	1081.88	10		0.00	0.21	0.21
	0.304	113.8	58.0	10		0.00	37.28	37.28	10		0.00	608.79	608.79	10		0.00	0.17	0.17
500	0.101	143.0	107.9	10		0.00	74.68	74.68	7	100.00	30.00	996.29	1057.40	10		0.00	0.17	0.17
	0.300	140.9	68.3	10		0.00	93.44	93.44	8	100.00	20.00	858.92	927.14	10		0.00	0.13	0.13
600	0.103	173.5	124.0	10		0.00	142.78	142.78	0	100.00	100.00		1200.00	10		0.00	0.36	0.36
	0.304	172.0	76.6	10		0.00	205.08	205.08	0	100.00	100.00		1200.00	10		0.00	0.17	0.17
700	0.101	197.7	130.6	10		0.00	342.65	342.65	0	100.00	100.00		1200.00	10		0.00	0.25	0.25
	0.299	200.2	84.3	10		0.00	389.96	389.96	0	100.00	100.00		1200.00	10		0.00	0.18	0.18
800	0.103	229.0	147.4	10		0.00	566.86	566.86	0	100.00	100.00		1200.00	10		0.00	0.26	0.26
	0.304	232.3	90.6	10		0.00	583.48	583.48	0	100.00	100.00		1200.00	10		0.00	0.19	0.19
900	0.099	257.4	156.2	8	96.81	19.36	652.82	762.26	0	100.00	100.00		1200.00	10		0.00	0.29	0.29
	0.299	255.5	98.6	7	91.86	27.56	1145.15	1161.61	0	100.00	100.00		1200.00	10		0.00	0.22	0.22
1000	0.105	286.2	166.1	6	97.29	38.92	829.80	977.88	0	100.00	100.00		1200.00	10		0.00	0.32	0.32
	0.297	285.9	107.2	9	93.29	9.33	938.76	964.89	0	100.00	100.00		1200.00	10		0.00	0.23	0.23
				190	**95.17**	**4.76**	**303.57**	**318.58**	**93**	**100.00**	**53.50**	**452.65**	**834.27**	**200**	**-**	**0.00**	**0.20**	**0.20**

Table 7.10. Experimental results for chordal graph instances of density 0.5 and 0.7 with small clusters.

				IP formulation					B&P					Decomposition				
n	Avg density	Avg # clust	Avg # cliques	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall
100	0.494	28.7	19.8	10		0.00	1.17	1.17	10		0.00	11.76	11.76	10		0.00	0.16	0.16
	0.699	28.8	15.1	10		0.00	1.25	1.25	10		0.00	11.14	11.14	10		0.00	0.15	0.15
200	0.495	57.8	28.9	10		0.00	5.16	5.16	10		0.00	69.63	69.63	10		0.00	0.14	0.14
	0.695	57.2	22.0	10		0.00	6.19	6.19	10		0.00	61.43	61.43	10		0.00	0.09	0.09
300	0.501	86.6	39.1	10		0.00	18.57	18.57	10		0.00	201.03	201.03	10		0.00	0.11	0.11
	0.706	86.7	27.0	10		0.00	23.54	23.54	10		0.00	181.26	181.26	10		0.00	0.09	0.09
400	0.499	114.8	46.1	10		0.00	41.88	41.88	10		0.00	505.76	505.76	10		0.00	0.12	0.12
	0.697	114.7	34.2	10		0.00	60.69	60.69	10		0.00	444.91	444.91	10		0.00	0.10	0.10
500	0.503	143.3	54.1	10		0.00	124.58	124.58	5	100.00	50.00	1012.02	1106.01	10		0.00	0.13	0.13
	0.705	141.7	40.6	10		0.00	141.98	141.98	0	100.00	100.00		1200.00	10		0.00	0.13	0.13
600	0.505	172.1	59.7	10		0.00	260.72	260.72	0	100.00	100.00		1200.00	10		0.00	0.14	0.14
	0.704	172.7	44.4	10		0.00	325.45	325.45	0	100.00	100.00		1200.00	10		0.00	0.12	0.12
700	0.496	200.2	64.5	10		0.00	432.24	432.24	0	100.00	100.00		1200.00	10		0.00	0.15	0.15
	0.704	200.1	49.1	10		0.00	695.57	695.57	0	100.00	100.00		1200.00	10		0.00	0.13	0.13
800	0.497	226.6	67.8	10		0.00	723.77	723.77	0	100.00	100.00		1200.00	10		0.00	0.16	0.16
	0.698	229.2	50.9	7	70.21	21.06	1051.41	1095.98	0	100.00	100.00		1200.00	10		0.00	0.14	0.14
900	0.504	255.0	74.6	2	86.32	69.06	936.59	1147.32	0	100.00	100.00		1200.00	10		0.00	0.18	0.18
	0.699	255.5	55.5	0	94.95	94.95		1200.00	0	100.00	100.00		1200.00	10		0.00	0.15	0.15
1000	0.500	285.8	78.8	3	93.54	65.47	1577.90	1313.37	0	100.00	100.00		1200.00	10		0.00	0.19	0.19
	0.702	284.5	58.3	0	93.99	93.99		1200.00	0	100.00	100.00		1200.00	10		0.00	0.17	0.17
				162	**90.67**	**17.23**	**357.15**	**440.95**	**85**	**100.00**	**57.50**	**277.66**	**789.65**	**200**	**-**	**0.00**	**0.14**	**0.14**

Table 7.11. Experimental results for chordal graph instances of density 0.1 and 0.3 with medium-sized clusters.

				IP formulation					B&P					Decomposition				
n	Avg density	Avg # clust	Avg # cliques	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall
100	0.103	18.3	38.2	10		0.00	0.28	0.28	10		0.00	4.68	4.68	10		0.00	0.10	0.10
	0.299	17.7	25.0	10		0.00	0.44	0.44	10		0.00	33.71	33.71	10		0.00	0.09	0.09
200	0.106	36.9	59.2	10		0.00	1.32	1.32	10		0.00	223.93	223.93	10		0.00	0.12	0.12
	0.306	36.2	37.8	10		0.00	2.20	2.20	10		0.00	233.31	233.31	10		0.00	0.10	0.10
300	0.108	54.7	75.8	10		0.00	4.56	4.56	7	100.00	30.00	964.26	1034.98	10		0.00	0.15	0.15
	0.298	54.1	48.3	10		0.00	5.90	5.90	10		0.00	823.78	823.78	10		0.00	0.11	0.11
400	0.101	72.3	96.3	10		0.00	10.17	10.17	1	100.00	90.00	895.76	1169.58	10		0.00	0.18	0.18
	0.304	72.3	58.0	10		0.00	14.69	14.69	5	100.00	50.00	1063.91	1131.96	10		0.00	0.13	0.13
500	0.101	90.7	107.9	10		0.00	24.35	24.35	0	100.00	100.00		1200.00	10		0.00	0.20	0.20
	0.300	90.3	68.3	10		0.00	29.90	29.90	0	100.00	100.00		1200.00	10		0.00	0.15	0.15
600	0.103	108.6	124.0	10		0.00	49.68	49.68	0	100.00	100.00		1200.00	10		0.00	0.23	0.23
	0.304	108.8	76.6	10		0.00	55.76	55.76	0	100.00	100.00		1200.00	10		0.00	0.16	0.16
700	0.101	126.7	130.6	10		0.00	86.66	86.66	0	100.00	100.00		1200.00	10		0.00	0.23	0.23
	0.299	128.1	84.3	10		0.00	111.53	111.53	0	100.00	100.00		1200.00	10		0.00	0.18	0.18
800	0.103	146.3	147.4	10		0.00	119.76	119.76	0	100.00	100.00		1200.00	10		0.00	0.29	0.29
	0.304	144.2	90.6	10		0.00	160.95	160.95	0	100.00	100.00		1200.00	10		0.00	0.20	0.20
900	0.099	162.2	156.2	10		0.00	302.82	302.82	0	100.00	100.00		1200.00	10		0.00	0.32	0.32
	0.299	164.3	98.6	10		0.00	255.49	255.49	0	100.00	100.00		1200.00	10		0.00	0.22	0.22
1000	0.105	182.7	166.1	10		0.00	322.25	322.25	0	100.00	100.00		1200.00	10		0.00	0.34	0.34
	0.297	181.0	107.2	10		0.00	397.19	397.19	0	100.00	100.00		1200.00	10		0.00	0.25	0.25
				200	**-**	**0.00**	**97.80**	**97.80**	**63**	**100.00**	**68.50**	**530.42**	**952.80**	**200**	**-**	**0.00**	**0.19**	**0.19**

Table 7.12. Experimental results for chordal graph instances of density 0.5 and 0.7 with medium-sized clusters.

				IP formulation					B&P					Decomposition				
n	Avg density	Avg # clust	Avg # cliques	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall
100	0.494	18.1	19.8	10		0.00	0.55	0.55	10		0.00	32.04	32.04	10		0.00	0.08	0.08
	0.699	18.3	15.1	10		0.00	0.61	0.61	10		0.00	38.33	38.33	10		0.00	0.07	0.07
200	0.495	36.4	28.9	10		0.00	2.75	2.75	10		0.00	132.44	132.44	10		0.00	0.09	0.09
	0.695	36.9	22.0	10		0.00	3.26	3.26	10		0.00	105.92	105.92	10		0.00	0.09	0.09
300	0.501	55.7	39.1	10		0.00	9.16	9.16	10		0.00	730.09	730.09	10		0.00	0.10	0.10
	0.706	55.7	27.0	10		0.00	11.46	11.46	10		0.00	515.32	515.32	10		0.00	0.09	0.09
400	0.499	72.7	46.1	10		0.00	21.51	21.51	6	100.00	40.00	1043.58	1106.15	10		0.00	0.11	0.11
	0.697	72.8	34.2	10		0.00	29.58	29.58	0	100.00	100.00		1200.00	10		0.00	0.11	0.11
500	0.503	90.3	54.1	10		0.00	44.77	44.77	0	100.00	100.00		1200.00	10		0.00	0.13	0.13
	0.705	89.2	40.6	10		0.00	62.08	62.08	0	100.00	100.00		1200.00	10		0.00	0.11	0.11
600	0.505	109.6	59.7	10		0.00	83.57	83.57	0	100.00	100.00		1200.00	10		0.00	0.14	0.14
	0.704	107.7	44.4	10		0.00	116.59	116.59	0	100.00	100.00		1200.00	10		0.00	0.12	0.12
700	0.496	125.9	64.5	10		0.00	140.76	140.76	0	100.00	100.00		1200.00	10		0.00	0.15	0.15
	0.704	128.5	49.1	10		0.00	206.58	206.58	0	100.00	100.00		1200.00	10		0.00	0.13	0.13
800	0.497	145.8	67.8	10		0.00	223.10	223.10	0	100.00	100.00		1200.00	10		0.00	0.16	0.16
	0.698	146.9	50.9	10		0.00	360.06	360.06	0	100.00	100.00		1200.00	10		0.00	0.14	0.14
900	0.504	162.9	74.6	10		0.00	375.52	375.52	0	100.00	100.00		1200.00	10		0.00	0.18	0.18
	0.699	164.5	55.5	10		0.00	623.74	623.74	0	100.00	100.00		1200.00	10		0.00	0.16	0.16
1000	0.500	181.9	78.8	10		0.00	568.22	568.22	0	100.00	100.00		1200.00	10		0.00	0.20	0.20
	0.702	181.7	58.3	10		0.00	1109.45	1109.45	0	100.00	100.00		1200.00	10		0.00	0.17	0.17
				200	**-**	**0.00**	**199.67**	**199.67**	**66**	**100.00**	**67.00**	**371.10**	**913.01**	**200**	**-**	**0.00**	**0.13**	**0.13**

Table 7.13. Experimental results for chordal graph instances of density 0.1 and 0.3 with large clusters.

				IP formulation					B&P					Decomposition				
n	Avg density	Avg # clust	Avg # cliques	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall
100	0.103	13.1	38.2	10		0.00	0.20	0.20	10		0.00	4.68	4.68	10		0.00	0.10	0.10
	0.299	13.4	25.0	10		0.00	0.37	0.37	4	100.00	60.00	147.05	778.82	10		0.00	0.09	0.09
200	0.106	26.4	59.2	10		0.00	0.80	0.80	10		0.00	38.33	38.33	10		0.00	0.13	0.13
	0.306	26.7	37.8	10		0.00	1.67	1.67	10		0.00	223.93	223.93	10		0.00	0.10	0.10
300	0.108	39.5	75.8	10		0.00	2.25	2.25	10		0.00	105.92	105.92	10		0.00	0.15	0.15
	0.298	39.2	48.3	10		0.00	4.07	4.07	7	100.00	30.00	964.26	1034.98	10		0.00	0.11	0.11
400	0.101	53.2	96.3	10		0.00	4.67	4.67	10		0.00	515.32	515.32	10		0.00	0.20	0.20
	0.304	53.1	58.0	10		0.00	10.02	10.02	1	100.00	90.00	895.76	1169.58	10		0.00	0.12	0.12
500	0.101	66.6	107.9	10		0.00	8.53	8.53	0	100.00	100.00		1200.00	10		0.00	0.19	0.19
	0.300	66.9	68.3	10		0.00	19.96	19.96	0	100.00	100.00		1200.00	10		0.00	0.14	0.14
600	0.103	79.8	124.0	10		0.00	21.14	21.14	0	100.00	100.00		1200.00	10		0.00	0.22	0.22
	0.304	80.5	76.6	10		0.00	37.86	37.86	0	100.00	100.00		1200.00	10		0.00	0.16	0.16
700	0.101	93.3	130.6	10		0.00	47.69	47.69	0	100.00	100.00		1200.00	10		0.00	0.25	0.25
	0.299	92.8	84.3	10		0.00	60.54	60.54	0	100.00	100.00		1200.00	10		0.00	0.17	0.17
800	0.103	106.7	147.4	10		0.00	83.86	83.86	0	100.00	100.00		1200.00	10		0.00	0.28	0.28
	0.304	105.8	90.6	10		0.00	137.56	137.56	0	100.00	100.00		1200.00	10		0.00	0.20	0.20
900	0.099	119.8	156.2	10		0.00	134.43	134.43	0	100.00	100.00		1200.00	10		0.00	0.31	0.31
	0.299	119.0	98.6	10		0.00	158.73	158.73	0	100.00	100.00		1200.00	10		0.00	0.23	0.23
1000	0.105	133.7	166.1	10		0.00	210.71	210.71	0	100.00	100.00		1200.00	10		0.00	0.35	0.35
	0.297	133.1	107.2	10		0.00	257.12	257.12	0	100.00	100.00		1200.00	10		0.00	0.25	0.25
				200	**-**	**0.00**	**60.11**	**60.11**	**62**	**100.00**	**69.00**	**361.91**	**913.58**	**200**	**-**	**0.00**	**0.19**	**0.19**

Table 7.14. Experimental results for chordal graph instances of density 0.5 and 0.7 with large clusters.

				IP formulation					B&P					Decomposition				
n	Avg density	Avg # clust	Avg # cliques	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall
100	0.494	13.1	19.8	10		0.00	0.53	0.53	7	100.00	30.00	28.44	379.91	10		0.00	0.07	0.07
	0.699	13.3	15.1	10		0.00	0.69	0.69	10		0.00	32.04	32.04	10		0.00	0.08	0.08
200	0.495	26.6	28.9	10		0.00	1.99	1.99	10		0.00	233.31	233.31	10		0.00	0.09	0.09
	0.695	26.2	22.0	10		0.00	2.30	2.30	10		0.00	132.44	132.44	10		0.00	0.08	0.08
300	0.501	40.1	39.1	10		0.00	6.40	6.40	10		0.00	823.78	823.78	10		0.00	0.10	0.10
	0.706	40.0	27.0	10		0.00	11.61	11.61	10		0.00	730.09	730.09	10		0.00	0.09	0.09
400	0.499	53.4	46.1	10		0.00	15.10	15.10	5	100.00	50.00	1063.91	1131.96	10		0.00	0.12	0.12
	0.697	53.1	34.2	10		0.00	21.13	21.13	6	100.00	40.00	1043.58	1106.15	10		0.00	0.10	0.10
500	0.503	67.0	54.1	10		0.00	31.87	31.87	0	100.00	100.00		1200.00	10		0.00	0.13	0.13
	0.705	66.6	40.6	10		0.00	45.04	45.04	0	100.00	100.00		1200.00	10		0.00	0.11	0.11
600	0.505	80.5	59.7	10		0.00	60.70	60.70	0	100.00	100.00		1200.00	10		0.00	0.15	0.15
	0.704	80.3	44.4	10		0.00	87.92	87.92	0	100.00	100.00		1200.00	10		0.00	0.12	0.12
700	0.496	93.3	64.5	10		0.00	96.09	96.09	0	100.00	100.00		1200.00	10		0.00	0.15	0.15
	0.704	93.5	49.1	10		0.00	172.24	172.24	0	100.00	100.00		1200.00	10		0.00	0.14	0.14
800	0.497	107.7	67.8	10		0.00	199.78	199.78	0	100.00	100.00		1200.00	10		0.00	0.17	0.17
	0.698	106.1	50.9	10		0.00	238.96	238.96	0	100.00	100.00		1200.00	10		0.00	0.15	0.15
900	0.504	119.3	74.6	10		0.00	244.41	244.41	0	100.00	100.00		1200.00	10		0.00	0.18	0.18
	0.699	120.2	55.5	10		0.00	368.76	368.76	0	100.00	100.00		1200.00	10		0.00	0.16	0.16
1000	0.500	132.4	78.8	10		0.00	397.46	397.46	0	100.00	100.00		1200.00	10		0.00	0.21	0.21
	0.702	133.4	58.3	10		0.00	626.37	626.37	0	100.00	100.00		1200.00	10		0.00	0.17	0.17
				200	**-**	**0.00**	**131.47**	**131.47**	**68**	**100.00**	**66.00**	**510.95**	**948.48**	**200**	**-**	**0.00**	**0.13**	**0.13**

to optimally solve many of the chordal graph instances with 800 or more vertices and could not give reasonable optimality gaps or even find a feasible solution within the allowed time limit. B&P method performs poorly as compared to both of the other methods; it fails to solve instances with 600 or more vertices in all cases.

Finally, we summarize our results in Table 7.15. As also seen in Tables 7.9–7.14, our method demonstrates a clear superiority to the other two methods in all respects; it is able to solve all of the instances optimally in 0.16 seconds on the average. The second-best performance is that of the IP formulation; 96% of a total of 1200 instances are optimally solved, whereas B&P could only solve about 36% of them optimally.

Table 7.15. Summary of experimental results for all chordal graph instances.

		IP formulation			**B&P**			**Decomposition**		
Sizes of clusters	**Density**	**# opt**	**Avg % gap**	**Avg time**	**# opt**	**Avg % gap**	**Avg time**	**# opt**	**Avg % gap**	**Avg time**
small	low	190	4.76	318.58	93	53.50	834.27	200	0.00	0.20
	high	162	17.23	440.95	85	57.50	789.65	200	0.00	0.14
	all	**352**	**10.99**	**379.77**	**178**	**55.50**	**811.96**	**400**	**0.00**	**0.17**
medium	low	200	0.00	97.80	63	68.50	952.80	200	0.00	0.19
	high	200	0.00	199.67	66	67.00	913.01	200	0.00	0.13
	all	**400**	**0.00**	**148.73**	**129**	**67.75**	**932.91**	**400**	**0.00**	**0.16**
large	low	200	0.00	60.11	62	69.00	913.58	200	0.00	0.19
	high	200	0.00	131.47	68	66.00	948.48	200	0.00	0.13
	all	**400**	**0.00**	**95.79**	**130**	**67.50**	**931.03**	**400**	**0.00**	**0.16**
		1152	**3.66**	**208.10**	**437**	**63.58**	**891.96**	**1200**	**0.00**	**0.16**

7.4. Experimental Results for Perfect Graphs

In this section, we present the results of the experiments we conducted on perfect graph instances in its general form. We generated our test instances via the algorithm we introduced in Section 6.1.4. The number of vertices n of our test instances vary from 50 to 500, and for each pair of n and average edge density, we generated five different random graphs.

In our first set of experiments, we test the performance of our decomposition approach for perfect graphs using the cutting plane algorithm by Düzgün [42] versus

using SDP in the subproblem. As mentioned at the beginning of this chapter, we use MOSEK version 8.1.0.24 as SDP solver. The reason for us to select this SDP solver among several others is that MOSEK turned out to be the best-performing one according to the results of computational experiments conducted on a large set of problem instances [72], both in terms of solution times and the number of instances that are solved optimally.

Table 7.16 summarizes the computational results for perfect graph instances with cluster sizes varying between two and five. The first three columns in this table present the number of vertices ("n"), average edge density ("Avg density"), and average number of clusters ("Avg # clust") across five random instances. In the next two groups of columns, we report the results of our experiments for the two versions of our algorithm for perfect graphs under "Decomp. for Perfect Gr. w/ SDP" and "Decomp. for Perfect Gr. w/ Cutting Plane" headings, respectively. For the decomposition method coupled with SDP method, columns 4–7 show the number of instances that could be optimally solved among five ("# opt"), average optimality gap percentages over instances that could not be solved to optimality within the given time limit of 1200 seconds ("Avg % gap in nonopt"), average solution time in seconds over instances that are optimally solved ("Avg time in opt"), and average percentage of time the algorithm spends in the subproblem ("Avg % time subpr"). Columns 8–11 list the same set of results as columns 4–7 but for the decomposition method coupled with cutting plane algorithm. In each row, we report average values across runs on five independent instances.

Our observation from the results listed in Table 7.16 is that solving the subproblem via the cutting plane algorithm clearly yields superior results in terms of the number instances solved to optimality, average optimality gap, and average amount of time spent on instances that are solved to optimality. As n and edge density increase, the performance of both methods deteriorate as expected; however, coupling of the decomposition method with the cutting plane algorithm method outperforms the other in every aspect for all the instances. Out of the 200 instances we experiment with, the version that uses the cutting plane algorithm could optimally solve 160 of

Table 7.16. Experimental results for perfect graph instances with small clusters to compare SDP with the cutting plane method.

			Decomp. for Perfect Gr. w/ SDP				Decomp. for Perfect Gr. w/ Cutting Plane			
n	Avg density	Avg # clust	# opt	Avg % gap nonopt	Avg time in opt	Avg % time subpr	# opt	Avg % gap nonopt	Avg time in opt	Avg % time subpr
50	0.110	14.4	5		3.15	95.57	5		0.59	76.13
	0.293	13.8	5		2.21	96.06	5		0.73	80.47
	0.495	14.2	5		3.44	97.09	5		0.85	81.31
	0.710	14.0	5		6.50	98.07	5		1.79	82.94
100	0.096	28.4	5		94.59	99.74	5		4.36	95.10
	0.300	29.4	5		88.51	99.83	5		1.74	94.52
	0.488	28.0	5		105.39	99.85	5		3.08	94.77
	0.705	28.8	5		314.56	99.68	5		3.31	80.29
150	0.098	42.6	3	50.00	580.73	99.65	5		3.94	96.93
	0.298	42.2	5		720.72	99.91	5		4.95	97.92
	0.498	44.2	1	34.58	624.60	99.58	5		6.27	96.68
	0.693	43.4	0	34.29		99.72	5		14.78	85.48
200	0.107	56.8	0	53.33		97.26	5		7.36	98.46
	0.304	57.2	0	48.33		97.26	5		8.21	98.31
	0.496	57.2	0	48.03		97.34	5		16.40	98.39
	0.703	57.6	0	56.20		98.08	4	9.09	262.90	33.93
250	0.112	71.4	0	77.33		86.37	5		19.99	98.93
	0.304	71.2	0	82.42		91.18	5		22.86	99.04
	0.497	72.2	0	76.96		89.08	5		36.84	98.60
	0.693	70.2	0	70.12		93.20	2	13.33	519.33	44.08
300	0.110	86.6	0			65.99	5		22.54	99.14
	0.302	88.6	0			50.07	5		95.83	99.42
	0.506	83.4	0	92.86		74.89	5		216.92	97.79
	0.691	85.6	0	86.41		82.08	1	18.43	415.54	49.95
350	0.117	102.6	0			0.00	5		62.83	99.41
	0.301	100.0	0			17.76	5		82.80	99.51
	0.508	98.4	0	96.91		48.10	4	25.00	203.99	96.85
	0.698	99.6	0	96.21		72.86	0	29.11		81.93
400	0.111	114.8	0				5		72.87	99.34
	0.315	114.0	0				5		82.91	99.47
	0.502	114.2	0			14.08	3	18.33	323.10	95.90
	0.692	112.8	0	97.71		43.30	0	32.79		87.70
450	0.117	130.2	0				5		91.81	99.55
	0.309	130.4	0				4	11.11	159.83	98.78
	0.507	128.4	0				2	29.33	407.26	95.76
	0.696	125.8	0			47.04	0	36.57		86.25
500	0.117	142.8	0				5		180.12	99.57
	0.296	143.0	0				5		257.66	99.37
	0.507	144.2	0				0			95.78
	0.695	141.0	0	98.36			0			94.80

them, whereas the one using SDP could only solve 49 instances to optimality. Moreover, when we use SDP in the subproblem, we observed that SDP could not even finish solving the maximum clique problem for the first selection the master problem outputs in many instances with 300 or more vertices. In such cases, no optimality gap could be reported, which is revealed by the empty cells in "Avg % gap in nonopt" column in groups of instances for which the number of optimally solved instances shown in the fourth column is zero.

Although Düzgün's cutting plane algorithm outperforms SDP, we also wanted to test its performance against a maximum clique algorithm from literature that works on any type of graph. So, next, we compare the performance of the cutting plane algorithm by Düzgün to Tomita *et al.*'s general maximum clique algorithm [43] by coupling each one of them with our decomposition approach for perfect graphs. Table 7.17 summarizes the computational results for the two alternatives.

The general structure of Table 7.17 is the same as Table 7.16; here, columns 4–7 and 8–11 list the results of the decomposition algorithm with cutting plane and with Tomita *et al.*'s algorithm in the subproblem, respectively. The total number of problem instances solved to optimality is 166 with the general purpose maximum clique algorithm by Tomita *et al.*, while we can solve 160 of the 200 instances optimally if we use Düzgün's algorithm. Also, the percentage of time spent in subproblem is considerably reduced with Tomita *et al.*'s algorithm.

We summarize the performance comparison of the three alternative algorithms for maximum cliques in Table 7.18. We provide the total number of instances solved to optimality ("# opt"), percentage optimality gap averaged over all the instances that were not solved to optimality ("Avg % gap nonopt"), and time in optimally solved instances averaged over all instances solved to optimality ("Avg time in opt") for each of the three alternatives SDP, Düzgün's cutting plane algorithm, and Tomita *et al.*'s algorithm, respectively. In terms of all three measures we list, Tomita et al.'s algorithm appears to be the best alternative to utilize in the subproblem of our decomposition

Table 7.17. Experimental results for perfect graph instances with small clusters to compare the cutting plane method with Tomita *et al.*'s method.

			Decomp. for Perfect Gr. w/ Cutting Plane				Decomp. for Perfect Gr. w/ Tomita *et al.*'s			
n	**Avg density**	**Avg # clust**	**# opt**	**Avg % gap nonopt**	**Avg time in opt**	**Avg % time subpr**	**# opt**	**Avg % gap nonopt**	**Avg time in opt**	**Avg % time subpr**
50	0.110	14.4	5		0.59	76.13	5		0.28	45.43
	0.293	13.8	5		0.73	80.47	5		0.20	37.60
	0.495	14.2	5		0.85	81.31	5		0.14	35.20
	0.710	14.0	5		1.79	82.94	5		0.35	50.82
100	0.096	28.4	5		4.36	95.10	5		0.29	44.59
	0.300	29.4	5		1.74	94.52	5		0.19	45.24
	0.488	28.0	5		3.08	94.77	5		0.34	55.38
	0.705	28.8	5		3.31	80.29	5		1.41	31.38
150	0.098	42.6	5		3.94	96.93	5		0.28	48.99
	0.298	42.2	5		4.95	97.92	5		0.20	44.94
	0.498	44.2	5		6.27	96.68	5		0.50	53.36
	0.693	43.4	5		14.78	85.48	5		6.12	19.41
200	0.107	56.8	5		7.36	98.46	5		0.21	54.51
	0.304	57.2	5		8.21	98.31	5		0.20	54.27
	0.496	57.2	5		16.40	98.39	5		0.67	55.49
	0.703	57.6	4	9.09	262.90	33.93	4	9.09	232.05	3.11
250	0.112	71.4	5		19.99	98.93	5		0.19	49.82
	0.304	71.2	5		22.86	99.04	5		0.42	51.83
	0.497	72.2	5		36.84	98.60	5		1.89	48.41
	0.693	70.2	2	13.33	519.33	44.08	2	13.33	254.25	1.94
300	0.110	86.6	5		22.54	99.14	5		0.23	57.38
	0.302	88.6	5		95.83	99.42	5		1.04	62.28
	0.506	83.4	5		216.92	97.79	5		4.72	26.70
	0.691	85.6	1	18.43	415.54	49.95	2	18.51	319.48	3.03
350	0.117	102.6	5		62.83	99.41	5		0.23	63.60
	0.301	100.0	5		82.80	99.51	5		0.51	69.82
	0.508	98.4	4	25.00	203.99	96.85	4	8.33	15.55	20.48
	0.698	99.6	0	29.11		81.93	0	18.72		3.26
400	0.111	114.8	5		72.87	99.34	5		0.34	70.10
	0.315	114.0	5		82.91	99.47	5		0.65	68.71
	0.502	114.2	3	18.33	323.10	95.90	4	8.33	25.32	31.24
	0.692	112.8	0	32.79		87.70	0	24.55		4.76
450	0.117	130.2	5		91.81	99.55	5		0.39	69.22
	0.309	130.4	4	11.11	159.83	98.78	5		2.02	76.74
	0.507	128.4	2	29.33	407.26	95.76	4	10.00	64.26	33.84
	0.696	125.8	0	36.57		86.25	0	33.30		6.73
500	0.117	142.8	5		180.12	99.57	5		0.59	71.36
	0.296	143.0	5		257.66	99.37	5		1.05	71.71
	0.507	144.2	0	40.17		95.78	1	15.22	74.45	9.07
	0.695	141.0	0	41.67		94.80	0	30.15		7.89

procedure to find maximum cliques. We note that even though Düzgün's algorithm is customized for perfect graphs and performs quite satisfactorily as compared to SDP, the general algorithm by Tomita *et al.* yields better results. So, we decided to utilize Tomita *et al.*'s maximum clique algorithm in the subproblem of our decomposition procedure for perfect graphs.

Table 7.18. Summary of the comparison of the three alternative methods in subproblem for perfect graphs.

SDP			**Düzgün's cutting plane algorithm**			**Tomita *et al.*'s algorithm**		
# opt	Avg % gap nonopt	Avg time in opt	# opt	Avg % gap nonopt	Avg time in opt	# opt	Avg % gap nonopt	Avg time in opt
49	68.86	231.31	160	25.41	103.32	166	17.23	28.08

In the remaining portion of this section, we present the experimental results of the IP formulation, B&P algorithm by Furini *et al.* [30], and our decomposition algorithm. Tables 7.19–7.20 summarize the results of our computational experiments for general perfect graph instances with edge densities 0.1&0.3 and 0.5&0.7, respectively, where cluster sizes vary between two and five (which we refer to as small clusters). Column definitions of these tables are the same as those for permutation and generalized split graphs (see Tables 7.1–7.7), except that the last columns ("Avg % time in subpr") show the percentage of time spent in the subproblem in this case. The bottom row of Tables 7.19–7.20 provides the totals for columns containing the number of instances solved optimally ("# opt"), and the averages for all other columns.

From the results listed in Tables 7.19–7.20, we observe that our approach yields superior results to both of the other two in terms of time and optimality gap. Our method solves all of the low-density instances optimally, while the IP formulation and the B&P method can solve 78% and 53% of low density instances optimally, respectively. As n grows, the performance of all three methods worsen in general. Increasing edge density, however, results in improved performance for B&P method, while deteriorating that of IP formulation and decomposition method, as previously. Nevertheless, in terms of overall average of percentage optimality gap, the outperformance of the

Table 7.19. Experimental results for low-density perfect graph instances with small clusters.

			IP formulation					B&P					Decomposition					
n	Avg density	Avg # clust	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg % time in subpr
50	0.110	14.4	5		0.00	0.13	0.13	5		0.00	0.14	0.14	5		0.00	0.28	0.28	45.43
	0.293	13.8	5		0.00	0.21	0.21	5		0.00	2.02	2.02	5		0.00	0.20	0.20	37.60
100	0.096	28.4	5		0.00	0.90	0.90	5		0.00	23.07	23.07	5		0.00	0.29	0.29	44.59
	0.300	29.4	5		0.00	3.24	3.24	5		0.00	32.02	32.02	5		0.00	0.19	0.19	45.24
150	0.098	42.6	5		0.00	2.97	2.97	5		0.00	146.58	146.58	5		0.00	0.28	0.28	48.99
	0.298	42.2	5		0.00	8.70	8.70	4	100.00	20.00	140.79	352.63	5		0.00	0.20	0.20	44.94
200	0.107	56.8	5		0.00	7.93	7.93	5		0.00	420.52	420.52	5		0.00	0.21	0.21	54.51
	0.304	57.2	5		0.00	36.53	36.53	5		0.00	293.05	293.05	5		0.00	0.20	0.20	54.27
250	0.112	71.4	5		0.00	30.25	30.25	5		0.00	1023.62	1023.62	5		0.00	0.19	0.19	49.82
	0.304	71.2	5		0.00	85.98	85.98	5		0.00	712.18	712.18	5		0.00	0.42	0.42	51.83
300	0.110	86.6	5		0.00	74.87	74.87	0	100.00	100.00		1200.00	5		0.00	0.23	0.23	57.38
	0.302	88.6	5		0.00	340.37	340.37	4	100.00	20.00	726.59	821.27	5		0.00	1.04	1.04	62.28
350	0.117	102.6	5		0.00	204.47	204.47	0	100.00	100.00		1200.00	5		0.00	0.23	0.23	63.60
	0.301	100.0	4	94.52	18.90	689.68	791.74	0	100.00	100.00		1200.00	5		0.00	0.51	0.51	69.82
400	0.111	114.8	5		0.00	265.87	265.87	0	100.00	100.00		1200.00	5		0.00	0.34	0.34	70.10
	0.315	114.0	0	51.02	51.02		1200.00	0	100.00	100.00		1200.00	5		0.00	0.65	0.65	68.71
450	0.117	130.2	3	98.42	39.37	523.80	794.28	0	100.00	100.00		1200.00	5		0.00	0.39	0.39	69.22
	0.309	130.4	0	99.17	99.17		1200.00	0	100.00	100.00		1200.00	5		0.00	2.02	2.02	76.74
500	0.117	142.8	1	98.59	78.87	1180.90	1196.18	0	100.00	100.00		1200.00	5		0.00	0.59	0.59	71.36
	0.296	143.0	0	100.00	100.00		1200.00	0	100.00	100.00		1200.00	5		0.00	1.05	1.05	71.71
			78	**90.29**	**19.37**	**203.34**	**372.23**	**53**	**100.00**	**47.00**	**320.05**	**731.35**	**100**	**-**	**0.00**	**0.48**	**0.48**	**57.91**

Table 7.20. Experimental results for high-density perfect graph instances with small clusters.

			IP formulation					B&P					Decomposition					
n	Avg density	Avg # clust	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg % time in subpr
50	0.495	14.2	5		0.00	0.62	0.62	5		0.00	2.33	2.33	5		0.00	0.14	0.14	35.20
	0.710	14.0	5		0.00	0.79	0.79	5		0.00	1.91	1.91	5		0.00	0.35	0.35	50.82
100	0.488	28.0	5		0.00	3.02	3.02	5		0.00	17.38	17.38	5		0.00	0.34	0.34	55.38
	0.705	28.8	5		0.00	5.65	5.65	5		0.00	14.91	14.91	5		0.00	1.41	1.41	31.38
150	0.498	44.2	5		0.00	23.10	23.10	5		0.00	68.55	68.55	5		0.00	0.50	0.50	53.36
	0.693	43.4	5		0.00	38.99	38.99	5		0.00	61.09	61.09	5		0.00	6.12	6.12	19.41
200	0.496	57.2	5		0.00	84.10	84.10	5		0.00	152.09	152.09	5		0.00	0.67	0.67	55.49
	0.703	57.6	4	9.09	1.82	637.24	749.79	5		0.00	102.61	102.61	4	9.09	1.82	232.05	425.64	3.11
250	0.497	72.2	5		0.00	480.98	480.98	5		0.00	253.53	253.53	5		0.00	1.89	1.89	48.41
	0.693	70.2	2	23.89	14.33	830.33	1052.13	5		0.00	215.79	215.79	2	13.33	8.00	254.25	821.70	1.94
300	0.506	83.4	2	24.68	14.81	500.43	920.17	5		0.00	507.82	507.82	5		0.00	4.72	4.72	26.70
	0.691	85.6	0	83.96	83.96		1200.00	5		0.00	358.41	358.41	2	18.51	11.10	319.48	847.79	3.03
350	0.508	98.4	0	94.75	94.75		1200.00	5		0.00	657.70	657.70	4	8.33	1.67	15.55	252.44	20.48
	0.698	99.6	0	89.50	89.50		1200.00	5		0.00	559.38	559.38	0	18.72	18.72		1200.00	3.26
400	0.502	114.2	0	86.91	86.91		1200.00	4	100.00	20.00	1092.62	1114.09	4	8.33	1.67	25.32	260.25	31.24
	0.692	112.8	0	97.81	97.81		1200.00	5		0.00	742.60	742.60	0	24.55	24.55		1200.00	4.76
450	0.507	128.4	0	98.72	98.72		1200.00	0	100.00	100.00		1200.00	4	10.00	2.00	64.26	291.41	33.84
	0.696	125.8	0	98.32	98.32		1200.00	2	100.00	60.00	975.35	1110.14	0	33.30	33.30		1200.00	6.73
500	0.507	144.2	0	99.59	99.59		1200.00	2	100.00	60.00	1193.47	1197.39	1	15.22	12.18	74.45	974.89	9.07
	0.695	141.0	0	100.00	100.00		1200.00	5		0.00	1059.82	1059.82	0	30.15	30.15		1200.00	7.89
			48	**75.60**	**44.03**	**236.84**	**707.97**	**88**	**100.00**	**12.00**	**423.02**	**469.88**	**66**	**17.23**	**7.26**	**62.59**	**434.51**	**25.07**

decomposition to the other two persists even in high-density instances.

We conducted two additional sets of experiments on perfect graphs in order to test the effect of cluster sizes. Tables 7.21–7.24 report the results obtained on the same set of graphs as before but with cluster sizes between 4–7 and 6–9, respectively. Comparing the results in Tables 7.19–7.20 to those in Tables 7.21–7.24, we observe considerable improvement in the performance of all methods. For a given n value, as average size of clusters increases, the total number of clusters and hence the number of variables and constraints in the IP formulation reduce. This shrinkage in the size leads to improved performance. When cluster sizes vary between four and seven, the IP formulation outperforms the B&P method in all respects in graphs with average density 0.1 and 0.3, as we can see from the bottom row of Table 7.21. For graphs with high density, i.e., those with average density 0.5 and 0.7 and with medium-sized clusters, the B&P algorithm yields the highest number of instances solved optimally. Nevertheless, in terms of overall average of percentage optimality gaps, our decomposition method performs the best as in the previous set with small clusters. Finally, when we examine the values in Tables 7.23–7.24, which present the results when the cluster sizes are large (between six and nine in particular), we observe that our approach yields the best results in all respects regardless of density.

Finally in this section, we want to provide a brief synopsis of our results in Table 7.25. Out of the 600 instances in total, the decomposition algorithm was able to solve about 92% of them to optimality, whereas IP and B&P could solve 80% and 84%, respectively. In terms of the average optimality gap, our algorithm yields an order of magnitude better optimality gaps on the average as compared to the B&P algorithm, and the average solution time is about 30% and 27% of those of IP and B&P, respectively. Although the B&P method is able to yield a higher number of optimally solved instances for high-density graphs in the case of small and medium-sized clusters, the decomposition method still delivers the best average optimality gaps both in high-density graphs and in general.

Table 7.21. Experimental results for low-density perfect graph instances with medium-sized clusters.

			IP formulation					B&P					Decomposition					
n	Avg density	Avg # clust	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg % time in subpr
50	0.110	9.0	5		0.00	0.08	0.08	5		0.00	0.01	0.01	5		0.00	0.10	0.10	25.34
	0.293	9.0	5		0.00	0.10	0.10	5		0.00	0.02	0.02	5		0.00	0.14	0.14	35.13
100	0.096	17.8	5		0.00	0.20	0.20	5		0.00	0.02	0.02	5		0.00	0.24	0.24	42.40
	0.300	17.8	5		0.00	1.04	1.04	5		0.00	0.06	0.06	5		0.00	0.26	0.26	55.77
150	0.098	27.6	5		0.00	0.76	0.76	5		0.00	0.08	0.08	5		0.00	0.38	0.38	57.27
	0.298	26.6	5		0.00	1.59	1.59	5		0.00	0.09	0.09	5		0.00	0.41	0.41	58.39
200	0.107	36.8	5		0.00	1.71	1.71	5		0.00	0.86	0.86	5		0.00	0.46	0.46	69.35
	0.304	36.0	5		0.00	7.41	7.41	5		0.00	50.36	50.36	5		0.00	0.70	0.70	75.50
250	0.112	45.0	5		0.00	4.60	4.60	5		0.00	1.29	1.29	5		0.00	0.43	0.43	62.99
	0.304	45.8	5		0.00	16.56	16.56	5		0.00	243.26	243.26	5		0.00	0.78	0.78	77.36
300	0.110	54.4	5		0.00	10.13	10.13	5		0.00	14.05	14.05	5		0.00	0.46	0.46	73.36
	0.302	54.2	5		0.00	94.55	94.55	5		0.00	475.23	475.23	5		0.00	2.17	2.17	72.51
350	0.117	63.6	5		0.00	22.88	22.88	5		0.00	163.62	163.62	5		0.00	1.21	1.21	84.68
	0.301	62.8	5		0.00	223.55	223.55	5		0.00	671.28	671.28	5		0.00	1.50	1.50	82.32
400	0.111	71.4	5		0.00	33.68	33.68	5		0.00	221.57	221.57	5		0.00	1.06	1.06	81.53
	0.315	73.0	5		0.00	494.34	494.34	0	100.00	100.00		1200.00	5		0.00	1.92	1.92	76.12
450	0.117	82.2	5		0.00	261.58	261.58	4	100.00	20.00	190.43	392.34	5		0.00	1.06	1.06	83.63
	0.309	81.6	4	100.00	20.00	733.12	826.50	0	100.00	100.00		1200.00	5		0.00	1.87	1.87	77.57
500	0.117	90.6	5		0.00	449.23	449.23	1	100.00	80.00	4.32	960.86	5		0.00	1.90	1.90	87.44
	0.296	91.4	3	87.50	35.00	961.09	1056.65	0	100.00	100.00		1200.00	5		0.00	2.17	2.17	84.64
			97	**93.75**	**2.75**	**165.91**	**175.36**	**80**	**100.00**	**20.00**	**119.80**	**339.75**	**100**	**-**	**0.00**	**0.95**	**0.95**	**68.16**

Table 7.22. Experimental results for high-density perfect graph instances with medium-sized clusters.

			IP formulation					B&P					Decomposition					
n	Avg density	Avg # clust	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg % time in subpr
50	0.495	9.0	5		0.00	0.15	0.15	5		0.00	0.04	0.04	5		0.00	0.22	0.22	45.21
	0.710	8.8	5		0.00	0.32	0.32	5		0.00	1.04	1.04	5		0.00	0.29	0.29	55.70
100	0.488	17.6	5		0.00	3.20	3.20	5		0.00	3.44	3.44	5		0.00	0.47	0.47	66.76
	0.705	18.0	5		0.00	1.62	1.62	5		0.00	9.91	9.91	5		0.00	1.64	1.64	60.08
150	0.498	26.6	5		0.00	7.38	7.38	5		0.00	20.24	20.24	5		0.00	0.57	0.57	73.52
	0.693	27.8	5		0.00	8.94	8.94	5		0.00	37.06	37.06	5		0.00	2.99	2.99	52.97
200	0.496	35.6	5		0.00	19.81	19.81	5		0.00	98.02	98.02	5		0.00	1.08	1.08	70.71
	0.703	36.2	5		0.00	33.79	33.79	5		0.00	76.94	76.94	5		0.00	4.43	4.43	44.63
250	0.497	45.2	5		0.00	58.95	58.95	5		0.00	209.77	209.77	5		0.00	0.65	0.65	70.86
	0.693	46.4	5		0.00	121.05	121.05	5		0.00	132.32	132.32	5		0.00	235.66	235.66	7.86
300	0.506	55.0	5		0.00	195.94	195.94	5		0.00	450.78	450.78	5		0.00	0.98	0.98	74.14
	0.691	54.4	4	20.00	4.00	250.95	440.76	5		0.00	226.42	226.42	4	20.00	4.00	257.77	446.21	23.36
350	0.508	63.8	4	100.00	20.00	407.27	565.81	5		0.00	734.28	734.28	5		0.00	1.71	1.71	67.28
	0.698	63.2	3	60.00	24.00	670.25	882.15	5		0.00	287.89	287.89	4	16.67	3.33	111.61	329.29	9.15
400	0.502	73.0	5		0.00	713.59	713.59	4	100.00	20.00	640.53	752.42	5		0.00	3.72	3.72	54.37
	0.692	73.2	0	82.86	82.86		1200.00	5		0.00	539.49	539.49	4	37.50	7.50	33.92	267.14	20.36
450	0.507	82.8	2	96.67	58.00	981.34	1112.53	3	100.00	40.00	980.21	1068.12	5		0.00	11.30	11.30	62.12
	0.696	81.2	0	95.33	95.33		1200.00	5		0.00	564.83	564.83	0	31.07	31.07		1200.00	4.26
500	0.507	91.2	0	99.20	99.20		1200.00	0	100.00	100.00		1200.00	5		0.00	6.62	6.62	54.75
	0.695	92.6	0	100.00	100.00		1200.00	5		0.00	895.48	895.48	1	31.80	25.44	406.27	1041.25	3.80
			73	**81.76**	**24.17**	**217.16**	**448.30**	**92**	**100.00**	**8.00**	**310.98**	**365.42**	**88**	**27.41**	**3.57**	**56.94**	**177.81**	**46.09**

Table 7.23. Experimental results for low-density perfect graph instances with large clusters.

n	Avg density	Avg # clust	IP formulation					B&P					Decomposition					
			# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg % time in subpr
50	0.110	6.6	5		0.00	0.06	0.06	5		0.00	0.01	0.01	5		0.00	0.17	0.17	19.72
	0.293	6.6	5		0.00	0.08	0.08	5		0.00	0.01	0.01	5		0.00	0.19	0.19	29.79
100	0.096	13.4	5		0.00	0.16	0.16	5		0.00	0.02	0.02	5		0.00	0.10	0.10	30.79
	0.300	13.2	5		0.00	0.72	0.72	5		0.00	0.03	0.03	5		0.00	0.16	0.16	43.86
150	0.098	19.4	5		0.00	0.53	0.53	5		0.00	0.08	0.08	5		0.00	0.23	0.23	38.11
	0.298	19.8	5		0.00	1.01	1.01	5		0.00	0.05	0.05	5		0.00	0.38	0.38	53.32
200	0.107	26.8	5		0.00	1.03	1.03	5		0.00	0.11	0.11	5		0.00	0.38	0.38	53.28
	0.304	26.6	5		0.00	3.37	3.37	5		0.00	0.14	0.14	5		0.00	0.50	0.50	64.61
250	0.112	33.2	5		0.00	2.67	2.67	5		0.00	0.24	0.24	5		0.00	0.23	0.23	50.76
	0.304	33.4	5		0.00	9.18	9.18	5		0.00	0.36	0.36	5		0.00	0.37	0.37	69.05
300	0.110	40.2	5		0.00	6.15	6.15	5		0.00	0.52	0.52	5		0.00	0.30	0.30	53.70
	0.302	40.2	5		0.00	22.06	22.06	5		0.00	4.85	4.85	5		0.00	0.63	0.63	71.75
350	0.117	46.2	5		0.00	14.54	14.54	5		0.00	0.75	0.75	5		0.00	0.29	0.29	60.45
	0.301	47.6	5		0.00	52.97	52.97	5		0.00	85.89	85.89	5		0.00	0.72	0.72	70.16
400	0.111	53.2	5		0.00	22.06	22.06	5		0.00	1.24	1.24	5		0.00	0.43	0.43	66.04
	0.315	52.6	5		0.00	113.07	113.07	5		0.00	1.51	1.51	5		0.00	0.90	0.90	81.08
450	0.117	59	5		0.00	137.30	137.30	5		0.00	1.90	1.90	5		0.00	0.47	0.47	69.24
	0.309	59.4	5		0.00	439.23	439.23	5		0.00	44.07	44.07	5		0.00	1.24	1.24	84.12
500	0.117	65.8	5		0.00	143.72	143.72	5		0.00	2.71	2.71	5		0.00	0.73	0.73	76.06
	0.296	66.4	5		0.00	480.95	480.95	5		0.00	42.40	42.40	5		0.00	0.95	0.95	82.08
			100	**-**	**0.00**	**72.54**	**72.54**	**100**	**-**	**0.00**	**9.34**	**9.34**	**100**	**-**	**0.00**	**0.47**	**0.47**	**58.40**

Table 7.24. Experimental results for high-density perfect graph instances with large clusters.

			IP formulation					B&P					Decomposition					
n	Avg density	Avg # clust	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	# opt	Avg % gap in nonopt	Avg % gap overall	Avg time in opt	Avg time overall	Avg % time in subpr
50	0.495	6.4	5		0.00	0.11	0.11	5		0.00	0.01	0.01	5		0.00	0.23	0.23	33.81
	0.710	7	5		0.00	0.25	0.25	5		0.00	0.38	0.38	5		0.00	0.26	0.26	59.92
100	0.488	13.2	5		0.00	2.13	2.13	5		0.00	0.05	0.05	5		0.00	0.22	0.22	53.08
	0.705	13.4	5		0.00	4.12	4.12	5		0.00	5.83	5.83	5		0.00	0.77	0.77	69.83
150	0.498	19	5		0.00	1.94	1.94	5		0.00	0.62	0.62	5		0.00	0.35	0.35	64.90
	0.693	20.2	5		0.00	4.65	4.65	5		0.00	26.65	26.65	5		0.00	0.95	0.95	61.71
200	0.496	26.6	5		0.00	7.13	7.13	5		0.00	12.97	12.97	5		0.00	0.98	0.98	58.60
	0.703	26.8	5		0.00	15.67	15.67	5		0.00	55.31	55.31	5		0.00	4.13	4.13	51.20
250	0.497	33.2	5		0.00	19.66	19.66	5		0.00	62.03	62.03	5		0.00	0.79	0.79	77.09
	0.693	33.4	5		0.00	42.21	42.21	5		0.00	129.70	129.70	5		0.00	4.05	4.05	51.07
300	0.506	40	5		0.00	47.66	47.66	5		0.00	123.92	123.92	5		0.00	1.36	1.36	80.21
	0.691	40.6	5		0.00	118.47	118.47	5		0.00	244.15	244.15	5		0.00	5.04	5.04	44.38
350	0.508	46.6	5		0.00	274.99	274.99	5		0.00	394.37	394.37	5		0.00	2.25	2.25	82.15
	0.698	46.8	4	100.00	20.00	208.45	406.76	5		0.00	407.26	407.26	5		0.00	8.06	8.06	40.88
400	0.502	52.8	5		0.00	298.38	298.38	5		0.00	728.67	728.67	5		0.00	2.58	2.58	64.96
	0.692	52.2	3	100.00	40.00	354.49	692.69	5		0.00	658.65	658.65	5		0.00	4.23	4.23	46.87
450	0.507	59.6	3	100.00	40.00	538.64	803.18	2	100.00	60.00	372.79	869.11	5		0.00	1.92	1.92	81.49
	0.696	60.6	4	100.00	20.00	896.27	957.01	5		0.00	733.06	733.06	5		0.00	85.12	85.12	12.77
500	0.507	66	1	100.00	80.00	960.58	1152.12	2	100.00	60.00	903.36	1081.34	5		0.00	4.65	4.65	69.60
	0.695	65	0	100.00	100.00		1200.00	3	100.00	40.00	1038.45	1103.07	5		0.00	40.48	40.48	25.01
			85	**100.00**	**15.00**	**199.78**	**302.46**	**92**	**100.00**	**8.00**	**294.91**	**331.86**	**100**	**-**	**0.00**	**8.42**	**8.42**	**56.48**

Table 7.25. Summary of experimental results for all perfect graph instances.

		IP formulation			B&P			Decomposition		
Sizes of clusters	**Density**	**# opt**	**Avg % gap**	**Avg time**	**# opt**	**Avg % gap**	**Avg time**	**# opt**	**Avg % gap**	**Avg time**
small	low	78	19.37	372.23	53	47.00	731.35	100	0.00	0.48
	high	48	44.03	707.97	88	12.00	469.88	66	7.26	434.51
	all	**126**	**31.70**	**540.10**	**141**	**29.50**	**600.62**	**166**	**3.63**	**217.50**
medium	low	97	2.75	175.36	80	20.00	339.75	100	0.00	0.95
	high	73	24.17	448.30	92	8.00	365.42	88	3.57	177.81
	all	**170**	**13.46**	**311.83**	**172**	**14.00**	**352.59**	**188**	**1.78**	**89.39**
large	low	100	0.00	72.54	100	0.00	9.34	100	0.00	0.47
	high	85	15.00	302.46	92	8.00	331.86	100	0.00	8.42
	all	**185**	**7.50**	**187.50**	**192**	**4.00**	**170.60**	**200**	**0.00**	**4.44**
		481	**17.55**	**346.48**	**505**	**15.83**	**374.60**	**554**	**1.80**	**103.77**

7.5. Experimental Results for General Graphs

The last section of our computational study is dedicated to testing the performance our algorithm on general graph instances and comparing it to the IP formulation and B&P algorithm by Furini *et al.*. To generate random general graphs, we used the well-known Erdős–Rényi model [73]; for a given n and desired edge density p, we create a random graph where each possible edge is formed with probability p independently of every other edge.

Our first set of experiments is conducted on a set of 24 Erdős–Rényi-type general graph instances with cluster sizes varying between two and five. The number of vertices ranges from 50 to 500, and we use the four average edge density values of 0.1, 0.3, 0.5, and 0.7 as before. For each pair of n and edge density value, we used a single problem instance. Table 7.26 summarizes the computational results we obtained on this set of problem instances. The first three columns in this table present the number of vertices ("n"), edge density ("Density"), and number of clusters ("# clust") for each random instance. In the next four groups of columns, we report the results of our experiments for the four alternatives under "IP formulation", "B&P", "Decomposition with B&C", and "Decomposition with exactcolors" headings, respectively. The last two sets of columns respectively list the results of our decomposition algorithm when our

Table 7.26. Experimental results for general graph instances with small clusters.

n	Density	# clust	IP formulation		B&P		Decomposition with B&C					Decomposition with exactcolors				
			% gap	Total time	% gap	Total time	% gap	% time in max cl	% time in min col	# calls to min col	Total time	% gap	% time in max cl	% time in min col	# calls to min col	Total time
50	0.088	13	0.00	0.39	0.00	0.04	0.00	0.42	34.81	1	0.47	0.00	0.37	1.38	1	0.36
	0.322	14	0.00	0.95	0.00	2.09	0.00	0.07	90.02	28	10.16	0.00	0.56	29.05	30	0.95
	0.505	17	0.00	2.77	0.00	2.56	0.00	0.13	90.70	149	33.97	0.00	1.18	30.12	56	3.22
	0.700	14	0.00	5.70	0.00	2.67	0.00	0.38	71.23	26	5.78	0.00	1.08	5.52	12	1.94
100	0.105	28	0.00	5.20	0.00	18.60	0.00	0.77	68.19	7	2.07	0.00	0.95	4.62	7	1.19
	0.299	29	0.00	45.70	0.00	16.06	0.00	0.08	96.26	3793	387.87	0.00	0.40	71.18	2769	116.92
	0.497	31	0.00	663.52	0.00	14.22	25.00	0.06	96.17	8295	1200.00	20.00	0.34	25.72	5830	1200.00
	0.697	27	16.67	1200.00	0.00	12.17	33.33	0.22	70.95	4798	1200.00	42.86	0.59	11.37	3854	1200.00
200	0.101	59	0.00	162.33	0.00	188.21	0.00	0.02	99.68	8819	1190.00	33.33	0.24	73.96	6258	1200.00
	0.300	55	40.00	1200.00	0.00	110.36	50.00	0.01	99.90	4315	1200.00	66.67	0.14	94.53	2692	1200.00
	0.498	56	92.11	1200.00	0.00	99.92	66.67	0.00	99.98	67	1200.00	80.00	0.00	99.96	27	1200.00
	0.698	61	78.26	1200.00	0.00	94.78	81.82	0.00	99.99	2	1200.00	88.24	0.00	99.88	3	1200.00
300	0.101	82	50.00	1200.00	0.00	747.14	50.00	0.01	99.88	7143	1200.00	50.00	0.09	98.17	1261	1200.00
	0.297	86	100.00	1200.00	0.00	539.93	60.00	0.00	99.96	3	1200.00	77.78	0.00	99.88	2	1200.00
	0.502	85	100.00	1200.00	0.00	427.59	75.00	0.00	99.98	2	1200.00	85.71	0.00	99.72	2	1200.00
	0.706	83	100.00	1200.00	0.00	326.00	75.00	0.01	99.98	3	1200.00	90.48	0.00	99.70	3	1200.00
400	0.099	111	100.00	1200.00	0.00	1048.98	66.67	0.00	99.97	3332	1200.00	80.00	0.01	99.92	16	1200.00
	0.299	115	100.00	1200.00	0.00	726.03	80.00	0.00	99.99	2	1200.00	90.95	0.00	99.11	3	1200.00
	0.500	115	99.12	1200.00	0.00	644.26	75.00	0.01	99.97	3	1200.00	89.47	0.00	95.95	2	1200.00
	0.701	117	100.00	1200.00	100.00	1200.00	85.71	0.02	99.97	3	1200.00	93.33	0.01	95.12	2	1200.00
500	0.100	146	100.00	1200.00	100.00	1200.00	75.00	0.01	99.97	1	1200.00	83.33	0.00	99.90	2	1200.00
	0.299	144	99.24	1200.00	0.00	1091.84	66.67	0.01	99.97	3	1200.00	92.31	0.00	97.11	2	1200.00
	0.501	143	100.00	1200.00	100.00	1200.00	77.78	0.01	99.98	2	1200.00	90.95	0.00	91.40	3	1200.00
	0.701	146	100.00	1200.00	100.00	1200.00	86.67	0.16	99.83	3	1200.00	97.14	0.06	67.20	2	1200.00
			8		**20**		**7**					**6**				
			57.31	**836.94**	**16.67**	**454.73**	**47.10**	**0.10**	**92.39**	**1700.00**	**917.93**	**56.35**	**0.25**	**70.44**	**951.63**	**905.19**

implementation of the B&C algorithm by Méndez-Díaz and Zabala [47] and when the open-source implementation *exactcolors* [48] is used to solve the coloring problem in the subproblem. Here, we reported the percentage of time spent in solving maximum clique and coloring algorithms separately, in order to point out the need for further improvement in the coloring algorithm. Percentage of time spent to solve maximum clique and minimum coloring problems ("% time in max cl", "% time in min col") as well as the number of times coloring problem is called ("# calls to min col") are given for two alternative applications of the decomposition approach with different coloring algorithms. The bottom two rows of the table show the number of optimally solved instances and the average values of columns for the four alternatives, respectively.

From the results listed in Table 7.26, we observe that the B&P algorithm by Furini *et al.* clearly outperforms both versions of the decomposition algorithm and also the IP formulation. As in the case of perfect graphs, the general performance of all three methods deteriorates as n grows. When we use the B&C method by Méndez-Díaz and Zabala to solve the coloring problem in the subproblem, we notice that the given time limit of 20 minutes can be consumed even with few calls to the coloring problem. Replacing the B&C method with exactcolors by Held *et al.* in the subproblem unfortunately did not improve the performance. In fact, it even yielded slightly worse results in the sense that the number of optimally solved instances and average number of times the algorithm was able to call the coloring problem have decreased, and the average optimality gap have increased. Thus, we decided to continue with the B&C method for solving the coloring problem in the subproblem. Using the same set of random graphs, we also experimented with medium-sized and large clusters, but the performance unfortunately did not improve significantly as compared with the other two methods.

The percentage of solution time that the decomposition algorithm spends on solving the coloring problem becomes nearly 100% in instances with 200 or more vertices, as the values in "% time in min col" columns of Table 7.26 reveal. So, it is the coloring problem that creates the bottleneck. As mentioned above, we gen-

erated the set of general graph instances using the well-known Erdős–Rényi model. There are, however, other types of models for random graph generation. We wondered if there exist other types of graphs whose inherent structure would have an impact on the algorithms we compare. To this end, we performed some preliminary experiments on instances produced by some well-known random graph models including the Watts-Strogatz model [74], Newman-Watts-Strogatz model [75], and Barabasi-Albert model [76], but results were similar. Then, we consider the class of unit disk graphs since a study by Mani and Petr [77] shows (by means of extensive simulation experiments) that the clique number and the chromatic number values are very close to each other in unit disk graphs. So, in order to possibly reduce the need to call the coloring problem, we tested our algorithm on unit disk graphs. We generated some random unit disk graph instances and conducted a preliminary set of experiments, whose results are reported in Table 7.27.

Table 7.27 has the same structure as the previous one except that we excluded the last set of columns under "Decomposition with exactcolors" heading in this case, because we decided to utilize the B&C method to solve the coloring problem. The number of vertices in our unit disk graph instances vary from 50 to 400, and we again use the same four approximate edge density values as before. Since our aim is to have a general idea about the performance of the three methods under consideration, we used a single problem instance for each pair of n and edge density value, which makes a total of 20 instances.

As in the previous case, the B&P method is able to solve all instances optimally within the given time limit of 20 minutes, whereas the decomposition method can solve only 30% of the instances. A key difference from the results obtained in Erdős–Rényi-type graphs is that the percentage of time spent in coloring problem decreases from approximately 90% to 10% in this case. However, the poor performance of our algorithm persists despite the fact that the time spent in coloring problem does not really create a bottleneck anymore. The reason for this is that the cuts (5.7), which we generate using the chromatic numbers, cannot provide very good lower bounds that would

expedite the process of reaching an optimal solution.

Table 7.27. Experimental results for unit disk graph instances with small clusters.

			IP formulation		B&P		Decomposition with B&C				
n	**Density**	**# clust**	**% gap**	**Total time**	**% gap**	**Total time**	**% gap**	**% time in max cl**	**% time in min col**	**# calls to min col**	**Total time**
50	0.155	13	0.00	0.18	0.00	1.59	0.00	0.71	6.40	2	0.42
	0.311	15	0.00	0.36	0.00	3.98	0.00	0.33	7.14	8	1.79
	0.534	13	0.00	0.38	0.00	1.68	0.00	0.21	1.71	16	21.88
	0.691	15	0.00	0.53	0.00	2.61	0.00	0.20	1.89	9	15.31
100	0.114	30	0.00	1.86	0.00	21.12	0.00	0.49	17.48	4	0.21
	0.322	26	0.00	1.79	0.00	8.57	25.00	0.01	0.12	29	1200.00
	0.534	28	0.00	2.54	0.00	8.23	28.57	0.14	0.40	35	1200.00
	0.708	30	0.00	2.05	0.00	13.31	33.33	0.13	0.39	25	1200.00
200	0.114	57	0.00	26.98	0.00	115.84	0.00	0.06	1.61	155	592.21
	0.300	58	0.00	124.08	0.00	81.51	37.50	0.13	1.73	346	1200.00
	0.511	61	0.00	522.32	0.00	87.25	46.15	0.25	3.02	635	1200.00
	0.711	58	0.00	138.83	0.00	111.81	50.00	0.29	0.86	145	1200.00
300	0.109	82	0.00	180.55	0.00	390.20	40.00	0.17	1.41	921	1200.00
	0.309	89	37.50	1200.00	0.00	270.08	50.00	0.28	2.56	235	1200.00
	0.516	86	78.08	1200.00	0.00	296.01	55.00	0.40	3.21	228	1200.00
	0.706	89	0.00	790.85	0.00	394.63	48.00	0.52	1.32	91	1200.00
400	0.106	113	95.58	1200.00	0.00	1062.52	50.00	0.29	3.83	1017	1200.00
	0.304	114	89.38	1200.00	0.00	449.75	62.50	0.06	98.62	143	1200.00
	0.503	114	82.46	1200.00	0.00	310.95	57.69	0.82	50.05	309	1200.00
	0.716	114	73.45	1200.00	0.00	437.41	57.14	0.95	1.54	55	1200.00
			14		**20**		**6**				
			22.82	**449.66**	**0.00**	**203.45**	**32.04**	**0.32**	**10.26**	**220.40**	**841.30**

Our conclusion from the results of our computational experiments we presented in this section is that the difficulty of the graph coloring problem on general graphs and the weakness of our related cut (5.7) constitute the two main sources of the poor performance of our decomposition approach.

8. CONCLUSION

In this dissertation, we studied the selective graph coloring problem, Sel-Col, which is a generalization of the classical graph coloring problem and is known to be NP-hard. Our study is comprised of two main parts; an exact solution approach to solve Sel-Col and random graph generation from specific families of graphs.

In the first part of this study, we presented a decomposition-based exact solution approach for Sel-Col, and applied it to selected classes of graphs. A natural first attempt was to focus on perfect graphs, whose structural properties we made use of to facilitate the solution procedure. We first concentrated on three subclasses of perfect graphs; namely, permutation, generalized split, and chordal graphs, and combined integer programming techniques and combinatorial algorithms in our decomposition procedure. Then, we turned our attention to the class of perfect graphs in its general form, rather than a subclass of it. Finally, we applied our decomposition approach to general (non-perfect) graphs.

The second part of this dissertation focuses on random graph generation, which emerged from our need for instances from specific classes of graphs. We provided algorithms to randomly generate permutation, generalized split, chordal, and perfect graphs with different size and densities. Especially our proposed algorithms for chordal and perfect graph generation bridge a gap in the literature; the former one for being based on a characterization that has not been directly utilized before and for providing the most varied outputs, and the latter one for being the first implementable algorithm that is used to generate perfect graphs.

To the best of our knowledge, our proposed chordal graph generation algorithm, which we call Algorithm ChordalGen, is the first one that is directly based on subtree intersection characterization of chordal graphs. We introduced three different subtree generation methods to be used within Algorithm ChordalGen, each of which provides

different structural properties in the output chordal graph. We showed that one of these subtree generation methods, which we call GrowingSubtree, yields the most varied chordal graphs as far as the distribution of maximal cliques are concerned. Since chordal graphs arise in practical applications from a wide variety of fields and a large number of existing algorithms are tailored for chordal graphs, proposition of a method that is able to produce diverse chordal graphs together with a large suite of instances that can be accessed online serves to meet a need in the literature.

Generation of perfect graphs in its general form was an area that the literature only provides theoretical studies about but lacks any actual implementable method. Perfect graphs being a class of graphs for which many algorithms for a variety of problems have been tailored in the literature, algorithms directed toward their random generation and/or availability of general-purpose instances are needed for performance testing in practice. Our study fills this gap by providing an implementable algorithm and also a large set of already generated instances of varying size and densities, which might well serve useful purposes in other studies concerned about perfect graphs.

We tested the performance of our decomposition algorithm on a large suite of randomly generated problem instances, and compared the results to those of an IP formulation and a state-of-the-art branch-and-price algorithm from the literature. Our computational results show that the decomposition approach significantly improves the solution performance in all test instances of permutation, generalized split, and chordal graphs, but especially in low-density instances. The improvement manifests most evidently in the class of chordal graphs. In the general class of perfect graphs, the overall performance of our decomposition algorithm is superior to that of the two other methods we compare our results to. Unfortunately, we could not obtain satisfactory results in the general case; both of the algorithms that we compare our results to, especially the branch-and-price algorithm from the literature, yielded superior results. The problem was that either a big proportion of solution time was spent in the coloring problem that we solve in the subproblem of our decomposition procedure, or even if we focus on a graph class where the coloring problem did not create a bottleneck in

practice, our cut returned by the coloring problem was not strong enough to help us reach an optimal solution quickly. So, the difficulty encountered in general graphs has been twofold: the inherent hardness of the coloring problem, and the weakness of our cut for general graphs.

A possible future research direction that emerges as a natural consequence of our observations on general graphs is to work on designing stronger cuts, possibly by making use of vertex-critical subgraphs via the algorithm presented in [78]. It may help us improve the performance of our decomposition strategy on general graphs. In the case of unit disk graphs, custom-tailored polynomial-time approximation algorithms for graph coloring (for instance the one presented in [79]) can be fed into the exact solution procedure for graph coloring to facilitate it further. Other possibilities could be to work on preprocessing strategies to reduce the search space, or to adapt our solution strategy to other graph classes possessing structural properties that may be utilized within this decomposition framework. Regarding random graph generation, our method to produce perfect graphs can be enriched by incorporating different perfection-preserving methods, and other subtree generation methods can be inserted into our chordal graph generation algorithm to obtain graphs with differently distributed maximal clique sizes.

REFERENCES

1. Demange, M., T. Ekim, B. Ries and C. Tanasescu, "On Some Applications of the Selective Graph Coloring Problem", *European Journal of Operational Research*, Vol. 240, No. 2, pp. 307–314, 2015.

2. Andreou, M. I., V. G. Papadopoulou, P. G. Spirakis, B. Theodorides and A. Xeros, "Generating and Radiocoloring Families of Perfect Graphs", *Experimental and Efficient Algorithms*, pp. 302–314, Springer, 2005.

3. Marx, D., "Graph Colouring Problems and Their Applications in Scheduling", *Periodica Polytechnica Electrical Engineering*, Vol. 48, No. 1-2, pp. 11–16, 2004.

4. Chaitin, G. J., "Register Allocation & Spilling via Graph Coloring", *ACM Sigplan Notices*, Vol. 17, pp. 98–105, ACM, 1982.

5. Lewis, R., *A Guide to Graph Colouring*, Springer, 2015.

6. Demange, M., J. Monnot, P. Pop and B. Ries, "On the Complexity of the Selective Graph Coloring Problem in Some Special Classes of Graphs", *Theoretical Computer Science*, Vol. 540, pp. 89–102, 2014.

7. Li, G. and R. Simha, "The Partition Coloring Problem and Its Application to Wavelength Routing and Assignment", *Proceedings of the First Workshop on Optical Networks*, 2000.

8. Garey, M. R., D. S. Johnson and L. Stockmeyer, "Some Simplified NP-Complete Problems", *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pp. 47–63, ACM, 1974.

9. Çalık, A., *The Selective Graph Coloring Problem*, Master's Thesis, Boğaziçi University, İstanbul, 2013.

10. Şeker, O., T. Ekim and Z. C. Taşkın, "A Decomposition Approach to Solve the Selective Graph Coloring Problem in Some Perfect Graph Families", *Networks*, 2018, `http://dx.doi.org/10.1002/net.21850`, Available online.

11. Şeker, O., T. Ekim and Z. C. Taşkın, "An Exact Cutting Plane Algorithm to Solve the Selective Graph Coloring Problem in Perfect Graphs", Submitted.

12. Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.

13. Golumbic, M. C., *Algorithmic Graph Theory and Perfect Graphs*, Vol. 57, Elsevier, 2004.

14. Chang, M.-S. and F.-H. Wang, "Efficient Algorithms for the Maximum Weight Clique and Maximum Weight Independent Set Problems on Permutation Graphs", *Information Processing Letters*, Vol. 43, No. 6, pp. 293–295, 1992.

15. Klein, P. N., "Efficient Parallel Algorithms for Chordal Graphs", *SIAM Journal on Computing*, Vol. 25, No. 4, pp. 797–827, 1996.

16. Hoàng, C. T., "Efficient Algorithms for Minimum Weighted Colouring of Some Classes of Perfect Graphs", *Discrete Applied Mathematics*, Vol. 55, No. 2, pp. 133–143, 1994.

17. Okamoto, Y., T. Uno and R. Uehara, "Counting the Number of Independent Sets in Chordal Graphs", *Journal of Discrete Algorithms*, Vol. 6, No. 2, pp. 229–242, 2008.

18. Grötschel, M., L. Lovász and A. Schrijver, "Polynomial Algorithms for Perfect Graphs", *North-Holland Mathematics Studies*, Vol. 88, pp. 325–356, 1984.

19. Şeker, O., P. Heggernes, T. Ekim and Z. C. Taşkın, "Linear-Time Generation of Random Chordal Graphs", *Lecture Notes in Computer Science*, Vol. 10236, pp.

442–453, 2017.

20. Şeker, O., P. Heggernes, T. Ekim and Z. C. Taşkın, "Efficient Generation of Random Chordal Graphs Using Subtrees of a Tree", Submitted.

21. Chudnovsky, M., N. Robertson, P. Seymour and R. Thomas, "The Strong Perfect Graph Theorem", *Annals of Mathematics*, pp. 51–229, 2006.

22. Gargano, L., P. Hell and S. Perennes, "Colouring Paths in Directed Symmetric Trees with Applications to WDM Routing", *Automata, Languages and Programming*, pp. 505–515, Springer, 1997.

23. Rabani, Y., "Path Coloring on the Mesh", *37th Annual Symposium on Foundations of Computer Science*, pp. 400–409, IEEE, 1996.

24. Deng, X., G. Li, W. Zang and Y. Zhou, "A 2-Approximation Algorithm for Path Coloring on a Restricted Class of Trees of Rings", *Journal of Algorithms*, Vol. 47, No. 1, pp. 1–13, 2003.

25. Bonomo, F., D. Cornaz, T. Ekim and B. Ries, "Perfectness of Clustered Graphs", *Discrete Optimization*, Vol. 10, No. 4, pp. 296–303, 2013.

26. Noronha, T. F. and C. C. Ribeiro, "Routing and Wavelength Assignment by Partition Colouring", *European Journal of Operational Research*, Vol. 171, No. 3, pp. 797–810, 2006.

27. Frota, Y., N. Maculan, T. F. Noronha and C. C. Ribeiro, "A Branch-and-Cut Algorithm for Partition Coloring", *Networks*, Vol. 55, No. 3, pp. 194–204, 2010.

28. Mehrotra, A. and M. A. Trick, "A Column Generation Approach for Graph Coloring", *INFORMS Journal on Computing*, Vol. 8, No. 4, pp. 344–354, 1996.

29. Hoshino, E. A., Y. A. Frota and C. C. De Souza, "A Branch-and-Price Approach

for the Partition Coloring Problem", *Operations Research Letters*, Vol. 39, No. 2, pp. 132–137, 2011.

30. Furini, F., E. Malaguti and A. Santini, "An Exact Algorithm for the Partition Coloring Problem", *Computers & Operations Research*, Vol. 92, pp. 170–181, 2018.

31. Sherali, H. D. and J. C. Smith, "Improving Discrete Model Representations via Symmetry Considerations", *Management Science*, Vol. 47, No. 10, pp. 1396–1407, 2001.

32. Prömel, H. J. and A. Steger, "Almost all Berge Graphs are Perfect", *Combinatorics, Probability and Computing*, Vol. 1, No. 01, pp. 53–79, 1992.

33. Eschen, E. M. and X. Wang, "Algorithms for Unipolar and Generalized Split Graphs", *Discrete Applied Mathematics*, Vol. 162, pp. 195–201, 2014.

34. König, D., "Graphen und Matrizen", *Matematikai Lapok*, Vol. 38, No. 1931, pp. 116–119, 1931.

35. Egerváry, E., "On Combinatorial Properties of Matrices, translated by H.W. Kuhn, Office of Naval Research Logistics Project Report", *Dept. Math. Princeton University*, 1953.

36. Fulkerson, D. and O. Gross, "Incidence Matrices and Interval Graphs", *Pacific Journal of Mathematics*, Vol. 15, No. 3, pp. 835–855, 1965.

37. Blair, J. R. and B. Peyton, "An Introduction to Chordal Graphs and Clique Trees", *Graph Theory and Sparse Matrix Computation*, pp. 1–29, Springer, 1993.

38. Lovász, L., "On the Shannon Capacity of a Graph", *IEEE Transactions on Information theory*, Vol. 25, No. 1, pp. 1–7, 1979.

39. Knuth, D. E., "The Sandwich Theorem", *The Electronic Journal of Combinatorics*,

Vol. 1, No. 1, p. 1, 1994.

40. Grötschel, M., L. Lovász and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Vol. 2, Springer Science & Business Media, 2012.

41. Yıldırım, E. A. and X. Fan-Orzechowski, "On Extracting Maximum Stable Sets in Perfect Graphs using Lovász's Theta Function", *Computational Optimization and Applications*, Vol. 33, No. 2-3, pp. 229–247, 2006.

42. Düzgün, A. c., *The Maximum Stable Set Problem in Perfect Graphs*, Master's Thesis, Boğaziçi University, İstanbul, 2017.

43. Tomita, E., Y. Sutani, T. Higashi, S. Takahashi and M. Wakatsuki, "A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique", *International Workshop on Algorithms and Computation*, pp. 191–203, Springer, 2010.

44. Wu, Q. and J.-K. Hao, "A Review on Algorithms for Maximum Clique Problems", *European Journal of Operational Research*, Vol. 242, No. 3, pp. 693–709, 2015.

45. Tomita, E. and T. Kameda, "An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique with Computational Experiments", *Journal of Global Optimization*, Vol. 37, No. 1, pp. 95–111, 2007.

46. Malaguti, E. and P. Toth, "A Survey on Vertex Coloring Problems", *International Transactions in Operational Research*, Vol. 17, No. 1, pp. 1–34, 2010.

47. Méndez-Díaz, I. and P. Zabala, "A Branch-and-Cut Algorithm for Graph Coloring", *Discrete Applied Mathematics*, Vol. 154, No. 5, pp. 826–847, 2006.

48. Held, S., W. Cook and E. C. Sewell, "Maximum-Weight Stable Sets and Safe Lower Bounds for Graph Coloring", *Mathematical Programming Computation*, Vol. 4, No. 4, pp. 363–381, 2012.

49. Şeker, O., T. Ekim and Z. C. Taşkın, *Instances from Perfect Graph Families*, 2018, `http://www.ie.boun.edu.tr/~taskin/data/scpgf/`, accessed at September 2018.

50. McDiarmid, C. and N. Yolov, "Random Perfect Graphs", *Random Structures & Algorithms*, 2016.

51. Dirac, G. A., "On Rigid Circuit Graphs", *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, Vol. 25, pp. 71–76, Springer, 1961.

52. Buneman, P., "A Characterisation of Rigid Circuit Graphs", *Discrete Mathematics*, Vol. 9, No. 3, pp. 205–212, 1974.

53. Gavril, F., "Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph", *SIAM Journal on Computing*, Vol. 1, No. 2, pp. 180–187, 1972.

54. Gavril, F., "The Intersection Graphs of Subtrees in Trees are Exactly the Chordal Graphs", *Journal of Combinatorial Theory, Series B*, Vol. 16, No. 1, pp. 47–56, 1974.

55. Rose, D. J., R. E. Tarjan and G. S. Lueker, "Algorithmic Aspects of Vertex Elimination on Graphs", *SIAM Journal on Computing*, Vol. 5, No. 2, pp. 266–283, 1976.

56. Pemmaraju, S. V., S. Penumatcha and R. Raman, "Approximating Interval Coloring and Max-Coloring in Chordal Graphs", *Journal of Experimental Algorithmics (JEA)*, Vol. 10, pp. 2–8, 2005.

57. Markenzon, L., O. Vernet and L. H. Araujo, "Two Methods for the Generation of Chordal Graphs", *Annals of Operations Research*, Vol. 157, No. 1, pp. 47–60, 2008.

58. Heggernes, P., "Minimal Triangulations of Graphs: A Survey", *Discrete Mathemat-*

ics, Vol. 306, No. 3, pp. 297–317, 2006.

59. Rodionov, A. S. and H. Choo, "On Generating Random Network Structures: Trees", *International Conference on Computational Science*, pp. 879–887, Springer, 2003.

60. Kobler, J., U. Schöning and J. Torán, *The Graph Isomorphism Problem: Its Structural Complexity*, Springer Science & Business Media, 2012.

61. Lueker, G. S. and K. S. Booth, "A Linear Time Algorithm for Deciding Interval Graph Isomorphism", *Journal of the ACM (JACM)*, Vol. 26, No. 2, pp. 183–195, 1979.

62. Knuth, D. E., "The Art of Computer Programming: Seminumerical Algorithms, volume 2, chapter 4", , 1969.

63. Şeker, O., P. Heggernes, T. Ekim and Z. C. Taşkın, *Chordal Graph Instances*, 2017, `http://www.ie.boun.edu.tr/~taskin/data/chordal/`, accessed at September 2018.

64. Chvátal, V., "Notes on Perfect Graphs", *Progress in Combinatorial Optimization*, pp. 107–115, 1984.

65. McKay, B., *Graphs*, 2016, `http://users.cecs.anu.edu.au/~bdm/data/graphs.html`, accessed at September 2018.

66. Lovász, L., "Normal Hypergraphs and the Perfect Graph Conjecture", *Discrete Mathematics*, Vol. 2, No. 3, pp. 253–267, 1972.

67. Lovász, L., "A Characterization of Perfect Graphs", *Journal of Combinatorial Theory, Series B*, Vol. 13, No. 2, pp. 95–98, 1972.

68. Berge, C. and E. Minieka, *Graphs and Hypergraphs*, Vol. 7, North-Holland Amsterdam, 1973.

69. Bixby, R. E., "A Composition for Perfect Graphs", *Annals of Discrete Mathematics*, Vol. 21, pp. 221–224, 1984.

70. Cunningham, W. H. and J. Edmonds, "A Combinatorial Decomposition Theory", *Canadian Journal of Mathematics*, Vol. 32, No. 3, pp. 734–765, 1980.

71. Şeker, O., T. Ekim and Z. C. Taşkın, *Perfect Graph Instances*, 2018, `http://www.ie.boun.edu.tr/~taskin/data/pg/`, accessed at September 2018.

72. Mittelmann, H. D., *Several SDP-Codes on Sparse and Other SDP Problems*, 2018, `http://plato.asu.edu/ftp/sparse_sdp.html`, accessed at September 2018.

73. Erdős, P. and A. Rényi, "On Random Graphs", *Publicationes Mathematicae Debrecen*, Vol. 6, pp. 290–297, 1959.

74. Watts, D. J. and S. H. Strogatz, "Collective Dynamics of 'Small-World' Networks", *Nature*, Vol. 393, No. 6684, p. 440, 1998.

75. Newman, M. E. and D. J. Watts, "Renormalization Group Analysis of the Small-World Network Model", *Physics Letters A*, Vol. 263, No. 4-6, pp. 341–346, 1999.

76. Barabási, A.-L. and R. Albert, "Emergence of Scaling in Random Networks", *Science*, Vol. 286, No. 5439, pp. 509–512, 1999.

77. Mani, P. and D. Petr, "Clique Number vs. Chromatic Number in Wireless Interference Graphs: Simulation Results", *IEEE Communications Letters*, Vol. 11, No. 7, 2007.

78. Desrosiers, C., P. Galinier and A. Hertz, "Efficient Algorithms for Finding Critical Subgraphs", *Discrete Applied Mathematics*, Vol. 156, No. 2, pp. 244–266, 2008.

79. Gräf, A., M. Stumpf and G. Weißenfels, "On Coloring Unit Disk Graphs", *Algorithmica*, Vol. 20, No. 3, pp. 277–293, 1998.