Enumeration of Nonisomorphic Graphs in Graph Classes

Kazuaki Yamazaki[1], Toshiki Saitoh[2], Masashi Kiyomi[3], and Ryuhei Uehara[1]

[1] School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Japan. {torus711,uehara}@jaist.ac.jp
[2] School of Computer Science and Systems Engineering, Kyushu Institute of Technology, Japan. toshikis@ces.kyutech.ac.jp
[3] International College of Arts and Sciences, Yokohama City University, Japan. masashi@yokohama-cu.ac.jp

Abstract. In this paper, a general framework for enumerating every element in a graph class is given. The main feature of this framework is that it is designed to enumerate only non-isomorphic graphs in a graph class. Each element in the class is output in a polynomial time delay. Applying this framework to the classes of interval graphs and permutation graphs, we give efficient enumeration algorithms for these graph classes. The experimental results are also provided. The catalogs of graphs in these graph classes are also provided.

1 Introduction

Recently we have to process huge amounts of data in the area of data mining, bioinformatics, etc. In most cases, we have to use some certain structure to solve problems efficiently. We need three efficiencies to deal with a complex structure; it has to be represented efficiently, essentially different instances have to be enumerated efficiently, and its properties have to be checked efficiently. From the viewpoint of the "difference," in graphs, it is natural to consider that two graphs are different when they are non-isomorphic. However, in general, the graph isomorphism problem is difficult to solve efficiently even on restricted graph classes (see [20]). Though, there are rich structures even if we restricted to the graph classes that allow us to solve graph isomorphism efficiently.

We investigate the enumeration of a graph class from this viewpoint in this paper. In this context, there are two previous results by some of the authors [18, 17]. In the paper, the authors gave efficient enumeration algorithms for proper interval graphs and bipartite permutation graphs. However, they are quite specific to some common properties of these graph classes, and it is unlikely to extend to other graph classes. Therefore, we focus on some graph classes such that graph isomorphism can be solved efficiently, and we develop a general framework that gives us to enumerate all non-isomorphic graphs with n vertices for a given integer n, in each of these graph classes. Intuitively, most of the graph classes in which graph isomorphism can be solved in polynomial time share a common property: Each graph in the graph class can be characterized by a canonical

tree structure, and graph isomorphism can be checked essentially by solving the graph isomorphism problem on these labeled trees [13].

There are two well-known graph classes that graph isomorphism can be solved in polynomial time in this manner; interval graphs [15] and permutation graphs [4]. We mention that these graph classes have been widely investigated since they have many applications, and they are very basic graph classes from the viewpoints of graph theory and algorithms. Therefore many useful properties have been revealed, and many efficient algorithms have been developed for them (see, e.g., [3, 8, 19]). From the practical point of view, when an efficient algorithm for a graph class is developed and implemented, we need many graphs belonging to the class to check the reliability of the algorithm. Thus, for such popular graph classes, efficient enumerations are required [10]. However, as far as the authors know, these concrete catalogs for these graph classes have never been provided.

In this paper, we first propose a general framework of enumeration of a graph class in which graph isomorphism can be solved in polynomial time. Then we turn to the details of applications of this framework to interval graphs and permutation graphs. We finally give the experimental results of the implementations for these graph classes. That is, we give the first actual catalogs of non-isomorphic graphs for these graph classes for small n, where n is the number of vertices. (We note that, for interval graphs, some related results can be found in [9] from the viewpoint of counting, not enumeration.) Due to space limitation, all proofs and some figures can be found in Appendix.

2 Preliminaries

We only consider simple graph $G = (V, E)$ with no self-loop and multiple edges. We assume $V = \{1, 2, \ldots, n\}$ for some n, and $|E| = m$. For two integers i, j, we denote by $G + \{i, j\}$ the graph $(V, E \cup \{\{i, j\}\})$, and by $G - \{i, j\}$ the graph $(V, E \setminus \{\{i, j\}\})$. Let K_n denote the complete graph of n vertices and P_n denote the path of n vertices of length $n - 1$.

A graph (V, E) with $V = \{1, 2, \ldots, n\}$ is an *interval graph* when there is a finite set of intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ on the real line such that $\{i, j\} \in E$ if and only if $I_i \cap I_j \neq \emptyset$ for each i and j with $0 < i, j \leq n$. We call the interval set $\mathcal{I}$ an *interval representation* of the graph. For each interval I, we denote by $L(I)$ and $R(I)$ the left and right endpoints of the interval, respectively (hence we have $L(I) \leq R(I)$ and $I = [L(I), R(I)]$).

A graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$ is a *permutation graph* when there is a permutation π over V such that $\{i, j\} \in E$ if and only if $(i-j)(\pi(i)-\pi(j)) < 0$. Intuitively, each vertex i in a permutation graph corresponds to a line ℓ_i joining two points on two parallel lines L_1 and L_2 such that two vertices i and j are adjacent if and only if the corresponding lines ℓ_i and ℓ_j intersect. We suppose that the indices $1, 2, \ldots, n$ of the vertices give 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. That is, ℓ_i joins the ith vertex on L_1 and the $\pi(i)$th vertex on L_2. We call this intersection model a *line representation* of the permutation graph. As an

example, a permutation graph and its line representation are shown in Fig. 4 in Appendix F.

We define a *graph isomorphism* between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ as follows. The graph G_1 is isomorphic to G_2 when there is a one-to-one mapping $\phi : V_1 \to V_2$ such that for any pair of vertices $u, v \in V_1$, $\{u, v\} \in E_1$ if and only if $\{\phi(u), \phi(v)\} \in E_2$. We denote by $G_1 \sim G_2$ for two isomorphic graphs G_1 and G_2.

3 General framework

For a graph class $\mathcal{C}$, we suppose that the graph isomorphism can be solved in polynomial time for $\mathcal{C}$. We denote by $\mathrm{Iso}(n)$ the time complexity for solving the graph isomorphism problem for two graphs G_1 and G_2 of n vertices in the class $\mathcal{C}$. Here we define the notion of the *canonical* graph for any given graph G in $\mathcal{C}$ with respect to the graph isomorphism. We first suppose that we can define a transitive ordering $<$ over isomorphic graphs in $\mathcal{C}$. That is, (1) either $G_1 < G_2$ or $G_2 < G_1$ holds for any given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ such that $G_1 \sim G_2$ and $E_1 \neq E_2$, and (2) when $G_1 < G_2$ and $G_2 < G_3$ for three isomorphic graphs G_1, G_2, G_3, we have $G_1 < G_3$. Then there exists a unique *minimal* graph G for any set of all isomorphic graphs in $\mathcal{C}$. We call this graph G the *canonical* graph. Our goal is to enumerate all canonical graphs in the class $\mathcal{C}$. To this goal, we will use the following properties of the class $\mathcal{C}$:

Canonical property: For any graph G in $\mathcal{C}$, we can compute its canonical graph in polynomial time. That is, the canonical property guarantees that any graph G can be dealt with its canonical form (in polynomial time).

We use *reverse search* technique to enumerate all graphs (see [1] for the details about reverse search). In reverse search, we define a family tree $\mathcal{T}$ over the graphs in the target graph class $\mathcal{C}$ by introducing a parent-child relationship between two graphs G and G' in $\mathcal{C}$. More precisely, in the class $\mathcal{C}$, we first fix a root node[4] $G_R \in \mathcal{C}$. In this paper, we will use K_n as the root node G_R, since K_n belongs to interval graphs and permutation graphs. For each graph $G \in \mathcal{C} \setminus \{G_R\}$, we assume that its *parent* G' of G is uniquely defined and computed in polynomial time. We will define the parent-child relationship so that it is acyclic, it forms a tree on the graphs rooted at G_R in $\mathcal{C}$. Thus we call the resulting tree spanning the class $\mathcal{C}$ *family tree*, and denoted by $\mathcal{T}$.

For the current graph G, we will modify G by some *basic operation* to find its parent or children in $\mathcal{T}$ of the class $\mathcal{C}$. In this paper, we will use "add an edge" as a basic operation to find its parent. The key requirement is that the parent should be uniquely determined for each graph except the root node in $\mathcal{T}$. In an interval graph (or in a permutation graph) G which is not K_n, there is at least one edge e such that $G + e$ is an interval graph (or a permutation graph,

[4] We use two terms "node" and "vertex" to indicate an element in a graph. When we use "vertex," it indicates a vertex in the original graph G in the class $\mathcal{C}$. On the other hand, when we use "node," it indicates meta-structure of graphs. That is, a "node" in $\mathcal{T}$ indicates a graph in the class $\mathcal{C}$.

respectively). When there are two or more such edges e, we have to determine the unique parent efficiently. To determine the unique parent for any given graph $G \in \mathcal{C} \setminus \{G_R\}$, we need the following *operational property*:

Operational property: Let G be any graph in $\mathcal{C} \setminus \{G_R\}$, where G_R is the root node of $\mathcal{T}$ of $\mathcal{C}$. Then, there exists at least one graph $G' \in \mathcal{C}$ such that G' is obtained from G by applying one basic operation. Moreover, we can find minimal G', which is determined uniquely, among them in polynomial time.

The operational property guarantees that we can find a unique parent of G for a given graph G in $\mathcal{C} \setminus \{G_R\}$ in polynomial time. However, in reverse search, a graph G produces the set of potential *children* $S(G)$. Precisely, the algorithm first produces a set of graphs $S'(G)$ that consists of the graphs obtained by applying the reverse of basic operation. In our context, $S'(G)$ is the set of graph $G - e$ for each edge e. It is guaranteed that all children in the family tree are in $S'(G)$, but there may be redundant graphs. There are three considerable cases. The first case is easy; when $G - e$ is not in $\mathcal{C}$, just discard it. The second case is that G produces two or more isomorphic graphs by the reverse of basic operation. For example, when G is a complete graph and the basic operation is "add an edge," G produces all graphs $G - \{i, j\}$ for all $1 \leq i, j \leq n$ as potential children of G. In this case, the algorithm discards all isomorphic graphs except one. Let $S(G)$ be the set of the nonisomorphic graphs in $\mathcal{C}$ obtained from G by the reverse of basic operation. The last considerable case is that the graph $G' \in S(G)$ has a different parent. This case occurs when G' has two (or more) edges e_1 and e_2 such that $G' + e_1 \in \mathcal{C}$ and $G' + e_2 \in \mathcal{C}$. In this case, G' appears both of $S(G' + e_1)$ and $S(G' + e_2)$. To avoid redundancy, G' will check which is the unique parent.

Now we are ready to show the outline of the enumeration algorithm:

Algorithm 1: Outline of Enumeration Algorithm based on Reverse Search

Input : An integer n
Output: All nonisomorphic graphs of n vertices in a graph class $\mathcal{C}$

A set $\mathcal{S}$ is initialized by the root node of the family tree of $\mathcal{C}$;
while $\mathcal{S}$ *is not empty* **do**
  Pick up one node that represents a graph $G = (V, E)$ in the class $\mathcal{C}$;
  Output G as an element in the class $\mathcal{C}$;
  Compute the set $S(G)$ of nonisomorphic graphs in $\mathcal{C}$ obtained by the reverse of basic operation;
  `// G may produce two or more isomorphic graphs, which should be avoided here.`
  foreach $G' \in S(G)$ **do**
    `// Check if G is the unique parent of G'.`
    Compute the unique parent $\hat{G}'$ of G';
    If $\hat{G}'$ is isomorphic to G, push G' into $\mathcal{S}$;

The algorithm enumerates all elements in breadth first search (BFS) manner when $\mathcal{S}$ is realized by a queue, and in depth first search (DFS) manner when $\mathcal{S}$

is realized by a stack. Hereafter, we suppose that it runs in BFS, which makes proof of correctness simpler.

Let $\mathcal{C}$ be the graph class satisfying the properties above. Then we have the main theorem for the framework:

Theorem 1. *We can enumerate all nonisomorphic graphs of n vertices in $\mathcal{C}$ with polynomial time delay. That is, the running time of the algorithm is $|\mathcal{C}_n|p(n)$ for some polynomial function p, where $\mathcal{C}_n$ denotes the subset of $\mathcal{C}$ that contains all graphs of n vertices in $\mathcal{C}$.*

By Theorem 1, we can establish that there are several graph classes that admit to enumerate all elements in the class in polynomial time delay. However, the efficiency of the enumeration is strongly depending on the detailed implementation for each class. We show two efficient implementations for two representative graph classes; interval graphs and permutation graphs. We also show experimental results, that is, we give catalogs for these graph classes. In both of interval graphs and permutation graphs, we let K_n be the root node of the family tree, and basic operation is "add an edge" to obtain the parent.

4 Enumeration of nonisomorphic interval graphs

We first focus on the enumeration of interval graphs of n vertices. Let $\mathcal{C}$ be the set of interval graphs of n vertices in this section. We first show the operational property for $\mathcal{C} \setminus \{G_R\}$, where $G_R \sim K_n$. (We note that K_n is not only an interval graph, but also a permutation graph, and we use it as a common root node of the family trees for both graph classes.)

Lemma 1 ([12]). *Let $G = (V, E)$ be any interval graph which is not K_n. Then G has at least one edge e such that $G + e$ is also an interval graph.*

4.1 Canonical representation

We turn to the canonical representation of an interval graph. We first show the canonical tree structure, and then we give how to obtain a canonical representation for the graph.

Canonical tree representation As the tree structure for an interval graph, we use the $\mathcal{MPQ}$-tree model. The notion was developed by Korte and Möhring [14] as a kind of labeled $\mathcal{PQ}$-tree introduced by Booth and Lueker [2]. We here give a brief idea, and the details can be found in Appendix B.

A *$\mathcal{PQ}$-tree* is a rooted tree $\mathcal{T}^*$ with two types of internal nodes: $\mathcal{P}$ and $\mathcal{Q}$, which will be represented by circles and rectangles, respectively. The leaves of $\mathcal{T}^*$ are labeled 1-1 with the maximal cliques of the interval graph G. The *frontier* of a $\mathcal{PQ}$-tree $\mathcal{T}^*$ is the permutation of the maximal cliques obtained by the ordering of the leaves of $\mathcal{T}^*$ from left to right. Two $\mathcal{PQ}$-trees $\mathcal{T}^*$ and $\mathcal{T}'^*$ are *equivalent*, if one can be obtained from the other by applying the following rules;(1) arbitrarily

permute the child nodes of a $\mathcal{P}$-node, or (2) reverse the order of the child nodes of a $\mathcal{Q}$-node. A graph G is an interval graph if and only if there is a $\mathcal{PQ}$-tree $\mathcal{T}^*$ whose frontier represents a consecutive arrangement of the maximal cliques of G. The $\mathcal{MPQ}$-tree $\mathcal{T}$ assigns sets of vertices (possibly empty) to the nodes of a $\mathcal{PQ}$-tree $\mathcal{T}^*$ representing an interval graph. A $\mathcal{P}$-node is assigned only one set, while a $\mathcal{Q}$-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 $\mathcal{T}$, but in no other cliques.

For a $\mathcal{Q}$-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 $\mathcal{T}_i$ be the subtree of $\mathcal{T}$ with root Q_i. We then assign a set S_i, called *section*, to each Q_i. Section S_i contains all vertices that are contained in all maximal cliques of $\mathcal{T}_i$ and some other $\mathcal{T}_j$, but not in any clique belonging to some other subtree of $\mathcal{T}$ that is not below Q. The key property of $\mathcal{MPQ}$-trees is summarized as follows:

Theorem 2 ([14, Theorem 2.1]). *Let $\mathcal{T}$ be the $\mathcal{MPQ}$-tree for an interval graph $G = (V, E)$. Then we have the following: (a) $\mathcal{T}$ can be computed in linear time and space. (b) Each maximal clique of G corresponds to a path in $\mathcal{T}$ from the root to a leaf, where each vertex $v \in V$ is as close as possible to the root. (c) In $\mathcal{T}$, 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 $j > 0$.*

For two interval graphs G_1 and G_2, let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the corresponding $\mathcal{MPQ}$-trees. Then $G_1 \sim G_2$ if and only if $T_1 \sim T_2$ (as labeled trees).

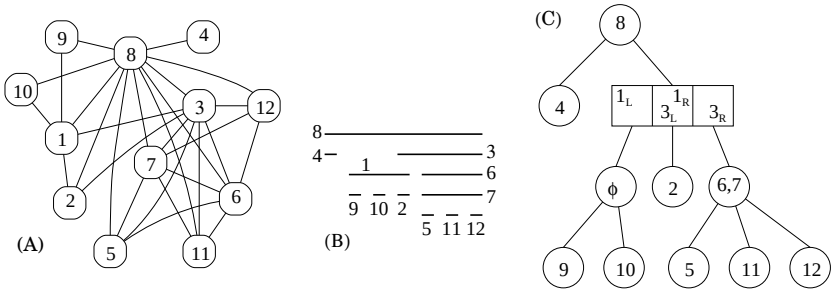

Fig. 1. An interval graph, its interval representation, and its corresponding $\mathcal{MPQ}$-tree.

A simple example is given in Fig. 1. For a given interval graph G in Fig. 1(A), its interval representation is given in Fig. 1(B), and the corresponding $\mathcal{MPQ}$-tree is given in Fig. 1(C).

Canonical string representation The $\mathcal{MPQ}$-tree $\mathcal{T}$ for an interval graph $G = (V, E)$ is the canonical form in the sense that for any two isomorphic interval graphs $G_1 \sim G_2$, the resulting $\mathcal{MPQ}$ trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are also isomorphic and

they can be used to solve the graph isomorphism problem for G_1 and G_2 in linear time since it can be solved in linear time on such labeled trees. We further introduce a *canonical string representation* for a given interval graph to decide the parent of an interval graph uniquely. Intuitively, we will introduce a string representation for an interval graph so that if two interval graphs are isomorphic, their corresponding strings are exactly the same. We here define two basic cases: a complete graph K_n is represented by $1234\ldots(n-1)nn(n-1)\ldots4321$ and a path P_n is represented by $1213243\ldots i(i-1)(i+1)i\ldots n(n-1)n$. To define general canonical string representations, we need more details. The translation from a given $\mathcal{MPQ}$-tree to the canonical string consists of three phases.

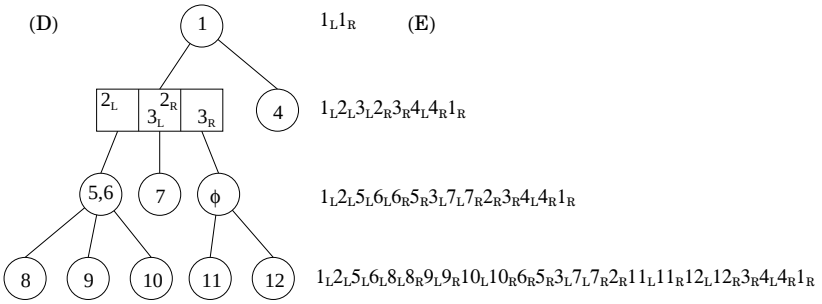

Fig. 2. The $\mathcal{MPQ}$-tree in left-to-right ordering with relabeling, and its canonical string.

First, we draw the $\mathcal{MPQ}$-tree as an ordered tree which is a rooted tree with left-to-right ordering specified by the children of each node. The children for a node are arranged in the ordering from "left-heavy" to "right-light." That is, we introduce a total ordering over all $\mathcal{MPQ}$-trees that is a transitive relationship. This idea can be found in [11], and the details of the ordering for an $\mathcal{MPQ}$-tree can be found in Appendix C. The key property of the ordering is that $\mathrm{Ind}(T_1)$ and $\mathrm{Ind}(T_2)$ for two $\mathcal{MPQ}$-trees are equal if and only if they are isomorphic. Once we draw the $\mathcal{MPQ}$-tree in the way of the ordered tree defined by the ordering, two drawings of $\mathcal{T}_1$ and $\mathcal{T}_2$ are the same (except vertex labelings) if and only if they are isomorphic.

In the second phase, we relabel the vertices $V = \{1, 2, 3, \ldots, n\}$ according to the ordering in the breadth first search manner on the drawing of the tree. (We suppose that a left node is visited before a right node.) By this traverse of vertices of V in the nodes in a $\mathcal{MPQ}$-tree with the basic rule that the canonical representation of K_n is $1234\ldots(n-1)nn(n-1)\ldots4321$, we can observe that two $\mathcal{MPQ}$-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ are isomorphic if and only if the resulting drawings are completely the same including the labels of vertices in V. In this sense, we call the relabeled $\mathcal{MPQ}$-tree $\mathcal{T}$ for an interval graph G the *canonical $\mathcal{MPQ}$-tree of G*. For example, when we apply this process to the $\mathcal{MPQ}$-tree in Fig. 1(C), we obtain the canonicalized $\mathcal{MPQ}$-tree in Fig. 2(D).

In the last phase, we again traverse this canonical $\mathcal{MPQ}$-tree $\mathcal{T}$ in breadth first search manner and generate the canonical string of $\mathcal{T}$ as follows: for a $\mathcal{P}$-node, the algorithm first outputs all left endpoints of the vertices in the node, make recursive calls for each of its children, and output all right endpoints of the vertices in the node following the basic rule of K_n. For a $\mathcal{Q}$-node, the algorithm processes each section by section in the $\mathcal{Q}$-node. The details can be found in Appendix D. Let $\mathrm{Str}_{\mathrm{I}}(G)$ be resulting string representation for a given interval graph G. From the canonicalized $\mathcal{MPQ}$-tree in Fig. 2(D), we obtain the canonical string "1 2 5 6 8 8 9 9 10 10 6 5 3 7 7 2 11 11 12 12 3 4 4 1." In Fig. 2(E), we add $_L$ and $_R$ that indicate left and right endpoints, respectively. We also give each corresponding string for each subtree rooted at the original root up to level 0, 1, 2, and 3. Combining the results in [14], definitions above, and definitions in Appendix D, we obtain the following theorem.

Theorem 3. *Let $G = (V, E)$ be any interval graph with $|V| = n$ and $|E| = m$. (1) The canonical $\mathcal{MPQ}$-tree of G and $\mathrm{Str}_{\mathrm{I}}(G)$ can be computed in $O(n + m)$ time. (2) $|\mathrm{Str}_{\mathrm{I}}(G)| = 2n$. (3) Two interval graphs G_1 and G_2 are isomorphic if and only if $\mathrm{Str}_{\mathrm{I}}(G_1) = \mathrm{Str}_{\mathrm{I}}(G_2)$.*

4.2 Parent-child relationship

As shown in Lemma 1, for any given interval graph $G = (V, E)$ with $G \not\sim K_n$, there is at least one edge $e = \{u, v\}$ with $e \notin E$ such that $G+e$ is also an interval graph. For the graph G, let $\mathcal{T}$ be the canonical $\mathcal{MPQ}$-tree of G. Without loss of generality, we assume that $\mathcal{T}$ is consistent to G from the viewpoint of labels. That is, when we make $\mathcal{T}$ from G, the relabeling process does not change any label of a vertex in V. By Theorem 3, these G and $\mathcal{T}$ can be obtained in linear time. Now we let $\hat{E} = \{e = \{u, v\} \mid G + e \text{ is an interval graph}\}$. Among $\hat{E}$, we can pick up a unique edge $\hat{e}$ that is the lexicographically smallest element in $\hat{E}$. We define the parent of G by $G + \hat{e}$. Clearly, the parent is uniquely determined.

Theorem 4. *Let $G = (V, E)$ be any interval graph with $|V| = n$ and $|E| = m$. Then its parent can be computed in $O(n^2(n + m))$ time.*

4.3 Algorithm analysis

We here analyze the algorithm and show that each graph is enumerated in polynomial time, which guarantees that this algorithm achieves the polynomial time delay for each graph. The root node can be enumerated in polynomial time since it contains K_n. For each graph G in $\mathcal{C}$, we evaluate its running cost consists of its output, the computation of $S(G)$, and the process for each $G' \in S(G)$. The output of G takes $O(n+m)$ time. In this framework with the basic operation, the set $S(G)$ contains at most m children, each of which is obtained from G by removing an edge. It takes $O(m(n+m))$ time (by maintaining the set of canonical string representations in a reasonable data structure, e.g., trie (or prefix tree), we can reduce isomorphic graphs in this process). Then we obtain the set of $O(m)$

graphs, and each G' of them has n vertices and $m-1$ edges. Now the algorithm checks if the unique parent of each G' is G or not. It takes $O(n^2(n+m))$ time by Theorem 4 for each. Thus, this process takes $O(n^2m(n+m))$ time in total. Therefore, each graph consumes $O(n^2m(n+m))$ time in total when it is output. Since $m = O(n^2)$ in general, our enumeration algorithm runs in $O(n^6)$ time per graph.

Our main theorem in this section is summarized as follows:

Theorem 5. *We can enumerate every nonisomorphic interval graph of n vertices. Each interval graph is enumerated in $O(n^6)$ time delay.*

4.4 Three variants of enumeration

Corollary 1. *The algorithm in Theorem 5 can be modified to enumerate (1) connected graphs, and/or (2) at most n vertices. In any variant, the delay is not changed from $O(n'^6)$, where n' is the number of vertices of the output graph.*

5 Enumeration of nonisomorphic permutation graphs

We next focus on the enumeration of permutation graphs of n vertices. Let $\mathcal{C}$ be the set of permutation graphs of n vertices in this section. We first show the operational property for $\mathcal{C} \setminus \{G_R\}$, where $G_R \sim K_n$.

Lemma 2. *Let $G = (V, E)$ be any permutation graph which is not K_n. Then G has at least one edge e such that $G + e$ is also a permutation graph.*

5.1 Canonical representation

Now we turn to the canonical representation of permutation graphs. First, we introduce the notion of *modular decomposition tree*.

Canonical tree representation For a graph $G = (V, E)$, a vertex set $X \subseteq V$ is a *module* if and only if every vertex x not in X, either every member of X is adjacent to x or no member of X is adjacent to x. (See [16] for the details.) *Trivial modules* are $\emptyset$, V, and all the singletons $\{v\}$ for $v \in V$. A graph (or a module) is *prime* if and only if all its modules are trivial. For any permutation graph G, G has a unique line representation (up to reversal) if and only if it is prime [7].

In [7], Gallai defined the *modular decomposition* recursively on a graph with vertex set V. (The modules of a permutation graph in Fig. 4 in Appendix F are given in Fig. 5 in Appendix F.) Intuitively, maximal modules give a unique partition of V recursively, and we have a tree structure with respect to the partition, which is called the *modular decomposition tree*. In a modular decomposition tree $\mathcal{T}$, if all children are joined by edges in the original graph, the parent of them is

called *series* node, and if all children are independent, the parent is called *parallel* node. (A modular decomposition tree of a permutation graph in Fig. 4 in Appendix F is given in Fig. 6 in Appendix F.) It is well known that the modular decomposition tree for a permutation graph (1) is canonical up to isomorphism [4], and (2) can be computed in linear time and space (see, e.g., [6]).

In out context, this fact can be summarized as follows. For two given permutation graphs G_1 and G_2, let $\mathcal{T}_1$ and $\mathcal{T}_2$ be their modular decomposition trees. Then $G_1 \sim G_2$ if and only if (1) $\mathcal{T}_1$ and $\mathcal{T}_2$ satisfy $\mathcal{T}_1 \sim \mathcal{T}_2$ (as labeled trees), and (2) corresponding prime modules are isomorphic.

Canonical string representation The modular decomposition tree $\mathcal{T}$ for a permutation graph $G = (V, E)$ is the canonical form. As considered for interval graphs, we again introduce a *canonical string representation* for a given permutation graph as follows.

We first consider the case that $G = (V, E)$ is a prime module. In this case, as mentioned, G has a unique line representation up to reversal, and hence G has two representations given by two permutations π and π' with $\pi = \pi'^{-1}$. Each permutation can be represented by a string of length n such that every integer $i \in \{1, \ldots, n\}$ appears exactly once. (E.g., P_3 is represented by either 231 or 312.) Therefore, we can choose lexicographically smaller one of π and π' as the canonical string representation of G. (E.g., the canonical string representation of P_3 is 231.)

Now we turn to the general case. This case is similar to the case of interval graphs. We first fix the drawing of the modular decomposition tree according to the total ordering defined in Appendix E. Then, we can fix the ordering of modules, and then the corresponding line representation is uniquely determined. We then relabel all vertices in V such that they appear as $1, 2, \ldots, n$ on L_1. From this line representation, we can obtain the unique permutation π on L_2. We regard this π as the canonical string representation of G. (The canonical string of a permutation graph in Fig. 4 in Appendix F is given in Fig. 8 in Appendix F.)

Now the following theorem is straightforward from the results in [7, 4, 6] and definitions above.

Theorem 6. *Let $G = (V, E)$ be any permutation graph with $|V| = n$ and $|E| = m$. (1) the canonical modular decomposition tree and the canonical string representation of G can be computed in $O(n + m)$ time. (2) Two permutation graphs G_1 and G_2 are isomorphic if and only if $\pi_1 = \pi_2$, where π_i is the canonical string representation of G_i.*

5.2 Parent-child relationship

As shown in Lemma 2, for any given permutation graph $G = (V, E)$ with $G \not\sim K_n$, there is at least one edge $e = \{u, v\}$ with $e \notin E$ such that $G + e$ is also a permutation graph. Therefore, we can use the same idea used in interval graphs. For a given permutation graph G, let $\mathcal{T}$ be the canonical modular

decomposition tree of G. We assume that we relabel G according to its canonical string representation, and $\mathcal{T}$ is the corresponding tree. It can be obtained from the original permutation graph in linear time by Theorem 6. Now we let $\hat{E} = \{e = \{u, v\} \mid G + e \text{ is a permutation graph}\}$. Let $\hat{e}$ be the lexicographically smallest element in $\hat{E}$. We define the unique parent of G by $G + \hat{e}$.

Theorem 7. *Let $G = (V, E)$ be any permutation graph with $|V| = n$ and $|E| = m$ except K_n. Then its parent can be computed in $O(n^2(n + m))$ time.*

5.3 Algorithm Analysis

We here turn to analyze the algorithm. Replacing Theorem 4 by Theorem 7, the analysis is as the same as the case on interval graphs. Therefore, we obtain the following theorem and corollary.

Theorem 8. *We can enumerate every nonisomorphic permutation graph of n vertices. Each permutation graph is enumerated in $O(n^6)$ time delay.*

Corollary 2. *The algorithm in Theorem 8 can be modified to enumerate (1) connected graphs, and/or (2) at most n vertices. In any variant, the delay is not changed from $O(n'^6)$, where n' is the number of vertices of the output graph.*

6 Experimental results

We implemented the proposed algorithms. The number of vertices and the number of non-isomorphic graphs are summarized as follows:

# of vertices	1	2	3	4	5	6	7	8	9	10	11
# of interval graphs	1	2	4	10	27	92	369	1807	10344	67659	491347
# of connected interval graphs	1	1	2	5	15	56	250	1328	8069	54962	410330
# of permutation graphs	1	2	4	11	33	138	-	-	-	-	-
# of connected permutation graphs	1	1	2	6	20	101	-	-	-	-	-

All these graphs are available at `http://www.jaist.ac.jp/~uehara/graphs`. We are now planning to run these programs on a supercomputer in our university to proceed to larger n. The updated data will be available on the web page.

7 Concluding remarks

We propose a general framework that enumerates all nonisomorphic elements in a graph class in which graph isomorphism can be solved in polynomial time. As applications, we give two implementations of the framework for interval graphs and permutation graphs. The first open problem is efficiency. We showed that the implementations for the graph classes ran in $O(n^6)$ time, and the real implementation ran up to some certain n, and we succeeded to give real catalogs for these classes. If we can improve running time, we can list up to larger n. The other future work is to extend this framework to more general classes. Even if graph isomorphism cannot be solved in polynomial time, we may enumerate all nonisomorphic graphs up to some certain n for some simple graph classes.

References

1. D. Avis and K. Fukuda. Reverse Search for Enumeration. *Discrete Applied Mathematics*, 65:21–46, 1996.
2. 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.
3. A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes: A Survey*. SIAM, 1999.
4. C.J. Colbourn. On Testing Isomorphism of Permutation Graphs. *Networks*, 11:13–21, 1981.
5. C.J. Colbourn and K.S. Booth. Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs. *SIAM Journal on Computing*, 10(1):203–225, 1981.
6. C. Crespelle and C. Paul. Fully Dynamic Algorithm for Recognition and Modular Decomposition of Permutation Graphs. *Algorithmica*, 58(2):405–432, 2009.
7. T. Gallai. Transitiv orientierbare Graphen. *Acta Mathematica Academae Scientiarum Hungaricae*, 18:25–66, 1967.
8. M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Annals of Discrete Mathematics 57. Elsevier, 2nd edition, 2004.
9. P. Hanlon. Counting Interval Graphs. *Transactions of the American Mathematical Society*, 272(2):383–426, 1982.
10. P. Heggernes. Personal communication. 2013.
11. 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. LNCS Vol. 3353, Springer-Verlag, 2004.
12. M. Kiyomi, S. Kijima, and T. Uno. Listing Chordal Graphs and Interval Graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2006)*, pages 68–77. LNCS Vol. 4271, Springer-Verlag, 2006.
13. J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhäuser, 1993.
14. N. Korte and R.H. Möhring. An Incremental Linear-Time Algorithm for Recognizing Interval Graphs. *SIAM Journal on Computing*, 18(1):68–81, 1989.
15. 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.
16. R. M. McConnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201:189–241, 1999.
17. T. Saitoh, Y. Otachi, K. Yamanaka, and R. Uehara. Random Generation and Enumeration of Bipartite Permutation Graphs. *Journal of Discrete Algorithms*, 10:84–97, 2012.
18. T. Saitoh, K. Yamanaka, M. Kiyomi, and R. Uehara. Random Generation and Enumeration of Proper Interval Graphs. *IEICE Transactions on Information and Systems*, E93-D(7):1816–1823, 2010.
19. J.P. Spinrad. *Efficient Graph Representations*. American Mathematical Society, 2003.
20. 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.

A Omitted Proofs

In this section, we show omitted proofs.

Proof. (of Theorem 1). Without loss of generality, we suppose that the root node is a complete graph K_n, and the basic operation is "add an edge" for easy to read. We first confirm that the family tree is well-defined if the parent-child relationship is defined properly. Since K_n is the root node, each graph in $\mathcal{C}_n \setminus \{K_n\}$ has its unique parent, and the parent-child relationship is acyclic by definition, we can observe that the directed graph $(\mathcal{C}_n, A)$ forms a directed spanning tree $\mathcal{T}$ rooted at K_n, where A is the set of arcs (u, v) such that v is a child of u. This $\mathcal{T}$ is the family tree of $\mathcal{C}_n$.

We define a *level* of a graph G in this family tree $\mathcal{T}$ as follows; K_n is of level 0, and for each $i = 1, 2, \ldots$, G has level i if it is a child of the graph of level $i-1$. In the current basic operation, we can observe that G has level i if and only if it has $\binom{n}{2} - i$ edges. Moreover, since the algorithm runs in BFS manner, the algorithm enumerates all graphs in level by level.

We first show that every nonisomorphic graph is enumerated exactly once by induction for the level i. When $i = 0$, the algorithm enumerates the complete graph K_n at the root node. The inductive hypothesis is that the claim holds up to level $i-1$. We assume $i > 0$. Let G' be any graph in the level i. Then, by the operational property, G' has at least one edge that can be added. Therefore, there is a set of graphs belonging to the level $i-1$. Among them, there is the parent $\hat{G}'$ of G' in the level $i-1$. By inductive hypothesis, $\hat{G}'$ was enumerated at the level $i-1$. When $\hat{G}'$ is enumerated, the algorithm constructs $S(\hat{G}')$ which contains a canonical graph G'' with $G'' \sim G'$. By the canonical property, G'' is put into $\mathcal{S}$ since $\hat{G}'$ is the parent of G'' and hence G'. Therefore, an isomorphic graph of G' is enumerated at least once. Now we show that G' is not enumerated twice or more. To derive contradictions, we suppose that G' and G'' are enumerated by the algorithm and $G' \sim G''$. By the canonical property, G' and G'' share their common parent $\hat{G}$. Therefore, G' and G'' are enumerated because they are put into $\mathcal{S}$ by when the algorithm deals with $\hat{G}$. However, this contradicts that $S(\hat{G})$ is the set of nonisomorphic graphs. Therefore, each graph is enumerated exactly once with respect to isomorphism.

Now we show the time complexity of the algorithm. We show that each node G in $\mathcal{T}$ uses polynomial time. It is easy to see that the claim holds when G is in level 0, or G is K_n. Thus we assume that $G \not\sim K_n$. When $G = (V, E)$ is picked up from $\mathcal{S}$, it is output at first. Then the algorithm constructs $S(G)$. The key property is that $S(G)$ can be constructed in polynomial time. In the basic operation, the number of elements in $S(G)$ can be bounded above by $|E|$. Thus the algorithm first makes all graphs G' obtained from G by applying the reverse of basic operation. Then it checks and reduces the redundant graphs if $S(G)$ contains two graphs G_1 and G_2 with $G_1 \sim G_2$. This can be done in $O(\binom{|E|}{2}\mathrm{Iso}(n))$ time, which is polynomial by assumption. For each $G' \in S(G)$, we compute its unique parent $\hat{G}'$ again. Since G' contains $|E|-1$ edges, the number of candidates of the parent is $\binom{n}{2} - |E| + 1$. Among them, we can determine the unique parent

$\hat{G}'$ in polynomial time by assumption. Next, the graph isomorphism problem that asks if $\hat{G}' \sim G$ or not is solved in $\mathrm{Iso}(n)$ time. In total, we can observe that the algorithm runs in polynomial for each element in the class $\mathcal{C}_n$, which completes the proof. □

Proof. (of Lemma 1). We here give a brief sketch, and readers can find the details in [12]. For any interval representation $\mathcal{I}_G$ of G, when G is not K_n, we can take a "closest" pair of vertices u, v such that $R(I_u) < L(I_v)$ and there are no other endpoints between them. Then by swapping $R(I_u)$ and $L(I_v)$ so that $L(I_v) < R(I_u)$, we have another interval graph $G + e$ for $e = \{u, v\}$. □

Proof. (of Theorem 4). We first check if each element $\{u, v\} \notin E$ is in $\hat{E}$ or not. This is simply done by using the recognition algorithm for an interval graph, e.g., in [14], which runs in $O(n + m)$ time for each element. Thus, in total, this step runs in $O((\binom{n}{2} - m)(n + m))$ time, or $O(n^2(n + m))$ time. Then we pick up the lexicographically smallest element in $\hat{E}$. This can be done in linear time, or $O(\binom{n}{2} - m)$ time. (We note that, from the practical viewpoint, the second phase is not required when we start searching in lexicographical ordering. Then the first element in $\hat{E}$ is the desired pair.) □

Proof. (of Corollary 1). By the definition of the $\mathcal{MPQ}$-tree, it is easy to observe that an interval graph $G = (V, E)$ is not connected if and only if its corresponding $\mathcal{MPQ}$-tree $\mathcal{T}$ has a $\mathcal{P}$-node R as a root, and R corresponding to an empty set of vertices in V. Therefore, when the algorithm considers for each $G' \in S(G)$, it is sufficient to discard G' if G' has the empty $\mathcal{P}$-node as the root node of the corresponding $\mathcal{MPQ}$-tree of G'. This check can be done in linear time, and it has no effect on the delay of other graphs. (Precisely, in the worst case, $\Theta(n)$ children may be discarded in $O(n^2)$ time, which has no effect on $O(n^6)$ time for the next delay.) It is easy to extend to "at most n vertices" by just repeating the algorithm for each of $1, 2, \ldots, n$. □

Proof. (of Lemma 2). Let $\mathcal{L}_G$ be any line representation of G that represents a permutation π. We remind that $1, 2, \ldots, n$ gives the ordering of endpoints on L_1, and $\pi(i)$ is the $\pi(i)$th endpoint on L_2. Then, since G is not K_n, it has at least two vertices u, v with $\{u, v\} \notin E$. Without loss of generality, we assume that $u < v$. Then $\pi(u) < \pi(v)$ since $\{u, v\} \notin E$. We take "closest" pair $\{u, v\}$ among them as follows. Let u, v be any pair that satisfies $u < v$ and $\pi(u) < \pi(v)$. We first consider the case that $v - u \leq \pi(v) - \pi(u)$. When $v - u > 1$, there is w with $u < w < v$. Then we have three cases: (1) $\pi(u) < \pi(w) < \pi(v)$, (2) $\pi(w) < \pi(u) < \pi(v)$, and (3) $\pi(u) < \pi(v) < \pi(w)$. In (1) or (2), we replace u by w and consider w, v is closer pair than u, v. In (3), we replace v by w and v, w is closer pair than u, v. We next consider the case that $v - u > \pi(v) - \pi(u)$. In this case, we perform the same thing above on L_2 instead of L_1. In any case, after this replacement, we can observe that $\min\{|v - u|, |\pi(v) - \pi(u)|\}$ decreases at least one. Therefore, repeating this process, we finally obtain a closest pair u, v such that $v - u = 1$ or $\pi(v) - \pi(u) = 1$. Then we define a new permutation π' as follows; $\pi(w) = \pi'(w)$ for each $w \in V \setminus \{u, v\}$, $\pi(u) = \pi'(v)$, and $\pi(v) = \pi'(u)$.

Intuitively, we cross the endpoints of two line segments $u\pi(u)$ and $v\pi(v)$. Now it is easy to see that $G + \{u, v\}$ is also a permutation graph characterized by π'. Thus we have the lemma. □

Proof. (of Theorem 7). Using the recognition algorithm for permutation graphs instead of interval graphs, the proof itself is the same as Theorem 4. □

Proof. (of Corollary 2). By the definition of the modular decomposition tree, it is easy to observe that a permutation graph $G = (V, E)$ is not connected if and only if its corresponding modular decomposition tree has a parallel node as a root. Therefore, when the algorithm considers for each $G' \in S(G)$, it is sufficient to discard G' if G' has a parallel node as the root node of the corresponding modular decomposition tree of G'. Thus we have the same conclusion of the case of interval graphs. It is easy to extend to "at most n vertices" by just repeating the algorithm for each of $1, 2, \ldots, n$. □

B Definition of $\mathcal{MPQ}$-trees

We introduce two notions of $\mathcal{PQ}$-trees and $\mathcal{MPQ}$-trees. The $\mathcal{PQ}$-tree was introduced by Booth and Lueker [2], and that can be used to recognize interval graphs as follows. A *$\mathcal{PQ}$-tree* is a rooted tree $\mathcal{T}^*$ with two types of internal nodes: $\mathcal{P}$ and $\mathcal{Q}$, which will be represented by circles and rectangles, respectively. The leaves of $\mathcal{T}^*$ are labeled 1-1 with the maximal cliques of the interval graph G. The *frontier* of a $\mathcal{PQ}$-tree $\mathcal{T}^*$ is the permutation of the maximal cliques obtained by the ordering of the leaves of $\mathcal{T}^*$ from left to right. $\mathcal{PQ}$-trees $\mathcal{T}^*$ and $\mathcal{T}'^*$ are *equivalent*, if one can be obtained from the other by applying the following rules a finite number of times;

1. arbitrarily permute the successor nodes of a $\mathcal{P}$-node, or
2. reverse the order of the successor nodes of a $\mathcal{Q}$-node.

In [2], Booth and Lueker showed that a graph G is an interval graph if and only if there is a $\mathcal{PQ}$-tree $\mathcal{T}^*$ whose frontier represents a consecutive arrangement of the maximal cliques of G. They also developed a linear time algorithm that either constructs a $\mathcal{PQ}$-tree for G, or states that G is not an interval graph. The algorithm by Booth and Lueker contains an update procedure that constructs, from a given $\mathcal{PQ}$-tree $\mathcal{T}^*$ for a system M, a $\mathcal{PQ}$-tree $\mathcal{T}'^*$ representing M plus one additional constraint set. This is done in a bottom-up way along the tree $\mathcal{T}^*$ by comparing parts of the tree with a fixed number of *patterns* that induce certain local *replacements* in $\mathcal{T}^*$. If G is an interval graph, then all consecutive arrangements of the maximal cliques of G are obtained by taking equivalent $\mathcal{PQ}$-trees. 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 $\mathcal{T}_1^*$ and $\mathcal{T}_2^*$ are isomorphic. Since we can determine if two labeled $\mathcal{PQ}$-trees $\mathcal{T}_1^*$ and $\mathcal{T}_2^*$ are isomorphic, the isomorphism of interval graphs can be determined in linear time (see [2, 15, 5] for further details).

The $\mathcal{MPQ}$-tree model, which stands for *modified $\mathcal{PQ}$-tree*, was developed by Korte and Möhring to simplify the $\mathcal{PQ}$-tree [14]. The $\mathcal{MPQ}$-tree $\mathcal{T}$ assigns sets of vertices (possibly empty) to the nodes of a $\mathcal{PQ}$-tree $\mathcal{T}^*$ representing an interval graph $G = (V, E)$. A $\mathcal{P}$-node is assigned only one set, while a $\mathcal{Q}$-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 $\mathcal{T}$, but in no other cliques.

For a $\mathcal{Q}$-node Q, the definition is more involved. Let $Q_1, \ldots, Q_m$ be the set of the children (in consecutive order) of Q, and let $\mathcal{T}_i$ be the subtree of $\mathcal{T}$ with root Q_i. (We note that $m \geq 3$ since it reduces to a $\mathcal{P}$-node when $m < 3$.) 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 $\mathcal{T}_i$ and some other $\mathcal{T}_j$, but not in any clique belonging to some other subtree of $\mathcal{T}$ that is not below Q. The key properties of $\mathcal{MPQ}$-trees are summarized as follows:

Theorem 9 ([14, Theorem 2.1]). *Let $\mathcal{T}^*$ be a $\mathcal{PQ}$-tree for an interval graph $G = (V, E)$ and let $\mathcal{T}$ be the associated $\mathcal{MPQ}$-tree. Then we have the following:*

(a) $\mathcal{T}$ and $\mathcal{T}^*$ can be computed in $O(|V| + |E|)$ time and space.
(b) Each maximal clique of G corresponds to a path in $\mathcal{T}$ from the root to a leaf, where each vertex $v \in V$ is as close as possible to the root.
(c) In $\mathcal{T}$, 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 $\mathcal{T}$ contains all vertices belonging to all maximal cliques, and the leaves contain the simplicial vertices of G. In [14], they did not state Theorem 9(c) explicitly. Theorem 9(c) is immediately obtained from the fact that the maximal cliques containing a fixed vertex occur consecutively in $\mathcal{T}$.

In order to solve the graph isomorphism problem, a $\mathcal{PQ}$-tree requires additional information which is called *characteristic node* in [15, 5]. This is the unique node which roots the subtree whose leaves are exactly the cliques to which the vertex belongs. As noted in [5, p.212], the term characteristic node to mean the leaf, $\mathcal{P}$-node, or portion of a $\mathcal{Q}$-node which contains those cliques. It is easy to see that each vertex v in $\mathcal{MPQ}$-tree directly corresponds to the characteristic node in the $\mathcal{PQ}$-tree.

C Definition of ordering of an $\mathcal{MPQ}$-tree

In this section, we define a total ordering over all $\mathcal{MPQ}$-trees. For an $\mathcal{MPQ}$-tree $\mathcal{T}$, we denote the index of the tree by $\mathrm{Ind}(\mathcal{T})$. Then it should be (1) for any two $\mathcal{MPQ}$-trees $\mathcal{T}_1$ and $\mathcal{T}_2$, $\mathrm{Ind}(\mathcal{T}_1) = \mathrm{Ind}(\mathcal{T}_2)$ if and only if $\mathcal{T}_1 \sim \mathcal{T}_2$, (2) for any three $\mathcal{MPQ}$-trees $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$, $\mathrm{Ind}(\mathcal{T}_1) < \mathrm{Ind}(\mathcal{T}_2)$ and $\mathrm{Ind}(\mathcal{T}_2) < \mathrm{Ind}(\mathcal{T}_3)$ imply $\mathrm{Ind}(\mathcal{T}_1) < \mathrm{Ind}(\mathcal{T}_3)$, and (3) for any two $\mathcal{MPQ}$-trees $\mathcal{T}_1$ and $\mathcal{T}_2$ with $\mathcal{T}_1 \not\sim \mathcal{T}_2$, we have either $\mathrm{Ind}(\mathcal{T}_1) < \mathrm{Ind}(\mathcal{T}_2)$ or $\mathrm{Ind}(\mathcal{T}_1) > \mathrm{Ind}(\mathcal{T}_2)$. In our purpose, we just need

to compare two trees, and determine which is "smaller." Therefore, hereafter, we do not give their indices explicitly, and give the rule that determines which is smaller.

We define the ordering step by step. We consider two $\mathcal{MPQ}$-trees $\mathcal{T}_1 = (V_1, E_1)$ with n_1 vertices and m_1 edges, and $\mathcal{T}_2 = (V_2, E_2)$ with n_2 vertices and m_2 edges. First, if the number of vertices are different, we define the ordering according to them. That is, $\mathrm{Ind}(\mathcal{T}_1) < \mathrm{Ind}(\mathcal{T}_2)$ if $n_1 < n_2$, and $\mathrm{Ind}(\mathcal{T}_1) > \mathrm{Ind}(\mathcal{T}_2)$ if $n_1 > n_2$. Therefore, hereafter, we assume that $n_1 = n_2$. We here define two orderings; (1) a $\mathcal{P}$-node is smaller than $\mathcal{Q}$-node, and (2) a node with fewer vertices is smaller than the other. The rule (1) precedes the rule (2). We then compare the root nodes of $\mathcal{T}_1$ and $\mathcal{T}_2$ according to this rule. If one of them is smaller, we are done. Therefore, we assume that they are the same nodes that consist of the same number of vertices. We have two cases.

Case (a): They are $\mathcal{P}$-nodes. We arrange all children of these $\mathcal{P}$-nodes according to their indices. Assume that the $\mathcal{P}$-node of $\mathcal{T}_1$ has k_1 children and the $\mathcal{P}$-node of $\mathcal{T}_2$ has k_2 children. If $k_1 \neq k_2$, the tree with fewer children is smaller. Therefore we assume that $k_1 = k_2$. In this case, we arrange these children from left to right according to their indices recursively. We compare these lists of children in the lexicographical manner. That is, we first take the first two children from k_1 children and k_2 children respectively, and compare them. If they are different, their ordering gives the ordering of the original $\mathcal{MPQ}$-trees. If they are the same (or isomorphic), we next take the second children from these two $\mathcal{MPQ}$-trees, and so on. If all children are in a tie, we can conclude $\mathcal{T}_1 \sim \mathcal{T}_2$.

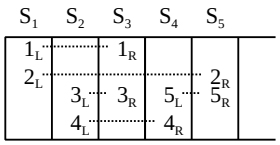

Fig. 3. An example of a $\mathcal{Q}$-node.

Case (b): They are $\mathcal{Q}$-nodes. We first define the ordering between two $\mathcal{Q}$-nodes Q_1 and Q_2 whose drawing are fixed, that is, they cannot be flipped. Let $S_1, S_2, \ldots, S_k$ be the sections of Q_1 in this ordering and $S'_1, S'_2, \ldots, S'_{k'}$ be the sections of Q_2 in this ordering. When $k \neq k'$, the $\mathcal{Q}$-node with fewer sections is smaller. Therefore we assume that $k = k'$. For each section S_i, we define a vector $(x_1, x_2, \ldots, x_{i-1}, y)$ as follows. For each $j = 1, 2, \ldots, i-1$, x_j is the number of intervals that have their right endpoints in this section S_i and their left endpoints are in S_j. The last variable y is the number of intervals that have their left endpoints in this section S_i. For example, we observe a $\mathcal{Q}$-node in Fig. 3. The vector for S_1 is (2) since it contains two left endpoints. The vector for S_2 is $(0, 2)$ since it contains no right endpoints. Then the vector for S_3 is $(1, 1, 0)$ since it contains the right endpoint of the vertex 1 and 3, and their left endpoints are in S_1 and S_2, respectively. The vectors of S_4 and S_5 are $(0, 1, 0, 1)$

and $(1, 0, 0, 1, 0)$, respectively. Then we compare these vectors from S_1 and S'_1 to S_k and S'_k in the lexicographical manner. That is, if j is the smallest index such that the vectors corresponding to S_i and S'_i are the same up to $j-1$, and they are different at j, we decide the ordering according to S_j and S'_j. When all of them are the same, we next compare the children of S_i and $S_{i'}$ in the same manner recursively. When S_j and S'_j have the nonisomorphic children, we can determine the ordering according to them.

Now we turn to the original problem that asks to determine the ordering of two $\mathcal{Q}$-nodes that are allowed to flip them. First, we take one $\mathcal{Q}$-node Q. Then we have two ways to draw it as sections $S_1, S_2, \ldots, S_k$ and $S_k, \ldots, S_2, S_1$ from left to right. We then compare these two drawings and take the smaller one as the description of Q. Similarly, we take another $\mathcal{Q}$-node Q', and fix its direction with respect to its ordering. Finally, we compare these two fixed $\mathcal{Q}$-nodes.

By induction for the depth of a $\mathcal{MPQ}$-tree, it is straightforward that the ordering defined above is a total ordering over all $\mathcal{MPQ}$-trees. We again note that we can compare two $\mathcal{MPQ}$-trees directly in the above manner, and we do not need to compute their indices explicitly.

D Generation of canonical string of an $\mathcal{MPQ}$-tree

In Appendix C, we define a total ordering of $\mathcal{MPQ}$-trees. For any $\mathcal{MPQ}$-tree, we can draw it in a left-heavy manner according to the comparison operation defined in Appendix C. In this way, for any given $\mathcal{MPQ}$-tree $\mathcal{T}$, its drawing can be fixed uniquely. Thus, for this $\mathcal{MPQ}$-tree $\mathcal{T}$ in uniquely fixed drawing, we can visit all $\mathcal{P}$-nodes and $\mathcal{Q}$-nodes, and visit all left endpoints and right endpoints of all intervals represented in this $\mathcal{MPQ}$-tree. Basically, when we traverse the $\mathcal{MPQ}$-tree $\mathcal{T}$, we output left endpoints in pre-order manner, and output right endpoints in post-order manner, it is easy to see that we can construct the corresponding unique interval representation of the original interval graph. For the sake of uniqueness, we define the ordering of left endpoints when $L(I) = L(J)$; when $R(I) < R(J)$, we output $L(I), L(J)$ in this order. Especially, when $L(I) = L(J)$ and $R(I) = R(J)$, we output $L(I), L(J), R(J), R(I)$. Moreover, we relabel the vertices to satisfy $L(I_1) \leq L(I_2) \leq L(I_3) \leq \cdots$. It is not difficult to see that this representation becomes the canonical representation of an interval graph.

E Definition of ordering of a modular decomposition tree

In this section, we define a total ordering over all modular decomposition trees. Basically, we use the same strategy as one used for $\mathcal{MPQ}$-trees. We consider two modular decomposition trees $\mathcal{T}_1 = (V_1, E_1)$ with n_1 vertices and m_1 edges, and $\mathcal{T}_2 = (V_2, E_2)$ with n_2 vertices and m_2 edges. First, if the number of vertices are different, we define the ordering according to them. That is, $\mathrm{Ind}(\mathcal{T}_1) < \mathrm{Ind}(\mathcal{T}_2)$ if $n_1 < n_2$, and $\mathrm{Ind}(\mathcal{T}_1) > \mathrm{Ind}(\mathcal{T}_2)$ if $n_1 > n_2$. We assume that $n_1 = n_2$. Then we define two orderings; (1) a leaf is smaller than a prime node, a prime node is smaller than a serial node, and a serial node is smaller than a parallel node, and

(2) a node with fewer vertices is smaller than the other. The rule (1) precedes the rule (2). We then compare the root nodes of $\mathcal{T}_1$ and $\mathcal{T}_2$ according to this rule. If one of them is smaller, we are done. Therefore, we assume that they are the same nodes that consist of the same number of vertices. This time, we have a similar case of the $\mathcal{P}$-nodes in an $\mathcal{MPQ}$-tree if the current nodes are not prime nodes. The children of each node is arranged from left to right according to their total ordering defined recursively. The last case we have to design is the case that when we deal with a prime node. As shown in [7], if we have a prime node, the corresponding permutation graph G has a unique line representation up to reversal. For this prime nodes, we have two permutations as discussed in Section 5.1. That is, this situation is the same as the case of $\mathcal{Q}$-node in an $\mathcal{MPQ}$-tree: we compare two possible arrangements, and take lexicographically smaller one as its canonical representation. An example is given in Appendix F.

F An example of permutation graph and its canonical representations

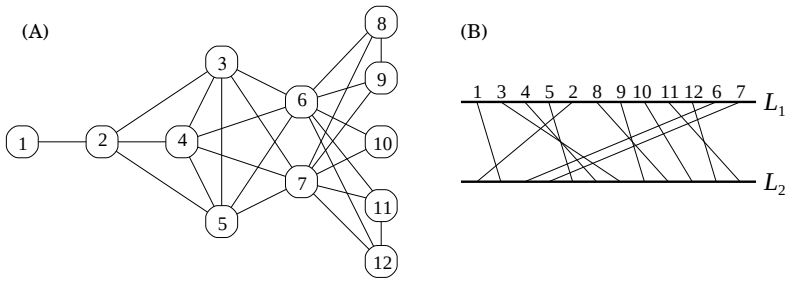

Fig. 4. A permutation graph and its line representation.

In this section, we give a simple example of a permutation graph and show how we will obtain its canonical string presentation. The graph in Fig. 4(A) is a permutation graph since it has a line presentation as shown in Fig. 4(B).

For this graph, its modulars are recursively determined as shown in Fig. 5, and hence we obtain the modular decomposition tree as shown in Fig. 6.

The modular decomposition tree is drawn in the left-to-right manner defined in Appendix E. Then we obtain the tree as shown in Fig. 7.

From the modular decomposition tree in Fig. 7, we can obtain the line representation as in Fig. 8. Then we relabel the vertices as $1, 2, \ldots, n$ on L_1. Now the permutation $\pi = (9, 10, 2, 1, 4, 3, 5, 12, 8, 7, 6, 11)$ appearing on L_2 is the canonical string of the permutation graph in Fig. 4(A).

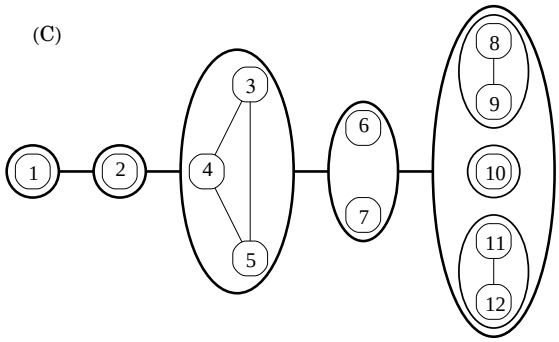

Fig. 5. Modulars in the permutation graph.

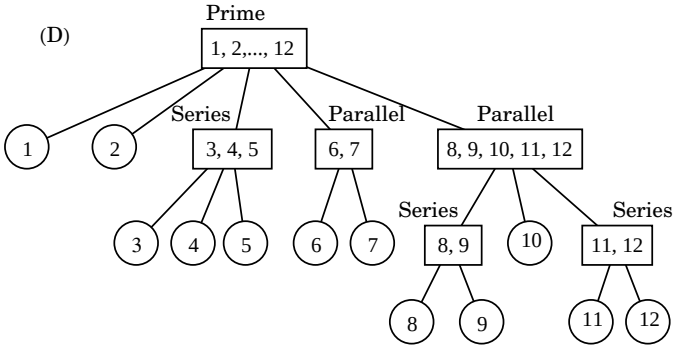

Fig. 6. Modular decomposition tree.

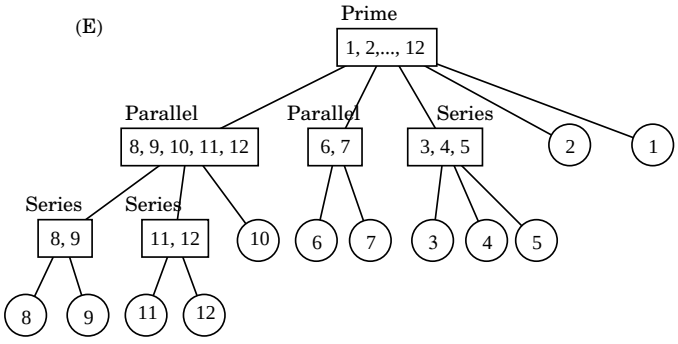

Fig. 7. Redrawn modular decomposition tree in the left-to-right manner.

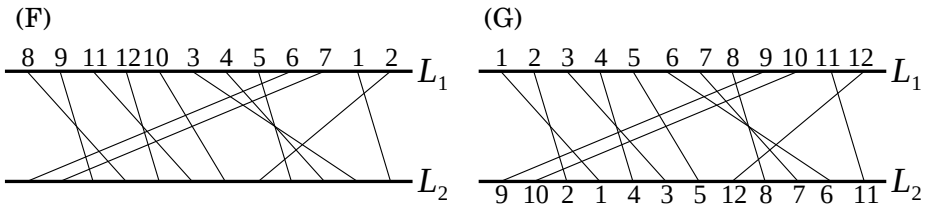

Fig. 8. Canonical string representation.