Studies on Subgraph and Supergraph Enumeration Algorithms

Masashi Kiyomi

DOCTOR OF
PHILOSOPHY

Department of Informatics
School of Multidisciplinary Sciences
The Graduate University for Advanced Studies

2006

Contents

Summary

Enumeration is listing all objects that satisfy given properties. We call enumeration of subgraphs of a given graph, such that those subgraphs have specified properties, as subgraph enumeration. Similarly We call enumeration of supergraphs of a given graph as subgraph enumeration. In this thesis, we will consider about subgraph/supergraph enumeration algorithms. In areas such as data mining or statistics, subgraph enumeration and supergraph enumeration play important roles to find frequent patterns or to draw on some rules satisfied by the inputs, etc.

We developed two types of algorithms of subgraph/supergraph enumeration for chordal and related graphs; one searches graphs to be enumerated by an edge addition or an edge removal; the other defines a neighbor of searching by a simplicial vertex elimination, which is specific for chordal graphs. The first type uses the fact that there are only $O(n^2)$ edges in a complete graph K_n, and achieves polynomial time delay algorithms. We can use this method to develop both subgraph enumeration algorithms and super graph enumeration algorithms. The second type uses nice properties of simplicial vertices and the fact that we can enumerate cliques in a chordal graph quickly. Using this type of algorithm for chordal subgraph enumeration is faster than doing so using the first type (it needs only constant time to enumerate each chordal graph). However, this method is only for the subgraph enumeration.

The organization of this thesis is as follows. We first introduce enumeration, focusing particularly on graph enumeration. Chapter 2 provides the preliminaries, notes about terms that we use in this thesis, and explanations about graph classes. In Chapter 3, we discuss the difficulties of our enumeration problems, and explain the framework of the reverse search method. In Chapter 4, we develop algorithms for our enumeration problems. These algorithms are based on the reverse search method. They are of two types: one defines parents such that the difference between a graph and its parent is exactly one, and the other defines parents such that the parent of a graph is obtained by eliminating a simplicial vertex. And, we conclude the thesis in Chapter 5.

Chapter 1

Introduction

Listing all the objects that satisfy a specified property, with no duplications, is called "*enumeration*". For example, the enumeration of substrings contained by a string "abcab" is "a", "b", "c", "ab", "bc", "ca", "abc", "bca", "cab", "abca", "bcab" and "abcab". Enumeration has many applications in engineerings such areas as data mining, optimization, and statistics, for example to find frequent patterns or to draw on some rules satisfied by the inputs. We sometimes use enumeration to prove mathematical theorems. Proving some mathematical theorems, such as the four-color theorem, requires considering whether they are true in so many cases that the help of computers is needed. In such a case, we use enumeration. In this thesis, we study enumeration with particular focus on graph enumeration: enumerating graphs belonging to some graph classes, such as chordal graphs, interval graphs, etc.

From early on, techniques to enumerate things in good ways are studied. Gray created an encoding of numbers so that successive numbers differ in exactly one bit [19]; this encoding is called "Gray code". The concept that two successive objects differ in small part has been used in enumeration. For example, Wells developed an algorithm to list permutations in this way [43]. Bitner et al. used Gray code to enumerate k-element subsets of an n-element set. It is important for these researches that the successive two enumerated objects differ in small part. With this property, we can efficiently do some computations that need enumeration (see for example [33]), and the question of whether there is a Gray code for a given class is itself an interesting mathematical problem. Classes to be enumerated by this method are often very simple in structure, such as permutations or k-element subsets. Since finding an encoding that satisfies Gray code like properties is difficult and differs for each enumeration problem, little research has treated the enumeration of complicated structures such as the graph classes that we treat in this thesis.

Enumeration is also used widely for solving problems in computer science, in which methods are often very easy, and one does not even recognize enumeration is being used. In the field of combinatorial optimization, we often use the branch-and-bound method to solve integer programming problems. The branch-

ing process does enumeration, enumerating all the feasible solutions that satisfy a given condition. The divide-and-conquer method also uses enumeration. It corresponds to the binary partition method in terms of enumeration. In column generation algorithms or in set covering approaches for some optimization problems, solutions of subproblems are enumerated, and an optimal solution is found by combining them. In combinatorial game theory, we use enumeration to find the best move of a player. Indeed, we enumerate all the possible moves and select the move that obtains the best evaluation value in end game. In these enumerations, the searches are done among tree structures, while enumerations using Gray codes search along path structures. Searching along tree structures enables us to enumerate more complicated structures. However, in these area, researchers have more actively studied how to omit enumeration where it is not needed; they have researched how to cut the feasible domain where optimal solutions never exist. With problems whose feasible domain and objective function are defined strictly, such as integer programming, this is natural, since omitting vain enumeration means that we can solve the problems simply and in a short time.

Recently, due to the increase of computation power, we have come to be able to treat huge amounts of data in practical amounts of time. Additionally, in areas such as genome science, and data mining, enumeration has begun to be used. In these areas, problems are often defined vaguely rather than strictly. Researchers in these areas want to find some meaningful structure in huge data sets [21, 34]. Enumeration algorithms for graph structures such as paths, trees, and cliques are used in frequent pattern mining problems. For example, we can find clusters by enumeration of cliques. Enumeration of bipartite cliques is used in frequent item set mining [39, 2, 1, 26]. In these areas, since problems are not strictly defined, research to obtain many possibly optimal objects or to obtain objects that at least have good properties is important, and enumeration has become a strong tool.

Once enumeration got to be a strong tool, demand surfaced for enumeration to apply more complicated structures. For example, some want to enumerate objects that satisfy some properties and are maximal; others want to enumerate very complicated graph structures in a given graph. Researches to enumerate these complicated structures in short time is thus important.

For that matter, even though we can enumerate objects in a wide class, it does not mean that we can enumerate objects in every subclasses of the class. It is characteristic of enumeration (in contrast, problems in areas such as optimization, can be solved if there is an algorithm for problems of their superclass's). For example, Chapter 4 contains a chordal subgraph enumeration algorithm that enumerates every chordal subgraph in a given graph and does so in a constant time for each, however, there is not developed a constant time interval subgraph enumeration algorithm, although interval graphs are a subclass of chordal graphs. Moreover, although we can of course enumerate every graph of n-vertices in constant time, for many graph classes we do not know whether or not we can enumerate them in constant time. Hence, it is not sufficient to develop an algorithm for solving an enumeration problem that enumerate graphs

in a large class.

Our goal in this thesis is to develop fast algorithms for graph enumeration. In general, the number of objects to be enumerated in an enumeration problem is very large. For example, think about enumeration of spanning trees in complete graph K_n. The number of spanning trees is n^{n-2}. Thus, even if we take only $\mathrm{O}(1)$ time to find each spanning tree, it costs $\Omega(n^{n-2})$ time. We thus need to reduce the time used to output each object in order to keep the total time reasonably short. Moreover, if the number of outputs is polynomial in the input size, enumerating each object in polynomial time in the input size automatically bounds the total time complexity to be polynomial in the input size. In order to use enumeration for solving wide-ranging problems, such as optimization or data mining, enumeration algorithms must be able to enumerate each object in polynomial time in the input size, and the faster this can be done, the better. Enumerating each object in constant time is the best in the sense of time complexity. Thus, we estimate the efficiency of enumeration algorithms by the time complexities for each output. If the delay between every consecutive two outputs is always polynomial in the input size, we call the algorithm "polynomial delay" or simply "polynomial". Even though it is difficult to develop a polynomial delay algorithm for enumeration problems, we can sometimes develop enumeration algorithms whose total time complexities to solve the problem are polynomial in the output size. Such enumeration algorithms are called "output polynomial". Polynomial delay enumeration algorithms are always output polynomial. These criteria can be used to estimate how an algorithm is output-sensitive. We also need to keep the total memory space reasonably small, as is the case for solving other computational problems. As for the space complexity, we use the usual space complexity criterion in enumeration problems, estimating the space complexity by the size of inputs.

Some variations exist in graph enumeration. We consider two of these variations in this thesis: "subgraph enumeration" and "supergraph enumeration". In a subgraph enumeration, we enumerate all the subgraphs of certain type in a given graph. An example is the chordal subgraph enumeration problem, i.e., the problem of listing all the chordal subgraphs in a given graph. Similarly, given a natural number n and a graph $G \in \mathrm{K}_n$, the problem of listing graphs in K_n containing G is the supergraph enumeration problem.

Subgraph and supergraph enumeration have many applications. These types of enumeration are special cases of graph enumeration in complete graphs. We can thus use them for problems such as frequent pattern mining or optimization, as stated above. One application of the subgraph/supergraph enumeration appears in graphical modeling, in which we use graphs to model some systems. The vertices correspond to random variables, and if two variables have a dependency, we connect them by an edge. If we know that graphs of a system belong to certain graph class such as chordal graphs, we can investigate the system by enumerating such graphs. For example, the system corresponding to chordal graphs is known as the decomposable model, and was researched by chordal graph enumeration [13]. However, the number of graphs of n vertices (n corresponds to the number of random variables) belonging to some graph class

is often very large and enumerating all of them is impractical. In such a case, if we know that some variables never have dependencies, we can omit enumerating many systems, and this is done by subgraph enumeration. Similarly, if we know that there must be some dependencies among some variables, we can use supergraph enumeration to reduce the total enumeration time.

Naive algorithms for an enumeration problem often take much time and/or space (time exponential in the output size and/or space exponential in the input size). Developing an output-sensitive enumeration algorithm that uses a small amount of memory is an important task. If we use an algorithm that finds neighbors (under some definition) recursively, the algorithm often needs to store the objects previously output in memory in order to avoid making duplicate outputs. However, when we enumerate exponentially many (in the input size) objects in output-sensitive computational time with a small memory, we have to avoid duplicate outputs without storing previously output objects in memory, since storing them would require the size of the to be exponentially large. Further, simple search strategies may fail with some problems. For example, branch-and-bound type algorithms are not efficient if the subproblems related to the bounding operations are hard. Though it is not easy to develop an efficient algorithm for enumeration problems, efficient algorithms have been provided for some enumeration problems, such as enumerations of vertices of a polytope, all cells in a hyperplane arrangement, spanning trees of a graph [3], maximal cliques of a graph [29] and perfect elimination orderings of a chordal graph [10].

There are some known results about subgraph enumeration. For example, given a graph $G = (V, E)$, we can enumerate paths and cycles in it in polynomial time. The time complexity for one output is $O(|E|)$ [35]. Given a graph G, we can enumerate every tree spanning G in constant time [40]. Here, the number of edges of a spanning tree is $O(|V|)$. Thus, we must need $O(|V|)$ time to output a spanning tree of G in the naive sense. However, if the differences of any two consecutive outputs are in constant size, and the algorithm always takes only a constant time to obtain a graph from the previous graph, we say the algorithm takes constant time to enumerate each graph. After the establishment of the reverse search method for enumeration problems by Avis and Fukuda [3], enumeration algorithms have made a notable amount of progress. Many classes have been proved to be enumerated in polynomial time in the input size. However, many graph classes remain that we do not know whether or not we can enumerate even in polynomial time. Moreover, as for supergraph enumeration problems, little research has been done on them to the best of our knowledge.

In this thesis, we introduce some schemes for graph enumeration for both subgraph enumeration problems and supergraph enumeration problems, and we develop our enumeration algorithms that enumerate each graph in polynomial time using the scheme. The algorithms are for chordal graphs, interval graphs, split graphs, block graphs, Ptolemaic graphs, strongly chordal graphs and weakly chordal graphs. These graph classes (except for chordal graphs itself and weakly chordal graphs) are subclasses of chordal graphs. To the best of our knowledge, our results are the first results about enumeration of these graphs.

Of existing methods for enumeration, the most classical one is giving one-to-one correspondence between each object and a natural number. An example of this method is constructing Gray code [37] for objects to be enumerated. Once we have such correspondence, it is clear that we can easily enumerate the objects. The problem is that finding such one-to-one correspondences is not so easy, and these correspondences are often specific to the individual problems.

The binary partition method is another method for enumeration. In enumerating objects in set S, it divides the problem into two smaller problems: enumerating objects in S_1 and enumerating objects in S_2, such that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$. As for graph enumeration, for example, S_1 is set of graphs that we want to enumerate and includes a certain edge e, and S_2 is that not including edge e. We solve these subproblems recursively until the number of objects to be enumerated is sufficiently small (typically one). This method searches objects on a binary tree structure, while the method of giving one-to-one correspondence with natural numbers searches on a path structure. This method corresponds to the divide and conquer method in combinatorial optimization, and is thus popular for researchers in that field. In fact, the divide and conquer method essentially enumerates feasible solutions by the binary partition method and finds the optimal solution by combining the results. As for the chordal subgraph enumeration problem, and for some other enumeration problems in this thesis, it seems impossible to construct algorithms that enumerate each output graph in polynomial time, by the binary search method. We explain the details about the difficulties in section 3.1.

The reverse search method by Avis and Fukuda [3], which we explain in section 3.2, was a breakthrough in this field. It searches objects on a general tree (or forest) structure. The method was originally developed for enumerating all vertices on a given polytope, and was applied to many enumeration problems for its simplicity and generality. It constructs a spanning tree (or a spanning forest, more generally) among the objects to be enumerated, and searches on the structure for every object. On every node in searching, if we know every branch of the structure incident to the node, we do not have to record the nodes already visited on memory. The reverse search method thus is efficient in memory complexity. It is also efficient in time complexity, if we can develop a good algorithm for finding all branches incident to a given node. In this thesis, we mainly develop reverse search type algorithms to enumerate graphs with polynomial time delay in the input size for each graph.

The organization of this thesis is as follows. We first introduce enumeration, focusing particularly on graph enumeration. Chapter 2 provides the preliminaries, notes about terms that we use in this thesis, and explanations about graph classes. In Chapter 3, we discuss the difficulties of our enumeration problems, and explain the framework of the reverse search method. In Chapter 4, we develop algorithms for our enumeration problems. These algorithms are based on the reverse search method. They are of two types: one defines parents such that the difference between a graph and its parent is exactly one, and the other defines parents such that the parent of a graph is obtained by eliminating a simplicial vertex. And, we conclude the thesis in Chapter 5.

Chapter 2

Preliminaries

2.1 Terms

A graph (V, E) is a set of vertices V and a (multi-)set of edges E. Every edge $e \in E$ connects two vertices in V. If we treat an edge as an set of two vertices, the edge is undirected. If we treat an edge as an pair of the tail vertex and head vertex, the edge is directed. A graph whose edge set consists of undirected edges is undirected, and a graph whose edge set consists of directed edges is directed. If all edges are distinct and every edge consists of two distinct vertices, the graph is simple. All graphs in this thesis are undirected and simple, unless otherwise stated.

We describe the vertex set of graph G as $V(G)$ and the edge set of G as $E(G)$.

Given a graph $G = (V, E)$, the neighbor of vertex $v \in V$ is the set of vertices $\{v' \in V \mid (v, v') \in E\}$.

Graph $G = (V, E)$ is a subgraph of $G' = (V', E')$ if and only if $V \subseteq V'$ and $E \subseteq E'$, and graph G is a supergraph of G' if and only if G' is a subgraph of G. Notice that we treat graph $A = (V_A, E_A)$ and graph $B = (V_B, E_B)$ that are subgraphs of G to be different subgraphs of G when $V_A \neq V_B$ or $E_A \neq E_B$ even if A and B are isomorphic.

Given a graph G that is a subgraph of the n-vertex complete graph K_n, we call finding (without duplications) all subgraphs of G that have a given specified property *subgraph enumeration*. The property is, for example, that the graph is chordal, or that the graph is an interval graph. Similarly, given graph G included in K_n, we call finding (without duplications) all subgraphs of K_n that are supergraphs of G and have specified properties as *supergraph enumeration*. Especially, when the property is that the graph is chordal (resp. an interval graph), we call the problems *chordal (interval) subgraph/supergraph enumeration*. Given an edge set E, it is easy to enumerate every graph whose edge set is E, that may contain some isolated vertices. Therefore, we enumerate graphs by enumeration of edge sets, (see 2.1). For simplicity, given an edge set

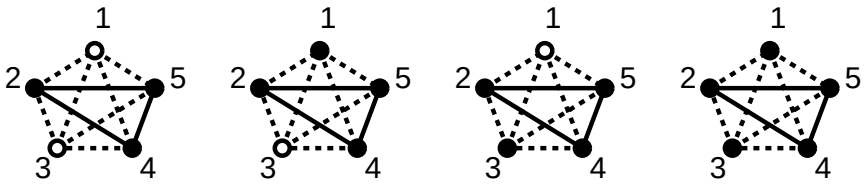

Figure 2.1: All the graphs in K_5 whose edge set is $\{(2,4),(2,5),(4,5)\}$. The filled vertices are in the vertex set, and non-filled vertices are not in the vertex set.

E, we denote by $G\langle E\rangle$ the graph whose vertex set is the set of vertices incident to edges in E and whose edge set is E.

Given a graph $G = (V, E)$, *elimination* of vertex $v \in V$ from G is removing v from V and the edges incident to v from E.

We denote by $G + e$ the graph obtained by adding edge e to the edge set of graph G, and we denote by $G - e$ the graph obtained by removing edge e from the edge set of graph G.

Usually, the size of output of an enumeration problem is very large, often exponential in the input size. The lower bound of the total computation time that an enumeration algorithm takes is thus often exponential in the input size. Hence, we here introduce the terms of time complexities with considering the size of output. If an algorithm terminates in time polynomial of the input size and output size, the algorithm is an **output polynomial time algorithm**. If an algorithm can enumerate each object in polynomial time in the input size, the algorithm is a **polynomial time delay algorithm**. Note that a polynomial time delay algorithm is always an output polynomial time algorithm.

2.2 Graph Classes

In this section, we introduce the definitions and known properties of the graph classes appearing in this thesis.

2.2.1 Chordal Graph

We begin by defining chordal graphs.

Definition 2.1 *A graph is chordal if and only if it has no induced chordless cycle of length more than three (see Figure 2.2) [17].*

Chordal graphs are also called rigid circuit graphs, triangulated graphs, or perfect elimination graphs. Chordal graphs have many good properties; an important one is that a chordal graph has at least one *simplicial vertex*, where a vertex is simplicial if and only if its neighbors induce a clique. Moreover, if the chordal graph has more than one vertex, there are at least two simplicial vertices in

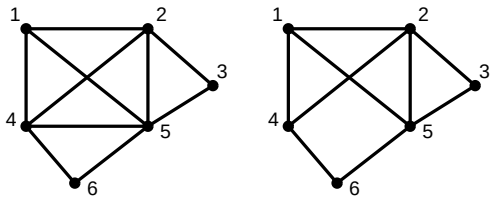

Figure 2.2: The left graph is chordal, while the right graph is not chordal, since $(1, 4, 6, 5)$ and $(2, 4, 6, 5)$ induce chordless cycles.

it [12]. An elimination of a simplicial vertex from a chordal graph results in another chordal graph, and the size of the vertex set of the new chordal graph is exactly smaller than that of the original chordal graph. Thus, we can iteratively eliminate simplicial vertices from a chordal graph until the graph has no vertex. The vertex ordering along which we eliminate the simplicial vertices is called *perfect elimination ordering*. It is known that a vertex ordering of graph $G = (V, E)$ is a perfect elimination ordering if and only if for all three vertices i, j, k in V satisfying $i < j < k$, $(i, j) \in E$ consists, and $(i, k) \in E$ and (j, k) are in the edge set of G. Given a general graph, we can find a perfect elimination ordering of the graph in the linear time in the graph size, if there exists at least one such ordering [36]. We can characterize chordal graphs by perfect elimination orderings; a graph is chordal if and only if it has a perfect elimination ordering. Hence, given a graph, we can recognize whether or not the graph is chordal in linear time in the input size. In Chapter 4, we develop two types of subgraph enumeration algorithms and a supergraph enumeration algorithm for chordal graphs.

2.2.2 Interval Graph

We next introduce interval graphs. an interval graph is a graph that represent relations of intervals. If two intervals has an intersection, we connect the vertices corresponding to the intervals. Interval graphs are widely used on archeology, biology, scheduling etc. [17]

Definition 2.2 *A graph G is an interval graph if and only if there is a one-to-one correspondence between its vertices and a set of intervals on the real line, such that two vertices are adjacent if and only if the corresponding intervals have an intersection [17].*

The set of intervals is called an interval representation of G (Figure 2.3). In general, an interval graph has many interval representations that are not isomorphic (Figure 2.4). We can represent an interval graph by a PQ-tree, which efficiently keeps information about interval representations [8]. Given a PQ-tree representation of an interval graph, we can obtain all the interval representations efficiently from the PQ-tree.

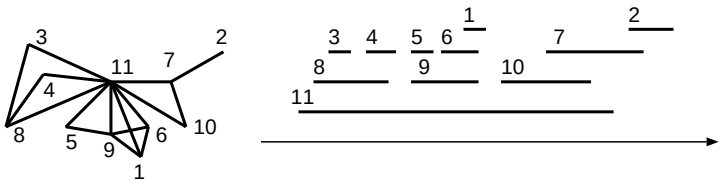

Figure 2.3: An interval graph and its interval representation.

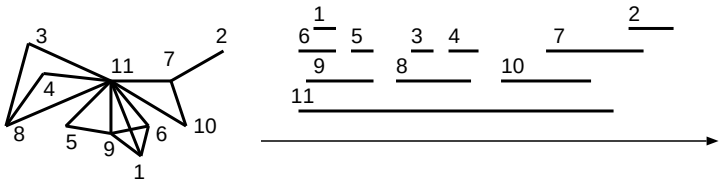

Figure 2.4: Another interval representation of the interval graph in Figure 2.3.

It is also well-known that a graph G is an interval graph if and only if G is chordal and asteroidal triple free, where asteroidal triple is a set of three distinct vertices (v_1, v_2, v_3) such that there exists a path connecting v_i and v_j that contains no neighbor of v_k $(i \neq k \neq j)$, for every combination of $1 \leq i, j, k \leq 3$. Lekkerkerker and Boland showed that a chordal graph is asteroidal triple free, and thus interval, if and only if it does not contain any of the graphs in Figure 2.5 as an induced subgraph [28]. Although we can solve optimization problems on interval graphs with algorithms for chordal graphs since interval graphs are chordal graphs, this is not the case for enumeration problems, since the number of chordal graphs with n vertices is exponentially larger than the number of interval graphs with n vertices, thus enumeration of chordal graphs makes too many redundancies.

Given a graph, we can recognize whether or not the graph is an interval graph in linear time of the input size. Booth and Lueker gave the first linear time algorithm for the recognition [8] using a PQ-tree. Corneil et al. developed simpler linear time algorithm [11]. In Chapter 4, we develop a subgraph enumeration algorithm and a supergraph enumeration algorithm for interval graphs.

2.2.3 Split Graph

Definition 2.3 *A graph is a spit graph if and only if its vertices can be partitioned into an independent set and vertices which induce a clique (see Figure 2.6) [15].*

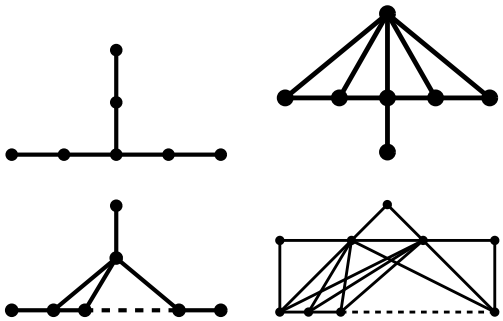

Figure 2.5: A chordal graph is asteroidal triple free if and only if it does not contain any of these graphs as an induced subgraph.

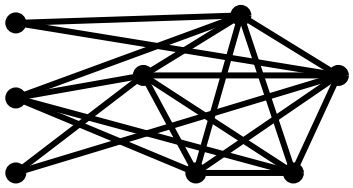

Figure 2.6: An example of a split graph, whose vertices are partitioned into an independent set of three vertices and a clique of five vertices.

For simplicity, given a split graph G, we call a vertex partition of $V(G)$, (V_I, V_C), such that V_I is an independent set and vertices in V_C induce a clique, as an I-C-decomposition of G.

It is known that a chordal graph whose complement is also a chordal graph is equivalent to a split graph [22]. We can recognize whether or not a given graph is a split graph in $O(n+m)$ time, where n is the number of vertices of the given graph and m is the number of the given graph [22]. Bender et al. showed that almost all chordal graphs are split [5].

In Chapter 4, we develop a subgraph enumeration algorithm. Such we can use a subgraph enumeration algorithm can be used for supergraph enumeration, since a split graph is self-complementary.

2.2.4 Block Graph

Definition 2.4 *A graph is a block graph if and only if it is connected and every maximal 2-connected component is a clique [20].*

It is known that a graph is a block graph if and only if the graph is chordal and diamond free, where diamond free means that no vertices of the graph induce the graph shown in Figure 2.7. Given a graph, we can recognize whether or not

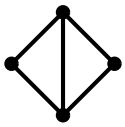

Figure 2.7: diamond.

the graph is a block graph in linear time in the input size. In Chapter 4, we develop a subgraph enumeration algorithm.

2.2.5 Ptolemaic Graph

Definition 2.5 *A connected graph is Ptolemaic if and only any four vertices satisfy the Ptolemaic inequality; that is any four vertices u, v, w, x satisfy*

$$\mathrm{d}(u,v)\ \mathrm{d}(w,x) \leq \mathrm{d}(u,w)\ \mathrm{d}(v,x) + \mathrm{d}(u,x)\ \mathrm{d}(v,w),$$

where $\mathrm{d}(v_1, v_2)$ is the length of the shortest path from v_1 to v_2 [20].

It is known that a graph is Ptolemaic if and only if the graph is chordal and gem free [9], where gem free means that no vertices of the graph induce the graph shown in Figure 2.8. Given a graph, we can recognize whether or not the

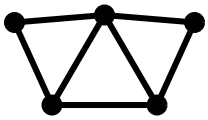

Figure 2.8: gem.

graph is Ptolemaic in linear time in the input size **??**. In Chapter 4, we develop a subgraph enumeration algorithm.

2.2.6 Strongly Chordal Graph

Definition 2.6 *A graph G is strongly chordal if and only if G is chordal and every even cycle of length six or more contains a chord splitting the cycle into two odd length paths (Figure 2.9) [14].*

It is known that a graph G is strongly chordal if and only if it has a strongly perfect elimination ordering. Where strongly perfect elimination ordering of G is a perfect elimination ordering of G and for any four vertices i, j, k, l of G satisfying $i < j < k < l$, if (i, k), (i, l) and (j, k) is an edge of G, (j, l) is also an edge of G. It is known that a graph is strongly chordal if and only if the graph is chordal and sun free [14], where sun free means that no vertices of the graph

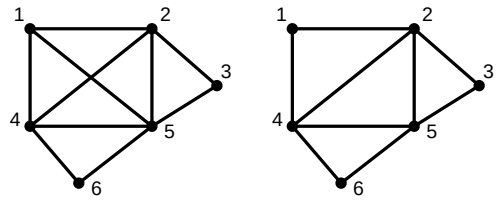

Figure 2.9: The left chordal graph is strongly chordal, while the right chordal graph is not strongly chordal, since the cycle $(1, 2, 3, 5, 6, 4)$ does not have a chord splitting it into two odd length paths.

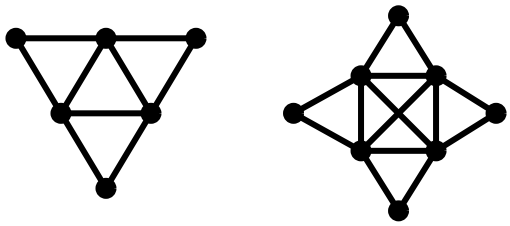

Figure 2.10: Examples of suns. A sun is a graph of $2n$ vertices ($n \geq 3$) whose vertex set can be partitioned into $W = \{w_1, w_2, \ldots w_n\}$ and $U = \{u_1, u_2, \ldots, u_n\}$, such that vertices of W induce a clique, V_2 is an independent set, and u_i is adjacent to w_j iff $i = j$ or $i = j + 1(\mathrm{mod} n)$.

induce "sun" (see Figure 2.10). Given a graph G, we can recognize whether or not G is strongly chordal in $O(\min(n^2, m \log n))$ time. Further, we can obtain a strongly perfect elimination ordering in the recognition process [38]. develop a subgraph enumeration algorithm and a supergraph enumeration algorithm.

2.2.7 Weakly Chordal Graph

Definition 2.7 *A graph G is weakly chordal if and only it is hole free and antihole free, that is, neither G nor the complement of G has any cycle of length more than four as an induced subgraph (Figure 2.11) [23].*

Given a graph $G = (V, E)$, we can recognize whether or not G is weakly chordal in $O(|E|^2)$ time [25]. In Chapter 4, we develop a subgraph enumeration algorithm.

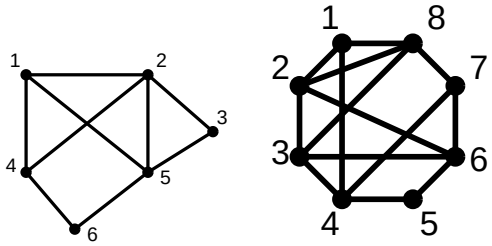

Figure 2.11: The left graph, the example of not chordal in Figure 2.2, is weakly chordal. The right graph is not weakly chordal, since $(1, 7, 2, 4, 6)$ is an anti-hole.

Chapter 3

Enumeration

3.1 Difficulties

Here we discuss about the difficulties of the enumeration of graphs. We often solve optimization problems by enumerating feasible solutions and finding an optimal solution that maximizes the objective function, when we have no other efficient (polynomial time) way to find an optimal solution. These enumerations are done in branch-and-bound type algorithms or divide-and-conquer type algorithms, and the methods are often intuitively understandable. Thus, one may consider enumeration to be easy. However, there are some difficulties in enumeration. The difficulties are deeply related to the structures of problems. Studying the difficulties of each enumeration problem thus helps us to understand the problem.

First, we consider enumeration of chordal graphs of less than or equal to n vertices. A naive algorithm for the problem is to enumerate all graphs of less than or equal to n vertices, check each graph to see whether or not it is chordal, and output only chordal graphs. This method clearly finds all the chordal graphs without duplications. However, this method takes an exponentially long time for each chordal graph to be output on average. This fact comes from the theorem by Bender et al. [5].

Theorem 3.1 (Bender et al.) *The number of chordal graphs of n vertices is asymptotic to*

$$s_n = \sum_{r=1}^{n} \binom{n}{r} 2^{r(n-r)} \quad .$$

Thus, the number of graphs of n vertices is exponentially larger than that of chordal graphs of n vertices. In fact, the number of all graphs of n vertices is

$2^{\frac{n(n-1)}{2}}$. Thus, we estimate the ratio as

$$\frac{2^{\frac{n(n-1)}{2}}}{\sum_{r=1}^{n}\binom{n}{r}2^{r(n-r)}} > \frac{2^{\frac{n(n-1)}{2}}}{\sum_{r=1}^{n}\binom{n}{r}2^{\frac{n}{2}\cdot\frac{n}{2}}} = \frac{2^{\frac{n^2}{4}-\frac{n}{2}}}{\sum_{r=1}^{n}\binom{n}{r}} > 2^{\frac{n^2}{4}-\frac{3}{2}n}.$$

Except for weakly chordal graphs, all the graphs treated in this thesis are chordal graphs (they are chordal graphs or they belong to subclasses of chordal graphs). The difficulties are thus apply to those graphs.

Observation 3.2 *We cannot obtain a polynomial time delay algorithm by generating all the graphs included in* K_n *and output only chordal (interval, etc.) graphs.*

Next, we consider the difficulty of developing a branch-and-bound type algorithm. Let us consider a chordal subgraph enumeration algorithm: given a graph $G = (V, E)$, we enumerate all chordal subgraph of G. At every level of the search, we choose an edge e in E, and divide the problem into two subproblems: an enumeration of chordal subgraphs containing edge e, and an enumeration of those not containing edge e. We stop dividing the problem when we know that there is no chordal subgraph in a subproblem. In this way, we can enumerate all chordal subgraphs of G by outputting all leaves of the search tree. The problem is that deciding whether or not there is a chordal subgraph in a subproblem is difficult. Let A be the current graph in the algorithm and B be the input graph. The problem to decide whether or not there is an chordal graph containing A and included in B is known as "graph sandwich problem". The graph sandwich problem is proved to be NP-complete.

Theorem 3.3 (Graph sandwich by Golumbic et al. [18]) *Given two graphs* A *and* B*, deciding whether or not there is a chordal graph* C *satisfying* $A \subseteq C \subseteq B$ *is NP-complete.*

The algorithm thus possibly takes an exponentially long time at the bounding phase, or the algorithm visits exponentially many subgraphs of G that are not chordal, unless P = NP. Hence, the algorithm may take an exponentially long time to output a chordal subgraph.

Observation 3.4 *It is hard for a naive binary partition algorithm to enumerate all chordal subgraphs of a given graph with polynomial time delay, unless* P = NP.

Golumbic et al. also showed that a graph sandwich problem for interval graphs, deciding whether or not there is an interval graph included in one given graph and containing another given graph, is NP-complete.

Theorem 3.5 (Graph sandwich by Golumbic et al. [18]) *Given two graphs* A *and* B*, deciding whether or not there is an interval graph* C *satisfying* $A \subseteq C \subseteq B$ *is NP-complete.*

The interval subgraph enumeration thus has a similar difficulty. In contrast, a graph sandwich problem for split graphs can be solved in polynomial time.

Theorem 3.6 (Graph sandwich by Golumbic et al. [18]) *Given two graphs A and B, we can decide whether or not there is a split graph C satisfying $A \subseteq C \subseteq B$ in time linear in the size of A and B.*

Thus, we can enumerate every split subgraph of G in polynomial time in the size of G by the binary partition method. Given a graph $G = (|V|, |E|)$, to obtain a split subgraph, we may possibly solve the graph sandwich problems $|E|$ time. We thus obtain the theorem below.

Theorem 3.7 *Given a graph $G = (|V|, |E|)$, we can enumerate every split subgraph of G in $O(|E| \times (|V| + |E|))$ time by the binary partition method.*

Next, we consider a graph search type algorithm. Suppose that, from a graph, we search graphs by adding or removing an edge (see Figure 3.1). This corresponds to that we consider a meta-graph whose vertices are graphs that we want to enumerate, and search on the meta-graph by some graph algorithm, where two graphs are connected by a directed edge if one of them is obtained by adding/removing an edge to/from the other. When such a type of algorithm

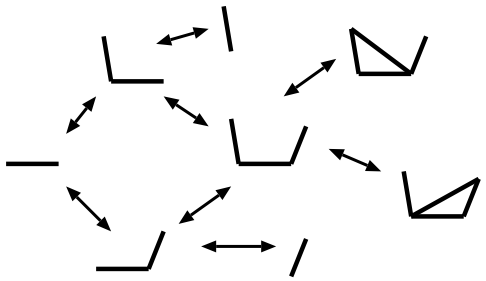

Figure 3.1: Searching graphs by adding or removing an edge.

is used, we must pay attention to ensure that the algorithm enumerates all the graphs we want (i.e., we have to make sure that the algorithm reach every vertex of the meta-graph). Even if the algorithm can enumerate every graph (reach every vertex of the meta-graph), those algorithms often require a huge size of memory, since we have to memory which graphs we have already output (which vertices on the meta-graph we have already searched) in order not to output identical graphs multiple times. If the number of graphs we want to enumerate is exponentially many, the necessary memory size gets exponentially large.

Observation 3.8 *A naive depth-first type algorithm requires exponentially large memory for chordal (interval, etc.) enumeration.*

3.2 Reverse search

3.2.1 Algorithm

For efficient enumerations, a good search method is necessary. The reverse search method suits this requirement. The reverse search is a sophisticated depth-first search type scheme for enumerations and was originally developed by Avis and Fukuda in [3]. Because of its simplicity and efficiency, the reverse search has been used in many algorithms for problems in many fields [3, 30, 31, 29].

Let $\mathcal{F}$ be the set of objects that we want to enumerate. For example, $\mathcal{F}$ is the set of chordal graphs included in a given graph. We define a parent-child relation by determining a parent for each object except for some specified objects called *root objects*, which do not have parents. The definition of the parents has to satisfy that no object is a proper ancestor of itself, that is, by starting from an object x and moving in continuing succession to its parent and its parent's parent, we never come to the start object x again. The graph representation of the relation induces a set of disjoint rooted trees spanning all objects in $\mathcal{F}$, in which paths from all leaves aim to the roots. Figure 3.2 illustrates an example of the graph representation. Each object to be enumerated is drawn by a point, and an object and its parent are connected by a directed arrow.

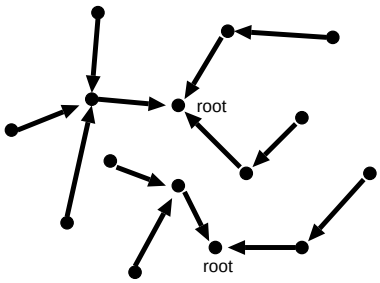

Figure 3.2: Spanning forest on the objects to be enumerated. Paths from all leaves aim to the roots.

Tracing each edge in the reverse direction from the root objects enables us to perform a depth-first search to visit all objects. We can thus find all objects without duplications by using two subalgorithms: one algorithm to find all root objects, and another to find all children of an object. We often have to store the nodes that we have visited in memory to avoid visiting them twice when performing depth-first search algorithms. In contrast with such cases, reverse search needs to store only the nodes on the path from the root in memory by limiting the base structure for searching as tree (forest) structure. In reverse searching, we only need to store objects in the path that connect the current object and its root.

3.2.2 Time Complexity

The time complexity of the reverse search method depends on that of finding children of each object. If we can find each child in T time, or if there is no child, we can know that in T time, the time complexity of the reverse search algorithm is T time for each on average. However, in a naive implementation of reverse search method, computation time between an object is output and the next one is output sometimes exceeds $O(T)$ time. For example, consider the case that T is $O(1)$ and the depth of the search tree is T' which is larger than $O(1)$. Then, after outputting a leaf object, we need to go back to the root object and find the next child. This takes $\Omega(T')$ time, since the length of the path we have to retrace is possibly T'. However, we can search every object in exactly $O(T)$ time, by revising the algorithm in this case. We next describe the revising by Nakano and Uno [41, 32].

If we have an enumeration algorithm traversing a search tree such that any parent and its child differ by a constant size, the maximum difference between two consecutive outputs can be bounded in a constant size with only a modification on the timing of output. The modification is that at the odd level of the recursion we output the objects before making recursive calls, and at the even level of the recursion, we output after the terminations of the recursive calls. In this way, at least one of three iterations outputs an object when the algorithm ascends or descends the search tree. If each iteration takes a constant amount of time to make a recursive call, the delay is also a constant amount of time.

Chapter 4

Algorithms

In this chapter, we describe our enumeration algorithms, for chordal graphs and its subclasses, based on reverse search. We developed two types of algorithms. One defines the parent-child relation on all graphs to be enumerated such that the difference between a child and its parent is exactly one edge. The other defines parents such that the parent of a graph is obtained by eliminating a simplicial vertex. There are many graph classes whose non-empty graphs are always able to be obtained by adding an edge to another graph of the class. There are also many graph classes whose graphs are always able to be obtained by removing an edge from another graph of the class. For these graph classes, we can develop the first type of algorithm. The second type of algorithm requires that graphs to be enumerated are chordal (subclasses of chordal graphs are possible), since otherwise, it is possible that some graphs to be enumerated have no simplicial vertex and we cannot define the parents of the graphs. We can use the first type of algorithm for both subgraph enumerations and supergraph enumerations, while the second one can be used only for subgraph enumerations (The exception is the case of split graphs. The subgraph enumeration of split graphs is automatically applied for the supergraph enumeration of split graphs). However, the second type of algorithm is much faster in the case of chordal subgraph enumeration.

4.1 Parent-Child Relation by Edge Removal/Addition

In this section, we develop an enumeration scheme by defining a parent in a reverse search by an edge removal or an edge addition. As described in Section 3.2.1, algorithms based on the reverse search method first find roots, and then search for the children recursively. In the case of the edge addition, the root object is K_n, and in the case of the edge removal, the root object is the empty graph. The idea of generating children is that we generate, from a graph H, every graph H' obtained by adding or removing an edge as a candidate of a child, then check if H' is a graph to be enumerated (e.g. if H' is chordal,

or if H' is an interval graph), and if so, check if the parent of H' is H. Only when the parent of H' is H and H' is a child of H, the algorithm outputs H' and then continues the recursion. There are only $O(n^2)$ edges in K_n, thus, even if we examine all the edges to be added, we only take polynomial time in the input size, provided that we can find the parent of any graphs in polynomial time. This scheme is very general. When we can prove only that we can always add an edge to or remove an edge from a graph that we are enumerating, the scheme works. Thus, we can show that many graphs have polynomial time delay enumeration algorithms. However, checking $O(n^2)$ edges costs at least $O(n^2)$ time. It is thus difficult to develop a constant time enumeration algorithm by this scheme.

4.1.1 Chordal/Interval Supergraph Enumeration

Our chordal supergraph enumeration algorithm proceeds as follows. Given a chordal graph H, we define the parent of H as a graph obtained by adding an edge to H, and we define the root as n vertex complete graph K_n. This definition involves two problems. The first problem is "Is there always a chordal graph obtained by adding an edge to H?", and the second problem is "If there are several chordal graphs obtained in such a way, which one should we choose?" We prepare a lemma below to answer these questions.

Lemma 4.1 *Given an n-vertex chordal graph $H = (V, E) \neq K_n$ and any vertex $v \in V$ of degree smaller than $n-1$, there exists a vertex $v' \in V$ not adjacent to v such that graph $H + (v, v')$ is chordal. Moreover, we can find such vertex v' in $O(n+m)$ time, where m is $|E|$.*

Proof It is known that there exists a perfect elimination ordering of H such that the vertex v is located at the tail[36]. We denote such a perfect elimination ordering by $P = (p_1, p_2, \ldots, p_n)$, where $p_n = v$. Let p_k be the last vertex in P that is not adjacent to v. We show that p_k is the desired vertex v' by proving that P is a perfect elimination ordering of $H' = H + (v, p_k)$ and H' is thus chordal.

We denote by H_j the subgraph of H induced by $\{p_j, p_{j+1}, \ldots, p_n\}$, and we denote by H'_j the subgraph of H' induced by $\{p_j, p_{j+1}, \ldots, p_n\}$. We denote by $N_i(v)$ (resp. $N'_i(v)$) the set of neighbors of vertex v in H_i (resp. H'_i).

Vertex p_i $(i = 1, 2, \ldots, k-1,\ k+1, k+2, \ldots, n)$ is a simplicial vertex of H'_i, for $N_i(p_i)$ and $N'_i(p_i)$ are identical. Since p_k is a simplicial vertex of H_k, $N_k(p_k)$ induces a clique in H_k and also in H'_k. Since all the vertices of $N_k(p_k)$ are adjacent to vertex p_n, $N'_k(p_k) = N_k(p_k) \cup \{p_n\}$ also induces a clique in H'_{p_k}. Hence, vertex p_k is a simplicial vertex of H'_{p_k}. P is thus a perfect elimination ordering of H'.

We can obtain a perfect elimination ordering P such that v is located at the tail in $O(n+m)$ time [36]. To find p_k from the ordering P, we need $O(n)$ time. Thus, we can find the desired v' in $O(n+m)$ time. □

Let A be an algorithm obtained from the proof of Lemma 4.1; A inputs a chordal graph H, and outputs an edge $e^*(H) = (v, v')$ such that v is the

youngest vertex with degree smaller than $n-1$ and $H+(e^*(H))$ is chordal. We can assume that A outputs the identical edge for identical graphs, since the pair of a graph with vertex indices can be represented in a canonical way. Let $e^*(H)$ be the output edge by A when we input H. We define the parent of a chordal graph $H \neq \mathrm{K}_n$ as $H + e^*(H)$. The definition is thus unique and well-defined.

Lemma 4.2 *For every chordal graph $H \neq \mathrm{K}_n$, the parent always exists and is unique.*

Lemma 4.3 *The parent-child relation induces a rooted tree, whose root is the complete graph K_n.*

We describe an outline of our algorithm below. We have to only call the procedure with the argument $H = \mathrm{K}_n$.

```
procedure enum_super_chordal(H, G)
H : chordal graph, G : graph;
begin
  output H;
  for every edge e ∈ E(H) \ E(G) do
    if H − e is chordal and
      the parent of H − e is H then
       enum_super_chordal(H − e, G);
  end for
end.
```

The algorithm performs a depth-first back tracking to enumerate all the chordal supergraphs included in K_n. From the lemmata above, we can easily confirm that this algorithm finds all the chordal supergraphs of G without duplications. We consider the complexity in the theorem below.

Theorem 4.4 *There is an algorithm for the chordal supergraph enumeration. The time complexity of the algorithm to find each chordal supergraph of a given graph is $O(n^3)$ time, and the space complexity is $O(n^2)$.*

Proof Given a chordal graph H, let v^* be the youngest vertex that has degree smaller than $n-1$.

For edge e incident to a vertex younger than v^* such that $H-e$ is chordal, the parent of $H-e$ is always H, by the definition of the parent. If more than one vertices are younger than v^*, these vertices induce a clique, since the degrees of them are $n-1$. We consider any two of such vertices, v_1 and v_2. From the symmetric property, if and only if edge e_1, which is incident to vertex v_1, and some vertex v satisfies that $H-e_1$ is chordal, $H-e_2$ is chordal, where edge e_2 is incident to vertex v_2 and v. Thus, we have to check chordality on only $\mathrm{O}(n)$ graphs to check, for every edge e incident to vertices younger than v^*, whether or not $H-e$ is chordal. We can do these checks in $\mathrm{O}(n^3)$ time, since we can check chordality of a graph in $\mathrm{O}(n^2)$ time. For edge $e = (v^*, v^+)$, where v^+ is elder than v^*, such that $H-e$ is chordal, we can check whether or not the

parents of $H - e$ is H in $O(n^2)$ time, since we can obtain the parent of $H - e$ in $O(n^2)$ time by Theorem 4.1. Hence, for all edges $e = (v^*, v^+)$, we can check whether or not $H - e$ is a child of H, i.e., $H - e$ is chordal and the parent of $H - e$ is H, in $O(n^3)$ time.

For edge e whose end points are elder than v^*, the parent of $H-e$ is never H, since we can add to $H - e$ an edge whose one end point is v^* keeping chordality by Theorem 4.1.

Hence, the total time complexity to check for every candidate of a child of H whether or not it is really child is $O(n^3)$. □

The algorithm and the analysis technique can be applied similarly to the case of the interval supergraph enumeration, since Lemma 4.5, which we show below, plays the same role as Lemma 4.1. We can thus enumerate interval graphs that contain given graph G, and the time complexity for finding every interval supergraph is $O(n^3)$.

Lemma 4.5 *Given an n-vertex interval graph $I = (V, E) \neq K_n$ and any vertex $v \in V$ of degree smaller than $n - 1$, there exists a vertex $v' \in V$ not adjacent to v such that graph $I + (v, v')$ is an interval graph. Moreover, we can find such vertex v' in O(n+m) time, where m is $|E|$.*

Proof We denote an interval representation of I by $(I_1, I_2, \ldots, I_n)$, where each interval I_i $(i = 1, 2, \ldots, n)$ corresponds to vertex $v_i \in V$. We can assume without loss of generality that all of the end points of the intervals are distinct. Let vertex v_j be the vertex v.

Since the degree of vertex v is smaller than $n-1$, the corresponding interval $I_j = [l_j, r_j]$ does not intersect some intervals. We can assume without loss of generality that there are some intervals at the right-side of I_j that do not intersect I_j, since the symmetric intervals also form an interval representation of interval graph I. Let $I_k = [l_k, r_k]$ be the interval such that the difference between r_j and the left end is the smallest among such intervals. We change the interval I_j to $[l_j, l_k + \epsilon]$, where ϵ is a sufficiently small number (Figure 4.1). Then, the resulting interval representation is that of an interval graph $I+(v, v_k)$. Hence, the vertex v_k is the desired vertex v'.

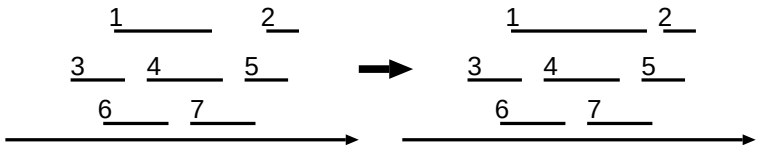

Figure 4.1: We can add a new edge incident to vertex 1.

We can obtain an interval representation $(I_1, I_2, \ldots, I_n)$ of I in $O(n + m)$ time [8, 11], and to find I_k from the representation, we need $O(n)$ time. We can thus find the desired v' in $O(n + m)$ time. □

Theorem 4.6 *There is an algorithm for the interval supergraph enumeration. The time complexity to find each interval supergraph of a given graph is $O(n^3)$ and the space complexity is $O(n^2)$.*

These algorithms and the analysis can be applied to the connected chordal/interval supergraph enumeration, as the parents of connected chordal/interval graphs are always connected.

4.1.2 Chordal/Interval Subgraph Enumeration

A similar type of algorithm can be considered for subgraph enumeration. Given a chordal graph H, we define the parent of H as a graph obtained by removing an edge from H, and we define the root as an empty graph $(\emptyset, \emptyset)$. Similar problems to those in chordal and interval supergraph enumeration arise: "Is there always a chordal graph obtained by removing an edge from H?" and "If there are multiple chordal graphs obtained in such a way, which one should we choose?". We give a lemma below to answer these problems.

Lemma 4.7 *Given an n-vertex connected chordal graph $H = (V, E)$ ($|E| \geq 1$), there exists an edge $e \in E$ such that $G\langle E \setminus \{e\}\rangle$ is a connected chordal graph. Moreover, we can find the edge e in $O(n+m)$ time, where m is $|E|$.*

Proof Let v be a simplicial vertex of H, and let $\bar{e}$ be an edge in E adjacent to v. If the degree of v is one, $G\langle E \setminus \{\bar{e}\}\rangle$ is a connected chordal graph obtained from H by eliminating simplicial vertex v. Otherwise, $G\langle E \setminus \{\bar{e}\}\rangle$ is connected and is chordal, since v is a simplicial vertex of $G\langle E \setminus \{\bar{e}\}\rangle$ and the graph obtained from $G\langle E \setminus \{\bar{e}\}\rangle$ by eliminating simplicial vertex v and the graph obtained from H by eliminating simplicial vertex v are identical. Hence $\bar{e}$ is a desired edge e.

To find e, we have only to find a simplicial vertex of H. The time complexity is thus $\mathrm{O}(n + m)$. □

In the case of connected chordal graph enumeration, the parent chordal graph of connected chordal graph H that is not the empty graph is a connected chordal graph obtained by removing edge $e^*(H)$ from H, where $e^*(H)$ is an edge obtained by the algorithm derived from Lemma 4.7. We define the root as the empty graph.

Now, we present our algorithm below. We have only to call the procedure with the argument $I = (\phi, \phi)$, i.e., the empty graph.

```
procedure enum_sub_chordal(H, G)
H : chordal graph, G : graph;
begin
  output H;
  for every edge e ∈ E(G) \ E(H) do
    if H + e is connected chordal and
      the parent of H + e is H then
       enum_sub_chordal(H + e, G);
  end for
end.
```

For graph H, we have to check $O(n + m)$ candidates of children. We can find the parent of the given chordal graph in $O(n + m)$ time, by Lemma 4.7. Thus, computational time of this algorithm to find each connected chordal graph included in graph G is $O((n+m)^2)$. In contrast with the case of the supergraph enumeration, we cannot remove an edge incident to an arbitrary vertex keeping the intervality. We thus can not reduce the time complexity for finding each connected chordal subgraph with this algorithm.

Notice that the (not necessarily connected) chordal subgraph enumeration algorithm is straight-forward from that of the connected version, since we can define a parent of a given (not necessarily connected) chordal graph as the parent of its one connected component determined by a certain deterministic algorithm.

Theorem 4.8 *There is an algorithm for the (connected) chordal subgraph enumeration. The time complexity of the algorithm to find each chordal subgraph of a given graph is $O((n + m)^2)$ and the space complexity is $O(n + m)$.*

A similar theorem can be obtained for interval subgraph enumeration. We give a lemma for this below.

Lemma 4.9 *Given an n-vertex connected interval graph $I = (V, E)$ ($|E| \geq 1$), there exists an edge $e \in E$ such that $G\langle E \setminus \{e\}\rangle$ is a connected interval graph. Moreover, we can find the edge e in $O(n+m)$ time, where m is $|E|$.*

Proof We denote an interval representation of I by $(I_1 = [l_1, r_1], I_2 = [l_2, r_2], \ldots, I_n = [l_n, r_n])$, where each interval $I_i (i = 1, 2, \ldots, n)$ corresponds to vertex $v_i \in V$. We can assume without loss of generality that all of the end points of the intervals are distinct. Let l_j be the smallest among l_i $(i = 1, 2, \ldots, n)$.

Since I is connected, $I_j = [l_j, r_j]$ intersects some intervals. Let $I_k = [l_k, r_k]$ be the interval that intersects I_j and the value l_k be the largest among such intervals. Setting r_j to $l_k - \epsilon$ produces a new interval graph that we obtain by removing an edge between vertices v_j and v_k. If the interval graph is connected, (v_j, v_k) is the desired edge. Otherwise, the graph is divided into two connected interval graphs, where the left-hand one contains I_j. Then, doing the process recursively with the right-hand intervals, we can find the desired edge eventually. At the end of the recursion, it is possible that the graph is divided into an isolated vertex and the rest of the graph. However, in the sense of an edge set, the graph is connected.

It takes $O(n+m)$ time to obtain an interval representation of I. We can find every k at every iteration by sweeping intervals from l_1 until we find the desired vertex. Each interval I_i $(i = 1, 2, \ldots, n)$ is swept at most once. Therefore, we can find all k's at every iteration in $O(n)$ time. Thus, the time complexity to find e is $O(n + m)$. □

This lemma plays the same role as Lemma 4.7, and we come to the lemma below.

Theorem 4.10 *There is an algorithm for the (connected) interval subgraph enumeration. The time complexity to find each interval subgraph of a given graph is $O((n+m)^2)$ and the space complexity is $O(n+m)$.*

4.1.3 Strongly Chordal Subgraph Enumeration

A subgraph enumeration algorithm can be developed for strongly chordal graphs as well. Similarly to the chordal/interval subgraph enumeration, we define the parent of a strongly chordal graph H, which is not an empty graph, as a graph obtained by removing an edge from H. Similar problems arise: "Is there any edge e such that a graph obtained by removing e from H is strongly chordal?" and "If there are some such edges, which one should we choose?"

To answer the first question, we prove the lemma below.

Lemma 4.11 *Given a non-empty strongly chordal graph H, there is an edge e such that $H-e$ is strongly chordal.*

Proof Since H is strongly chordal, H has a strongly perfect elimination ordering P; for any four vertex indices i,j,k,l of H satisfying $i<j<k<l$, if (v_i,v_k), (v_i,v_l) and (v_j,v_k) are edges of H, (v_j,v_l) is also an edge of H. Let v be the first vertex in P of degree at least one, and let v' be the first vertex to which v is adjacent. P is a strongly perfect elimination ordering of $H-(v,v')$, since for any four vertex indices i,j,k,l of $H-(v,v')$ satisfying $i<j<k<l$, if (v_i,v_k), (v_i,v_l) and (v_j,v_k) are edges of G, (v_j,v_l) is also an edge of $H-(v,v')$. (v,v') is thus the desired edge e. □

The proof gives us an algorithm to obtain an edge $e^*(H)$ such that $H-e^*(H)$ is strongly chordal. The time complexity to obtain $e^*(H)$ is $\mathrm{O}(\min(m\log n, n^2))$, since we can compute a strongly perfect elimination ordering of H in that time [38]. We define the parent of a strongly chordal graph H as strongly chordal graph obtained by removing the edge $e^*(H)$. We show our subgraph enumeration algorithm below.

```
procedure enum_sub_strongly_chordal(H, G)
H : strongly chordal graph, G : graph;
begin
  output H;
  for every edge e ∈ E(G) \ E(H) do
    if H + e is strongly chordal and
      the parent of H + e is H then
       enum_sub_strongly_chordal(H + e, G);
  end for
end.
```

Theorem 4.12 *There is an algorithm for the strongly chordal subgraph enumeration. The time complexity to find each strongly chordal subgraph of a given graph is $O(m\cdot\min(m\log n, n^2))$ and the space complexity is $O(n+m)$.*

4.1.4 Strongly Chordal Supergraph Enumeration

For the super graph enumeration of strongly chordal graph, we have to prove that we can add an edge to any strongly chordal graph $H \neq \mathrm{K}_n$ keeping the strongly chordality.

Lemma 4.13 *Given a strongly chordal graph $H \neq \mathrm{K}_n$, there is an edge e such that $H + e$ is strongly chordal.*

Proof There is a strongly perfect elimination ordering $P = (p_1, p_2, \ldots, p_n)$ of H. Let p_k be the last vertex in P that is not adjacent to p_n. We show that P is a strongly perfect elimination ordering of $H' = H + (v, p_k)$ and H' is thus chordal.

Since a strongly perfect elimination ordering is a perfect elimination ordering, P is a perfect elimination ordering of H' by the proof of Lemma 4.1.

Hence, we have only to prove that

$$\begin{array}{l}\text{for any four vertices } i, j, k, l \text{ of } H' \text{ satisfying } i < j < k < l, \\ \text{if } (i, k), (i, l) \text{ and } (j, k) \text{ are edges of } H', (j, l) \text{ is also an edge of } H'.\end{array} \tag{4.1}$$

It is sufficient for the proof to consider the case in which (p_k, p_n) corresponds to (i, l), since p_n is the tail of P. By the definition of p_k, every vertex $p_{k+1}, \ldots, p_{n-1}$ is adjacent to p_n. Hence, (j, l) is always an edge of H'. □

The proof gives us an algorithm to obtain an edge $\bar{e}(H)$ such that $H + \bar{e}(H)$ is strongly chordal. The time complexity to obtain $\bar{e}(H)$ is $\mathrm{O}(\min(m \log n, n^2))$, as we can compute a strongly perfect elimination ordering of H in that time [38]. We define the parent of a strongly chordal graph H as strongly chordal graph obtained by removing the edge $\bar{e}(H)$.

Theorem 4.14 *There is an algorithm for the strongly chordal supergraph enumeration. The time complexity to find each strongly chordal supergraph of a given graph is $O(m \cdot n^2)$ and the space complexity is $O(n^2)$.*

4.1.5 Weakly Chordal Subgraph Enumeration

A subgraph enumeration algorithm can also be developed for weakly chordal graphs by defining the parent as a graph that differs in only one edge from the children. We define the parent of a weakly chordal graph H, which is not an empty graph, as a graph obtained by removing an edge from H.

To ensure that we can always remove an edge keeping weakly chordality, we introduce a theorem by Hayward [24].

Theorem 4.15 (Hayward) *A graph is weakly chordal if and only it can be generated in the following manner:*

- *Start with a graph G_0 with no edges.*
- *Repeatedly add an edge e_j to G_{j-1} to create the graph G_j, such that e_j is not the middle edge of any P_4 of G_j, where P_4 is a path of four vertices.*

By the theorem, it is clear that, given a weakly chordal graph H, there is at least one edge e such that the graph obtained by removing e from H is weakly chordal.

Then, if there are some edges that we can remove, which one should we remove? We can simply choose the youngest edge of such edges in order to obtain an algorithm that enumerates each weakly chordal graph in polynomial time. The algorithm is shown below.

```
procedure enum_sub_weakly_chordal(H, G)
H : weakly chordal graph, G : graph;
begin
  output H;
  for every edge e ∈ E(G) \ E(H) do
    if H + e is weakly chordal and
      the parent of H + e is H then
       enum_sub_weakly_chordal(H + e, G);
  end for
end.
```

The time complexity is somehow large compared to the other algorithms. Recognizing whether or not $H+e$ is weakly chordal require m^2 time, and finding the parent of $H+e$ requires m^3 time, since we have to determine for every edge e' of H younger than e whether or not $H + e - e'$ is weakly chordal. The time complexity of our weakly chordal subgraph enumeration algorithm is thus $\mathrm{O}(m^4)$ for each output. Theorem 4.15 does not say that given a weakly chordal graph G, $G - e$ is weakly chordal if edge e is not a middle edge of any P_4 of G.

Theorem 4.16 *There is an algorithm for the weakly chordal subgraph enumeration. The time complexity to find each weakly chordal subgraph of a given graph is $O(m^4)$ and the space complexity is $O(n + m)$.*

4.1.6 Split Subgraph Enumeration

We address a subgraph enumeration of split graphs in this subsection. We showed in Theorem 3.7 that there is an $\mathrm{O}(m(n + m))$ time algorithm for the split subgraph enumeration problem. Moreover, a faster algorithm can easily be developed if the input graph is a complete graph. In fact, since the number of n-vertex split graphs is asymptotic to the number of n-vertex chordal graphs [5], we can enumerate split graphs by enumerating chordal graphs and checking to see whether or not each graph is a split graph. A graph is a split graph if and only if both it and its compliment are chordal. The time complexity to determine whether or not a graph is a split graph is thus equivalent to the time complexity to find out whether or not a graph is chordal, and is $\mathrm{O}(n^2)$. Hence, given a graph G, we can enumerate each split graph included in G in $\mathrm{O}(n^2)$ time on average with the algorithm in Section 4.2.1, provided n is sufficiently large. However, with the algorithm, time delay between two consecutive objects being output is not always $\mathrm{O}(n^2)$, since it is possible that many, say $\mathrm{O}(n)$ or more,

consecutive chordal graphs generated by the algorithm in Section 4.2.1 are not split graphs. We introduce another faster algorithm here; it enumerates each split subgraph of arbitrary graph G in exactly $O(n)$ time for every n.

Given a split graph H of at least one edge, we define the parent of H as a graph obtained by removing an edge e from H. First, we show that there is at least one edge e such that $H - e$ is a split graph.

Lemma 4.17 *Given an n-vertex split graph $H = (V, E)$ of at least one edge and any vertex $v \in V$ of degree at least one, there exists a vertex $v' \in V$ adjacent to v in H such that $H - (v, v')$ is a split graph.*

Proof This regards an I-C-decomposition (V_I, V_C) of H.

If v is a vertex in V_I, we can remove every edge e incident to v keeping graph $H - e$ as a split graph, as V_I is still an independent set and the vertices in V_C induce a clique, after we remove edge e.

Next, consider the case that v is a vertex in V_C. If v is adjacent to a vertex $\bar{v}$ in V_I, we can remove the edge $(v, \bar{v})$ keeping graph $H - e$ as a split graph by the same reason as in the case that v is in V_I. If v is not adjacent to any vertices in V_I, $V_I' = V_I \cup \{v\}$ is an independent set and vertices in $V_C' = V_C \setminus \{v\}$ induce a clique. We can thus remove any edge e incident to v keeping graph $H - e$ as a split graph by the same reason of the case v is in V_I.

Hence, there exists a vertex $v' \in V$ adjacent to v in H such that $H - (v, v')$ is a split graph. □

We define the parent of a split graph H having at least one edge as a graph obtained by removing an edge $e = (v, v')$ from H, such that v is the youngest vertex of degree at least one. If there are two or more such edges, we choose v' to be the youngest.

In order to develop a fast algorithm, we consider some properties of split graphs. We first categorize the vertices of a split graph. Given a split graph $H = (V, E)$, the vertex set V is divided into two sets V_I and V_C; V_I is an independent set, and vertices of V_C induce a clique. In general, V_I is not unique, that is there is a redundancy in the method of selection of V_I. Thus, it is possible that there are some different I-C-decompositions of H. We prepare another partition $I(H)$, $C(H)$, and $M(H)$ of vertices that has no redundancy.

Definition 4.18 *Given a split graph $H = (V, E)$, we categorize vertices in V as below.*

$I(H)$ *the vertex set whose elements are always in V_I for any I-C-decomposition (V_I, V_C) of H,*

$C(H)$ *the vertex set whose elements are always in V_C for any I-C-decomposition (V_I, V_C) of H, and*

$M(H)$ *the vertices in H, and neither in $I(H)$ nor in $C(H)$.*

We call the partition "I-C-M-decomposition" of H. We investigate the properties of the I-C-M-decomposition, below.

Lemma 4.19 *Given a split graph $H = (V, E)$, the I-C-M-decomposition of H satisfies the properties below.*

(i) Vertices in $I(H)$ are an independent set and are never adjacent to vertices in $M(H)$.

(ii) Vertices in $C(H)$ induce a clique. Every vertex in $C(H)$ is adjacent to every vertex in $M(H)$.

(iii) Vertices in $M(H)$ are an independent set, or vertices in $M(H)$ induce a clique.

(iv) If $M(H)$ is an independent set, for every vertex $v_1 \in I(H)$, there is a vertex $v_2 \in C(H)$ such that v_1 and v_2 are not adjacent.

(v) If vertices in $M(H)$ induce a clique, every vertex in $C(H)$ is adjacent to at least one vertex in $I(H)$.

Proof First, we notice about the compliment of H. Let $\bar{H}$ be the compliment of H. $\bar{H}$ is also a split graph. Since a clique of H is always an independent set of $\bar{H}$ and vice versa, $C(H)$ is equal to $I(\bar{H})$. Similarly, $I(H)$ is equal to $C(\bar{H})$, and thus $M(H)$ is equal to $M(\bar{H})$. Hence, we sometimes omit proofs of redundant statements that we have already proved by thinking of $\bar{H}$.

(i) Assume that there are vertices $v_1, v_2 \in I(H)$ and v_1 is adjacent to v_2. Let (V_I, V_C) be an I-C-decomposition of H. Since $v_1 \in V_I$ and $v_2 \in V_I$, this is a contradiction. Hence, $I(H)$ is an independent set.

Assume that there is a vertex $v_1 \in I(H)$ and there is a vertex $v_2 \in M(H)$ adjacent to v_1. Let (V_I, V_C) be an I-C-decomposition of H where v_2 is in V_I. Since v_1 is in V_I, this is a contradiction. Hence, vertices in $I(H)$ are never adjacent to vertices in $M(H)$.

(ii) Think the compliment of H. We can prove this statement directly from the proof of (i).

(iii) Assume that $M(H)$ is not an independent set, and does not induce a clique. Then, there are vertices $v_1, v_2, v_3 \in M(H)$, v_1 is adjacent to v_2, and v_1 is not adjacent to v_3. There are two possibilities: (a) v_2 is adjacent to v_3 and (b) v_2 is not adjacent to v_3. In the case of (a), let (V_I, V_C) be an I-C-decomposition of H where v_2 is in V_I. Since v_2 is in V_I, v_1 and v_3 should be in V_C. However, v_1 and v_3 are not adjacent. This is a contradiction. In the case of (b), think the compliment of H. The same contradiction occurs. Hence, vertices in $M(H)$ is an independent set, or vertices in $M(H)$ induce a clique.

(iv) Assume that $M(H)$ is an independent set and there is a vertex $v \in I(H)$ that is adjacent to every vertex in $C(H)$. Then, $M(H) \cup I(H) \setminus \{v\}$ is an independent set and vertices $C(H) \cup \{v\}$ induce a clique. This is a contradiction, since $v \in I(H)$. Hence, if $M(H)$ is an independent set, for every vertex $v_1 \in I(H)$, there is a vertex $v_2 \in C(H)$ such that v_1 and v_2 are not adjacent.

(v) Think the compliment of H. We can easily prove the statement from the proof of (iv). □

We call the properties (i), ..., (v) in Lemma 4.19 "I-C-M-conditions".

Lemma 4.20 *Given a split graph $H = (V, E)$, and a partition of V: $I'(H)$, $C'(H)$ and $M'(H)$ which satisfies the I-C-M-conditions, $(I'(H), C'(H), M'(H))$ is always the I-C-H-decomposition of H, i.e., $I(H) = I'(H)$, $C(H) = C'(H)$ and $M(H) = M'(H)$.*

Proof Given a vertex $v_1 \in I'(H)$, assume that there is an I-C-decomposition (V_I, V_C) of H such that v_1 is in V_C. Then since every vertex in $M'(H)$ is not adjacent to v_1, vertices in $M'(H)$ are in V_I. Therefore, $M'(H)$ is an independent set. Thus, there is a vertex $v_2 \in C'(H)$ not adjacent to v_1. Since v_1 is in V_C, v_2 is in V_I. However, v_2 is adjacent to every vertex in $M'(H)$ which is in V_I. This is a contradiction. Hence, every vertex in $I'(H)$ is in $I(H)$.

We can easily check that $I'(H)$ is equal to $C'(\bar{H})$, $C'(H)$ is equal to $I'(\bar{H})$, and $M'(H)$ is equal to $M'(\bar{H})$, where $\bar{H}$ is the compliment of H. Hence, from the proof above, we can prove that every vertex in $C'(H)$ is in $C(H)$, by considering $\bar{H}$.

From Lemma 4.19, we know that every vertex in $I(H)$ is in $I'(H)$, and every vertex in $C(H)$ is in $C'(H)$. Hence, we obtain that $I(H) = I'(H), C(H) = C'(H)$ and $M(H) = M'(H)$. □

Now, we consider the enumeration of children of a split graph H. For every edge $e \in E(G) \setminus E(H)$, we should check whether or not $H + e$ is a split graph. We can determine this by the lemma below.

Lemma 4.21 *Given a split graph H and two vertices $v, w \in V(H)$ such that $(v, w) \notin E(H)$, $H + (v, w)$ is a split graph if and only if*

- *both v and w are in $M(H)$, or*
- *one of v and w is in $I(H)$ and the other is in $C(H) \cup M(H)$.*

Proof Since $(v, w) \notin E(H)$, there are three possibilities for $e = (v, w)$: (a) e is in $M(H) \times M(H)$, (b) e is in $I(H) \times C(H) \cup M(H)$, and (c) e is in $I(H) \times I(H)$.

First, we show that if both v and w are in $M(H)$, $H + (v, w)$ is a split graph. Since (v, w) is not an edge of H, $M(H)$ is an independent set in H. Thus, $I(H) \cup M(H) \setminus \{v, w\}$ is an independent set in $H + (v, w)$, and vertices in $C(H) \cup \{v, w\}$ induce a clique in $H + (v, w)$. $H + (v, w)$ is thus a split graph.

Next, we show that if $v \in I(H)$ and $w \in C(H) \cup M(H)$, $H + (v, w)$ is a split graph. $I(H)$ is an independent set in $H + (v, w)$, and vertices in $C(H) \cup M(H)$ induce a clique in $H + (v, w)$. $H + (v, w)$ is thus a split graph.

Last, we show that if both v and w are in $I(H)$, $H + (v, w)$ is not a split graph. We show that any vertex $p \in C(H)$ is in $C(H + (v, w))$. Assume that there is an I-C-decomposition of $H + (v, w)$, V_I and V_C, such that p is in V_I. Then, all the vertices in $C(H) \setminus \{p\} \cup M(H)$ should be in V_C, since they are adjacent to p in $H + (v, w)$. Thus, vertices in $M(H)$ induce a clique (both in H and in $H + (v, w)$). Therefore, by lemma 4.19, there is a vertex $q \in I(H)$ adjacent to p. Since q is adjacent to p, q should be in V_C. This means q is adjacent to every vertex in $C(H)$, and thus $q \in M(H)$. This is a contradiction. Hence, p is in $C(H + (v, w))$.

Since v and w are adjacent to each other in $H + (v, w)$, it is impossible that both $v \in I(H + (v, w))$ and $w \in I(H + (v, w))$ consist. We can assume without loss of generality that v is in $C(H + (v, w)) \cup M(H + (v, w))$. Then, v should be adjacent to all the vertices in $C(H + (v, w))$ by Lemma 4.19. Therefore, v is adjacent to every vertex in $C(H)$, since we have proven that $C(H) \subseteq C(H + (v, w))$.

If $|M(H)| \leq 1$, then $(V_I = I(H) \cup M(H) \setminus \{v\}, V_C = C(H) \cup \{v\})$ V_I is an I-C-decomposition of H. This is a contradiction, since $v \in I(H)$. Thus, we assume that $|M(H)| > 1$, below. From the observations above, we know that if $|M(H)| > 1$, the vertices in $M(H)$ induce a clique.

Now, we consider an I-C-decomposition (V_I, V_C) of $H + (v, w)$. If w is in V_I, v is in V_C. Since v is not adjacent to vertices in $M(H)$, vertices in $M(H)$ are in V_I. However, vertices in $M(H)$ induce a clique. This is a contradiction. Thus, w is in V_C. Since vertices in $M(H)$ induce a clique, at least one vertex in $M(H)$ should be in V_C. However, vertices in $M(H)$ are not adjacent to w. This is a contradiction. □

By this lemma, keeping vertex sets $I(H), C(H)$ and $M(H)$ enables us to check whether or not each $H + e$ is a split graph in constant time.

Split graphs are self-complementary. Thus, we easily obtain the lemma below, by considering complements of the lemma above.

Lemma 4.22 *Given a split graph H and two vertices $v, w \in V(H)$ such that $(v, w) \in E(H)$, $H - (v, w)$ is a split graph if and only if*

- *both v and w are in $M(H)$, or*
- *one of v and w is in $C(H)$ and the other is in $I(H) \cup M(H)$.*

Proof By Lemma 4.21 and the fact that split graphs are self-complementary. Note that $C(\bar{H}) = I(H)$, $I(\bar{H}) = C(H)$, and $M(\bar{H}) = M(H)$, where $\bar{H}$ is the complement of H. □

To generate children of split graph H quickly, we need to maintain C-I-M-decomposition of H in the algorithm, and the time cost to update the C-I-M-decomposition should be short. The lemmata below enable us to do this.

Lemma 4.23 *Given a split graph H and vertices $v, w \in M(H)$ such that $(v, w) \notin E(H)$,*

- *if $|M(H)| \neq 2$, $I(H + (v, w)) = I(H) \cup M(H) \setminus \{v, w\}$, $C(H + (v, w)) = C(H)$, and $M(H + (v, w)) = \{v, w\}$,*
- *if $|M(H)| = 2$, $I(H + (v, w)) = I(H)$, $C(H + (v, w)) = \{x \in C(H) \mid \exists y \in I(H), s.\ t.\ (x, y) \in E(H)\}$, and $M(H + (v, w))$ is the rest vertices in $V(H)$.*

Lemma 4.24 *Given a split graph H and vertices $v \in I(H)$ and $w \in C(H)$ such that $(v, w) \notin E(H)$,*

- *if $deg_H(v) = |C(H)| - 1$, and $|M(H)| = 0$, $I(H + (v, w)) = I(H) \setminus \{v\}$, $C(H + (v, w)) = \{x \in C(H) \mid \exists y \in I(H + (v, w)), s.\ t.\ (x, y) \in E(H)\}$, and $M(H + (v, w))$ is the rest vertices in $V(H)$,*

- *if* $deg_H(v) = |C(H)| - 1$*, and* $M(H) \neq \emptyset$ *is an independent set,* $(v, w) \notin E(H)$*,* $I(H + (v, w)) = I(H) \setminus \{v\}$*,* $C(H + (v, w)) = C(H)$*, and* $M(H + (v, w)) = M(H) \cup \{v\}$*,*
- *otherwise,* $I(H + (v, w)) = I(H)$*,* $C(H + (v, w)) = C(H)$*, and* $M(H + (v, w)) = M(H)$*.*

Lemma 4.25 *Given a split graph* H *and vertices* $v \in I(H)$ *and* $w \in M(H)$ *such that* $e = (v, w) \notin E(H)$*,*

- *if* $deg_H(v) = |C(H)|$ *and* $|M(H)| = 2$*,* $I(H + (v, w)) = I(H) \setminus \{v\}$*,* $C(H + (v, w)) = C(H) \cup \{w\}$*, and* $M(H + (v, w)) = M(H) \cup \{v\} \setminus \{w\}$*,*
- *otherwise, if vertices in* $M(H)$ *induce a clique,* $I(H + (v, w)) = I(H)$*,* $C(H + (v, w)) = C(H) \cup \{w\}$*, and* $M(H + (v, w)) = M(H) \setminus \{w\}$*.*
- *otherwise,* $I(H + (v, w)) = I(H) \cup M(H) \setminus \{w\}$*,* $C(H + (v, w)) = C(H) \cup \{w\}$*, and* $M(H + (v, w)) = \emptyset$*.*

Proof of Lemma 4.23 First, consider the case that $|M(H)| = 2$.

Let v' be a vertex in $I(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$, such that $v' \in V_C$. Since $v, w \in M(H)$, there is no edge between v' and v and between v' and w. Thus, v and w are in V_I. However, there is an edge connecting v and w in $H + (v, w)$. This is a contradiction. Therefore, every vertex in $I(H)$ is in $I(H + (v, w))$.

Let v' be a vertex in $C(H)$ and let v'' be a vertex in $I(H)$ adjacent to v. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. We proved above that a vertex in $I(H)$ is in $I(H + (v, w))$. Thus, v'' is in V_I. However, there is an edge connecting v' and v''. This is a contradiction. Therefore, v' is in $C(H + (v, w))$.

Let v' be a vertex in $C(H)$ and be adjacent to no vertex in $I(H)$. $I(H)$ is an independent set and vertices in $C(H) \cup M(H)$ induce a clique in $H + (v, w)$. On the other hand, $I(H) \cup v'$ is an independent set and vertices in $C(H) \cup M(H) \setminus v'$ induce a clique in $H + (v, w)$. Therefore v' is in $M(H + (v, w))$.

Let v' be a vertex in $M(H)$, which means that v' is v or w. $I(H)$ is an independent set and vertices in $C(H) \cup M(H)$ induce a clique in $H + (v, w)$. On the other hand, $I(H) \cup v'$ is an independent set and vertices in $C(H) \cup M(H) \setminus v'$ induce a clique in $H + (v, w)$. Therefore v' is in $M(H + (v, w))$.

Next, consider the case that $|M(H)| \neq 2$.

Let v' be a vertex in $I(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. Since $v, w \in M(H)$, there is no edge between v' and v and between v' and w. Thus, v and w are in V_I. However, there is an edge connecting v and w in $H + (v, w)$. This is a contradiction. Therefore, every vertex in $I(H)$ is in $I(H + (v, w))$.

Let v' be a vertex in $M(H) \setminus \{v, w\}$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. Since there is no edge connecting v and w in H, $M(H)$ is an independent set of H. Hence, there is no edge between v' and v and between v' and w. Thus, v and w are in V_I. However, there is an

edge connecting v and w in $H + (v, w)$. This is a contradiction. Therefore, v' is in $I(H + (v, w))$.

Let v' be a vertex in $C(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. Since $|M(H)| \neq 2$ and $v, w \in M(H)$, there is a vertex $v'' \in M(H)$. Since there is no edge connecting v and w in H, $M(H)$ is an independent set of H. Thus, there is no edge connecting v and v'' in $H + (v, w)$. Since every vertex in $M(H)$ is adjacent to $v' \in C(H)$, v, w and v'' are in V_C. This is a contradiction. Therefore v' is in $C(H + (v, w))$.

$I(H) \cup M(H)$ is an independent set and vertices in $C(H)$ induce a clique in $H + (v, w)$. $I(H) \cup M(H) \setminus \{v\}$ is an independent set and vertices in $C(H) \cup \{v\}$ induce a clique in $H + (v, w)$. $I(H) \cup M(H) \setminus \{w\}$ is an independent set and vertices in $C(H) \cup \{w\}$ induce a clique in $H + (v, w)$. Therefore, v and w are in $M(H + (v, w))$. □

Proof of Lemma 4.24 First, we consider the case that $\deg_H(v) = |C(H)| - 1$ and $M(H) = \emptyset$. Let v' be a vertex in $I(H) \setminus \{v\}$. Assume that there is an I-C decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. v is not adjacent to v'. Since $M(H)$ is the empty set and is thus an independent set, there is a vertex $v'' \in C(H)$ that is not adjacent to v'. Hence v and v'' are both in V_I. This is a contradiction, since v and v'' are adjacent in $H + (v, w)$. Therefore, v' is in $I(H + (v, w))$.

Let v' be a vertex in $\{x \in C(H) \mid \exists y \in I(H + (v, w)), s.\ t.\ (x, y) \in E(H)\}$. Assume that there is an I-C decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. Let $v'' \in I(H + (v, w)) = I(H) \setminus \{v\}$ be a vertex adjacent to v'. In $H + (v, w)$, both v and v'' are adjacent to v', and are thus in V_C. However, v and v'' are not adjacent in $H + (v, w)$. This is a contradiction. Therefore, v' is in $C(H)$.

$(I(H), C(H))$, $(I(H) \setminus \{v\}, C(H) \cup \{v\})$ are I-C-decompositions of $H + (v, w)$. For any $v' \in \{x \in C(H) \mid \nexists y \in I(H + (v, w)), (I(H) \setminus \{v'\}, C(H) \cup \{v'\})$ is also an I-C-decomposition of $H + (v, w)$. Therefore, v and v' are in $M(H + (v, w))$.

Next, we consider the case that $\deg_H(v) = |C(H)| - 1$ and $M(H)$ is an independent set and not empty. Let v' be a vertex in $I(H) \setminus \{v\}$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. v is not adjacent to v'. Since $M(H)$ is an independent set, there is a vertex $v'' \in C(H)$ that is not adjacent to v'. Hence v and v'' are both in V_I. This is a contradiction, since v and v'' is adjacent in $H + (v, w)$. Therefore, v' is in $I(H + (v, w))$.

Let v' be a vertex in $C(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. v is adjacent to v' in $H + (v, w)$. Since $M(H)$ is not empty, there is a vertex $v'' \in M(H)$, and v'' is adjacent to v'. Thus, v and v'' are in V_C. However, there is no edge between v and v''. This is a contradiction. Therefore, v' is in $C(H)$.

$(I(H) \cup M(H), C(H))$, $(I(H) \cup M(H) \setminus \{v\}, C(H) \cup \{v\})$ are I-C-decompositions of $H + (v, w)$. For any $v' \in M(H)$, $(I(H) \cup M(H) \setminus \{v'\}, C(H) \cup \{v'\})$ is also an I-C-decomposition of $H + (v, w)$. Therefore, v and v' are in $M(H + (v, w))$.

Next, we consider the case that $\deg_H(v) = |C(H)| - 1$, $M(H)$ is not an independent set and $M(H)$ is not empty. Note that it is equivalent that $\deg_H(v) = |C(H)| - 1$, $|M(H)| \geq 2$ and vertices in $M(H)$ induce a clique.

Let v' be a vertex in $I(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. v' is not adjacent to two vertices v'' and v''' in $M(H)$. Thus, v'' and v''' are in V_I. However, there is an edge between v'' and v'''. This is a contradiction. Therefore, v' is in $I(H + (v, w))$.

Let v' be a vertex in $C(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. v is adjacent to v' in $H + (v, w)$. Since $M(H)$ is not empty, there is a vertex $v'' \in M(H)$, and v'' is adjacent to v'. Thus, v and v'' are in V_C. However, there is no edge between v and v''. This is a contradiction. Therefore, v' is in $C(H)$.

$(I(H), C(H) \cup M(H))$ is an I-C-decomposition of $H + (v, w)$. For any $v' \in M(H)$, $(I(H) \cup \{v'\}, C(H) \cup M(H) \setminus \{v'\})$ is also an I-C-decomposition of $H + (v, w)$. Therefore, v' is in $M(H + (v, w))$.

Last, we consider the case that $\deg_H(v) < |C(H)| - 1$.

Let v' be a vertex in $I(H) \setminus \{v\}$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. Since, v is not adjacent to v', v is in V_I. Since, w is adjacent to v, w is in V_C. Thus, (v, w) is an edge in $V_I \times V_C$. Hence, V_I is an independent set and vertices in V_C induce a clique. That is (V_I, V_C) is an I-C-decomposition of H, and $v' \in I(H)$ is in V_C. This is a contradiction. Therefore, v' is in $I(H)$.

Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v \in V_C$. Since $\deg_{H+(v,w)}(v) < |C(H)|$, there is a vertex $v' \in C(H)$ that is not adjacent to v in $H + (v, w)$. Since a vertex in $C(H)$ is always adjacent to a vertex in $I(H) \cup M(H)$, v' is adjacent to $v'' \in I(H) \cup M(H)$. Since v' and v'' are not adjacent to v, v' and v'' are in V_I. However, there is an edge connecting v' and v''. This is a contradiction. Therefore, v is in $I(H + (v, w))$.

Let v' be a vertex in $C(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. We showed above that vertices in $I(H)$ belong to $I(H + (v, w))$ and are thus in V_I. Hence v' is not adjacent to any vertex in $I(H)$. Thus, v' is adjacent to at least one vertex in $M(H)$ (otherwise, v' should be in $M(H)$) and vertices in $M(H)$ do not induce a clique (by Lemma 4.19). This means that there are at least two vertices v'' and v''' in $M(H)$, and v'' and v''' are not adjacent to each other. v' is adjacent to both v'' and v''', since v' is in $C(H)$ and v'', v''' are in $M(H)$. Thus, v'' and v''' are in V_C. However, there is no edge connecting v'' and v'''. This is a contradiction. Therefore, v' is in $C(H + (v, w))$.

If $M(H)$ is an independent set, $(I(H) \cup M(H), C(H))$ is an I-C-decomposition of $H + (v, w)$. For any $v' \in M(H)$, $(I(H) \cup M(H) \setminus \{v'\}, C(H) \cup \{v'\})$ is also an I-C-decomposition of $H + (v, w)$. Therefore, v' is in $M(H + (v, w))$. If vertices in $M(H)$ induce a clique, $(I(H), C(H) \cup M(H))$ is an I-C-decomposition of $H + (v, w)$. For any $v' \in M(H)$, $(I(H) \cup \{v'\}, C(H) \cup M(H) \setminus \{v'\})$ is also an I-C-decomposition of $H + (v, w)$. Therefore, v' is in $M(H + (v, w))$. □

Proof of Lemma 4.24 First, we consider the case that $\deg_H(v) = |C(H)|$ and

$|M(H)| = 2$. In this case, there is an edge connecting the two vertices in $M(H)$, since otherwise, $(I(H) \cup M(H), \setminus\{v\}, C(H) \cup \{v\})$ is an I-C-decomposition of H, and v should be in $M(H)$.

Let v' be a vertex in $I(H) \setminus \{v\}$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. Since vertices in $M(H)$ are not adjacent to v', they are in V_I. However, there is an edge connecting them. This is a contradiction. Therefore, v' is in $I(H + (v, w))$.

Let v' be a vertex in $C(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. Since v is adjacent to v', v is in V_C. Since the two vertices in $M(H)$ are adjacent to v', they are also in V_C. However, one of the two vertices in $M(H)$ is not adjacent to v in $H + (v, w)$. This is a contradiction. Therefore, v' is in $C(H + (v, w))$.

Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $w \in V_I$. Since v is adjacent to w in $H + (v, w)$, v is in V_C. Let v' be the vertex in $M(H)$ which is not w. Since v' is adjacent to w, v' is also in V_C. However, there is no edge connecting v and v'. This is a contradiction. Therefore, w is in $C(H + (v, w))$.

Let v' be the vertex in $M(H)$ which is not w. $(I(H), C(H) \cup M(H))$ is an I-C-decomposition of $H + (v, w)$. $(I(H) \setminus \{v\}, C(H) \cup M(H) \cup \{v\})$ is an I-C-decomposition of $H + (v, w)$. $(I(H) \cup \{v'\}, C(H) \cup M(H) \setminus \{v'\})$ is an I-C-decomposition of $H + (v, w)$. Therefore, v and v' are in $M(H + (v, w))$.

Next, we consider the case that $(\deg_H(v) < |C(H)|$ or $|M(H)| \neq 2)$ and vertices in $M(H)$ induce a clique.

Let v' be a vertex in $I(H) \setminus \{v\}$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. Since v' is not adjacent to v, v is in V_I. Since w is adjacent to v in $H + (v, w)$, w is in V_C. Thus, (v, w) is an edge in $V_I \times V_C$. Hence, (V_I, V_C) is an I-C-decomposition of H, and $v' \in I(H)$ is in V_C. This is a contradiction. Therefore, v' is in $I(H + (v, w))$.

When $\deg_H(v) < |C(H)|$, assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v \in V_C$. Since $\deg_H(v) < |C(H)|$, there is a vertex $v' \in C(H)$ that is not adjacent to v. v' is in V_I. Since vertices in $M(H)$ induce a clique, v' is adjacent to a vertex $v'' \in I(H)$. v'' is in V_C. However, there is no edge connecting v and v''. This is a contradiction. Therefore, v is in $I(H)$.

When $|M(H)| \neq 2$, $|M(H)|$ should be more than two, since if $|M(H)| = 1$, v is isomorphic to the vertex in $M(H)$ and should be in $M(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v \in V_C$. There are two vertices in $M(H)$ that are not w. Since they are not adjacent to v, they are in V_I. However, there is an edge connecting them. This is a contradiction. Therefore, v is in $I(H)$.

Let v' be a vertex in $C(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. Since vertices in $M(H)$ induce a clique, v' is adjacent to a vertex v'' in $I(H)$. If $v'' \neq v$, v'' and w are in V_C, since they are adjacent to v'. This is a contradiction. If $v'' = v$ there is a vertex $v''' \in C(H) \cup M(H)$ that is not adjacent to v, since $\deg_H(v) < |C(H)|$ or $|M(H)| > 2$ consist. However, v and v''' are in V_C, since they both are adjacent to v'. This is a contradiction. Therefore, v' is in $C(H)$.

Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that w is in V_I. There is a vertex $v' \in C(H) \cup M(H) \setminus \{w\}$ that is adjacent to w and not adjacent to v, since $\deg_H(v) < |C(H)|$ or $|M(H)| > 2$ consist. v' and v are in V_C, since they are adjacent to w. However, they are not adjacent in $H + (v, w)$. This is a contradiction. Therefore, w is in $C(H + (v, w))$.

Let v' be a vertex in $M(H)$ which is not w. $(I(H), C(H) \cup M(H))$ is an I-C-decomposition of $H + (v, w)$. $(I(H) \cup \{v'\}, C(H) \cup M(H) \setminus \{v'\})$ is an I-C-decomposition of $H + (v, w)$. Therefore, v and v' are in $M(H + (v, w))$.

Last, consider the case that vertices in $M(H)$ do not induce a clique in H. In this case, there is at least one vertex w' in $M(H)$ that is not equal to w.

Let v' be a vertex in $I(H) \setminus \{v\}$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. Since v and w are not adjacent to v', they are in V_I. However, there is an edge connecting them in $H + (v, w)$. This is a contradiction. Therefore, v' is in $I(H + (v, w))$.

Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v \in V_C$. Since w' is not adjacent to v, w' is in V_I. There is a vertex v' in $C(H)$ not adjacent to v, since otherwise v is isomorphic to w and should be in $M(H)$. v' is in V_I. However, there is an edge connecting v' and w'. This is a contradiction. Therefore, v is in $I(H + (v, w))$.

Let v' be a vertex in $M(H) \setminus \{w\}$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_C$. Since $M(H)$ is an independent set, there is a vertex $v'' \in C(H)$ not adjacent to v. Since v'' and w are adjacent to v', they are in V_I. However, there is an edge connecting them in $H + (v, w)$. This is a contradiction. Therefore, v' is in $I(H + (v, w))$.

Let v' be a vertex in $C(H)$. Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $v' \in V_I$. Since w and w' are adjacent to v', they are in V_C. However, there is no edge connecting w and w'. This is a contradiction. Therefore, v' is in $C(H + (v, w))$.

Assume that there is an I-C-decomposition (V_I, V_C) of $H + (v, w)$ such that $w \in V_I$. There is a vertex $v' \in C(H)$ not adjacent to v, since otherwise v is isomorphic to w in H, and should be in $M(H)$. Since v and v' are adjacent to w, they are in V_C. However, there is no edge connecting them. This is a contradiction. Therefore, w is in $C(H + (v, w))$. □

Now, we consider that, given a split graph H, for what edge (v, w), $H+(v, w)$ is a child of H.

Lemma 4.26 *Given a non-empty split graph H whose parent is $H - (v^*, w^*)$, $H + (v, w) - (v^*, w^*)$ is not a split graph if and only if*

- *one of v and w is in $I(H)$, and the other is in $M(H)$, and*
- *one of v^* and w^* is in $C(H)$, and the other is equal to v or w.*

Lemma 4.27 *Given a non-empty split graph H containing an edge (v', w') such that $H+(v, w)$ is a split graph and $H-(v', w')$ is not a split graph, $H+(v, w)-(v', w')$ is a split graph if and only if*

- *both v' and w' are in $C(H)$, and*

- – *$M(H) = \{v, w\}$ and at least one of v' and w' is adjacent to no vertex in $I(H)$, or*
  – *one of v and w are in $I(H)$ and the other is in $C(H)$ (we assume that $v \in I(H)$ and $w \in C(H)$, in this case), $\deg_H(v) = |C(H)| - 1$ one of v' and w' is not adjacent to any vertex in $I(H)$ but v, and $M(H)$ is empty.*

Proof of Lemma 4.26 Since $H+(v,w)-(v^*,w^*)$ is not a split graph, (v^*,w^*) is an edge in $C(H+(v,w)) \times C(H+(v,w))$ (otherwise, $H+(v,w)-(v^*,w^*)$ is a split graph by Lemma 4.22). Since $H-(v^*,w^*)$ is a split graph, (v^*,w^*) is an edge not in $C(H) \times C(H)$. Conversely, if (v^*,w^*) is in $C(H+(v,w)) \times C(H+(v,w))$ and not in $C(H) \times C(H)$, $H+(v,w)-(v^*,w^*)$ is not a split graph and $H-(v^*,w^*)$ is a split graph, by Lemma 4.22.

By Lemmata 4.23,...,4.25, $C(H+(v,w))$ contains at most one vertex that is not in $C(H)$. And if $C(H+(v,w))$ contains such a vertex v', v' is equal to v or w. Moreover, if $C(H+(v,w))$ contains v', $\{v,w\}$ is in $I(H) \times M(H)$. Contrary, if $\{v,w\}$ is in $I(H) \times M(H)$, $C(H+(v,w))$ always contains a vertex not in $C(H)$. □

Proof of Lemma 4.27 Since $H-(v',w')$ is not a split graph, both v' and w' are in $C(H)$, by Lemma 4.22.

If $M(H) = \{v,w\}$, and one of v' and w' is not adjacent to $I(H)$, then either v' or w' is in $M(H+(v,w))$, by Lemma 4.23. Thus, $H+(v,w)-(v',w')$ is a split graph, by Lemma 4.22.

If $\{v,w\} \in I(H) \times C(H)$, $\deg_H(v) = |C(H)| - 1$, v' is not adjacent to any vertex in $I(H)$ but v, and $M(H)$ is empty, then v' is in $M(H+(v,w))$, by Lemma 4.24. Thus, $H+(v,w)-(v',w')$ is a split graph.

Otherwise, both v' and w' are in $C(H+(v,w))$, by Lemma 4.23, ..., and Lemma 4.25. Thus, $H+(v,w)-(v',w')$ is not a split graph, by Lemma 4.22. □

From Lemmata 4.26 and 4.27, we can obtain the necessary and sufficient condition for a graph to be a child of the given split graph. For simplicity, given two edges $e_1 = (v_1, v_2)$ and $e_2 = (w_1, w_2)$, we write $e_1 < e_2$ if the younger vertex of v_1 and v_2 is younger than the younger vertex of w_1 and w_2, or the younger vertices of v_1, v_2 and w_1, w_2 are identical and the elder vertex of v_1, v_2 is younger than the elder vertex of w_1, w_2.

Lemma 4.28 *Given a non-empty split graph H and vertices $v, w \in V(H)$ such that $(v,w) \notin E(H)$, let $H-(v^*,w^*)$ be the parent of H. Then, $H+(v,w)$ is a child of H if and only if*

- *when v is in $I(H)$ and w is in $C(H)$,*
  – *if $M(H) = \emptyset$, $\deg_H(v) = |C(H)|-1$, one of v' and w' is not adjacent to any vertex in $I(H)$ but v, $(v,w) < (v^*,w^*)$ and $(v,w) < (v',w')$, where (v',w') is an edge in $C(H) \times C(H)$ whose one vertex is not adjacent to any vertex in $I(H)$,*
  – *if otherwise, $(v,w) < (v^*,w^*)$.*

- *when v is in $I(H)$ and w is in $M(H)$,*
  - *$(v, w) < (v^*, w^*)$, or*
  - *if $v^* \notin M(H)$ or $w^* \notin M(H)$, $H + (v, w)$ is a child of H.*
- *when both v and w are in $M(H)$,*
  - *if $M(H) = \{v, w\}$, $(v, w) < (v^*, w^*)$ and $(v, w) < (v', w')$, for any edge $(v', w') \in C(H) \times C(H)$ whose one vertex is not adjacent to any vertex in $I(H)$,*
  - *otherwise, $(v, w) < (v^*, w^*)$.*

In Lemma 4.28, we do not make the necessary and sufficient condition clear in the case that v is in $I(H)$, w is in $M(H)$, and $v^* \notin M(H)$ or $w^* \notin M(H)$. Anyway, we show that we can obtain every child of H in $O(n)$ time, if we know $I(H), C_1(H) = \{v \in C(H) \mid \exists w \in I(H), s.t. (v, w) \in E(H)\}, C_2(H) = C \setminus C_1$ and $M(H)$.

Lemma 4.29 *Given a graph $G = (V, E)$ and a split graph H that is a subgraph of G, we can obtain every child of H in $O(|V|)$ time, if we know $I(H), C_1(H) = \{v \in C(H) \mid \exists w \in I(H), s.t. (v, w) \in E(H)\}, C_2(H) = C \setminus C_1$ and $M(H)$.*

Proof In the case that v is in $I(H)$, w is in $M(H)$, and $v^* \notin M(H)$ or $w^* \notin M(H)$, in that case we do not show the necessary and sufficient condition in Lemma 4.28. In other cases, it is clear that we can find each edge (v, w) that satisfies the condition in Lemma 4.28 in $\mathrm{O}(|V|)$ time.

In this case, one of v^* and w^* is in $C(H)$ and the other is not in $C(H)$. Let $\bar{v}$ be the vertex not in $C(H)$ among them. Then by Lemma 4.26, $H + (v, w) - (v^*, w^*)$ is not a split graph, if v or w is equal to $\bar{v}$. Hence, we have to check, for each $H' = H + (v^*, w)$ and for each $H' = H + (v, v^*)$, if H' is a child of H or not. To do this, $\mathrm{O}(V(H))$ time is needed. □

By Lemmata 4.24,4.25 and 4.23, it is clear that we can maintain $I(H), C_1(H), C_2(H)$ and $M(H)$.

Lemma 4.30 *Given a graph $G = (V, E)$, split graphs H, $H + (v, w)$ such that H and H' are subgraphs of G and H is a parent of $H + (v, w)$, and $I(H), C_1(H), C_2(H)$ and $M(H)$, we can obtain $I(H'), C_1(H'), C_2(H')$ and $M(H')$ in $O(|V|)$ time.*

By Lemmata 4.29 and 4.30, the following theorem can be obtained.

Theorem 4.31 *There is an algorithm for the split subgraph enumeration. The time complexity of the algorithm to find each split subgraph of a given graph is $O(n)$ and the space complexity is $O(n + m)$.*

By considering the compliment, the following theorem is also obtained.

Theorem 4.32 *There is an algorithm for the split supergraph enumeration. The time complexity to find each split supergraph of a given graph is $O(n)$ and the space complexity is $O(n + m)$.*

4.2 Parent-Child Relation by Simplicial Vertex Elimination

In this section, we develop an enumeration scheme by defining a parent through a reverse search by a removal of a simplicial vertex. As contrasted with the case of the previous section, given a graph G of n vertices, there are generally exponentially many candidates of the children of G. However, we can enumerate some graphs more quickly by this scheme than by that of the previous section. In fact, chordal subgraph enumeration can be done in O(1) time for each chordal graph, which we take $O(n^4)$ time by the algorithm in the previous section. Moreover, we can enumerate some graphs that are subclasses of chordal graphs even if we do not know about the relation by edge removals/additions.

4.2.1 Chordal Subgraph Enumeration

Definition of Parents

Let $G = (V, E)$ be an arbitrary graph, and $V = \{1, \ldots, n\}$. Suppose H is a chordal subgraph of G and H has more than one edges. We define *minimum degree simplicial vertex* of H as the simplicial vertex having the minimum degree, and denote it by $s^*(H)$. If there are more than one such simplicial vertices, we choose the youngest (in vertices number) as $s^*(H)$, so that $s^*(H)$ is defined uniquely. Note that any chordal graph has at least one simplicial vertex, hence we can define $s^*(H)$ for any chordal graph. We define the parent of H as the graph obtained by eliminating $s^*(H)$ from H. Since an elimination of a simplicial vertex from a chordal graph results in another chordal graph, the parent of H is also a chordal graph. If H is connected, then the parent of H is also connected, since for any two neighbors of a simplicial vertex there is an edge that connects the vertices (see Figure 4.2). The number of edges on the parent chordal graph is always strictly less than that of its child. Thus, no chordal graph becomes an ancestor of itself. Therefore, both the parent–child relation defined on all chordal subgraphs of G and the parent–child relation defined on all connected chordal subgraphs of G satisfy the conditions to be used in the reverse search described in section 3.2.1. We illustrate an example of the parent–child relation of chordal graphs in Figure 4.2. Root chordal graphs are the graphs with exactly one edge.

Enumeration of Children

To enumerate all chordal graphs included in a given graph, we need an algorithm to enumerate all root chordal graphs; we also need an algorithm to enumerate all children of a given chordal graph. The former algorithm is very straightforward: simply generate all subgraphs having exactly one edge. In the following, we describe the algorithm to generate children of chordal graphs included in an arbitrary graph. We consider both the case that the chordal graphs to be enumerated are connected and the case that they do not need to be connected.

Figure 4.2: The left chordal graph has simplicial vertices 1, 3, 4 and 7. The simplicial vertices of the minimum degree are 4 and 7. $s^*(H)$ is 4. The right chordal graph obtained by eliminating 4 from the left graph is the parent of the left graph.

The parent of a chordal graph is obtained by eliminating a simplicial vertex. Hence, given a connected chordal graph H in G, any of its connected children is obtained by adding a vertex v to H. Adding a vertex v to H means adding the vertex v to the vertex set of H and adding edges connecting the vertex v and some other vertices $C \subseteq V(H)$ to the edge set $E(H)$. To obtain a child, it is necessary that C contains at least one vertex, and is a clique in H so that v is a simplicial vertex of the child. In the following, we characterize a necessary and sufficient condition for the resulting graph to be a child of H.

We first introduce some notations. We denote the set of simplicial vertices in H by $S(H)$. The minimum degree in $S(H)$ is denoted by $k(H)$. We define $S_d(H)$ as the set of simplicial vertices of degree d in H, and particularly, we denote $S_{k(H)}(H)$ by $S^*(H)$. We denote the youngest vertex in a vertex set X by $\min(X)$. If $X = \emptyset$, we define $\min(X)$ is $+\infty$. Suppose H be a connected chordal graph included in G. Let v be a vertex of G and not of H. We denote by $N(H, v)$ the subgraph of H induced by the vertices which are adjacent to v in G. We note that $N(H, v) = H$ holds for any H and v if G is a complete graph. Let C be a vertex subset in $N(H, v)$. We denote by $G_H(v, C)$ the graph obtained by adding v and edges connecting v and all vertices in C to H. It is necessary that any connected child H' of a chordal graph H satisfies that $H' = G_H(v, C)$ for some v and C, and C is a clique of H. We show below the necessary and sufficient condition for $H' = G_H(v, C)$ to be a child of a chordal graph H.

Lemma 4.33 *For a vertex $v \notin V(H)$ and a clique C in $N(H, v)$, $G_H(v, C)$ is a child of H if and only if one of the following conditions holds.*

(1) $|C| < k(H)$

(2) $|C| = k(H)$ *and* $v < \min(S^*(H) \setminus C)$

(3) $|C| = k(H) + 1$, $S^*(H) \subseteq C$ *and* $v < \min(S^*(H) \cup (S_{k(H)+1}(H) \setminus C))$.

In order to prove Lemma 4.33, we first claim the following propositions.

Proposition 4.34 *Any simplicial vertex u ($\neq v$) in $G_H(v, C)$ is simplicial in H.*

Proof If u is not adjacent to v, the neighbors of u form a clique in H. If u is adjacent to v, the elimination of v from the neighbors of u also forms a clique in H. Thus, in both cases, u is simplicial in H. □

Proposition 4.35 *$G_H(v, C)$ is a connected chordal graph, and v is simplicial in $G_H(v, C)$.*

Proof Let X be a cycle in $G_H(v, C)$ having at least four edges. If X does not include v, then X is included in H, hence X has a chord. If X includes v, then X includes at least two neighbors of v. The edge connecting the two neighbors is a chord of X, thus any cycle in $G_H(v, C)$ of at least four edges has a chord. Since the neighbors of v form C which is a clique, v is simplicial in $G_H(v, C)$. □

Proof of Lemma 4.33 The above two propositions show that $G_H(v, C)$ is a connected child of H if and only if $s^*(G_H(v, C)) = v$. Thus, in order to prove the statement, we only need to check whether or not $s^*(G_H(v, C)) = v$ holds in the case of the conditions (1), (2) and (3). We consider the following four cases according to the size of C.

(a) $|C| < k(H)$: The degree of v in $G_H(v, C)$ is $|C|$ and is smaller than degrees of any other simplicial vertex in H. From Proposition 4.34, any simplicial vertex in $G_H(v, C)$ is a simplicial vertex in H, hence v is the unique minimum degree vertex among simplicial vertices in $G_H(v, C)$. Thus, $s^*(G_H(v, C)) = v$.

(b) $|C| = k(H)$: Similarly to the above, v has the minimum degree ($k(H)$) among simplicial vertices in $G_H(v, C)$. However it is possible that there are some other simplicial vertices whose degree are also $k(H)$. Since the degree of vertices of C in $G_H(v, C)$ is larger than $k(H)$, $S^*(G_H(v, C))$ is equal to $(S^*(H) \setminus C) \cup \{v\}$. Thus, $s^*(G_H(v, C)) = v$ if and only if v is younger than $\min(S^*(H) \setminus C)$.

(c) $|C| = k(G) + 1$: In this case, v has the minimum degree in $S(G_H(v, C))$ if and only if $S^*(H) \subseteq C$ holds. Thus, $s^*(G_H(v, C)) = v$ if and only if $S^*(H) \subseteq C$ and $v < \min(S^*(H) \cup (S_{k(H)+1}(H) \setminus C))$ hold.

(d) $|C| > k(H) + 1$: In this case, $C \cap S^*(H) = \emptyset$ since any vertex in $S^*(H)$ is adjacent to exactly $k(H)$ vertices. Thus, it is clear that $s^*(G_H(v, C))(= s^*(H))$ is never equal to v.

From these observations, we obtain that for a vertex $v \notin H$ and a clique C in H,

- if one of (1), (2) or (3) holds, $G_H(v, C)$ is a connected child of H,

and

- if none of (1), (2) and (3) holds, i.e.,
  one of the below holds,
  - (2') $|C| = k(H)$ and $v > \min(S^*(H) \setminus C)$,
  - (3') $|C| = k(H) + 1$ and $S^*(H) \setminus C \neq \emptyset$,
  - (3") $|C| = k(H) + 1$ and $v > \min(S^*(H) \cup (S_{k(H)+1}(H) \setminus C))$,
  - (4') $|C| > k(H) + 1$,

  then $G_H(v, C)$ is not a child of H.

□

Lemma 4.33 characterizes the children of a connected chordal graph efficiently. From the lemma, we obtain an algorithm to enumerate the children of a connected chordal graph. It directly leads an algorithm to enumerate connected chordal graphs in an arbitrary graph $\bar{G}$. In Figure 4.3, we describe the algorithm. Note that it is easy to generate all cliques whose size are restricted, hence, if $\bar{G}$ is a complete graph, the algorithm becomes more simple.

procedure enum_sub_connected_chordal(H, $G = (V, E)$)
H: **chordal graph**, G: **graph**;
begin
1: **output** H;
2: **if** $|V(H)| = n$ **then return;**
3: **if** $k(H) = 1$ **then do**
  for each pair of vertices $v \notin H$ and $u \in H$
    such that $(u, v) \in E$ and $v < \min(S^*(H) \setminus \{u\})$,
    call enum_sub_connected_chordal($G_H(v, \{u\})$, G);
  return;
 end;
4: **for each** d such that $S_d(H) \neq \emptyset$,
  compute $S_d(H)$ and $\min(S_d(H))$;
5: **for each** vertex $v \notin V(H)$ with non-empty $N(H, v)$,
  for each clique C in $N(H, v)$ of size at most $k(H) - 1$ in H,
    call enum_sub_connected_chordal ($G_H(v, C)$, G)
      for each $v \notin G$;
6: **for each** pair of vertex $v \notin V(H)$ and clique C in $N(H, v)$ of size $k(H)$
    such that $v < \min(S^*(H) \setminus C)$,
  call enum_sub_connected_chordal ($G_H(v, C)$, G);
7: **for each** pair of vertex $v \notin V(H)$ and clique C in $N(H, v)$ of size $k(H) + 1$
    such that $S^*(H) \subseteq C$ and $v < \min(S^*(H) \cup S_{k(H)+1}(H))$,
  call enum_sub_connected_chordal ($G_H(v, C)$, G);
end.

Figure 4.3: Algorithm for connected chordal subgraph enumeration in an arbitrary graph.

Not Necessarily Connected Case

To characterize the children for the parent–child relation on general chordal graphs, we simply modify the condition "C has to be a clique of H" to "C has to be a clique of H, or a singleton of a vertex not in $V(H) \cup \{v\}$", which is equivalent to "C is a clique in the graph $(V(G), E(H))$". We can prove the lemma below analogously. We do not need to modify the definition of parents of chordal graphs.

Lemma 4.36 *Let H be a chordal subgraph of $G = (V, E)$. Let C be a clique of H, or a singleton of a vertex not in $V(H) \cup \{v\}$. For a vertex $v \notin V(H)$, $G_H(v, C)$ is a child of H if and only if one of the following conditions holds.*

(1) $|C| < k(H)$

(2) $|C| = k(H)$ *and* $v < \min(S^*(H) \setminus C)$

(3) $|C| = k(H) + 1$, $S^*(H) \subseteq C$ *and* $v < \min(S^*(H) \cup (S_{k(H)+1}(H) \setminus C))$.

In the case of Lemma 4.33, it is clear that there is a bijection between $G_H(v, C)$ and a pair of (v, C). However, in the case of Lemma 4.36, if C is a singleton of a vertex $w \notin V(H) \cup \{v\}$, we can sometimes swap v and w, that is, $G_H(w, \{v\})$ is also a child of H and $G_H(w, \{v\})$ is equal to $G_H(v, C)$. Thus, if we generate all children satisfying one of (1), (2) or (3), we sometimes generate the same child twice. To avoid this, we must not generate a child if C is a singleton w and w is younger than v.

From the proofs of the lemmas, we directly obtain the following corollary.

Corollary 4.37 *If $G_H(v, C)$ is a child of H, then $k(G_H(v, C)) = |C|$.*

Enumeration of Cliques in a Chordal Graph

The algorithm described in Figure 4.3 requires a subroutine to enumerate cliques of sizes at most k in a given chordal graph. We can enumerate cliques in a general graph $G = (V, E)$ in $O(|V|)$ time for each by a simple backtracking algorithm. However, we cannot use this algorithm for an enumeration algorithm that takes $O(1)$ time for each clique. Here, we describe a new algorithm to enumerate all size-restricted cliques in a chordal graph H that takes $O(1)$ time for each clique.

Our algorithm to do this is:

```
procedure enum_cliques(H, k)
H : chordal graph, k : integer;
begin
   Let v be a simplicial vertex of H.
   Enumerate all cliques of size at most k that contain v.
   Eliminate v from H.
   enum_cliques(H,k);
end.
```

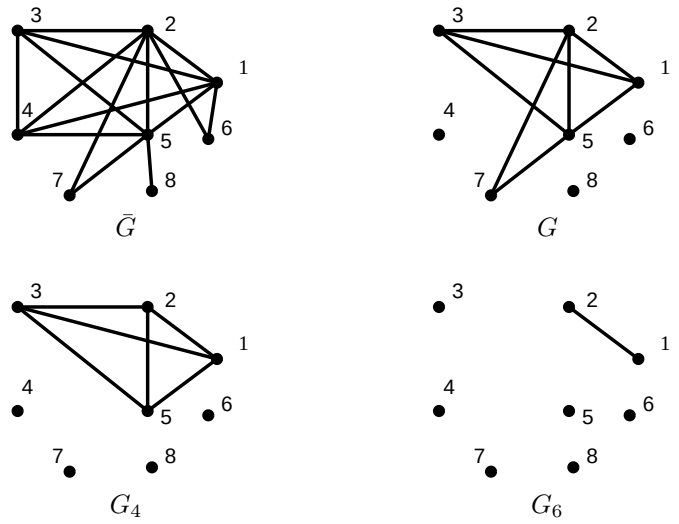

Figure 4.4: Examples of G, H, G_4 and G_6.

Let v be a simplicial vertex of H. Consider a partition of the set of cliques in H, all cliques including v, and all cliques not including v. Since the neighbors of v form a clique, v and its neighbors induce a complete graph. Thus, we can enumerate, as a vertex set, every clique that includes v and whose size is up to k by combining v and every vertex subset whose size is at most $k-1$ and whose elements are neighbors of v. We can enumerate the cliques of size at most k and not including v recursively, that is, we can enumerate all cliques of size at most k in the graph obtained by eliminating v from H. We choose the vertex v in each level of the recursive calls along a perfect elimination ordering so that only constant time is needed to obtain a simplicial vertex in each level of the recursive calls, if a perfect elimination ordering of H is already known. Note that we can generate each subset of size at most $k-1$ with reverse search, where we define the parent subset of a subset P including more than one elements as a subset obtained by removing the biggest element, as element number, from P. It is easy to obtain every child of a subset in constant time simply by adding to a subset of size at most $k-2$ an element whose element number is bigger than those of all elements in it; then the resulting subset is a child of the subset, and all children are obtained in this way. Thus, the following theorems are obtained.

Theorem 4.38 *We can enumerate all cliques of sizes at most k in a chordal graph in constant time for each clique and additional time to obtain a perfect elimination ordering, so that the size of the difference between two consecutive cliques is constant.*

If we think of the case that k is equal to n, the following theorem is also obtained.

Theorem 4.39 *We can enumerate all cliques in a chordal graph in constant time for each clique, so that the size of the difference between two consecutive cliques is constant.*

Time Complexity

The remaining problem is about time complexity. We refer to the time complexity of our algorithms here. First, we show that our algorithm to enumerate all connected chordal graphs in a complete graph costs constant time on average to enumerate every chordal graph. To show that the time complexity to enumerate every chordal graph in a complete graph is constant is done analogously. Bounding the time complexity in the case of enumerating chordal graphs in G, which is not a complete graph, needs some additional observations.

We define an iteration of the algorithm as the operations in a vertex of the computation tree, which is a tree representation of the recursive structure of an execution of the algorithm. Thus, an iteration corresponds to the operations in an execution of enum_sub_connected_chordal excluding the operations in the recursive calls generated from it. Iterations and connected chordal graphs have a one-to-one correspondence, thus we call an iteration inputting a chordal graph H *iteration of* H. In the rest of this subsection, we show that the computation time of an iteration of H is linear in the number of children of H. To show this, we bound the computation time of each step in an iteration one by one.

It is clear that we can run steps 1 and 2 in Figure 4.3 in constant time. From Corollary 4.37, we can compute $k(H)$ in constant time. Thus, we can perform the conditional branch in step 3 in constant time. In order to execute step 3 quickly in the case $k(H) = 1$, we maintain a sorted list of the vertices in $S_1(H)$ at every iteration. We can delete a vertex from the list in constant time. When the algorithm constructs $G_H(v, C)$ and adds v to $S_1(H)$ in some iteration, v is always younger than any vertex in $S_1(H)$. Thus, we can add a vertex v to $S_1(H)$ in constant time by attaching v to the head of the list.

To compute $S_d(H)$ and $\min(S_d(H))$ in step 4 takes $\mathrm{O}(|V(H)|)$ time. When we execute step 4, we have $k(H) \geq 2$. Thus, the set of cliques of sizes at most $k(H) - 1$ includes cliques whose sizes are one. We can also execute step 7 in $\mathrm{O}(|V(H)|)$ time, since at most one clique satisfies the condition of step 7. Since the number of cliques of one vertex in H is $|V(H)|$, H has at least $|V(H)|$ children. Therefore, the computation time for steps 4 and 7 is linear in the number of children of H. This means that the computation time is at most linear in the number of children.

The enumeration algorithm of cliques in a chordal graph requires a perfect elimination ordering. Since in each iteration the algorithm adds a simplicial vertex to the graph, the ordering of the vertices added to obtain H reversely forms a perfect elimination ordering. Thus, we can keep a perfect elimination ordering of the current operating graph in memory, and update it in constant time at each iteration.

Step 6 takes a long time in a straightforward way. To avoid this, we use cliques C' of size $k(H) - 1$ found in step 5. We find all vertices $u \in V(H)$ such

that there is a vertex $v \notin V(H)$ satisfying $v < \min(S^*(H) \setminus (C' \cup \{u\}))$. To satisfy the condition, $S^*(H) \setminus C'$ includes at most one vertex younger than the minimum vertex not included in H. We can check this in $\mathrm{O}(|C'|)$ time.

Above, we saw that steps 5, 6 and the maintenance of the sorted list of the vertices in $S_1(H)$ take $\mathrm{O}(|C|)$ time for each child. We can reduce the time to compute $G_H(v, C)$ in step 5 by using $G_H(v, C')$ where C' is the last clique obtained. By modifying $G_H(v, C')$ to obtain $G_H(v, C)$, we can reduce the time complexity to $\mathrm{O}((C \setminus C') \cup (C' \setminus C))$. Thus, from Theorem 4.38, the reduced computation time complexity is constant time for each on average. From similar observation, the computation time complexities for steps 5, 6 and the maintenance of the sorted list of the vertices in $S_1(H)$ are bounded by a constant for each child on average. Therefore, the following theorem is obtained.

Theorem 4.40 *For a given complete graph K_n, we can enumerate all connected chordal graphs, which are equivalent to all edge subsets of K_n inducing connected chordal graphs, in constant time for each edge subset, and in $O(n^2)$ memory.*

In an analogous way, the following theorem is also obtained.

Theorem 4.41 *For a given complete graph K_n, we can enumerate all chordal graphs, which are equivalent to all edge subsets of K_n inducing chordal graphs, in constant time for each edge subset, and in $O(n^2)$ memory.*

In the case of generating all connected or not necessarily connected chordal graphs in an arbitrary graph $G = (V, E)$, we have to compute $N(H, u)$ for all $u \in V \setminus V(H)$, since $N(H, u) \setminus N(G_H(v, C), u) = \{v\} \neq \emptyset$ (In the case of enumerating connected chordal graphs, since $N(H, u)$ is always equal to $N(G_H(v, C), u)$, we do not have to compute $N(H, u)$ for each iteration). Computing $N(H, u)$ for each u with no information takes a long time. In order to reduce the time, we maintain $N(H, u)$ along the changes of the current chordal graph H. As a result, we will show that the computation time to obtain all $N(H', u)$ to be used is $\mathrm{O}(|\mathrm{Chd}(H')| + 1)$, where $\mathrm{Chd}(G')$ is the set of children of H'. This implies that the computational time with respect to this operation is constant for each output chordal graph on average, since the sum of the number of children over all vertices in a tree is less than or equal to the number of vertices in the tree.

For each vertex not in H, let $M(H, v)$ be the set of vertices in $V \setminus (V(H) \cup \{v\})$ and adjacent to v. We keep $M(H, u)$ $(u \in V \setminus V(H))$ sorted in their indices. Suppose that in an iteration we obtain a child $H' = G_H(v, C)$ of H for some v and a clique C in $N(H, v)$. To generate a recursive call with respect to H', we compute $N(H', u)$ and $M(H', u)$ from $N(H, u)$ and $M(H, u)$ for each $u \in V \setminus (V(H) \cup \{v\})$. The computation with respect to $M(H', u)$ is $\mathrm{O}(|M(H, v)|)$, since $M(H', u) = M(H, u) \setminus \{v\}$ if $u \in M(H, v)$ and $M(H', u) = M(H, u)$ otherwise.

For any vertex u in neither H nor $M(H, v)$, $N(H', u)$ is equal to $N(H, u)$. For any vertex $u \in M(H, v)$, $N(H', u)$ is obtained from $N(H, u)$ by adding v and edges connecting v and vertices both in $N(H, v)$ and C. In this way, the computation time to obtain $N(H', u)$ for all $u \in M(H, v)$ is $\mathrm{O}(\sum_{u \in M(H,v)} |V(N(H, u))|)$,

where $V(N(H,u))$ is the vertex set of $N(H,u)$. If $|C| \geq 2$, we obtain a child by adding an edge (u,w) to H' for any w in $N(H',u)$, since $k(H') \geq 2$ holds. Hence, we have

$$|M(H,v)| + \sum_{u \in M(H,v)} |V(N(H,u))| \leq \mathrm{O}(|\mathrm{Chd}(G')|).$$

Suppose that $|C| = 1$. We denote the unique element in C by w (i.e., $C = \{w\}$). In this case, $N(H',u) \setminus N(H,u)$ includes at most three edges, which are (u,v), (u,w) and (v,w). Thus, we can obtain $N(H',u)$ from $N(H,u)$ in constant time by looking the adjacency matrix of G. The computation time to obtain $N(H',u)$ for all $u \in M(H,v)$ is $\mathrm{O}(|M(H,v)|)$. We first consider the case that $|S_1(H')| = 1$, that is, v is the unique simplicial vertex in H' whose degree is one. In this case, for each $u \in M(H,v)$, H' has a child obtained by adding the edge (u,v) to H'. Hence, $|M(H,v)| = \mathrm{O}(|\mathrm{Chd}(H')|)$, and the time to compute all $N(H',u)$ for all $u \in M(H,v)$ is $\mathrm{O}(|\mathrm{Chd}(H')|)$.

Next, consider the case that $|S_1(H')| > 1$. Then, v is the minimum vertex among $S_1(H')$. Let v' be the second minimum in $S_1(H')$. We have the following property.

Lemma 4.42 *For any descendant $\tilde{H}$ of H', $s^*(\tilde{H}) < v'$, the degree of $s^*(\tilde{H})$ is one, and $S_1(\tilde{H})$ includes at least two vertices no greater than v' that are not adjacent to each other.*

Proof Suppose that $\tilde{H}$ does not satisfy the condition, and without loss of generality, any ancestor of $\tilde{H}$ satisfies the condition. Let $P(\tilde{H})$ be the parent of $\tilde{H}$. From the assumption, $s^*(P(\tilde{H})) < v'$, the degree of $s^*(P(\tilde{H}))$ is one, and $S_1(P(\tilde{H}))$ includes at least two vertices not greater than v' that are not adjacent to each other. Then, $s^*(\tilde{H})$ has to be connected to at least two vertices in $S_1(P(\tilde{H}))$ that are not adjacent to each other. This contradicts the fact that the neighbors of $s^*(\tilde{H})$ form a clique. □

From the lemma, we can see that for any descendant of H', $u > v'$ will never be added. Thus, $N(H,u)$ does not need to be computed for any $u > v'$. We compute $N(H',u)$ and $M(H',u)$ for all vertices $u \in M(H,v)$ with $u < v'$. We can do this in constant time for each $u \in M(H,v)$ with $u < v'$ by tracing $M(H,v)$ in the ascending order. For each $u \in M(H,v)$ with $u' > v$, H' has a child obtained by adding the edge (u,v) to H'. Thus, $|\{u | u \in M(H,v), u < v'\}| = \mathrm{O}(|\mathrm{Chd}(H')|)$.

In every case, we have proved that the computation time to obtain $N(H',u)$ and $M(H',u)$ is $\mathrm{O}(|\mathrm{Chd}(H')| + 1)$. Thus, the following theorem is obtained.

Theorem 4.43 *For a given graph G, we can enumerate all chordal graphs, which are equivalent to all edge subsets of G inducing chordal graphs, in constant time for each. The memory space necessary to run the algorithm is $O(n^3)$.*

4.2.2 Subgraph Enumerations with Forbidden Induced Subgraphs

Many graph classes are known to be subclasses of chordal graphs. Some of them are characterized by forbidden induced subgraphs, where graphs are characterized by forbidden induced subgraph F means that the graphs do not contain subgraphs isomorphic to F as induced subgraphs. (We sometimes say that the graphs are F-free.) Chordal graphs are characterized by cycles of the size more than three as forbidden induced subgraphs. Interval graphs are characterized as asteroidal triple-free chordal graphs. Block graphs/Ptolemaic graphs are also diamond/gem-free chordal graphs (see Section 2.2). In this subsection, we consider subgraph enumeration of graphs belonging to subclasses of chordal graphs and characterize by forbidden induced subgraphs.

Notice that we have only to consider the case that the forbidden graph F is chordal, since otherwise F contains a chordless cycle as an induced subgraph, and it means that graphs containing F as induced subgraphs automatically contain cordless cycles as induced subgraphs and are not chordal.

The main idea of our algorithm is to enumerate chordal graphs in the way of the previous section (Section 4.2.1), and when a forbidden graph is reached, to stop searching the descendants. First, we show that, in this way, we can enumerate all chordal graphs that have no given induced subgraphs.

Lemma 4.44 *Given a graph F and given a non-F-free chordal graph H, any descendant of H in the enumeration tree described in Section 4.2.1 is non-F-free.*

Proof We prove the equivalent statement: The ancestors of F-free chordal graph H' are always F-free.

Given an F-free chordal graph H', the parent of H' in the enumeration tree described in Section 4.2.1 is an induced subgraph of H'. It thus does not contain a graph isomorphic to F as an induced subgraph, as well. Applying this recursively to the ancestors of H', we obtain the statement. □

In the algorithm, we generate every child of a chordal graph H, and continue the recursion only when the child does not contain the forbidden graph as an induced subgraph. From the lemma above, the algorithm generates all the F-free chordal graphs included in a given graph G.

We can enumerate F-free chordal subgraphs in given graph G by the algorithm described above. The problem is that there may be exponentially many children for a chordal graph. Thus, if we check each of them to determine whether or not they contain the forbidden graph as an induced subgraph, and if many of them do contain the forbidden graphs as induced subgraphs, the algorithm becomes a non-polynomial time delay one. Therefore, it is not true that we can always develop polynomial time delay algorithms in a naive way. However, if F has some good properties, such as the sizes of F is constant, we can develop polynomial time delay subgraph enumeration algorithms.

For the simplest example, we consider the case of block graphs. Given a block graph H, we add a simplicial vertex v to a clique C in H and obtain

$G_H(v, C)$ in the algorithm. When $|C| = 1$, there is no possibility that $G_H(v, C)$ becomes a non-block graph (it is impossible that $G_H(v, C)$ contains a diamond as an induced subgraph).

When $|C| = 2$, it is possible that $G_H(v, C)$ contains diamond as an induced subgraph. If the vertices of C have a common neighbor v', then v, C and v' induce a diamond (see Figure 4.5). Similarly, when $|C| > 2$, if two vertices in C have a common neighbor $v' \notin C$, then v, v' and the two vertices in C also induce a diamond.

Observation 4.45 *Given a block graph H, clique C of H and a vertex $v \notin V(H)$, $G_H(v, C)$ is not a block graph if and only if any two vertices in C do not have a common neighbor $v' \notin C$ in H.*

This observation is equivalent to the one below.

Observation 4.46 *Given a block graph H, clique C of H and a vertex $v \notin V(H)$, $G_H(v, C)$ is a block graph if and only if every vertex v' adjacent to two vertices of C is in C.*

In order to avoid generating exponentially many chordal graphs whose vertices induce diamonds, we use the binary partition method to enumerate every clique C such that $G_H(v, C)$ is a child of H; we divide the problem into two subproblems: to enumerate every clique C containing edge e where $G_H(v, C)$ is a child of H, and to enumerate every clique not containing edge e where $G_H(v, C)$ is a child of H.

To enumerate cliques containing edge $e = (v_1, v_2)$, we first enumerate all the vertices that are adjacent to both v_1 and v_2 in H. We denote the set of such vertices by $\tilde{V}(v_1, v_2)$. From the observation 4.46, when C contains v_1 and v_2, $G_H(v, C)$ is diamond-free only if $C = \tilde{V}(v_1, v_2) \cup \{v_1, v_2\}$. Therefore, if vertices of $\tilde{V}(v_1, v_2)$ do not induce a clique, the child $G_H(v, C)$ for any C containing v_1 and v_2 is always non-block. Nor, if some edge connecting v and a vertex of $\tilde{V}(v_1, v_2)$ is not in G, that is the edge is not allowed to be added, $G_H(v, C)$ for any C containing v_1 and v_2 is always non-block. Thus, in such cases, we stop enumerating cliques containing e. Otherwise, we have only to determine whether or not $G_H(v, \tilde{V}(v_1, v_2) \cup \{v_1, v_2\})$ is a block graph, and if it is a block graph, output $\tilde{V}(v_1, v_2) \cup \{v_1, v_2\}$. From the observation above, there is no other clique C' containing e and $G_H(v, C')$ is a block graph. To enumerate cliques not containing e, we only have to remove the edge e from H and recursively continue the enumeration by the binary partition.

The time complexity to enumerate one clique is thus $\mathrm{O}(mn^2)$.

Theorem 4.47 *There is an algorithm for block subgraph enumeration algorithm that enumerates each block graph in $O(mn^2)$ time.*

A Ptolemaic subgraph enumeration algorithm can be developed similarly. $G_H(v, C)$ is Ptolemaic unless three vertices in C and a vertex $w \in V(H)$ induce a diamond. Thus, the observation for block graphs is slightly changed.

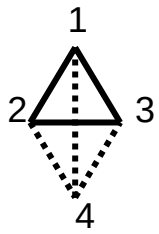

Figure 4.5: When we add a simplicial vertex 4 and connect it to vertices 2 and 3, we also have to connect vertex 1, since otherwise vertices 1, 2, 3, and 4 induce a diamond.

Observation 4.48 *Given a Ptolemaic graph H, clique C of H and a vertex $v \notin V(H)$, $G_H(v, C)$ is not Ptolemaic if and only if every vertex v' adjacent to two vertices $v_1, v_2 \in C$ and a vertex that is adjacent to v_1 or v_2, is in C.*

We redefine $\tilde{V}(v_1, v_2)$ as vertices that satisfy the below, so that $\tilde{V}(v_1, v_2)$ keeps the property that when C contains v_1 and v_2, $G_H(v, C)$ is diamond-free only if $C = \tilde{V}(v_1, v_2) \cup \{v_1, v_2\}$.

- adjacent to v_1 and v_2,
- adjacent to $w \neq v_1, v_2$ which is adjacent to v_1 and not adjacent to v_2, or
- adjacent to $w \neq v_1, v_2$ which is not adjacent to v_1 and is adjacent to v_2.

Thus, obtaining $\tilde{V}(v_1, v_2)$ costs $\mathrm{O}(n^3)$. However it is still polynomial in n.

Theorem 4.49 *There is an algorithm for a Ptolemaic subgraph enumeration algorithm that enumerates each Ptolemaic graph in $O(mn^3)$ time.*

Chapter 5

Conclusion

Focusing on polynomial time delay subgraph/supergraph enumeration algorithms, we developed two types of algorithms based on the reverse search method. One type defines a parent by an edge addition or an edge removal and the other defines a parent by a simplicial vertex elimination. The first type uses the fact that there are only $O(n^2)$ edges in K_n, and achieves polynomial time delay algorithms. Both subgraph enumeration algorithms and supergraph enumeration algorithms can be developed by this method. The second type of algorithm uses nice properties of simplicial vertices and the fact that cliques in a chordal graph can be enumerated quickly. This type of algorithm for chordal subgraph enumeration is faster than the first type (it needs only constant time to enumerate each chordal graph). However, only subgraph enumeration algorithms can be developed by this method.

In developing the first type of algorithm, We focused on edge additions and edge removals on such graphs as chordal graphs and interval graphs. We showed that rooted tree structures can be defined by these additions and removals on all the chordal (or interval, strongly chordal, etc.) supergraphs of a given graph, and on all the chordal (etc.) subgraphs of a given graph. Using the structures, we introduced algorithms for the subgraph and supergraph enumerations. The time complexity to find every chordal, interval, strongly chordal or split graph containing a given graph are $O(n^3), O(n^3)$, $O(m \cdot n^2)$ and $O(n)$ for each, respectively. The time complexity to find all chordal, interval, strongly chordal, weakly chordal or split graphs included in a given graph is $O((n+m)^2), O((n+m)^2), O(m \cdot \min(m \log n, n^2))$, $O(m^4)$ and $O(n)$ for each, respectively. This method is general, and we think that it can be used to enumerate many graphs not stated in this thesis. To develop a polynomial time delay subgraph enumeration algorithm for graph class G, we have only to show that we can always obtain a graph in G by removing an edge from a non-empty graph in G. To develop a polynomial time delay supergraph enumeration algorithm for graph class G, we have only to show that we can always obtain a graph in G by adding an edge to a graph in G that is not K_n, though, in order to obtain faster algorithms, we have to prove some additional facts specific to each problem.

For the second type of algorithm, we developed an algorithm to enumerate all the cliques of a chordal graph in constant time for each. Since we define the parent of a chordal graph by simplicial vertex elimination, the children of a chordal graph are obtained by adding a simplicial vertex to cliques. Thus, using this algorithm, every child of a chordal graph can be generated efficiently, in O(1) time for each. This method is specific for chordal graphs. However, we also developed polynomial time delay algorithms for subgraph enumeration of graphs which are chordal and have some forbidden induced subgraphs.

Bibliography

[1] Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases, Proceedings of VLDB '94 (1994) 487–499

[2] Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., Arikawa, S.: Efficient substructure discovery from large semi-structured data, Proceedings of Second SIAM International Conference on Data Mining 2002 (2002) 158–174

[3] Avis, D., and Fukuda, K.: Reverse search for enumeration, Discrete Applied Mathematics **65** (1996) 21–46

[4] Beeri, C., Fagin, R., Maier, D., and Yanakakis, M.: On the desirability of acyclic database schemes, Journal of the ACM **30** (1983) 479–513

[5] Bender, E. A., Richmond, L. B., and Wormald, N. C.: Almost all chordal graphs split, Journal of the Australian Mathematical Society, (A)**38** (1985) 214–221

[6] Bitner, J. R., Ehrlich, G., and Reingold, E.: Efficient generation of the binary reflected Gray code, Communication of the ACM **19(9)** (1976) 517–521

[7] Blair, J. R. S., and Peyton, B.: An introduction to chordal graphs and clique trees, Graph Theory and Sparse Matrix Computation, **IMA56** (1993) 1–29

[8] Booth, K. S., and Lueker, G. S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ tree algorithms, Journal of Computing and System Sciences **13** (1976) 335–379

[9] Brandstädt, A., Le, V. B., and Spinrad, J.: Graph classes: a survey, SIAM Monographs on Discrete Math and Applications, **3** (1999)

[10] Chandran, L. S., Ibarra, L., Ruskey, F., and Sawada, J.: Fast generation of all perfect elimination orderings of a chordal graph, Theoretical Computer Science, **307** (2003) 303–317

[11] Corneil, D. G., Olariu, S., and Stewart L.: The ultimate interval graph recognition algorithm?, Proceedings of 9th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, 1998, 175–180

[12] Dirac, G. A.: On rigid circuit graphs, Abhandl. Math. Seminar Univ. Hamburg **25** (1961) 71–76

[13] Endo, Y.: Bunkaikano moderu no rekkyo arugorizumu (in Japanese), graduation thesis at Univ. Tokyo 2004

[14] Farber, M.: Characterizations of strongly chordal graphs, Discrete Mathematics **43** (1983) 173–189

[15] Földes, S., and Hammer, P. L.: Split graphs, Proceedings of 8th South–Eastern Conference on Combinatorics, Graph Theory and Computing (1977) 311–315

[16] Goldberg, L. A.: Efficient algorithms for listing combinatorial structures, Cambridge University Press, New York, 1993

[17] Golumbic, M. C.: Algorithmic graph theory and perfect graphs (2nd), Elsevier, 2004

[18] Golumbic, M. C., Kaplan, H., and Shamir R.: Graph sandwich problems, Journal of Algorithms **19** (1995) 449–473

[19] Gray, F.: Pulse code communications, U. S. Patent 2632058, 1953

[20] Kay, D. C., and Chartrand, G.: A characterization of certain ptolemaic graphs, Canadian Journal of Mathematics **17** (1965) 342–346

[21] Micheal, H., Kurtz, S., and Ohlebusch, E.: Efficient multiple genome alignment, Bioinformatics **18(1)** (2002) S312–S320

[22] Hammer, P. L., and Simeone B.: The splittance of a graph, Combinatorica **1** (1981) 275–284

[23] Hayward, R. B.: Weakly triangulated graphs, Journal of Combinatoric Theory **(B) 39** (1985) 200–208

[24] Hayward, R. B.: Generating weakly triangulated graphs, Journal of Graph Theory **21** (1996) 67-69

[25] Hayward, R. B., Spinrad, J., and Sritharan, R.: Weakly Chordal Graph Algorithms via Handles, Proceedings of 11th SODA (2000) 42–49

[26] Inokuchi, A., Washio, T., Motoda, H.: An apriori-based algorithm for mining frequent substructures from graph data, Lecture Notes in Computer Science **1910** (2000) 13–23

[27] Kiyomi, M., and Uno, T.: Generating chordal graphs included in given graphs, IEICE Transactions on Information and Systems **E89-D** (2006) 763–770

[28] Lekkerkerker, C. G., and Boland, J. C.: Representation of a finite graph by a set of intervals on the real line, Fund Math 51 (1962), 45–64

[29] Makino, and K. Uno, T.: New algorithms for enumerating all maximal cliques, Lecture Notes in Computer Science **3111** (2004) 260–272

[30] Nakano, S.: Enumerating floorplans with n rooms, Lecture Notes in Computer Science **2223** (2001) 107–115

[31] Nakano, S.: Efficient generation of triconnected plane triangulations, Computational Geometry Theory and Applications **27(2)** (2004) 109–122

[32] Nakano, S., and Uno, T.: Constant time generation of trees with specified diameter, Lecture Notes in Computer Science **3353** (2004) 33–45

[33] Nijenhuis, A., and Wilf, H. S.: Combinatorial algorithms for computers and calculators, Academic Press, 1978

[34] Ott, S., and Miyano, S.: Enumeration of likely gene networks and network motif extraction for large gene networks, Genome Informatics **14** (2003) 354–355

[35] Read, R. C., and Tarjan, R. E.: Bounds on backtrack algorithms for listing cycles, paths, and spanning trees, Networks **5** (1975) 237-252

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

[37] Savage, C.: A survey of combinatorial Gray codes, SIAM Review **39** (1997) 605–629

[38] Spinrad, J. P.: Doubly lexical ordering of dense 0-1 matrices, Information Processing Letters **45** (1993) 229–235

[39] Taniguchi, K., Sakamoto, H., Arimura, H., Shimozono, S., Arikawa, S.: Mining semi-structured data by path expressions, Lecture Notes in Artificial Intelligence **2226** (2001) 378–388

[40] Uno, T.: A new approach for speeding up enumeration algorithms, Lecture Notes in Computer Science **1533** (1998) 287–296

[41] Uno, T.: Two general methods to reduce delay and change of enumeration algorithms, National Institute of Informatics (in Japan) technical report (2003)

[42] Uno, T., and Uehara, R.: Laminar structure of Ptolemaic graphs and its applications, Lecture Notes in Computer Science **3827** (2005) 186–195

[43] Wells, M. B.: Generation of permutations by transposition, Mathematics of Computation **15** (1961) 192–195

[44] Whittaker, J.: Graphical models in applied multivariate statistics, Wiley, New York, 1990

Publication List

- Masashi Kiyomi, and Takeaki Uno: Generating chordal graphs included in given graphs, Transactions on Information and Systems, IEICE, 2006
- Masashi Kiyomi, Shuji Kijima, and Takeaki Uno: Listing chordal graphs and interval graphs, Proceedings of 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (2006)
- Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura: LCM ver 3.: Collaboration of array, bitmap and prefix tree for frequent itemset mining, Proceedings of Open Source Data Mining Workshop (2005)
- Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura: LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets, Proceedings of Workshop on Frequent Itemset Mining Implementations (2004) (Best Implementation Award)
- Masashi Kiyomi, and Takeaki Uno: Enumerating split graph, The 9th Japan-Korea Joint Workshop on Algorithms and Computation (2006)
- Masashi Kiyomi, and Takeaki Uno: Enumerating labeled chordal graph on complete graph, Proceedings of 4th Japanese-Hungarian Symposium on Discrete Mathematics and Its Applications (2005)

Acknowledgements

My supervisor Professor Takeaki Uno helped me many time to write this thesis and other various thing. I began this research by his influence. He and his family helped life at Switzerland where I visited him to discuss about this thesis. And, of cource, he gave me many appropriate advices to write this thesis. I thank him about these and other many things.

Mr. Shuji Kijima at the University of Tokyo gave some ideas in lemmas about chordal graphs. He simplified the proof of Lemma 4.1. I enjoyed the discussions with him.