UNIVERSITÄT LEIPZIG

A Comprehensive Approach for Quadratic-Time Recognition of Permutation Graphs

Thesis submitted to the faculty of Mathematics and Computer Science for the degree of Bachelor of Science

by Tino Fels[1]

13th of October 2021

[1] Supervisor: Dr. N. Wieseke, Department of Computer Science, Swarm Intelligence and Complex Systems Group

Abstract

In this thesis I present the results of an easy-to-implement algorithmic approach designed by my supervisor and myself to recognise arbitrary permutation graphs in $\mathcal{O}(n^2)$ time. Additionally we propose an $\mathcal{O}(n)$ space data structure that allows us to implement multiple useful algorithms for the represented permutation graphs and their respective permutation. The recognition uses a pipeline to build up the inspected graph in a vertex-by-vertex fashion. With the help of our data structure we test in each step, whether the regarded subgraph is a permutation graph. In the positive case, an admissible permutation is given as a certificate.

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Related Work . . . 3

2 Preliminaries **5**
- 2.1 Permutations and Permutation Graphs . . . 5
  - 2.1.1 Basic Graph Theory . . . 5
  - 2.1.2 Permutations . . . 7
  - 2.1.3 Permutation Graphs . . . 7
  - 2.1.4 Important Properties . . . 9
- 2.2 Modular Decomposition . . . 12
  - 2.2.1 Modules . . . 14
  - 2.2.2 Strong Intervals . . . 14
  - 2.2.3 Modular Decomposition Tree . . . 15

3 Data Structure and Algorithms **21**
- 3.1 Base Algorithm . . . 21
  - 3.1.1 Outline . . . 21
  - 3.1.2 Vertex Insertion for Primitive Graphs . . . 23
  - 3.1.3 Finding Twins . . . 25
- 3.2 Data Structure . . . 27
  - 3.2.1 Finding a Representation . . . 27
  - 3.2.2 Proposed Implementation . . . 27
  - 3.2.3 Constructing Permutations of Representative Graphs . . . 30
- 3.3 Generating Insertion Positions . . . 31
  - 3.3.1 Motivation and Linking the Tree . . . 31
  - 3.3.2 Finding the Insertion Node . . . 33
  - 3.3.3 Substitution Process . . . 35
  - 3.3.4 Rearranging the Tree T_q . . . 39

4 Conclusion **49**

Glossary **51**

Bibliography **54**

1 Introduction

Permutation graphs (see Section 2.1 for a more formal introduction) are a subclass of *perfect graphs* and also contain other graph classes, such as cographs, or overlap other useful classes, e.g. (co-)interval graphs. See Figure 1 for an exemplary overview.

Their first description dates back to the work of Dushnik and Miller [1] on the subject of partially ordered sets (posets). Three decades later Pnueli et al. [2] and Even et al. [3], who derived their ideas from Gallai's work [4] which first introduced 'Transitiv orientierbare Graphen' (transitively orientable graphs, another superclass of permutation graphs), sketched a first recognition algorithm for permutation graphs. For a more detailed survey, the inclined reader may refer to the books by Golumbic [5, Chapter 9], and Brandstädt et al. [6, Section 4.7], or the more recent work of Hartmann et al. [7] who provide a full characterization of edge-coloured permutation graphs. The latter is at the moment of writing of this thesis only available as a preprint, but provides an exhaustive overview of the topic.

Permutation graphs are of interest since certain problems, such as the clique problem [5] or the treewidth and pathwidth problems [8], can be solved in polynomial time. There exist other polynomial, or even linear time, algorithms for solving some problems such as specific (sub-)graph isomorphisms [9]. Mondal et al. found several optimal sequential algorithms for a variety of problems [10, 11] with even sub-linear time algorithms under the EREW PRAM model. Additionally, permutation graphs have particular applications, e.g. in flight path calculations [5], specific matching algorithms [12], or historically in memory allocation [3]. In a wider sense permutations, and therefore permutation graphs and interval graphs in particular, can be vital for applications in genomic rearrangement [13], as well as comparative genomics in general [14]. There are many more applications than the ones mentioned above, for a recent overview the reader may refer to the book by Pal et al. [15, Chapter 2].

1.1 Overview

This thesis is structured in three main chapters. Chapter 2 will ease the reader into the mathematical necessities and provides a broad overview on the topics. I will first present some important notions that stem from graph theory and characterise permutations and permutation graphs. The next section will focus on modular decomposition theory which is needed to understand the building of the data structure and the algorithms dependent on it. The reader should direct their main focus towards the understanding of basic modular decomposition of permutation graphs and the

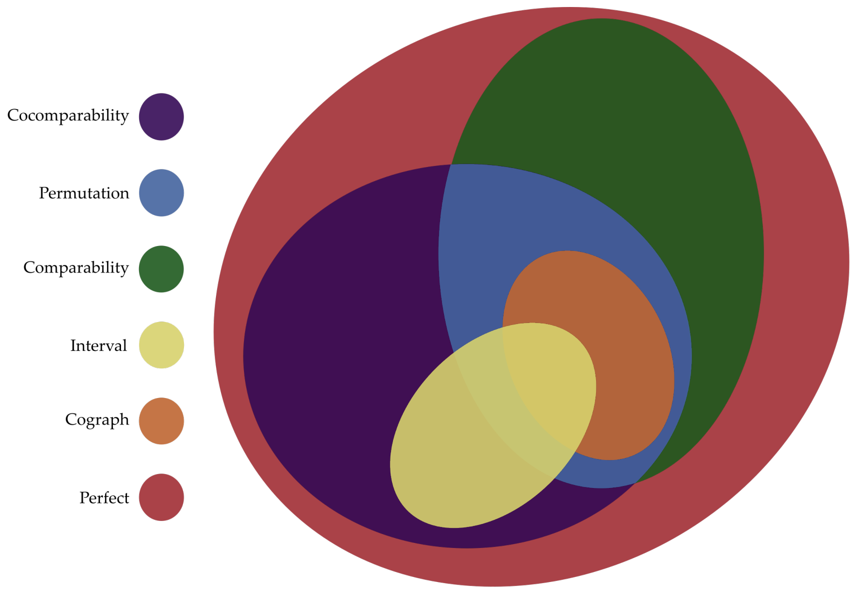


Figure 1: Some perfect graph families and their relations [6, 16]. Note that sizes are not to scale and do not reflect the relative size of the respective graph classes.

labelling of inner nodes. Also of great importance are the contained lemmas 1-3 as well as the basic definition for permutation graphs.

In Chapter 3 I first present the naïve insertion algorithm for primitive permutation graphs and explain why on its own it does not suffice to recognise non-primitive permutation graphs. What follows is a description of the proposed data structure and Algorithm 2 to build the needed tree. Throughout that chapter I tried to stress certain points by providing exemplary figures for the described algorithms, so the reader might refer to a concrete example when trying to follow specific steps along the way. Algorithms are woven into the text, because I deemed it important to have a formal algorithmic description side-by-side to a textual one. The final part of Chapter 3 follows the algorithmic approach for dynamic permutation graph recognition by Crespelle and Paul [17].

Chapter 4 concludes the thesis and outlines the findings presented in the previous chapter by providing an overview of a pipeline to utilise the proposed algorithms and recognise any permutation graph by computing an admissible permutation for it.

Appended to the main matter of this thesis the reader will find a glossary, where I tried to give brief descriptions of the most important terms. I hope altogether the reader is provided with sufficient tools to follow the arguments and algorithms presented in this thesis and thus they have an enjoyable journey into the topic.

1.2 Current and Past Algorithmic Approaches

G is a permutation graph if and only if there exists an admissible permutation π of G. To verify whether π is an admissible permutation of G, we test if only the edges given by π, and no less, also occur in G. This verification can be clearly done in $\mathcal{O}(|E|)$ time, since we can test for each vertex, whether or not it possesses the edges admitted by π.

As shown by McConnell and Spinrad [18], it is possible to recognize permutation graphs, and thus generate their permutation, in $\mathcal{O}(|V| + |E|) = \mathcal{O}(n + m)$ time. The algorithm uses modular decomposition of a given graph and its complement to compute their respective transitive orientations, and from these construct two linear orders. Since for complete graphs $|E| \in \mathcal{O}(|V|^2)$, optimal algorithms must also run in quadratic time, like ours, although there exist some cases (sparse graphs) where $|E| \approxeq |V|$ and algorithms relying on transitive orientation, for example, become superior, since they infer the permutation from edges in G. Confusingly, $\mathcal{O}(n + m)$ is often called ‘linear time’, although it is quadratic in $|V| = n$. In this thesis I will therefore only use the term ‘linear’ if we are indeed in $\mathcal{O}(n)$.

Most recognition algorithms for related graph classes (other *perfect graphs*), however, use ‘fully dynamic’ recognition, in the sense that the underlying data structure, in most cases some kind of modular decomposition tree, is maintained under vertex insertion and deletion. Like *line graphs* by Degiorgi and Simon [19] and Mancini and Heggernes [20]; *proper interval graphs* by Hell et al. [21]; *chordal* [22, 23], *split* [24], and *interval graphs* [23] by Ibarra; and more recently, papers about P_4*-sparse graphs* by Nikolopoulos et al. [25] and *circular-arc graphs* by Soulignac [26]. The work of Crespelle and Paul on fully dynamic permutation graph recognition [17], even inspired the solution presented in this thesis. However, their approach was mainly of theoretical nature and the algorithmic part fell too short in our opinion. Also the core algorithm *InsPrime* of their paper did seem a little bit too complicated for an easy implementation.

Like our approach, fully dynamic graph recognition algorithms often use the same concept of *modular decomposition* (see Section 2.2), sometimes called *substitution decomposition*. This concept is outlined by Möhring [27], and Möhring and Radermacher [28] who give an excellent and exhaustive overview of the matter at that time, resulting in the well-known *modular decomposition theorem* [28]. Later, Cournier and Habib [29] described a modular decomposition algorithm of arbitrary undirected graphs in a depth-first manner and $\mathcal{O}(n + m)$ time. There are other tree-based recognition algorithms as well, cographs (P_4-free permutation graphs), for example, can utilize cotrees [30] which are isomorphic to their modular decomposition trees.

More recent works include the *repeated LBFS* algorithm by Dusart and Habib [31] for cocomparability graphs, a superclass of permutation graphs, in $\mathcal{O}(|V| \cdot |E|) =$

$\mathcal{O}(n \cdot m)$ time. For circular-arc graphs, an overlapping sibling-class of permutation graphs, Kaplan and Nussbaum [32] proposed a linear-time recognition algorithm which they claim improves on the non-dynamic recognition algorithms by McConnell and Spinrad [33] and another by Eschen and Spinrad [34].

2 Preliminaries

Before we proceed to the main part of this thesis, the data structure and algorithms, I will try to ease the reader into the theoretical foundations of permutations, permutation graphs and their modular decomposition.

If not deemed necessary the formal definitions can be skipped. I do recommend, however, to at least skim the examples and figures of this chapter, to get familiar with the most important terms and notations.

2.1 Permutations and Permutation Graphs

This section is mainly concerned with mathematical and graph-theoretic principles which lead to the main subject of permutation graphs and their permutations. For a much more detailed overview and other classes of perfect graphs, the reader may refer to the textbooks of e.g. Brandstädt et al. [6] and Golumbic [5, especially chapter 7, p. 157 sqq.], or to the characterisation of edged-coloured permutation graphs by Hartmann et al. [7], only regarding the 2-coloured case (edges and non-edges). The latter is, at the time of writing of this thesis, only available as a preprint.

2.1.1 Basic Graph Theory

First, let me define a graph (*see also Brandstädt et al. [6, Definition 1.1.1 sqq.]*) as follows:

Definition 2.1 (Vertices, Edges, Graphs)**.** Let V be a finite set of elements (**vertices**) and let $E \subseteq \mathcal{P}_2(V)$ be a set of **edges**, where $\mathcal{P}_2(V)$ denotes the two-element subsets of V.
$G = (V, E)$ is then called an (undirected) **graph**.

Notation. Since V and E are finite sets, their respective sizes can be denoted as $n = |V|$ and $m = |E|$.

Throughout this thesis only finite, undirected graphs that are loop-free and without parallel edges will be regarded, namely *simple graphs.*

Notation. For two vertices $x, y \in V$, xy will denote the undirected edge $\{x, y\} \in E$. I will denote the neighbourhood of a vertex $x \in V$ as $N(x)$ that is every vertex y, such that $y \in N(x)$ if and only if $xy \in E$. For a subset $N \subseteq N(x)$, $xN \subseteq E$ denotes the set of edges that connect x to any element in N.

There are several basic derivative notions which simplify notations and related definitions of graphs. For example we can restrict a given graph to only a subset of its vertices and remove all edges that do not connect elements of the given subset. This leads to the following definition:

Definition 2.2 (Induced Subgraph [6, Definition 1.1.3.])**.** Given a graph $G = (V, E)$, an arbitrary subset of its vertices $S \subset V$ defines the **induced subgraph** $G[S] = (S, E[S])$, where $E[S] \subseteq E$ for $E[S] = \{xy \in E \mid x, y \in S\}$.

Notation. For special (super-)subgraphs, where only a single vertex x (and its neighbouring edges $xN(x)$) is removed (respectively added), I denote $G - x$ as the induced subgraph $G[S]$, with $S = V \setminus \{x\}$ (resp. $G + x = (V \cup \{x\}, E \cup xN(x))$).

Now let us direct our focus towards the relation between vertices of a graph and structures that set constraints to their respective edge sets.

Definition 2.3 (Connectivity, Twins [17])**.** Two vertices $x, y \in V$ are **connected** or **adjacent** if and only if $y \in N(x)$ ($x \in N(y)$ respectively), otherwise they are **disconnected**. Two subsets of vertices $S_1, S_2 \subseteq V$ are adjacent if every vertex $v \in S_1$ is connected to every vertex $u \in S_2$.
Moreover, two vertices are called **twins** if and only if $N(x) \setminus \{y\} = N(y) \setminus \{x\}$.

The non-neighbourhood between all vertices also defines a graph which is called *complementary graph* or just *complement of G*.

Definition 2.4 (Complement)**.** Let $G = (V, E)$ be an undirected graph. $\overline{G} = (V, \overline{E})$ defines the **complementary graph**, where $\overline{E} = \{xy \notin E \mid x, y \in V\} = \mathcal{P}_2(V) \setminus E$.

Looking at specific graphs we can identify two trivial cases of connectivity between their respective vertices. Namely the *complete graph* $K_n = (V, \mathcal{P}_2(V))$ of $n = |V| \geq 2$ vertices and the complementary *empty graph* $\overline{K}_n = (V, \emptyset)$.
Furthermore, we can generalise these concepts to induced subgraphs of any given graph.

Definition 2.5 (Clique, Stable Set [6, Definition 1.1.8])**.** Let $G = (V, E)$ be a graph. If a given subset of vertices $C \subseteq V$ induces a complete subgraph $G[C] = K_{|C|}$ it is called a **clique**.
Conversely, if the induced subgraph is empty it is called a **stable** or independent set.

Another important concept, and a crucial structure for future notions, are trees. For the sake of completeness, let me also give a definition for these special graphs:

Definition 2.6 (Cycle-Free, Tree [6, Porposition 1.1.1])**.** A graph $G = (V, E)$ is **cycle-free** if there is no closed path from any edge to itself, i.e. for any sequence of connected vertices (path) $p = (v_1, \dots, v_k)$, where $v_i v_{i+1} \in E$, $1 \leq i < k \leq n$, it holds that $v_1 \neq v_k$.
A maximal (with respect to edge insertion), cycle-free graph is called a **tree**.

Note that because trees are maximal with respect to edge insertion, i.e. adding any edge will result in a cycle, there exist no stable subsets.

2.1.2 Permutations

After building the foundations, I will now proceed to define permutations and their graphs. The reader should keep this and the next section handy, since they provide definitions and remarks to comprehend these two core concepts.

An algebraic understanding of permutations is not crucial for this thesis overall. Only a basic combinatoric notion of the matter should suffice. Consider an ordered sequence $(i_1\ i_2\ \dots\ i_n)$ of arbitrary objects, represented by natural numbers. $[1:n]$ will denote the (unordered) set of these objects $\{1, \dots, n\}$.

Definition 2.7 (Permutation, Rank [7])**.** Given a set $[1:n]$ of totally ordered objects, a **permutation** $\pi = (\pi(1)\ \pi(2)\ \dots\ \pi(n))$ of these elements is a bijective mapping $\pi : [1:n] \to [1:n]$ that assigns to each element $i \in [1:n]$ a unique element $\pi(i) \in [1:n]$.
The **rank** i or inverse of an element j in a given permutation of $[1:n]$ is then defined by the natural inverse mapping $\pi^{-1}(j) = i$ if and only if $\pi(i) = j$.

Thus, permutations can be regarded as defining 'another ordering' between these objects.

Notation.

(i) $n = |\pi|$ denotes the size, or length of π.

(ii) The sequence $(1\ 2\ \dots\ n)$ is also called the *identity permutation* (of length n) and will be denoted as id_n.

Similarly to graphs, I will also use the following notations:

(iii) $\overline{\pi}$ will denote the *reversed permutation*, i.e. $(\pi(n)\ \pi(n-1)\ \dots\ \pi(1))$, where $\overline{\pi}(i) = \pi(n+1-i)$

(iv) $\pi[i:j]$ or $\pi[X]$ are the *sliced* (sub-)permutations that are restricted to a subsequence $[i:j]$ or an arbitrary subset of its elements X. The order between the chosen elements, as defined by π, will be preserved.

For case *(iv)* the sliced permutation $\pi[X]$ is often normalised in such a way that it only contains consecutive elements in the given order, starting with 1.

After looking at permutation graphs, the reasoning behind those notations will become more apparent. But let me give some examples, first.

Example 1. Given a sequence of objects $[1:4]$, we can define the following permutation $\pi_1 = (3\ 1\ 4\ 2)$.
Where, e.g. $\pi_1(1) = 3$, $\pi_1(3) = 4$, and $\pi_1^{-1}(1) = 2$, $\pi_1^{-1}(3) = 1$. One could also reverse the permutation, resulting in $\overline{\pi}_1 = \pi_2 = (2\ 4\ 1\ 3)$, or only regard the sub-sequences $\pi_1[1:3] = (3\ 1\ 2)$ or $\pi_2[\{1,3,4\}] = (4\ 1\ 3)$ that both normalise to $(3\ 1\ 2)$.
It should be mentioned that at first glance, the latter sub-permutation seems not very useful, since it does not consist of consecutive elements ('2' is missing). But we will later see, that in some instances a carefully defined subset of elements will make sense and this notation gives us the opportunity to do so.

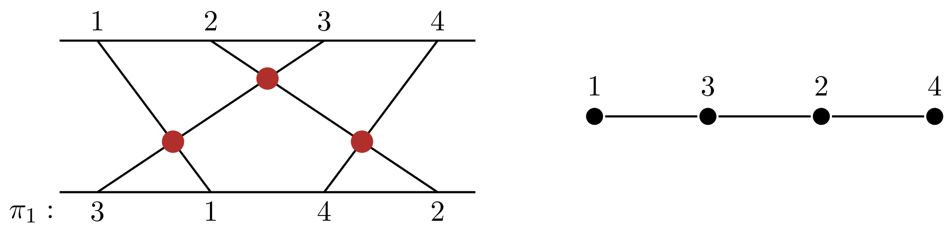
2.1.3 Permutation Graphs

Permutations allow us to depict a defined order of arbitrary objects, and if these objects are the vertices of a graph, the graph is said to be *transitively orientable* [4], or called a *comparability graph*. Dushnik and Miller [1] and Brandstädt et al. [6, Theorem 4.7.1, p. 56] characterise permutation graphs as 'A graph G is a permutation graph if and only if G and $\overline{G}$ are comparability graphs'. This characterisation, however, makes us unable to visualise the essence of permutation graphs, and I will therefore try to show a more comprehensive approach with the next couple of definitions and examples that roughly follow the reasoning of Hartmann et al. [7, Definition 3.1 sqq.].

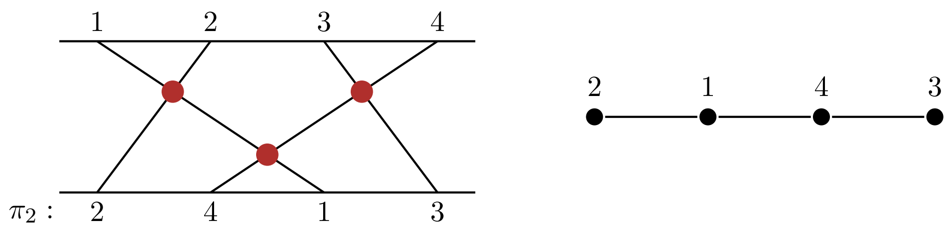
Intuitively, it should be obvious that an arbitrary graph $G = (V, E)$ is a permutation graph if we can find two distinct total orders on its vertices, respectively describing the neighbourhood and non-neighbourhood for each vertex (by transitive orientation [4], that is constructing a digraph from (V, E) in such a way that two vertices $u, v \in V$ are connected by a directed edge (u, v) if there exists a third vertex $w \in V$ such that $(u, w) \in E$ and $(w, v) \in E$). By assigning a label between $[1 : n]$, $n = |V|$ to each vertex, we can depict the first ordering as the ascending sequence $(1\ 2\ \ldots\ n) = \mathrm{id}_n$ and the second as a permutation π of length n. This fact is summarised by the following definitions:

Definition 2.8 (Labelling [7, Definition 3.1])**.** A **labelling** λ of a graph $G = (V, E)$ is a bijective mapping $\lambda : V \to [1 : n]$ that associates each vertex $v \in V$ with a unique natural number $\lambda(v) \in [1 : n]$ ranging from 1 to $n = |V|$.

Given this approach, we can characterise permutation graphs via this labelling and a permutation of the natural order of the labels $[1 : n]$. Where two vertices $u, v \in V$ are



(a) Line graph and labelled permutation graph G_{π_1}.



(b) Line graph and labelled permutation graph G_{π_2}.

Figure 2: Visualisation for the construction of $G_{\pi_1} = (P_4, \lambda_1)$, a permutation graph of the permutation $\pi_1 = (3\ 1\ 4\ 2)$, and its complementary graph $G_{\pi_2} = \overline{G}_{\pi_1} = (P_4, \lambda_2)$ from their respective line graphs. Their labellings are given on the right-hand-side. Intersection points resulting in edges are highlighted red.

connected if and only if the order of their labels $D_\lambda(u,v) \coloneqq \lambda(u) - \lambda(v)$ is reversed by the permutation $(D_\pi(v,u) \coloneqq \pi^{-1}(\lambda(v)) - \pi^{-1}(\lambda(u)))$.

Notation. The notations $D_\lambda(u,v) \coloneqq \lambda(u) - \lambda(v)$ (resp. $D_\pi(u,v) \coloneqq \pi^{-1}(\lambda(u)) - \pi^{-1}(\lambda(v))$) will be used for the differences of the labels (of the ranks in permutation π resp.) of two vertices $u, v \in V$.

Note, that $D_\pi(u,v) = -D_\pi(v,u)$. The same holds for D_λ.

Definition 2.9 (Permutation Graph, Connectivity [2, 6])**.** Given a graph $G = (V, E)$, a labelling λ on its nodes and a permutation $\pi = (\pi(1) \ \dots \ \pi(n))$ such that for all distinct $u, v \in V$ it holds that

$$uv \in E \iff c(u,v) = D_\lambda(u,v)D_\pi(v,u) > 0$$

G is called **permutation graph** of π.
The exact value of the **connectivity** $c(u,v)$ is irrelevant, only whether it is positive or negative.

Note, that the connectivity can never equal zero, since both mappings are bijective and the respective differences are therefore always non-zero values.

Notation. If not obvious from context, I will use $G_\pi = (G, \lambda_\pi)$ to refer to the permutation graph of permutation π and just λ for its labelling. In a slight abuse of the formal notation $\pi^{-1}(v)$ will denote a shorthand for the rank $\pi^{-1}(\lambda(v))$ of vertex $v \in V$ in permutation π, inferring that v can only be referenced by its label $\lambda(v)(= \lambda_\pi(v))$ in π. With that, e.g., the equivalence in Definition 2.9 can be expanded into

$$uv \in E \iff c(u,v) = D_\lambda(u,v)D_\pi(v,u) = (\lambda(v) - \lambda(u))(\pi^{-1}(u) - \pi^{-1}(v)) > 0$$

Definition 2.9 leads to a neat depiction for permutation graphs as *line graphs* or *intersection graphs* of two parallel lines, one being id_n, representing the natural order of the labelling λ, and the other its permutation π. For an example see Figure 2 (also called *matching diagrams*) [5, 6]. Other authors use the term *realiser* [17] or *two-dimensional poset* [6] for the pair of these two linear orders $R = (\lambda, \pi)$ that are a certificate for a permutation graph G_R. The only difference to our notation is the fixed ordering of λ as ascending natural numbers.

2.1.4 Important Properties of Permutation Graphs

The following lemma plays an important part to prove certain properties of permutation graphs and their permutations.

Lemma 1 ([4, 5, 17], see especially Hartmann et al. [7, Lemma 3.3, p. 7])**.** *The class of permutation graphs is hereditary. That is, given a permutation graph G_π, any induced subgraph $G_\pi[S]$, $S \subseteq V$ is also a permutation graph.*

It follows from the fact that comparability graphs (and by extension permutation graphs as well) are closed under *substitution decomposition*, a result shown by Gallai [4]. This also becomes apparent, since removing any vertex from a permutation graph G_π does not change the transitive orientation of the remaining edges, and therefore also the ordering of its other vertices. Example 2 together with Figure 3 tries to build a visual intuition for that.

Another important result arising from that property, is the fact that given a permutation graph G_π, the permutation for any induced subgraph $G_\pi[S]$ can be constructed by *slicing* π accordingly into $\pi[S]$. To strictly match the given definitions, resulting gaps between the sliced elements need to be closed by decrementing each necessary label, while preserving their natural order. See Example 2.

Example 2. Consider the permutation $\pi_3 = (8\ 3\ 1\ 4\ 2\ 6\ 5\ 7)$ and its permutation graph G_{π_3} as shown in Figure 3a and 3b. In π_3 we have the sub-sequence $\pi_3[1:4] = (3\ 1\ 4\ 2)$, and if we have a look at the induced subgraph $G_{\pi_3}[1:4]$ as seen in Figure 3c, one can easily spot that it is isomorphic to the graph of a path of length 4, or P_4 for short, which we already looked at in Figure 2. We can also determine the exact permutation for the given subgraph labelling by just slicing out the permutation $\pi_3[1:4]$, which also proves by Lemma 1 that P_4 is a permutation graph.

In Figure 3d, we can see the permutation graph induced by the remaining vertices $S_2 = [5:8]$. The respective slicing of π_3 would yield $\pi_3[5:8] = (8\ 6\ 5\ 7)$. However, this sub-permutation does not start at 1. We can relabel all elements by regarding the sorted list $[5\ 6\ 7\ 8]$ and re-map these labels into $(5 \mapsto 1, 6 \mapsto 2, 7 \mapsto 3, 8 \mapsto 4)$ resulting in permutation $(4\ 2\ 1\ 3)$ for our graph $G_{\pi_3}[S_2]$.

This relabelling process is not very cost-effective, since the optimal sorting algorithm takes an average of $\mathcal{O}(n \log n)$ time (where n is the length of the sub-sequence), but we will later see that our proposed data structure with Procedure getPi, only spends $\mathcal{O}(n)$ time for this process, since the given sub-sequences contain only consecutive numbers.

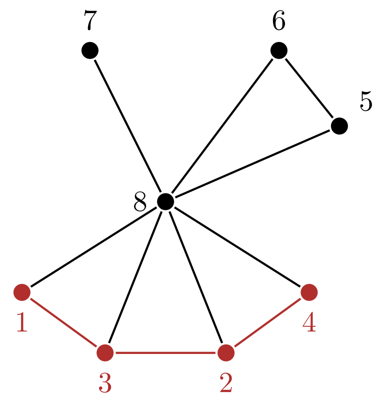
As seen in Figure 2 topologically identical graphs can have different permutations. This relies on the fact that a mirrored line graph of a permutation still results in the identical graph, but with different labelling. In Figure 2, for example, we have horizontal mirror images of one another resulting in an isomorphic permutation graph. I will now show how to retrieve the permutations and labellings from these mirroring processes.

Lemma 2. *Given a graph $G_\pi = (G, \lambda)$, we can construct $G_\mathrm{h} = (G, \lambda^\mathrm{h})$ as an isomorphic graph with different labelling and permutation π^h. Where for any node $v \in V$, $\lambda^\mathrm{h}(v) := \pi^{-1}(\lambda(v))$ and $[\pi^\mathrm{h}]^{-1}(\lambda^\mathrm{h}(v)) := \lambda(v)$.*
We call G_π^h the horizontally mirrored graph and its permutation π^h.

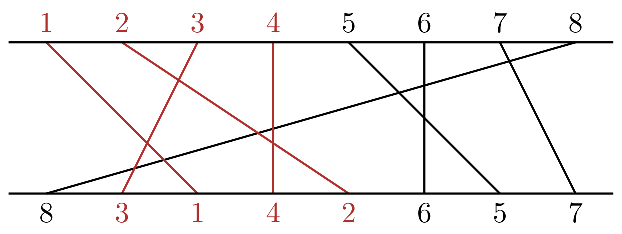
Proof. Trivially, $[\pi^\mathrm{h}]^{-1}$ (the square brackets are for better readability and serve no other purpose), λ^h and π^h are bijective, since their defining mappings are. Each domain also remains the same. Therefore, it suffices to show that the connectivity $c(u, v)$ between two arbitrary vertices $u, v \in V$ is retained.

Using Definition 2.9, we have $c_h = (\lambda^\mathrm{h}(u) - \lambda^\mathrm{h}(v))([\pi^\mathrm{h}]^{-1}(\lambda^\mathrm{h}(v)) - [\pi^\mathrm{h}]^{-1}(\lambda^\mathrm{h}(u)))$. By Lemma 2, $c_h = (\pi^{-1}(\lambda(u)) - \pi^{-1}(\lambda(v)))(\lambda(v) - \lambda(u)) = c(u, v)$. ∎

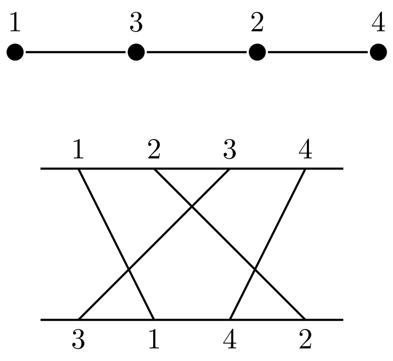
Lemma 3. *Given a graph $G_\pi = (G, \lambda)$, we can construct $G_\mathrm{v} = (G, \lambda^\mathrm{v})$ as an isomorphic graph with different labelling. Where for any node $v \in V$, $\lambda^\mathrm{v}(v) :=$*



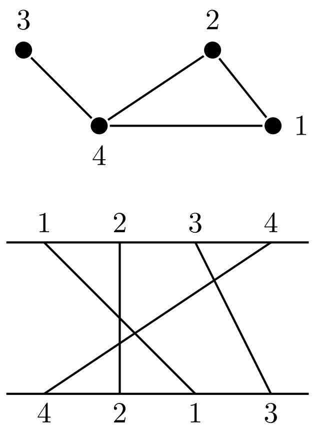
(a) Labelled permutation graph G_{π_3}. Induced subgraph $G_{\pi_3}[S_1]$ for $S_1 = [1:4]$ is highlighted.



(b) Line graph belonging to G_{π_3}. Elements of sub-permutation $\pi_3[S_1]$ are highlighted.



(c) Induced subgraph $G_{\pi_3}[S_1] = G_{\pi_1} \cong P_4$ and its line graph.



(d) Induced subgraph $G_{\pi_3}[S_2]$ and its line graph with permutation $\pi_3[S_2] = (8\ 6\ 5\ 7) \cong (4\ 2\ 1\ 3)$, where each label has been decremented accordingly. $S_2 = [5:8]$.

Figure 3: Visualisation of Example 2, where induced subgraphs are constructed from the permutation graph given by $\pi_3 = (8\ 3\ 1\ 4\ 2\ 6\ 5\ 7)$.

$n+1-\lambda(v)$ and $[\pi^{\mathrm{v}}]^{-1}(\lambda^{\mathrm{v}}(v)) := n+1-\pi^{-1}(\lambda(v))$ with $n = |V|$ being the length of π. We call G^{v}_{π} the vertically mirrored graph of permutation π^{v}.

Proof. Both mappings remain bijective with identical domains, since they are only relabelled (shifted) and reordered reversely, i.e., $1 \mapsto n$, $2 \mapsto n-1$, …, $n \mapsto 1$, etc. It follows by Definition 2.9, that for two vertices $u, v \in V$ $c_v = (\lambda^{\mathrm{v}}(u) - \lambda^{\mathrm{v}}(v))([\pi^{\mathrm{v}}]^{-1}(\lambda(u)) - [\pi^{\mathrm{v}}]^{-1}(\lambda(v)))$, and with the definitions from Lemma 3 we can write $((n+1-\lambda(u)) - (n+1-\lambda(v)))((n+1-\pi^{-1}(\lambda(v))) - (n+1-\pi^{-1}(\lambda(u)))) = (\lambda(u) - \lambda(v))(\pi^{-1}(\lambda(v)) - \pi^{-1}(\lambda(u)) = c(u,v)$, which shows that u, v are connected in G_{v} if and only if they are in G_{π}. ■

Notation. Naturally, combining the operations from Lemma 2 and 3 will result in another isomorphic graph, which I will denote as $G^{\mathrm{m}} = (G, \lambda^{\mathrm{m}})$ and its permutation as π^{m}.

Remark. The process in Lemma 2 can also be described as transposing the two orderings of λ and π, i.e., a vertex labelled i with rank j in π, will have label j and rank i in π^{h}. We can therefore retrieve the rank of a vertex labelled i in π by simply referring to $\pi^{\mathrm{h}}(i)$ instead of $\pi^{-1}(i)$.
Both algorithms sketched in Lemma 2 and 3 clearly have $\mathcal{O}(n)$ time- and space-complexity. For example the labellings for each vertex can be updated one after another, and then be inserted into an empty array at the position according to their rank $[\pi^{\mathrm{h}}]^{-1}$, resulting in the respectively mirrored permutation. An exemplary implementation for a specific purpose is given in Procedure mirror.

It was already mentioned that transitive orientation of a graph G and its complement $\overline{G}$ [4] leads to a permutation, as described by Pnueli et al. [2] for example. This leads to the following proposition.

Proposition 2.1 ([4])**.** *The permutation π of a primitive prime permutation graph G_{π} together with the resulting labelling λ is unique except for reversals (horizontal and/or vertical mirroring). The graphs G_{τ} resulting from $\tau \in \{\pi, \pi^{\mathrm{h}}, \pi^{\mathrm{v}}, \pi^{\mathrm{m}}\}$ are isomorphic, except for their labellings.*

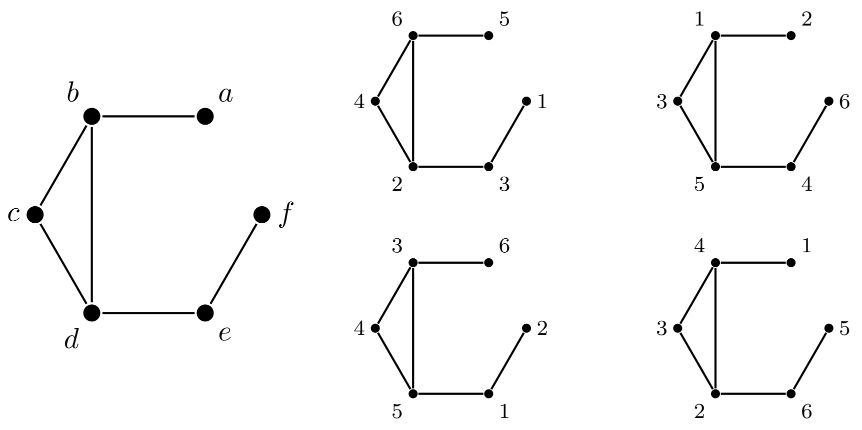
Depicted in Figure 4 is an exemplary graph and each of the resulting permutations.

Another important property relies on the fact that sub-sequences of π identify specific structures (*modules*, see Section 2.2.1 and Section 2.2.2) in their respective permutation graph G_{π}.

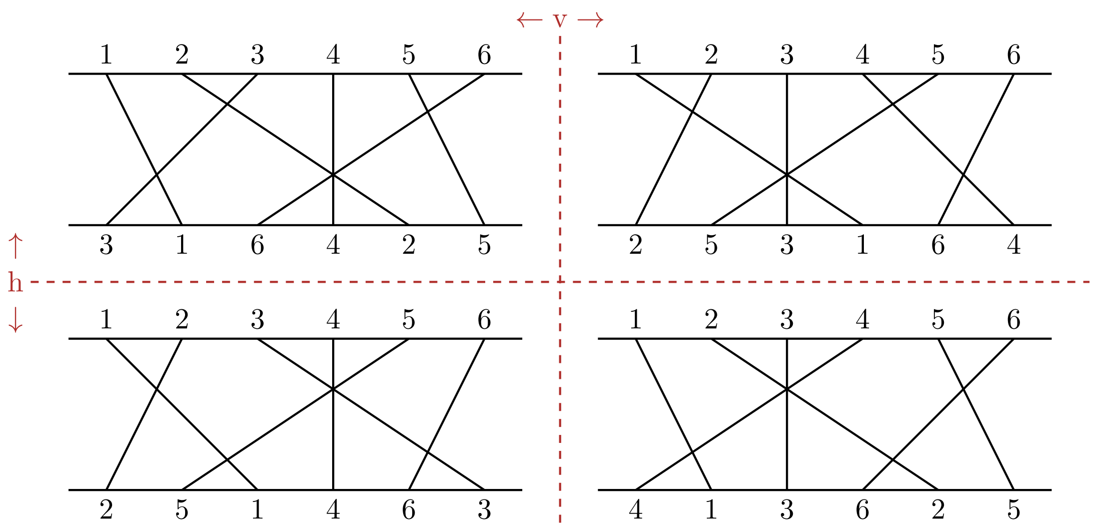
Proposition 2.2 ([7, Proposition 3.4] and particularly [5, Remark p. 159])**.** *Let (G, λ) be a permutation graph with permutation π. The cliques (resp. independent sets) of G_{π} are exactly the descending (resp. ascending) sub-sequences in π. Furthermore, if π is an descending (resp. ascending) sequence, G_{π} is a complete (resp. empty) graph.*

2.2 Modular Decomposition

A crucial part of most recognition algorithms, or more precisely their underlying data-structures, is *modular decomposition*, sometimes called *substitution decomposition* [6]. Since our algorithmic pipeline relies on some claims proven by Crespelle and Paul



(a) Graph G (primitive) and its labelled versions. Each node $a, \dots, f$ can be labelled according to a given permutation (clockwise: $G_\pi, G_\pi^{\mathrm{v}}, G_\pi^{\mathrm{m}}, G_\pi^{\mathrm{h}}$). For example the resulting labelling λ_π of the graph G_π is: $a \mapsto 5$, $b \mapsto 6$, $c \mapsto 4$, $d \mapsto 2$, $e \mapsto 1$, $f \mapsto 3$



(b) Line graphs of permutation $\pi = (3\,1\,6\,4\,2\,5)$ (upper left) and its derived permutations (clockwise: $\pi, \pi^{\mathrm{v}}, \pi^{\mathrm{m}}, \pi^{\mathrm{h}}$) from the mirroring operations. For each operation the connecting lines in the line graph are 'mirrored' and the resulting graph is relabelled according to the fixed order $1, 2, \dots, 6$. The dashed red lines are virtual 'mirror axes' that help visualising each operation. π^{m} can be therefore found across the diagonal.

Figure 4: Permutations $\pi, \pi^{\mathrm{h}}, \pi^{\mathrm{v}}, \pi^{\mathrm{m}}$ are different 'mirror images' of each other, resulting in differently labelled but topologically isomorphic permutation graphs.

[17] together with their recognition algorithm, I will define the most important terms in this section, and, in the end, combine both, the properties of permutation graphs with modular decomposition theory. A good intuition for the last part is especially crucial, because we are going to use this together with Bergeron et al. [35] algorithms in our pipeline as well.

For a broader overview of this topic, the reader can again refer to the aforementioned textbook of Brandstädt et al. [6, Section 1.5], or the characterisation of edge-coloured permutation graphs by Hartmann et al. [7]. Another extensive survey, not only from a graph-theoretic perspective, deliver the works of Möhring and Radermacher [28], as well as Möhring [27, 'Modular Decomposition Theorem'], which are commonly cited on this subject.

2.2.1 Modules

Generally speaking, a module is an equivalence class of vertices with respect to connectivity to any vertex outside the regarded module, i.e. every vertex contained in a module has the same connectivity to vertices outside the module [6, p. 13 sqq.]. This notion is formalised by the following definition.

Definition 2.10 (Uniform, Module, Linked [6, Definition 1.5.1])**.** Let $G = (V, E)$ be an undirected graph. A subset $M \subset V$ is **uniform** with respect to $x \in V \setminus M$ if $M \subseteq N(x)$ (**linked**), or $M \subseteq \overline{N}(x)$ (**notlinked**); otherwise it is **mixed**.
If M is uniform to all vertices $v \in V \setminus M$, then it is called a **module**.
The vertex set V and the singleton sets $\{v\} \subseteq V$ are called **trivial modules** of G.

This property alone does not yet suffice for modular decomposition in general. The definition needs to be narrowed a little, because we want to to use modules of a graph to construct a *modular decomposition tree.*

Definition 2.11 (Overlap, (Maximal) Strong Module [6, 17])**.** Let $G = (V, E)$ be an undirected graph and $M_1, M_2 \subseteq V$ two arbitrary subsets of its vertices. M_1 and M_2 **overlap** each other if $M_1 \cap M_2 \neq \emptyset$ and if M_1 and M_2 are mutually exclusive, i.e., $M_1 \not\subseteq M_2$ and $M_2 \not\subseteq M_1$.
A given module M_1 is **strong** if it does not overlap any other module $M_2 \subset V$.
If M_1 is also maximal with respect to inclusion, and distinct from V, it is called a **maximal strong module**.

Notation. $\mathbb{M}_{\text{str}}(V)$ will denote the set of strong modules and $\mathbb{M}_{\max}(V)$ the maximal strong modules of a graph $G = (V, E)$.

The trivial modules and *(co-)connected components* of a graph are always strong modules, the latter even being exactly the maximal strong modules of any graph [17].

2.2.2 Strong Intervals

The time-complexity of modular decomposition algorithms for arbitrary graphs is only linear (i.e. $\mathcal{O}(n)$, $n = |V|$) if a *factorizing permutation* of the given graph is known [35, 36]. But since permutation graphs are themselves defined over such a permutation, we can follow for example the outline given by Bergeron et al. [35, section 6.] to construct a modular decomposition tree in $\mathcal{O}(n)$. Before we introduce the main notion needed for Bergeron et al.'s algorithms, strong intervals, we have to define *intervals*, sometimes called *common intervals*, that are sub-sequences containing only consecutive elements of a given permutation π.

Definition 2.12 (Interval, Strong Intervals [17, 37])**.** For a given permutation π, we define an **interval** as a sub-sequence of consecutive elements $\pi[i : j]$, $1 \leq i \leq j \leq n$ in arbitrary order. If an interval does not overlap any other intervals in π it is called **strong**.

Intuitively, this definition seems similar to Definition 2.11. As we can see, for example, in Figure 2, the strong modules of a graph always contain consecutively labelled vertices. These facts lead to the following proposition:

Proposition 2.3 ([17, Proposition 1] and [38])**.** *Let $G_\pi = (G, \lambda)$ be a permutation graph and π its permutation. The strong intervals in π are in exact one-to-one correspondence with the strong modules in G_π.*

Remark. This is an important equivalence, because then, several properties that apply to strong intervals can be applied to the strong modules of a permutation graph and vice versa. For example, the number of strong modules $|\mathbb{M}_{\text{str}}(V)|$ in a graph G is in $\mathcal{O}(n)$ [39], as is the number of strong intervals in a permutation [35]

Notation. After identifying the strong modules/intervals of a permutation graph, we can mark them with `‘(’, ‘)’` in the permutation, e.g., see Example 3. For the sake of better readability, I will sometimes omit the brackets for singleton modules. A permutation written this way will be called a *bracketed permutation* Π.

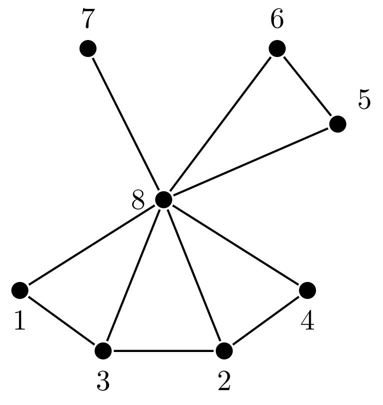
2.2.3 Modular Decomposition Tree

I will now proceed to give definitions for terms depicted in Figure 5. As mentioned above, we can understand any strong module $M \in \mathbb{M}_{\text{str}}(V)$ as an equivalence class of its elements. Also $\mathbb{M}_{\max}(V)$ is a *congruence partition* of V (i.e., by definition $\mathbb{M}_{\max}(V)$ divides V in mutually exclusive subsets, that are also equivalence classes in V) [7, 17]. This allows for the following definition:

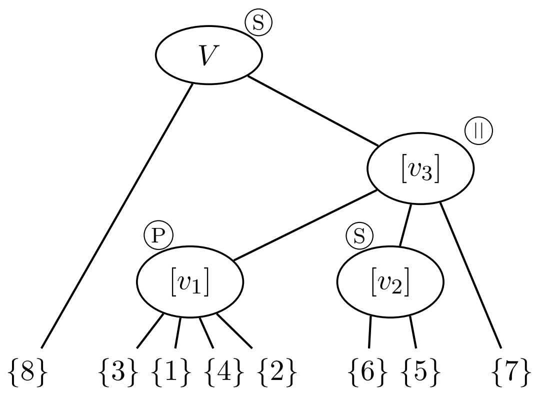
Definition 2.13 (Quotient Graph [4, 17] and [7])**.** Let $G = (V, E)$ be an arbitrary graph and $\mathbb{M}_{\max}(V)$ the set of its maximal strong modules. By choosing one representative vertex $v_i \in M_i$ and its equivalence class $[v_i] \equiv M_i$ for each maximal strong module $M_i \in \mathbb{M}_{\max}(V)$, we can construct $G/\mathbb{M}_{\max}(V) = G[V_{\max}]$, $V_{\max} = \{v_i \mid [v_i] \equiv M_i \in \mathbb{M}_{\max}(V)\}$ the **quotient graph** of G.

Since, by definition, strong modules of a graph $G = (V, E)$ mutually do not overlap, we can order them by inclusion in a so called *modular decomposition tree* (e.g., Figure 5b), where the trivial modules are represented by the root (V) and the leaves (singletons, $\{v\} \subset V$), respectively.

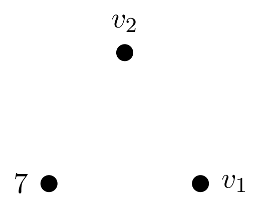
Proposition 2.4 ([28, ‘Modular Decomposition Theorem’])**.** *Let $G = (V, E)$ be a graph. There exists exactly one unique modular decomposition of its vertices $\mathbb{M}_{str}(V)$ where each element is a strong module. These modules $M_i \in \mathbb{M}_{str}(V)$ can be ordered by inclusion in a so called* ***modular decomposition tree*** *$T_G = (V(T_G) = \{p_i \mid [p_i] = M_i \in \mathbb{M}_{str}(V), p_i \in V\}, E(T_G) = \{p_i p_j \mid [p_i] \subset [p_j]\})$, where each node $p_i \in V(T_G)$ represents a strong module $[p_i] = M_i \in \mathbb{M}_{str}(V(G))$ and two nodes p_i, p_j representing strong modules $M_i, M_j \in \mathbb{M}_{str}(V(G))$ are connected by an (undirected) edge if one module contains the other.*



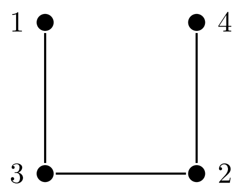
(a) Graph G_{π_3}. Its strong modules are $M_1 = [1 : 4]$, $M_2 = [5 : 6]$ and $M_3 = [1 : 7]$.



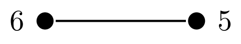
(b) Typed modular decomposition tree of G_{π_3}. The identifiers for each inner node, representing module M_j, are the respective representative vertices v_j.



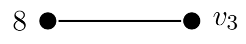
(c) Quotient graph $\textcircled{\parallel}$ G_{M_3} with v_1 and v_2 representing its maximal strong modules M_1 and M_2 respectively.



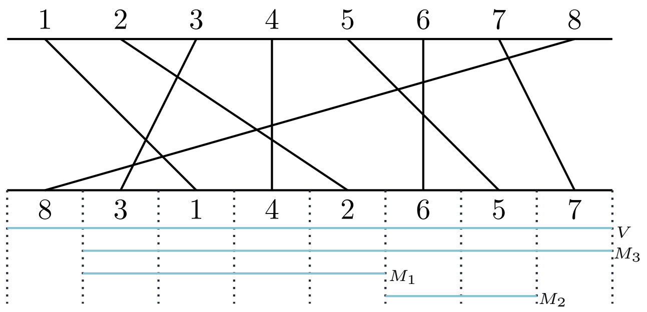
(d) Quotient graph $\textcircled{P}$ $G_{M_1} \cong P_4$.



(e) Quotient graph $\textcircled{S}$ G_{M_2}.



(f) Quotient $\textcircled{S}$ graph G_V, where v_3 represents its maximal strong module M_3.



(g) Line graph with marked strong intervals for each strong module (singleton modules are omitted). The bracketed permutation is therefore $\Pi_3 = ((8)\ (((3)\ (1)\ (4)\ (2))\ ((6)\ (5))\ (7)))$

Figure 5: G_{π_3} and its modular decomposition tree. See also Example 3.

Each inner node represents a strong module $[v_j] \equiv M_j \in \mathbb{M}_{\text{str}}(V)$ and possesses a quotient graph $G_{M_j} = G[M_j]/\mathbb{M}_{\text{str}}(M_j)$ representing the induced subgraph $G[M_j]$.

Before proceeding with some properties of modular decomposition trees of permutation graphs, let us introduce the following notation and subsequent definition.

Notation. Given a modular decomposition tree $T_G = (V(T_G), E(T_G))$ of a graph $G = (V, E)$. The set of children of a node $p \in V(T_G)$ is denoted $\mathcal{C}(p)$, while the set of all leaves below p is denoted $\mathcal{L}(p)$. It should be noted that in a slight abuse of notation, we say that a vertex $v \in V$ is an element of $\mathcal{L}$, when we actually mean that the leaf $\ell_v \in V(T_G)$ representing the singleton set $L_v = \{v\}$ is.

Definition 2.14 (Lowest Common Ancestor)**.** Given a modular decomposition tree $T_G = (V(T_G), E(T_G))$ of a graph $G = (V, E)$ and three of its nodes $p, q, r \in V_T$ representing the modules $P, Q, R \subseteq V$. Node r is an **ancestor** of another node p if there exists a path $(r, c_1, \ldots, c_k, p) \in E_T^{k+2}$ such that $c_1 \in \mathcal{C}(r)$, $c_{i+1} \in \mathcal{C}(c_i)$ and $p \in \mathcal{C}(c_k)$ $p \prec_{T_G} r$ denotes that p lies below r in T_G, i.e., it holds that r is an ancestor of p and p is a **descendant** of r, thus $P \subseteq R$.
Furthermore, $r = \text{lca}(p, q)$ defines the **lowest common ancestor** of two nodes, that is the lowest node r in T_G such that $p \preceq_{T_G} r$ and $q \preceq_{T_G} r$.

Remark. Note, that if p is a descendant of q, then P is a strong module of $G[Q]$.

Notation. Given an inner node p of a modular decomposition tree T_G, where G_π is a permutation graph with permutation π. The leaves below p define a vertex set $\{l \in \mathcal{L}(p)\} = P \subseteq V$ and therefore G_p will denote the quotient graph $G[P]/\mathbb{M}_{\max}(P)$ of the maximal strong modules of $G[P]$ and $\pi_p = \pi[V_{\max}(p)]$, λ_p its permutation and labelling respectively. Given a vertex set $N \subseteq V$, $N_p = \{u \mid u \in V(G_p)$ and $[u_i] \in \mathbb{M}_{\max}(P), 1 \leq i \leq |P|\}$ denotes the vertex set of representative vertices in $V(G_p)$ with respect to N.

Note that the the permutation π_p is different from $\pi[P]$, where we slice π with respect to all vertices in $P \subseteq V$ (a strong interval in π) and obtain a permutation of $G[P]$.

Furthermore, we can assign a type to each of the inner nodes (including the root V) of the modular decomposition tree, depending on the following types of its quotient graph.

Proposition 2.5 (Series, Parallel, Prime [4, (1.8) Satz], [6, Prop. 1.5.1 sqq., especially Thm. 1.5.1] and [28, 'Modular Decomposition Theorem' Thms. 4.1.2 sqq.])**.** *Let $G = (V, E)$ be an arbitrary undirected graph. Each inner node p of its modular decomposition tree T_G, representing a strong module $P \in \mathbb{M}_{str}(V)$, can be assigned exactly one of these mutually exclusive types, depending on its representative quotient graph $G_p = G[P]/\mathbb{M}_{max}(P)$:*
Ⓟ *prime, if and only if G_p and $\overline{G}_p$ are connected,*
Ⓢ *series, if and only if $\overline{G}_p$ is not connected, or*
⓪ *parallel, if and only if G_p is not connected.*

If a node (or a module respectively) is not prime (i.e., Ⓢ or ⓪) it is called *degenerate*. Another important aspect for the modular decomposition of permutation graphs is the fact that for example quotient graphs are also *primitive graphs*, sometimes called *simple decomposable graphs*, or, confusingly *prime graphs*, with very helpful properties. The latter name, *prime*, stems from the fact that these kind of graphs turn out to be indecomposable by modular decomposition and are therefore the elementary ‘building-blocks’ for all graphs. To avoid confusion with the label *prime* or Ⓟ given in Proposition 2.5 the term *primitive* will be used from now on.

Definition 2.15 (Primitive Graph, [7, Definition 2.5])**.** A graph $G = (V, E)$ is called **primitive** if and only if its modular decomposition tree is a star, i.e., G only has trivial modules and $|\mathbb{M}_{\max}(V)| = |V|$.

From this we can deduce that all quotient graphs are indeed primitive graphs by definition [7].

Furthermore, if G is also a permutation graph $G_\pi = (G, \lambda)$ with permutation π, there exists a simple way to deduce its type, or more precisely the type of its root node in T_G. Be reminded of Proposition 2.2 where it was shown that ascending and descending sequences correspond to empty and complete graphs, respectively. By using this proposition, we can deduce the following lemma:

Lemma 4. *A primitive permutation graph $G_\pi = (G, \lambda)$, $\mathbb{M}_{max}(V) = \emptyset$ with permutation π of length $n > 1$ can be typed as follows:*
(i) series if $\pi(1) = n$,
(ii) parallel if $\pi(1) = 1$,
(iii) prime otherwise.

Proof. Using proof by contradiction, let $G_\pi = (G, \lambda)$ be a primitive permutation graph of either ⦷ or Ⓟ type, but with $\pi(1) = n$. Proposition 2.2 shows that G can be no parallel graph. We can therefore assume G to be prime. By Definition 2.9 the vertex $v = \lambda^{-1}(n)$ at rank 1 will be connected to any other vertex $u \in V \setminus \{v\}$, because it has the greatest possible label. Since G is primitive, it only has trivial modules (Definition 2.15) and in particular π contains no strong intervals (Proposition 2.3), other than $[1 : n]$ (the whole permutation), and $[i : i]$ (singletons), $1 \leq i \leq n$. Because we choose G not to be series, $\pi(2) \neq n - 1$. But this already contradicts our assumption that $\pi(1) = n$, as then $\pi[1 : n - 1] = (\pi(2)\ \pi(3)\ \ldots\ \pi(n))$ would constitute a strong interval of π (they all connect to v). So G must be series.

By an analogous reasoning we can show that $\pi(1) = 1$ implies that G is parallel, and since the three types are mutually exclusive, the third case follows. ■

Corollary 1. *The number of nodes $|V(T_G)|$ in a modular decomposition tree T_G of a graph $G = (V(G), E(G))$, $|V(G)| = n$, is in $\mathcal{O}(n)$.*

Proof. For a primitive graph the number of nodes is exactly $n + 1$, i.e., the root and singleton modules. This is the lower bound for $|V(T_G)|$.
An upper bound is given by the structure of T_G. In the worst case, that is the most number of nodes, T_G is a binary tree, since each non-trivial strong module must contain at least two strong children modules by definition. Therefore, with a binary tree having n leaves the upper bound becomes $2n - 1$ and we have proven that $|V(T_G)| \in \mathcal{O}(n)$. ■

Example 3. Going back to the graph G_{π_3} introduced in Example 2 and depicted again in Figure 5a. Without using the exact algorithm of [35], or with any regard for time complexity, we can identify the strong intervals of its permutation, as shown in Figure 5g. The trivial ones: $V = [1:8]$ and the singletons $[1:1]$, $[2:2]$, etc. As well as the non-trivial strong intervals, resulting in the inner nodes of its modular decomposition tree: $M_1 = [1:4]$, $M_2 = [5:6]$ and $M_3 = [2:7]$. Note, that intervals like $[1:6]$, indeed represent modules, but since they overlap $[5:7]$, for example, they are not strong.

We can then extract the permutation for each quotient graph $\pi[M_j]$, $j \in \{1,2,3\}$, by choosing one representative element for each maximal strong module of its induced subgraph, e.g., for $\pi[M_3] = \pi_3(((3)\ (1)\ (4)\ (2))\ ((6)\ (5))\ (7))$ we choose $\{3,5,7\}$ to represent each maximal strong module of $G_{\pi_3}[M_3]$, which yields $\pi[\{3,5,7\}] = (3\ 5\ 7)$. We decrement each element in this permutation while maintaining their order, until we obtain a permutation of $[1:3]$ with $\pi[M_3] = (1\ 2\ 3)$. By Lemma 4, M_3 can then be typed as parallel $\textcircled{\scriptsize\parallel}$.

The process for V (Ⓢ), M_1 (Ⓟ), and M_2 (Ⓢ) is similar and left to the reader, if they choose to check the results of Figure 5.

3 Proposed Data Structure and Algorithms

After finishing the theoretical introduction we can now proceed to the algorithmic part of this thesis. In this chapter I will outline the pipeline and its algorithms which are intertwined with our proposed data structure. Each algorithm, and especially the data structure, should be reviewed with an object-oriented paradigm in mind.

I will give pseudo code for each of our algorithms, and also for the construction of the modular decomposition tree (our data structure) from a bracketed permutation. Other algorithms on the other hand, like the ones of Bergeron et al. [35], will at most be sketched briefly. Their application for our purposes and their place in our proposed pipeline, however, will be part of this chapter.

Therefore I will introduce the outline of our proposed pipeline in the first section, where I also discuss the original notion that led to the base algorithm and why it needs to be extended by additional concepts. The subsequent section will draft the data structure, a typed modular decomposition tree, which is built utilising results on modular decomposition from permutations by Bergeron et al. [35] and is used for generating a permutation for primitive prime graphs. Generating such a permutation follows some results presented by Crespelle and Paul [17] in their work about dynamic permutation graph recognition.

3.1 Base Algorithm

3.1.1 Outline

The original notion that the reader should have in mind is that of Lemma 1, the heredity property of permutation graphs. Given a permutation graph $G_\pi = (G, \lambda)$ we can remove any vertex x (and its connecting edges) and still end up with a permutation graph $G_\pi - x$. This leads to the following question:
Can we recognise a permutation graph as such, by successively building it up from the empty graph in a vertex-by-vertex fashion, and while maintaining a valid permutation at each insertion step?
The answer is: Almost. The algorithms 1a and 1b proposed in Section 3.1.2 only work for certain types of permutation graphs, e.g. only graphs that are primitive and prime, or in general, permutations that allow a maximum of two neighbouring insertion

Algorithm 1a: First part of the insertion algorithm. Scans given permutation for a valid insertion position and constraints for the inserted label.

```
Input: permutation $\pi$ of permutation graph $G_\pi = (V, E, \lambda)$,
       bijective mapping $\lambda : V \mapsto [1 : n]$ of vertices and labels in $V$, and
       neighbourhood $N(\mathsf{x}) \subseteq V$ of the vertex x about to be inserted in $G$
Output: updated permutation $\pi_{+\mathsf{x}}$ and mapping $\lambda_{+\mathsf{x}}$, or
        the result that $G + \mathsf{x}$ is not a permutation graph
1  /* initalize arrays at starting positions of the iterations,
      where no neighbours are present                                  */
2  lowerB_r[$|\pi| + 1$] $\leftarrow 0$
3  upperB_r[$|\pi| + 1$] $\leftarrow |\pi| + 1$
4  lowerB_l[1] $\leftarrow 0$
5  upperB_r[1] $\leftarrow |\pi| + 1$
6  /* start iterations to find insertion position and label by
      checking the neighbours                                          */
7  for $i = 2, \ldots, |\pi| + 1$ do                      // from left to right
8      yLabel $\leftarrow \pi(i-1)$            // vertex label at $(i-1)$-th position
9      y $\leftarrow \lambda^{-1}$(yLabel)                 // lhs neighbour of x in $\pi$
10     if y $\in N(\mathsf{x})$ then
11         upperB_l[$i$] $\leftarrow \min\{$upperB_l[$i-1$], yLabel$\}$
12         lowerB_l[$i$] $\leftarrow$ lowerB_l[$i-1$]
13     else
14         lowerB_l[$i$] $\leftarrow \max\{$lowerB_l[$i-1$], yLabel$\}$
15         upperB_l[$i$] $\leftarrow$ upperB_l[$i-1$]
16     end
17 end
18 for $i = |\pi|, |\pi| - 1, \ldots, 1$ do                 // from right to left
19     yLabel $\leftarrow \pi(i)$                  // vertex label at $i$-th position
20     y $\leftarrow \lambda^{-1}$(yLabel)                 // rhs neighbour of x in $\pi$
21     if y $\in N(\mathsf{x})$ then
22         lowerB_r[$i$] $\leftarrow \max\{$lowerB_r[$i+1$], yLabel$\}$
23         upperB_r[$i$] $\leftarrow$ upperB_r[$i+1$]
24     else
25         upperB_r[$i$] $\leftarrow \min\{$upperB_r[$i+1$], yLabel$\}$
26         lowerB_r[$i$] $\leftarrow$ lowerB_r[$i+1$]
27     end
28 end
```

positions. For that reason, we need to check the graph's modular decomposition beforehand. This process and the algorithmic principles leading to it, are discussed in Section 3.2.

In the negative case, where our algorithm does not find an insertion position, however, we compute the modular decomposition tree and manipulate a certain set of its nodes using results of Crespelle and Paul [17]; leading to either a modular

Algorithm 1b: Second part of the insertion algorithm. Finalizes mappings according to the found insertion position.

```
/* combine constraints and search for possible insertion
   position, only the leftmost position will be returned      */
for $i = 1, \dots, |\pi| + 1$ do          // check validity of each position
    upperBound ← $\min\{$upperB_l[$i$], upperB_r[$i$]$\}$
    lowerBound ← $\max\{$lowerB_l[$i$], lowerB_r[$i$]$\}$
    if upperBound = lowerBound + 1 then
        xLabel ← upperBound
        increment all labels in $\pi$ and $\lambda$ greater or equal than xLabel by 1
        $\pi_{+x}$ ← insert xLabel at position $i$ in $\pi$
                        // shift rhs elements of $i$ in $\pi$ to the right
        $\lambda_{+x} \leftarrow \lambda \cup \{$xLabel $\mapsto$ x$\}$
        return $\pi_{+x}, \lambda_{+x}$
    end
end
return $G + x$ is not a permutation graph.
```

decomposition tree from which we can construct π_{+x}, or to the result that $G + x$ is not a permutation graph. That topic is discussed in detail in Section 3.3.

3.1.2 Vertex Insertion for Primitive Graphs

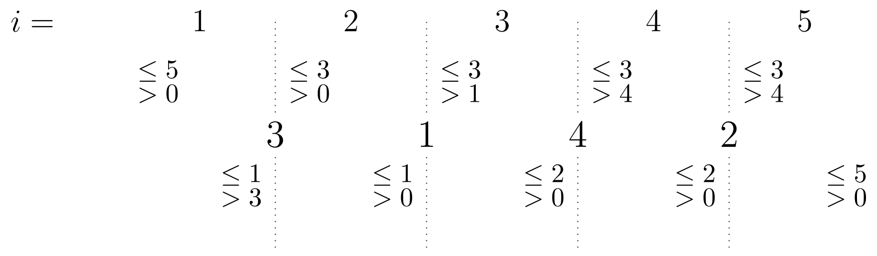


Figure 6: Example run of Algorithm 1a on $\pi = (3\ 1\ 4\ 2)$ with $N(x) = 3$. i marks the possible insertion positions, whereas the constraints of x's label are updated with each step of the respective `for`-loop. The element in π leading to the respective constraints is the one depicted closest to each pair, e.g., at insertion position 2 it is $\pi(1) = 3$ for $0 < \lambda(x) \leq 3$ and $\pi(2) = 1$ for $0 > \lambda(x) \leq 1$. 2 is also one of the two possible insertion positions leading to the graph depicted in Figure 7 with $\lambda(x) = 1$ and permutation $\pi_{+x} = (4\ 1\ 2\ 5\ 3)$

First, consider a given permutation graph $G_\pi = (G, \lambda)$, $G = (V, E)$ of permutation π and a vertex x with neighbourhood $N(x) \subseteq V$. Algorithm 1a scans every possible insertion position two times, once for each direction, and checks whether the left-hand side (resp. right-hand) vertex y, represented by its label $\lambda(y)$, is also in $N(x)$. Here position 1 is left to the first element of π while position $n+1$ is right to the last (n-th) element. For each step the upper and lower bound for x's label at that position are set according to Definition 2.9 by regarding the (non-)neighbourhood of vertex y and comparing its label and the previously set constraints.

Let us now proceed to proving the correctness of Algorithm 1a and Algorithm 1b (collectively called Algorithm 1). It will be shown in Theorem 1 that Algorithm 1 finds an insertion position in π and a label for the inserted vertex x if G is a primitive prime permutation graph.

Proposition 3.1 ([17, Lemma 10])**.** *Let π be a permutation and λ a labelling of a primitive permutation graph $G = (V, E)$ with $|V| = n$. Given a vertex x with neighbourhood $N(x) \subseteq V$ to be inserted into G, then $G + x$ is a permutation graph if and only if $\forall u \in N(x), D_\lambda(u, x) \leq 0$ iff $D_\pi(x, u) < 0$ and $\forall v \in \overline{N}(x), D_\lambda(v, x) \leq 0$ iff $D_\pi(x, v) \geq 0$. For a label $\lambda(x)$ and insertion position $\pi^{-1}(x)$ of x, to match the definitions of D_λ and D_π respectively.*

Notation. Given a permutation graph $G_\pi = ((V, E), \lambda)$ with permutation π and a vertex $x \notin V$ with neighbourhood $N(x) \subseteq V$. We denote the permutation after inserting x into G, if possible, as $\pi^{+x(l,r)}$. Where l is the label $\lambda_{+x}(x)$ of x in the new permutation and r its rank $\pi^{-1}_{+x}(x)$ resulting from the insertion process.

The proposition above was shown by Crespelle and Paul and gives us the opportunity to prove the algorithm's correctness, simply by showing that the neighbourhoods for all involved vertices is retained. We will show that Algorithm 1 finds a matching label and rank for x, called *insertion position* by Crespelle and Paul, that satisfies the conditions above. This leads to the following theorem:

Theorem 1 (Correctness of Algorithm 1)**.** *For the neighbourhood $N(x)$ of a vertex x to be inserted in the permutation π of a primitive prime permutation graph, Algorithm 1 returns a label and rank for x in permutation $\pi^{+x(l,r)}$ or the result that $G + x$ is not a permutation graph and stops in $\mathcal{O}(n)$ $(n = |V|)$ time.*

Proof. Let $G_\pi + x = ((V + x, E + xN(x)), \lambda_{+x})$ be the resulting permutation graph of π_{+x} and let $l = \lambda_{+x}(x)$ and $r = \pi^{-1}_{+x}(x)$. First consider the permutation π_{+x} since no vertices from G were removed, we can restrict $\pi_{+x}[V]$. This is clearly the inverse process of Algorithm 1b and thus all remaining vertices retain their connectivity between each other.

Now consider a vertex $y \in N(x)$. Algorithm 1a finds a label for x that is smaller than $\lambda(y)$ if $\pi^{-1}_{+x}(y) < r$ (line 11). Thus $D_{\pi+x}(x, y) = r - \pi^{-1}_{+x}(y) > 0$ and $D_{\lambda+x}(y, x) = \lambda_{+x}(y) - l > 0$ which satisfies the conditions of Proposition 3.1. The same holds for the case, when y is to the right of x in π_{+x} (line 22) but with all inequalities reversed in the other direction, i.e., $\lambda(y) \leq l \implies D_{\lambda+x}(y, x) \leq 0$ and $\pi^{-1}_{+x}(y) > r \implies D_{\pi+x}(x, y) < 0$. This concludes the case, if $y \in N(x)$.

The last part of the correctness proof will consider the case, where $y \in \overline{N}(x)$ and is somewhat analogous to the previous one. We see in lines 14 and 25 that $D_{\pi+x}(x, y)$ and $D_{\lambda+x}(y, x)$ always have opposite signs, and therefore also satisfy the conditions of Proposition 3.1.

The algorithm clearly spends $\mathcal{O}(|\pi|) = \mathcal{O}(n)$ time, since π is iterated over exactly twice in Algorithm 1a and the time spend for Algorithm 1b is at most $3n$ (iterate boundary-array, update π and λ), if x has to be inserted at the front position with label 1. ■

3.1.3 Finding Twins

Procedure `findTwin(`π_{+x}, λ^{-1}_{+x}, $\pi^{-1}_{+x}(x)$`)`

```
$r \leftarrow \pi^{-1}_{+x}(x)$                                  // better readibility
$l_x \leftarrow \pi_{+x}[r]$                                     // get label at rank $r$
if $r > 1$ then
    $l_l \leftarrow \pi_{+x}[r\text{- }1]$                       // lhs neighbour in $\pi_{+x}$
    if $|l_l - l_x| = 1$ then return $\lambda^{-1}_{+x}[l_l]$
else if $r < |\pi_{+x}|$ then
    $l_r \leftarrow \pi_{+x}[r + 1]$                             // rhs neighbour in $\pi_{+x}$
    if $|l_r - l_x| = 1$ then return $\lambda^{-1}_{+x}[l_r]$
else
    return $x$ has no twin
end
```

Procedure `isTwin(`$N(x)$, t, λ, π`)`

```
foreach vertex $v \in N(x) \setminus \{t\}$ do
    if $D_\lambda(v,t)D_\pi(t,v) < 0$ then                         // $v \notin N(t)$
        return $t$ is not a twin of $x$ in $G_\pi$
    end
end
return $t$ is a twin of $x$ in $G_\pi$
```

Beside performing the insertion for primitive prime graphs, Algorithm 1 can detect the so called *twin* of the inserted vertex x, if it exists. Recall, that twin vertices share the exact same neighbourhood in a graph, except for maybe each other. (See also Definition 2.3) This means that in a given graph, twins always belong to the same (non-trivial) strong module and are therefore of particular interest for identifying almost trivial sub-cases for finding permutations when inserting a vertex. To summarise, we can find those following two properties:

Proposition 3.2 (Crespelle and Paul [17] Theorem 4)**.** *Let $G_\pi = (G, \lambda)$, $G = (V, E)$ be a labelled primitive prime permutation graph and x be a vertex to be inserted into G_π such that V is not uniform with respect to x. There are at most two insertion positions for x in π and at most two possible labels. x has a twin in G if and only if there exist two insertion positions next to each other and their labels are interchangeable and differ by one.*

The result of Algorithm 1, namely the insertion rank $\pi^{-1}_{+x}(x)$ and insertion label $\lambda_{+x}(x)$, together with Proposition 3.2 can be used to identify a twin in π_{+x}. This can be even done in constant time, given the inverse mapping of λ_{+x} as shown in Procedure findTwin. Or in time $\mathcal{O}(|N(x)|)$ for a given vertex t as shown in Procedure isTwin.

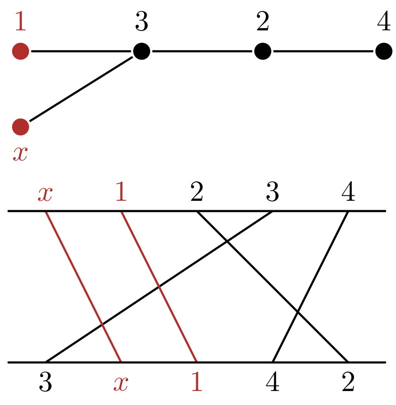


Figure 7: Twins have lines next to one another in the linear orders of their line graph. In this example x and the vertex labelled 1 are twins.

Corollary 2. *Let $G_\pi = (G, \lambda)$ be a labelled primitive prime permutation graph and $x \notin V(G)$ a vertex that is inserted into G_π. If x has a twin $t \in V(G)$ then $\{x, t\}$ is the only non-trivial strong module of $G_\pi + x$.*

Proof. Since G_π was primitive and prime, it had no strong modules, except for trivial ones. By Proposition 2.3 this means that π has also no strong intervals. Inserting x into π next to y will result in a strong interval of at least length two, since $|\lambda_{+x}(x) - \lambda_{+x}(y)| = 1$ and $|\pi^{-1}_{+x}(x) - \pi^{-1}_{+x}(y)| = 1$ by Proposition 3.2. Suppose now τ is the strong interval containing x and its twin t and possibly other elements. Let v be the element next to t in π, i.e. $\pi^{-1}(v) = \pi^{-1}(t) - 1$ or $\pi^{-1}(v) = \pi^{-1}(t) + 1$ if they exist.

Without loss of generality, let $\pi^{-1}_{+x}(x) > \pi^{-1}_{+x}(t)$ (x is inserted to the right of t in π), since their labels can be swapped arbitrarily. Then u can not be an element of τ, because its label is either incremented by 1 if it had a larger label than t in π or, conversely, it remains the same if it was smaller. In the former case it holds that $\lambda_{+x}(v) > \max\{\lambda_{+x}(x), \lambda_{+x}(t)\} + 1$ and $\lambda_{+x}(v) < \min\{\lambda_{+x}(x), \lambda_{+x}(t)\} - 1$ in the latter. Thus the strong interval containing x and t has at most length two, and we can conclude that the new (and only) non-trivial strong module of $G_\pi + x$ is indeed $\{x, t\}$. ■

Remark. This holds independent of connectivity between the twins since their labels are interchangeable. Also the created strong module $\{x, t\}$ is either Ⓘ or Ⓢ, depending on said connectivity.

3.2 Data Structure

3.2.1 Finding a Representation

Since Algorithm 1 does only cover the cases, where the permutation π of a graph G has an insertion position (i.e. a rank and a label) for x, the other cases have to be covered otherwise. The most sensible approach is using modular decomposition theory (see Section 2.2 for an introduction). It serves simultaneously as an underlying data structure (namely the *modular decomposition tree*) and a guideline for the algorithmic design. The modular decomposition tree can be built in a bottom-up manner, as described in Algorithm 2, and traversed in linear time ($\mathcal{O}(n)$), because the number of nodes in such a tree is linear with $n = |V|$ as was shown in Corollary 1. For Algorithm 2 the strong modules (strong intervals) of G_π (in π) need to be computed to provide the bracketed permutation Π for the desired tree-structure. One can use Bergeron et al.'s [35] linear-time algorithm that uses LIFO-queues and several properties of permutations to obtain a string, representing the bracketed permutation Π of π. I will not provide their algorithm here, since that would be beyond the scope of this thesis. But essentially they iterate a constant amount of time over the permutation to find, for each element, the indices of left and right bounds of the strong intervals they are contained in.

Proposition 3.3 (Bergeron et al. [35, see especially Prop. 9])**.** *Given a permutation π then the bracketed permutation Π can be computed in $\mathcal{O}(n) = \mathcal{O}(|\pi|)$ time. Also the number of strong intervals and thus the length of a bracketed permutation Π is in $\mathcal{O}(n)$ space.*

3.2.2 Proposed Implementation

We propose Algorithm 2 to build a representation of the modular decomposition tree T_G of a permutation graph G_π with permutation π. It relies on abstract objects of the type `Node` that correspond to the nodes of T_G. Each `Node` possesses a type (⦷, Ⓢ, Ⓟ or `LEAF`) and is arranged in doubly-linked lists, where each parent `Node` p is doubly-linked to a list of its children $\mathcal{C}(p)$, which are themselves arranged in an ordered doubly-linked list, such that they represent the permutation π_p of the quotient graph G_p of their parent.

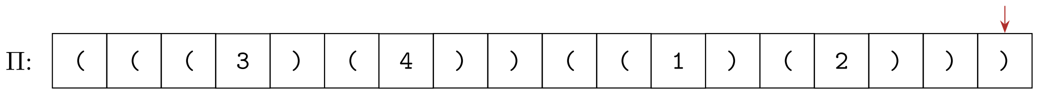
Unique to leaves is a stack of nodes `[`$p_1, \dots, p_j$`]` that contains pointers to all ancestor nodes p_i, $1 \le i \le j$, where it is the minimum element (by label) of the quotient permutation π_p of p_i's parent, i.e. if $\lambda_p(p_i) = 1$. These stacks are used for each child to store the information of its label and rank in its parent's quotient permutation (see also example in Figure 8) and can be discarded after the building process, since they are no longer needed later on. This ensures that a query for the permutation of the quotient graph G_p of any inner node p can be resolved in $\mathcal{O}(|\mathcal{C}(p)|)$ time, where $\mathcal{C}(p)$ is the set of children of p.

Of each inner node p, the minimum and maximum rank in π, i.e., the bounds of the strong interval of the represented strong module $P \subseteq V$, is stored. Additionally, to also achieve fast queries for permutations $\pi[P]$ of the respective induced subgraphs $G_\pi[P]$ of a strong module P, the two mappings between vertices $v \in V$ and their

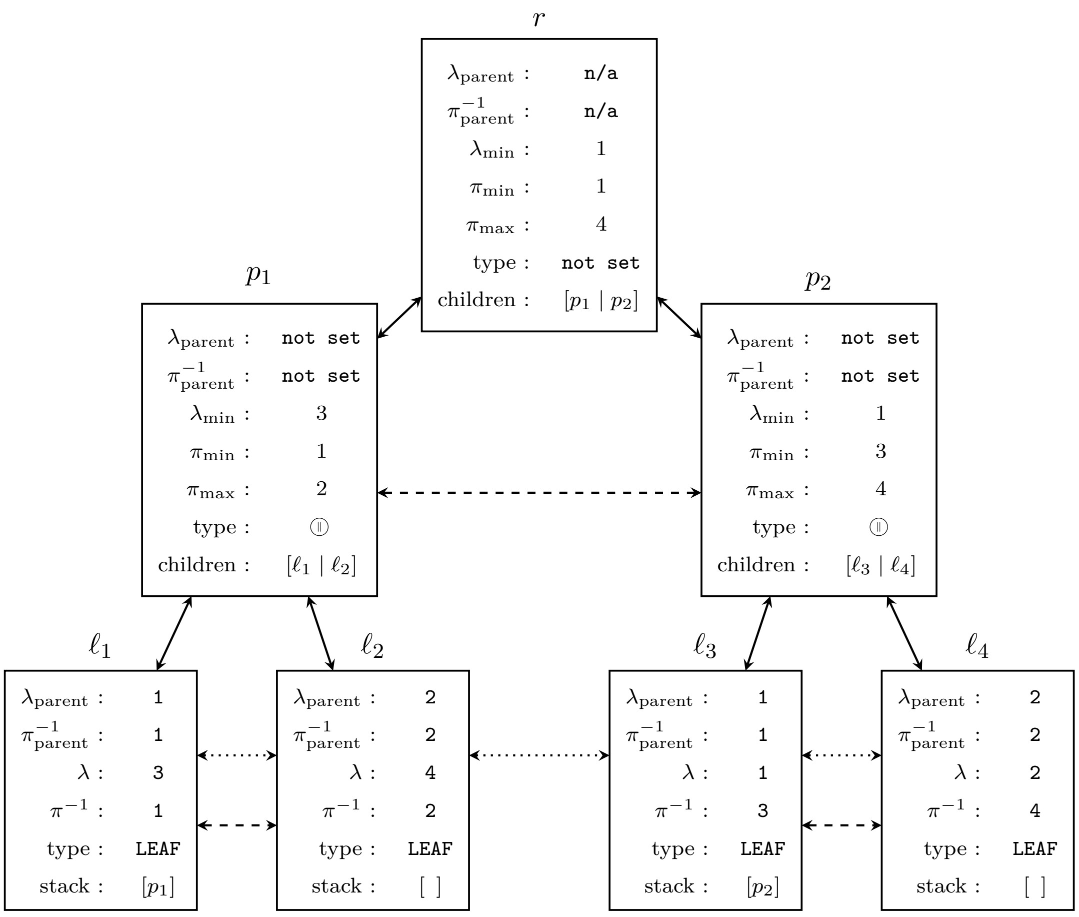
Algorithm 2: Algorithm to build a the data structure from a bracketed permutation Π that is derived from π of a permutation graph G_π, e.g., given as a `char` array.

```
Input: Bracketed permutation Π, where all strong intervals are enclosed
       with brackets.
Output: root Node of a modular decomposition tree.
Procedure buildTree(Π):
    root ← Node()                                  // create return object
    $p$ ← root                                           // active node
    current_rank ← 1        // current rank in $\pi$ (without brackets)
    foreach char in Π do   // skip first char, root already created
        if char is '(' then
            $p_{\text{new}}$ ← Node()  // will be set up after all its children
            Set $p$ as parent of $p_{\text{new}}$ and add $p_{\text{new}}$ to $\mathcal{C}(p)$
            $p \leftarrow p_{\text{new}}$
        else if char is an Integer then          // active Node is a leaf
            $\pi^{-1}(p)$ ← current_rank
            $\lambda(p)$ ← char
            current_rank ← current_rank + 1
        else if char is ')' then     // end of module, all $\mathcal{C}(p)$ set up
            if $p$ is not a leaf then
                foreach $p_i \in \mathcal{C}(p)$ do Assign respective rank $\pi^{-1}_{\text{parent}}(p_i)$ end
                finaliseLabelling($p$)                         // see below
                Assign decomposition type ($\textcircled{\parallel}$, $\textcircled{P}$, $\textcircled{S}$) to $p$.
            end
            Push $p$ on the stack of its minimal (by label) leaf.
            if $p$ is not root then
                Update parent of $p$ by updating $\lambda_{\min}$, $\pi_{\min}$ and $\pi_{\max}$ for
                  leaves.
                $p$ ← parent of $p$
            end
        end
    end
return root

Procedure finaliseLabelling($q$):
    $\Lambda_\ell \leftarrow \lambda_{\min}(q)$         // minimal leaf label of current iteration
    $\Lambda_c \leftarrow 1$          // label assigned to child in current iteration
    while $\Lambda_c \leq |\mathcal{C}(q)|$ do
        $\ell_v \leftarrow \mathcal{L}(T_G)[\Lambda_\ell]$          // call by label from leafset $\mathcal{L}(T_G)$
        $c \leftarrow \ell_v$.stack.pop()
        $\lambda(c) \leftarrow \Lambda_c$
        $\Lambda_c \leftarrow \Lambda_c + 1$
        $\Lambda_\ell \leftarrow \Lambda_\ell + |\mathcal{L}(c)|$     // skip $|\mathcal{L}(c)| = \pi_{\max}(c) - \pi_{\min}(c) + 1$ leaves
    end
end
```



(a) Depiction of the char array for the bracketed permutation Π. The red pointer marks the `char` read in the current iteration (all but the last character have been processed).


r
λ_{parent} : n/a
π^{-1}_{parent} : n/a
$\lambda_{\min}$: 1
$\pi_{\min}$: 1
$\pi_{\max}$: 4
type : not set
children : $[p_1 \mid p_2]$
p_1
λ_{parent} : not set
π^{-1}_{parent} : not set
$\lambda_{\min}$: 3
$\pi_{\min}$: 1
$\pi_{\max}$: 2
type : Ⓘ
children : $[\ell_1 \mid \ell_2]$
p_2
λ_{parent} : not set
π^{-1}_{parent} : not set
$\lambda_{\min}$: 1
$\pi_{\min}$: 3
$\pi_{\max}$: 4
type : Ⓘ
children : $[\ell_3 \mid \ell_4]$
ℓ_1
λ_{parent} : 1
π^{-1}_{parent} : 1
λ : 3
π^{-1} : 1
type : LEAF
stack : $[p_1]$
ℓ_2
λ_{parent} : 2
π^{-1}_{parent} : 2
λ : 4
π^{-1} : 2
type : LEAF
stack : [ ]
ℓ_3
λ_{parent} : 1
π^{-1}_{parent} : 1
λ : 1
π^{-1} : 3
type : LEAF
stack : $[p_2]$
ℓ_4
λ_{parent} : 2
π^{-1}_{parent} : 2
λ : 2
π^{-1} : 4
type : LEAF
stack : [ ]

(b) MDT of data structure for permutation Π. The dotted lines mark the doubly-linked relations of leaves, while a dashed line marks the same for children of the same parent node. The solid lines mark the parent-child-relation.

Figure 8: Schematic visualisation of the data structure that shows the state before the last step of an example run of Algorithm 2. The bracketed permutation Π = `(((3) (4)) ((1) (2)))` was processed for the preceding 17 iterations (see Figure 8a) and the active node now is r (root of the tree). Algorithm 2 now branches into line 14 where the rank in the quotient permutation π_r is assigned to the respective child by its position in the array of children of r ($[p_1 \mid p_2] \rightarrow \pi_r^{-1}(p_1) = 1, \pi_r^{-1}(p_2) = 2$). Then we can compute the labelling of the quotient as follows: First find the minimal leaf with $\lambda_{\min}$ ($\lambda(\ell_3) = 1 = \lambda_{\min}$), then pop the topmost element of its LIFO-queue (p_2) and that node will be the child with label 1. Skip $|\mathcal{L}(p_2)| = \pi_{\max} - \pi_{\min} + 1 = 2$ leaves to get the label of the next leaf $\lambda(\ell_1) = 3$. Repeat this process until all children were assigned a label. In this example $\lambda_r(p_1) = 2, \lambda_r(p_2) = 1$. These two steps produce the quotient permutation $\pi_r = (2\ 1)$ and we can assign the decomposition type, Ⓢ in this case, using Lemma 4 in line 18.

respective leaf $\ell_v \in \mathcal{L}(V)$, $v \mapsto \ell_v$, $\ell_v \mapsto v$, should be stored as well as the permutation π itself (without brackets). This allows fast ($\mathcal{O}(1)$) referencing, e.g., to slice out certain strong intervals (modules) from π. It should be noted that with these mappings and the ordering of leaves according to π, the strong intervals $\pi[P]$ are in one-to-one correspondence with the ordered leafsets $\mathcal{L}(p)$ in the proposed data structure.

Also note, that for a child p of a parent node q, the index of p in the doubly-linked list $\mathcal{C}(q)$ is always exactly the rank of p in π_q, the quotient permutation of q. So swapping two children must always update both.

Theorem 2. *The time complexity of* `buildTree` *is in* $\mathcal{O}(n)$, $n = |\pi|$. *The data structure generated by*`buildTree` *is in* $\mathcal{O}(n)$ *space.*

Proof. Every operation in `buildTree` can be done in constant time, except for lines 16 and 17 (assignment of labels and ranks for the children nodes). But, by definition, each node can only be the child of exactly one parent, hence in a the whole run of Algorithm 2 every node will only be accessed one time in these ways and thus only a constant amount of time. The algorithm ends after $|\Pi|$ iterations of the `for`-loop in line 5, and since $|\Pi| \in \mathcal{O}(n)$ we can conclude that `buildTree` has a time complexity in $\mathcal{O}(n)$.

For the space complexity we can refer to Corollary 1 for the size of the modular decomposition tree. The additional mappings can also be implemented in linear space. ■

3.2.3 Constructing Permutations of Representative Graphs

It was shown earlier that retrieving the permutation of quotients and slices $\pi[S]$ of the original permutation works intuitively from that proposed data structure. But in the following section I will introduce rearrangement operations that change the rank and labels of each inner node, and by extension the leafs, that would require a costly updating of each leaf in the process. Since we have to stick to linear time complexity, another solution is needed.

Because the rank for each leave is given by the structure of the tree, we can construct $\pi[S]$ by a top-down iteration of T_s in the order of each node's quotient permutation as shown in Procedure getPi.

Corollary 3. *The quotient permutation* π_q *and the permutation* $\pi[Q]$ *of a node* $q \in V(T_G)$, *as well as their labellings can be constructed in* $\mathcal{O}(|Q|)$ *time.*

Proof. In Procedure getPi, each node or leaf is visited once and the calculation of the leaf sums can be done in $|\mathcal{C}(q)|$ for each node q, which results in the desired linear time.
The quotient permutation $\pi[Q]$ can be retrieved even in $\mathcal{O}(|\mathcal{C}(q)|)$ time, since the children are arranged in a doubly linked list ordered by their ranks. ■

Procedure getPi traverses the tree in a top-down manner. Since the children are ordered by their parent's quotient permutation this allows us 'count' how many labels are already assigned to leaves in the same module in the array $\Sigma_{\mathcal{L}}$. If the

Procedure getPi(r)

```
Input: Root Node $r$ of a (sub-)tree $T_r$ of a graph $G[R]$ whose permutation
       $\pi[R]$ is to be retrieved.
Output: Permutation $\pi[R]$ and labelling $\lambda_\pi[R]$ restricted to $R$.
1  $\varrho \leftarrow 1$          // count current prosition in $\pi$ (processed leaves)
2  $s \leftarrow$ pre_order($r$)   // use inverse result of post_order procedure
     in Algorithm 3, children are doubly-linked according to the
     quotient permutation of their parent
3  while $s$ is not empty do
4      $q \leftarrow s$.pop()
5      if $q$ is root $r$ then
6          $\Lambda(q) \leftarrow 1$      // $\Lambda$ is a mapping containing token labels for
             inner nodes
7      else
8          $p \leftarrow$ parent of $q$
9          $\Lambda(q) \leftarrow \Lambda(p) + \Sigma_{\mathcal{L}}(p)[\lambda_p(q)]$  // $\lambda_p(q)$ is the label of $q$ in its
             parent's quotient
10     end
11     if $q$ is a leaf then                       // update (inverse-)mappings
12         $\lambda(q) \leftarrow \Lambda(q)$
13         $\pi(\varrho) \leftarrow \lambda(q)$
14         $\varrho \leftarrow \varrho + 1$
15     else
16         $\Sigma_{\mathcal{L}}(q)[1] \leftarrow 0$
17         for $i \in \{2, \dots, |\pi_q|\}$ do       // $\pi_q$ is the quotient permutation
18             $c \leftarrow \lambda_q^{-1}(i)$              // child labelled $i$ in qotient of $q$
19             $\Sigma_{\mathcal{L}}(q)[i] \leftarrow \Sigma_{\mathcal{L}}(q)[i - 1] + |\mathcal{L}(c)|$ // $|\mathcal{L}(c)| = \pi_{\max}(c) - \pi_{\min}(c) + 1$
20         end
21     end
22 end
23 return $\pi$, $\lambda$      // mappings are bijective and their inverses are
     also needed
```

inverse labelling λ_q^{-1} in line 18 is not known, we can replace this step by iterating once over the doubly-linked list of children outside of the surrounding `for`-loop and assign the value λ_{parent} to the identical index in an array we can then refer to in line 18.

3.3 Generating Insertion Positions

3.3.1 Motivation and Linking the Tree

To generate a suitable permutation we have to rearrange the modular decomposition tree T_G in such a way that we do not add or remove any edges of the given graph

Algorithm 3: `link_tree` and `post_order` procedures.

```
Input: Root Node $r$ of a modular decomposition tree $T_G = (V(T_G), E(T_G))$
       and neighbourhood $N(x)$ of the inserted vertex $x$.
Output: Root $r$ of linked tree $T_G$.
Procedure link_tree($r$):
    $s \leftarrow$ post_order($r$)
    while $s$ is not empty do          // initalise attribute for all nodes
        $p \leftarrow s$.pop()
        $L_x(p) \leftarrow$ 00                              // 00 $\equiv$ not set
    end
    $s \leftarrow$ post_order($r$)
    while $s$ is not empty do
        $p \leftarrow s$.pop()
        if $p$ is a leaf then
            if $\lambda^{-1}(p) \in N(x)$ then               // $\lambda^{-1}(p) \equiv v \in V(G)$
                $L_x(p) \leftarrow L_x(p)$ & LINKED
            else
                $L_x(p) \leftarrow L_x(p)$ & NOTLINKED
            end
        end
        if $p$ is not $r$ then
            $q \leftarrow$ parent of $p$
            $L_x(q) \leftarrow L_x(q)$ | $L_x(p)$        // 00 | xx = xx, 10 | 01 = 11
            $\mathcal{C}_m(q) \cup \{p\}$               // keep track of mixed children
        end
    end
return $r$

Procedure post_order($r$):
    $s_1$.push($r$)                        // postorder stack (LIFO-queue)
    $s_2$.push($r$)                        // LIFO-queue for building $s_1$
    while $s_2$ is not empty do
        $q \leftarrow s_2$.pop()
        while $\mathcal{C}(q) = \emptyset$ do           // leaves have no children
            $q \leftarrow s_2$.pop()                    // remove all leaves
        end
        $s_1$.push($\mathcal{C}(q)$)
        $s_2$.push($\mathcal{C}(q)$)
    end
return $s_1$
```

G_π and use Algorithm 1 for specific tasks. After rearranging, we can construct π_{+x} from that updated tree. But the question remains, what operations we can actually perform and how many nodes in T_G need rearranging, while being constricted to linear-time operations only. As it will turn out, the authors Crespelle and Paul [17] in their work about permutation graph recognition were able to identify exactly the node q in T_G, they call *insertion node* (see also Section 3.3.2), whose subtree $T_q = (V(T_{G[Q]}), E(T_{G[Q]}))$ is the only one that requires rearranging. The obtaining of q and the rearrangement of its sub-tree will be the topic for the ensuing sections.

Recall, that $G + x$ describes a graph G and a vertex x that is inserted into G according to a given neighbourhood $N(x)$. For the following steps to work, T_G needs to be *linked*. Thus, consider the following notation:

Notation. For a graph $G = (V, E)$, its modular decomposition tree T_G and a vertex $x \notin V$ with neighbourhood $N(x) \subseteq V$ that shall be inserted into G, $\mathcal{C}_{\mathrm{nl}}(p) \subseteq \mathcal{C}(p)$ denotes the set of not linked children of an inner node p in T_G, i.e., $\mathcal{C}_{\mathrm{nl}}(p) = \{p_j \in \mathcal{C}(p) \mid P_j \subseteq \overline{N}(x)\}$. Similarly we denote $\mathcal{C}_l(p) = \{p_j \in \mathcal{C}(p) \mid P_j \subseteq N(x)\}$ as the set of linked children. Moreover, $\mathcal{C}_{\mathrm{m}}(p) = \mathcal{C}(p) \setminus (\mathcal{C}_{\mathrm{nl}}(p) \cup \mathcal{C}_{\mathrm{l}}(p))$ denotes the set of mixed children.

As seen in Algorithm 3, `link_tree` is a bottom-up process that uses a postorder depth-first traversal to assign a linking-type (`LINKED`, `NOTLINKED`, `MIXED`) to each node. This particular implementation uses bit masks for the four states of linking type $\mathtt{L}_x$ with respect to the inserted vertex x, i.e., `00` for `not set`, `01` for `LINKED`, `10` for `NOTLINKED`, and finally `11` for `MIXED`, so each state can be set by using bit-wise operations `&` and `|` (`AND`, `OR`).

For the postorder stack creation the procedure `post_order` uses two stacks (`LIFO`-queues). The implementation of these stacks should support the usual operations `pop()` and `push()`, with the only peculiarity being that if the input of `push()` is a set of objects S, each element in S is pushed on the stack separately in the order given by S, if it exists.

Corollary 4. *Linking can be done in $\mathcal{O}(n)$ time.*

Proof. In Algorithm 3 each node is visited a constant amount of time and since there are $\mathcal{O}(n)$ nodes (Corollary 1), the algorithm stops after this time when all nodes are linked. ■

3.3.2 Finding the Insertion Node

The following result of Crespelle and Paul [17] shows that the insertion node q is the only node whose strong modules need splitting up or rearranging.

Proposition 3.4 ([17, Lemma 8])**.** *Let q be the insertion node of a vertex x in a Graph G. Let $G' = G + x$. G is a permutation graph if and only if the induced subgraph $G'[Q'] = G[Q] + x$ is a permutation graph. Moreover, if G' is a permutation graph, $T_{G'}$ is constructed from T_G by replacing the subtree T_q of T_G with $T_{G'[Q']}$, and by replacing the (quotient) permutations of nodes in T_q with ones in $T_{G'[Q']}$.*

By extension this means, that if x can be inserted into $G[Q]$ then it can also be inserted into G as a whole.

To find the insertion node q, the given modular decomposition tree T_G of a permutation graph G needs to be built from its bracketed permutation and linked according to the inserted vertex x by Algorithm 2 and Algorithm 3.

If we consider a *uniformly* linked (or notlinked) strong module $P \subseteq V(G)$ and its corresponding node $p \in V(T_G)$, it is easy to see that when augmenting $G[P]$ with x to $G[P] + x$, we can obtain a valid permutation with the following lemmas.

Lemma 5. *Inserting a vertex x with neighbourhood $N(x) = V$ in permutation graph $G_\pi = ((V, E), \lambda)$ with permutation π, results in $\pi^{+x(1,n+1)}$ for the graph $G + x$, i.e. x has rank $\pi^{-1}_{+x}(x) = n + 1 = |V| + 1$ and label $\lambda(x) = 1$.*

Proof. Since the original ranks of the elements in permutation π remain unchanged and their respective labels are incremented by 1, their connectivity does not change as well. It also holds that inserting x into the last position of permutation π connects

Algorithm 4: Algorithm to find the insertion node q of the maximal subtree that has to be rearranged when inserting x.

```
   Input: Root Node $r$ of a linked modular decomposition tree
          $T_G = (V(T_G), E(T_G))$.
   Output: Insertion node $q$
1  Procedure find_inode($r$):
2      $q \leftarrow r$
3      while $q$ is MIXED do          // if root is not mixed, no node is
4          if $\mathcal{C}_\mathrm{m}(q) \neq 1$ then return $q$
5          switch decomposition type of $q$ do
6              case Ⓢ do
7                  foreach $p \in \mathcal{C}(q) \setminus \mathcal{C}_\mathrm{m}(q)$ do
8                      if $p$ is NOTLINKED then return $q$
9                  end
10                 $\{q\} \leftarrow \mathcal{C}_\mathrm{m}(q)$       // there is exactly one mixed child
11             end
12             case Ⓘ do
13                 foreach $p \in \mathcal{C}(q) \setminus \mathcal{C}_\mathrm{m}(q)$ do
14                     if $p$ is LINKED then return $q$
15                 end
16                 $\{q\} \leftarrow \mathcal{C}_\mathrm{m}(q)$
17             end
18             case Ⓟ do
19                 if $x$ has no twin in $G_q$ then // consider the mixed child
                    to be not linked
20                     return $q$
21                 end
22                 $\{q\} \leftarrow \mathcal{C}_\mathrm{m}(q)$   // the mixed child must be identical to
                    the twin
23             end
24         end
25     end
26 return $V(G)$ is uniform with respect to $x$
```

this vertex to all vertices in $V(G)$ which labels are greater than $\lambda(x) = 1$ because all other vertices have smaller ranks. These are all other nodes, and therefore the above statement holds. ■

Lemma 6. *Inserting a vertex x with neighbourhood $N(x) = \emptyset$ in permutation graph $G_\pi = ((V, E), \lambda)$ with permutation π, results in $\pi^{+x(n+1,n+1)}$ for the graph $G + x$, i.e. x has rank $\pi^{-1}_{+x}(x) = n + 1 = |V| + 1$ in π_{+x} and label $\lambda(x) = n + 1$.*

Proof. The element in the last position (at $|V| + 1$) will be connected to all vertices that have larger labels then itself. But since x has the largest possible label, it will have no edges and its neighbourhood is empty. ■

This shows that the ranks and labels for vertices in $G[P]$ remain unchanged, if P is uniform with respect to x. And thus, these nodes can be disregarded as insertion nodes.

Another trivial case is, if x has a twin in G_p as defined in Definition 2.3 and shown in Corollary 2, since x and its twin constitute a module in $G_p + x$ and x is inserted into π next to its twin with a larger or smaller label, depending on whether they are linked or not (see Proposition 3.2).

To reliably identify the insertion node q in a top-down search, Crespelle and Paul [17] give the following definition for a property possessed by all nodes $p \succ_{T_G} q$ that covers the trivial cases mentioned above.

Definition 3.1 (Proper, [17, Definition 2])**.** Let $x \notin V$ a vertex to be inserted into a graph $G = (V, E)$. A node p of a modular decomposition tree T_G is **proper** if and only if p has one of the following properties:

(i) p is uniform with respect to x, i.e. either `LINKED` or `NOTLINKED`, or

(ii) p is a mixed node with at most one mixed child $f \in V(T_G)$ such that $F \cup \{x\}$ is a module in $G[P] + x$ ($F = V(f)$, $P = V(p)$).

Otherwise p is **non-proper**.

Proposition 3.5 (Crespelle and Paul [17])**.** *Let $G' = G + x$ and let q be the least common ancestor of non-proper nodes of T_G. $T_{G'}$ is obtained from T_G by replacing the subtree T_q of T_G rooted at q with $T_{G'}[Q'] = T_{G[Q]+x}$.*

Definition 3.1 and Proposition 3.5 allow the algorithmic approach sketched in Algorithm 4. In a top-down approach we follow the path of mixed proper nodes with at most one mixed child. The algorithm stops, when the current node q is non-proper, which is the insertion node by Definition 3.1. The check whether or not x has a twin in a regarded prime node

3.3.3 Substitution Process

After the insertion node q is found, the problem reduces to finding a valid permutation or, by extension, valid insertion position for $G[Q] + x$. Substitution of elements in π may be visualised as the inverse process of slicing depicted in the last chapter in Figure 3. All we do, is replacing a single line in a line graph (some element y in π) by another line graph (of another permutation τ) and obtain an augmented version, without interfering with any intersections of lines contained in $\pi - x$ or τ. The following Definition 3.2 and Theorem 3 will be used to produce a valid permutation for $G + x$ by the few steps depicted in Algorithm 5.

Algorithm 5: Brief algorithmic scheme of the idea behind the insertion process.

```
   Input: Root Node $r$ of a linked modular decomposition tree $T_G$, the
          insertion node $q$ and the neighbourhood $N(x)$ of $x$ in $V(G)$.
   Output: Permutation $\pi_{+x}$.
1  Retrieve $\pi[Q]$ and $\tau := \pi[V(G) \setminus Q]$ from $T_G$.
2  Insert $x$ into $\tau$ at the former rank of the first element of $\pi[Q]$ with label
     $\lambda_{\text{min}}(q)$, obtaining $\tau_{+x}$.
3  Insert $x$ into $\pi[Q]$, obtaining $\pi[Q]_{+x}$.
4  Substitute $x$ in $\tau_{+x}$ with $\pi[Q]_{+x}$, obtaining $\pi_{+x} = \tau_{+x}^{x \to \pi[Q]_{+x}}$.
5  return $\pi_{+x}$.
```

Definition 3.2 (Permutation Substitution)**.** Given a permutation π of a permutation graph $G_\pi = ((V(G), E(G)), \lambda)$ and another permutation τ of a permutation graph $H_\tau = ((V(H), E(H)), \lambda_\tau)$, substituting some vertex $x \in V(G)$ with H, that is connecting all vertices $u \in V(H)$ to the neighbours $v \in N(x)$ and deleting x together with all its edges results in permutation graph $G^{x \to H} = (V(H) \cup V(G) \setminus \{x\}, V(H)N(x) \cup E_H \setminus xN(x)$ with **substituted permutation** $\pi^{x \to H}$. To obtain $\pi^{x \to H}$, the element representing x at rank $\pi^{-1}(x)$ in π is replaced by permutation τ where each label in $\tau = (\tau(1)\, \tau(2)\, \dots\, \tau(m))$ is incremented by $\lambda(x) - 1$ and each label in π that is larger than $\lambda(x)$ is incremented by $m - 1 = |V(H)| - 1$. Subsequently all elements to the right of the element representing x in π $\pi(v) > \pi(x)$ are shifted by $m - 1$ to the right.

Theorem 3. *Let $x \in V$ be a vertex of a permutation graph $G_\pi = (G, \lambda)$. Substituting x with a permutation graph H_τ as described in Definition 3.2 resulting in $G_\pi^{x \to H_\tau}$ with permutation $\pi^{x \to \tau}$ can be done in such a way that for any vertex $v \in V(G)$ it holds that $\lambda^{x \to \tau}(v) \geq \lambda(v)$.*

Proof. Let π_{subs} and λ_{subs} denote the permutation $\pi^{x \to \tau}$ and labelling $\lambda^{x \to \tau}$ of the graph $G^{x \to H}$, where some vertex $x \in V(G)$ is substituted by a permutation graph H_τ. Let G_π, H_τ be two permutation graphs matching Definition 3.2. Let $|V(G)| = n$ and $|V(H)| = m$. Without loss of generality let $m \geq 2$ and $n \geq 1$. We show that under the substitution process the sign of each vertex pair's connectivity $c(u, v) = D_\lambda(u, v) D_\pi(v, u)$, i.e., edges in H and $G - x$, does not change and that all neighbours of x are connected to each vertex in H in $G^{x \to H}$.

Let $u, v \in V(G) \setminus \{x\}$ be two distinct vertices and let $\lambda(u) = j$ and $\lambda(v) = i$ be the labels of these vertices before the substitution. Without loss of generality, assume that $i < j$ (for the other case, just switch the operands of D). If it holds for both labels that $i, j > \lambda(x)$ then their new labels become $\lambda_{\text{subs}}(u) = j + m - 1$ and $\lambda_{\text{subs}}(v) = i + m - 1$ and we have $D_{\lambda_{\text{subs}}}(u, v) = (j + m - 1) - (i + m - 1) = j - i = D_\lambda(u, v)$. So in this

case $D_{\lambda_{\text{subs}}} = D_\lambda$.
On the other hand, if $i, j < \lambda(x)$ it can be trivially concluded that $D_\lambda(u,v) = D_{\lambda_{\text{subs}}}(u,v)$ since both labels remain unchanged.
In the last case we have $i < \lambda(x) < j$, and thus $D_{\lambda_{\text{subs}}}(u,v) = j + m - 1 - i = D_\lambda(u,v) + (m-1)$. But since we assumed that $m \geq 2$ we can conclude that $D_\lambda > 0$. Furthermore, because a positive integer $m - 1 > 0$ is added it follows that also $D_{\lambda_{\text{subs}}} > 0$.
Conclusively, the above reasoning shows that under substitution $D_{\lambda_{\text{subs}}}(u,v) \geq D_\lambda(u,v)$ for all $u, v \in V(G)$.
The same reasoning can be applied for $D_{\pi_{\text{subs}}}$. If $\pi^{-1}(i)$ and $\pi^{-1}(j)$ are both smaller or greater than $\pi^{-1}(x)$ it holds that $D_\pi(u,v) = D_{\pi_{\text{subs}}}(u,v)$ or in the other case if $\pi^{-1}(i) < \pi^{-1}(x) < \pi^{-1}(j)$, then $D_{\pi_{\text{subs}}}(u,v) > D_\pi(u,v)$ (the inequality is flipped if the operands are swapped, i.e., $D_{\pi_{\text{subs}}}(v,u) < D_\pi(v,u)$).
These two results can be combined to form the expression for these two vertices' connectivity $c_{\text{subs}}(u,v) = D_{\lambda_{\text{subs}}}(u,v) D_{\pi_{\text{subs}}}(v,u)$. In case u and v are connected, we have $c(u,v) > 0$ and $D_\pi(v,u)$ and $D_\lambda(u,v)$ have the same sign. It therefore holds under our assumption $i < j$ that both are positive and necessarily $\pi^{-1}(j) < \pi^{-1}(i)$.
Conclusively, the reasoning above yields that $D_\lambda(u,v) \leq D_{\lambda_{\text{subs}}}(u,v)$ and $D_\pi(v,u) \leq D_{\pi_{\text{subs}}}(v,u)$, which results in $c_{\text{subs}} \geq c(u,v)$ and that $uv \in E(G^{x \to H})$

Now consider two vertices $u, v \in V(H)$ from the graph that is substituted for x. Again, let $\lambda_\tau(u) = i$ and $\lambda_\tau(v) = j$. Since u and v are both vertices of H_τ their label is incremented by the same amount $\lambda(x) - 1$. This results in the same reasoning as the first case above, so $D_{\lambda_\tau}(u,v) = j - i = j + (\lambda(x) - 1) - (i + (\lambda(x) - 1)) = D_{\lambda_{\text{subs}}}(u,v)$. Note, that also $D_{\pi_{\text{subs}}}(u,v) = D_\tau(u,v)$, since their ranks are incremented by the same amount. So we can conclude this case with $c(u,v) = c_{\text{subs}}(u,v)$

For the last combination of vertices, let $u \in V(H)$ and $v \in V(G)$. Therefore, the relevant differences are $D_\lambda(x,v)$ and $D_{\lambda_{\text{subs}}}(u,v)$. Let $\lambda(v) = j$ and $\lambda_\tau(u) = i$. Now, assume that $vx \in E(G)$. Also we assume that $j < \lambda(x)$ and cover the inverse case later. Under these conditions, it then holds that $\lambda_{\text{subs}}(v) = \lambda(v)$ and that $D_\pi(x,v)$ and $D_\lambda(v,x)$ must have the same sign, since $c(v,x) > 0$. Therefore it holds for both $D_\pi(x,v)$, $D_\lambda(v,x) < 0$.
For $D_{\lambda_{\text{subs}}}(v,u) = \lambda_{\text{subs}}(v) - \lambda_{\text{subs}}(u) = \lambda(v) - \lambda_\tau(u) - \lambda(x) + 1 = D(v,x) - (\lambda_\tau(u) - 1)$ we can deduce that $D_{\lambda_{\text{subs}}}(v,u) < 0$ because $\lambda_\tau(u) - 1$ is a positive integer. In case of $D_\pi(x,v) < 0$ it must hold that $\pi^{-1}(x) < \pi^{-1}(v)$ (the element representing v in π is to the right of x) and thus ${\pi_{\text{subs}}}^{-1}(v) = \pi^{-1}(v) + |V(H)| - 1$. Therefore, the following equalities hold: $D_{\pi_{\text{subs}}}(u,v) = {\pi_{\text{subs}}}^{-1}(u) - {\pi_{\text{subs}}}^{-1}(v) = \pi^{-1}(x) - 1 + \tau(u) - \pi^{-1}(v) - |V(H)| + 1 = D_\pi(x,v) + \tau(u) - |V(H)|$. And since, by definition, $\tau(u) - |V(H)| \leq 0$ we can deduce that $D_{\pi_{\text{subs}}}(u,v) < 0$.
Now let $j > \lambda(x)$. In that case $D_\lambda(v,x)$ and $D_\pi(v,x)$ are both positive and it holds that $D_{\lambda_{\text{subs}}}(v,u) = D_\lambda(v,x)$. Furthermore, since $D_\pi(x,v) > 0$ we have $\pi^{-1}(v) < \pi^{-1}(x)$ and therefore $D_{\pi_{\text{subs}}}(u,v) = D_\pi(x,v)$. Thus, we can conclusively state that $c_{\text{subs}}(u,v) = c(v,x)$ in this case.

For the last case, consider $vx \notin E(G)$. Since $c(v,x) < 0$, $D_\pi(x,v)$ and $D_\lambda(v,x)$ must have opposite signs. For the first sub-case we assume $D_\pi(x,v) < 0$ and $D_\lambda(v,x) > 0$. Therefore, $\pi^{-1}(x) < \pi^{-1}(v)$ and thus $D_{\pi_{\text{subs}}}(u,v) = \pi^{-1}(x) - 1 + \tau(u) - \pi^{-1}(v) - |V(H)| + 1 = D_\pi(x,v) + \tau(u) - |V(H)| < 0$ as shown above.

The label of v is also incremented and the expression for $D_{\lambda_{\text{subs}}}(v,u)$ becomes $D_{\lambda_{\text{subs}}}(v,u) = \lambda_{\text{subs}}(v) - \lambda_{\text{subs}}(u) = \lambda(v) + |V(H)| - 1 - \lambda_\tau(u) - \lambda(x) + 1 = D_\lambda(v,x) + |V(H)| - \lambda_\tau(u)$. Since $|V(H)| \geq \lambda_\tau(u)$ it holds that $D_{\lambda_{\text{subs}}}(v,u) > 0$ and therefore $c_{\text{subs}}(v,u) = D_{\lambda_{\text{subs}}}(v,u) D_{\pi_{\text{subs}}}(u,v) < 0$. Similar arguments hold for the other sub-case where $D_\lambda(v,u) < 0$ and $D_\pi(u,v) > 0$.

With the results above we have now shown that with the given substitution procedure the connectivity between all vertices is retained and that the neighbourhood $N(u)$ of any vertex $u \in V(H)$ is extended by only $N(x)$. Furthermore, we can deduce that $\lambda_{\text{subs}}(w) > \lambda(w)$, $w \in V(G^{x \to H})$, since the procedure only increments given labels or ranks of any involved vertices. ■

Algorithm 6: In place substitution of an element x in permutation π with another permutation τ.

```
Input: Two permutations $\pi$ and $\tau$ and a vertex $x \in V(G_\pi)$.
Output: Permutation $\pi^{x\to\tau}$ where $x$ in $\pi$ is replaced by $\tau$ and its mapping
        $\lambda^{x\to\tau}$.
1  foreach element $e$ in $\pi$ do
2      $l \leftarrow \lambda_\pi(e)$
3      if $l > \lambda_\pi(x)$ then
4          $\lambda^{x\to\tau}(e) \leftarrow l + |\tau| - 1$
5      else
6          $\lambda^{x\to\tau}(e) \leftarrow l$
7      end
8      if $\pi^{-1}(e) > \pi^{-1}(x)$ then
9          $[\pi^{-1}]^{x\to\tau}(e) \leftarrow \pi^{-1}(e) + 1$
10     else
11         $[\pi^{-1}]^{x\to\tau}(e) \leftarrow \pi^{-1}(e)$
12     end
13 end
14 foreach element $f$ in $\tau$ do
15     $\lambda^{x\to\tau}(f) \leftarrow \lambda_\tau(f) + \lambda_\pi(x) - 1$
16     $[\pi^{-1}]^{x\to\tau}(f) \leftarrow \tau^{-1}(f) + \pi^{-1}(x) - 1$
17 end
18 return $\pi^{x\to\tau}$, $\lambda^{x\to\tau}$
```

In Algorithm 5 the retrieving of the two permutations in line 1 for example, can be done by using Procedure getPi on the tree rooted at q for $\pi[Q]$ and by leaving out the strong interval $\pi[Q]$ in π. Since Q is a module of G, it is also a strong interval in π and therefore consists of consecutive elements. After removing $\pi[Q]$ from π we can iterate once over the resulting permutation and decrement the labels greater than $\lambda_{\max}(q)$ by $|\pi[Q]|$ for each element in τ and decrement every elements' label in $\pi[Q]$ by $\lambda_{\min}(q)$.

The substitution in line 4 is the inverse process: In the former case just insert x at rank $\pi_{\min}(q)$ with label $\lambda_{\min}(q)$ into τ and then proceed substituting that with $\pi[Q] + x$ by incrementing all labels in $\pi[Q] + x$ by $\lambda_{\min}(q)$ and all element's labels greater than $\lambda_{\max}(q) + 1$ in τ by $|\pi[Q]| + 1$ as described in Definition 3.2 and Algorithm 6. Note, that in either case $\pi_{\min}(q)$ and $\lambda_{\min}(q)$, etc. are saved for each node $p \in T_G$ in the data structure.

The last step that remains is to augment the permutation $\pi[Q]$ with x by inserting it. This will be the topic of the last section in this chapter.

3.3.4 Rearranging the Tree T_q

Recall, that the insertion node q represents the last common ancestor of non-proper modules and is itself non-proper. It therefore holds that q has one of the following properties:

Proposition 3.6 ([17, Definition 4])**.** *Let x be the vertex to be inserted in a permutation graph G_π. The insertion node q in T_G must have one of the following properties:*

(i) q is a `MIXED` degenerate node (either $\textcircled{S}$ or $\textcircled{\parallel}$) with no mixed children,

(ii) q is a prime node with no mixed child, but a node t that is a twin of x in G_q,

(iii) q is a degenerate node with at least one `MIXED` child, i.e., $\mathcal{C}_{\mathrm{m}}(q) > 0$, or

(iv) q is a prime node and x has no twin in G_q.

Trivial Cases

Let us now focus on each of these properties one after another. As it turns out the first *(i)* and the second property *(ii)* have almost trivial solutions for inserting x, since their permutations require no rearranging of any tree. Crespelle and Paul [17] in their work collectively call these properties that the insertion node q is *cut*.

Proposition 3.7 ([17, Theorem 3])**.** *Let $G_\pi = (G, \lambda)$ be a permutation graph and $x \notin V(G)$ a vertex to be inserted into G. Then $G + x$ is a permutation graph if and only if either the insertion node $q \in V(T_G)$ of the modular decomposition tree is mixed, but has no mixed children or q is a prime node with no mixed child but one child is a twin of x, i.e., there exists some node $p \in \mathcal{C}(q)$ such that for all $v \in P$ $N(x) = N(v) \setminus P$. Otherwise $G + x$ is a permutation graph if and only if Proposition 3.8 and Proposition 3.9 apply.*

But first, consider this following lemma.

Lemma 7. *Let $G_\pi = ((V, E), \lambda)$ be a permutation graph of permutation π. It holds for the vertex $\pi(n) = u \in V$ at rank $n = |V|$ in π that $\lambda(u) = |\overline{N}(u)| + 1 = n - |N(u)|$ and for the vertex $\pi(1) = v \in V$ at rank 1 that $\lambda(v) = n - |\overline{N}(v)| = |N(u)| + 1$.*

Proof. Consider the first element of π, namely $\pi(1) = \lambda(u)$, $\lambda^{-1}(\lambda(u)) = u \in V$. For every vertex y in the neighbourhood of u, $y \in N(u)$, it must hold, that $D_\lambda(u, y)D_\pi(y, u) > 0$. Since $\pi^{-1}(u) = 1$ and $\pi^{-1}(x) > 1$ for all other vertices $V \ni y \neq u$, it holds that $D_\pi(y, u) < 0$. Therefore, $D_\lambda(u, y)$ must be negative as well, because by assumption $uy \in E$, and thus $\lambda(u) - \lambda(y) < 0 \implies \lambda(u) < \lambda(y)$ for all $y \in N(u)$. Since the mapping λ is bijective, we can deduce that these vertices have indeed the labels $[1 : |N(u)|]$. This also means that $|N(u)|$ is a lower bound for $\lambda(u)$.

A similar argument holds for $n - |\overline{N}(u)|$ being an upper bound of $\lambda(u)$. Therefore, $\lambda(u)$ must have a value between $n - |\overline{N}(u)|$ and $|N(u)|$ and since $|N(u)| + |\overline{N}(u)| + 1 = n$ we can deduce that $\lambda(u) = |N(u)| + 1$ if $\pi^{-1}(u) = 1$. The proof for the other case when $\pi^{-1}(v) = n$ is similar. ■

This means for the former case *(i)* that x can be inserted with label $\lambda(x) = |\overline{N}(x)[Q]| + 1$ at position $\pi^{-1}(x) = n + 1 = |Q| + 1$ and we obtain a valid permutation $\pi[Q]^{+x(|\overline{N}(x)[Q]|+1,|Q|+1)}$ for $G[Q] + x$. The insertion in the latter case *(ii)* can be resolved according to Corollary 2 and the subsequent remark. Depending, whether the twin is connected $t \in N(x)[Q]$, or not, we insert x next to it with a label one smaller or one larger than $\lambda(t)$ and obtain $\pi[Q]^{+x(\pi^{-1}(t)\pm 1,\lambda(t)\pm 1)}$. Any of these permutations can then be re-substituted into the remaining part of the original permutation π according to Algorithm 5 and Algorithm 6.

Remark. Lemma 7 directly implies that a vertex v with label $\lambda(v) = n$ must have rank $\pi^{-1}(v) = n - |N(v)|$, since Lemma 7 must also hold for π^{h} where $\lambda^{\mathrm{h}} = n - |N(v)|$. The same applies to $\lambda(u) = 1$ and $\pi^{-1}(u) = |N(u)| + 1$ respectively.

Constructing π_{+x} with the help of T_q

We shift our focus now on cases *(iii)* and *(iv)*. They require a more careful approach and rearranging of the data structure (namely rearrangement of the doubly linked lists of the children of each mixed node $p \preceq_{T_G} q$) with the aim to get every vertex (leaf) at its final rank in $\pi[Q] + x$ with the correct label and a set insertion position for x. After that rearrangement the tree might not be a modular decomposition tree any more , but we use it in a final step to just retrieve $\pi[Q] + x$ and can thus be discarded afterwards.

To find the correct arrangement, we use, yet again, propositions shown by Crespelle and Paul [17]. They give sufficient and necessary conditions for $G + x$ being a permutation graph, depending on the properties given in Proposition 3.6 for the insertion node q.

Notation ([17]). Given a graph $G = (V, E)$, the set of its maximal strong modules $\mathbb{M}_{\max}(V)$, a node p of its modular decomposition tree T_G representing a vertex set $P \subseteq V$ and a vertex $x \notin V$. We denote as H_p the following graph $H_p = (V[P], E[P \setminus \bigcup_{c \in \mathcal{C}_{\mathrm{m}}(p)} V(c)\})/(\mathbb{M}_{\max}(P) \cup \{\{x\}\})$ that is the quotient graph $G_p + x$ of $G[P] + x$, where the all edges leading from mixed children of p to x are removed and $\{x\}$ is part of the partition of $P \cup \{x\}$.

Proposition 3.8 ([17, Theorem 3.1])**.** *If Proposition 3.7 does not hold, then $G + x$ is a permutation graph if and only if the insertion node q has at most two mixed children f_1 and f_2 and H_q is a permutation graph such that in any of its admissible permutations π_{H_q} $\lambda_{H_q}(x) = \lambda_{H_q}(f_1) \pm 1$ and ${\pi_{H_q}}^{-1}(x) = {\pi_{H_q}}^{-1}(f_2) \pm 1$ if f_1 and f_2 exists respectively. Or, if q has no mixed child, that x can be inserted into $H_q \equiv G_q$. Moreover, for any node $p \prec_{T_G} q$ Proposition 3.9 has to hold.*

Proposition 3.9 ([17, Theorem 3.2])**.** *Any node $p \prec_{T_G} q$ satisfies the following conditions: p has at most one mixed child f and H_p is a permutation graph with* *permutation π_{H_p} such that $\lambda_{H_p}(x) = \lambda_{H_p}(f) \pm 1$ if f exists and $\pi_{H_p}(|V(H_p)|) = x$, i.e., x is the last element in π_{H_p}.*

Due to space restrictions, and because I deemed it too similar, this thesis will not include the proof for the correctness of this part. Instead the reader may refer to the proof given by Crespelle and Paul [17] on what they call Theorem 3 contains detailed descriptions how to obtain $\pi[Q]_{+x}$ in an inductive manner. Our algorithm differs for example, because it produces permutations π_{p+x} for each $p \prec_{T_q} q$ such that x is the last element, instead of the first. Other minor details are different as well, mainly because of restrictions given by our overall approach and data structure. As a consequence, I will give a detailed overview of the algorithmic approach and hint the basic notions on the construction of permutations for H_p and with them $\pi[Q]_{+x}$.

Algorithm 7a: Algorithm to find the permutation $\pi[Q]_{+x}$, part 1: where descendants of q in T_q are handled according to Proposition 3.9. q has to suffice properties *(iii)* or *(iv)* of Proposition 3.6.

Input: Insertion node q of a linked modular decomposition tree T_G with respect to the inserted vertex x.

Output: Permutation $\pi[Q]_{+x}$ or the result that $G + x$ is not a permutation graph.

```
s ← post_order($q$)
$p$ ← $s$.pop()
while $p \neq q$ do            // $q$ is the last element of a postorder stack
    while $p$ is LEAF do $p$ ← $s$.pop()
    if $\mathcal{C}_\mathrm{m}(p) > 1$ then return $G + x$ is not a permutation graph.
    if $p$ is $\textcircled{P}$ then
        $N_p(x) \leftarrow \mathcal{C}_\mathrm{l}(p)$
        Compute $\pi_{H_p}$ with $N_p(x)$ and $\pi_p$ as input for Algorithm 1.
        reorderPrime($p$, $\pi_{H_p}$)
        if $\mathcal{C}_\mathrm{m}(p) = 1$ then
            $\{f\} \leftarrow \mathcal{C}_\mathrm{m}(p)$
            $\Lambda_x \leftarrow ||\mathcal{C}(p)| - |N_p(x)||$
            if $\lambda_\mathrm{parent}(f) - \Lambda_x \neq 0$ and $\lambda_\mathrm{parent}(f) + 1 - \Lambda_x \neq 0$ then // i.e.,
              $\lambda_{H_p}(f) \neq \lambda_{H_p}(x) \pm 1$
                return $G + x$ is not a permutation graph.
            end
        end
    else                                               // $p$ is $\textcircled{S}$ or $\textcircled{\parallel}$
        reorderDegenerate($p$)
    end
    $p$ ← $s$.pop()
end
```

The main reordering procedures are given in Algorithm 7a, 7b and 7c, from now on collectively referred to as Algorithm 7. Let me now try to dissect it and give some context for the most important steps.

Algorithm 7b: Algorithm to find the permutation $\pi[Q]_{+x}$, part 2: to handle cases where $|\mathcal{C}_\mathrm{m}(q)| \leq 1$ according to Proposition 3.8. q has to suffice properties *(iii)* or *(iv)* of Proposition 3.6.

```
if $\mathcal{C}_\mathrm{m}(q) = 0$ then                    // by Proposition 3.6 (iii) $q$ is $\textcircled{P}$
    $N_q(x) \leftarrow \mathcal{C}_\mathrm{l}(q)$
    Compute $\pi_{H_q}$ with $N_q(x)$ and $\pi_q$ as input for Algorithm 1.
     // computation fails if $H_q$ is not a permutation graph.
    Attach $x$ as LEAF to $q$ according to $\pi_{H_q}$.
    return getPi($q$)
else if $\mathcal{C}_\mathrm{m}(q) = 1$ then
    $\{f\} \leftarrow \mathcal{C}_\mathrm{m}(q)$
    if $q$ is $\textcircled{S}$ or $\textcircled{\parallel}$ then
        reorderDegenerate($q$)
    else
        $N_q(x) \leftarrow \mathcal{C}_\mathrm{l}(q)$
        Compute $\pi_{H_q}$ with $N_q(x)$ and $\pi_q$ as input for Algorithm 1.
        if $\lambda_{H_q}(f) \neq \lambda_{H_q}(x) \pm 1$ then
            if $\pi^{-1}_{H_q}(f) = \pi^{-1}_{H_q}(x) \pm 1$ then
                mirror($q$, h)
            else
                return $G + x$ is not a permutation graph.
            end
        end
    end
    $\pi[Q] \leftarrow$ getPi($q$)
    return $\pi[Q]^{+x(|Q|-|N(x)[Q]|,|Q|+1)}$             // insert $x$ with label
     $|Q| - |N(x)[Q]|$ at the last position $|Q| + 1$.
```

In a first step we apply Proposition 3.9 to all descendants of insertion node $p \prec_{T_q} q$. For degenerate nodes this process is pretty much straight forward, since the quotient permutation of these nodes allows any possible label for any possible child. Because we arbitrarily choose x to have the largest rank in H_p, Lemma 7 restraints us in Procedure reorderDegenerate to sort every $\textcircled{\parallel}$ node such that each NOTLINKED child is labelled $[1 : |\mathcal{C}_\mathrm{nl}(p)|]$ and the LINKED ones as $[|\mathcal{C}_\mathrm{l}(p)| : |\mathcal{C}(p)|]$ with identical ranks respectively. In case p is $\textcircled{S}$ we chose the same labelling, but inverse ranks, i.e., $\pi_p^{-1}(p_i) = |\mathcal{C}(p)| - \lambda_p(p_i)$. See Figure 10a and Figure 10b for two example of rearranged degenerate nodes. If a mixed child exists it is inserted in between these two runs with label $|\mathcal{C}_\mathrm{l}(p)| + 1 = |\mathcal{C}_\mathrm{nl}(p)|$, thus allowing to have consecutive labelling with x in an admissible permutation $\pi_q^{+x(|\mathcal{C}_\mathrm{nl}(p)|+1,|\mathcal{C}(p)|+1)} = \pi_{H_q}$ for the graph H_q as was shown in Lemma 7.

In case p is $\textcircled{P}$, the solution requires some more work. As was discussed in Proposition 2.1, a prime node is uniquely represented by its permutation π, except for reversals by the mirroring-operations π^h, π^v and π^m. We can transform each node's

Algorithm 7c: Algorithm to find the permutation $\pi[Q]_{+x}$, part 3: to handle cases where $|\mathcal{C}_{\mathrm{m}}(q)| \geq 2$ according to Proposition 3.8. q has to suffice properties *(iii)* or *(iv)* of Proposition 3.6.

```
else if $\mathcal{C}_{\mathrm{m}}(q) = 2$ then
    if $q$ is $\textcircled{S}$ or $\textcircled{\parallel}$ then
        reorderDegenerate($q$) with the additional constraint for $f_2$ to be
          the last element of $\pi_q$
    else                                                   // i.e., $q$ is $\textcircled{P}$
        $N_q(x) \leftarrow \mathcal{C}_{\mathrm{l}}(q)$
        Compute $\pi_{H_q}$ with $N_q(x)$ and $\pi_q$ as input for Algorithm 1.
        $\{f_1, f_2\} \leftarrow \mathcal{C}_{\mathrm{m}}(q)$ such that $\pi^{-1}_{H_q}(f_1) < \pi^{-1}_{H_q}(f_2)$
        if $\lambda_{H_q}(f_1) \neq \lambda_{H_q}(x) \pm 1$ then
            if $\pi^{-1}_{H_q}(f_1) = \pi^{-1}_{H_q}(x) \pm 1$ and $\lambda_{H_q}(f_2) \neq \lambda_{H_q}(x) \pm 1$ then
                mirror($q$, h)
                if $\pi_q^{-1}(f_1) > \pi_q^{-1}(f_2)$ then mirror($q$, v)
            else
                return $G + x$ is not a permutation graph.
            end
        else if $\pi^{-1}_{H_q}(f_2) \neq \pi^{-1}_{H_q}(x) \pm 1$ then
            return $G + x$ is not a permutation graph.
        end
    end
    $s \leftarrow$ post_order($f_2$)
    while $s$ is not empty do
        $p \leftarrow s$.pop()
        if $\lambda_q(f_2) < \lambda_q(f_1)$ then                      // $f_1 f_2 \in E(q)$
            mirror($p$, h)  // reorders node such that $\lambda_{H_p}(x) = |V(H_p)|$
        else
            mirror($p$, h)
            mirror($p$, v)         // reorders node such that $\lambda_{H_p}(x) = 1$
        end
    end
    $\Lambda_x \leftarrow \lambda_{\min}(f_1) + |F_1| - |N(x)[F_1]|$
    if $\lambda_q(f_2) < \lambda_q(f_1)$ then
        $\varrho_x \leftarrow \pi_{\max}(f_2) + 1 - |N(x)[F_2]|$       // insertion label will be
          maximal in $f_2$
    else
        $\varrho_x \leftarrow \pi_{\min}(f_2) + |N(x)[F_2]|$ // insertion label will be minimal
          in $f_2$
    end
    $\pi[Q] \leftarrow$ getPi($q$)
    return $\pi[Q]^{+x(\Lambda_x, \varrho_x)}$
else                                                          // $\mathcal{C}_{\mathrm{m}}(q) > 2$
    return $G + x$ is not a permutation graph.
end
```

quotient, and therefore the order of children in the `Node` object by applying Procedure reorderPrime, that performs the respective mirroring with Procedure mirror to the children of p if applicable. In a last step for this part (line 27) we have to check whether the labellings of x and a mixed child, if present, are consecutive.

Since all descendants of q are now arranged in a manner that they admit for a permutation of H_p where x is the last element, it is obvious by induction and Lemma 7 that a permutation also exists for H_q if it has no mixed child. If q has a single mixed child f we have to apply a similar process like we did for its descendants. In either case Algorithm 7b then returns a admissible permutation for $\pi[Q]_{+x}$ by inserting x into the last position with a label in range of $\lambda[\mathcal{L}(f)]$. This is possible by the same inductive reasoning, because all mixed descendants of f have consecutive labels to x in their strong module. Hence, in $\pi[Q]_{+x}$ the leaves of module f are split by x into adjacent and non-adjacent and constitute for two new modules in $G[Q] + x$.

The last part Algorithm 7c handles the most complex case, where q has two mixed children $\mathcal{C}_{\mathrm{m}}(q) = \{f_1, f_2\}$. In Figure 9 the branching statements of lines 51 sqq. are depicted in another manner. These allow us to find the right configuration for q so we can apply the correct mirroring to the subtree T_{f_2} and thus deduce a correct insertion position for x. Recall, that elements in $\pi^{\mathrm{h}}[F_2]$ have transposed ranks and labels to $\pi[F_2]$, hence we can assume that x is inserted into the interval $\pi^{\mathrm{h}}[F_2]$ with a label smaller or larger than all elements in F_2, depending on the adjacency of f_1 and f_2 in G_q. Therefore, the rearranging of T_{f_2} aims at making the strong interval applicable for the remark following Lemma 7.

Because x is inserted with a label of range $\lambda[\mathcal{L}(f_1)]$ into the interval $\pi[\mathcal{L}(f_2)]$ it creates four new modules in π_{+x} as it splits the mixed children into two parts respectively. Figure 10 shows an example, where two mixed degenerate nodes are joined by an edge in π_q and were rearranged to produce a fitting permutation for the insertion of x into $\pi[Q]$. With the same reasoning, it is intuitively obvious given the two degrees of freedom by x's rank and label in π_{+x} that two is maximum number of mixed children q can have when inserting x.

Let me conclude this chapter with a short discussion regarding the time complexity of Algorithm 7. Each sequential step can be done in at most $\mathcal{O}(n)$ time, since, again, we only use quotient permutations except when returning the full permutation of $\pi[Q]_{+x}$. But inserting x with a known label and position into a given permutation can be done in $\mathcal{O}(n)$ time, because the larger labels can be incremented by a one-time iteration over all $\mathcal{O}(n)$ elements in $\pi[Q]_{+x}$.

Corollary 5. *Algorithm 7 returns $\pi[Q]_{+x}$ or the result that $G+x$ is not a permutation graph in $\mathcal{O}(n)$ time.*

Proof. See discussion above. ■

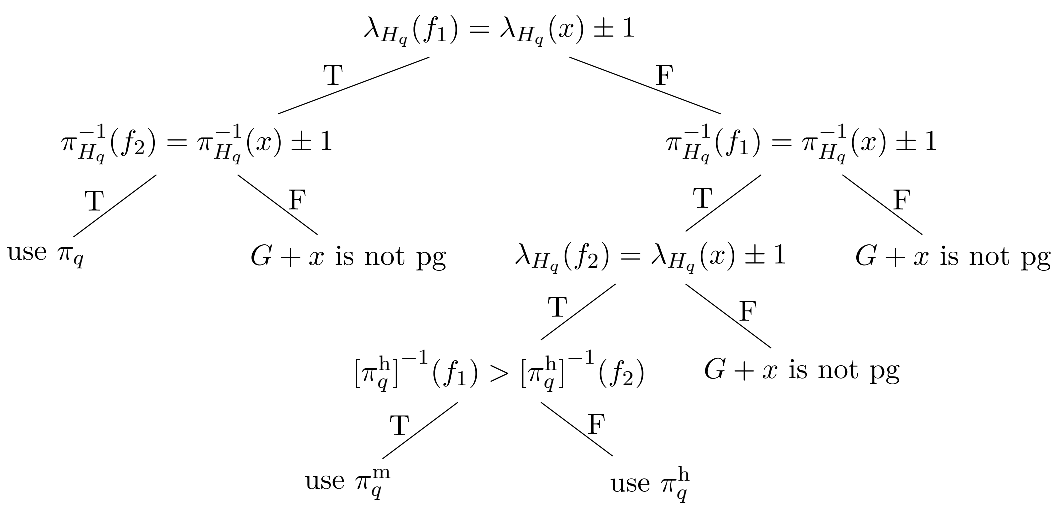
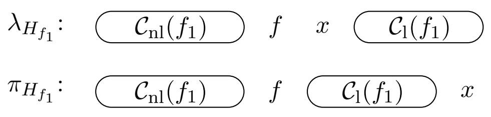
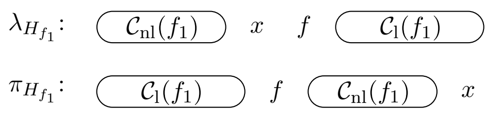

$\lambda_{H_q}(f_1) = \lambda_{H_q}(x) \pm 1$
T
F
$\pi^{-1}_{H_q}(f_2) = \pi^{-1}_{H_q}(x) \pm 1$
$\pi^{-1}_{H_q}(f_1) = \pi^{-1}_{H_q}(x) \pm 1$
use π_q
$G + x$ is not pg
$\lambda_{H_q}(f_2) = \lambda_{H_q}(x) \pm 1$
$G + x$ is not pg
$[\pi^{\mathrm{h}}_q]^{-1}(f_1) > [\pi^{\mathrm{h}}_q]^{-1}(f_2)$
$G + x$ is not pg
use π^{m}_q
use π^{h}_q

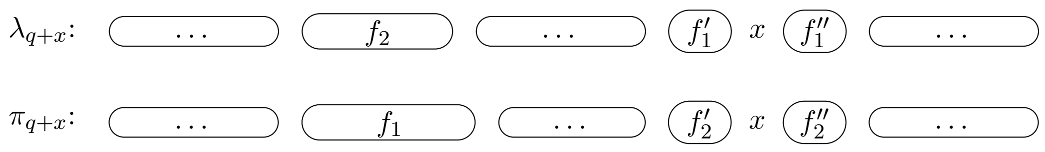
Figure 9: Decision tree, in case $\mathcal{C}_{\mathrm{m}}(q) = \{f_1, f_2\}$ with $\pi^{-1}(f_1) < \pi^{-1}(f_2)$, to find correct configuration for π_q with mirroring operations, given H_q. Use T path if the assertion above is true, F otherwise. See also Algorithm 7c lines 51 sqq.


$\lambda_{H_{f_1}}$:
$\mathcal{C}_{\mathrm{nl}}(f_1)$
f
x
$\mathcal{C}_{\mathrm{l}}(f_1)$
$\pi_{H_{f_1}}$:
$\mathcal{C}_{\mathrm{nl}}(f_1)$
f
$\mathcal{C}_{\mathrm{l}}(f_1)$
x

(a) Schematic line graph of H_{f_1}. f_1 is a Ⓛ mixed child of q. The connecting lines between the two linear orders are omitted. f is the only mixed child of f_1. The configuration between x and a mixed child f is shared across all nodes $p_1 \preceq f_1$.


$\lambda_{H_{f_1}}$:
$\mathcal{C}_{\mathrm{nl}}(f_1)$
x
f
$\mathcal{C}_{\mathrm{l}}(f_1)$
$\pi_{H_{f_1}}$:
$\mathcal{C}_{\mathrm{l}}(f_1)$
f
$\mathcal{C}_{\mathrm{nl}}(f_1)$
x

(b) Schematic line graph of H_{f_2}. f_2 is a Ⓢ mixed child of q. f is the only mixed child of f_2. The configuration between x and a mixed child f is shared across all nodes $p_2 \preceq f_2$.



(c) Schematic line graph of the permutation for π_{q+x}. In $\pi[Q]_{+x}$ the leaf sets f_1', f_1'', f_2' and f_2'' would need to resolved further. When inserting x into $\pi[Q]$ it will have a label in range $[\lambda_{\min}(f_1) : \lambda_{\min}(f_1) + |F_1| + 1]$ and a rank between $[\pi_{\min}(f_2) : \pi_{\max}(f_2) + 1]$, thus splitting up the intervals of the modules F_1 and F_2.

Figure 10: Overview of an example on Algorithm 7c. Here q is a Ⓟ node and both mixed children are degenerate nodes of opposite types and connected in G_q.

Procedure `reorderPrime(`p, π_{H_p}`)`

```
Input: Ⓟ Node $p$ of a modular decomposition tree $T_G$ and the permutation
       $\pi_{p+x}$.
Output: Reordered Node $p$ or the result that $G + x$ is not a permutation
        graph.
init array new_pi of $|\mathcal{C}(p)|$ Node pointers
if $\pi^{-1}_{H_p}(x) = |\pi_{H_p}|$ then           // $x$ is at the last position in $\pi_{H_p}$
    return $p$
else if $\pi^{-1}_{H_p}(x) = 1$ then                // $x$ is at the last position in $\pi^{v}_{H_p}$
    mirror($p$, v)
    return $p$ with updated children according to quotient permutation $\pi^{v}_{p}$.
else if $\lambda_{H_p}(x) = |\mathcal{C}(p)|$ then  // $x$ is at the last position in $\pi^{h}_{H_p}$
    mirror($p$, h)
    return $p$ with updated children according to quotient permutation $\pi^{h}_{p}$.
else if $\lambda_{H_p}(x) = 1$ then                 // $x$ is at the last position in $\pi^{m}_{H_p}$
    mirror($p$, v)
    mirror($p$, h)
    return $p$ with updated children according to quotient permutation $\pi^{m}_{p}$.
else
    return $G + x$ is not a permutation graph.
end
```

```
Procedure mirror($p$, flag $\in \{\mathtt{h}, \mathtt{v}\}$)
  Input: Node $p$ of a modular decomposition tree $T_G$ and a flag to indicate the
         operation to perform.
  Output: Reordered Node $p$.
  init array new_pi of $|\mathcal{C}(p)|$ Node pointers
  if flag $=$ v then                          // vertically mirror, $\lambda_{\text{new}} = n + 1 - \lambda$
      foreach child Node $c \in \mathcal{C}(p)$ do
          $\pi^{-1}_{\text{parent}}(c) \leftarrow |\mathcal{C}(p)| + 1 - \pi^{-1}_{\text{parent}}(c)$
          $\lambda_{\text{parent}}(c) \leftarrow |\mathcal{C}(p)| + 1 - \lambda_{\text{parent}}(c)$
          $\Delta \leftarrow \pi_{\max}(c) - \pi_{\min}(c) + 1$
          $\pi_{\min}(c) \leftarrow |\mathcal{C}(p)| + 1 - \pi_{\max}(c)$
          $\pi_{\max}(c) \leftarrow \pi_{\min}(c) + \Delta$
          $\lambda_{\min}(c) \leftarrow |\mathcal{C}(p)| + 1 - \lambda_{\min}(c) - \Delta$
          new_pi[$\pi^{-1}_{\text{parent}}(c)$] $\leftarrow c$
      end
  else if flag $=$ h then                     // horizontally mirror, $\lambda_{\text{new}} = \pi^{-1}$
      foreach child Node $c \in \mathcal{C}(p)$ do
          $\pi^{-1}_{\text{parent}}(c) \leftarrow \pi^{-1}_{\text{parent}}(c) + \lambda_{\text{parent}}(c)$
          $\lambda_{\text{parent}}(c) \leftarrow \pi^{-1}_{\text{parent}}(c) - \lambda_{\text{parent}}(c)$
          $\pi^{-1}_{\text{parent}}(c) \leftarrow \pi^{-1}_{\text{parent}}(c) - \lambda_{\text{parent}}(c)$
          $\Delta \leftarrow \pi_{\max}(c) - \pi_{\min}(c) + 1$
          $\pi_{\min}(c) \leftarrow \lambda_{\min}(c)$
          $\lambda_{\min}(c) \leftarrow \pi_{\max}(c) - \Delta$
          $\pi_{\max}(c) \leftarrow \pi_{\min}(c) + \Delta$
          new_pi[$\pi^{-1}_{\text{parent}}(c)$] $\leftarrow c$
      end
  Transform new_pi into a doubly linked list and replace $\mathcal{C}(p)$.
  return $p$ with updated children according to quotient permutation $\pi_p^{\mathtt{flag}}$.
```

Procedure `reorderDegenerate(`p`)`

```
Input: $\textcircled{\parallel}$ or $\textcircled{S}$ Node $p$ of a modular decomposition tree $T_G$.
Output: Reordered Node $p$ with (if it exists) the mixed child's label
        equalling the future label of $\lambda_{H_p}(x)$.
init array $\pi_{\text{new}}$ of $|\mathcal{C}(p)|$ Node pointers
if $p$ is typed $\textcircled{\parallel}$ then
    $\varrho \leftarrow |\mathcal{C}(p)|$                    // index for rank
    $\delta \leftarrow -1$                              // offset for next index
else
    $\varrho \leftarrow 1$
    $\delta \leftarrow 1$
end
$\Lambda \leftarrow 1$
foreach $n \in \mathcal{C}_{\text{nl}}(p)$ do
    $\pi_{\text{new}}[\varrho] \leftarrow n$
    $\lambda_{\text{parent}}(n) \leftarrow \Lambda$
    $\pi^{-1}_{\text{parent}}(n) \leftarrow \varrho$
    $\Lambda \leftarrow \Lambda + 1$
    $\varrho \leftarrow \varrho + \delta$
end
foreach $m \in \mathcal{C}_{\text{m}}(p)$ do
    $\pi_{\text{new}}[\varrho] \leftarrow m$
    $\lambda_{\text{parent}}(m) \leftarrow \Lambda$
    $\pi^{-1}_{\text{parent}}(m) \leftarrow \varrho$
    $\Lambda \leftarrow \Lambda + 1$
    $\varrho \leftarrow \varrho + \delta$
end
foreach $l \in \mathcal{C}_{\text{l}}(p)$ do
    $\pi_{\text{new}}[\varrho] \leftarrow l$
    $\lambda_{\text{parent}}(l) \leftarrow \Lambda$
    $\pi^{-1}_{\text{parent}}(l) \leftarrow \varrho$
    $\Lambda \leftarrow \Lambda + 1$
    $\varrho \leftarrow \varrho + \delta$
end
Transform $\pi_{\text{new}}$ into doubly linked list of children of $p$.
return $p$
```

4 Conclusion

I have shown in the previous chapter how to utilise our proposed data structure to compute whether inserting a single vertex x into a permutation graph G_π results in a new permutation graph $G_\pi + x$. We have discussed the time complexity and the different constraints on T_G to recognise the augmented graph as permutation graph. Algorithm 8 summarises the algorithmic approach and shows a possible pipeline to recognise an undirected graph $H = (V, E)$ as permutation graph and even return a vertex labelling λ and an admissible permutation π. The algorithm does not build up H_π from a potential empty graph, but starts with choosing to arbitrary vertices and generates a primitive $\textcircled{S}$ or $\textcircled{\parallel}$ graph, depending on their adjacency.

It should be noted that the `for`-loop, and thus Algorithm 8, fails if and only if the computation of $\pi[Q]_{+x}$ with Algorithm 7 in line 39 fails. The previous steps from lines 13 sqq. only handle the trivial insertion cases and are not meant to fail the whole procedure if the conditions are not met. Although our data structure is needed throughout the whole process, it does not need to be maintained, since each iteration of the `for`-loop (lines 10 sqq.) computes a new bracketed permutation of π and builds a new tree in $\mathcal{O}(|V(G)|)$ time. The mapping λ_π between vertices and their labels, on the other hand, is maintained throughout the whole procedure, e.g., when in Procedure getPi, Procedure mirror and Algorithm 6.

Theorem 4. *Given a procedure as described in Algorithm 8 we can recognise any graph $H = (V, E)$ as permutation graph, if it is possible and compute an admissible permutation and vertex labelling in $\mathcal{O}(|V|^2)$ time and $\mathcal{O}(|V|)$ space.*

Proof. The algorithm spends $\mathcal{O}(|V(G)|)$ time for each iteration of the `for`-loop, as was shown for each of the used algorithms in the previous section. Because with each iteration $|V(G)|$ grows by one vertex, until $|V(G)| = |V| = n$, we conclude that Algorithm 8 runs in $\mathcal{O}(n^2)$ time.

Since the data structure is basically a modified modular decomposition tree, we can conclude through Corollary 1 that space complexity is indeed in $\mathcal{O}(n)$. ■

To summarise: I have shown that an arbitrary undirected graph $H = (V, E)$ can be recognised as permutation graph by an iterative process building up a permutation graph G_π until either $G_\pi \cong H$ or failure. The resulting permutation can even be used to build our modular decomposition tree-based data structure that could be a vantage point for follow-up research, i.e., how to implement further algorithms, such as finding isomorphisms and others.

Algorithm 8: Algorithmic pipeline to recognise a given undirected graph $H = (V, E)$ as permutation graph.

```
Input: Graph $H = (V, E)$ to be recognised as permutation graph.
Output: Permutation $\pi$ of $H_\pi = (H, \lambda)$ and mapping $\lambda$ if $\pi$ exists.
Choose two arbitrary vertices $\{u, v\} \in V$
if $uv \in E$ then
    $\pi \leftarrow (2\ 1)$
else
    $\pi \leftarrow (1\ 2)$
end
Create arbitrary mapping $\lambda$ from $\pi$
$G_\pi \leftarrow (H[\{u, v\}], \lambda)$
foreach $x \in V \setminus V(G)$ do
    Compute bracketed permutation $\Pi$ from $\pi$
    $r \leftarrow$ buildTree($\Pi$)
    link_tree($r, N(x)[V(G)]$)
    if $r$ is LINKED then
        $\pi \leftarrow \pi^{+x(1,|\pi|+1)}$
        continue                          // w/ next $x$ from $V(H) \setminus V(G_\pi)$
    else if $r$ is NOTLINKED then
        $\pi \leftarrow \pi^{+x(|\pi|+1,|\pi|+1)}$
        continue
    end
    $q \leftarrow$ find_inode($r$)
    if $\mathcal{C}_{\mathrm{m}}(q) = \emptyset$ then
        if $q$ is $\textcircled{P}$ and has a twin $t$ then
            if $t \in N(x)$ then
                $\pi \leftarrow \pi^{+x(\lambda(t),\pi^{-1}(t)+1)}$
                continue
            else
                $\pi \leftarrow \pi^{+x(\lambda(t),\pi^{-1}(t))}$
                continue
            end
        else if $q$ is $\textcircled{S}$ or $\textcircled{\parallel}$ then
            $\varrho_x \leftarrow \pi_{\max}(q) + 1$
            $\Lambda_x \leftarrow \lambda_{\min} + |\overline{N}(x)[Q]| + 1$
            $\pi \leftarrow \pi^{+x(\Lambda_x,\varrho_x)}$
            continue
        end
    end
    $\tau_{+x} \leftarrow \pi[V(G) \setminus Q]^{+x(\lambda_{\min}(q),\pi_{\min}(q))}$
    Compute $\pi[Q]_{+x}$
    $\pi \leftarrow \tau_{+x}^{x \to \pi[Q]_{+x}}$                  // use Algorithm 7
end
return $\pi$, $\lambda$ or $H$ is not a permutation graph. // iff Algorithm 7 fails
```

Glossary

Please take note that most definitions are referenced where they occur first in this thesis. This only serves as an overview

adjacent Two vertex sets (modules) $S, S' \subset V$ are adjacent if for any vertices $s \in S$ and $s' \in S'$ it holds that $s \in N(s')$ and vice versa.

degenerate node Either parallel node or series node.

insertion node The insertion node q is the **least common ancestor** of non-proper nodes in T_G.
Note that the insertion node is precicely the **root** of the **subtree** of T_G which has to be updated when inserting x in G, constructing T_{G+x}.

interval An **interval** of a linear order (e.g. a permutation) is a subset of consecutive elements.
For permutations an interval is a run of consecutively labelled elements. *See also:* strong interval.

linked (node) *See* linked node.

linked node A node p of T_G is linked to a vertex $x \in V$ if its representing vetex set $P = V(p) \subseteq V$ is uniform with respect to x and $P \subseteq N(x)$ or notlinked if $P \subseteq \overline{N}(x)$.
Otherwise it is mixed.
The set of linked children of p is denoted $\mathcal{C}_l(p)$, $\mathcal{C}_{nl}(p)$ and $\mathcal{C}_m(p)$ for notlinked and mixed respectively.
See also: linked, notlinked node, mixed node.

maximal strong module If $M \neq V$ is maximal wrt inclusion it is called a maximal strong module. **Connected** and **co-connected components** of G are precisely the maximal strong modules.
See also: module, strong module.

mixed (node) *See* mixed node.

mixed (vertex set). Not uniform.
See also: mixed node.

mixed node Node p of a **modular decomposition tree** T_G that is neither linked nor notlinked.
See also: linked node, notlinked node.

module $M \subseteq V$ is a module iff it is uniform wrt any $x \in V \setminus M$. Particularly V and $\{x\}$ are trivial modules.
See also:.

non-proper node Not proper node.
notlinked (node) *See* notlinked node.
notlinked node *See* linked node.

overlap Two vertex sets (modules) S, S' overlap if $S \cap S' \neq \emptyset$ and they are no subsets of one another, i.e. $S \setminus S' \neq \emptyset$ and $S' \setminus S \neq \emptyset$.
Denoted $S \perp S'$.

parallel node A node p of T_G, representing a strong module $P \subseteq V$ in a **modular decomposition tree** T_G of a graph G, is called parallel if the quotient graph G_p is empty (fully disconnected).
(*Alternatively:* $G[P]$, the **induced subgraph**, is disconnected.)
See also: prime node, series node.
permutation graph A Graph $G = (V, E)$ is a permutation graph if there exists a permutation π with labelling λ such that $\forall uv \in E(G) \iff D_\lambda(u, v) D_\pi(v, u) > 0$.
Furthermore, iff the quotient graph associated with each prime node of the **modular decomposition tree** T_G is a permutation graph, G_π is also a permutation graph.
See also:.
prime node A node p of T_G, representing a strong module $P \subseteq V$ in a **modular decomposition tree** T_G of a graph G, is called prime if the quotient graphs G_p and $\overline{G}_p$ are primitive graphs. Not degenerate node.
(*Alternatively:* $G[P]$, the **induced subgraph**, and $\overline{G}[P]$ are both connected or both disconnected. I.e. not parallel or series.)
See also: parallel node, series node.
primitive graph A Graph G is prime if all its modules are trivial.
proper node A node p of a **modular decomposition tree** T_G is a proper node wrt a vertex $x \in V$ iff the vertex set P in G, corresponding to p, is either uniform wrt x or P is mixed and has a unique mixed child f such that $V(f) = F \cup \{x\}$ is a module of $G'[F \cup \{x\}]$ (G' being the graph $G + x$).

quotient graph [5] If $\bigcup_i M_i = \mathcal{P} \subseteq 2^V$ is a **congruence partition** (e.g. of modules M_i) of a graph $G = (V, E)$, then $G/\mathcal{P}$ describes the graph in which each $M_i \in \mathcal{P}$ is represented by a **representative vertex**, and M_i, M_j are connected if there exists an edge between their **representative vertices** in G.
See also:.

series node A node p of T_G, representing a strong module $P \subseteq V$ in a **modular decomposition tree** T_G of a graph G, is called series if the quotient graph $\overline{G}_p$ complete.
(*Alternatively:* $\overline{G}[P]$, the complementary **induced subgraph**, is not connected.)
See also: parallel node, prime node.

strong interval A interval is called strong if it does not overlap any other interval.
See also: interval.

strong module M is a strong module if it does not overlap any other modules, i.e. there are no other modules with common vertices.
See also: module, maximal strong module.

twin Two vertices x and y are twins iff $N(x) = N(y)$ or $N(x) \cup \{x\} = N(y) \cup \{y\}$.

uniform A subset $S \subseteq V$ is uniform wrt $x \in V \setminus S$ iff $S \subseteq N(x)$ (neighbours of x) or $S \subseteq \overline{N}(x)$, mixed otherwise.
See also: linked node, notlinked node, mixed.

References

[1] B. Dushnik and E. W. Miller. ‘Partially Ordered Sets’. In: *American Journal of Mathematics* 63.3 (1941), pp. 600–610.

[2] A. Pnueli, A. Lempel, and S. Even. ‘Transitive Orientation of Graphs and Identification of Permutation Graphs’. en. In: *Canadian Journal of Mathematics* 23.1 (1971), pp. 160–175.

[3] S. Even, A. Pnueli, and A. Lempel. *Permutation Graphs and Transitive Graphs.* 1972.

[4] T. Gallai. ‘Transitiv orientierbare Graphen’. In: *Acta Mathematica Academiae Scientiarum Hungaricae* 18.1-2 (1967), pp. 25–66.

[5] M. C. Golumbic. *Algorithmic graph theory and perfect graphs. 2nd ed.* English. 2nd ed. Vol. 57. Amsterdam: Elsevier, 2004, pp. xxvi + 314.

[6] A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: A Survey.* Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[7] T. Hartmann et al. ‘Complete Edge-Colored Permutation Graphs’. In: (2020).

[8] H. L. Bodlaender, T. Kloks, and D. Kratsch. ‘Treewidth and Pathwidth of Permutation Graphs’. In: *SIAM Journal on Discrete Mathematics* 8.4 (1995), pp. 606–616.

[9] C. J. Colbourn. ‘On testing isomorphism of permutation graphs’. In: *Networks* 11.1 (1981), pp. 13–21.

[10] S. Mondal, M. Pal, and T. K. Pal. ‘An Optimal Algorithm to Solve the All-Pairs Shortest Paths Problem on Permutation Graphs’. en. In: *Journal of Mathematical Modelling and Algorithms* 2.1 (2003), pp. 57–65.

[11] S. Mondal, M. Pal, and T. Pal. ‘Optimal Sequential And Parallel Algorithms To Compute A Steiner Tree On Permutation Graphs’. In: *International Journal of Computer Mathematics* 80.8 (2003), pp. 937–943.

[12] S. Ramnath and S. Sunder. ‘On two-processor scheduling and maximum matching in permutation graphs’. In: *Information Processing Letters* 57.6 (1996), pp. 321–327.

[13] S. Bhatia, P. Feijão, and A. R. Francis. ‘Position and Content Paradigms in Genome Rearrangements: The Wild and Crazy World of Permutations in Genomics’. en. In: *Bulletin of Mathematical Biology* 80.12 (2018), pp. 3227–3246.

[14] G. R. Galvão, O. Lee, and Z. Dias. 'Sorting signed permutations by short operations'. In: *Algorithms for Molecular Biology* 10.1 (2015), p. 12.

[15] M. Pal, S. Samanta, and A. Pal. *Handbook of Research on Advanced Applications of Graph Theory in Modern Society.* English. IGI Global, 2020.

[16] S. Hougardy. 'Classes of perfect graphs'. en. In: *Discrete Mathematics.* Creation and Recreation: A Tribute to the Memory of Claude Berge 306.19 (2006), pp. 2529–2571.

[17] C. Crespelle and C. Paul. 'Fully Dynamic Algorithm for Recognition and Modular Decomposition of Permutation Graphs'. en. In: *Algorithmica* 58.2 (2010), pp. 405–432.

[18] R. M. McConnell and J. P. Spinrad. 'Modular decomposition and transitive orientation'. en. In: *Discrete Mathematics* 201.1 (1999), pp. 189–241.

[19] D. G. Degiorgi and K. Simon. 'A dynamic algorithm for line graph recognition'. en. In: *Graph-Theoretic Concepts in Computer Science.* Ed. by M. Nagl. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1995, pp. 37–48.

[20] F. Mancini and P. Heggernes. 'A completely dynamic algorithm for split graphs.' In: *Electronic Notes in Discrete Mathematics* 27 (2006), pp. 69–70.

[21] P. Hell, R. Shamir, and R. Sharan. 'A Fully Dynamic Algorithm for Recognizing and Representing Proper Interval Graphs'. In: *SIAM Journal on Computing* 31.1 (2001), pp. 289–305.

[22] L. Ibarra. 'Fully Dynamic Algorithms for Chordal Graphs'. In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '99. Baltimore, Maryland, USA: Society for Industrial and Applied Mathematics, 1999, pp. 923–924.

[23] L. Ibarra. 'Fully dynamic algorithms for chordal graphs and split graphs'. In: *ACM Transactions on Algorithms* 4.4 (2008), pp. 1–20.

[24] L. Ibarra. 'A Fully Dynamic Graph Algorithm for Recognizing Interval Graphs'. en. In: *Algorithmica* 58.3 (2010), pp. 637–678.

[25] S. D. Nikolopoulos, L. Palios, and C. Papadopoulos. 'A fully dynamic algorithm for the recognition of P4-sparse graphs'. en. In: *Theoretical Computer Science* 439 (2012), pp. 41–57.

[26] F. J. Soulignac. 'Fully Dynamic Recognition of Proper Circular-Arc Graphs'. en. In: *Algorithmica* 71.4 (2015), pp. 904–968.

[27] R. H. Möhring. 'Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions'. In: *Annals of Operations Research* 4.1 (1985), pp. 195–225.

[28] R. H. Möhring and F. J. Radermacher. *Substitution Decomposition for Discrete Structures and Connections with Combinatorial Optimization.* 1984.

[29] A. Cournier and M. Habib. 'A new linear algorithm for Modular Decomposition'. en. In: *Trees in Algebra and Programming — CAAP'94.* Ed. by S. Tison. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1994, pp. 68–84.

[30] D. G. Corneil, Y. Perl, and L. K. Stewart. 'A Linear Recognition Algorithm for Cographs'. In: *SIAM Journal on Computing* 14.4 (1985), pp. 926–934.

[31] J. Dusart and M. Habib. 'A new LBFS-based algorithm for cocomparability graph recognition'. In: *Discrete Applied Mathematics* 216 (2017), pp. 149–161.

[32] H. Kaplan and Y. Nussbaum. 'A Simpler Linear-Time Recognition of Circular-Arc Graphs'. en. In: *Algorithmica* 61.3 (2011), pp. 694–737.

[33] R. M. McConnell and J. P. Spinrad. 'Linear-time transitive orientation'. In: *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms.* 1997, pp. 19–25.

[34] E. M. Eschen and J. Spinrad. '$\mathcal{O}(n^2)$ recognition and isomorphism algorithms for circular arc graphs'. In: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Austin, TX, SIAM, Philadelphia.* 1993, pp. 128–137.

[35] A. Bergeron et al. 'Computing Common Intervals of K Permutations, with Applications to Modular Decomposition of Graphs'. en. In: *Algorithms – ESA 2005.* Ed. by G. S. Brodal and S. Leonardi. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 779–790.

[36] C. Capelle, M. Habib, and F. Montgolfier. 'Graph Decompositions and Factorizing Permutations'. en. In: *Discrete Mathematics and Theoretical Computer Science* 5 (2002), pp. 55–70.

[37] T. Uno and M. Yagiura. 'Fast Algorithms to Enumerate All Common Intervals of Two Permutations'. en. In: *Algorithmica* 26.2 (2000), pp. 290–309.

[38] F. d. Montgolfier, M. Habib, and Université des sciences et techniques de Montpellier 2 (1970-2014). 'Décomposition modulaire des graphes: théorie, extensions et algorithmes'. French. OCLC: 493114818. PhD thesis. S.l., 2003.

[39] A. Ehrenfeucht et al. 'An $\mathcal{O}(n^2)$ Divide-and-Conquer Algorithm for the Prime Tree Decomposition of Two-Structures and Modular Decomposition of Graphs'. en. In: *Journal of Algorithms* 16.2 (1994), pp. 283–294.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Ort, Datum　　　　　　　　　　Unterschrift