JAIST Repository
https://dspace.jaist.ac.jp/

Title	幾何的特徴を持つグラフクラスに対する効率のよいアルゴリズムに関する研究
Author(s)	齋藤，寿樹
Citation	
Issue Date	2010-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/8866
Rights	
Description	Supervisor:上原隆平，情報科学研究科，博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Efficient Algorithms for Geometric Graph Classes

by

Toshiki SAITOH

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Associate Professor Ryuhei Uehara

School of Information Science
Japan Advanced Institute of Science and Technology

January 8, 2010

Abstract

It is said that every NP-hard problem has no efficient algorithm. However, many NP-hard problems on general graphs can be solved efficiently if we restrict graphs to a geometric graph class. For example, interval graphs form one of the geometric graph class. Coloring problem which is well known NP-hard problem can be solved in linear time on interval graphs. A variety of geometric graph classes have been proposed and studied. In this paper, we treat with some problems for geometric graph classes. These problems are random generation, enumeration, and graph reconstruction, mainly.

We treat with unlabeled graphs to avoid redundancy. We propose random generation and enumeration algorithms for connected proper interval graphs. We use counting for random generation algorithms, so we first give the number of connected proper interval graphs of n vertices. Based on the number, we present a simple algorithm that generates a connected proper interval graph uniformly at random up to isomorphism. Next we propose an enumeration algorithm of connected proper interval graphs. This algorithm is based on the reverse search, and it outputs each connected proper interval graph in O(1) time. Then we propose random generation and enumeration algorithms for connected bipartite permutation graphs. These algorithms are extension of the algorithms of proper interval graphs.

The graph reconstruction conjecture is a long-standing open problem in graph theory. There are many algorithmic studies related it besides mathematical studies, such as deck checking, legitimate deck, preimage construction, and preimage counting. We study these algorithmic problems limiting the graph classes to interval graphs, permutation graphs, and distance-hereditary graphs. Since we can solve graph isomorphism problem for these graph classes in polynomial time, deck checking for these graph classes are easily done in polynomial time. Since the number of interval graphs that can be obtained from a graph by adding a vertex and edges incident to it can be exponentially large, developing polynomial time algorithms for legitimate deck, preimage construction, and preimage counting on these graphs are not trivial. We present that these problems are solvable in polynomial time on these graph classes.

Acknowledgments

First of all, I would like to express my sincere gratitude to my principal adviser Professor Ryuhei Uehara of Japan Advanced Institute of Science and Technology for his academic advice and kind guidance during this work. His persistent encouragement and support were really helpful, and his way of looking at problem, way of presenting materials, and everything were very exciting to me. He has had a profound influence throughout my academic career. At the most basic level, he introduced me to the exciting subject of graph algorithm, and provided key insights and direction on the research side; problem-solving techniques, publications, collaborations, and academic politics. Especially, he provided me with experience of meeting to many advanced research topics and great researchers who work world wide and actively in the field of theoretical computer science. He also gave me some jobs as assistant and the pay was helpful. Again, I show my gratitude to my supervisor.

I would like to thank my adviser Professor Tetsuo Asano of Japan Advanced Institute of Science and Technology for his helpful suggestions, encouragements. He always allowed me to make remarks somewhat puerile or nonsense idea, and made some of them into interesting research themes with fruitful and conscientious discussions.

I would like to express my gratitude to Professor Mineo Kaneko who kindly admitted to be minor-research adviser, for helpful suggestions and encouragements.

I am no less grateful to the following people for their excellent comments and substantial supports: Associate Professor Mitsuo Motoki of Kanazawa Technical College, Assistant Professor Masashi Kiyomi of Japan Advanced Institute of Science and Technology, and the enumeration algorithm seminar's member.

Some of chapters in the thesis are based on joint papers with the following collaborators: Assistant Professor Katsuhisa Yamanaka of University of Electro-Communications and Mr. Yota Otachi of Gunma University.

Finally, I deeply thank my family for their love, patience, and encouragement, and for all that they have done for my sake; this work is dedicated to them.

Contents

Abstract **i**

Acknowledgments **ii**

1 Introduction **1**
1.1 Random Generation and Enumeration Problems 1
1.2 Graph Reconstruction Problem 2
1.3 Framework 3

2 Preliminaries **5**
2.1 Basic Graph Notations 5
2.2 Interval Graphs 6
2.2.1 Definition of Interval Graphs 6
2.2.2 Compact Interval Representation 7
2.2.3 $\mathcal{PQ}$-tree and $\mathcal{MPQ}$-tree 9
2.3 Proper Interval Graphs 10
2.3.1 Definition of Proper Interval Graphs 11
2.3.2 String Representation 11
2.4 Permutation Graphs 12
2.4.1 Definition of Permutation Graphs 12
2.4.2 Modular Decomposition 14
2.5 Bipartite Permutation Graphs 15
2.6 Distance-Hereditary Graphs 18
2.7 Dyck path and Motzkin path 19
2.8 Computational Model 20

3 Random Generation and Enumeration **22**
3.1 Random Generation of Proper Interval Graphs 22
3.2 Enumeration of Proper Interval Graphs 24
3.3 Random Generation of Bipartite Permutation Graphs 29
3.4 Enumeration of Bipartite Permutation Graphs 33

4 Reconstruction **39**
4.1 Interval Graphs 39
4.1.1 Deck Checking 39
4.1.2 Non-interval Graph Preimage Case 40
4.1.3 Connected Preimage Case 41
4.1.4 Disconnected Preimage Case 45

4.2 Permutation Graphs 45
4.2.1 Deck Checking 46
4.2.2 Non-permutation Graph Preimage Case 46
4.2.3 Non-critical Case 46
4.2.4 Critical Case 48
4.3 Distance-hereditary Graphs 50
4.3.1 Deck Checking 50
4.3.2 Non-distance-hereditary Graph Preimage Case 51
4.3.3 Distance-hereditary Preimage Case 51

5 Efficient Algorithm for $\mathcal{MPQ}$-tree **53**
5.1 Ordered Compact Interval Representation 53
5.2 Find all $\mathcal{P}$-nodes and $\mathcal{Q}$-nodes 55
5.3 Construct $\mathcal{MPQ}$-tree 57

6 Concluding Remarks **60**

A The canonical $\mathcal{MPQ}$-tree for an interval graph **61**

References **63**

Publications **67**

List of Figures

2.1 An interval graph and its interval representation. 7
2.2 The forbidden graphs of interval graphs. The part described k contains k vertices ($k \geq 0$). Thus (c) is a chordless cycle of more than three vertices, (d) has more than five vertices, and (e) has more than five vertices. 7
2.3 A compact interval representation of an interval graph. 8
2.4 (a) An interval graph G. (b) A $\mathcal{PQ}$-tree obtained from G with maximal cliques C_i ($i = 1, \ldots, 4$). (c) A $\mathcal{MPQ}$-tree of G. 9
2.5 (a) A proper interval graph G. (b) A proper interval representation of G. (c) A unit interval representation of G. 11
2.6 (a) A permutation graph. (b) its line representation. 13
2.7 (a) A line representation $\mathcal{L}$. (b)$\mathcal{L}^H$. (c)$\mathcal{L}^V$. (d)$\mathcal{L}^R$. 13
2.8 Forbidden graphs of a comparability graph ($k \geq 0$). 14
2.9 Graph and its modular decomposition 15
2.10 Graph H_n. 16
2.11 A bipartite permutation graph with its line representation. 17
2.12 Proper interval graphs from the bipartite permutation graph in Figure 2.11(a). . 17
2.13 Distance-hereditary graph. 19
2.14 Forbidden graphs of distance-hereditary graphs. The part described k contains k vertices ($k \geq 0$). (a) hole. (b) house. (c) domino. (d) gem. 19
2.15 Dyck path 20
2.16 Motzkin path 20

3.1 Family tree T_6 26
3.2 Case analysis of candidate indices. 27
3.3 An example of the bijection 29
3.4 The root in $S_{4,3}$. 33
3.5 Examples of the parents. 34
3.6 Family tree of $S_{4,3}$. 35
3.7 Construction of a representation in $S_{7,4}$ from the jump representation in $S_{6,5}$. . . 38

4.1 Constructing graph G' from candidate graph G for deck checking 40
4.2 Vertices corresponding to the enclosed intervals are end-vertex set. 41
4.3 Compact interval representations of G and $G - s$. In $G - s$, $S \setminus s$ is end-vertex set. 43
4.4 Compact interval representations of G and $G - s$. In $G - s$, $S \setminus s$ is not end-vertex set. 43
4.5 Adding an interval $[-1, -1]$ 44
4.6 Strong modules M_1, M_2, and M_3 are minimal. We add a line segment in the line representation of $G[M_3]$. 47

5.1 (*a*) An input interval representation. (*b*) The compact interval representation corresponding to (*a*). (*c*) Data structure of ordered compact interval representation. 54

List of Algorithms

1 find-all-child-strings 26
2 find-all-strings 27
3 find-all-child-rep 37
4 deck-checking 40
5 connected-interval-preimage 44
6 non-critical-preimage 48
7 critical-preimage 49
8 reconstruct-distance-hereditary 52
9 construct-$\mathcal{MPQ}$-tree 53
10 ordered-compact-interval-rep 55
11 find-all-$\mathcal{Q}$-node 56
12 determine-parent-child-relation 58
13 create-sections 58

Chapter 1

Introduction

It is said that every NP-hard problem has no efficient algorithm [18]. However, many NP-hard problems on general graphs can be solved efficiently if we restrict graphs to a geometric graph class. For example, interval graphs is one of the geometric graph class. Coloring problem which is well known NP-hard problem can be solved in linear time on interval graphs. A variety of geometric graph classes have been proposed and studied [9, 21, 48]. In this paper, we treat with some problems for geometric graph classes. These problems are random generation, enumeration, and graph reconstruction, mainly.

1.1 Random Generation and Enumeration Problems

Recently there has arisen need to process huge amounts of data in the areas of data mining, bioinformatics, etc. In order to find and classify knowledge automatically from the data, we assume that the data have a certain structure. We have to attain three efficiencies to deal with the complex structures: the structure has to be represented efficiently; essentially different instances have to be enumerated efficiently; and the properties of the structure have to be checked efficiently. In the area of graph drawing, there are several papers [7, 26, 35, 41]. From the viewpoint of graph classes, the previously studied structures are relatively primitive, and there are many unsolved problems for more complex structures: Trees are widely investigated as a model of such structured data [19, 29, 39, 40], and recently, distance-hereditary graphs are studied [42].

In this paper, we investigate counting, random generation, and enumeration of graph classes called proper interval graphs and bipartite permutation graphs. More precisely, we aim to count, generate, and enumerate unlabeled connected proper interval graphs and bipartite permutation graphs. From the practical point of view, "unlabeled" and "connected" are reasonable properties to avoid redundancy. On the other hand, however, they are also challenges to develop efficient algorithms. Especially, unlabeled property requires us to avoid generating isomorphic graphs. In other words, we have to recognize isomorphic graphs and suppress generating/counting/enumerating them twice or more. Roughly speaking, the graph isomorphism problem has to be solved efficiently for our target graph classes in this context. The graph isomorphism problem is one of well-known basic problems, and it is still hard on very restricted graph classes [51]. There are two well known graph classes that the graph isomorphism problem can be solved in polynomial time; interval graphs [36] and permutation graphs [10]. Hence, these graph classes are the final goal in this framework. We mention that these graph classes have been widely investigated since they are very basic graph classes from the viewpoint of

graph theory. Moreover, many algorithms have been developed that run efficiently on these graph classes (see, e.g., [9, 21, 48]) since they have useful properties. From the practical point of view, when an efficient algorithm is developed and implemented, we have to check its reliability. In the time, we have to prepare many or all graphs in the class. Hence, for such popular graph classes, efficient random generation and enumeration are important.

Unlabeled proper interval graphs can be naturally represented by a language over an alphabet $\Sigma = \{`[', `]'\}$. The number of strings representing proper interval graphs is strongly related to a well known notion called Dyck path, which is a staircase walk from $(0, 0)$ to $(2n, 0)$ that lies strictly below (but may touch) the diagonal $x = 0$. The number of Dyck paths of length n is equal to Catalan number $C(n)$. Thus, our results for counting and random generation of proper interval graphs with n vertices are strongly related to $C(n)$. The main difference is that we have to consider isomorphism and symmetry in the case of proper interval graphs. For example, to generate an unlabeled connected proper interval graph uniformly at random, we have to consider the number of valid representations of each graph since it depends on the symmetry of the graph. For example, to generate an unlabeled connected proper interval graph uniformly at random, we have to consider the number of valid representations of each graph since it depends on the symmetricity of the graph. We show in Section 3.1 that the number of connected proper interval graphs of $n + 1$ vertices is $\frac{1}{2}(C(n) + \binom{n}{\lfloor n/2 \rfloor})$. Extending the result, we give an $O(n)$ time and a linear space algorithm that generates a connected proper interval graph with n vertices uniformly at random.

In Section 3.3, we will show that an unlabeled connected bipartite permutation graph is strongly related to an extension of a Motzkin path. Motzkin path is one natural extension of the notion of Dyck path; a Dyck path can be seen as a sequence of $+1$ and -1, and a Motzkin path can be seen as a sequence of $+1$, -1, and 0. An unlabeled connected bipartite permutation graph related to a 2-Motzkin path that consists of $+1$, -1, $+0$, and -0. As we will see, bipartite permutation graphs have a certain structure, which can be seen as a generalization of the structure appearing in proper interval graphs implicitly. That is, developing some new nontrivial techniques based on the results in proper interval graphs, we advance the random generation algorithm of proper interval graphs to bipartite permutation graphs.

Enumeration algorithms of proper interval graphs and bipartite permutation graphs are based on the reverse search developed by Avis and Fukuda [2]. We design a good parent-child relation among the geometric representations of these graph classes in order to perform the reverse search efficiently. The relation allows us to perform each step of the reverse search in $O(1)$ time, and hence we have efficient algorithms that enumerates every unlabeled connected proper interval graph and bipartite permutation graphs with n vertices in $O(1)$ time and $O(n)$ space. (Each graph G is output in the form of the difference of edges between G and the previous one so that the algorithm can output it in $O(1)$ time.)

1.2 Graph Reconstruction Problem

Given a simple graph $G = (V, E)$, we call the multi-set $\{G - v \mid v \in V\}$ the *deck* of G where $G - v$ is a graph obtained from G by removing vertex v and the incident edges. The graph reconstruction conjecture by Ulam and Kelly[1] is that for any multi-set D of graphs with at least two vertices there is at most one graph whose deck is D. We call a graph whose deck

[1]Determining the first person who proposed the graph reconstruction conjecture is difficult, actually. See [24] for the detail.

is D a *preimage* of D. No counter example is known for this conjecture, and there are many mathematical results about this conjecture. For example trees, regular graphs, and disconnected graphs are reconstructible (i.e. the conjecture is true for these classes) [28]. Almost all graphs are reconstructible from three well-chosen graphs in its deck [5]. Rimscha showed that many subclasses of perfect graphs, for example interval graphs and permutation graphs, including perfect graphs themselves are recognizable (i.e. looking at the deck of G one can decide whether or not G belongs to perfect graphs) [45]. Rimscha also showed some of subclasses including unit interval graphs are reconstructible. There are many good surveys about this conjecture. See for example [6, 24].

Besides these mathematical results, there are some algorithmic results. We enumerate the algorithmic problems that we address in this paper.

- Given a graph G and a multi-set D of graphs, check whether D is a deck of G (*deck checking*).
- Given a multi-set D of graphs, determine whether there is a graph whose deck is D (*legitimate deck*).
- Given a multi-set D of graphs, construct a graph whose deck is D (*preimage construction*).
- Given a multi-set D of graphs, compute the number of (pairwise nonisomorphic) graphs whose decks are D (*preimage counting*).

Kratsch and Hemaspaandra showed that these problems are solvable in polynomial time for graphs of bounded degree, partial k-trees for any fixed k, and graphs of bounded genus, in particular for planar graphs [33]. In the same paper they proved many graph isomorphism(GI)-related complexity results. Hemaspaandra et al. extended the results [25].

In this paper, we treat with some graph classes that isomorphism problem can be solved in polynomial time. Concretely, these graph classes are interval graphs, permutation graphs, and distance-hereditary graphs. The graph isomorphism problem can be solved in polynomial time on these graph classes, so developing a polynomial time algorithms for deck checking for these graph classes is easy. However, the number of the ways of adding one vertex simply is exponential so the number of preimage candidates of input graphs is exponential. Thus the key is how to decrease the candidates.

In this paper, we propose polynomial time reconstruction algorithms for interval graphs, permutation graphs, and distance-hereditary graphs.

1.3 Framework

We first prepare to propose our algorithms in Chapter 2. First, we state terminologies of graphs in Section 2.1. Then, we define some graph classes and introduce some properties of these graph classes. In Section 2.7, we explain a Dyck path and a Motzkin path for random generation of proper interval graphs and bipartite permutation graphs.

We propose random generation and enumeration algorithms for proper interval graphs and bipartite permutation graphs in Chapter 3. We show the random generation and enumeration algorithms for proper interval graphs in Section 3.1 and 3.2, respectively, and for bipartite permutation graphs in Section 3.3 and 3.4, respectively. We use counting argument for random generation, so we count proper interval graphs in Section 3.1, and bipartite permutation graphs in Section 3.3.

We present reconstruction algorithms for interval graphs, permutation graphs, and distance-hereditary graphs in Chapter 4. In each section of Chapter 4, we first propose a deck checking algorithm. Then we discuss that a preimage of input graphs is not the same graph class of the input graphs. Finally, we present the reconstruction algorithms when a preimage of input graphs is same the graph class of the input graphs.

In Chapter 5, we propose a simple construction $\mathcal{MPQ}$-tree algorithm. $\mathcal{MPQ}$-trees are informative data structure for interval graphs. By using $\mathcal{MPQ}$-trees, we can solve the isomorphism problem for interval graphs. Additionally, we use the $\mathcal{MPQ}$-tree for the reconstruction algorithm of interval graphs, implicitly. However, construction algorithm of $\mathcal{MPQ}$-tree in [32] has several templates, so the implementation of the algorithm is not easy. Our algorithm is simple and efficient.

Finally we make some remarks in Chapter 6.

Chapter 2

Preliminaries

2.1 Basic Graph Notations

A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq V^2$; that is, the elements of E are 2-element subsets of V [15]. The elements of V are the *vertices* (or *nodes*) of the graph G and the elements of E are its *edges*.

Let $G = (V, E)$ be a graph, and edge $e \in E$ be $e = \{u, v\}$. Two vertices u, v are *incident* with an edge e, and u is *adjacent* to v. The *neighbor set* of v is the set $N(v) = \{u \in V \mid \{u, v\} \in E\}$. The *closed neighbor set* of v is the set $N(v) \cup \{v\}$, and we denote by $N[v]$. Vertices u and v are called *weak twins* if $N(u) = N(v)$, and *strong twins* if $N[u] = N[v]$.

The *degree* of a vertex v is $|N(v)|$ denoted by $\deg(v)$. A vertex v is called a *pendant* if v is a degree one vertex. A vertex of degree 0 is *isolated*. The sum of degrees of all vertices in graph G is denoted by $\deg(G)$. Note that $\deg(G)$ is equal to twice the number of edges in G.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if and only if there is a one-to-one mapping $\phi : V \to V'$ which satisfies $\{u, v\} \in E$ if and only if $\{\phi(u), \phi(v)\} \in E'$ for every pair of vertices u and v. When G is isomorphic to G', we denote it by $G \sim G'$. The mapping ϕ is called *isomorphism* from G to G'. Given graphs G and G', *graph isomorphism problem (GI)* is the problem to determine whether or not $G \sim G'$.

A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. A subgraph $G' = (V', E')$ is an *induced subgraph* of $G = (V, E)$ if $E' = \{\{u, v\} \mid u, v \in V' \text{ and } \{u, v\} \in E\}$. We say that G' is induced by V' and write $G[V']$ for G'. For a vertex $v \in V$, we denote by $G - v$ the graph obtained by removing v and its incident edges from G. Let S be a set, and $s \in S$. We denote $S \setminus \{s\}$ by $S - s$.

A graph $G' = (V', E')$ is *complement* of $G = (V, E)$ if $V' = V$ and $E' = \{\{u, v\} \mid u, v \in V, u \neq v, \text{ and } \{u, v\} \notin E\}$, and we denote complement of G by $\overline{G} = (V, \overline{E})$.

For a given graph $G = (V, E)$, a sequence of distinct vertices $v_0, v_1, \cdots, v_\ell$ is a *path*, denoted by $(v_0, v_1, \cdots, v_\ell)$, if $\{v_j, v_{j+1}\} \in E$ for each $0 \leq j \leq \ell - 1$. The *length* of a path is the number of edges on the path. A sequence $v_0, v_1, \cdots, v_\ell, v_0$ is a *cycle* if $v_0, v_1, \cdots, v_\ell$ is a path and $\{v_\ell, v_0\} \in E$. The *length* of a cycle is the number of edges on the cycle.

A graph $G = (V, E)$ is *connected* if for every pair of vertices $u, v \in V$, there is a path from u to v. A graph G is *disconnected* if G is not connected. A maximal connected subgraph of G is called a *component* of G.

A graph G is a *tree* if G is connected and G contains no cycle. We consider one vertex of a tree as special, such a vertex is called the *root* of the tree. A tree with a fixed root is a *rooted tree*. In a rooted tree, *ancestors* of v are vertices in the path from v to root. If u is ancestor of

v, and u and v are adjacent, we call that u is *parent* of v and v is *child* of u. A tree has a vertex which has no child, such a vertex is called a *leaf*. An *ordered tree* is a rooted tree for which an ordering is specified for the children of each vertex.

A graph $G = (V, E)$ is *complete* if all the vertices of G are pairwise adjacent. A complete graph on n vertices is a K_n. A subset $V' \subseteq V$ is a *clique* in G if $G[V']$ is complete. A vertex $v \in V$ is *simplicial* in G if $N(v)$ is a clique in G. A subset $V' \subseteq V$ is an *independent set* in G if no two vertices in V' are adjacent.

A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two disjoint sets X and Y such that for every $x_1, x_2 \in X$, $\{x_1, x_2\} \notin E$ and for every $y_1, y_2 \in Y$, $\{y_1, y_2\} \notin E$. We denote a bipartite graph by $G = (X, Y, E)$.

A vertex v is *universal* in graph G if v connects to every vertex in G. We denote by $\tilde{G}$ the graph obtained by adding one universal vertex to the graph G.Thus, $\tilde{G}$ is always connected.

Given two graphs G_1 and G_2, we define the disjoint union $G_1 \dot{\cup} G_2$ of G_1 and G_2 as $(V_1 \dot{\cup} V_2, E_1 \dot{\cup} E_2)$ such that (V_1, E_1) is isomorphic to G_1, and (V_2, E_2) is isomorphic to G_2, where $\dot{\cup}$ means the disjoint union.

2.2 Interval Graphs

This section deals with interval graphs. First, we define interval graphs and their properties. Next, we explain compact interval representation. We use compact interval representation for reconstruction algorithm of interval graphs in Section 4.1. However, we use the $\mathcal{MPQ}$-tree instead of compact interval representation for the reconstruction algorithm, implicitly. Construction algorithm of $\mathcal{MPQ}$-tree in [32] has several templates, so the implementation of the algorithm is not easy. We propose a simple algorithm that constructs $\mathcal{MPQ}$-tree from a interval representation in Chapter 5.

2.2.1 Definition of Interval Graphs

A graph (V, E) with $V = \{v_1, v_2, \cdots, v_n\}$ is an *interval graph* if there is a set of intervals $\mathcal{I} = \{I_{v_1}, I_{v_2}, \cdots, I_{v_n}\}$, such that $\{v_i, v_j\} \in E$ if and only if $I_{v_i} \cap I_{v_j} \neq \emptyset$ for each i and j with $1 \leq i, j \leq n$. We call the set $\mathcal{I}$ of intervals *interval representation* of the graph. We show an example of a interval graph and interval representation in Figure 2.1. For each interval I, we denote by $L(I)$ and $R(I)$ the left and right endpoints of the interval, respectively (hence we have $L(I) \leq R(I)$). Without loss of generality, we can assume that every interval is closed, so we denote an interval $I = [L(I), R(I)]$. For two intervals I and J, we write $I \prec J$ if $L(I) \leq L(J)$ and $R(I) \leq R(J)$. Interval I and interval J *overlap* if $L(I) < L(J) \leq R(I) < R(J)$ or $L(J) < L(I) \leq R(J) < R(I)$. In the Figure 2.1, I_a and I_c, and I_d and I_e overlap.

We introduce famous properties for interval graphs below.

Proposition 2.1. *Any induced subgraph of an interval graph is an interval graph.*

Lemma 2.2 (Fulkerson and Gross [16])**.** *An interval graph on n vertices has at most n maximal cliques.*

Theorem 2.3 (Gilmore and Hoffman [20])**.** *Graph G is an interval graph if and only if the maximal cliques of G can be linearly ordered such that, for every vertex x of G, the maximal cliques containing x occur consecutively.*

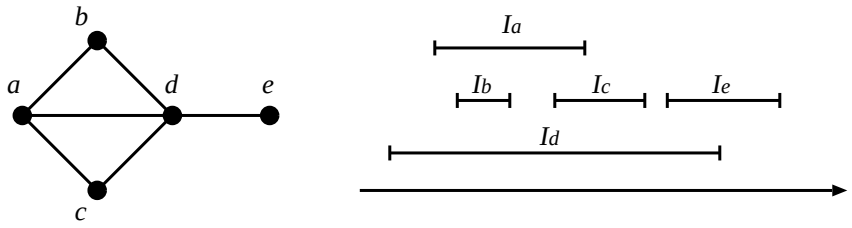


Figure 2.1: An interval graph and its interval representation.

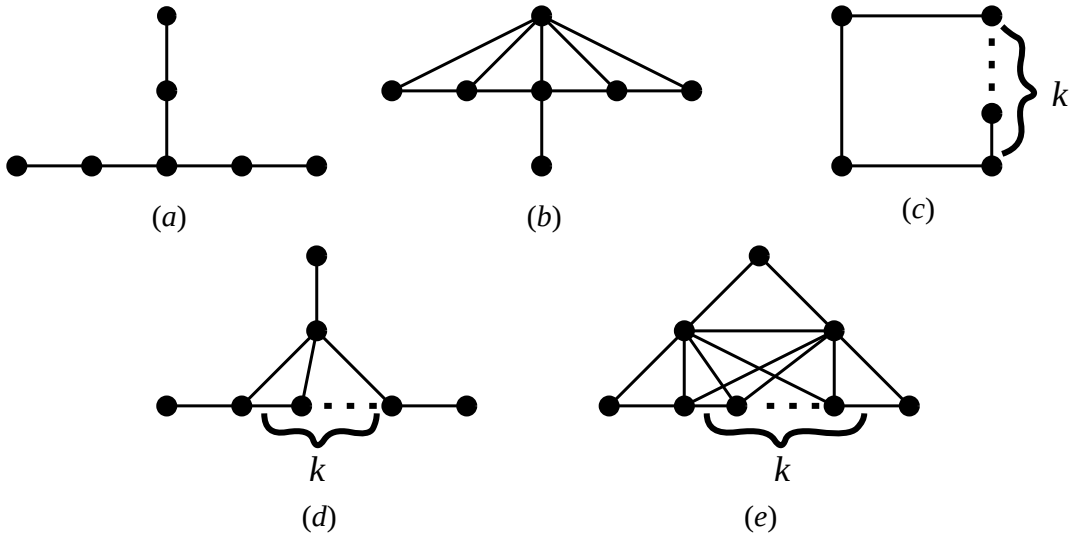


Figure 2.2: The forbidden graphs of interval graphs. The part described k contains k vertices ($k \geq 0$). Thus (c) is a chordless cycle of more than three vertices, (d) has more than five vertices, and (e) has more than five vertices.

Theorem 2.4 (Lekkerkerker and Boland [34])**.** *Graph G is an interval graph if and only if G has no graph described in Figure 2.2 as an induced subgraph.*

2.2.2 Compact Interval Representation

In this section, we define a compact interval representation and state its basic properties.

Definition 2.5 ([52])**.** *An interval representation $\mathcal{I}$ of an interval graph $G = (V, E)$ is* compact *if and only if*

- *coordinates of endpoints of intervals in $\mathcal{I}$ are finite non-negative integers (We denote by K the largest coordinates of endpoints for convenience. We sometimes call K the length of $\mathcal{I}$),*
- *there exists at least one endpoint whose coordinate is k for every integer $k \in [0, K]$, and*
- *interval multi-set $\mathcal{I}_k = \{I \in \mathcal{I} \mid k \in I\}$ differs from $\mathcal{I}_l = \{I \in \mathcal{I} \mid l \in I\}$, and they do not include each other, for every distinct integers $k, l \in [0, K]$.*

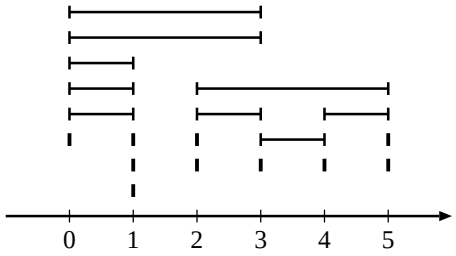


Figure 2.3: A compact interval representation of an interval graph.

We show an example of a compact interval representation of an interval graph in Figure 2.3. Note that there may still be many compact interval representations of an interval graph. However compact interval representations have some good properties.

Lemma 2.6. *Let $\mathcal{I}$ and $\mathcal{J}$ be compact interval representations of an interval graph $G = (V, E)$, and let K_1 be the length of $\mathcal{I}$, and let K_2 be the length of $\mathcal{J}$. Then the following holds.*

$$\begin{aligned}&\{\{I \in \mathcal{I} \mid 0 \in I\}, \{I \in \mathcal{I} \mid 1 \in I\}, \ldots, \{I \in \mathcal{I} \mid K_1 \in I\}\}\\ =\ &\{\{I \in \mathcal{J} \mid 0 \in I\}, \{I \in \mathcal{J} \mid 1 \in I\}, \ldots, \{I \in \mathcal{J} \mid K_2 \in I\}\}\end{aligned}$$

Proof. We denote by $\overline{\mathcal{I}}$ the set of multi-set of intervals $\{\{I \in \mathcal{I} \mid 0 \in I\}, \{I \in \mathcal{I} \mid 1 \in I\}, \ldots, \{I \in \mathcal{I} \mid K_1 \in I\}\}$, and we denote by $\overline{\mathcal{J}}$ the set of multi-set of intervals $\{\{I \in \mathcal{J} \mid 0 \in I\}, \{I \in \mathcal{J} \mid 1 \in I\}, \ldots, \{I \in \mathcal{J} \mid K_2 \in I\}\}$. The vertices represented by the multi-set of intervals $\mathcal{I}_i = \{I \in \mathcal{I} \mid i \in I\}$ correspond to a clique in G. Assume that $\mathcal{I}_i$ never appears in $\overline{\mathcal{J}}$ for some i. Since $\mathcal{I}_i$ represents a clique C, there must be a set of intervals representing a clique C' containing C in $\mathcal{J}$ (otherwise, clique C cannot be represented in $\mathcal{J}$). Then for the same reason, $\overline{\mathcal{I}}$ must contain a set of intervals representing a clique containing C'. This contradicts the compactness of $\mathcal{I}$. □

From the proof of Lemma 2.6, the following lemmas are straightforward.

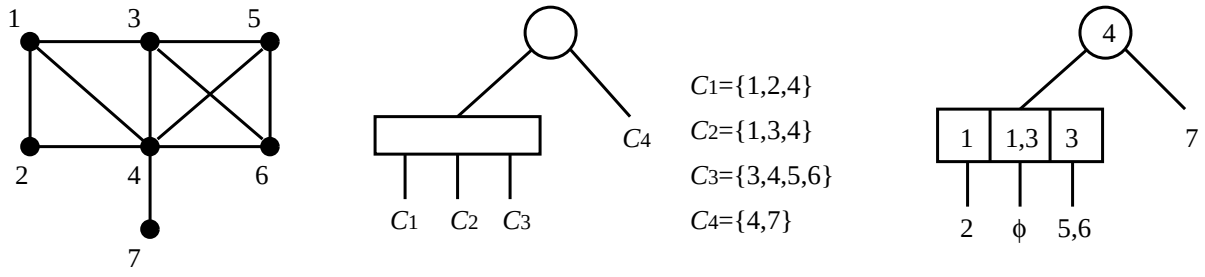
Lemma 2.7. *Let $\mathcal{I}$ be a compact interval representation of an interval graph $G = (V, E)$, and let K be the length of $\mathcal{I}$. Then $\{I \in \mathcal{I} \mid i \in I\}$ for each $i \in \{0, \ldots, K\}$ corresponds to each maximal clique of G.*

Lemma 2.8. *The length of a compact interval representation of an n-vertex interval graph is at most n.*

Lemma 2.9. *All the compact interval representations of an interval graph have the same length. Intervals in different compact interval representations corresponding to an identical vertex have the same length.*

From Lemma 2.9, lengths of intervals corresponding to a vertex that corresponds to an interval of length zero in some compact interval representation are always (i.e. in any compact interval representation) zero.

Lemma 2.10. *Vertices corresponding to intervals of length zero in a compact interval representation are simplicial.*



(*a*) (*b*) (*c*)

Figure 2.4: (*a*) An interval graph G. (*b*) A $\mathcal{PQ}$-tree obtained from G with maximal cliques C_i ($i = 1, \ldots, 4$). (*c*) A $\mathcal{MPQ}$-tree of G.

2.2.3 $\mathcal{PQ}$-tree and $\mathcal{MPQ}$-tree

The $\mathcal{PQ}$-tree was introduced by Booth and Lueker [8]. We can use it for recognizing interval graphs. A *$\mathcal{PQ}$-tree* is a rooted tree T with two types of internal nodes, $\mathcal{P}$- and Q-nodes. The leaves of T are labeled one-to-one with the maximal cliques of the interval graph G. The *frontier* of a $\mathcal{PQ}$-tree T is the permutation of the maximal cliques obtained by the ordering of the leaves of T from left to right. The definition that $\mathcal{PQ}$-tree T corresponds to an interval graph G is given as follows [8].

Definition 2.11. *A $\mathcal{PQ}$-tree T corresponds to an interval graph G, if and only if, for every $\mathcal{PQ}$-tree T' obtained from T by applying the following rules (1) and (2) a finite number of times, there is a consecutive arrangement of the maximal cliques on G that represents for the frontier of T':*

(1) Arbitrarily permute the successor nodes of a $\mathcal{P}$-node, or

(2) reverse the order of the successor nodes of a Q-node.

See Figure 2.4(*b*); we designate a $\mathcal{P}$-node by a circle and a Q-node by a wide rectangle. Booth and Lueker developed a linear time algorithm that either constructs a $\mathcal{PQ}$-tree for G, or states that G is not an interval graph.

The $\mathcal{PQ}$-tree with appropriate label defined by the maximal cliques is *canonical*; that is, given interval graphs G_1 and G_2 are isomorphic if and only if corresponding labeled $\mathcal{PQ}$-trees T_1 and T_2 are isomorphic. Since we can determine if two labeled $\mathcal{PQ}$-trees T_1 and T_2 are isomorphic in linear time, the graph isomorphism problem of interval graphs can be solved in linear time (see [8, 36] for further details).

The $\mathcal{MPQ}$-tree model, which stands for *modified $\mathcal{PQ}$-tree*, is developed by Korte and Möhring to simplify the algorithm for the $\mathcal{PQ}$-tree [32]. The $\mathcal{MPQ}$-tree T^* assigns sets of vertices (or intervals from the view of interval representation) to the nodes of a $\mathcal{PQ}$-tree T representing an interval graph $G = (V, E)$. It is possible that no vertices is assigned to some nodes. A $\mathcal{P}$-node is assigned only one set, while a Q-node has a set for each of its children (ordered from left to right according to the ordering of the children). For a $\mathcal{P}$-node P, this set consists of those vertices of G contained in all maximal cliques represented by the subtree of P in T, but in no other cliques.

For a Q-node Q, the definition is more involved. Let $Q_1, \cdots, Q_m$ be the set of the children (in consecutive order) of Q, and let T_i be the subtree of T with root Q_i (note that $m \geq 3$). We

then assign a set S_i, called *section*, to Q for each Q_i. Section S_i contains all vertices that are contained in all maximal cliques of T_i and some other T_j, but not in any clique belonging to some other subtree of T that is not below Q (see Figure 2.4(c)). The key property of $\mathcal{MPQ}$-trees is summarized as follows:

Theorem 2.12 ([32, Theorem 2.1])**.** *Let T be a $\mathcal{PQ}$-tree for an interval graph $G = (V, E)$ and let T^* be the associated $\mathcal{MPQ}$-tree. Then we have the following:*
(a) T^* can be obtained from T in $O(|V| + |E|)$ time and represents G in $O(|V|)$ space.
(b) Each maximal clique of G corresponds to a path in T^* from the root to a leaf, where each vertex $v \in V$ is as close as possible to the root.
(c) In T^*, each vertex v appears in either one leaf, one $\mathcal{P}$-node, or consecutive sections $S_i, S_{i+1}, \cdots, S_{i+j}$ for some $\mathcal{Q}$-node with $j > 0$.

Property (b) is the essential property of $\mathcal{MPQ}$-trees. For example, the root of T^* contains all vertices belonging to all maximal cliques, and the leaves contain the simplicial vertices of G. In [32], they did not state Theorem 2.12(c) explicitly. Theorem 2.12(c) is immediately obtained from the fact that the maximal cliques containing a fixed vertex occur consecutively in T(c.f. Theorem 2.3 and Lemma 2.7). Korte and Möhring state the following lemma in [32] as the essential properties of the $\mathcal{MPQ}$-tree:

Lemma 2.13 ([32, Lemma 2.2])**.** *Let N be a $\mathcal{Q}$-node. Let $S_1, \ldots, S_m$ (in this order) be the sections of N, and let V_i denote the set of vertices occurring below S_i in the $\mathcal{MPQ}$-tree T with $1 \leq i \leq m$. Then we have the following:*
(a) $S_{i-1} \cap S_i \neq \emptyset$ for $i = 2, \ldots, m$.
(b) $S_1 \subseteq S_2$ and $S_{m-1} \supseteq S_m$.
(c) $V_1 \neq \emptyset$ and $V_m \neq \emptyset$.
(d) $S_i \cap S_{i+1} \setminus S_1 \neq \emptyset$ and $S_{i-1} \cap S_i \setminus S_m \neq \emptyset$ for $i = 2, \ldots, m-1$.

However, under this conditions, the $\mathcal{MPQ}$-tree is not uniquely determined. There exist two or more nonisomorphic $\mathcal{MPQ}$-trees for an interval graph. The reason is that two consecutive sections S_i and S_{i+1} can be equal. In the case, we swap them and obtain the different $\mathcal{MPQ}$-trees. We note that this fact does not imply that the results in [32] is wrong. The uniqueness of the $\mathcal{MPQ}$-tree is not required in their paper, and they did not mind it [37]. However, their algorithms for the construction of an $\mathcal{MPQ}$-tree surely produce the unique $\mathcal{MPQ}$-tree, which satisfies the following additional condition (see Appendix A for further details):

(e) $S_{i-1} \neq S_i$ for $i = 2, \ldots, m-1$.

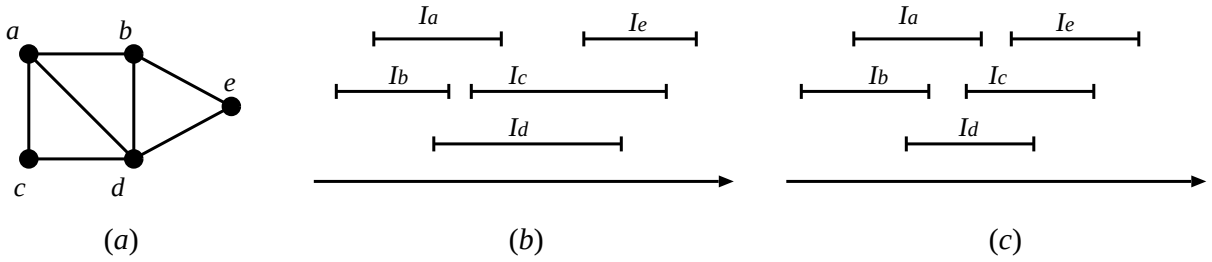
The condition (e) implies that we can rewrite the condition (b) as follows:

(b) $S_1 \subset S_2$ and $S_{m-1} \supset S_m$.

Hereafter, we will use the conditions from (a) to (e) as the basic properties of an $\mathcal{MPQ}$-tree.

2.3 Proper Interval Graphs

In this section, we introduce proper interval graphs which form a subclass of interval graphs. Proper interval graphs correspond to strings, and we use string representation of a proper interval graph for random generation (Section 3.1) and enumeration algorithms (Section 3.2). We will explain string representation of proper interval graphs.



(*a*) (*b*) (*c*)

Figure 2.5: (*a*) A proper interval graph G. (*b*) A proper interval representation of G. (*c*) A unit interval representation of G.

2.3.1 Definition of Proper Interval Graphs

An interval representation is *proper* if no two distinct intervals I and J exist such that I properly contains J or vice versa. That is, either $I \prec J$ or $J \prec I$ holds for every pair of intervals I and J. An interval graph is *proper* if it has a proper interval representation (see Figure 2.5(*a*) and (*b*)). If an interval graph G has an interval representation $\mathcal{I}$ such that every interval in $\mathcal{I}$ has the same length, G is said to be a *unit interval graph*. Such interval representation is called a *unit interval representation* (see Figure 2.5(*c*)). It is well known that proper interval graphs coincide with unit interval graphs [46]. That is, given a proper interval representation, we can transform it to a unit interval representation. A simple constructive way of the transformation can be found in [4]. With perturbations if necessary, we can assume without loss of generality that $L(I) \neq L(J)$ (and hence $R(I) \neq R(J)$), and $R(I) \neq L(J)$ for any two distinct intervals I and J in a unit interval representation $\mathcal{I}$. In a unit interval representation, we assume that the intervals are sorted by left endpoint values.

2.3.2 String Representation

We denote an alphabet {'[', ']'} by Σ throughout the paper. We encode a unit interval representation $\mathcal{I}$ of a unit interval graph G by a string $s(\mathcal{I})$ in Σ^* as follows; we sweep the interval representation from left to right, and encode $L(I)$ by '[' and encode $R(I)$ by ']' for each $I \in \mathcal{I}$ (e.g., $s(\mathcal{I})$ = [[[][][]]] in Figure 2.5(*c*)). We call the encoded string a *string representation* of G. We say that a string x in Σ^* is *balanced* if the number of '['s in x is equal to that of ']'s. Clearly $s(\mathcal{I})$ is a balanced string of $2n$ letters. Using the construction in [4], $s(\mathcal{I})$ can be constructed from a proper interval representation $\mathcal{I}$ in $O(n)$ time and vice versa since the ith '[' and the ith ']' give the left and right endpoints of the ith interval, respectively.

We denote $\overline{\text{'['}} = \text{']'}$ and $\overline{\text{']'}} = \text{'['}$ respectively. For two strings $x = x_1x_2\cdots x_n$ and $y = y_1y_2\cdots y_m$ in Σ^*, we say that x is *smaller* than y if (1) $n < m$, or (2) $n = m$ and there exists an index $i \in \{1, \ldots, n\}$ such that $x_{i'} = y_{i'}$ for all $i' < i$ and $x_i =$ '[' and $y_i =$ ']'. If x is smaller than y, we denote $x < y$. We note that the balanced string $x = [[\cdots[]]\cdots]$ is the smallest among those of the same length. For a string $x = x_1x_2\cdots x_n$ we define the *reverse* $\overline{x}$ of x by $\overline{x} = \overline{x}_n\overline{x}_{n-1}\cdots\overline{x}_1$. A string x is *symmetric* if $x = \overline{x}$. Here we have the following lemma:

Lemma 2.14 (See, e.g., [13, Corollary 2.5])**.** *Let G be a connected proper interval graph, and $\mathcal{I}$ and $\mathcal{I}'$ be any two unit interval representations of G. Then either $s(\mathcal{I}) = s(\mathcal{I}')$ or $s(\mathcal{I}) = \overline{s(\mathcal{I}')}$ holds. That is, the unit interval representation and hence the string representation of a proper interval graph is determined uniquely up to isomorphism.*

Note that G is supposed to be connected in Lemma 2.14. If G is disconnected, we can obtain several distinct string representations by arranging the connected components.

A connected proper interval graph G is said to be *symmetric* if its string representation is symmetric.

It is easier for our purpose (counting, random generation, and enumeration of unlabeled proper interval graphs) to deal with the encoded strings in Σ^* than to use interval representations. Given an interval representation $\mathcal{I}$ of a proper interval graph G, the smaller of the two string representations $s(\mathcal{I})$ and $\overline{s(\mathcal{I})}$ is called *canonical*. If $s(\mathcal{I})$ is symmetric, $s(\mathcal{I})$ is the canonical string representation. Hereafter we sometimes identify a connected proper interval graph G with its canonical string representation.

For a string $x = x_1 x_2 \cdots x_n \in \Sigma^n$ of length n, we define the *height* $h_x(i)$ ($i \in \{0, \ldots, n\}$) as follows;

$$h_x(i) = \begin{cases} 0 & \text{if } i = 0, \\ h_x(i-1) + 1 & \text{if } x_i = \text{`['}, \\ h_x(i-1) - 1 & \text{if } x_i = \text{`]'}. \end{cases}$$

We say that a string x is *nonnegative* if $\min_i\{h_x(i)\}$ is equal to 0 (we do not have $\min_i\{h_x(i)\} > 0$ since $h_x(0) = 0$). The following observation is immediate:

Observation 2.15. *Let $x = x_1 x_2 \cdots x_{2n}$ be a string in Σ^{2n}. (1) x is a string representation of a (not necessarily connected) proper interval graph if and only if x is balanced and nonnegative.*

(2) x is a string representation of a connected proper interval graph if and only if $x_1 =$ '[' and $x_{2n} =$ ']', and the string $x_2 \cdots x_{2n-1}$ is balanced and nonnegative.

A balanced nonnegative string of length $2n$ corresponds to a well-known notion called Dyck path. We will explain Dyck path in section 2.7.

2.4 Permutation Graphs

In this section, first, we define permutation graphs and explain its basic properties. Next, we explain modular decompositions. Modular decomposition deeply relates to permutation. For example, using the modular decomposition, we can solve the recognition and isomorphism problems of permutation graphs. In Section 4.2, we will propose a reconstruction algorithm for permutation by using modular decomposition.

2.4.1 Definition of Permutation Graphs

A graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$ is said to be a *permutation graph* if and only if there is a permutation π over V such that $\{v_i, v_j\} \in E$ if and only if $(i - j)(\pi(v_i) - \pi(v_j)) < 0$. Intuitively, each vertex v_i in a permutation graph corresponds to a line ℓ_i joining two points on two parallel lines L_1 and L_2. Then two vertices v_i and v_j are adjacent if and only if the corresponding lines ℓ_i and ℓ_j intersect. The ordering of vertices gives the ordering of the points on L_1, and the ordering by permutation π over V gives the ordering of the points on L_2. We call the intersection model a *line representation* of the permutation graph. For example, Figure 2.6 is a permutation and its line representation, and a permutation $\pi = (3, 4, 1, 6, 5, 2)$ of Figure 2.6. Precisely, a line representation $\mathcal{L}$ of a permutation graph $G = (V, E)$ with $|V| = n$ consists of two parallel lines L_1 and L_2, and n points are at regular intervals on L_1 and L_2, respectively. We

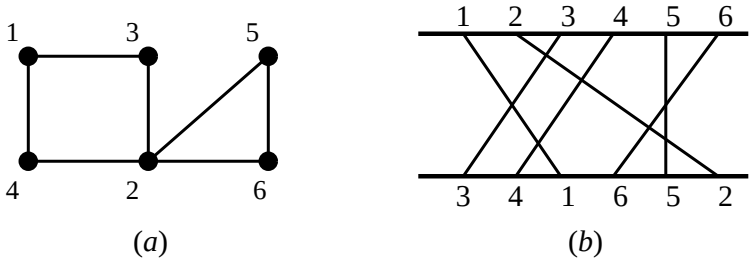


Figure 2.6: (*a*) A permutation graph. (*b*) its line representation.

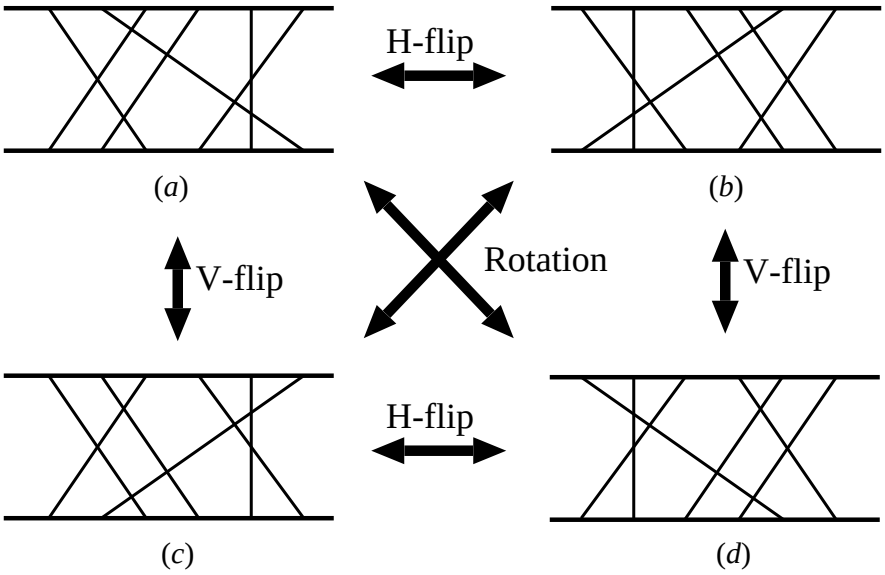


Figure 2.7: (*a*) A line representation $\mathcal{L}$. (*b*)$\mathcal{L}^H$. (*c*)$\mathcal{L}^V$. (*d*)$\mathcal{L}^R$.

suppose that these points are numbered from 1 to n on the lines from left to right. Then each vertex $v_i \in V$ corresponds to a pair of points $(i, \pi(i))$, which means the point i on L_1 and the point $\pi(i)$ on L_2 are joined by the corresponding line ℓ_i. For two line representations $\mathcal{L}$ and $\mathcal{L}'$, suppose $\mathcal{L}$ contains (i, j) if and only if $\mathcal{L}'$ contains (i, j). Then we call them *isomorphic* and denote by $\mathcal{L} = \mathcal{L}'$.

Let $\mathcal{L} = (L_1, L_2)$ be a line representation of a permutation graph $G = (V, E)$. For a connected permutation graph G, we can construct essentially equivalent representations by flipping $\mathcal{L}$. On a *horizontal flip* $\mathcal{L}^H$ (H-flip for short) of $\mathcal{L}$, each line (i, j) on $\mathcal{L}$ is mapped to the line $(n - i + 1, n - j + 1)$. On a *vertical flip* $\mathcal{L}^V$ (V-flip for short) of $\mathcal{L}$, each line (i, j) on $\mathcal{L}$ is mapped to the line (j, i). For a line representation $\mathcal{L}$, it is not difficult to see that $(\mathcal{L}^H)^V = (\mathcal{L}^V)^H$ gives us a *rotation* of $\mathcal{L}$. Hence we denote the line representation by $\mathcal{L}^R$ after this operation (see Figure 2.7).

We introduce famous properties for permutation graphs below.

Proposition 2.16. *An induced subgraph of a permutation graph is a permutation graph.*

Theorem 2.17 (Pnueli, Lempel, and Even [44])**.** *Graph G is a permutation graph if and only if G is a comparability graph and a co-comparability graph.*

Lemma 2.18 (Gllai [17])**.** *Graph G is a comparability graph if and only if G is* (C_{k+6}, T_2, X_2, X_3, X_{30}, X_{31}, X_{32}, X_{33}, X_{34}, X_{36}, XF_1^{2k+3}, XF_2^{k+1}, XF_3^k, XF_4^k, XF_5^{2k+3}, XF_6^{2k+2})*-free; that is, G has no*

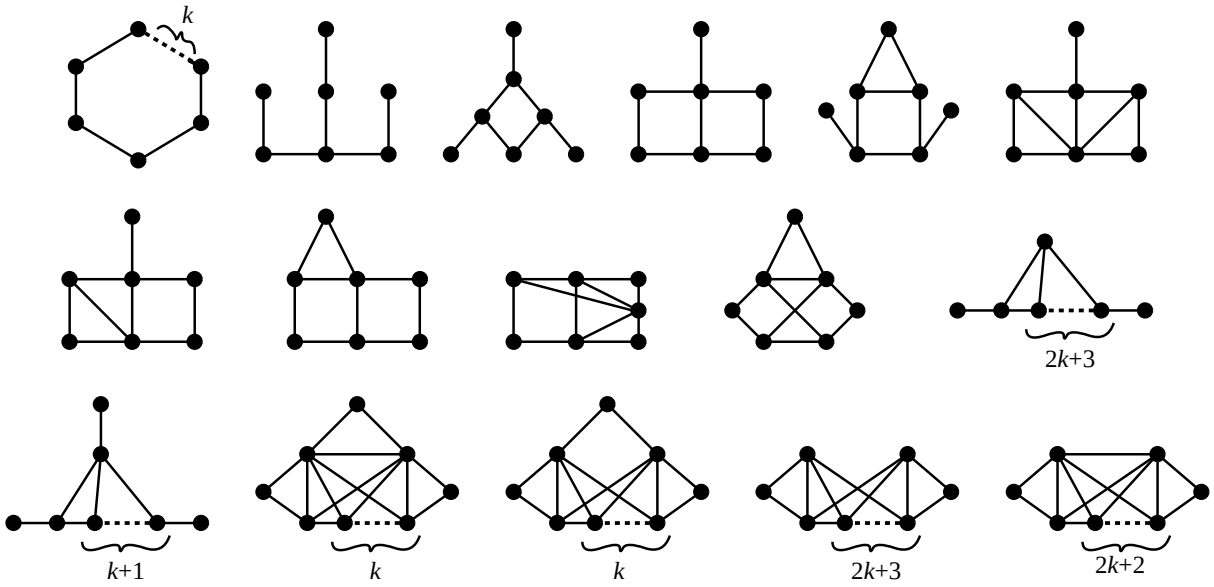


Figure 2.8: Forbidden graphs of a comparability graph ($k \geq 0$).

graph described in Figure 2.8 as an induced subgraph.

From Theorem 2.17 and Lemma 2.18, we can obtain next theorem, immediately.

Theorem 2.19. *Graph G is a permutation graph if and only if G is T_2, X_2, X_3, X_{30}, X_{31}, X_{32}, X_{33}, X_{34}, X_{36}, XF_1^{2k+3}, XF_2^{k+1}, XF_3^k, XF_4^k, XF_5^{2k+3}, XF_6^{2k+2})-free and co – (C_{k+6}, T_2, X_2, X_3, X_{30}, X_{31}, X_{32}, X_{33}, X_{34}, X_{36}, XF_1^{2k+3}, XF_2^{k+1}, XF_3^k, XF_4^k, XF_5^{2k+3}, XF_6^{2k+2})-free; that is, G has no graph described in Figure 2.8 and the complements of them as an induced subgraph.*

2.4.2 Modular Decomposition

Modular decomposition is a strong tool for developing fast algorithms in many areas. Here we summarize it. For the detail see for example [9, 48].

Let $G = (V, E)$ be a graph. The subset $M \subset V$ is a *module* in G, if for all vertices $u, v \in M$ and $w \in V \setminus M$, $\{u, w\} \in E$ if and only if $\{v, w\} \in E$. A module M in G is *trivial* if $M = V$, $M = \emptyset$, or $|M| = 1$. G is called a *prime* (with respect to modular decomposition) if G contains only trivial modules. A module M is *strong* if it does not overlap any other modules in G, i.e.

$$M \cap M' = \emptyset,\ M \subset M',\ \text{or } M' \subset M$$

holds for any other module M' in G. We call a module that contains at least two vertices a *multi-vertex module*.

A *modular decomposition tree* of a graph G is a rooted tree whose each node corresponds to each strong module of G such that for any two nodes N_1 and N_2 which correspond to modules M_1 and M_2 respectively, N_1 is an ancestor of N_2 if and only if M_1 contains M_2. We sometimes say that strong module M_1 is the parent of strong module M_2, and M_2 is a child of M_1, if the node corresponding to M_1 is the parent of the node corresponding to M_2 in the modular decomposition tree (see Figure 2.9).

A strong multi-vertex module M in graph G whose child modules are disconnected to each other in $G[M]$ is a *parallel module*. A strong multi-vertex module M in graph G whose child

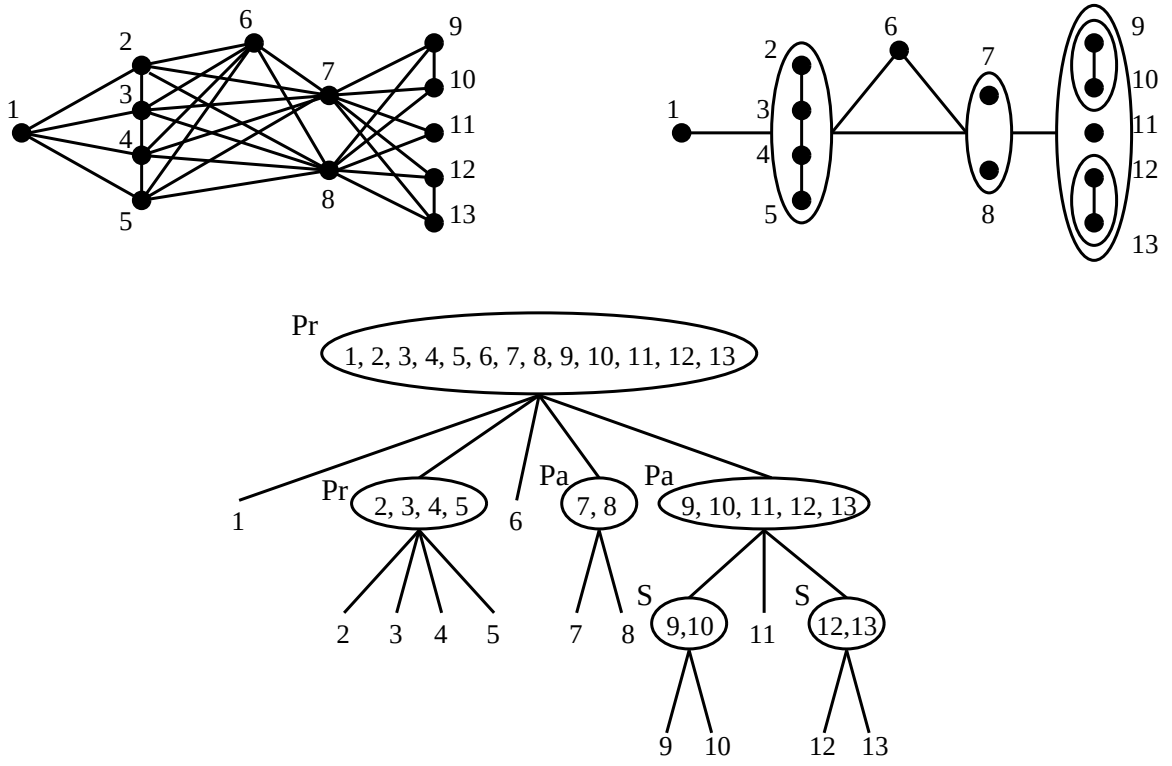


Figure 2.9: Graph and its modular decomposition

modules are disconnect to each other in $\overline{G[M]}$ is a *series module*. Let M' be a strong multi-vertex module. If M' is not a parallel module, and M' is not a series module, then M' is called a *prime module*. A graph induced by a prime module is connected in both G and $\overline{G}$ [48].

We say a strong multi-vertex module M is *minimal* if every child of M is a module of one vertex. Note that every graph of the size more than one has at least one minimal strong multi-vertex module. We introduce a basic lemma.

Lemma 2.20 (Gallai [17])**.** *A minimal strong multi-vertex module that is a prime module induces a prime.*

Lemma 2.21. *A minimal strong multi-vertex module is either a clique, independent set, or prime.*

Let $G = (V, E)$ be a prime. We say that G is *critical* if $G - v$ is not a prime for any $v \in V$. We define graph H_n. H_n is a bipartite graph (X, Y, E) such that $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$, and $\{x_i, y_j\} \in E$ if and only if $i \leq j$. See Figure 2.10.

Theorem 2.22 (Schmerl and Trotter [47])**.** *Given prime graph $G = (V, E)$ with $|V| \geq 2$, G is critical if and only if G is isomorphic to H_n or to $\overline{H_n}$.*

Hence the number of vertices in a critical graph is always even.

2.5 Bipartite Permutation Graphs

When a permutation graph is bipartite, it is said to be a *bipartite permutation graph* (see Figure 2.11). Then the following lemma holds:

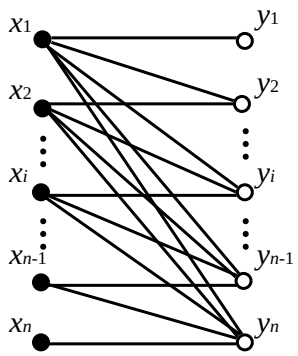


Figure 2.10: Graph H_n.

Lemma 2.23. *Let $G = (X, Y, E)$ be a connected bipartite permutation graph with $|X| > 0$ and $|Y| > 0$ and $\mathcal{L} = (L_1, L_2)$ its line representation. Without loss of generality, we assume that $v_1 \in X$ corresponds to $(1, i)$ for some i with $1 \leq i \leq n$. Then X and Y satisfy that $X = \{v_i \mid v_i$ corresponds to (i, j) with $i < j\}$ and $Y = \{v_i \mid v_i$ corresponds to (i, j) with $i > j\}$.*

Proof. If $v_1 \in X$ corresponds to $(1, 1)$, G is disconnected. Hence $v_1 = (1, i)$ with $i > 1$ and there is a vertex $v_{i'}$ corresponding to $(i', 1)$ with $i' > 1$. Clearly, ℓ_1 and $\ell_{i'}$ intersect. Hence $v_{i'} \in Y$, and v_1 and $v_{i'}$ satisfy the condition.

To derive a contradiction, we assume that there is a $v_j \in X$ that corresponds to (j, j') with $j \geq j'$ in G. Without loss of generality, every vertex corresponding to $\ell_k = (k, k')$ with $k < j$ satisfies the condition of the lemma. Then let x_j be the number of vertices in X placed before v_j on L_1, and y_j the number of vertices in Y placed before v_j on L_2, respectively. Moreover, let y'_j be the number of vertices in Y placed before v_j on L_1. If $j = j'$, we have $j - x_j = y'_j = y_j$. Hence G is disconnected, which is a contradiction. Thus assume $j > j'$. Then, we have $y_j + x_j = j' - 1 < j - 1 = x_j + y'_j$, equivalently, $y'_j > y_j$. Thus there exists $v_k \in Y$ with $\ell_k = (k, k')$ such that $k < j$ and $j' < k'$. We suppose that v_k is the leftmost one among such vertices. If $N(v_k) \cap X \cap \{v_1, \ldots, v_{k-1}\}$ is empty, it is not difficult to see that G is not connected (since v_j and v_k are the leftmost pair of the second connected component). Hence v_k has some neighbor, say v_x, in $X \cap \{v_1, \ldots, v_{k-1}\}$. By the assumption, for $\ell_j = (j, j')$, $\ell_k = (k, k')$, and $\ell_x(x, x')$, we have $x < k < j$ and $j' < k' < x'$. This implies that ℓ_j and ℓ_x intersect, which contradicts that v_j and v_x are in X. With a symmetric argument for Y, the lemma follows. □

One important property is that they are unique up to isomorphism like Lemma 2.14:

Lemma 2.24. *Let $G = (V, E)$ be a connected bipartite permutation graph, and $\mathcal{L}$ and $\mathcal{L}'$ any two line representations of G. Then one of $\mathcal{L} = \mathcal{L}'$, $\mathcal{L} = \mathcal{L}'^H$, $\mathcal{L} = \mathcal{L}'^V$, and $\mathcal{L} = \mathcal{L}'^R$ holds. That is, the line representation of G is unique up to isomorphism.*

Proof. By Lemma 2.23, we can partition V to X and Y. Let $G^2[X] = (X, E_X)$ be a graph obtained from G by joining two vertices $x, x' \in X$ if and only if $N(x) \cap N(x') \neq \emptyset$. That is, two vertices x and x' are joined in $G^2[X]$ if the distance between them is 2. In other words, x and x' are joined by some vertex in Y. We first show that $G^2[X]$ is a connected proper interval graph. Intuitively, from a line representation of G, we can obtain the interval representation of $G^2[X]$ as follows (see Figure 2.12(a)): we first rearrange the vertices in Y to vertical lines at regular intervals, and next make the vertices x in X be horizontal intervals spanning $N(x)$. Then

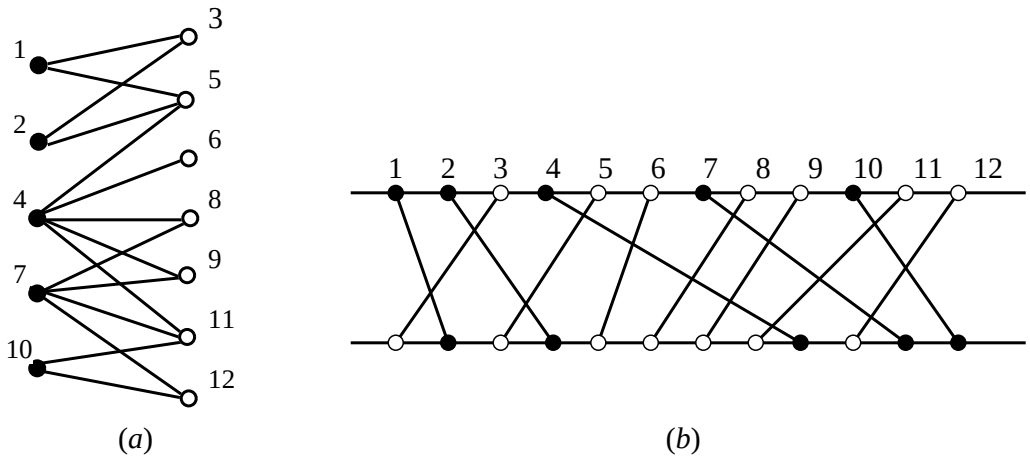


Figure 2.11: A bipartite permutation graph with its line representation.

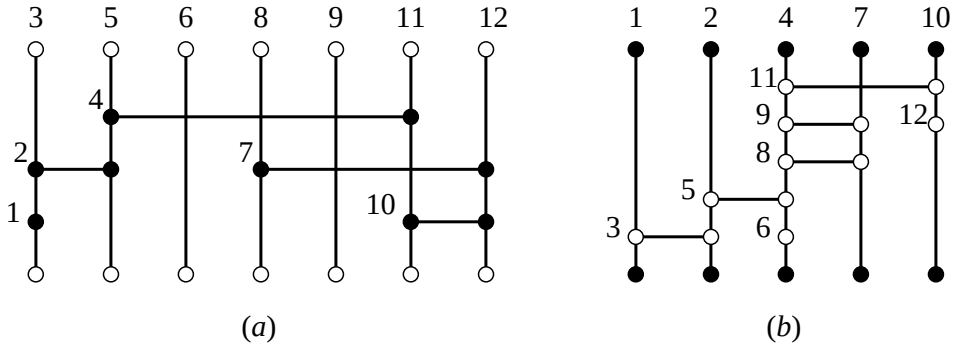


Figure 2.12: Proper interval graphs from the bipartite permutation graph in Figure 2.11(*a*).

the resultant intervals corresponding to the vertices x in X are proper, and this proper interval representation can be transformed to the unit interval representation in a straightforward way in [4]. The resultant graph $G^2[X]$ is also connected. Thus Lemma 2.14 implies that the resultant unit interval representation is unique up to reversal. $G^2[Y]$ can be defined in a symmetric way (see Figure 2.12(b)).

Now, we consider the rewind of this process. Given connected bipartite permutation graph $G = (V, E)$, X and Y are determined from G uniquely by Lemma 2.23. Then, by the discussion above, two proper interval graphs $G^2[X]$ and $G^2[Y]$ are uniquely determined. By Lemma 2.14, these unit interval graphs correspond to the unique interval representations. Thus, these unit interval representations give the unique orderings of X and Y in a natural way, respectively. Thus, combining these two orderings on X and Y with $G = (X, Y, E)$, we can construct the line representation of G uniquely as follows. First, we pick up the "leftmost" vertex x_1 in X according to the ordering of X. Then pick up the "leftmost" vertex y_1 from $N(x_1)$ according to the ordering of Y. Now all vertices in $N(x_1)$ are placed before x_1 on L_2 according to the ordering of Y, and all vertices in $N(y_1)$ are placed before y_1 on L_1 according to the ordering of X. Next we proceeds to x_2 and y_2, and so on. By a simple induction for the size of graph, we can show that the line representation of G is uniquely determined up to isomorphism. □

Let $G = (V, E)$ be a connected bipartite permutation graph, and $\mathcal{L}, \mathcal{L}^H, \mathcal{L}^V, \mathcal{L}^R$ its four line representations. It is easy to see that some of them can be isomorphic. We say G is *H-symmetric*, *V-symmetric*, and *R-symmetric* if $\mathcal{L} = \mathcal{L}^H$, $\mathcal{L} = \mathcal{L}^V$, and $\mathcal{L} = \mathcal{L}^R$, respectively.

Here, we map each representation $\mathcal{L}$ to a string $s(\mathcal{L})$ in Σ^* as follows. We first sweep the endpoints from left to right on L_1, and construct a string $s_1(\mathcal{L})$ by adding '[' when the endpoint is in X, and ']' when the endpoint is in Y (e.g., $s_1(\mathcal{L})$ = [[][]][]][] in Figure 2.11). Next we sweep the endpoints from left to right on L_2, and construct a string $s_2(\mathcal{L})$ by adding '[' when the endpoint is in Y, and ']' when the endpoint is in X (e.g., $s_2(\mathcal{L})$ = [][][[[[]]] in Figure 2.11). Finally, we concatenate $s_2(\mathcal{L})$ after $s_1(\mathcal{L})$ and obtain the resultant string (e.g., $s(\mathcal{L})$ = [[][]][]][][][[[[]]] in Figure 2.11).

Using the string, we define a *canonical* representation of G as follows. We first suppose that G is neither H-symmetric, V-symmetric, nor R-symmetric. Thus all strings $s(\mathcal{L})$, $s(\mathcal{L}^H)$, $s(\mathcal{L}^V)$, $s(\mathcal{L}^R)$ are distinct. Then the canonical representation is the one corresponding to the smallest string. When G satisfies exactly one symmetricalness with respect to H-flip, V-flip, or rotation, then four possible representations give two distinct strings. Then the canonical representation is the one corresponding to the smaller string. If G satisfies two symmetricalnesses, the last symmetricalness is also satisfied. Hence, in the case, four representations are isomorphic and this gives the unique canonical representation. By Lemma 2.24, this rule gives us a one-to-one mapping between bipartite permutation graphs and canonical representations.

2.6 Distance-Hereditary Graphs

An undirected graph G is *distance-hereditary* if and only if G is connected, and for any two distinct vertices x and y in G the length of induced paths between x and y are the same. It is convenient to define a new graph class not to care if graphs are connected. We define an undirected graph G is *weakly distance-hereditary* if and only if for any two distinct vertices x and y in G the length of induced paths between x and y are the same.

We define a useful pruning sequence for distance-hereditary graphs. We introduce the following two lemmas.

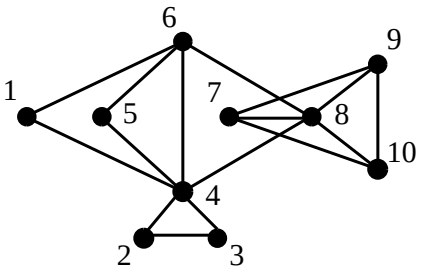
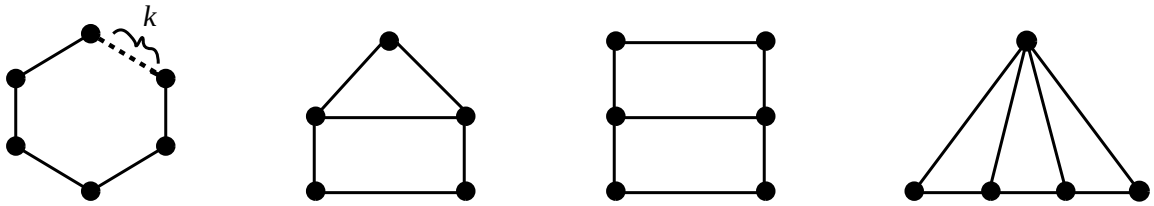


Figure 2.13: Distance-hereditary graph.



(*a*) (*b*) (*c*) (*d*)

Figure 2.14: Forbidden graphs of distance-hereditary graphs. The part described k contains k vertices ($k \geq 0$). (*a*) hole. (*b*) house. (*c*) domino. (*d*) gem.

Lemma 2.25 (Bendelt and Mulder [3])**.** *Given a graph $G = (V, E)$, and two vertices $u, v \in V$ are twin. Then G is distance-hereditary if and only if $G - u$ and $G - v$ are distance-hereditary.*

Lemma 2.26 (Bendelt and Mulder [3])**.** *Graph $G = (V, E)$ has a pendant vertex $w \in V$. Then G is distance-hereditary if and only if $G - w$ is distance-hereditary.*

Lemmas 2.25 and 2.26 give rise to an elimination scheme characterization of distance-hereditary graphs. A sequence $v_1 \dots v_n$ is a *pruning sequence* if each v_i either a pendant vertex in the graph induced by $\{v_i \dots v_n\}$, or a twin of some vertex in the graph induced by $\{v_i \dots v_n\}$.

Theorem 2.27 (Bendelt and Mulder [3])**.** *Graph G is distance-hereditary if and only if G has a pruning sequence.*

Nakano et al. proposed a DH-tree for a distance-hereditary graph [42]. We can see the tree as a uniquely defined canonical form of distance-hereditary graphs isomorphic to each other.

Theorem 2.28 (Bendelt and Mulder [3])**.** *Graph G is distance-hereditary if and only if G is connected, and (hole, house, domino, gem)-free; that is, G has none of the graphs in Figure 2.14 as an induced subgraph.*

2.7 Dyck path and Motzkin path

We explain a *Dyck path* and a *Motzkin path* for random generation of proper interval graphs and bipartite permutation graphs. A path in the (x, y) plane from $(0, 0)$ to $(2n, 0)$ with steps $(1, 1)$ and $(1, -1)$ is called a *Dyck path* of length $2n$ if it never pass below the x-axis (see Figure 2.15). It is well known that the number of Dyck paths of length n is given by the nth *Catalan number* $C(n) := \frac{1}{n+1}\binom{2n}{n}$ (see [50, Corollary 6.2.3] for further details). We will use one of the generalized

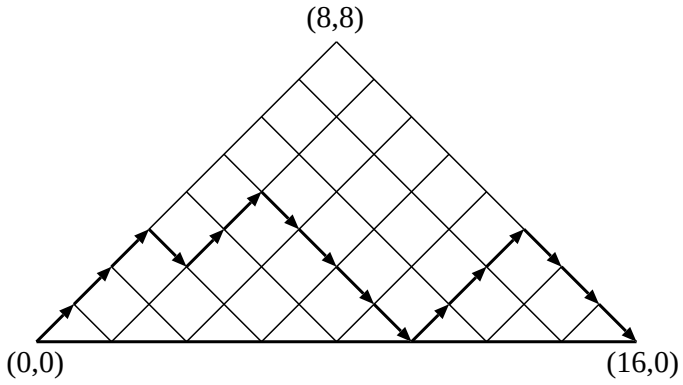


Figure 2.15: Dyck path

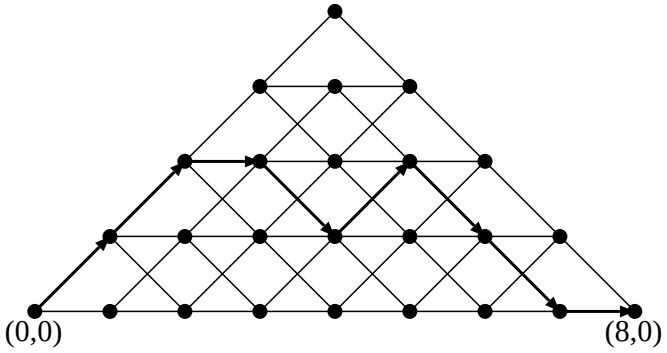


Figure 2.16: Motzkin path

notions of Catalan number; $C(n,k) := \frac{k+1}{n+1}\binom{n+1}{(n-k)/2}$, which gives us the number of subpaths of Dyck paths from $(0,0)$ to (n,k). This can be obtained by a generalized Raney's lemma about m-Raney sequences with letting $m = 2$, length n, and total sum k; see [22, Equation (7.69), p. 349] for further details.

A path in the (x,y) plane from $(0,0)$ to $(n,0)$ with steps $(1,0)$, $(1,1)$, and $(1,-1)$ is called a *Motzkin path* of length n if it never go below the x-axis (see Figure 2.16 and [50, Exercise 6.38] for further details). The number of Motzkin paths of length n is called *Motzkin number* $\mathcal{M}(n)$; e.g., $\mathcal{M}(1) = 1, \mathcal{M}(2) = 2, \mathcal{M}(3) = 4, \mathcal{M}(4) = 9, \mathcal{M}(5) = 21, \mathcal{M}(6) = 51$. A *2-Motzkin* path is a Motzkin path that has two kinds of step $(1,0)$. We distinguish them by $(1,+0)$ and $(1,-0)$. Deutsch and Shapiro show that 2-Motzkin paths have correspondences to ordered trees and others [14].

In paths above, each step consists of $(1,x)$ for some x in $\{\pm 1, \pm 0\}$. Hence we will denote a path by a sequence of such integers x in $\{\pm 1, \pm 0\}$.

2.8 Computational Model

Time complexity is measured by the number of arithmetic operations in the random generation algorithms. Especially we assume that each binomial coefficient and each (generalized) Catalan number can be computed in O(1) time. Moreover we assume that the basic arithmetic operations of these numbers can be done in O(1) time. Actually, a binomial coefficient and a (generalized) Catalan number require linear bits, so this assumption is clearly out of the standard RAM model.

We have to multiply the time complexity of calculation of these numbers to the complexities we show to obtain the time complexity in the standard RAM model. We employ the assumption in Section 3.1, 3.3 to simplify the discussion. It is worth remarking that the other algorithms does not require the assumption, and all the results are valid on the standard RAM model.

Chapter 3

Random Generation and Enumeration

3.1 Random Generation of Proper Interval Graphs

In this section, we count the number of mutually nonisomorphic proper interval graphs. We also propose an algorithm that efficiently generates a proper interval graph uniformly at random.

The number of proper interval graphs has been given by the recurrence equation in [23]. The closed equation of the number of proper interval graphs has been mentioned informally by Karttunen in 2002 [27]. We here give an explicit proof so that we use some notions for random generation.

Theorem 3.1 (Karttunen 2002)**.** *For any positive integer n, the number of connected proper interval graphs of $n+1$ vertices is $\frac{1}{2}(C(n) + \binom{n}{\lfloor n/2 \rfloor})$.*

Proof. We define three sets $S(n), T(n)$, and $U(n)$ of strings in Σ^{2n} of length $2n$ by

$S(n) = \{x \mid x$ is balanced, nonnegative, $|x| = 2n$, and x is symmetric$\}$,

$T(n) = \{x \mid x$ is balanced, nonnegative, $|x| = 2n$, and x is not symmetric$\}$, and

$U(n) = \{x \mid x$ is balanced, nonnegative, and $|x| = 2n\}$.

A balanced nonnegative string corresponds to a Dyck path, so $|U(n)| = C(n)$.

The number of connected proper interval graphs of $n+1$ vertices is equal to $|T(n)|/2 + |S(n)| = |U(n)|/2 + |S(n)|/2 = \frac{1}{2}(C(n) + |S(n)|)$, by Observation 2.15. The number of elements in $S(n)$ is equal to that of nonnegative strings x' of length n, since each symmetric string x is obtained by the concatenation of strings x' and $\overline{x'}$.

Now the task is the evaluation of the number of nonnegative strings x of length n with $h_x(n) = h$. Clearly we have $C(n,h) = 0$ if $h > n$. The following equations hold for each integers i and k with $0 \le i \le k$.

(1) $C(2k, 2i+1) = 0$, $C(2k+1, 2i) = 0$,

(2) $C(2k, 0) = C(k)$, $C(k,k) = 1$, and

(3) $C(k,i) = C(k-1, i-1) + C(k-1, i+1)$.

It is necessary to show $\sum_{i=0}^{n} C(n,i) = \binom{n}{\lfloor n/2 \rfloor}$ to complete the proof. This equation can be obtained from Equation (5.18) in [22]. □

Next we consider the uniform random generation of a proper interval graph of n vertices.

Theorem 3.2. *For any given positive integer n, a connected proper interval graph with n vertices can be generated uniformly at random in $O(n)$ time and $O(n)$ space. The time complexity to convert the string representation to a graph representation is $O(n+m)$ where m is the number of edges of the created graph.*

Proof. We denote by $y = y_1 \cdots y_{2n}$ the canonical string of a connected proper interval graph $G = (V, E)$ to be obtained. We fix $y_1 =$ '[' and $y_{2n} =$ ']' and generate $x = x_1 \cdots x_{2n'}$ with $y = [x]$, where $n' = n - 1$ and x is a balanced nonnegative string.

The idea is simple; just generate a balanced nonnegative string x. However each non-symmetric graph corresponds to two balanced nonnegative strings, while each symmetric graph corresponds to exactly one balanced nonnegative (symmetric) string. We use the equation $|T(n')|/2 + |S(n')| = |U(n')|/2 + |S(n')|/2 = \frac{1}{2}(C(n') + |S(n')|)$ in Theorem 3.1 in order to adjust the generation probabilities. The algorithm first selects which type of string to generate: (1) a balanced nonnegative string (that can be symmetric) with probability $|U(n')|/(|U(n')| + |S(n')|) = C(n')/(C(n') + \binom{n'}{\lfloor n'/2 \rfloor})$ or (2) a balanced nonnegative symmetric string with probability $|S(n')|/(|U(n')| + |S(n')|) = \binom{n'}{\lfloor n'/2 \rfloor}/(C(n') + \binom{n'}{\lfloor n'/2 \rfloor})$. This probabilistic choice adjusts the generation probabilities between symmetric graphs and non-symmetric graphs.

In each case, the algorithm generates each string uniformly at random using the function $C(n, h)$ introduced in the proof of Theorem 3.1 as follows:

Case 1: Generation of a balanced nonnegative string of length $2n'$ uniformly at random. There is a known algorithm for this purpose [1]. We simply generate sequence of '[' and ']' from left to right. Assume that the algorithm has already generated a nonnegative string $x_1 \cdots x_k$ of length k with $k < 2n'$. Next, we choose either '[' or ']' as x_{k+1}. The choice between alternative next states must be made on the basis of the proportion of terminal strings reached through the alternatives. The number of nonnegative strings that the next letter is '[' is $p = C(r, h_x(k) + 1)$, and the number of nonnegative strings that the next letter is ']' is $q = C(r, h_x(k) - 1)$, where r is equal to $2n' - k - 1$. Choose '[' as the next letter with probability $p/(p + q) = \frac{(h_x(k)+2)(r-h_x(k)+1)}{2(r+1)(h_x(k)+1)}$ and choose ']' with probability $p/(p+q) = \frac{h_x(k)(r+h_x(k)+3)}{2(r+1)(h_x(k)+1)}$. Then we have a balanced nonnegative string of length $2n'$ uniformly at random.

Case 2: Generation of a balanced nonnegative symmetric string of length $2n'$ uniformly at random. The desired balanced nonnegative symmetric string x can be represented as $x = x_1 x_2 \cdots x_{n'-1} x_{n'} \overline{x}_{n'} \overline{x}_{n'-1} \cdots \overline{x}_2 \overline{x}_1$, where $x_1 x_2 \cdots x_{n'}$ is a nonnegative string of length n'. We thus generate a nonnegative string $x' := x_1 x_2 \cdots x_{n'}$ of length n' uniformly at random.

Unfortunately, a similar approach to Case 1 does not work; given a positive prefix $x_1 x_2 \cdots x_i$, it seems to be hard to generate $x_{i+1} \cdots x_{n'}$ that ends at some $h_x(n')$ uniformly, since the string may pass below both of $h_x(i)$ and $h_x(n')$.

The key idea is to generate the desired string backwardly. This step consists of two phases. The algorithm first chooses the height $h_x(n')$ of the last letter $x_{n'}$ randomly. Then the algorithm randomly selects the height $h_x(i)$ of the ith letter x_i from $h_x(i + 1)$ for each $i = n' - 1, n' - 2, \ldots, 1$. That is, we have either $h_x(i) := h_x(i + 1) - 1$ or $h_x(i) := h_x(i + 1) + 1$ in general, and $h_x(0) = 0$ at last. From the sequence of the heights, we can construct $x = x_1 x_2 \cdots x_{n'}$ in $O(n)$ time and space: If $h_x(i) = h_x(i + 1) - 1$, we have $x_i =$ '[', and if $h_x(i) = h_x(i + 1) + 1$, we have $x_i =$ ']'.

We first consider the first phase. By the proof of Theorem 3.1, the number of nonnegative strings ending at height h is $C(n', h)$, and $\sum_{i=0}^{n'} C(n', i) = \binom{n'}{\lfloor n'/2 \rfloor}$. Hence, for each h with $0 \le h \le n'$, the algorithm sets $h_x(n') = h$ with probability $C(n', h)/\binom{n'}{\lfloor n'/2 \rfloor}$.

Next we consider the second phase. For general i with $1 \le i < n'$, the height $h_x(i)$ is either $h_x(i) = h_x(i + 1) + 1$ or $h_x(i) = h_x(i + 1) - 1$. The number of nonnegative strings of length i ending at the height $h_x(i + 1) + 1$ is $p = C(i, h_x(i + 1) + 1)$, and the number of nonnegative strings of length i ending at the height $h_x(i + 1) - 1$ is $q = C(i, h_x(i + 1) - 1)$. The algorithm sets $h_x(i) = h_x(i + 1) + 1$ with probability $p/(p+q) = \frac{(h_x(i+1)+2)(i-h_x(i+1)+1)}{2(i+1)(h_x(i+1)+1)}$ and sets $h_x(i) = h_x(i + 1) - 1$

with probability $q/(p+q) = \frac{h_x(i+1)(i+h_x(i+1)+3)}{2(i+1)(h_x(i+1)+1)}$. The algorithm finally obtains $h_x(1) = 1$ and $h_x(0) = 0$ with probability 1 after repeating this process. The string $x' = x_1x_2\cdots x_{n'-1}x_{n'}$ can be computed from the sequence of heights by traversing the sequence of the heights backwards and hence we can obtain $x = x'\overline{x'}$.

□

Note that the only part that requires $O(n)$ space is the generation of $x' = x_1x_2\cdots x_{n'}$ from the sequence of their heights $h_x(n'), h_x(n'-1), \ldots, h_x(1)$ in Case 2 in the proof of Theorem 3.2. Therefore, if we are admitted to output the symmetric string $x\overline{x}$ by just $\overline{x}$, the algorithm in Theorem 3.2 requires space of only $O(n)$ bits.

In the RAM model, binomial coefficient $\binom{n}{k}$ can be computed in $O(k^2 + k\log k)$ time and $O(k)$ space with Iriyama's algorithm[31]. Thus Catalan number and its generalization can be computed in $O(n^2)$ time. Since we compute the generalized Catalan number $\frac{n}{2}$ times in the first phase in Case 2, our random generation algorithm can be performed in $O(n^3)$ time. Note that $C(n)$ is exponentially larger than $\binom{n}{\lfloor n/2\rfloor}$ so the probability of selecting Case 2 is close to 0. Therefore our algorithm runs in $O(n^2)$ expected time on the RAM model.

3.2 Enumeration of Proper Interval Graphs

We here enumerate all connected proper interval graphs with n vertices. It is sufficient to enumerate each string representation of connected proper interval graphs, by Lemma 2.14. Let S_n be the set of balanced and canonical strings $x = x_1x_2\cdots x_{2n}$ in Σ^{2n} such that $x_1 =$ '[', $x_{2n} =$ ']', and the string $x_2\cdots x_{2n-1}$ is nonnegative. We define a tree structure, called *family tree*, in which each vertex corresponds to each string in S_n. We enumerate all the strings in S_n by traversing the family tree. Since S_n is trivial when $n = 1, 2$, we assume $n > 2$.

We start with some definitions. Let $x = x_1x_2\cdots x_{2n}$ be a string in Σ^{2n}. If $x_ix_{i+1} =$ [ ], i is called a *front index* of x. Contrary, if $x_ix_{i+1} =$][, i is called a *reverse index* of x. For example, a string [[[[ ][ ][ ]][ ]][ ][ ]] has 6 front indices 4, 6, 8, 11, 14, and 16, and has 5 reverse indices 5, 7, 10, 13, and 15. The string $[^n]^n$ in S_n is called the *root* and denoted by r_n. Let $x = x_1x_2\cdots x_{2n}$ be a string in Σ^{2n}. We denote the string $x_1x_2\cdots x_{i-1}\overline{x_i}\,\overline{x_{i+1}}x_{i+2}\cdots x_{2n}$ by $x[i]$ for $i = 1, 2, \ldots, 2n-1$. We define $P(x)$ by $x[j]$ for $x \in \Sigma^{2n} \setminus \{r_n\}$, where j is the minimum reverse index of x. For example, for $x =$[[[[ ][ ]][ ]][ ][ ]], we have $P(x) =$[[[[ ][ ]][ ]][ ][ ]] (the flipped pair is enclosed by the grey box and the minimum reverse indices are underlined).

Lemma 3.3. *For every $x \in S_n \setminus \{r_n\}$, we have $P(x) \in S_n$.*

Proof. For any $x \in S_n \setminus \{r_n\}$, it is easy to see that $P(x) = x'_1x'_2\cdots x'_{2n}$ satisfies that $x'_1 =$ '[', $x'_{2n} =$ ']', $x'_2\cdots x'_{2n-1}$ is balanced and nonnegative. Thus we show that $P(x)$ is canonical.

We first assume that x is symmetric. Then the minimum reverse index j of x satisfies $j \le n$, since x is symmetric and is not the root. If $j = n$, $P(x)$ is still symmetric and hence $P(x)$ is canonical. When $j < n$, we have $x_{j'} = \overline{x}_{2n-j'+1}$ for each $1 \le j' < j$, $x_j =$ '[', and $x'_{2n-j+1} =$ '['. Hence $P(x) < \overline{P(x)}$.

Next we consider the case that x is not symmetric. There must be an index i such that $x_i = x_{2n-i+1} =$ '[' and $x_{i'} = \overline{x}_{2n-i'+1}$ for all $1 \le i' < i$, since x is canonical. Moreover we have $1 \le i < n$, since x is balanced. Let ℓ be the minimum reverse index of x. We first observe that $\ell \ne i$ since $x_\ell =$ ']'. We also see that $\ell < 2n - i + 1$ since $x_{i'} = \overline{x}_{2n-i'+1}$ for all $1 \le i' < i$. If $\ell < i - 1$, using a similar argument above, we have $P(x) < x < \overline{P(x)}$. If $\ell > i$, the changes to

the string has no effect; we still have $x'_i =$ '[' and $x'_{2n-i+1} =$ '[' and hence $P(x) < \overline{P(x)}$. The last case is $\ell = i - 1$. In this case, we have $x_{i-1}x_i =$ '][', $x_{n-i+1}x_{n-i+2} =$ '[[' and $x_{i'} = \overline{x}_{2n-i'+1}$ for all $1 \le i' < i$. Thus we have $x'_{i-1}x'_i =$ '[]', $x'_{n-i+1}x'_{n-i+2} =$ '[[', and $x'_{i'} = \overline{x}'_{2n-i'+1}$ for all $1 \le i' < i - 1$. This implies that $P(x) < \overline{P(x)}$. □

Next we define the family tree among strings in S_n. We call $P(x)$ the *parent* of x, and x is a *child* of $P(x)$ for each $x \in S_n \setminus \{r_n\}$. Note that $x \in S_n$ may have multiple or no children while each string $x \in S_n \setminus \{r_n\}$ has the unique parent $P(x) \in S_n$. Given a string x in $S_n \setminus \{r_n\}$, we have the unique sequence $x, P(x), P(P(x)), \ldots$ of strings in S_n by repeatedly finding the parent. We call it the *parent sequence* of x. For example, for $x =$[[[ ][ ]][ ], we have $P(x) =$[[[[ ]]][ ], $P(P(x))=$[[[[ ][ ]]], $P(P(P(x))) =$[[[[ ][ ]]]], and $P(P(P(P(x)))) = r_5$. The next lemma ensures that the root r_n is the common ancestor of all the strings in S_n.

Lemma 3.4. *The parent sequence of x in S_n eventually ends up with r_n.*

Proof. For a string $x = x_1x_2 \cdots x_{2n}$ in S_n, we define a potential function $p(x) = \sum_{i=1}^{n} 2^{n-i}b(x_i) + \sum_{i=1}^{n} 2^{i-1}(1 - b(x_{n+i}))$, where $b($'['$) = 0$ and $b($']'$) = 1$. For any $x \in S_n$, $p(x)$ is a non-negative integer, and $p(x) = 0$ if and only if $x = r_n$.

Suppose x is not the root r_n. Then x has the minimum reverse index, say j. If $j = n$, it is easy to see that $p(P(x)) = p(x) - 2$. We suppose that $j < n$. Then we have $p(P(x)) = p(x) - 2^{n-j} + 2^{n-j-1} = p(x) - 2^{n-j-1} < p(x)$ by the definitions of the parent and the potential function. The case $j > n$ is symmetric and we obtain $p(P(x)) < p(x)$. Therefore we eventually obtain the root r_n by repeatedly finding the parent of the derived string, which completes the proof. □

We have the *family tree* T_n of S_n by merging all the parent sequences. Each vertex in the family tree T_n corresponds to each string in S_n, and each edge corresponds to each parent-child relation. See Figure 3.1 for example.

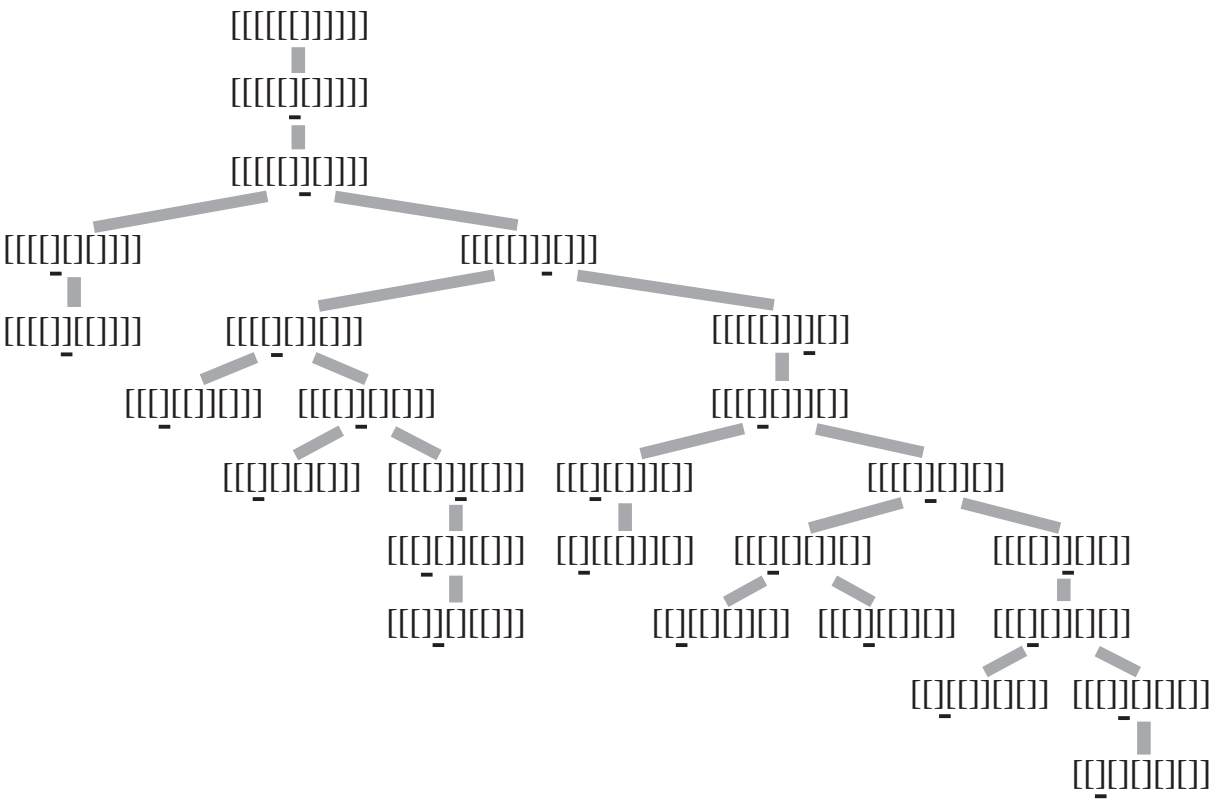
Now we give an algorithm that enumerates all the strings in S_n. The algorithm traverses the family tree by reversing the procedure of finding the parent as follows. Given a string x in S_n, we enumerate all the children of x. Every child of x is in the form $x[i]$ where i is a front index of x. We consider the following cases to find every i such that $x[i]$ is a child of x.

Case 1: String x is the root r_n. The string x has exactly one front index n. Since $P(x[n]) = x$, $x[n]$ is a child of x. Since $x[i]$ is not a child of x when i is not a front index, x has exactly one child.

Case 2: String x is not the root. In this case, x has at least two front indices. Let i be any front index, and j be the minimum reverse index. If $i > j + 1$ then $x \neq P(x[i])$, since i is not the minimum reverse index of $x[i]$. If $i \le j + 1$, $x[i]$ may be a child of x. Thus we call i satisfying the minimum front index or $j + 1$ a *candidate index* of x. For a candidate index i, if $x[i]$ is in S_n (i.e. $x[i]$ is canonical), i is called a *flippable index* and $x[i]$ is a child of x. There must exist a reverse index between any two front indices. Since only the indices the satisfying the minimum front index or $j + 1$ can be candidate indices, x has at most two candidate indices. Thus x has at most two children. For example, $x =$ [[[]][]] has two candidate indices 3 and 6, one reverse index 5, and one child $x[3] =$ [[][][]].

Given a string x in S_n, we can enumerate all the children of x by the case analysis above. We can traverse T_n by repeating this process from the root recursively. Thus we can enumerate all the strings in S_n.

Now we have the algorithms and lemma.


[[[[[[]]]]]]
[[[[[]]]]]
[[[[]]]]
[[[[]]]]
[[[[]]]]
[[[[]]]]
[[[]]]]
[[[]]]]

Algorithm 2: `find-all-strings`

Input: integer n

```
1 begin
2 |   Output the root x = r_n.
3 |   find-all-child-strings (x[n]);                    /* Case 1 */
4 end
```

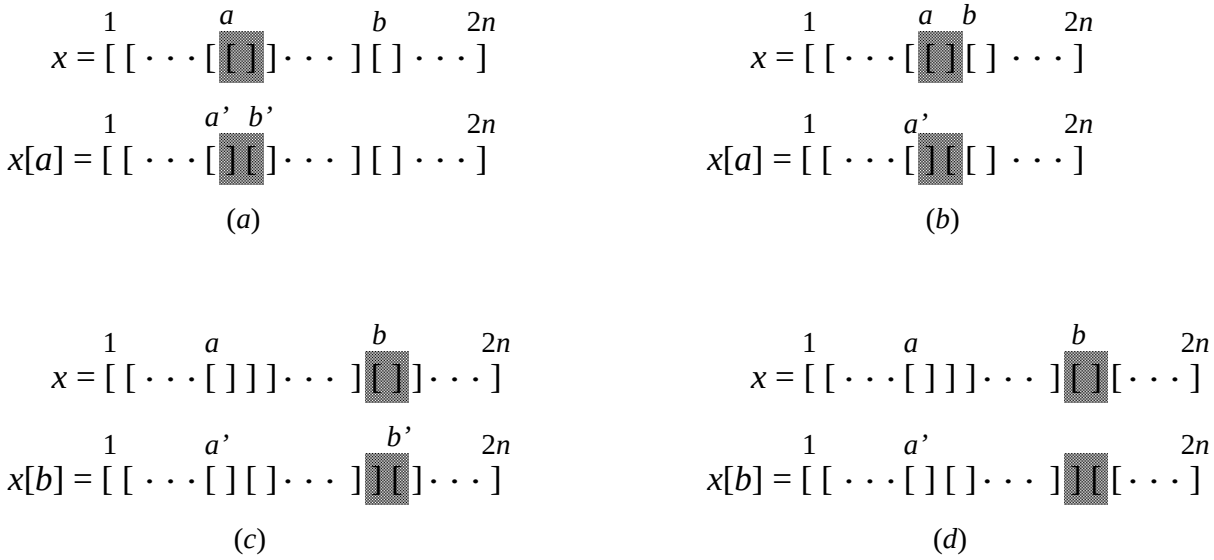


Figure 3.2: Case analysis of candidate indices.

Case 2: A child $x[b]$ of x. If x_{b+2} = ']', we have two candidate indices $a' = a$ and $b' = b + 1$ (see Figure 3.2(c)). Otherwise we have one candidate index $a' = a$ (see Figure 3.2(d)).

By the above case analysis, a candidate index of a child either (1) appears in the previous or next index of a or b, or (2) is identical to one of x's. □

Since the number of candidate indices of x is at most two, our family tree is a binary tree. We note that a candidate index of x can become "non-candidate". In the case, such index does not become a candidate index again.

Next lemma shows that there is a method of determining whether a candidate index is flippable.

Lemma 3.7. *One can determine whether or not a candidate index is flippable in O*(1) *time.*

Proof. Let $x = x_1x_2 \cdots x_{2n}$ be a string in S_n and a be a candidate index of x. We denote $x[a] = y = y_1y_2 \cdots y_{2n}$. A candidate index a is flippable if and only if $x[a]$ is canonical and $y_2y_3 \cdots y_{2n-1}$ is nonnegative.

We first check whether or not a string $y_2y_3 \cdots y_{2n-1}$ is nonnegative. We have $h_y(a) = h_x(a) - 2$ and $h_y(i) = h_x(i)$ for each $1 \leq i < a$ and $a < i \leq 2n$, since $a > 1$, y_ay_{a+1} = '][', x_ax_{a+1} = '[]', and $x_i = y_i$ for each $1 \leq i < a$ and $a + 1 < i \leq 2n$. Thus $y_2y_3 \cdots y_{2n-1}$ is nonnegative if and only if $h_x(a) > 2$. Therefore we can check the negativity of $y_2y_3 \cdots y_{2n-1}$ in O(1) time using an array of size n to maintain the sequence of heights of the string. Updates of the array also can be done in O(1) time.

We next check whether or not a string is canonical. We call $x^{\mathrm{L}} = x_1 x_2 \cdots x_n$ the *left string* of x, and $x^{\mathrm{R}} = \overline{x}_{2n}\overline{x}_{2n-1}\cdots\overline{x}_{n+1}$ the *right string* of x. Then x is canonical if and only if $x^{\mathrm{L}} \leq x^{\mathrm{R}}$. We maintain a doubly linked list L in order to check it in O(1) time. The list L maintains the indices of different characters in x^{L} and x^{R}. First L is initialized by an empty since $x^{\mathrm{L}} = x^{\mathrm{R}}$ for $x = r_n$. In general L is empty if and only if x is symmetric. We can check whether $x^{\mathrm{L}} < x^{\mathrm{R}}$ by comparing $x^{\mathrm{L}}_{L[1]}$ and $x^{\mathrm{R}}_{L[1]}$.

Now we have that x is canonical and nonnegative, $x[a]$ is nonnegative, and L consists of the different indices of x^{L} and x^{R}. Then we have $x^{\mathrm{L}}_{L[1]}$ is '[' and $x^{\mathrm{R}}_{L[1]}$ is ']'. We introduce two pointers associated to the candidate index a to update the list efficiently; two pointers p^{L}_a and p^{R}_a that point two elements in the list L. Intuitively p^{L}_a and p^{R}_a give the two indices $L[i]$ and $L[i+1]$ such that a is between $L[i]$ and $L[i+1]$. When L is empty, p^{L}_a and p^{R}_a are also empty. Assume that L consists of k elements $L[1]$, $L[2]$, ..., $L[k]$. Then we have one of the following three cases. (1) If a is between $L[i]$ and $L[i+1]$, p^{L}_a and p^{R}_a point $L[i]$ and $L[i+1]$, respectively. More precisely, this case occurs either $1 \leq a \leq n$ and $L[i] \leq a < L[i+1]$ for some i or $n+1 \leq a \leq 2n$ and $L[i] \leq 2n - a + 1 < L[i+1]$ for some i. (2) If a is less than $L[1]$, p^{L}_a and p^{R}_a point $L[1]$. This case occurs either $a < L[1]$ or $a > 2n - L[1]$. (3) Otherwise, i.e., the case $L[k] \leq a \leq 2n - L[k]$. In this case p^{L}_a and p^{R}_a point $L[k]$. Now we assume that we update x by $x[a]$, $x_a x_{a+1}$ = '[]' is replaced by $x_a x_{a+1}$ = '][]'. It is straightforward and tedious that we can update the list L in O(1) time; typically, if $L[p^{\mathrm{L}}_a] < a$ and $a + 1 < L[p^{\mathrm{R}}_a]$, the algorithm inserts two new elements between p^{L}_a and p^{R}_a in L. When p^{L}_a points a, the algorithm remove it from L. The other cases are similar, and hence omitted.

The flippable index a is updated by $a - 1$ or $a + 1$ by Lemma 3.6. Hence the update of p^{L}_a and p^{R}_a can be done in O(1) time, which completes the proof. □

Lemmas 3.6 and 3.7 show that we can maintain the list of flippable indices of each string in O(1) time, during the traversal of the family tree. Thus we have the following lemma.

Lemma 3.8. Algorithm 2 *uses $O(n)$ space and runs in $O(|S_n|)$ time.*

By lemma 3.8, **Algorithm 2** generates each string in S_n in O(1) time "on average". However it may have to return from the deep recursive calls without outputting any string after generating a string corresponding to the leaf of a large subtree in the family tree. This takes much time. Therefore each string may not be generated in O(1) time in the worst case.

This delay can be canceled by outputting the strings in the "prepostorder" manner in which strings are outputted in the preorder (and postorder) manner at the vertices of odd (and even, respectively) depth of the family tree. See [43] for further details of this method; in [43] the method was not explicitly named, and the name "prepostorder" was given by Knuth [29]. Now we have the main theorem in this section.

Theorem 3.9. *After outputting the root in $O(n)$ time, the algorithm enumerates every string in S_n in $O(1)$ time.*

Let G and $G[i]$ be two proper interval graphs corresponding to a string x and its child $x[i]$, respectively. We note that $G[i]$ can be obtained from G by removing the one edge which represents an intersection between (1) the interval with the right endpoint corresponding to x_{i+1} and (2) one with the left endpoint corresponding to x_i. Moreover, the root string represents a complete graph. Therefore **Algorithm 2** can be modified to deal with the graphs themselves without loss of efficiency. Note that it is not true that every constant delay enumeration algorithm for parentheses applies to that for proper interval graphs since the sizes of differences may not equal among string representations and graph representations.

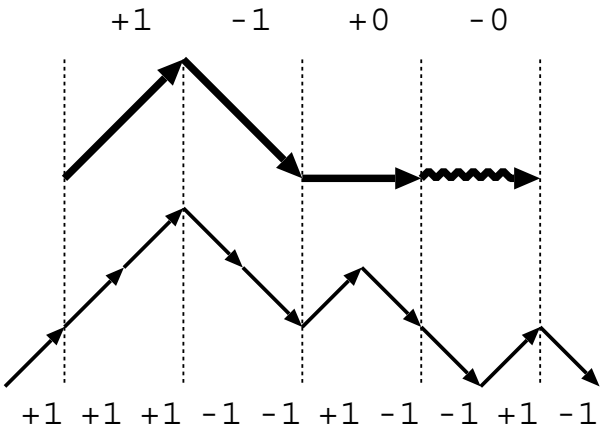


Figure 3.3: An example of the bijection

Theorem 3.10. *After outputting the n-vertex complete graph in $O(n^2)$ time, the algorithm enumerates every connected proper interval graph of n vertices in $O(1)$ time.*

3.3 Random Generation of Bipartite Permutation Graphs

Let $P(n)$ be the set of permutations corresponding to connected bipartite permutation graphs of n vertices, and $\mathcal{B}_n$ the set of distinct (up to isomorphism) connected bipartite permutation graphs of n vertices. We denote the line representation of a permutation π by $\mathcal{L}_\pi = (L_1, L_2)$, and the graph of π by $G_\pi = (X, Y, E)$. Without loss of generality, we assume that X contains the vertex corresponding to $(1, \pi(1))$ in $\mathcal{L}_\pi$ for $\pi(1) > 1$. Now, we construct a 2-Motzkin path as follows. For each i with $1 \leq i \leq n$, we see the endpoints at i on L_1 and L_2. Let p_i and q_i be the endpoints on L_1 and L_2, and we say that p_i is in X (and Y) if p_i is the endpoint of a vertex corresponding to $(i, \pi(i))$ in X (and Y, respectively). If G_π is not connected, in each connected component, we assume that the vertex corresponding to the leftmost point on L_1 belongs to X. Then the value z_i is defined as follows;

$$z_i = \begin{cases} +1 & \text{if } p_i \text{ is in } X \text{ and } q_i \text{ is in } Y, \\ -1 & \text{if } p_i \text{ is in } Y \text{ and } q_i \text{ is in } X, \\ +0 & \text{if } p_i \text{ and } q_i \text{ are in } X, \\ -0 & \text{if } p_i \text{ and } q_i \text{ are in } Y. \end{cases}$$

That is, two values $+0$ and -0 are distinguished (for counting) but have the same value. From the sequence $z_1, \ldots, z_n$, we can consider a path $Z_\pi = (z_1, \ldots, z_n)$. Note that $\pi = \pi'$ if and only if $Z_\pi = Z_{\pi'}$. For the path Z_π, we define its *height at point i* by $\sum_{j=1}^{i} z_j$. To simplify, we define that the height at point 0 is 0. We show that Z_π is a 2-Motzkin path that has positive height at point i, $1 < i < n$, if and only if $\pi \in P(n)$. To this end, we need a property of connected permutation graphs.

Lemma 3.11 ([30, Lemma 3.2])**.** *Let π be a permutation on $\{1, \ldots, n\}$. Then G_π is disconnected if and only if there exists $k < n$ such that $\{\pi(1), \pi(2), \ldots, \pi(k)\} = \{1, 2, \ldots, k\}$.*

Then we have the following lemma.

Lemma 3.12. *A sequence $Z = (z_1, \ldots, z_n)$ on the alphabet $\{+1, -1, +0, -0\}$ is constructed from $\pi \in P(n)$ in the above way if and only if Z is a 2-Motzkin path such that Z has height 0 at point 0 and n, and positive height at point i with $0 < i < n$.*

Proof. ($\Longrightarrow$) Clearly, $z_1 = +1$ and $z_n = -1$ since $G_\pi = (X, Y, E)$ is connected, and X and Y are nonempty. It is easy to see that the number of $+1$ is equal to the one of -1 in Z. Thus $\sum_{i=1}^{n} z_i = 0$. If Z has height 0 at some point k with $0 < k < n$, we have that $\pi(i) \in \{1, \ldots, k\}$ for $1 \le i \le k$. From Lemma 3.11, we have that G_π is disconnected, which is a contradiction.
($\Longleftarrow$) We can construct a line representation $\mathcal{L} = (L_1, L_2)$ from Z as follows:

1. At point i ($1 \le i \le n$) on L_1, put $\mathtt{x}$ if $z_i \in \{+1, +0\}$, otherwise put $\mathtt{y}$;
2. At point i ($1 \le i \le n$) on L_2, put $\mathtt{x}$ if $z_i \in \{-1, +0\}$, otherwise put $\mathtt{y}$;
3. Draw a line segment from the ith $\mathtt{x}$ on L_1 to the ith $\mathtt{x}$ on L_2 for each i;
4. Draw a line segment from the ith $\mathtt{y}$ on L_1 to the ith $\mathtt{y}$ on L_2 for each i.

Then, we have a permutation π of $\mathcal{L}$. Thus, it suffices to show that $\pi \in P(n)$, that is, G_π is connected and bipartite. Clearly, two lines in $\mathcal{L}$ intersect only if one of them is a line from $\mathtt{x}$ to $\mathtt{x}$ and another line is from $\mathtt{y}$ to $\mathtt{y}$. So, G_π is bipartite. If G_π is disconnected then there exists an index $k < n$ such that $\pi(i) \in \{1, \ldots, k\}$ for $1 \le i \le k$ (Lemma 3.11). Obviously, this implies $\sum_{i=1}^{k} z_i = 0$, which contradicts the assumption. □

From the above characterization, we can count the number of elements in $P(n)$. Deutsch and Shapiro [14] have shown the following bijection between 2-Motzkin paths of length n and Dyck paths of length $2(n+1)$: In a 2-Motzkin path, we replace $+1$ by $(+1, +1)$, -1 by $(-1, -1)$, $+0$ by $(+1, -1)$, and -0 by $(-1, +1)$; Then add $+1$ before the obtained sequence, and add -1 after the sequence. Figure 3.3 shows an example. Note that a 2-Motzkin path has height k at point i if and only if the corresponding Dyck path has height $2k + 1$ at point $2i + 1$. The following lemma follows from the bijection.

Lemma 3.13 ([14])**.** *The number of 2-Motzkin paths of length n is $C(n + 1)$.*

Corollary 3.14. $|P(n)| = C(n - 1)$.

Proof. Let $\pi \in P(n)$. Since π bijectively corresponds to Z_π, it suffices to count the number of Z_π. Lemma 3.12 and its proof imply that Z_π bijectively corresponds to a 2-Motzkin path of length $n - 2$ (as the first and the last step in Z_π is removed). The corollary follows from Lemma 3.13. □

We can show that the bijection is also a bijection for restricted paths. For $z \in \{+1, -1, +0, -0\}$, we define $-z$ naturally. That is, $-z = \pm b$ if and only if $z = \mp b$ for $b \in \{0, 1\}$. A Dyck path $D = (d_1, \ldots, d_{2n})$ is *symmetric* if $z_i = -z_{n-i+1}$ for $1 \le i \le n$. The number of symmetric Dyck paths is $\binom{n}{\lfloor n/2 \rfloor}$ from the proof of Theorem 3.1. A 2-Motzkin path $Z = (z_1, \ldots, z_n)$ is *semi-symmetric* if $z_i = -z_{n-i+1}$ for $1 \le i \le n$, and Z is *symmetric* if $z_i = -z_{n-i+1}$ for $z_i \in \{+1, -1\}$ and $z_i = z_{n-i+1}$ for $z_i \in \{+0, -0\}$. Note that a 2-Motzkin path can be semi-symmetric only if its length is even. Obviously, the bijection is also a bijection between symmetric 2-Motzkin paths of length n and symmetric Dyck paths of length $2(n + 1)$. Furthermore, if n is even, there is a bijection between semi-symmetric 2-Motzkin paths of length n and symmetric Dyck paths of length $2(n+1)$, since a semi-symmetric 2-Motzkin path can be bijectively transformed to a symmetric 2-Motzkin path by flipping the signs of 0s in the right half. From the above observation and Theorem 3.1, we have the following corollary.

Corollary 3.15. *The number of symmetric 2-Motzkin paths of length n is* $\binom{n+1}{\lfloor (n+1)/2 \rfloor}$. *If n is even, the number of semi-symmetric 2-Motzkin paths of length n is also* $\binom{n+1}{\lfloor (n+1)/2 \rfloor}$.

Any given $\pi \in P(n)$, Lemma 2.24 implies that there exist at most four line representations $\mathcal{L}_\pi$, $\mathcal{L}_\pi^H$, $\mathcal{L}_\pi^V$, and $\mathcal{L}_\pi^R$ for a graph G_π. We define four subsets of $P(n)$ as follows: (1) $P^H(n) = \{\pi \in P(n) \mid \mathcal{L}_\pi \text{ is H-symmetric}\}$, (2) $P^V(n) = \{\pi \in P(n) \mid \mathcal{L}_\pi \text{ is V-symmetric}\}$, (3) $P^R(n) = \{\pi \in P(n) \mid \mathcal{L}_\pi \text{ is R-symmetric}\}$, and (4) $P^F(n) = P^H(n) \cap P^R(n) \cap P^V(n)$.

Proposition 3.16. *(1) If n is odd,* $P^H(n)$ *and* $P^V(n)$ *are empty. (2)* $P^F(n) = P^H(n) \cap P^V(n) = P^V(n) \cap P^R(n) = P^R(n) \cap P^H(n)$.

Proof. (1) Both H-flip and V-flip exchange X and Y, which are determined uniquely by Lemma 2.23. Thus $P^H(n)$ and $P^V(n)$ can be nonempty only if $|X| = |Y|$. Therefore, they are empty if $|X| + |Y|$ is odd.

(2) Let $\pi \in P^H(n) \cap P^V(n)$. Then $\mathcal{L}_\pi = \mathcal{L}_\pi^H = \mathcal{L}_\pi^V$. Since $\mathcal{L}_\pi^R = \left(\mathcal{L}_\pi^H\right)^V$ for any π, we have that $\mathcal{L}_\pi^R = \left(\mathcal{L}_\pi^H\right)^V = \mathcal{L}_\pi^V = \mathcal{L}_\pi$. Hence $\pi \in P^R(n)$. The remaining two cases are similar. □

Lemma 3.17. $|\mathcal{B}_n| = \frac{1}{4}\left(|P(n)| + \left|P^H(n)\right| + \left|P^V(n)\right| + \left|P^R(n)\right|\right)$.

Proof. From Lemma 2.24 and Proposition 3.16, each connected bipartite permutation graph corresponds to four, two, and one permutations if it has no, one, and three symmetricalness, respectively. According to the number of corresponding permutations, we can partition $\mathcal{B}_n$ into three sets $\mathcal{B}_n^4$, $\mathcal{B}_n^2$, and $\mathcal{B}_n^1$. Each element of $\mathcal{B}_n^i$ corresponds to exactly i permutations in $P(n)$: For $G \in \mathcal{B}_n^1$, there exists $\pi \in P^F(n)$ such that $G \simeq G_\pi$; For $G \in \mathcal{B}_n^2$, there exist two permutations π_1 and π_2 in $(P^H(n) \cup P^V(n) \cup P^R(n)) \setminus P^F(n)$ such that $G \simeq G_{\pi_1} \simeq G_{\pi_2}$; For $G \in \mathcal{B}_n^4$, there exist four permutations π_i, $1 \le i \le 4$, in $P(n) \setminus \left(P^H(n) \cup P^V(n) \cup P^R(n)\right)$ such that $G \simeq G_{\pi_i}$ for $1 \le i \le 4$. Combining the inclusion-exclusion principle with Proposition 3.16 implies that

$$\left|P^H(n) \cup P^V(n) \cup P^R(n)\right| = \left|P^H(n)\right| + \left|P^V(n)\right| + \left|P^R(n)\right| - 2\left|P^F(n)\right|.$$

So, we have that

$$\begin{aligned}
|\mathcal{B}_n| &= \left|\mathcal{B}_n^1\right| + \left|\mathcal{B}_n^2\right| + \left|\mathcal{B}_n^4\right| \\
&= \left|P^F(n)\right| + \frac{1}{2}\left(\left|P^H(n)\right| + \left|P^V(n)\right| + \left|P^R(n)\right| - 3\left|P^F(n)\right|\right) \\
&\qquad + \frac{1}{4}\left(|P(n)| - \left|P^H(n)\right| - \left|P^V(n)\right| - \left|P^R(n)\right| + 2\left|P^F(n)\right|\right) \\
&= \frac{1}{4}\left(|P(n)| + \left|P^H(n)\right| + \left|P^V(n)\right| + \left|P^R(n)\right|\right),
\end{aligned}$$

as required. □

Lemma 3.17 implies that it suffices to count the elements of $P(n)$, $P^H(n)$, $P^V(n)$, and $P^R(n)$ to show the size of $\mathcal{B}_n$. For the random generation, $\left|P^F(n)\right|$ is also necessary.

Lemma 3.18. $\left|P^V(n)\right| = C(n/2 - 1)$ *for even n.*

Proof. Let $n = 2m$ and $\pi \in P^V(2m)$. We claim that $Z_\pi = (z_1, \ldots, z_{2m})$ contains neither $+0$ nor -0. If $z_i = +0$ for some i, $1 \le i \le 2m$, $\mathcal{L}_\pi$ contains the lines (i, j) and (k, i) for some j and k, $k < i < j$. However, since $\mathcal{L}_\pi$ is V-symmetric, $\mathcal{L}_\pi$ contains (j, i) as well. This implies that $j = k$, a contradiction. The proof of $z_i \neq -0$ is almost the same. Thus Z_π bijectively corresponds to a Dyck path of length $2(m-1)$, as required. □

Lemma 3.19. $\left|P^R(n)\right| = \binom{n-1}{\lfloor (n-1)/2 \rfloor}$.

Proof. From Corollary 3.15, it suffices to show that $\pi \in P^R(n)$ if and only if the 2-Motzkin path Z_π is symmetric and has positive height at point i with $1 < i < n$.
($\Longrightarrow$) Suppose $z_i = +1$. Then the lines (i, j) and (k, i), $i < j$ and $i < k$, are in $\mathcal{L}_\pi$. Since $\pi \in P^R(n)$, we have that $(n - j + 1, n - i + 1)$ and $(n - i + 1, n - k + 1)$ are also in $\mathcal{L}_\pi$. Therefore, $z_{n-i+1} = -1$ since $i < j$ and $i < k$. The case $z_i = -1$ is similar.

Next, suppose $z_i = +0$. Then the lines (i, j) and (k, i), $k < i < j$, are in $\mathcal{L}_\pi$. Since $\pi \in P^R(n)$, we have that $(n - j + 1, n - i + 1)$ and $(n - i + 1, n - k + 1)$ are also in $\mathcal{L}_\pi$. Therefore, $z_{n-i+1} = +0$ since $k < i < j$. The case $z_i = -0$ is similar.
($\Longleftarrow$) Clearly, $\pi \in P(n)$. Let $(i, j) \in \mathcal{L}_\pi$. We show that $(n - j + 1, n - i + 1)$ is also in $\mathcal{L}_\pi$. Without loss of generality, we assume that $i < j$, namely $(i, j) \in X$. Let i and j be the kth endpoints of lines in X, on L_1 and L_2, respectively. For $1 \le \ell < i$, the number of indices ℓ such that $z_\ell \in \{+1, +0\}$ is $k - 1$. Since Z_π is symmetric, for $n - i + 1 < \ell \le n$ the number of indices ℓ such that $z_\ell \in \{-1, +0\}$ is also $k-1$. This implies that the point $n-i+1$ on L_2 is the $(|X|-k+1)$th endpoint of a line in X. Similarly, we can show that the point $n - j + 1$ on L_1 is the $(|X| - k + 1)$th endpoint of a line in X. Therefore, $(n - j + 1, n - i + 1) \in \mathcal{L}_\pi$. □

Lemma 3.20. $\left|P^H(n)\right| = \binom{n-1}{\lfloor (n-1)/2 \rfloor}$ *for even n.*

Proof. The idea of proof is almost the same as the one of Lemma 3.19.

Let $n = 2m$. From Corollary 3.15, it suffices to show that $\pi \in P^H(2m)$ if and only if the 2-Motzkin path Z_π is semi-symmetric and has positive height at point i with $1 < i < 2m$.
($\Longrightarrow$) Let $(i, j), (k, i) \in \mathcal{L}_\pi$. Since $\pi \in P^H(2m)$, we have that $(2m - i + 1, 2m - j + 1)$ and $(2m - k + 1, 2m - i + 1)$ are also in $\mathcal{L}_\pi$. It is easy to see that (i, j) is positive if and only if $(2m - i + 1, 2m - j + 1)$ is negative. In the same way, we can see that (k, i) is positive if and only if $(2m - k + 1, 2m - i + 1)$ is negative. Thus, $z_i = -z_{2m-i+1}$.
($\Longleftarrow$) Clearly, $\pi \in P(2m)$. Let $(i, j) \in \mathcal{L}_\pi$. We show that $(2m - i + 1, 2m - j + 1)$ is also in $\mathcal{L}_\pi$. Without loss of generality, we assume that $i < j$, namely $(i, j) \in X$. Let i and j be the kth endpoints of lines in X, on L_1 and L_2, respectively. For $1 \le \ell < i$, the number of indices ℓ such that $z_\ell \in \{+1, +0\}$ is $k - 1$. Since Z_π is semi-symmetric, for $2m - i + 1 < \ell \le 2m$ the number of indices ℓ such that $z_\ell \in \{-1, -0\}$ is also $k - 1$. This implies that the point $2m - i + 1$ on L_1 is the $(|X| - k + 1)$th endpoint of a line in Y. Similarly, we can show that the point $2m - j + 1$ on L_2 is the $(|X| - k + 1)$th endpoint of a line in Y. Therefore, $(2m - i + 1, 2m - j + 1) \in \mathcal{L}_\pi$. □

Lemma 3.21. $\left|P^F(n)\right| = \binom{(n-2)/2}{\lfloor (n-2)/4 \rfloor}$ *for even n.*

Proof. Let $n = 2m$. From Theorem 3.1, it suffices to show that $\pi \in P^F(2m)$ if and only if the 2-Motzkin path Z_π is a symmetric Dyck path and has positive height at point i with $1 < i < 2m$. This is implied by the proofs of Lemmas 3.18 and 3.20. □

Lemmas 3.17, 3.18, 3.19, and 3.20, and Proposition 3.16 together show the number of elements of $\mathcal{B}_n$. We use a well-known relation $2\binom{2m-1}{m-1} = \binom{2m}{m}$ for the even case.

Theorem 3.22. *For $n \ge 2$, the number of connected bipartite permutation graphs of n vertices is given by*

$$|\mathcal{B}_n| = \begin{cases} \frac{1}{4}\left(C(n-1) + C(n/2-1) + \binom{n}{n/2}\right) & \text{if } n \text{ is even,} \\ \frac{1}{4}\left(C(n-1) + \binom{n-1}{(n-1)/2}\right) & \text{if } n \text{ is odd.} \end{cases}$$

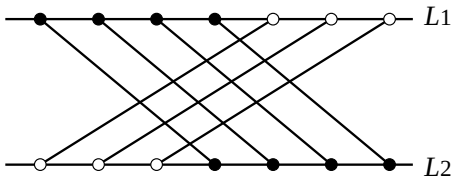


Figure 3.4: The root in $S_{4,3}$.

Theorem 3.23. *For any given positive integer n, a connected bipartite permutation graph with n vertices can be generated uniformly at random in $O(n)$ time and $O(n)$ space.*

Proof. Basically, using the same idea as random generation of a proper interval graph with Lemma 3.13, the algorithm generates a 2-Motzkin path uniformly at random, and outputs the corresponding graph. However, this straightforward algorithm does not generate a connected bipartite permutation graph uniformly at random since it does not consider symmetricalness of the graph. That is, comparing to an asymmetric graph, the chances of graphs with one symmetricalness and three symmetricalness are only a half and a quarter, respectively. Hence the algorithm adapts the probability as follows. From the lemma 3.17, the algorithm first chooses one of four sets $P(n)$, $P^H(n)$, $P^V(n)$,and $P^R(n)$ with probabilities $|P(n)| / |\mathcal{B}_n|$, $\left|P^H(n)\right| / |\mathcal{B}_n|$, $\left|P^V(n)\right| / |\mathcal{B}_n|$, and $\left|P^R(n)\right| / |\mathcal{B}_n|$, respectively. Next, in each case, the algorithm generates each element uniformly at random.
Case 1: Generation of an element of $P(n)$ uniformly at random. The algorithm simply picks up an element by generation a 2-Motzkin path same as Case 1 of Theorem 3.2.
Case 2: Generation of an element of $P^H(n)$ uniformly at random. The algorithm generates a semi-symmetric 2-Motzkin path. The algorithm first constructs the left half of the semi-symmetric 2-Motzkin path by using Case 2 of Theorem 3.2. Then the right half can be constructed from the left half since the resultant 2-Motzkin path has to be semi-symmetric.
Case 3: Generation of an element of $P^V(n)$ uniformly at random. The algorithm generates a 2-Motzkin path that consists of only $+1$ and -1, or consequently a Dyck path. Hence we can use the same algorithm of Theorem 3.2 Case 1.
Case 4: Generation of an element of $P^R(n)$ uniformly at random. This case is similar to Case 2. The algorithm first generates a nonnegative 2-Motzkin path of half length, and extends it to be symmetric. □

3.4 Enumeration of Bipartite Permutation Graphs

In this section we give an efficient algorithm to enumerate all bipartite permutation graphs of n vertices. Our algorithm can enumerate such graphs in O(1) time for each.

Our approach is to repeatedly enumerate all bipartite permutation graphs of the specified number of vertices. If we can enumerate all bipartite permutation graphs with $p = |X|$ and $q = |Y|$, such graphs of n vertices can be enumerated by repeating the method for each pair of $(p, q) = (\lceil \frac{n}{2} \rceil, \lfloor \frac{n}{2} \rfloor), (\lceil \frac{n}{2} \rceil + 1, \lfloor \frac{n}{2} \rfloor - 1), \ldots, (n-1, 1)$. By the above observation and Lemma 2.24, it is sufficient to enumerate all canonical representations of bipartite permutation graphs with $p = |X|$ and $q = |Y|$.

We first define the family tree among the set of canonical representations of bipartite permutation graphs. Then we give an algorithm to traverse the family tree efficiently.

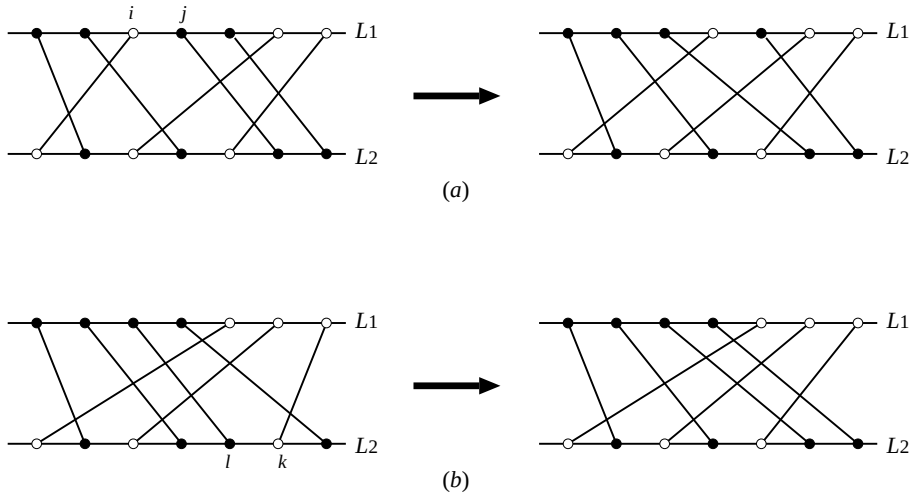


Figure 3.5: Examples of the parents.

We need some definitions. Let $S_{p,q}$ be the set of canonical representations of bipartite permutation graphs of p vertices in X and q vertices in Y. Since a graph corresponding to a representation in $S_{p,q}$ also corresponds to a representation in $S_{q,p}$, we assume $p \geq q$ without loss of generality. The *root*, denoted by $R_{p,q}$, in $S_{p,q}$ is the smallest representation in $S_{p,q}$, that is, $s(R_{p,q}) = [[\cdots[]]\cdots][[\cdots[]]\cdots]$. See Figure 3.4 for an example. As we will see, the root corresponds to the root vertex in a tree structure among $S_{p,q}$.

Let $\mathcal{L} = (L_1, L_2)$ be a representation in $S_{p,q} \setminus \{R_{p,q}\}$. Let $s(\mathcal{L}) = x_1 x_2 \cdots x_{2n}$. We denote $s_1(\mathcal{L}) = x_1 x_2 \cdots x_n$ and $s_2(\mathcal{L}) = x_{n+1} x_{n+2} \cdots x_{2n}$. Thus, let $s_1(R_{p,q}) = [[\cdots[]]\cdots]$, that consists of p '['s and q ']'s. Now we define "the parent" $P(\mathcal{L})$ of the representation $\mathcal{L}$ in $S_{p,q}$ as follows. We have two cases.

Case 1: $s_1(\mathcal{L}) \neq s_1(R_{p,q})$. Let i be the index of $s_1(\mathcal{L})$ such that $x_i =$ ']' and $x_{i'} =$ '[' for all $i' < i$, and j be the index of $s_1(\mathcal{L})$ such that $x_j =$ '[' and $x_{j'} =$ ']' for all $i \leq j' < j$. Then j is called *the swappable point* of $\mathcal{L}$. $P(\mathcal{L})$ is the representation obtained from $\mathcal{L}$ by swapping two endpoints at $j-1$ and j on L_1. See Figure 3.5(a).

Case 2: $s_1(\mathcal{L}) = s_1(R_{p,q})$. In this case we define $P(\mathcal{L})$ by swapping two endpoints on L_2. Let k be the index of $s_2(\mathcal{L})$ such that $x_k =$ '[' and $x_{k'} =$ ']' for all $k < k'$, and l be the index of $s_2(\mathcal{L})$ such that $x_l =$ ']' and $x_{l'} =$ '[' for all $l < l' \leq k$. Then l is called *the swappable point* of $\mathcal{L}$. $P(\mathcal{L})$ is the representation obtained from $\mathcal{L}$ by swapping two endpoints at l and $l+1$ on L_2. See Figure 3.5(b).

In both cases $P(\mathcal{L})$ is called *the parent* of $\mathcal{L}$ and $\mathcal{L}$ is called a *child* of $P(\mathcal{L})$. We can observe that $s(P(\mathcal{L}))$ is smaller than $s(\mathcal{L})$. The parent $P(\mathcal{L})$ of $\mathcal{L}$ in $S_{p,q} \setminus \{R_{p,q}\}$ is always defined, since there exists the swappable point of $\mathcal{L}$. The next lemma shows we finally obtain the root in $S_{p,q}$ by repeatedly finding the parent.

Lemma 3.24. *Let $\mathcal{L}$ be a representation in $S_{p,q} \setminus \{R_{p,q}\}$. The sequence obtained by repeatedly finding the parent ends up with the root $R_{p,q}$.*

Proof. For a representation $\mathcal{L}$ with $s(\mathcal{L}) = x_1 x_2 \cdots x_{2n}$, we define a potential function $f(\mathcal{L}) = \Sigma_{i=1}^{2n} 2^{2n-i} g(x_i)$, where $g($'['$) = 0$ and $g($']'$) = 1$. $f(\mathcal{L})$ is a mapping from $\mathcal{L}$ into non-negative integer. We can observe that $f(R_{p,q})$ is the smallest among values of representations in $S_{p,q}$.

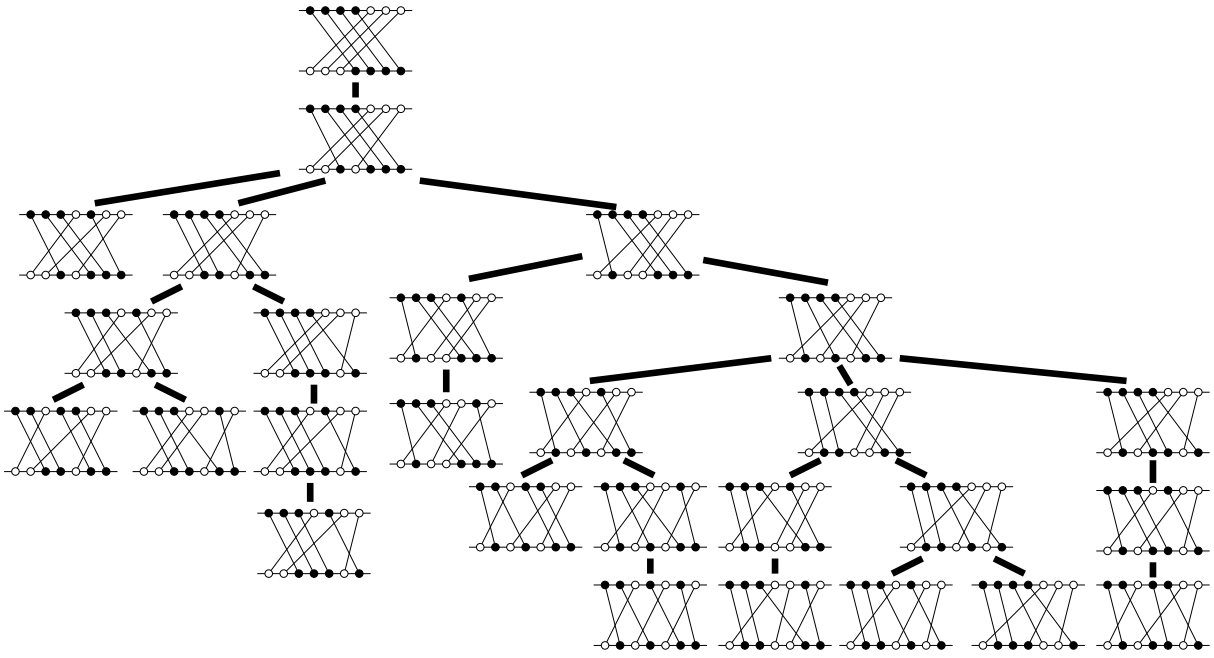


Figure 3.6: Family tree of $S_{4,3}$.

Let j be the swappable point of $\mathcal{L}$. In Case 1, we have $f(P(\mathcal{L})) = f(\mathcal{L}) - 2^{2n-(j-1)} + 2^{2n-j} = f(\mathcal{L}) - 2^{2n-j} < f(\mathcal{L})$ by the definition of the parent and the potential function. Similarly, in Case 2, we have $f(P(\mathcal{L})) = f(\mathcal{L}) - 2^{2n-(j+n)} + 2^{2n-(j+n+1)} = f(\mathcal{L}) - 2^{2n-(j+n)-1} < f(\mathcal{L})$. Therefore $f(P(\mathcal{L})) < f(\mathcal{L})$ holds. Since the parent of $\mathcal{L}$ is always defined for $\mathcal{L}$ in $S_{p,q} \setminus \{R_{p,q}\}$, we eventually obtain $R_{p,q}$ by repeatedly finding the parent of the derived representation, which completes the proof. □

By merging all these sequences we can have the *family tree* of $S_{p,q}$, denoted by $T_{p,q}$. The root vertex of $T_{p,q}$ corresponds to $R_{p,q}$, the vertices of $T_{p,q}$ correspond to representations in $S_{p,q}$ and each edge corresponds to a relation between a representation in $S_{p,q} \setminus \{R_{p,q}\}$ and its parent. See Figure 3.6 for an example.

Now we give an algorithm that enumerates all representations in $S_{p,q}$. The algorithm traverses a family tree and enumerates canonical representations corresponding to the vertices of the family tree. To traverse a family tree, we design finding all children of a given canonical representation.

We need some definitions. $\mathcal{L}_1[i]$ is the line representation obtained from $\mathcal{L}$ by swapping two endpoints at i and $i+1$ on L_1, and similarly $\mathcal{L}_2[i]$ is the line representation obtained from $\mathcal{L}$ by swapping two endpoints at $i-1$ and i on L_2. If $\mathcal{L} = P(\mathcal{L}_1[i])$ (and $\mathcal{L} = P(\mathcal{L}_2[i])$), we say i is a *nominated point* on L_1 (and L_2, respectively). $\mathcal{L}_1[i]$ (and $\mathcal{L}_2[i]$) is a child of $\mathcal{L}$ only if i is a nominated point on L_1 (and L_2) and $\mathcal{L}_1[i]$ (and $\mathcal{L}_2[i]$, respectively) is connected and canonical.

For a string $s(\mathcal{L}) = x_1 x_2 \cdots x_{2n}$, we define the *connectivity value* $c(i)$ for $i = 0, 1, \ldots, 2n$ as follows:

$$c(i) = \begin{cases} 0 & \text{if } i = 0, n \\ c(i-1) + 1 & \text{if } (x_i = \text{`['} \text{ and } i < n) \text{ or } (x_i = \text{`]'} \text{ and } i > n) \\ c(i-1) - 1 & \text{if } (x_i = \text{`]'} \text{ and } i < n) \text{ or } (x_i = \text{`['} \text{ and } i > n) \end{cases}$$

Intuitively, $c(i)$ for $i < n$ is the number of '['s minus the number of ']'s in $x_1 x_2 \cdots x_i$, and $c(i)$ for $i > n$ is the number of ']'s minus the number of '['s in $x_{n+1} x_{n+2} \cdots x_i$. We note that if

$c(i) = c(n+i)$ holds, then the bipartite permutation graph corresponding to $\mathcal{L}$ is disconnected. A bipartite permutation graph is connected if and only if we have $c(i) \neq c(n+i)$ for each $i = 1, 2, \ldots, n-1$. We say $\mathcal{L}$ is *connected* if $c(i) \neq c(n+i)$ for each $i = 1, 2, \ldots, n-1$.

A naive way to generate all children is as follows. We construct $\mathcal{L}_1[i]$ for each $i = 1, 2, \ldots, n-1$, then check whether or not (1) i is a nominated point on L_1, (2) $\mathcal{L}_1[i]$ is connected and (3) $\mathcal{L}_1[i]$ is canonical. If all conditions are satisfied, $\mathcal{L}_1[i]$ is a child. Similarly, we check whether or not $\mathcal{L}_2[i]$ is a child for each $i = 2, 3, \ldots, n$. This method takes much running time to generate all children.

To improve the running time, we first show that the list of nominated points can be maintained efficiently and next propose an efficient method to check whether or not a canonical representation is connected and canonical.

Lemma 3.25. *Let $\mathcal{L} = (L_1, L_2)$ be a representation in $S_{p,q}$. There exist at most 3 nominated points on L_1 and L_2.*

Proof. Let $s(\mathcal{L}) = x_1 x_2 \cdots x_{2n}$. We consider the following two cases.

Case 1: $s_1(\mathcal{L}) \neq s_1(R_{p,q})$. Let i be the index of $s_1(\mathcal{L})$ such that $x_i =$ ']' and $x_{i'} =$ '[' for all $i' < i$. Then $i-1$ is a nominated point on L_1. Let j be the index of $s_1(\mathcal{L})$ such that $x_j =$ '[' and $x_{j'} =$ ']' for all $i \leq j' < j$. If $x_{j+1} =$ ']' holds, then j is a nominated point. Other points on L_1 are not nominated points and there is no nominated point on L_2.

Case 2: $s_1(\mathcal{L}) = s_1(R_{p,q})$. Clearly we have one nominated point p on L_1, where p is equal to the number of '['s in $x_1 x_2 \cdots x_n$. Now we consider nominated points on L_2. Let k be the index of $s_2(\mathcal{L})$ such that $x_k =$ '[' and $x_{k'} =$ ']' for all $k < k'$. Then $k+1$ is a nominated point on L_2. Let l be the index of $s_2(\mathcal{L})$ such that $x_l =$ ']' and $x_{l'} =$ '[' for all $l < l' \leq k$. If $x_{l-1} =$ '[' holds, then l is a nominated point on L_2. Other points on L_2 are not nominated. □

Lemma 3.26. *Given $\mathcal{L}$ and its nominated points, we can construct the list of nominated points of each child in $O(1)$ time.*

Proof. We first consider the nominated points on L_1. Let n_1, n_2 ($n_1 < n_2$) be two nominated points on L_1. We consider each case of $\mathcal{L}_1[n_1]$ and $\mathcal{L}_1[n_2]$.

Case 1: $\mathcal{L}_1[n_1]$. If $x_{n_1+2} =$ '[' then $n_2 = n_1 + 2$ holds or $\mathcal{L}$ has only one nominated point n_1. In this case $\mathcal{L}_1[n_1]$ has one nominated point $n_1 - 1$ on L_1. Otherwise, $x_{n_1+2} =$ ']', $\mathcal{L}_1[n_1]$ has two nominated points $n_1 - 1$ and $n_1 + 1$ on L_1. $\mathcal{L}_1[n_1]$ has no nominated point on L_2.

Case 2: $\mathcal{L}_1[n_2]$. If $x_{n_2+2} =$ '[', then $\mathcal{L}_1[n_2]$ has one nominated point n_1. Otherwise, $x_{n_2+2} =$ ']', $\mathcal{L}_1[n_2]$ has two nominated points n_1 and $n_2 + 1$.

Therefore each nominated point of $\mathcal{L}_1[n_1]$ and $\mathcal{L}_2[n_2]$ (1) appears in the previous or next point of n_1 or n_2, (2) disappears from the list, or (3) is identical to one of $\mathcal{L}$'s.

The case on L_2 is symmetric and hence omitted. □

Now we have the algorithm in **Algorithm 3**, that generates all children of a given representation $\mathcal{L}$. For each nominated point i on L_1 (and L_2), it first checks whether $\mathcal{L}_1[i]$ (and $\mathcal{L}_2[i]$) is connected and canonical, and then recursively calls it for $\mathcal{L}_1[i]$ (and $\mathcal{L}_2[i]$, respectively) if it satisfies the conditions. Starting at the root in $S_{p,q}$, by calling the algorithm recursively, we can traverse the family tree $T_{p,q}$ and generate all representations in $S_{p,q}$.

By Lemma 3.26, steps 3 and 6 can be done in O(1) time in each recursive call. The remaining task is checking whether or not $\mathcal{L}$ is connected and canonical efficiently.

Algorithm 3: `find-all-child-rep`

Input: line representation $\mathcal{L}$

```
1 begin
2 |  Output L;
3 |  foreach nominated point i on L_1 do
4 |  |  if L_1[i] is connected and canonical then find-all-child-rep(L_1[i]).
5 |  end
6 |  foreach nominated point i on L_2 do
7 |  |  if L_2[i] is connected and canonical then find-all-child-rep(L_2[i]).
8 |  end
9 end
```

We first consider the check of connectivity of a representation. By symmetry we only consider $\mathcal{L}_1[i]$ without loss of generality. Assume $\mathcal{L}$ is connected. Then $\mathcal{L}_1[i]$ is connected only if $c(i) \neq c(n+i)$ and $c(i+1) \neq c(n+i+1)$. We can check such conditions in O(1) time using an array of size $2n$ to maintain the sequences of connectivity values of $\mathcal{L}_1[i]$. Update of the array also can be done in O(1) time. Therefore, the connectivity of $\mathcal{L}_1[i]$ can be checked in O(1) time.

Next we check whether or not $\mathcal{L}$ is canonical. When $p \neq q$, $s(\mathcal{L})$ is canonical if $s(\mathcal{L})$ is the smallest string among $s(\mathcal{L}^V)$, $s(\mathcal{L}^H)$ and $s(\mathcal{L}^R)$. If $p = q$, we need more discussions. Let $\mathcal{L}$ be a representation in $S_{p,q}$ and G be the bipartite permutation graph corresponding to $\mathcal{L}$. Then there exists a line representation $\mathcal{L}'$ obtained from $\mathcal{L}$ by swapping lines corresponding to vertices in X and ones in Y. Similarly, we denote by $\mathcal{L}^{V'}$, $\mathcal{L}^{H'}$, $\mathcal{L}^{R'}$ the representations obtained from $\mathcal{L}^V$, $\mathcal{L}^H$, $\mathcal{L}^R$ by swapping lines corresponding to vertices in X and ones in Y, respectively. Then $\mathcal{L}$ is canonical if and only if $s(\mathcal{L})$ is the smallest string among $s(\mathcal{L}^V)$, $s(\mathcal{L}^H)$, $s(\mathcal{L}^R)$, $s(\mathcal{L}')$, $s(\mathcal{L}^{V'})$, $s(\mathcal{L}^{H'})$ and $s(\mathcal{L}^{R'})$.

If we can check whether given two strings $s(\mathcal{L})$ and $s(\mathcal{I})$ for any $\mathcal{I} \in \{\mathcal{L}^H, \mathcal{L}^V, \mathcal{L}^R, \mathcal{L}', \mathcal{L}^{V'}, \mathcal{L}^{H'}, \mathcal{L}^{R'}\}$ satisfy $s(\mathcal{L}) < s(\mathcal{I})$, then we can check whether $s(\mathcal{L})$ is canonical by applying the method for each pair of $s(\mathcal{L})$ and other strings.

Lemma 3.27. *One can determine whether or not $\mathcal{L} = (L_1, L_2)$ is canonical in $O(1)$ time.*

Proof. Let $s(\mathcal{L}) = x_1 x_2 \cdots x_{2n}$ and $s(\mathcal{I}) = y_1 y_2 \cdots y_{2n}$ for any $\mathcal{I} \in \{\mathcal{L}^H, \mathcal{L}^V, \mathcal{L}^R, \mathcal{L}', \mathcal{L}^{V'}, \mathcal{L}^{H'}, \mathcal{L}^{R'}\}$. We maintain a doubly linked list L in order to check $s(\mathcal{L}) < s(\mathcal{I})$ in O(1) time. The list L maintains the indices of different characters in $s(\mathcal{L})$ and $s(\mathcal{I})$. L is empty if and only if $s(\mathcal{L}) = s(\mathcal{I})$. We can check whether $s(\mathcal{L}) < s(\mathcal{I})$ by comparing $x_{L[1]}$ and $y_{L[1]}$, where $L[i]$ is the ith element in L.

The update of L is as follows. Let n_1, n_2 be the nominated points on L_1 of $\mathcal{L}$. We maintain i such that $L[i] \leq n_1 < L[i+1]$ and j such that $L[j] \leq n_2 < L[j+1]$. It is easy to see we can update L using i and j in O(1) time. Since nominated point n_1 (and n_2) is updated by n_1 or $n_1 - 1$ (and $n_1 + 1$ or $n_2 + 1$, respectively) by Lemma 3.25, i and j can be updated in O(1) time. The case on L_2 is similar and hence omitted. □

Therefore steps 4 and 7 in **Algorithm 3** can be computed in O(1) time.

Lemma 3.28. *Our algorithm uses $O(n)$ space and runs in $O(|S_{p,q}|)$ time.*

By Lemma 3.28, our algorithm generates each representation in O(1) time "on average". **Algorithm 3** may return from the deep recursive calls without outputting any representation

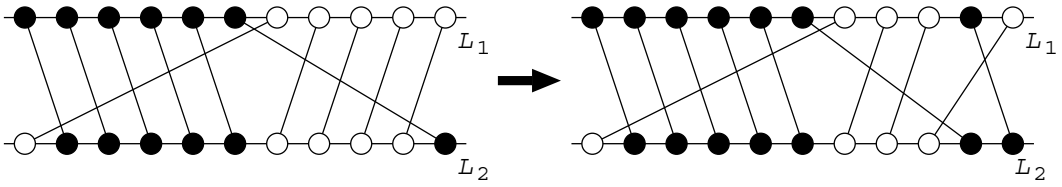


Figure 3.7: Construction of a representation in $S_{7,4}$ from the jump representation in $S_{6,5}$.

after generating a representation corresponding to the leaf of a large subtree in the family tree. This takes much running time. Therefore each representation cannot be generated in O(1) time in worst case. This delay can be canceled by outputting the representations in the "prepostorder" in which representation are outputted in the preorder (and postorder) at the vertices of odd (and even, respectively) depth of a family tree. In such manner delay process can be bounded by at most 3 edge traversals of a family tree.

Lemma 3.29. *After outputting the root in $O(n)$ time, our algorithm enumerates every representation in $S_{p,q}$ in $O(1)$ time in worst case.*

Now we consider to enumerate all canonical representations corresponding to bipartite permutation graphs of n vertices. By applying Lemma 3.29 for each $(p,q) = (\lceil \frac{n}{2} \rceil, \lfloor \frac{n}{2} \rfloor), (\lceil \frac{n}{2} \rceil + 1, \lfloor \frac{n}{2} \rfloor - 1), \ldots, (n-1, 1)$ in this order, we can enumerate all such representations. Every representation which is not the root is generated in O(1) time. However, the root in $S_{p,q}$ is not constructed from the last outputted representation in $S_{p-1,q+1}$ in O(1) time.

This delay can be canceled as follows. Let $\mathcal{L} = (L_1, L_2)$ be a representation in $S_{p,q}$. Then $\mathcal{L}$ is *jump representation* if $s_1(\mathcal{L}) = s_1(R_{p,q})$ and $s_2(\mathcal{L}) = []]\cdots][[\cdots[]$. See Figure 3.7. When jump representation in $S_{p,q}$ is generated, we construct a representation $\mathcal{K}$ in $S_{p+1,q-1}$ by swapping the three lines (p,n), $(n-1,n-2)$, $(n,n-1)$ to $(p,n-1)$, $(n-1,n)$, $(n,n-2)$, respectively. See Figure 3.7. We note that the line $(n-1,n-2)$ is switched to a line corresponding to a vertex in X, and $\mathcal{K}$ can be generated from $\mathcal{L}$ in O(1) time. Then we enumerate all representations in $S_{p+1,q-1}$ by traversing $T_{p+1,q-1}$ as follows. After $\mathcal{K}$ is generated, the descendants of $\mathcal{K}$ in $T_{p+1,q-1}$ are enumerated by **Algorithm 3**, and we construct $P(\mathcal{K})$. Then we traverse the descendants of $P(\mathcal{K})$ except the subtree rooted at $\mathcal{K}$ and construct $P(P(\mathcal{K}))$. We repeat this process until the root is generated. We note that $P(\mathcal{K})$ can be generated in O(1) time by maintaining the swappable point and its data structures can be updated in O(1) time. Therefore we have the following theorem.

Theorem 3.30. *After outputting the root in $S_{\lceil \frac{n}{2} \rceil, \lfloor \frac{n}{2} \rfloor}$, one can enumerate every canonical representation of a bipartite permutation graph of n vertices in $O(1)$ time.*

We note that (1) swapping two endpoints of a canonical representation corresponds to adding or removing one edge in the corresponding graph and (2) a graph can be constructed from the graph corresponding to a jump representation by a constant number of operations to add and remove edges. We have the following theorem.

Theorem 3.31. *The algorithm enumerates every connected bipartite permutation graph of n vertices in $O(1)$ time.*

Chapter 4

Reconstruction

4.1 Interval Graphs

In this section, we propose reconstruction algorithms for interval graphs. First, we present a deck checking algorithm for interval graphs. Our reconstruction algorithm enumerates the preimage candidates, and checks whether each candidate is really a preimage of the input deck. Thus the deck checking algorithm is one of the basic part of reconstruction algorithm.

Our algorithm outputs preimages that are interval graphs. However it is possible that a non-interval graph has a deck that consists of interval graphs, though it is exceptional. Since considering this case all the time in the reconstruction algorithm makes it complex, we attempt to get done with this special case.

Next, we present reconstruction algorithms for interval graph preimage. Our algorithms have two cases that connected preimage case and disconnected preimage case. First, we propose the algorithm for connected case by using the compact interval representation. Then we consider the disconnected preimage case. If we use the connected case algorithm in the disconnected case directly, we have to consider many cases. Our algorithm for the disconnected case transforms the input graphs to the multi-set of connected preimage, then we use the connected case algorithm.

4.1.1 Deck Checking

First of this subsection, we introduce a famous theorem below.

Theorem 4.1 (Lueker and Booth [36])**.** *Given two interval graphs G_1 and G_2, we can determine whether they are isomorphic in $O(n + m)$ time, where n is the number of vertices of G_1 and G_2, and m is the number of edges of G_1 and G_2.*

We next show following lemma for deck checking algorithm.

Lemma 4.2. *Given an interval graph G which can be connected or disconnected, the graph $\tilde{G}$ obtained by adding one universal vertex to the graph G is always a connected interval graph.*

Proof. It is obvious that $\tilde{G}$ is connected. Consider a compact interval representation $\mathcal{I}$ of G. Let K be the length of $\mathcal{I}$. Then $\mathcal{I} \cup \{[0, K]\}$ is an interval representation of $\tilde{G}$. Therefore $\tilde{G}$ is a connected interval graph. □

We have the following theorem and **Algorithm 4**.

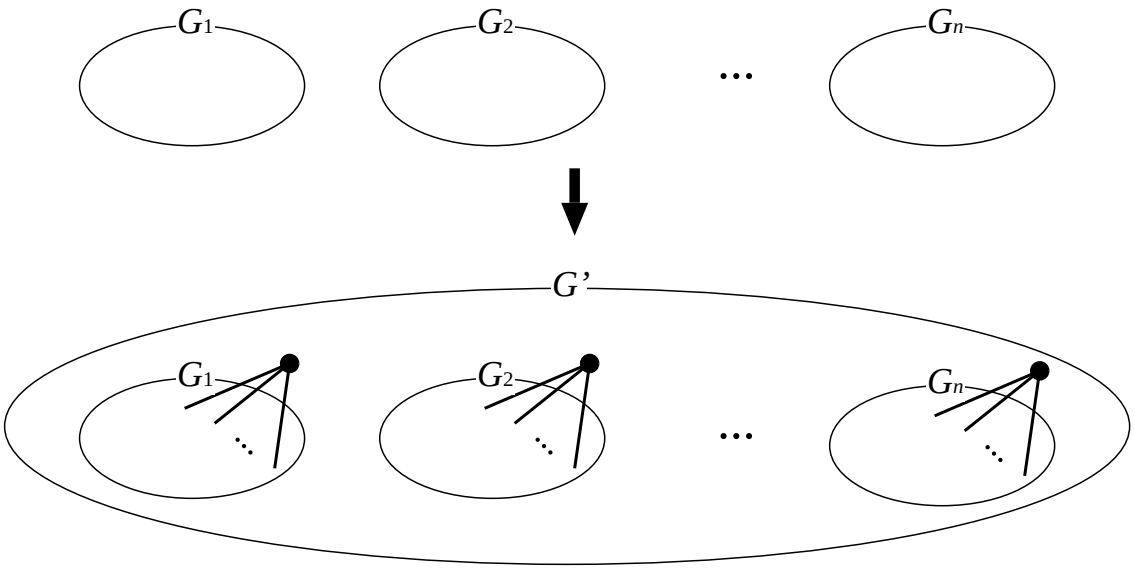


Figure 4.1: Constructing graph G' from candidate graph G for deck checking

Theorem 4.3. *There is an $O(n(n+m))$ time algorithm of deck checking for n-vertex m-edge graph and its deck (or a deck candidate) that consists of interval graphs.*

Proof. Let $G = (V, E)$ be a graph, where V is $\{1, 2, \ldots, n\}$, and $|E|$ is equal to m. Let G_i ($i \in V$) be a graph obtained by removing vertex i from G. Suppose that $G_1, G_2, \ldots, G_n$ are interval graphs. It is clear that $\{G'_1, G'_2, \ldots, G'_n\}$ is a deck of G if and only if the multi-set $\{\tilde{G}_1, \tilde{G}_2, \ldots, \tilde{G}_n\}$ is equal to the multi-set $\{\tilde{G}'_1, \tilde{G}'_2, \ldots, \tilde{G}'_n\}$. Hence we can determine whether or not the given multi-set $D = \{G'_1, G'_2, \ldots, G'_n\}$ is a deck of the input graph G by checking whether or not $G' = \tilde{G}_1 \dot{\cup} \tilde{G}_2 \dot{\cup} \ldots \dot{\cup} \tilde{G}_n$ is isomorphic to $\tilde{G}'_1 \dot{\cup} \tilde{G}'_2 \dot{\cup} \ldots \dot{\cup} \tilde{G}'_n$. Since the disjoint union of two interval graphs is an interval graph, we can use well-known linear time isomorphism algorithm [36] for this checking. We describe the algorithm in **Algorithm 4**. Since the number of vertices of $\tilde{G}_1 \dot{\cup} \ldots \dot{\cup} \tilde{G}_n$ is $O(n^2)$, and since the number of edges of $\tilde{G}_1 \dot{\cup} \ldots \dot{\cup} \tilde{G}_n$ is $O(mn + n^2)$, the time complexity of this algorithm is $O(n(n+m))$. □

Algorithm 4: `deck-checking`

Input: graph $G = (V, E)$, multi-set $D = \{G'_1, G'_2, \ldots, G'_n\}$

```
1 begin
2 |  Let G' be an empty graph.
3 |  foreach vertex v ∈ V do  G' = G' ∪̇ (G̃ − v);
4 |  if G' is isomorphic to G̃'_1 ∪̇ G̃'_2 ∪̇ ··· ∪̇ G̃'_n then return True.
5 |  else return False.
6 end
```

4.1.2 Non-interval Graph Preimage Case

Note that we can easily prove that all the members in a deck of an interval graph are interval graphs from Theorem 2.4.

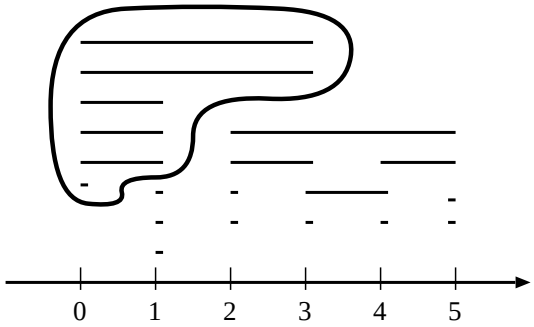


Figure 4.2: Vertices corresponding to the enclosed intervals are end-vertex set.

Theorem 4.4. *If n interval graphs $G_1, G_2, \ldots, G_n$ have a preimage G that is not an interval graph, we can reconstruct G from $G_1, G_2, \ldots G_n$ in $O(n^2)$ time.*

Proof. Assume that $G_1, G_2, \ldots, G_n$ are the deck of G, and G is not an interval graph. Then G must be one of the graphs described in Figure 2.2, since any graph that is obtained by removing a vertex from G is an interval graph (containing none in Figure 2.2). It is clear that $G_1, G_2, \ldots, G_n$ have the same number of vertices, $n-1$, and the number of vertices in G is n. Since the number of graphs of size n in Figure 2.2 is O(1), we can check if one of them is a preimage of the input graphs in polynomial time with deck checking algorithm. The time complexity is $O(n(n+m))$ from Theorem 4.3, where m is the number of edges of a preimage. Since the numbers of edges in (*a*), (*b*), (*c*), (*d*), and (*e*) are $O(n)$, the time complexity is definitely $O(n^2)$. □

Therefore we concentrate on an algorithm that tries to reconstruct an interval graph whose deck is the set of the input graphs in the remaining subsections.

4.1.3 Connected Preimage Case

First we define *end-vertex set*. The end-vertex set is intuitively a set of vertices whose corresponding intervals are at the left end in at least one interval representation. Our algorithm adds a vertex adjacent to all the vertices in an end-vertex set of an interval graph in the input deck. This enables us to avoid exponential times' constructions of preimage candidates.

Definition 4.5. *For an interval graph $G = (V, E)$, we call a vertex subset $S \subset V$ an* end-vertex set *if and only if in some compact interval representation of G all the coordinates of the left endpoints of intervals corresponding to vertices in S are 0, and S is maximal among such vertex subsets with respect to the interval representation.*

See Figure 4.2 for example. It is clear from the definition of compact interval representations that an end-vertex set contains at least one simplicial vertex.

We show some lemmas about end-vertex sets. We can estimate that the number of essentially different preimage candidates is $O(n^2)$ by these lemmas.

Lemma 4.6. *Let S be an end-vertex set of an interval graph $G = (V, E)$. If two vertices v and w in S have the same degree, then $N[v]$ is equal to $N[w]$.*

Proof. Since v and w is in S, on some compact representation of G, the interval corresponding to v is $I_v = [0, k_v]$, and the interval corresponding to w is $I_w = [0, k_w]$ for some k_v and k_w. Assume that k_v is not equal to k_w. We can assume that k_v is greater than k_w without loss of generality.

Then $N[w] \subsetneq N[v]$ holds due to the definition of a compact representation. This contradicts the fact that v and w have the same degree (see Figure 4.2 for the better understanding). □

Lemma 4.7. *A connected interval graph has at most $O(n)$ end-vertex sets.*

Proof. An end-vertex set of an interval graph G is in the form $\{I \in \mathcal{I} \mid 0 \in I\}$ for some interval representation $\mathcal{I}$ of G. Thus, from Lemmas 2.6 and 2.8, there are at most $O(n)$ end-vertex sets for G. □

Now we refer the well-known lemma about the degree sequence.

Lemma 4.8 (Kelly's Lemma [28])**.** *We can compute the degree sequence of a preimage of the input n graphs in $O(n)$ time, if we know the number of edges in each input graph.*

Proof. Let $G_1, G_2, \ldots, G_n$ be the input graphs. Assume that graph G has a deck $\{G_1, G_2, \ldots, G_n\}$. Then there are vertices $v_1, v_2, \ldots, v_n$ such that G_i is obtained by removing v_i from G for each i in $\{1, 2, \ldots, n\}$. Thus

$$\deg(G_i) = \deg(G) - 2\deg(v_i)$$

holds for each $i \in \{1, 2, \ldots, n\}$. Hence we have

$$\deg(G) = \frac{\sum_{i=1}^{n} \deg(G_i)}{n-2}.$$

Therefore we can easily calculate the degree sequence of G, i.e., $(\deg(G)-\deg(G_1))/2, (\deg(G)-\deg(G_2))/2, \ldots, (\deg(G) - \deg(G_n))/2$.

We can calculate $\deg(G_i)$ in constant time, provided we know the number m_i of edges in G_i, for $\deg(G_i)$ is equal to $2m_i$. Thus the time complexity to calculate $\deg(G)$ is $O(n)$, and the total time complexity to obtain the degree sequence of G is also $O(n)$.

□

Now we present an algorithm for reconstructing a connected interval graph. Suppose that an n-vertex connected interval graph G has a deck of interval graphs $\{G_1, G_2, \ldots, G_n\}$. Let $\mathcal{I}$ be a compact interval representation of G. There must be an index $i \in \{1, \ldots, n\}$ such that G_i is obtained by removing a simplicial vertex s in the end-vertex set S corresponding to $\mathcal{I}$. We show that we can reconstruct G from G_i. Moreover we can check whether or not G_j is G_i for every $j \in \{1, ..., n\}$. Therefore we can reconstruct G by checking if G_j is the desired G_i for every $j \in \{1, \ldots, n\}$.

There are two cases about the number of simplicial vertices in S.

(i) S has only one simplicial vertex s.

(ii) S has at least two simplicial vertices.

In the case (ii), $S \setminus \{s\} = N_G(s)$ is still an end-vertex set of G_i (see Figure 4.3). In the case (i), $S \setminus \{s\}$ is contained by vertices corresponding to $\{I \in \mathcal{I} \mid 1 \in I\}$ which is an end-vertex set of G_i (see Figure 4.4). In both the cases, we thus have to add a simplicial vertex to an end-vertex set of G_i in order to reconstruct G. We denote the end-vertex set of G_i by $\tilde{S}$ ($S \setminus \{s\} \subset \tilde{S}$ holds). Since there are $O(n)$ end-vertex sets of G_i by Lemma 4.7, checking if each end-vertex set S' in G_i is $\tilde{S}$ takes $O(n)$ times iterations. Checking whether an end-vertex set S' is $\tilde{S}$ is simple. Since if S' is $\tilde{S}$, we can reconstruct G, we simply try to reconstruct G. If we can reconstruct G, S' is $\tilde{S}$, and we of course obtain G. Otherwise, S' is not $\tilde{S}$.

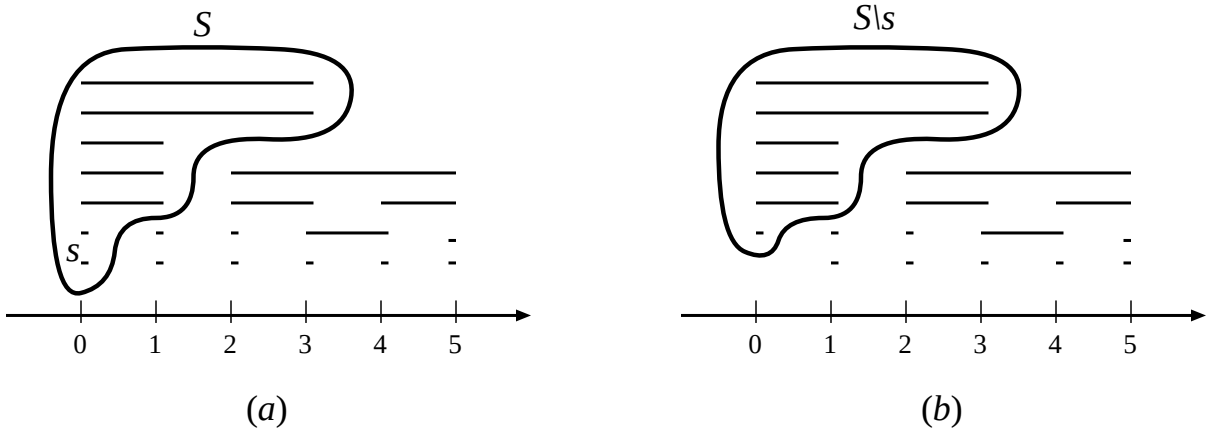


Figure 4.3: Compact interval representations of G and $G - s$. In $G - s$, $S \setminus s$ is end-vertex set.

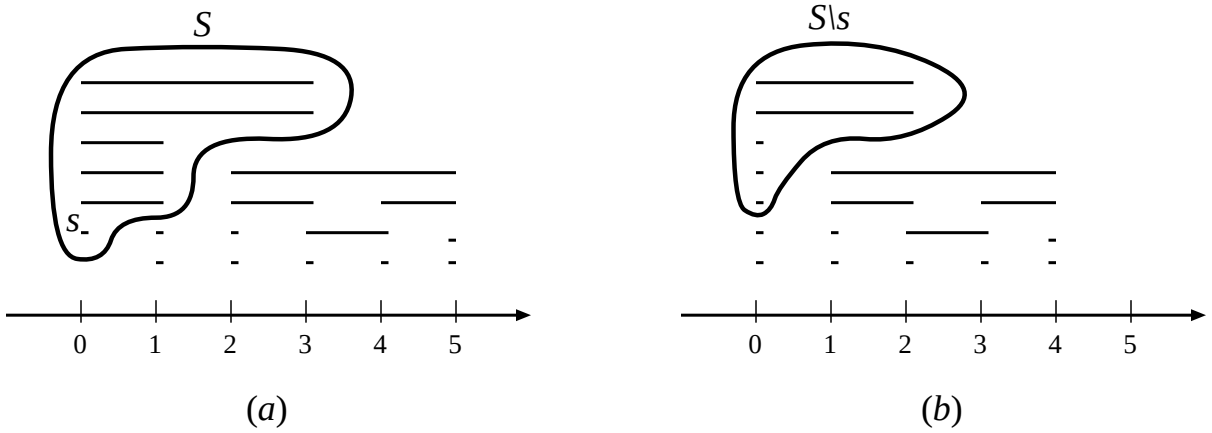


Figure 4.4: Compact interval representations of G and $G - s$. In $G - s$, $S \setminus s$ is not end-vertex set.

Now we explain how to reconstruct G from S' that is a candidate of $\tilde{S}$. Let $\mathcal{I}'$ be an interval representation of G_i whose corresponding end-vertex set is S'. Note that $\mathcal{I}'$ is easily obtained in $O(n + m)$ time by using the data structure called $\mathcal{MPQ}$-tree [32]. Since $S \setminus \{s\} \subset \tilde{S}$ holds, $S \setminus \{s\} \subset S'$ holds, if S' is $\tilde{S}$. Hence we can obtain an interval representation of G by extending intervals corresponding to vertices in $S \setminus \{s\} \subset S'$ to the left by one and adding an interval $[-1, -1]$. If we know $S \setminus \{s\}$, we can obtain G, since G has the interval representation obtained from $\tilde{\mathcal{I}}$ by extending intervals corresponding to vertices in $S \setminus \{s\}$ to the left by one and adding an interval $[-1, -1]$ (see Figure 4.5). In order to specify $S \setminus \{s\}$ in the polynomial time, we show the following lemma.

Lemma 4.9. *Let G be an interval graph. Let S be an end-vertex set of G, and let $\mathcal{I}$ be an compact interval representation of G whose corresponding end-vertex set is S. Let S_1 and S_2 be subsets of S such that the degree sequence of vertices in S_1 and the degree sequence of vertices in S_2 are the same. Let G_1 be a graph whose interval representation is obtained by $\mathcal{I}$ extending interval corresponding to S_1 to the left by one and adding an interval $[-1, -1]$, and let G_2 be a graph whose interval representation is obtained by $\mathcal{I}$ extending interval corresponding to S_2 to the left by one and adding an interval $[-1, -1]$. Then G_1 is isomorphic to G_2.*

Proof. The neighbor sets of S_1 and S_2 are the same due to Lemma 4.6. Hence G_1 and G_2 are

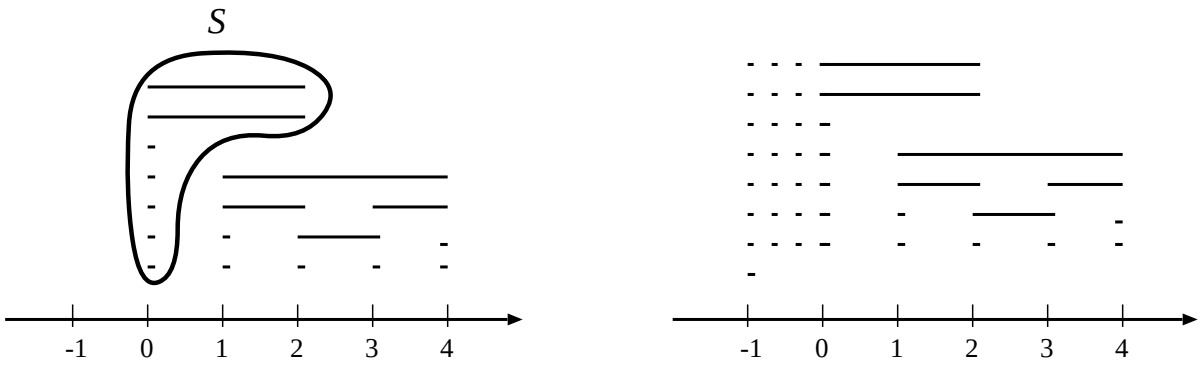


Figure 4.5: Adding an interval $[-1,-1]$

isomorphic. □

Since we know the degree sequence of G_i, and we can know the degree sequence of G by Lemma 4.8, we can know the degree sequence of $S \setminus \{s\}$. We denote the degree sequence by $(d_1, d_2, \ldots, d_l)$. Now we can specify $S \setminus \{s\} \subset S'$; $S \setminus \{s\}$ is the subset of S' such that whose degree sequence in G_i is $(d_1 - 1, d_2 - 1, \ldots, d_l - 1)$. Note that there may be exponentially many subsets of S' whose degree sequences in G_i are $(d_1 - 1, d_2 - 1, \ldots, d_l - 1)$. However Lemma 4.9 guarantees that any of such subsets can be $S \setminus \{s\}$, i.e. all the graphs reconstructed under the assumption that some subset of S' whose degree sequence is $(d_1 - 1, d_2 - 1, \ldots, d_l - 1)$ are $S \setminus \{s\}$ are isomorphic to each other. Therefore we can specify $S \setminus \{s\}$ in S'. To be more precise, if we can find such $S \setminus \{s\}$, S' is the desired $\tilde{S}$, and we can thus reconstruct G. The whole algorithm is described in **Algorithm 5**.

Algorithm 5: `connected-interval-preimage`

Input: multi-set $D = \{G_1, G_2, \cdots, G_n\}$

```
1  begin
2      foreach G_i ∈ D do
3          foreach end-vertex set S' of G_i do
4              Let I' be an interval representation of G_i whose corresponding end-vertex set
               is S'.
5              Compute the degree sequence (d_1, ··· , d_l) of S \ {s};
6              Let S* be a subset of S' whose degree sequence in G_i is (d_1 − 1, ··· , d_l − 1).
7              if there does not exist such S* then go to the next iteration;
8              Let G* be an interval graph whose interval representation is obtained from I'
               by extending interval corresponding to vertices in S* to the left by one and
               adding an interval [−1, −1].
9              if deck-checking(G*, D)=True then Output G*.
10         end
11     end
12     if output no graph then return No.
13 end
```

Now we consider the time complexity of this algorithm. For each G_i, construction of an $\mathcal{MPQ}$-tree of G_i in $O(n + m)$ time helps us to list each S' and $\mathcal{I}'$ in $O(n)$ time. Computing

the degree sequence $(d_1, d_2, \ldots, d_l)$ takes $O(n)$ time from Lemma 4.8. Since obtaining S^* needs sorting of the degree sequence, it requires $O(n \log n)$ time. It is clear that reconstructing an interval graph from its interval representation takes $O(n + m)$ time, if the endpoints of intervals are sorted. deck checking algorithm costs $O(n(n + m))$. Therefore the total time complexity of this algorithm is $O(n((n+m) + n(n+m+n\log n + n(n+m)))) = O(n^3(n+m))$. Note that we have to check every output preimage is not isomorphic to each other for preimage counting. Since the number of output preimage may be $O(n^2)$, we need $O(n^4(n + m))$ time for this checking. If the graph reconstruction conjecture is true, the time complexity of this checking can be omitted.

Theorem 4.10. *There is a polynomial time algorithm that lists up connected interval graphs that are preimages of the input n interval graphs. The time complexity for outputting one connected interval graph is $O(n^3(n + m))$, and that for outputting all is $O(n^4(n + m))$.*

4.1.4 Disconnected Preimage Case

Consider the case that the input graphs $G_1, G_2, \ldots, G_n$ have a disconnected preimage G. Then from the argument in the Theorem 4.4, G must be an interval graph. Further it is proven that the graph reconstruction conjecture is true in this case [28] (note that this fact does not imply that the reconstruction can be done in polynomial time). Lemma 4.2 and the fact that $\{G_1, G_2, \ldots, G_n\}$ is a deck of G if and only if $\{\tilde{G}_1, \tilde{G}_2, \ldots, \tilde{G}_n\} \cup \{G\}$ is a deck of $\tilde{G}$ simplify our algorithm in this case.

Since we can know the degree sequence of G by Lemma 4.8, we can know the degree sequence of $\tilde{G}$ by Lemma 4.8. Thus we can obtain $\tilde{G}$ by the algorithm described in the previous subsection. Note that we do not know G, thus in fact we cannot use the algorithm itself. However in the algorithm we can omit the case that G_i in the algorithm is G, since every interval graph has at least two end-vertex set and so does $\tilde{G}$. Further we can omit checking if G is in the deck of $\tilde{G}$. If the new algorithm (omitting checking if G is in the deck of $\tilde{G}$) returns some $\tilde{G}$, we can construct G from it. Then now we can check if G is a preimage of $G_1, \ldots, G_n$. Therefore we have the theorem below.

Theorem 4.11. *There is a polynomial time algorithm that outputs a disconnected interval graph that is the preimage of the input n interval graphs, if there exists. The time complexity of the algorithm is $O(n^3(n + m))$.*

Therefore we have the following theorem from Theorems 4.4, 4.10, and 4.11.

Theorem 4.12. *There are $O(n^3(n + m))$ time algorithms for legitimate deck and preimage construction, and there is an $O(n^4(n + m))$ time algorithm for preimage counting, where n is the number of vertices of preimage and m is the number of edges of preimage.*

4.2 Permutation Graphs

In this section, we propose a reconstruction algorithm for permutation graphs. The framework of this section is the same as previous section. First, we present a deck checking algorithm for permutation graphs. Since an $O(n^2)$ time isomorphism algorithm for permutation graphs [49] is known, developing a polynomial time deck checking algorithm for permutation graphs is not very difficult. Next, we consider the non-permutation preimage case to simplify our algorithms. Finally, we present the algorithms for permutation graph preimage. Our algorithms have two

parts. One is for a preimage G that has a minimal strong multi-vertex module M such that $G[M]$ is not critical, and the other part is for otherwise. In both the parts, we construct polynomially many candidates of a preimage, and use deck checking algorithm to check whether each candidate is a preimage. Since we of course do not know the properties of a preimage when we are given a input deck, we execute both these two parts for the input deck.

4.2.1 Deck Checking

First of this subsection, we introduce a famous theorem below.

Theorem 4.13 (Spinrad and Valdes[49])**.** *Given two permutation graphs G_1 and G_2, we can determine whether they are isomorphic in $O(n^2)$ time, where n is the number of vertices of G_1 and G_2.*

Thus developing a polynomial time algorithm for deck checking for permutation graphs is easy.

We present a deck checking algorithm. This algorithm is the same as **Algorithm 4**. Given a multi-set D that consists of permutation graphs, and given a preimage candidate $G = (V, E)$ whose deck consists of permutation graphs, we first prepare the deck $\hat{D}$ of G in $O(n(n + m))$ time, where n is the number of vertices of G and m is the number of edges of G. We then add a universal vertex to every graph in D and $\hat{D}$ in order to make each graph connected. Note that for any permutation graph G, $\tilde{G}$ is also a permutation graph. Since the disjoint union of permutation graphs is clearly a permutation graph, we can check if D and $\hat{D}$ are isomorphic in $O((n(n+1))^2)$=$O(n^4)$ time by applying the isomorphism algorithm for permutation graphs to the disjoint union of graphs in D and the disjoint union of graphs in $\hat{D}$. Now we obtain the theorem below.

Theorem 4.14. *There is an $O(n^4)$ time deck checking algorithm for a deck that consists of permutation graphs, and a preimage candidate $G = (V, E)$ whose deck consists of n permutation graphs.*

4.2.2 Non-permutation Graph Preimage Case

Let $D = \{G_1, G_2, \cdots, G_n\}$ be a deck consisting of n graphs $G_1, G_2, \ldots, G_n$. It is clear that $G_1, G_2, \ldots, G_n$ have the same number of vertices $n - 1$, and that the number of vertices in a preimage G is n. Since the number of the forbidden graphs of the size n is O(1), we can check if one of them is a preimage of the input graphs in the polynomial time with deck checking algorithm. The time complexity is $O(n^4)$, since the time complexity of the deck checking algorithm is $O(n^4)$.

Theorem 4.15. *If n permutation graphs $G_1, G_2, \ldots, G_n$ have a preimage G that is not a permutation graph, we can reconstruct G from $G_1, G_2, \ldots, G_n$ in $O(n^4)$ time.*

4.2.3 Non-critical Case

First we consider the case that a preimage $G = (V, E)$ has a minimal strong multi-vertex module M such that $|M| \geq 3$, and $G[M]$ is not critical. If M is a prime module, since $G[M]$ is a prime due to Lemma 2.20, $G[M]$ has a vertex v such that $G[M]-v$ is a prime, and hence $M-v$ is a minimal strong multi-vertex module of $G[M] - v$. If M is not a prime module, due to the definition of

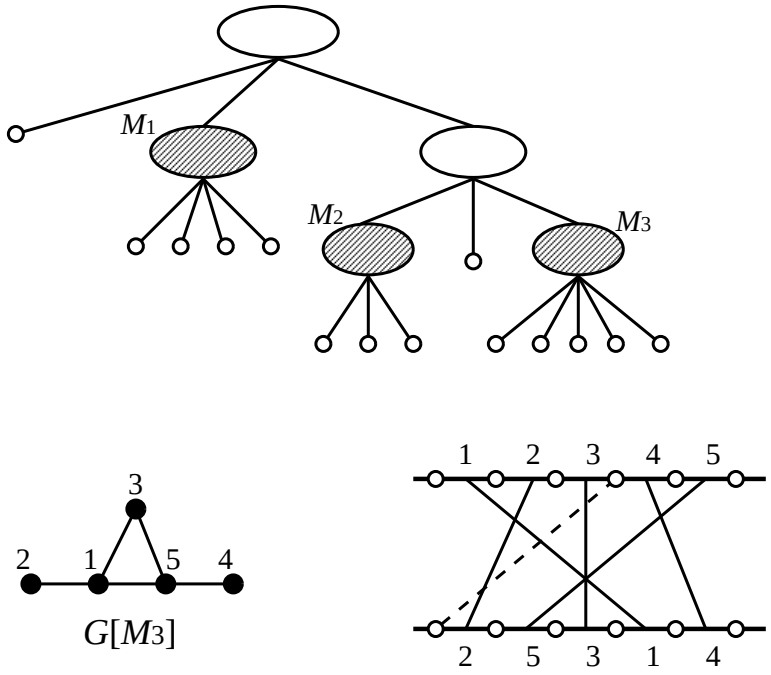


Figure 4.6: Strong modules M_1, M_2, and M_3 are minimal. We add a line segment in the line representation of $G[M_3]$.

modular decomposition, $G[M]$ is a complete graph, or $G[M]$ consists of independent set. And thus $G[M]$ also has a vertex v such that $M - v$ is a minimal strong multi-vertex module of $G[M] - v$.

We search for a preimage by adding a vertex v to every minimal strong multi-vertex module M' of every graph in the deck to check if M' is the desired $M - v$. For every candidate, we use the deck checking algorithm to check if it is a preimage.

If we can specify $N_G(v)$, we can construct a candidate of G. We can easily specify $N_G(v) \setminus M'$, since $M' \cup \{v\}$ should be a module in G, i.e. every vertex in M' and v should seem the same from the vertices in $V \setminus M'$. Thus the remaining task is specification of $N(v) \cap M'$.

Due to the definition of a modular decomposition, M' is one of a clique, an independent set, and a module that induces a prime. It is not difficult to construct the candidate of G if M' is a clique, or M' consists of independent set, since we know the degree sequence of G from Lemma 4.8, that is, we know the degree $\deg(v)$ of v in G. To be concrete, we have to connect v to $\deg(v) - |N(v) \setminus M'|$ vertices in M'.

Next we consider the case that $G[M']$ is a prime. A permutation graph that is a prime with respect to modular decomposition has a unique representation [9, 38]. Thus there are only $O(|M'|^2)$ ways of connection of v and vertices in M'. Note that the number of permutation diagrams obtained by adding a line segment to a permutation diagram is clearly $O(|M'|^2)$, since there are $O(|M'|)$ choices for the end-point on L_1, and there are $O(|M'|)$ choices for the end-point on L_2 (see Figure 4.6). Therefore by checking each of $O(|M'|^2)$ candidates whether it is a preimage with the deck checking algorithm, we have a polynomial time algorithm. We show in **Algorithm 6** the whole algorithm for the case that a preimage has a module that does not induce a critical graph.

We now mention the time complexity of the algorithm in **Algorithm 6**. There are n graphs in the deck. Each graph G in the deck has $O(n)$ minimal strong multi-vertex modules. Note

Algorithm 6: `non-critical-preimage`

Input: multi-set $D = \{G_1, G_2, \cdots, G_n\}$

```
1  begin
2      foreach graph G_i ∈ D do
3          foreach minimal strong multi-vertex module M' of G_i do
4              Prepare an isolated vertex v.
5              Connect v to vertices in V \ M' suitably.
6              if M' is a clique, or M' are independent set then
7                  Connect v to deg(v) − |N(v) \ M'| vertices in M'.
8                  deck-checking(G_i + v, D).
9              else
10                 Create a unique permutation diagram of G[M'].
11                 foreach way of adding v do deck-checking(G_i + v, D);
12             end
13         end
14     end
15 end
```

that the total number of the size of minimal strong multi-vertex modules in G is $O(n)$, and there are $O(|M'|)$ candidates for each minimal strong multi-vertex modules M', so we generate $O(n^2)$ candidates. We can compute these modules in $O(n+m)$ time [12]. The time complexity of deck checking is $O(n^4)$. We can compute a permutation diagram of a permutation graph in $O(n+m)$ time. Therefore the time complexity of the algorithm is $O(n \cdot (n \cdot (n+m) + n^2 \cdot n^4)) = O(n^7)$. Hence we have the theorem below.

Theorem 4.16. *If a preimage $G = (V, E)$ that is a permutation graph has a minimal strong multi-vertex module M such that $|M| \geq 3$, and $G[M]$ is not critical, we can reconstruct G in $O(n^7)$ time.*

4.2.4 Critical Case

Lastly we consider the case that for every minimal strong multi-vertex module M of a preimage $G = (V, E)$, $G[M]$ is critical, or every minimal strong multi-vertex module has the size two.

Assume that all the minimal strong multi-vertex modules of G have the size two. Since a module of the size two makes twins, the reconstruction of G is easy in this case. Any graph G' in the deck is obtained by removing a vertex that is one of twins from G. Thus G can be reconstructed by copying a vertex in G'. We make weak and strong twins of each vertex of every graph in the deck, and check whether the obtained graph is a preimage by the deck checking algorithm. Thus, this algorithm runs in polynomial time.

Now we consider the case that some of minimal strong multi-vertex modules in G have the size more than two. Let M be a minimal strong multi-vertex module of G whose size is more than two. Then since $G[M]$ is a critical graph, $G[M]$ is isomorphic to $H_{|M|}$ or $\overline{H_{|M|}}$. x_1 and x_2 are almost twins in both the $H_{|M|}$ and $\overline{H_{|M|}}$ (see Figure 2.10). In fact $N_{H_{|M|}}(x_1)$ and $N_{H_{|M|}}(x_2)$ differ only in y_1, and $N_{\overline{H_{|M|}}}[x_1]$ and $N_{\overline{H_{|M|}}}[x_2]$ also differ only in y_1. We denote by v_1 and v_2 the vertices in M corresponding to x_1 and x_2 such that $\left|N_{G[M]}(v_1)\right| = \left|N_{G[M]}(v_2)\right| + 1$, or $\left|N_{G[M]}[v_1]\right| = \left|N_{G[M]}[v_2]\right| + 1$

holds. Since M is a module of G, $N_G(v_1)$ contains exactly one vertex in addition to the vertices in $N_G(v_2)$, or $N_G[v_1]$ contains exactly one vertex in addition to the vertices in $N_G[v_2]$.

Now we consider $G - v_2$. $G - v_2$ must be in the deck. Thus we check for every graph G' in the deck if it is $G - v_2$. If G' is $G - v_2$, we can reconstruct G from G' by copying a vertex in G' and removing an edge. We show the algorithm in **Algorithm 7**.

Algorithm 7: `critical-preimage`

Input: multi-set $D = \{G_1, G_2, \cdots, G_n\}$

```
1  begin
2      foreach graph G_i ∈ D do
3          foreach vertex v of G_i do
4              Make weak twin v' of vertex v.
5              deck-checking(G_i + v', D).
6              foreach edge e of N(v') do
7                  Remove e.
8                  deck-checking(G_i + v', D).
9                  Add e.
10             end
11             Remove v'.
12             Make strong twin v' of vertex v.
13             deck-checking(G_i + v', D).
14             foreach edge e of N(v') do
15                 Remove e.
16                 deck-checking(G_i + v', D).
17                 Add e.
18             end
19             Remove v'.
20         end
21     end
22 end
```

We mention the time complexity. There are $O(n)$ graphs in the deck. The number of vertices in each graph is $O(n)$. We have to remove $O(n)$ edges in each iteration. The time complexity of deck checking is $O(n^4)$. Thus the total time complexity of the algorithm is $O(n \cdot n \cdot n \cdot n^4) = O(n^7)$. Thus we have the theorem below.

Theorem 4.17. *If every minimal strong multi-vertex module of a graph G induces a critical graph, or if every minimal strong multi-vertex module of a graph G has the size two, we can reconstruct G in $O(n^7)$ time.*

Combining Theorem 4.15, 4.16 and 4.17, we have the following theorem.

Theorem 4.18. *There is an $O(n^7)$ time preimage construction algorithm for a deck D consisting of n permutation graphs.*

Since we can use preimage construction algorithms for legitimate deck and preimage counting, we also have the legitimate deck and preimage counting algorithms running in the same time complexity for permutation graphs.

4.3 Distance-hereditary Graphs

In this section, we propose a reconstruction algorithm for distance-hereditary graphs. The framework of this section is the same as previous section. First, we present a deck checking algorithm for the basic part of reconstruction algorithm. Next, we consider that a preimage is not a non-distance-hereditary graph. Finally, we propose a reconstruction algorithm for distance-hereditary graphs by using pruning sequence.

4.3.1 Deck Checking

We have to be more careful in the case of distance-hereditary graphs, since a distance-hereditary graph must be connected, and adding a universal vertex breaks (weakly) distance-hereditariness. First we show the isomorphism algorithm for weakly distance-hereditary graphs.

Lemma 4.19. *For two weakly distance-hereditary graphs G_1 and G_2, we can check if G_1 and G_2 are isomorphic in $O(n + m)$ time, where n is the number of vertices in G_1 (and of course in G_2), and m is the number of edges in G_1.*

Proof. The O(m) isomorphism algorithm in [42] does not explicitly use the property that distance-hereditary graphs are connected. It makes two DH-trees corresponding to the two input distance-hereditary graphs, and compare them. Each node of a DH-tree corresponds to an operation of adding twins or adding pendants, and the root corresponds to K_2. We only have to replace the root K_2 by K_1. Since adding $k-1$ weak twins to K_1 results in k isolated vertices, we can generate any disconnected weakly distance-hereditary graphs from K_1. It is straightforward to modify the algorithm in [42] to handle such a case without affecting the time complexity. □

Moreover the following lemma is useful for the deck checking algorithm.

Lemma 4.20. *Given two sets of weakly distance-hereditary graphs $S_1 = \{G_1, \ldots, G_k\}$ and $S_2 = \{G'_1, \ldots, G'_k\}$, we can determine if S_1 is equal to S_2 in $O(k(n + m))$ time, where n is the maximum number of vertices in $G_1, \ldots, G_k$ and $G'_1, \ldots, G'_k$, and m is the maximum number of edges in $G_1, \ldots, G_k$ and $G'_1, \ldots, G'_k$.*

Proof. We extend the DH-tree for a weakly distance-hereditary graph described above to the DH-tree for a set S of weakly distance-hereditary graphs. The root corresponds to an empty graph, and the DH-trees of all the elements in S are the children of the root. Then we can use the similar algorithm to that in [42]. □

Now we describe deck checking algorithm for distance-hereditary graphs. Given a deck D that consists of weakly distance-hereditary graphs at least two of which are connected, and given a distance-hereditary preimage candidate $G = (V, E)$, we prepare the deck $\hat{D}$ of G in O($|V| \cdot |E|$) time. We can check if D and $\hat{D}$ are equivalent in O($|V| \cdot |E|$) time by Lemma 4.20. We thus obtain the theorem below.

Theorem 4.21. *There is $O(|V| \cdot |E|)$ time deck checking algorithm for a deck that consists of weakly distance-hereditary graphs at least two of which are connected, and for a preimage candidate $G = (V, E)$ which is a distance-hereditary graph.*

4.3.2 Non-distance-hereditary Graph Preimage Case

A distance-hereditary graph G is connected, and has no cycle of length more than five, no house, no domino, and no gem as an induced subgraph from Theorem 2.28. This means that the forbidden graphs of weakly distance-hereditary graphs are cycles of length more than five, a house, a domino, and a gem. If a connected graph G is not distance-hereditary (it turns out that G has some forbidden graph as an induced subgraph), and if G has a deck consisting of weakly distance-hereditary graphs, and at least two of them are connected, then G must be the one of a cycle of length more than five, a house, a domino, or a gem, since otherwise some graphs in the deck have the forbidden induced subgraphs. We can check if the input deck is a deck of a house, of a domino, or of a gem in constant time, since the size of these graphs are constant. We can check if the input deck is a deck of a cycle in $O(n^2)$ time, since the deck of a cycle of length n consists of n paths of length $n-2$. Thus we have the theorem below.

Theorem 4.22. *If n weakly distance-hereditary graphs $G_1, G_2, \ldots, G_n$ including at least two connected graphs have a non-distance-hereditary preimage G, we can reconstruct G from $G_1, G_2, \ldots, G_n$ in $O(n^2)$ time.*

4.3.3 Distance-hereditary Preimage Case

A distance-hereditary graph has twins or a pendant from Theorem 2.27. It is easy to develop a polynomial time preimage construction algorithm for the deck of a graph that has twins or a pendant. If a preimage has twins, we can reconstruct it by copying every vertex in the deck and checking if the resulting graph is a preimage by deck checking algorithm. If a preimage has a pendant, we can reconstruct it by adding a degree one vertex to every vertex in the deck and checking if it is a preimage by deck checking algorithm. Thus we have the theorem below.

Theorem 4.23. *Given a deck $D = \{G_1, \ldots, G_n\}$ consisting of weakly distance-hereditary graphs at least two of which are connected, we can list up every distance-hereditary graph whose deck is D, if any, in $O(n^3m)$ time, where n is the number of graphs in D, and m is the number of edges in a preimage.*

Proof. Copying every vertex in every graph in D requires $O(nm)$ time. Adding a pendant to each vertex in every graph in D requires $O(n^2)$ time. Each deck checking costs $O(nm)$ time. The maximum number of deck checking executions is $O(n^2)$. Hence we need $O(nm + n^2 + nm \cdot n^2) = O(n^3m)$ time. □

Since we can use preimage construction algorithms for legitimate deck and preimage counting, we also have the legitimate deck and preimage counting algorithms running in the same time complexity for distance-hereditary graphs.

Algorithm 8: `reconstruct-distance-hereditary`

Input: multi-set $D = \{G_1, G_2, \cdots, G_n\}$

```
1  begin
2      foreach graph G_i ∈ D do
3          foreach vertex v ∈ G_i do
4              Adding a pendant vertex v' that adjacent to v.
5              deck-checking(G_i + v', D).
6              Remove v'.
7          end
8          foreach vertex v ∈ G_i do
9              Adding a weak twin vertex v' of v.
10             deck-checking(G_i + v', D).
11             Adding an edge {v', v}.
12             deck-checking(G_i + v', D).
13             Remove v'.
14         end
15     end
16 end
```

Chapter 5

Efficient Algorithm for $\mathcal{MPQ}$-tree

$\mathcal{MPQ}$-trees are informative data structure for interval graphs. By using $\mathcal{MPQ}$-trees, we can solve the isomorphism problem for interval graphs. Additionally, we use the $\mathcal{MPQ}$-tree for the reconstruction algorithm, implicitly. However, the construction algorithm of $\mathcal{MPQ}$-tree in [32] has several templates, so the implementation of the algorithm is not easy.

We propose a simple constructing $\mathcal{MPQ}$-tree algorithm in this chapter. We show the outline of our algorithm in **Algorithm 9**. In this algorithm, we give an interval representation of an interval graph as an input.

Algorithm 9: `construct-`$\mathcal{MPQ}$`-tree`

Input: interval representation $\mathcal{I}$

1 **begin**
2 Sort endpoints of $\mathcal{I}$.
3 Convert $\mathcal{I}$ to a "ordered" compact interval representation $\mathcal{I}'$.
4 Partition intervals into sets such that each set corresponds to a ($\mathcal{P}$- or $\mathcal{Q}$-) node in the $\mathcal{MPQ}$-tree.
5 Determine the parent-child relations of the nodes, and create sections of $\mathcal{Q}$-nodes.
6 **end**

5.1 Ordered Compact Interval Representation

In this section, we present algorithms to transform a given interval representation $\mathcal{I}$ into a ordered compact interval representation $\mathcal{I}'$.

First, we sort the interval representation $\mathcal{I}$. Figure 5.1(a) is an example of input interval representation. After step 2 of **Algorithm 9**, we have an interval representation in the form of a linked list of endpoints; e.g., $(A, B, C, D, a, c, E, F, d, G, f, b, e, g)$, where the upper and lower case letters stand for the left and right endpoints, respectively. The algorithm uses an array of lists storing endpoints of intervals to represent a compact interval representation; Figure 5.1(c) shows the data structure of the compact interval representation drawn in Figure 5.1(b). We can convert a sorted interval representation to a compact interval representation in O(n) time [52].

We here introduce an ordering of intervals that makes our algorithm simple.

Definition 5.1. *We say that a compact interval representation is* ordered *if the representation satisfies the following conditions:*

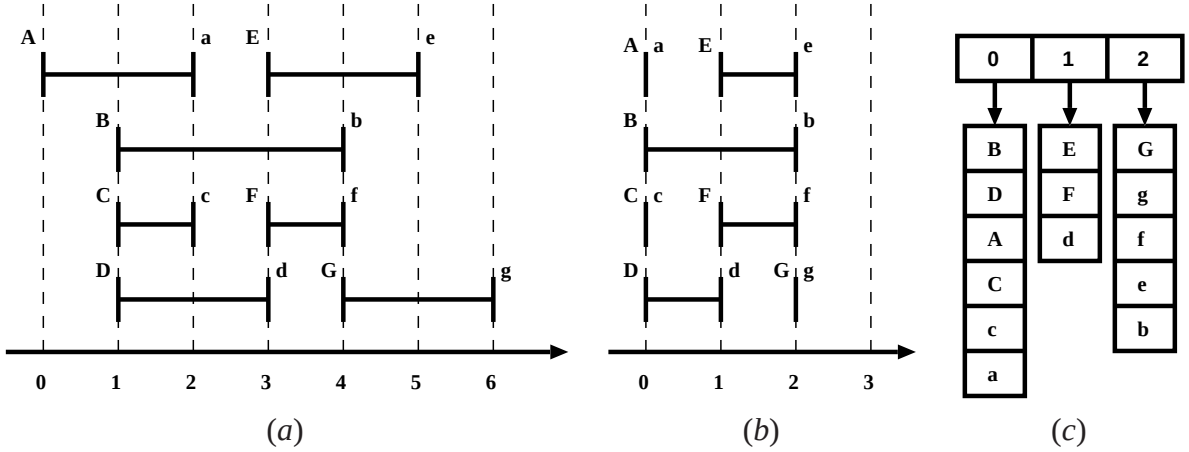


Figure 5.1: (*a*) An input interval representation. (*b*) The compact interval representation corresponding to (*a*). (*c*) Data structure of ordered compact interval representation.

1. *Left endpoints $L(i)$ precede right endpoints $R(j)$ with $L(i) = R(j)$.*
2. *A left endpoint of an interval i precedes any left endpoints of intervals j with $L(i) = L(j)$ and $R(i) > R(j)$.*
3. *A right endpoint of an interval i precedes any right endpoints of intervals j with $R(i) = R(j)$ and $L(i) < L(j)$.*
4. *For a pair i and j with $R(i) = R(j)$ and $L(i) = L(j)$, the right endpoint of i precedes the right endpoint of j if and only if the left endpoint of j precedes the left endpoint of i.*

We next show the following theorem and **Algorithm 10** which constructs the ordered compact interval representation from the compact interval representation.

Theorem 5.2. *For any given compact interval representation (in a form of an array of lists storing endpoints of intervals), we can compute the ordered compact interval representation (in the same form) in $O(n)$ time and $O(n)$ space.*

Proof. First, we sweep the endpoints in a given compact interval representation one by one. For each interval I, we assign the label i that the left endpoint of I appears to the ith left endpoint in the ordered compact interval representation. In this process, we only use an array that store the number of unlabeled left endpoint for each coordinate.

Next, we append the left endpoint in order of the small label. Then, we append the right endpoint in order of the large label. Therefore, this algorithm converts the compact interval representation to the ordered compact interval representation.

Using an array that maintains the correspondence of left and right endpoints, we can write the index in O(1) time. Thus, the ordered compact interval representation can be computed in $O(n)$ time and space. □

Algorithm 10: `ordered-compact-interval-rep`

Input: compact interval representation $\mathcal{I}$

```
1  begin
2      foreach endpoint e ∈ I do
3          if endpoint e is left then
4              Increment the number of left endpoint in the coordinate of e.
5          else
6              Determine the label of the interval I which has the right endpoint e from the
               coordinate of left endpoint of I.
7              Update the array of assigned label.
8          end
9      end
10     for i = 1 to n do  Append left endpoint which has label i.
11     for i = n downto 1 do  Append right endpoint which has label i.
12 end
```

5.2 Find all $\mathcal{P}$-nodes and $\mathcal{Q}$-nodes

We explain how to find all $\mathcal{Q}$-nodes in this section. We can find $\mathcal{P}$-nodes easily after finding all $\mathcal{Q}$-nodes.

We first give the following lemma.

Lemma 5.3. *Let T be an $\mathcal{MPQ}$-tree corresponding to an interval graph G, and $\mathcal{I}$ be a compact interval representation of G. If two intervals i and j in $\mathcal{I}$ overlap, two vertices v_i and v_j corresponding to i and j appear in the same $\mathcal{Q}$-node.*

Proof. We assume that v_i and v_j appear in different nodes N_i and N_j in T, respectively. Because i and j overlap, v_i adjacent to v_j. From the definition of $\mathcal{MPQ}$-tree, N_i is an ancestor of N_j or N_j is an ancestor of N_i. Without loss of generality, we assume that N_i is an ancestor of N_j. Then, the maximal cliques C_i containing v_i contains the maximal cliques C_j containing v_j, or the intersection of C_i and C_j is empty. However, i and j overlap in the compact interval representation by Lemma 2.7, so $C_i \not\subseteq C_j$ and $C_i \cap C_j \neq \phi$. It contradicts the assumption, so v_i and v_j appear in the same node N in T. And by the same reason, it is clear that N is a $\mathcal{Q}$-node. □

We can obtain next lemma from Theorem 2.12 and Lemma 5.3, immediately.

Lemma 5.4. *Let T be an $\mathcal{MPQ}$-tree corresponding to an interval graph G. Each $\mathcal{Q}$-node q on T consists of intervals $\mathcal{I}'$ on any interval representation $\mathcal{I}$ of G, where I' satisfies the following properties.*

1. *$I \in \mathcal{I}'$ overlaps with some other interval in $\mathcal{I}'$, or*
2. *$I \in \mathcal{I}'$ is union of the other intervals in $\mathcal{I}'$.*

We show **Algorithm 11**. This algorithm finds intervals overlapping with some other interval. The algorithm maintains $\mathcal{Q}$-node candidates in a stack and updates $\mathcal{Q}$-nodes candidates

Algorithm 11: `find-all-`Q`-node`

Input: ordered compact interval representation $\mathcal{I}$

```
1  begin
2      initialize stack S and state;
3      foreach endpoint e ∈ I (e is an endpoint of interval i) do
4          if e is left then
5              Push i to S.
6          else if the top of S is interval i and state[i] = 0 then
7              Pop the left endpoint from S.
8          else
9              if state[i]=0 then
10                 Make a new Q-node q of intervals on S from interval i to the top.
11             else
12                 Merge intervals in the Q-node to which i belongs and intervals on S from
                   interval i to the top into a new Q-node q.
13             end
14             Set states of intervals in q whose state is 0 to 1.
15             Set state[i] to 2.
16             if every state of interval in q in 2 then
17                 Remove endpoints of intervals in q from S.
18             end
19         end
20     end
21 end
```

efficiently by using an array *state*. For each interval i, the array *state* stores as follows:

$$state[i] = \begin{cases} 0 & \text{right endpoint of } i \text{ is not processed, and } i \text{ belongs to no } Q\text{-node.} \\ 1 & \text{right endpoint of } i \text{ is not processed, and } i \text{ belongs to a certain } Q\text{-node.} \\ 2 & \text{right endpoint of } i \text{ is processed.} \end{cases}$$

On step 3, endpoints are searched according to the ordering under the conditions 1 to 4 in Definition 5.1. This can be done by simply sweeping the data structure of the ordered compact interval representation. If the endpoint e is left, the algorithm pushes interval i which has endpoint e in the stack S (see steps 4 and 5). If e is right, we compare i and the top of the stack S. At this time, if i is not equal to the top of S, intervals on S from interval i to the top belong to the same Q-node from Definition 5.1 and Lemma 5.4. Since Q-nodes whose intervals have been processed will be removed on steps 16 and 17, a Q-node obtained by this algorithm exactly contains intervals that must be in the node.

We here show the complexity. A naive implementation of this algorithm does not achieve O(n) time. We introduce a good data structure to run the algorithm in O(n) time. We maintain each Q-node candidate as a pair of two indices in the stack. The lemma below guarantees the validness of the method.

Lemma 5.5. *Vertices in a Q-node candidate are consecutive on stack.*

Therefore, the algorithm can maintain a set of Q-node candidates by top and bottom of the set on stack. Thus, two sets of Q-node candidates are merged in O(1) time.

Lemma 5.6. *The total computational cost of merging Q-nodes in the algorithm is $O(n)$.*

Proof. At first, the number of sets of Q-node candidates is at most n. One merger, which takes O(1) time, decreases the number of sets by one. The total number of mergers is thus at most n. Therefore, it takes O(n) time in total to merge the Q-node candidate sets. □

We obtain the following lemma from the Lemmas 5.5 and 5.6.

Theorem 5.7. *Our algorithm partitions intervals into sets such that each set corresponds to a ($\mathcal{P}$ or Q)-node in the $\mathcal{MPQ}$-tree, and runs in $O(n)$ time.*

5.3 Construct $\mathcal{MPQ}$-tree

We construct an $\mathcal{MPQ}$-tree from the ordered interval representation $\mathcal{I}$ and each node of the $\mathcal{MPQ}$-tree. Concretely, we first determine the parent-child relations of the nodes. We next create sections of Q-nodes.

We define *node interval* for each node on $\mathcal{MPQ}$-tree. Let P be a $\mathcal{P}$-node on the $\mathcal{MPQ}$-tree T, and let $\mathcal{I}_p$ be a multi-set of intervals that belong to P. For any two intervals $i, j \in \mathcal{I}_p$, two vertices corresponding to i and j are contained in all maximal cliques represented by the subtree of P in T, but in no other cliques, so i and j correspond to the same interval from Lemma 2.7. We call some interval $i \in \mathcal{I}_P$ $\mathcal{P}$-node P interval.

Let Q be a Q-node on the $\mathcal{MPQ}$-tree T, and let $\mathcal{I}_Q$ be a multi-set of intervals that belong to Q. We define Q-node Q interval as $i = \bigcup_{j \in \mathcal{I}_Q} j$.

We have the following lemma.

Lemma 5.8. *For any two node intervals i and j, i do not overlap j, i.e.*

$$i \cap j = \emptyset,\ i \subset j,\ \text{or}\ j \subset i.$$

Proof. Let i be a node interval of node N_i, and j be a node interval of N_j. If two node intervals i and j overlap, there are intervals $i' \in N_i$ and $j' \in N_j$ that overlap. From Lemma 5.3, i' and j' must belong to the same $\mathcal{Q}$-node. □

Algorithm 12: `determine-parent-child-relation`

Input: ordered compact interval representation $\mathcal{I}$, $\mathcal{P}$-nodes and $\mathcal{Q}$-nodes

```
1  begin
2      foreach endpoint e ∈ I (e is endpoint of interval i) do
3          Let N be a node on the MPQ-tree which contains i.
4          if e is left and N is unprocessed then
5              Set N to the child of the top node of the stack S.
6              Push N to the stack S.
7          end
8          if e is right and intervals in N are processed then Pop S (remove N from S).
9      end
10 end
```

By Lemma 5.8 and definition of $\mathcal{MPQ}$-tree, node N is an ancestor of N' if and only if node interval of N contains node interval of N'. Therefore, we can design **Algorithm 12** that determines parent-child relation of the $\mathcal{MPQ}$-tree. This algorithm maintains only stack, so the algorithm runs in O(n) time.

Theorem 5.9. Algorithm 12 *determines parent-child relations on the $\mathcal{MPQ}$-tree corresponding to interval representation $\mathcal{I}$, and runs in O(n) time.*

Algorithm 13: `create-sections`

Input: ordered compact interval representation $\mathcal{I}$, $\mathcal{P}$-nodes and $\mathcal{Q}$-nodes

```
1  begin
2      foreach Q-node Q do
3          foreach child node N of Q do
4              Create section S_N.
5              Assign intervals that right endpoints are unprocessed when N is made.
6              if S_N is equal to the left section of S_N then
7                  Merge S_N and the left section of S_N.
8              end
9          end
10     end
11 end
```

Next, we design **Algorithm 13** for creating sections of each $\mathcal{Q}$-node Q. The algorithm creates sections for each child of Q, and assigns intervals to the sections. However, if we only

process avobe, section S_i is equal to other section S_j, so the algorithm leaves one section S_i and removes other sections which is equal to S_i.

We show the time complexity. The number of nodes on $\mathcal{MPQ}$-tree is O(n). We process nodes in order of determining the parent-child relation from **Algorithm 12**, because we can remember the endpoints which are assigned section S_N. If S_N has left endpoint or the left of S_N has right endpoint, we must not merge S_N into the left of S_N. Otherwise we merge S_N into the left of S_N, so we can perform steps 4 to 7 in O(1) time. Therefore, **Algorithm 13** runs in O(n) time.

Theorem 5.10. Algorithm 13 *creates sections for each Q-node on the $\mathcal{MPQ}$-tree corresponding to interval representation $\mathcal{I}$, and runs in O(n) time.*

Therefore we have the following theorem from Theorems 5.2, 5.7, 5.9, and 5.10.

Theorem 5.11. *If the input is given in the interval representation with the endpoints sorted by the coordinates, we can obtain an $\mathcal{MPQ}$-tree corresponding to an interval graphs in O(n) time.*

Chapter 6

Concluding Remarks

We proposed efficient random generation and enumeration algorithms for proper interval graphs and bipartite permutation graphs. We investigated unlabeled connected graphs. To deal with unlabeled graphs, it is important to determine whether or not two unlabeled graphs are isomorphic. In this sense, counting/random generation/enumeration on a graph class seems to be intractable if the isomorphism problem for the class is as hard as that for general graphs (see [51] for further details of this topic). It is known that the graph isomorphism problem can be solved in polynomial time for interval graphs and permutation graphs. Hence the future work would be the extensions of our algorithms to general unlabeled interval graphs and permutation graphs.

We presented polynomial time reconstruction algorithms for interval graphs, permutation graphs, and distance-hereditary graphs. These results do not help directly the proofs of the graph reconstruction conjecture on these graph classes. The conjecture on these graph classes still remains to be open.

Kratsch and Hemaspaandra showed that preimage construction on graph class C is GI-hard if the graph isomorphism is GI-hard on C [33]. Remaining famous graph classes that we can find in [9] on which graph isomorphism are not GI-hard contain circular-arc graphs and circle graphs (of course there are other non-GI-hard classes such as threshold graphs. However we mention here higher classes in the hierarchy of the inclusion relation). preimage construction on circular-arc graphs may be a challenging problem. Ma and Spinrad showed that a circle graph G has a unique representation if G is a prime with respect to split decomposition [38]. Split decomposition is a generalization of modular decomposition. Therefore it may be possible that preimage construction on circle graphs is solvable in polynomial time in a similar way described in this paper. Circle graphs contain permutation graphs and distance-hereditary graphs.

Appendix A

The canonical $\mathcal{MPQ}$-tree for an interval graph

In [32], Korte and Möhring proposed two algorithms that construct an $\mathcal{MPQ}$-tree for given interval graph $G = (V, E)$ in $O(n + m)$ time, where $n = |V|$ and $m = |E|$.

The first one constructs a $\mathcal{PQ}$-tree T of G and labels it. The constructed $\mathcal{MPQ}$-tree is essentially the same as the labeled $\mathcal{PQ}$-tree which is used by Colbourn and Booth in [11] to solve the graph isomorphism problem for interval graphs, and hence it is canonical.

The second one incrementally constructs an $\mathcal{MPQ}$-tree from G with the vertex set ordered by LexBFS. The authors do not mind if the constructed $\mathcal{MPQ}$-tree by the second algorithm is unique or not in the paper.

In this section, we show that the second $\mathcal{MPQ}$-tree T is isomorphic to the first one. In other words, the $\mathcal{MPQ}$-tree T constructed by the second algorithm is also canonical. Korte and Möhring show and use the following conditions, which are insufficient to the uniqueness [32, Lemma 2.2]: Let N be a Q-node. Let $S_1, \ldots, S_m$ (in this order) be the sections of N, and let V_i denote the set of vertices occurring below S_i in T with $1 \leq i \leq m$. Then we have the following conditions:

(a) $S_{i-1} \cap S_i \neq \emptyset$ for $i = 2, \ldots, m$.
(b) $S_1 \subseteq S_2$ and $S_{m-1} \supseteq S_m$.
(c) $V_1 \neq \emptyset$ and $V_m \neq \emptyset$.
(d) $S_i \cap S_{i+1} \setminus S_1 \neq \emptyset$ and $S_{i-1} \cap S_i \setminus S_m \neq \emptyset$ for $i = 2, \ldots, m-1$.

The conditions and their definition of an $\mathcal{MPQ}$-tree allow that $S_i = S_{i+1}$ for some i with $1 \leq i \leq m-1$. In the case, the $\mathcal{MPQ}$-tree can have redundancy and we can obtain different $\mathcal{MPQ}$-trees by swapping the sections S_i and S_{i+1} with associated subtrees induced by V_i and V_{i+1}. Thus $\mathcal{MPQ}$-tree is not uniquely determined up to isomorphism for an interval graph. On the other hand, the $\mathcal{MPQ}$-tree constructed by the first algorithm is unique. Hence, to make the second $\mathcal{MPQ}$-tree unique, it is sufficient to reduce the redundancy. Precisely, it is sufficient to add the following condition:

(e) $S_{i-1} \neq S_i$ for $i = 2, \ldots, m-1$.

As a result, by the condition (e), we can replace the condition (b) by the following one:

(b') $S_1 \subset S_2$ and $S_{m-1} \supset S_m$.

We here show the main theorem in this section:

Theorem. *The second algorithm proposed by Korte and Möhring in [32] produces the $\mathcal{MPQ}$-tree that satisfies the conditions (b') and (e). Thus, the second algorithm surely produces the canonical $\mathcal{MPQ}$-tree for an interval graph.*

Proof. The second algorithm in [32] incrementally modifies the current $\mathcal{MPQ}$-tree T for each vertex $v_1, v_2, \ldots, v_n$, which is ordered by LexBFS. We prove the correctness by the induction of n. At the first step, T consists of a $\mathcal{P}$-node containing v_1, and hence T has no redundancy for the sections. Now we assume that T does not have the redundancy before adding a vertex v_i, and show that the addition of v_i does not generate the redundancy.

Then there is a path P in T such that all nodes that have to be modified by the addition of v_i are on P (Lemma 4.1 in [32]). Moreover, P is a subpath of a path from a leaf to the root of T. Let P be the path $(u_1, u_2, ..., u_k)$ such that the algorithm starts the modification from u_1 and ends at u_k. (In the notation in [32], $u_1 = N_*$ and $u_k = N^*$.) We check each modification does not produce the redundancy step by step, based on the case analysis in [32]. We have two cases.

Case (a): $k = 1$. The algorithm uses templates either ($L1$), ($P1$), the upper one in ($Q1$), or ($Q2$). Templates ($L1$) and ($P1$) do not generate $\mathcal{Q}$-node, and we have done. In the upper one in ($Q1$), the algorithm generates new sections $B_1, ..., B_m$. Those sections were $A \cup B_1, A \cup B_2, \ldots, A \cup B_m$, and the common set A is removed. By the inductive hypothesis with the property (e), we have $A \cup B_i \neq A \cup B_{i+1}$ and hence $B_i \neq B_{i+1}$. Hence we still have the property (e). In the case ($Q2$), we have two cases. In any case, for the leftmost two consecutive sections, we have $A \subset S_1$ and $A \subset (A \cup B)$ by the inductive hypothesis. Hence we have no redundancy.

Case (b): $k > 1$. In the case, we have three subcases for u_1, u_k, and u_j with $1 < j < k$. We remind that they are processed from u_1, u_j for each $j = 2, 3, \ldots, k-1$, and u_k.

Case (b-1): For u_1. Templates either ($L2$), ($P2$), the lower one in ($Q1$), or ($Q2$) is applied to deal with the node u_1. In templates ($L2$) and ($P2$), a redundant $\mathcal{Q}$-node is generated. More precisely, we have the following invariance:

($*$) The leftmost two consecutive sections contains the same set A in the $\mathcal{Q}$-node.

In template ($Q1$), two $\mathcal{Q}$-nodes are generated. The lower one does not have the redundancy with the same reason in the case (a), and the upper one has the same redundancy with the invariance ($*$). In template ($Q2$), it does not have the redundancy if $B \neq \emptyset$, but it has the redundancy with the invariance ($*$) when $B = \emptyset$.

Case (b-2): For u_j with $1 < j < k$. In the case, one of templates ($P3$) and ($Q3$) is applied. By the invariance of the case (b-1), only possible redundant sections are S_0 and S_1 in the figure. Hence, if $B \neq \emptyset$, the modification avoids the redundancy. On the other hand, when $B = \emptyset$, we still have the same redundancy with the invariance ($*$). This is repeated for each $j = 2, 3, \ldots, k-1$. Hence the leftmost two consecutive sections may have the redundancy.

Case (b-3): For u_k. In the case, template ($Q3$) is applied. The node u_k, which is denoted by N^* in [32], always contains the vertices not in $N(v_i)$, where v_i is the vertex added to T. Hence we have $B \neq \emptyset$. Before applying the template ($Q3$), we may have the redundancy $S_0 = S_1$. However, after applying, the sections are modified to $A \cup S_0, A \cup B \cup S_1$. Hence, by $B \neq \emptyset$, we always have $A \cup S_0 \subset A \cup B \cup S_1$ at this last step.

Summarizing up, the algorithm in [32] constructs the canonical $\mathcal{MPQ}$-tree, which has no redundancy. Hence it is isomorphic to the $\mathcal{MPQ}$-tree constructed by the first algorithm. □

Bibliography

[1] D. B. Arnold and M. R. Sleep. Uniform Random Generation of Balanced Parenthesis Strings. *ACM Transaction Programming Languages and Systems*, 2(1):122–128, 1980.

[2] D. Avis and K. Fukuda. Reverse Search for Enumeration. *Discrete Applied Mathematics*, 65:21–46, 1996.

[3] H. J. Bandelt, and H. M. Mulder. Distance-hereditary graphs. *Journal of Combinatorial Theory, Series B*, 41:182–208, 1986.

[4] K. P. Bogart and D. B. West. A short proof that 'proper=unit'. *Discrete Mathematics*, 201:21–23, 1999.

[5] B. Bollobás. Almost every graph has reconstruction number three. *Journal of Graph Theory*, 14:1–4, 1990.

[6] J. A. Bondy. A graph reconstructor's manual. *Surveys in Combinatorics, London Mathematical Society Lecture Note Series*, 166:221–252, 1991.

[7] N. Bonichon. A bijection between realizers of maximal plane graphs and pairs of non-crossing Dyck paths. *Discrete Mathematics*, 298:104–114, 2005.

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

[9] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: A Survey*. SIAM, 1999.

[10] C. J. Colbourn. On Testing Isomorphism of Permutation Graphs. *Networks*, 11:13–21, 1981.

[11] C. J. Colbourn and K. S. Booth. Linear time automorphism algorithms for trees, interval graphs, and planar graphs. *SIAM Journal on Computing*, 10:203-225,1981.

[12] E. Dahlhaus, J. Gustedt, and R. M. McConnell. Efficient and practical algorithms for sequential modular decomposition. *Journal of Algorithms*, 41:360–387, 2001.

[13] X. Deng, P. Hell, and J. Huang. Linear-time Representation Algorithms for Proper Circular-arc Graphs and Proper Interval Graphs. *SIAM Journal on Computing*, 25(2):390–403, 1996.

[14] E. Deutsch and L. W. Shapiro. A bijection between ordered trees and 2-Motzkin paths and its many consequences. *Discrete Mathematics*, 256(3):655–670, 2002.

[15] R. Diestel. *Graph Theory*. Springer, 3rd edition, 2006.

[16] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15:835–855, 1965.

[17] T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18:25–66, 1967.

[18] M. R. Garay and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*. W. H. Freeman and Company, 1979.

[19] R. Geary, N. Rahman, R. Raman, and V. Raman. A Simple Optimal Representation for Balanced Parentheses. In *Symposium on Combinatorial Pattern Matching (CPM 2004)*, pages 159–172. Lecture Notes in Computer Science Vol. 3109, Springer-Verlag, 2004.

[20] P. C. Gilmore and A. J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canadian Journal of Mathematics*, 16:539-548, 1964.

[21] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Annals of Discrete Mathematics 57. Elsevier, 2nd edition, 2004.

[22] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, 1989.

[23] P. Hanlon. Counting interval graphs. *Transactions of the American Mathematical Society*, 272(2):383–426, 1982.

[24] F. Harary. A survey of the reconstruction conjecture. Graphs and Combinatorics, *Lecture Notes in Mathematics*, Vol. 406, 18–28, 1974.

[25] E. Hemaspaandra, L. Hemaspaandra, S. Radziszowski, and R. Tripathi. Complexity results in graph reconstruction. *Discrete Applied Mathematics*, 152:103–118, 2007.

[26] Y. Kaneko and S.-i. Nakano. Random Generation of Plane Graphs and Its Application. *IEICE Transactions on Fundamentals*, J85-A(9):976–983, 2002.

[27] A. Karttunen. Personal communication. 2008.

[28] P. J. Kelly. A congruence theorem for trees. *Pacific Journal of Mathematics*, 7:961–968, 1957.

[29] D. E. Knuth. *Generating All Trees*, volume 4 of *The Art of Computer Programming*. Addison-Wesley, fascicle 4 edition, 2005.

[30] Y. Koh and S. Ree. Connected permutation graphs. *Discrete Mathematics*, 307(21):2628–2635, 2007.

[31] Y. Komaki, and M. Arisawa. *Nano Piko Kyoushitsu* (in Japanese). Kyouritsu shuppan, 1990.

[32] N. Korte and R. H. Möhring. An Incremental Linear-Time Algorithm for Recognizing Interval Graphs. *SIAM Journal on Computing*, 18(1):68–81, 1989.

[33] D. Kratsch and L. A. Hemaspaandra. On the complexity of graph reconstruction, *Mathematical Systems Theory*, 27:257–273, 1994.

[34] C. G. Lekkerkerker and J. C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51:45–64, 1962.

[35] Z. Li and S.-i. Nakano. Efficient generation of plane triangulations without repetitions. In *International Colloquium Automata, Languages and Programming (ICALP 2001)*, pages 433–443. Lecture Notes in Computer Science Vol. 2076, Springer-Verlag, 2001.

[36] G. S. Lueker and K. S. Booth. A Linear Time Algorithm for Deciding Interval Graph Isomorphism. *Journal of the ACM*, 26(2):183–195, 1979.

[37] R. H. Möhring. Personal communication. 2003.

[38] T. H. Ma, and J. P. Spinrad. An $O(n^2)$ algorithm for undirected split decomposition. *Journal on Algorithms*, 16:145–160, 1994.

[39] J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31:762–776, 2001.

[40] S.-i. Nakano. Efficient Generation of Plane Trees. *Information Processing Letters*, 84(3):167–172, 2002.

[41] S.-i. Nakano. Enumerating Floorplans with *n* Rooms. *IEICE Transactions on Fundamentals*, E85-A(7):1746–1750, 2002.

[42] S.-i. Nakano, R. Uehara, and T. Uno. A New Approach to Graph Recognition and Applications to Distance Hereditary Graphs. *Journal of Computer Science and Technology*, 24(3):517–533, 2009.

[43] S.-i. Nakano and T. Uno. Constant time generation of trees with specified diameter. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2004)*, pages 33–45. Lecture Notes in Computer Science Vol. 3353, Springer-Verlag, 2004.

[44] A. Pnueli, S. Even, and A. Lempel. Transitive orientation of graphs and Identification of Permutation Graphs. *Canadian Journal of Mathematics*, 23:160-175, 1971.

[45] M. von Rimscha. Reconstructibility and perfect graphs. *Discrete Mathematics*, 47:283–291, 1983.

[46] F. S. Roberts. Indifference graphs. In F. Harary, editor, *Proof Techniques in Graph Theory*, pages 139–146. Academic Press, 1969.

[47] J. H. Schmerl, and W. T. Trotter. Critically indecomposable partially ordered sets, graphs, tournaments and other binary relational structures. *Discrete Mathematics*, 113:191–205, 1993.

[48] J. P. Spinrad. *Efficient Graph Representations*. American Mathematical Society, 2003.

[49] J. P. Spinrad, and J. Valdes. Recognition and isomorphism of two-dimensional partial orders. In *International Colloquium Automata, Languages and Programming (ICALP 1983)*, pages 433–443. Lecture Notes in Computer Science Vol. 154,Springer-Verlag, 1983.

[50] R. P. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge, 1999.

[51] R. Uehara, S. Toda, and T. Nagoya. Graph Isomorphism Completeness for Chordal Bipartite Graphs and Strongly Chordal Graphs. *Discrete Applied Mathematics*, 145(3):479–482, 2004.

[52] R. Uehara and Y. Uno: On computing longest paths in small graph classes. *International Journal of Foundation of Computer Science*, 18:911–930, 2007.

Publications

[1] 斎藤 寿樹, 清見 礼, 上原 隆平. 区間表現から $\mathcal{MPQ}$-tree を構築するアルゴリズム. **計算機科学の理論とその応用（冬の *LA* シンポジウム）**, pp.16:1-16:10, 2007.

[2] Toshiki Saitoh, Masashi Kiyomi, and Ryuhei Uehara. Simple Efficient Algorithm for $\mathcal{MPQ}$-tree of an Interval Graph. *IEICE Technical Report, COMP2007-24*, pp.49-54, 2007.

[3] Toshiki Saitoh, Masashi Kiyomi, and Ryuhei Uehara. Simple Efficient Algorithm for $\mathcal{MPQ}$-tree of an Interval Graph. *KOREA-JAPAN Joint Workshop on Algorithms and Computation (WAAC 2007)*, pp.121-126, 2007.

[4] 斎藤 寿樹, 山中 克久, 清見 礼, 上原 隆平. Proper Interval Graphs のランダム生成と列挙. **夏の *LA* シンポジウム**, pp.22:1-22:8, 2008.

[5] Masashi Kiyomi, Toshiki Saitoh, and Ryuhei Uehara. Reconstruction of Connected Interval Graphs. *Acceleration and Visualization of Computation for Enumeration Problems*, pp.128-134, 2008.

[6] Toshiki Saitoh, Katsuhisa Yamanaka, Masashi Kiyomi, and Ryuhei Uehara. Random Generation and Enumeration of Proper Interval Graphs. *The 3rd Annual Workshop on Algorithms and Computation (WALCOM 2009)*, Lecture Notes in Computer Science Vol. 5431, pp.177-189, 2009.

[7] Masashi Kiyomi, Toshiki Saitoh, and Ryuhei Uehara. Reconstruction of Interval Graphs. *The 15th International Computing and Combinatorics Conference (COCOON 2009)*, Lecture Notes in Computer Science Vol. 5609, pp.106-115, 2009

[8] 斎藤 寿樹, 大舘 陽太, 山中 克久, 上原 隆平. Bipartite Permutation Graph のランダム生成と列挙. **夏の *LA* シンポジウム**, 2009.

[9] Toshiki Saitoh, Yota Otachi, Katsuhisa Yamanaka, and Ryuhei Uehara. Random Generation and Enumeration of Bipartite Permutation Graphs. *IEICE Technical Report, COMP2009-30*, pp.35-42, 2009.

[10] Masashi Kiyomi, Toshiki Saitoh, and Ryuhei Uehara. Reconstruction Algorithms for Permutation Graphs and Distance-Hereditary Graphs. *IPSJ SIG Technical Report, 2009-AL-126*, pp.5:1-5:8, 2009.

[11] Toshiki Saitoh, Yota Otachi, Katsuhisa Yamanaka, and Ryuhei Uehara. Random Generation and Enumeration of Bipartite Permutation Graphs. *The 20th International Symposium on Algorithms and Computation (ISAAC 2009)*, Lecture Notes in Computer Science Vol.5868, pp.1104-1113, 2009.

[12] Masashi Kiyomi, Toshiki Saitoh, and Ryuhei Uehara. Reconstruction Algorithm for Permutation Graphs. *The 4th Annual Workshop on Algorithms and Computation (WALCOM 2010)*, Lecture Notes in Computer Science Vol. 5942, pp.125-135, 2010.