Adding an Edge in Comparability and Permutation Graphs

A Thesis

submitted to the designated

by the Assembly

of the Department of Computer Science and Engineering

Examination Committee

by

Konstantinos Stamatis

in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER

SYSTEMS ENGINEERING

WITH SPECIALIZATION

IN DATA SCIENCE AND ENGINEERING

University of Ioannina

School of Engineering

Ioannina 2025

Examining Committee:

- **Leonidas Palios,** Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)

- **Euripides Markou,** Professor, Department of Computer Science and Engineering, University of Ioannina

- **Christos Nomikos,** Associate Professor, Department of Computer Science and Engineering, University of Ioannina

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

Konstantinos Stamatis, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, June 2025.

Adding an Edge in Comparability and Permutation Graphs

Supervisor: Leonidas Palios, Professor

Expanding a graph $G(V, E)$ of a class C by introducing a node $v \notin V$ and connecting it with a certain node $v_i \in V$ does not necessarily produce a graph that belongs to C. This thesis is concerned with finding and implementing algorithms that decide if the mere addition of an edge between v and v_i to G results in a graph that belongs to C, and if not, calculates the minimum number of edges that need to be added to the resulting graph for it to belong to C, where C either refers to comparability or permutation graphs. In the case of comparability graphs, we prove that our problem reduces to finding a transitive orientation $\vec{F}$ of G in which v_i has minimum out-degree, and we construct an algorithm by taking advantage of G's maximal multiplices' connection to its transitive orientations. We use the connection between the classes of permutation and comparability graphs to transform that algorithm to one that provides an answer for permutation graphs. We also provide another algorithm that solves the problem for permutation graphs and relies on the permutations that represent G. The algorithms were implemented in the Python programming language and explanation on the implementation is provided.

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Κωνσταντίνος Σταμάτης, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2025

Προσθήκη Ακμής σε Μεταβατικά και Μεταθετικά Γραφήματα

Επιβλέπων: Λεωνίδας Παληός, Καθηγητής

Η επέκταση ενός γραφήματος $G(V, E)$ μιας κλάσης C μέσω της εισαγωγής ενός κόμβου $v \notin V$ και της σύνδεσής του με έναν κόμβο $v_i \in V$ δεν παράγει απαραίτητα ένα γράφημα που ανήκει στην C. Αυτή η διατριβή αφορά την εύρεση και υλοποίηση αλγορίθμων που αποφασίζουν εάν η απλή προσθήκη μιας ακμής μεταξύ v και v_i στο G έχει ως αποτέλεσμα ένα γράφημα που ανήκει στην C, και εάν όχι υπολογίζει το ελάχιστο πλήθος ακμών που χρειάζεται να προστεθούν στο γράφημα που προέκυψε ώστε αυτό να ανήκει στην C, όπου C είτε αναφέρεται σε μεταβατικά (comparability) ή μεταθετικά (permutation) γραφήματα.

Στην περίπτωση των μεταβατικών γραφημάτων, αποδεικνύουμε ότι το πρόβλημα ανάγεται στον υπολογισμό μιας μεταβατικής κατεύθυνσης (transitive orientation) $\vec{F}$ του G στην οποία η v_i έχει ελάχιστο out-degree, και κατασκευάζουμε έναν αλγόριθμο εκμεταλλευόμενοι το γεγονός ότι η ένωση των μεταβατικών κατευθύνσεων όλων των μεγιστικών multiplices ενός γραφήματος είναι μεταβατική κατεύθυνση του γραφήματος αυτού, που μας επιτρέπει να ενδιαφερθούμε για το out-degree της v_i μόνο στα μεγιστικά multiplices στα οποία ανήκουν ακμές που την περιέχουν.

Χρησιμοποιούμε το γεγονός ότι τα μεταθετικά γραφήματα είναι ακριβώς τα μεταβατικά γραφήματα με συμπλήρωμα που είναι μεταβατικό γράφημα για να μετατρέψουμε τον παραπάνω αλγόριθμο σε έναν αλγόριθμο που δίνει απάντηση για τα μεταθετικά γραφήματα.

Επιπλέον, παραθέτουμε τη σχεδίαση ενός ακόμα αλγορίθμου που λύνει το πρόβλημα για μεταθετικά γραφήματα, ο οποίος βελτιώνει σημαντικά την εξαντλητική αναζήτηση κατά την

οποία προστίθεται ένας αριθμός για τον νέο κόμβο v σε κάθε μετάθεση που αναπαριστά το G έτσι ώστε να σχηματίζεται αναστροφή με τον ακέραιο που αντιστοιχεί στον v_i.
Οι αλγόριθμοι υλοποιήθηκαν στη γλώσσα προγραμματισμού Python και τα γραφήματα μοντελοποιούνται μέσω του module networkx, και εξηγούνται αναλυτικά μετά την παράθεση του κάθε αλγορίθμου.

CHAPTER 1

INTRODUCTION

1.1 Theoretical Framework

Let X, Y be sets, i.e. collections of unique items (called *elements* of the set). A *function* from X to Y, denoted $f: X \longrightarrow Y$ is a rule which associates to each element of X exactly one element of Y. f is called *one-to-one* if $\forall x, y \in X: f(x) = f(y) \Rightarrow x = y$. f is called *onto* if $\forall y \in Y, \exists x \in X: f(x) = y$. A function that is both one-to-one and onto is called a *bijection.* For two logical propositions p and q, their logical conjunction will be denoted by $p \wedge q$ and their logical disjunction by $p \vee q$ [1].

If A and B are two sets, A is *contained in* B (or B *contains* A), denoted by $A \subseteq B$, if $\forall x: x \in A \Rightarrow x \in B$. A is *equal to* B, denoted by $A = B$, if $A \subseteq B \wedge B \subseteq A$. A is *properly contained in* B (or B *properly contains* A) if $A \subseteq B \wedge A \neq B$ [1].

Let V be a *nonempty set*, i.e. a set with at least one element. A *binary relation* R on V is a function $R: V \to \mathcal{P}(V)$, where $\mathcal{P}(V)$ is V's *powerset,* i.e. the set that contains all V's subsets [1]. R is *reflexive* if and only if $\forall x \in V: x \in R(x)$. R is *irreflexive* if and only if $\forall x \in V: x \notin R(x)$. R is *symmetric* if and only if $\forall x, y \in V: x \in R(y) \Leftrightarrow y \in R(x)$. R is *antisymmetric* if and only if $\forall x, y \in V: x \in R(y) \Rightarrow y \notin R(x)$. R is *transitive* if and only if $\forall x, y, z \in V: (z \in R(y) \wedge y \in R(x)) \Rightarrow z \in R(x)$. It is typical to denote R as a collection of ordered pairs of the type

(x, y) where $(x, y) \in R \Leftrightarrow y \in R(x)$. A binary relation R is called an *equivalence relation* if it is reflexive, symmetric, and transitive, a *partial order* (or *ordering*) if it is reflexive, antisymmetric, and transitive, and, finally, a *strict partial order* if it is irreflexive and transitive. The *reflexive and transitive closure* R^* of a binary relation $R: X \to X$ is the smallest with respect to $\subseteq$ relation that contains R that is also reflexive and transitive [2].

The union, intersection, and difference of two sets A and B will be denoted as $A \cup B$, $A \cap B$ and $A \backslash B$ respectively. If $\{A_i\}_{i \in I}$, where I is a set of indices, is a family (or simply set) of sets their union and intersection will be symbolized as $\bigcup_{i \in I} A_i$ and $\bigcap_{i \in I} A_i$ respectively. Two sets A and B are called *disjoint* if $A \cap B = \emptyset$, which shall denote the *empty set* [1]. The sets in $\{A_i\}_{i \in I}$ are called *pairwise disjoint* if and only if $\forall i, j \in I, i \neq j: A_i \cap A_j = \emptyset$. If the sets in $\{A_i\}_{i \in I}$ are pairwise disjoint subsets of a set A and $\bigcup_{i \in I} A_i = A$ then they are a *partition* of A. When some sets are pairwise disjoint, we may use the symbol $+$ to denote their union instead of $\cup$ and their union will be called a *disjoint union*. The *cartesian product* of sets $\{X_i\}_{i \in T_k}$, for some $k \in \mathbb{N} = \{1,2,3 \dots\}$, where $T_k = \{1, \dots, k\}$ will be denoted as $X_1 \times \dots \times X_k$ or $\prod_{i=1}^{k} X_i$ and is defined as $\{(x_1, \dots, x_k) \mid \forall i \in T_k: x_i \in X_i\}$. If $\forall i, j \in T_k: X_i = X_j = X$, then $X_1 \times \dots \times X_k$ will be called the *cartesian product of X with itself k times* and will be denoted by X^k [3]. If A is a set, then $|A|$ shall denote the number of its elements (also referred to as its *cardinality*).

A *directed graph* G consists of a finite set V and an irreflexive binary relation on V, which will be represented as a collection E of ordered pairs or as a function $Adj: V \to \mathcal{P}(V)$. V shall be G's *node set* or *set of nodes* and its elements shall be called *nodes*. We shall call $Adj(u)$ the *adjacency set* of node u and $(u, v) \in E$ an *edge*. Clearly, $(u, v) \in E \Leftrightarrow v \in Adj(u)$. If this is the case u and v are *adjacent* and they are also *endpoints* of the edge (u, v), more specifically, u is the *head* and v is the *tail* of the edge. For the sake of simplicity, we will denote edges as $\overrightarrow{xy} \in E$ instead of $(x, y) \in E$. The *out-degree* of a node v in G is $|Adj(v)|$ while its *in-degree* is $|\{u \in V \mid v \in Adj(u)\}|$. A node is called a *source* when its in-degree is 0 and a sink when its out-degree is 0.

If H is a collection of ordered pairs then we define its *inverse* as $H^{-1} = \{(x, y) \mid (y, x) \in H\}$. An *undirected graph* (or simply *graph*) G is a directed graph whose edge set is equal to its inverse, i.e. $E = E^{-1}$ or equivalently E is symmetric. This means that $\forall \overrightarrow{ab} \in E: \overrightarrow{ba} \in E$. If v is a node in an undirected graph, its *degree* will be $|Adj(v)|$. In undirected graphs, we will denote the existence of both $\overrightarrow{ab}$ and $\overrightarrow{ba}$ in E with $ab \in E$ and ab will be called an *undirected edge*. Graphs, both directed and directed, shall be denoted just by their name

(e.g. G) or by their name followed by an ordered pair that includes their node and edge sets in this order (e.g. $G(V,E)$).

An undirected graph is called *complete* if every distinct pair of its nodes is adjacent. A complete graph with n nodes is notated by K_n. The *complement* of an undirected graph $G(V,E)$ is notated by $\bar{G}(V,\bar{E})$ where $\bar{E} = \{(x,y) \in V^2 | \ x \neq y \ \wedge \ (x,y) \notin E\}$. If $S \subseteq V$ then the *subgraph induced by* S is defined as $G[S](S, E[S])$ where $E[S] = \{(x,y) \in E \mid \{x,y\} \subseteq S\}$. $G[S]$ is an *induced subgraph* of G. A *partial subgraph* of $G(V,E)$ is any graph $H(V',E')$ such that $V' \subseteq V \wedge E' \subseteq E$. An undirected graph $G(V,E)$ is called *connected* if $\forall u,v \in V \ \exists (u_1,\dots,u_k) \in V^k : u = u_1 \wedge v = u_k \wedge (\forall i \in T_{k-1} : u_i u_{i+1} \in E)$. Such an element of V^k is called a *chain from* u *to* v.

Two graphs $G(V,E)$ and $G'(V',E')$ are called *isomorphic*, denoted $G \cong G'$, if there exists a bijection $f : V \to V'$ such that $\forall x,y \in V : (x,y) \in E \Leftrightarrow (f(x), f(y)) \in E'$. Such a bijection will be called an *isomorphism* between G and G'.

If $G(V,E)$ is an undirected graph, then a collection of ordered pairs $\vec{F}$ is an *orientation* of G (or of E or of the edges of G) if $\vec{F} \cap \vec{F}^{-1} = \emptyset$ and $\vec{F} + \vec{F}^{-1} = E$. An orientation $\vec{F}$ of G (or of E) is *transitive* if $\{\overrightarrow{ac} \mid \exists b \in V : \overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}\} \subseteq \vec{F}$. G is called a *comparability graph* if there exists a transitive orientation of its edges.

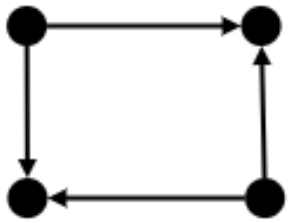

Figure 1.1: A comparability graph (with a transitive orientation)

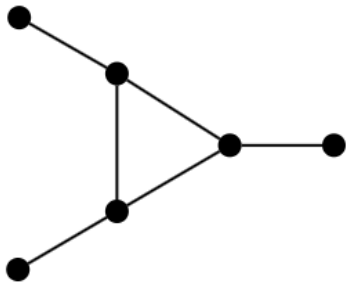

Figure 1.2: A non-comparability graph

If $\pi = [\pi_1, \dots, \pi_n]$ is a permutation of T_n, $n \geq 1$ then its inversion graph $G[\pi](V,E)$ is defined as: $V = T_n$ and $ij \in E \Leftrightarrow (i-j)\left(\pi_i^{-1} - \pi_j^{-1}\right) < 0$. An undirected graph G is called a *permutation graph* if there exists a permutation π such that $G \cong G[\pi]$. Every permutation π whose inversion graph is isomorphic to G is said to *represent* G. It is noted that $G[X]$ will refer to an induced subgraph of $G(V,E)$ If $X \subseteq V$ and to an inversion graph if X is a permutation of T_n for some $n \in \mathbb{N}$.

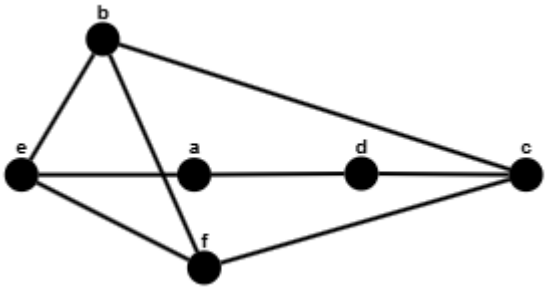

Figure 1.3: A graph isomorphic to $G[4,3,6,1,5,2]$

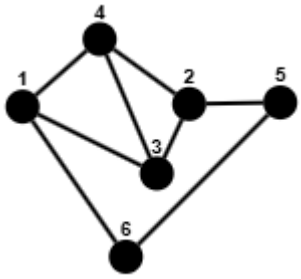

Figure 1.4: $G[4,3,6,1,5,2]$

All definitions apart from the ones whose source is cited are taken from [4].

1.2 Objective of the Thesis

Consider the following question: If $G(V, E)$ is a graph belonging to a graph class C, $v_i \in V$, $v \notin V$, and $G + v_i v(V_{new}, E_{new})$ is a graph where $V_{new} = V \cup \{v\}$ and $E_{new} = E \cup \{v_i v\}$ then does $G + v_i v$ belong to the class C, and if not, what is the smallest number possible of edges incident on v that should be added to $G + v_i v$ in order for the resulting graph to belong to C and how can we compute that efficiently?

This question has a trivial answer when C represents the class of connected graphs or trees: in both cases, $G + v_i v$ is in C. There are also linear time algorithms for split, quasi-threshold, threshold and P_4-sparse graphs that take advantage of the structure of graphs in these classes [5]. However, it seems that not many results that concern other perfect graph classes exist. When it comes to comparability and permutation graphs specifically, this apparent lack of such fruitful research (apart from works like [6] and [7] that tackle related questions for comparability graphs) was the main motivation behind us working on this problem for these two classes.

Despite this being the main goal, a deeper knowledge and understanding of the structures and properties of both comparability and permutation graphs was not only a means through which we hoped to attain some algorithmic approach to our inquiries, but also a goal itself. This thesis also provided us with the opportunity to not only gain such knowledge by

studying the works of some of the most important figures in Graph Theory and possibly designing some novel algorithms for our main question, but also to implement our algorithms in code that would be as precise, effective and thoroughly explained as possible.

1.3 Thesis Structure

Apart from the chapter that you are currently reading, this thesis contains three more chapters. The two that directly follow this one are each entirely dedicated to comparability and permutation graphs respectively, containing all our mathematical results, required definitions not included in this chapter, a description of our algorithms design and an in-depth explanation of our code implementation for each class. The last chapter contains comments on our presented work and possible directions for future research.

The inner structure of the chapters that concern a single graph class contain at least three discernible sections. The first one contains the definitions and mathematical results that support our algorithm and provide a direction for its design. The second one contains a detailed explanation of the design along with proof/examination of its correctness and complexity and, finally, the third one contains a step-by-step guide to understanding our code implementation. In the chapter that concerns permutation graphs there exists a fourth section in which an alternative algorithm is suggested and thoroughly described. The last chapter is made up of only two paragraphs, the first one concerning comments on our work and another one which we hope will provide motivation for further research in the field.

CHAPTER 2

ON COMPARABILITY GRAPHS

2.1 Definitions & Main Idea

In this section, we will explore the suggested approach for solving our problem for comparability graphs, which takes advantage of the existence of groups of edges called multiplices, whose transitive orientations when combined always result in a transitive orientation of a comparability graph.

Let $G(V, E)$ be a comparability graph with $V = \{v_1, \dots, v_n\}$, let v be a vertex not in V and $i \in T_n$. If $G + v_i v$ (V_{new}, E_{new}) where $V_{new} = V \cup \{v\}$ and $E_{new} = E \cup \{v_i v\}$ then we want to know if $G + v_i v$ is a comparability graph and if it is not, we need to figure out a way to turn it into one by adding to it the smallest number possible of edges incident on v.

The truth is that there is an equivalent condition to $G + v_i v$ being a comparability graph mentioned in [8] with an adumbration of its proof. This condition, followed by a more formal proof, is exactly the following:

Lemma 2.1. *$G + v_i v$ is a comparability graph if and only if there exists a transitive orientation of G in which v_i is a sink.*

Proof. (⇒) Let $G + v_iv$ be a comparability graph. Since v is only connected to v_i in $G + v_iv$, v will be either a source or a sink in any transitive orientation of $G + v_iv$. Let $\vec{F}$ be a transitive orientation of $G + v_iv$ in which v is a source. Such an orientation exists because $\vec{F}$ is a strict partial ordering of V_{new} [4] and the inverse of a strict partial ordering is also a strict partial ordering [1]. Thus $\vec{F}^{-1}$ is also a transitive orientation of $G + v_iv$. Then, since $\forall j \in T_n\backslash\{i\}$, $vv_j \notin E$ and $\overrightarrow{vv_i} \in \vec{F}$, we have that $\forall j \in \{k \in T_n\backslash\{i\} \mid v_kv_i \in E\}$: $\overrightarrow{v_jv_i} \in \vec{F}$, therefore v_i is a sink in $\vec{F}$. Let $\vec{F}' = \vec{F}\backslash\{\overrightarrow{vv_i}\}$. Obviously v_i is a sink in $\vec{F}'$ and this orientation is an orientation of G: Let $\overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}'$. Since $\overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}$, which is a transitive orientation, we have that $ac \in \vec{F}$ and, since $a, b, c \neq v$, it is true that $ac \in \vec{F}'$. Thus, $\vec{F}'$ is a transitive orientation of G in which v_i is a sink.

(⇐) If $\vec{F}$ is a transitive orientation of G in which v_i is a sink, let $\vec{F}' = \vec{F} \cup \{\overrightarrow{vv_i}\}$, an orientation of $G + v_iv$, and let $\overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}'$. We shall prove that $\overrightarrow{ac} \in \vec{F}'$. If we assumed that $a = v$ we would have that $b = v_i$, which would mean that there is an edge of the type $\overrightarrow{v_ix}, x \in V_{new}\backslash v_i$ in $\vec{F}'$ which is not the case for any such x because v_i is a sink in $\vec{F}$, so $a \neq v$. Similarly, $b \neq v$ because if $b = v$ it would mean that $\exists x \in V$: $\overrightarrow{xv} \in \vec{F}'$ which again is not the case and, finally, for the same exact reason, $c \neq v$. So, $\{a, b, c\} \cap \{v\} = \emptyset$ or, equivalently, $\{a, b, c\} \subseteq V$. We thus deduce that $\overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}$, which is a transitive orientation of G, therefore $ac \in \vec{F} \subseteq \vec{F}'$. □

An immediate consequence of *Lemma 2.1* is that if a transitive orientation of G in which v_i is a sink does not exist, we would need to add at least one more edge of the type $vx, x \in V$ to $G + v_iv$ to create a comparability graph $G_v(V_{new}, E_v)$. Given a transitive orientation $\vec{F}$ of G, an easy way to produce a comparability graph by adding edges incident on v to $G + v_iv$ is to connect v with every vertex in $S = \{x \in V \mid \overrightarrow{v_ix} \in \vec{F}\}$, as shown in the following lemma.

Lemma 2.2. *Let $S = \{x \in V \mid \overrightarrow{v_ix} \in \vec{F}\}$. If $E_v = E_{new} \cup \{vx \mid x \in S\}$ and $\vec{F}$ is a transitive orientation of G then $G_v(V_{new}, E_v)$ is a comparability graph and $\vec{F}' = \vec{F} \cup \{\overrightarrow{vx} \mid x \in S \cup \{v_i\}\}$ is a transitive orientation of the edges of G_v.*

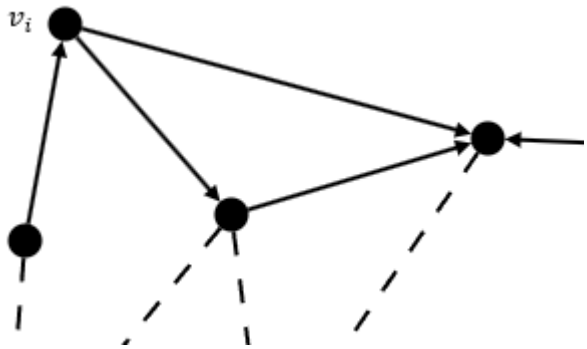

Figure 2.1: An indicative initial condition

Proof. Let $\overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}'$. We shall prove that $\overrightarrow{ac} \in \vec{F}'$. If $\{a, b, c\} \cap \{v\} = \emptyset$ then $a, b, c \in V, \overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}$ and since $\vec{F}$ is a transitive orientation of G, $\overrightarrow{ac} \in \vec{F} \subseteq \vec{F}'$. Moreover, b and c can never be v since there is no edge of the type $\overrightarrow{xv}, x \in V$ in $\vec{F}'$. If $a = v$ then b is either v_i or a neighbor of v_i in G such that $\overrightarrow{v_i b} \in \vec{F}$, i.e. $b \in S$. If $b = v_i$ then obviously, by the definition of $\vec{F}', \overrightarrow{ac} \in \vec{F}'$. If, on the other hand, b is a neighbor of v_i in G such that $\overrightarrow{v_i b} \in \vec{F}$, then since $b, c \in V$ and $\overrightarrow{bc} \in \vec{F}'$ then $\overrightarrow{bc} \in \vec{F}$ which in turn means that $\overrightarrow{v_i c} \in \vec{F}$ (due to $\vec{F}$ being transitive) and this, by the definition of $\vec{F}'$ proves that $\overrightarrow{ac} \in \vec{F}'$. □

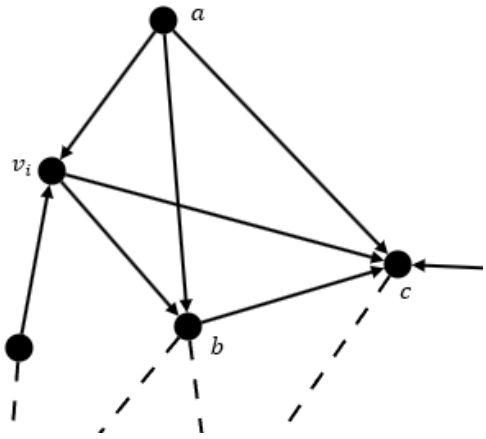

Figure 2.2: b as a neighbor of v_i

This lemma provides us, apart from a guarantee that the construction of a comparability graph by adding zero or more edges to $G + v_i v$ that are incident on v is feasible, with a method of extending $G + v_i v$ into a bigger comparability graph G_v while constructing a transitive orientation $\vec{F}'$ of G_v that includes all directed edges of a given transitive orientation $\vec{F}$

of G and edges that connect v with v_i and all vertices in S, with v being a source. This procedure adds precisely as many edges as the out-degree of v_i in $\vec{F}$ to E_{new}. The main proposition of this chapter, which establishes the connection between the smallest possible number of edges incident on v that must be added to $G + v_i v$ for the result to be a comparability graph and the smallest possible out-degree of v_i amongst all transitive orientations of G, is proven right after the proof of the following lemma.

Lemma 2.3. *Let $G(V, E)$ be a comparability graph, $\vec{F}$ be a transitive orientation of G and $S = \{x \in V \mid \overrightarrow{v_i x} \in \vec{F}\} \subseteq V$. Then $\overrightarrow{F_S} = \{\overrightarrow{xy} \in \vec{F} \mid x, y \in S\}$ is a transitive orientation of $G[S]$.*

Proof. Let $\overrightarrow{ab}, \overrightarrow{bc} \in \overrightarrow{F_S}$. Since $\overrightarrow{F_S} \subseteq \vec{F}$ we have that $\overrightarrow{ab}, \overrightarrow{bc} \in \vec{F}$ and since $\vec{F}$ is a transitive orientation of G, $\overrightarrow{ac} \in \vec{F}$. However, the fact that $\overrightarrow{ab}, \overrightarrow{bc} \in \overrightarrow{F_S}$ means that $a, c \in S$. Therefore $\overrightarrow{ac} \in \overrightarrow{F_S}$ and $\overrightarrow{F_S}$ is a transitive orientation of $G[S]$. □

Proposition 2.4. *The smallest number of edges incident on v that we need to add to $G + v_i v$ for it to become a comparability graph is equal to the smallest possible out-degree of v_i amongst all transitive orientations of G.*

Proof. Let $x \in \mathbb{N} \cup \{0\}$ be the smallest number of edges incident on v that we need to add to $G + v_i v$ for it to become a comparability graph. Trivially, the method that is described in *Lemma 2.2* would produce a comparability graph by adding less than x edges to $G + v_i v$ if there existed a transitive orientation of G in which v_i had an out-degree less than x to be used in the method, therefore the smallest possible out-degree of v_i amongst all transitive orientations of G is at least x. Let us now assume that the smallest possible out-degree of v_i amongst all transitive orientations of G is $y > x$. Let G_v be a comparability graph that results from adding x edges incident on v to $G + v_i v$ and let $\vec{F}'$ be a transitive orientation of this graph. Since $\vec{F} = \vec{F}' \backslash \{\overrightarrow{vu}, \overrightarrow{uv} : u \in V\}$ is the restriction of $\vec{F}'$ to the nodes in $V \subseteq V_{new}$, by *Lemma 2.3* it is a transitive orientation of G, and thus the out-degree of v_i in $\vec{F}$ is at least y, which is also true for $\vec{F}'$. If $\overrightarrow{vv_i} \in \vec{F}'$ then, because $\vec{F}'$ is a transitive orientation of G_{new}, $\overrightarrow{vu} \in \vec{F}'$ $\forall u \in \{t \in V \mid \overrightarrow{v_i t} \in \vec{F}\}$. But $|\{t \in V \mid \overrightarrow{v_i t} \in \vec{F}\}| \geq y > x$ which is a contradiction. If, on the other hand, $\overrightarrow{v_i v} \in \vec{F}'$, similarly, $\overrightarrow{uv} \in \vec{F}'$ $\forall u \in \{t \in V \mid \overrightarrow{t v_i} \in \vec{F}\}$. However, $|\{t \in V \mid \overrightarrow{t v_i} \in \vec{F}\}| \geq y > x$ since if there existed a transitive orientation of G in which v_i's in-degree was

smaller than y, this would mean that v_i's out-degree in the inverse of this orientation is also smaller than y, which is impossible, so we are again led to a contradiction. Thus, the smallest possible out-degree of v_i among all transitive orientations of G is x. □

This proposition allows us to reduce our problem to computing a transitive orientation of G in which v_i has the smallest possible out-degree. Before we can do this, we will need to provide a few important definitions that we will be using extensively throughout the rest of this thesis.

Definition 2.5. The binary relation Γ on the edges of an undirected graph $G(V, E)$ is defined as: $\overrightarrow{ab}\ \Gamma\ \overrightarrow{a'b'} \Leftrightarrow (a = a' \wedge \overrightarrow{bb'} \notin E) \vee (\overrightarrow{aa'} \notin E \wedge b = b')$. [4]

It is easy to show that the reflexive, transitive closure Γ^* of Γ is an equivalence relation on E [4], which allows us to provide the following definition.

Definition 2.6. The equivalence classes into which Γ^* partitions the edge set E of a graph $G(V, E)$ will be called *implication classes* of G. A set $C \subseteq E$ is called a *color class* of G if there exists an implication class A of G such that $C = A \cup A^{-1}$. [4]

For example, the implication classes of the graph in Figure 2.3 are, by the definitions of Γ, Γ^* and an equivalence class, $A_1 = \{\overrightarrow{ac}, \overrightarrow{ad}, \overrightarrow{ae}\}$, $A_2 = \{\overrightarrow{cb}, \overrightarrow{db}, \overrightarrow{eb}\}$, $A_3 = \{\overrightarrow{cd}\}$, $A_4 = \{\overrightarrow{ab}\}$, $A_5 = A_1^{-1}$, $A_6 = A_2^{-1}$, $A_7 = A_3^{-1}$ and $A_8 = A_4^{-1}$. It follows that its color classes are $C_1 = A_1 \cup A_1^{-1} = \{ac, ad, ae\}$, $C_2 = A_2 \cup A_2^{-1} = \{cb, db, de\}$, $C_3 = A_3 \cup A_3^{-1} = \{cd\}$ and $C_4 = \{ab\}$ as is shown in Figure 2.3 where the edges of each color class are presented in a different style.

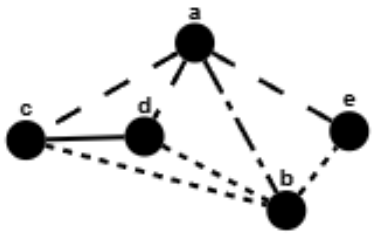

Figure 2.3: Color classes example

Definition 2.7. Let $G(V, E)$ be an undirected graph. A complete subgraph of G induced by V_S (whose edge set we will denote by S) on $r + 1$ nodes is called a *simplex* of *rank* r if each undirected edge ab of S is contained in a different color class of G. A simplex is called *maximal* if it is not properly contained in any larger simplex. [4]

To demonstrate what a simplex is, we can observe that the graph in Figure 2.3 has four complete subgraphs on 3 nodes (the ones induced by $\{a, c, d\}$, $\{c, b, d\}$, $\{a, b, d\}$ and $\{a, b, e\}$). In the subgraphs induced by the first two node sets, there are pairs of edges that belong to the same color class, therefore these subgraphs are not simplices. On the other hand, the complete subgraphs induced by $\{a, b, d\}$ and $\{a, b, e\}$ are simplices of rank 2 because their three edges belong to three different color classes. Since there are no larger complete subgraphs than these in the graph of Figure 2.3, there are also no simplices of rank higher than 2 (which guarantees that the simplices we mentioned are maximal). It is also worth noting that any two neighboring nodes induce a complete subgraph with exactly one edge, thus the subgraph is trivially a simplex of rank 1.

It is noted that the node set or even the edge set of a simplex may be referred to as simplices. The following and final definition of this chapter introduces the notion of the multiplex which will be essential to our algorithm.

Definition 2.8. The *multiplex* generated by a simplex S of rank r of a graph $G(V, E)$ is defined to be the partial subgraph $G_M(V_M, M)$ of G where $M = \{\overrightarrow{ab} \in E \mid \exists \overrightarrow{xy} \in S: \overrightarrow{ab}\ \Gamma^* \overrightarrow{xy}\}$. A multiplex is *maximal* if it is not properly contained in any larger multiplex. [4]

A careful look at Definition 2.8 shall reveal that a multiplex is nothing more than the union of the color classes to which the edges of a particular simplex belong. For example, returning to the graph in Figure 2.3, the multiplex M generated by the simplex $S = \{a, b, e\}$ contains all edges of the graph minus cd. It follows that a simplex of rank 1 generates a multiplex whose edge set is a color class. We will from now on, often use the name "multiplex" to refer to just the edges of a multiplex instead of the subgraph itself.

There are many important results that concern simplices, multiplices and how they are related to the transitive orientations of a comparability graph [4], [9], but the three that are the most significant for this thesis all appear in [4] and are presented below.

Theorem 2.9. *Let M be the multiplex generated by a simplex S. Then, M is a maximal multiplex if and only if S is a maximal simplex.*

Theorem 2.10. *If M_1 and M_2 are maximal multiplices of an undirected graph G, then either $M_1 \cap M_2 = \emptyset$ or $M_1 = M_2$.*

Theorem 2.11. *Let $G(V, E)$ be an undirected graph and let $E = M_1 + \cdots + M_k$ where each M_i is a maximal multiplex of E.*
(i) If $\vec{F}$ is a transitive orientation of G, then $F \cap M_i$ is a transitive orientation of (V_{M_i}, M_i).
(ii) If $\overrightarrow{F_1}, \ldots, \overrightarrow{F_k}$ are transitive orientations of $M_1, \ldots, M_k$, respectively, then $\overrightarrow{F_1} + \cdots + \overrightarrow{F_k}$ is a transitive orientation of G.
(iii) $t(G) = t(M_1) \cdot \ldots \cdot t(M_k)$ where $t(U)$ is the number of possible transitive orientations of an undirected graph U.
(iv) If G is a comparability graph and $r_i = rank(M_i)$, then $t(G) = \prod_{i=1}^{k}(r_i + 1)!$.

Among the four statements of *Theorem 2.11*, the one that is the most relevant for this section's method is (ii), which essentially states that the disjoint union of transitive orientations of all different unique maximal multiplices of a comparability graph is a transitive orientation of that graph. Thus, the main idea behind our method is to calculate all maximal multiplices of G and then find a transitive orientation of each maximal multiplex that does not include v_i and a transitive orientation of each maximal multiplex that includes v_i in which v_i has the smallest out-degree possible and then combine all these transitive orientations into a transitive orientation $\vec{F}$ of G in which v_i has minimum out-degree. This minimum out-degree of v_i will be equal to the smallest possible number of edges incident on v that we could add to $G + v_i v$ in order to produce a comparability graph, and $S = \{x \in V \mid \overrightarrow{v_i x} \in \vec{F}\}$ will be the set of nodes that we would need to connect v with.

A more thorough explanation of our algorithm and all its steps is provided in the next paragraph, along with proof of its correctness.

2.2 Algorithm-Multiplices

To fully execute the algorithm that was just briefly described, we would need to fulfill the following steps in the order that they appear, except one specific case where the order does

not impact the execution, and this case is noted. The following steps only describe what is executed after the graph is provided by the user.

First, we enumerate all cliques of G and save them. We are going to verify which of these cliques are simplices and which are not.

Then, we calculate and save all color and implication classes of G which are needed to determine whether a certain clique $S \subseteq V$ is a simplex or not, and to then generate the maximal multiplices of G by its simplices that are maximal. These first two steps could be executed in reverse order since neither of these two impacts the other.

Right after the completion of the first two steps, in whichever order they are implemented, we need to figure out which of the aforementioned cliques are indeed simplices. After doing that, the focus shall be on what simplices are maximal. These simplices are the only ones that we need to have saved from now on since the other simplices are inconsequential due to them not generating maximal multiplices, which are decisive to the answering of our question.

We immediately use G's maximal simplices to generate its maximal multiplices. After removing any duplicates that may have occurred during this process, we are only left with a list where each maximal multiplex appears only once and all (undirected versions of) edges of E appear in exactly one multiplex.

At this point, for every one of the maximal multiplices that we have saved we examine whether they contain an edge that is incident on v_i and, depending on this, we produce a transitive orientation of the multiplex if it does not contain such an edge, or a transitive orientation of the multiplex in which v_i has minimum out-degree in the second case. To achieve the latter, we consider the implication classes $A_1, \dots, A_k, A_1^{-1}, \dots, A_k^{-1}$ that make up the maximal multiplex and for every $i \in T_k$ we, for all $S_i \subseteq T_k$ with $|S_i| = i$, produce all possible orientations of the multiplex through $(\bigcup_{s \in S_i} A_s) \cup (\bigcup_{s \in T_k \setminus S_i} A_s^{-1})$. Note that not all orientations produced in this fashion are transitive, but every transitive orientation of any multiplex can be constructed in this fashion due to *Theorem 2.5(i)*. For each transitive orientation of such a maximal multiplex that is produced, we calculate v_i's out-degree and thus find the transitive orientation in which this out-degree is minimum. All edges of desirable transitive orientations of maximal multiplices produced are appended to an initially empty list which by the end of this process will contain edges that make up a transitive orientation $\vec{F}$ of G in which v_i has minimum out-degree.

Our algorithm ends by checking whether v_i's out-degree in $\vec{F}$ is 0 or not. If it is, it prints a message that the mere addition of the edge vv_i to G produces a comparability graph and then prints $\vec{F}$, otherwise it prints a message that v needs to also be connected to all nodes in $S = \{x \in V \mid \overrightarrow{v_i x} \in \vec{F}\}$ for the result to be a comparability graph and then prints $\vec{F}$.

A pseudocode version of the algorithm is provided below.

Algorithm 2.1 Algorithm-Multiplices

Require: $G(V, E)$ a permutation graph, $v_i \in V$

enumerate all cliques of G
enumerate the implication/color classes of G
for c **in** cliques of G **do**
    if c is a simplex **do**
        append c to a list named $simplices$
for c **in** $simplices$ **do**
    if c is not maximal **do**
        delete c **from** $simplices$
$multiplices \leftarrow$ the multiplices generated by the simplices in $simplices$
delete all duplicates **from** $multiplices$
for m **in** $multiplices$ **do**
        if m contains v_i **do**
            compute a transitive orientation of m in which v_i has minimum out degree
        else do
            compute a transitive orientation of m
combine all these transitive orientations into a transitive orientation F of G
print v_i's out-degree in F and the nodes in $\{x \in V \mid \overrightarrow{v_i x} \in F\}$

Obviously, this is a rather high-level presentation of the algorithm's steps, so in the next and final section of this chapter the focus will be placed on our code implementation and all details that were left unexplored will be examined there. However, before moving on with that, we need to provide formal proof that *Algorithm-Multiplices* is indeed guaranteed to provide a correct answer to our question and analyze its time complexity.

Proposition 2.12. *When given a comparability graph G and one of its nodes v_i as input, Algorithm-Multiplices will find a transitive orientation $\vec{F}$ of G in which v_i's out-degree is the smallest possible amongst all other transitive orientations of G.*

Proof. It is true that $\vec{F}$ will be a transitive orientation of G because $\vec{F}$ is the disjoint union of transitive orientations of the unique maximal multiplices of G (*Theorem 2.5(ii)*). Let $\overrightarrow{F_1}, \dots, \overrightarrow{F_k}$ be the transitive orientations whose disjoint union produces $\vec{F}$ and $M_1, \dots, M_k$ be the respective unique maximal multiplices of G. Then, since v_i has minimum out-degree in $\overrightarrow{F_j}, j \in \{t \in T_k \mid \exists x \in V: v_i x \in M_t\}$, there is no way to construct a transitive orientation of G by combining transitive orientations of G's unique maximal multiplices in which v_i has a smaller out-degree than its one in $\vec{F}$. And, due to *Theorem 2.5(i)*, we have that each transitive orientation's of G restriction to the edges of any of G's maximal multiplices is a transitive orientation of that multiplex, so every transitive orientation of G is a disjoint union of transitive orientations of all unique maximal multiplices of G, therefore the out-degree of v_i in $\vec{F}$ is indeed the smallest possible amongst all transitive orientations of G. □

When it comes to the algorithm's complexity, its first step, i.e. the computation of its cliques, is the most computationally expensive. Since a graph can have up to $3^{n/3}$ maximal cliques, an algorithm like the one provided by Bron and Kerbosch (with a time complexity of $O(3^{n/3})$) is optimal for enumerating G's maximal cliques [10]. From these maximal cliques, we can derive the remaining of the graph's cliques (by taking any maximal clique's induced subgraph). When it comes to the computation of G's implication/color classes, this task can be completed in $O(|E|^2)$ time since for each $\overrightarrow{ab} \in E$, we could examine whether any other $\overrightarrow{xy} \in E$ satisfies $\overrightarrow{ab}\ \Gamma\ \overrightarrow{xy}$ in order to add $\overrightarrow{xy}$ to $\overrightarrow{ab}$'s implication class. Deciding which cliques are indeed simplices can be done in $O(q_c(e_c)^2)$ where q_c is the number of cliques in G (bounded by 2^n which is the number of subsets of a set with n elements [1]) and e_c is the number of nodes in the largest clique of G. This is the case because such a clique has $e_c(e_c - 1)/2$ edges [11] and of course these are no more than this many edges in any other clique of G. The discarding of all non-maximal simplices can be done in $O((q_s)^2)$ time where q_s is the number of simplices in G, since we can compare two simplices and discard one if it is properly contained in the other, and once this is done we can generate the maximal multiplices in linear time for

each maximal simplex, for we only need to combine the color classes that make up each maximal simplex. Just like the simplices, the discarding of duplicate maximal multiplices that may have occurred can be done in $O((q_m)^2)$ time, where q_m are the maximal multiplices generated. The computation of a proper transitive orientation for each maximal multiplex requires $O(k^3 2^h)$ time, where k is the number of edges in the multiplex and h is th number of transitive orientations that comprise it. That is the case because 2^h represents the ways in which we could combine transitive orientations of the h implication classes that comprise a multiplex, and for each such combination, all pairs of the k edges of that multiplex, if they are of the type $\overrightarrow{ab}$ and $\overrightarrow{bc}$, would need to checked as to whether $\overrightarrow{ac}$ belongs to the orientation of the multiplex we produced. Uniting the transitive orientations that are enumerated during this process into a transitive orientation $\vec{F}$ of G takes linear time with respect to the number of unique maximal multiplices of G, and outputting the results takes $O(|\vec{F}|)$ time since we only need to know which edges of $\vec{F}$ are of the type $\overrightarrow{v_i x}$ for some x in V.

2.3 Comments and Explanations on Code

The code implementations for this thesis are all done in Python 3.13.0 and the graphs are created and handled through the *networkx* module exclusively due to it including a plethora of methods that directly correspond to the processes used in our algorithm.

Initially, the user shall enter the number k of undirected edges that make up G and then input the edges one by one by providing both endpoints in a *while* loop that is active until k edges have been given. For an edge ab the user only needs to enter $\overrightarrow{ab}$ or $\overrightarrow{ba}$ but not both.

Matters like verifying whether the user did not include both directed versions of an undirected edge, or whether the user has or has not included an invalid edge (e.g. $aa, a \in V$) or, finally, verifying if the provided graph is indeed comparability are considered beyond the interests of this thesis, so such checks have not been implemented. Each edge along with its inverse is appended upon input to a list named E and these edges indirectly indicate G's node set V, which however never gets a variable dedicated to it in our code since we can always access the node set through the *nodes()* method in networkx.

Appending $\overrightarrow{ba}$ to E for every $\overrightarrow{ab}$ that the user inputs is not only done for the code to be consistent with the theory upon which it is based (indeed, it would be possible to execute all of our algorithms steps without performing this step), but it is chosen as our approach due to it simplifying our future actions as will soon be demonstrated.

Now that E has the desired form, we are finally able to define our graph G as a networkx graph which gets all edges of E added to it via *add_edges_from*, a networkx method that adds to a networkx graph the edges present in the argument the method takes. At this point the user is required to choose v_i, the node to which the new node v is going to be connected to. Again, v_i (represented in the code by the variable vi) obviously needs to be in V for this input and follow-up execution to be meaningful but the validity of the user's input is not checked.

After v_i is given, all the required user inputs have been gathered and the implementation of our algorithm begins. We immediately save all cliques of G in a list that we call $G\_cliques$ by using the networkx method *enumerate_all_cliques.* This method returns a generator of lists of nodes that form a clique in G (we transform the generator into a list for convenience reasons) where the cliques appear in an ascending order of size. After obtaining G's cliques in this form, we proceed to print them out.

In the following section of code, we perform the essential process of calculating all color and implication classes of G. For the former, we initialize an empty list CC, an auxiliary empty list $temp\_cc$, and a list $temp\_E$ to temporarily hold the edges of E left to be explored at any point of the process, initially identical to E. While there are still edges left in $temp\_E$, we randomly choose r, an edge of $temp\_E$, through the *choice* method of the *random* module which is immediately appended to $temp\_cc$ while r and its inverse are removed from $temp\_E$. The edge r is going to be the edge whose color/implication class $temp\_cc$ we will enumerate so the aforementioned removal is necessary to avoid choosing r or $(r[1], r[0])$ again and possibly recalculating the same color/implication classes more than once. Then, a double *for* loop begins where for each i in $temp\_cc$ we check whether there is any edge j in $temp\_E$, i.e. G's edges that have not yet been added to a color/implication class, that is directly forced by i, i.e. $i \; \Gamma \; j$. This is precisely what is expressed in the *if* statement in line 35 (Figure 2.1). If so, we add j to $temp\_cc$ and remove its inverse from $temp\_E$. A couple of things need to be made clear at this point. Firstly, since $temp\_cc$ is potentially getting edges added to it during the execution of this *for* loop, it is guaranteed that after the completion of this double loop, $temp\_cc$ will not only contain the edges that r directly forces, but its whole implication class. Also, the removal of $(j[1], j[0])$ from $temp\_E$ in the case where j gets appended to $temp\_cc$ is done not to avoid the risk of $(j[1], j[0])$ also getting added to $temp\_cc$ since such a thing would be impossible due to G being a comparability graph [4]. Instead, this is done to avoid calculating a color/implication class that includes $(j[1], j[0])$ because the latter is going to be

the inverse of the implication class that is enumerated in $temp\_cc$ once the double *for* loop finishes and the former, meaning the color class, is going to be the union of these two implication classes. After finding all directed edges j of $temp\_E$ that a particular i of $temp\_cc$ directly forces we perform the command of line 38 (Figure 2.1) in order to also remove all edges that were appended to $temp\_cc$ via this process from $temp\_E$. When we finish the double *for* loop, an implication class (which practically grants us its corresponding color class and will from now on be used as such) has been enumerated and is appended to CC where eventually all color classes of G will end up. Then $temp\_cc$ is again initialized to an empty list and the *while* loop proceeds. The section of code that corresponds to CC's calculation can be viewed in Figure 2.4.

```
25  CC=[]
26  temp_cc=[]
27  temp_E=E
28  while (len(temp_E)>0):
29      r=random.choice(temp_E)
30      temp_cc.append(r)
31      temp_E.remove(r)
32      temp_E.remove((r[1],r[0]))
33      for i in temp_cc:
34          for j in temp_E:
35              if ((i[0]==j[0] and not(G.has_edge(i[1],j[1]))) or (not(G.has_edge(i[0],j[0])) and i[1]==j[1])):
36                  temp_cc.append(j)
37                  temp_E.remove((j[1],j[0]))
38          temp_E=list(set(temp_E) - set(temp_cc))
39      CC.append(temp_cc)
40      temp_cc=[]
```

Figure 2.4: The enumeration of CC.

It is true that by the end of this process CC will be a list of lists, where each of CC's list elements will be an implication class and not a color class. However, since any color class is the union of two implication classes that are each other's inverse, we accept this abuse of notation and take advantage of it directly to save all implication classes in IC in the next few lines of code. Each element of CC is an implication class so it gets appended to the initially empty list IC and then for each element of IC we save its inverse in a new initially empty list named $temp\_in\_ic$. After the execution of this loop is done, $temp\_in\_ic$ shall contain the inverse of all implication classes that were already in IC so we append these lists to IC which now contains all implication classes of G. We then proceed to print out all color and implication classes. Note that since we have committed an abuse of notation when it comes to the color classes, these do not contain lists of undirected edges, i.e. pairs of directed edges that

are each other's inverse, but a single 'representative' directed edge for each undirected edge that belongs to the color class.

What follows is the calculation of G's simplices which can be seen in Figure 2.5. To this end, we initialize an empty list named $simplices$, which eventually will hold only the maximal simplices of G, and enter the following loop: For each $clique$ of G, if $clique$ contains more than one node (thus containing at least one edge and therefore being eligible to be a simplex), we initialize a counter c equal to 0 and an empty list s which gets all indices within the range of CC's length appended to it in ascending order (i.e. if CC contains four color classes then $s = [0,1,2,3]$, if it contains two color classes then $s = [0,1]$ etc.). The reason why we need s is that we now enter a nested double *for* loop that goes through pairs of nodes of $clique$ and tries to track down the exact color class their edge can be found in. The index of that color class in CC is deleted from s because if we were to locate another edge of $clique$ in the same color class that would mean that clique would not be a simplex of G, and we increase c by 1. After this double *for* loop ends, we check whether c is equal to $len(clique) \cdot (len(clique) - 1)/2$, which is the number of edges in a clique made up of $len(clique)$ nodes [11]. If this is true, then and only then, due to us removing from s any index where a certain edge was found at in CC, $clique$ is a simplex, so it gets appended to $simplices$. Before discovering which of these simplices are in fact maximal, we print them out just so that the user can have a better grasp of the simplices in the graph that they have input and maybe even compare these simplices with the maximal ones that will be printed upon their calculation.

```
60  simplices=[]
61  for clique in G_cliques:
62      if (len(clique)>1):
63          c=0
64          s=[]
65          for i in range(len(CC)):
66              s.append(i)
67          for v1 in clique:
68              for v2 in clique:
69                  if (v1!=v2):
70                      for index in s:
71                          if (v1,v2) in CC[index]:
72                              s.remove(index)
73                              c+=1
74          if (c==len(clique)*(len(clique)-1)/2):
75              simplices.append(clique)
```

Figure 2.5: $simplices$ calculation

To obtain a list of only maximal simplices we are going to be evaluating which elements of $simplices$ are non-maximal and removing them from the list. Instead of removing a simplex in $simplices$ upon realizing that it is not maximal, we compute and save all indices i for which

$simplices[i]$ is non-maximal and then remove these elements from $simplices$. After initializing an empty list named $indices\_to\_delete$ which, as its name suggests, will eventually host all indices whose corresponding element in $simplices$ will be removed, we begin, in line 81, a double *for* loop that runs through two indices i and j of $simplices$ with j being larger than i. Since $G\_cliques$ holds G's cliques in an ascending order of size and $simplices$ has occurred by handling $G\_cliques$' elements in the order that they appear, $simplices[j]$ has at least as many nodes as $simplices[i]$ in it. We therefore initialize a Boolean variable named $flag$ as $True$ and check whether there exists a node in $simplices[i]$ that is not in $simplices[j]$. If this is the case $flag$'s value is changed to $False$ and the loop in which we perform this test is broken, for the discovery of such a node means that $simplices[j]$ does not, speaking in set theory terms, properly include $simplices[i]$ and thus $simplices[j]$ is not a simplex that guarantees that $simplices[i]$ is non-maximal. If, on the other hand, we find no node of $simplices[i]$ that is not also in $simplices[j]$, $flag$ will continue to be $True$ which will mean that $simplices[i]$ is non-maximal and therefore shall be deleted from $simplices$. For this to be done after this *for* loop is over, we append i to $indices\_to\_delete$ and break the inmost loop. The breaking of the inmost loop is used because once we know that $simplices[i]$ is non-maximal, we do not need to compare it to any other simplex x that lies to the right of $simplices[j]$ in $simplices$ to see if it is properly contained in x. If $simplices[i]$ is not properly contained in $simplices[j]$ $\forall j \in \{i+1, \dots, len(simplices)-1\}$ then it is maximal and will not be deleted from $simplices$. Once this double *for* loop is finished and $indexes\_to\_delete$ only holds the indices of $simplices$ where non-maximal simplices lie, we remove these simplices in the lines 92-96, where the variable k allows us to circumvent the changing of indexing that occurs within a list when elements are removed from it. After this section of the code is finished and $simplices$ only contains all of G's maximal simplices, we proceed to print its elements. The code section in which $indexes\_to\_delete$ is calculated is presented in Figure 2.6.

```
80    indexes_to_delete=[]
81    for i in range(len(simplices)):
82        for j in range(i+1,len(simplices)):
83            flag=True
84            for vertex in simplices[i]:
85                if vertex not in simplices[j]:
86                    flag=False
87                    break
88            if flag:
89                indexes_to_delete.append(i)
90                break
```

Figure 2.6: Computing $indices\_to\_delete$

We now possess all the tools we need to finally calculate all maximal multiplices of G, these tools being CC and $simplices$. After the initialization of $multiplices$ as an empty list which will, at the end of this section of code, contain all unique maximal multiplices of G, we begin a *for* loop that goes through each $maxSimplex$ of $simplices$, in which we will compute the maximal multiplex that $maxSimplex$ generates. To this end, we initialize, right after the beginning of the *for* loop, an empty list $temp\_multiplex$ which will eventually contain the maximal multiplex we want to enumerate. Then, a double *for* loop begins where for each pair of nodes $v1$ and $v2$ of $maxSimplex$, when $v1 \neq v2$ we look through each color class cc in CC to see if the edge $v1v2$ can be found in cc. A few things must be noted here. First, the reason why we do this is because the maximal multiplex that is generated by a maximal simplex is nothing more than a combination of all the edges in the different color classes to which the edges of the maximal simplex belong. Secondly, we check whether $v1$ and $v2$ are equal in line 105 just because if they are, then an edge between them does not exist in G and therefore will never be found in any element of CC, thus rendering the search for a color class that contains such an edge futile. However, for the same reasons, the code would work and still produce results even if this check was not implemented. Last but not least, this double *for* loop is constructed in a way that if x, y are two nodes in some $maxSimplex$ of $simplices$, then both $\overrightarrow{xy}$ and $\overrightarrow{yx}$ are searched for in G's color classes, which is necessary since we constructed the elements of CC in a way that only includes one directed version of every edge they contain. Continuing, when the edge $\overrightarrow{v1v2}$ is traced in some cc of CC then we append cc to $temp\_multiplex$. The reason why we append cc itself to $temp\_multiplex$ and not its edges one by one is going to be explained later. After the ending of this double *for* loop where the color class in which each of $maxSimplex$'s edges belongs has been appended to

$temp\_multiplex$, we append $temp\_multiplex$ itself to $multiplices$. Figure 2.7 contains the section of code we just described.

```
100   multiplices=[]
101   for maxSimplex in simplices:
102       temp_multiplex=[]
103       for v1 in maxSimplex:
104           for v2 in maxSimplex:
105               if (v1!=v2):
106                   for cc in CC:
107                       if (v1,v2) in cc:
108                           temp_multiplex.append(cc)
109       multiplices.append(temp_multiplex)
```

Figure 2.7: G's maximal multiplices

After this is done for every element of $simplices$, we will have calculated all maximal multiplices of G, but this process may have produced some duplicates. To delete these duplicates we again initialize $indexes\_to\_delete$ as an empty list (we use the same variable name as before since these lists are used to perform essentially the same task and the contents that $indexes\_to\_delete$ holds are useless after the removal of all non-maximal simplices from $simplices$). This time, checking whether a certain maximal multiplex shall be deleted from $multiplices$ is not such a delicate matter. In the double *for* loop implemented between lines 111 and 115 of the code, if two multiplices are equal then the index of the one that appears first in $multiplices$ is appended to $indexes\_to\_delete$. Once the appending is done the inmost loop is broken, not only due to the need for the multiplex in question to be deleted now being established, but also because we do not wish to append the same index to $indexes\_to\_delete$ again, which would happen in the event that the multiplex in question appears three or more times in $multiplices$. When this double *for* loop is finished we remove duplicates in $multiplices$ in the exact same manner that we removed non-maximal simplices from $simplices$, leaving $multiplices$ to contain all unique maximal multiplices of G, which we print out.

Before we move on to the most essential and potentially complicated section of the algorithm's implementation, we need to examine the issue of why, instead of following the strict definition of a (maximal) multiplex and making each list in $multiplices$ contain edges of a (maximal) multiplex, we structure each element of $multiplices$ as a list that contains the color classes (which are lists themselves) that comprise the multiplex. The reason for that is

quite simple and is related to the relative convenience that this representation of a multiplex provides regarding generating possible orientations of said multiplex, compared to a representation faithful to the definition. Every transitive orientation of a (maximal) multiplex can be obtained, as stated in the previous section, by considering the implication classes $A_1, \dots, A_k, A_1^{-1}, \dots, A_k^{-1}$ that make up the (maximal) multiplex and for every $i \in T_k$ and all $S_i \subseteq T_k$ with $|S_i| = i$ and producing all possible transitive orientations of the multiplex through $(\bigcup_{s \in S_i} A_s) \cup (\bigcup_{s \in T_k \setminus S_i} A_s^{-1})$. Of course, not every orientation of the multiplex produced via this method is transitive so each of the orientations' transitive statuses will have to be verified. But this is a very easy to implement idea that will allow us to efficiently compute all transitive orientations of a maximal multiplex, that requires that we not only consider the multiplices as collections of color classes, but also that we only handle them as such. Hence, we are not only allowed to express multiplices as lists made up of the color classes (that are also expressed as lists) in which their edges belong, but we are also required to do so if we want to find all transitive orientations of some maximal multiplices through this convenient method.

Now that the issue of how we save the maximal multiplices of G has been resolved, we can move on to the explanation of the final section of our code implementation for comparability graphs, the one in which we will determine the minimum number of edges incident on v that shall be added to $G + v_i v$ in order for the result to be a comparability graph. We initially define F as an empty list which will eventually hold a transitive orientation of G in which v_i will have its minimum out-degree, thus being evidence of not only how many edges we shall add to $G + v_i v$, but also dictating what these edges are according to *Lemma 2.2.* We then enter a *for* loop that goes through each maximal multiplex (represented by the variable $multiplex$) in $multiplices$, aiming to find an appropriate transitive orientation of $multiplex$ depending on whether $multiplex$ contains an edge in which v_i is an endpoint or not. We perform this check immediately once the *for* loop begins by initializing a variable c as 0, planning to change this variable's value if $multiplex$ contains v_i and thus to let c indicate whether v_i is or is not included in $multiplex$. This is done in the double *for* loop that directly succeeds c's initialization, where we look through each color/implication class ic of $multiplex$ and through each $edge$ in ic to determine whether $edge[0]$ or $edge[1]$ are equal to v_i. If this happens to be the case at any point, c increases by 1 and the double *for* loop is broken. If, on the other hand, at no point in the double *for* loop's execution such a condition is satisfied, c's value is never altered. So, after the normal termination or the breaking of the double *for* loop, c's value is 0 if and only if $multiplex$ does not contain v_i and 1 if and only if it contains v_i.

Now what naturally follows is an *if-else* statement, where we perform different tasks to calculate a transitive orientation of $multiplex$ depending on $c's$ value. We begin by considering the case of $c == 0$, which means, as stated above, that $multiplex$ does not contain v_i. In this case, we have established that what we need is just any transitive orientation of $multiplex$. So, it follows from the arguments given in the algorithm's initial description and the previous paragraph that we now shall combine the edges of the implication classes that make up $multiplex$ as they are or inverted in all possible ways for a transitive orientation of it to occur. This is done by using the *itertools* module in Python, and specifically its *product* method to calculate the cartesian product of $\{0,1\}$ with itself as many times as the color classes that make up $multiplex$, i.e. $\{0,1\}^{len(multiplex)}$. Since this method returns a generator but we would rather use a list, we initialize a variable ic_index_combos in line 137 to obtain the desired cartesian product in list form. What ic_index_combos allows us to do is that each of its elements can be used to dictate possible ways of combining proper or inverted implication classes that make up $multiplex$ in order to produce an orientation of its edges in the following way: If the i-th element in an element of ic_index_combos is 0 then we can use all directed edges of the i-th color/implication class in $multiplex$ exactly as they appear for our orientation, whereas if it is 1, we can invert all the directed edges in the same color/implication class before we add them to our orientation. By having access to $\{0,1\}^{len(multiplex)}$'s elements we can produce all orientations of $multiplex$ possible in this manner and then check if they indeed are transitive.

We implement this methodology in the following way: Firstly, we initialize a Boolean variable named $transitiveOrientation$ as $False$. This variable's truth value will change to $True$ once we find a transitive orientation of $multiplex$ and then we will be in a position to add this transitive orientation's edges to F and move on to the next element of $multiplices$ if it exists. Hence, we begin a *while* loop which will go on for as long as $transitiveOrientation$'s truth value is $False$. To find a transitive orientation of $multiplex$, as stated, we will go through the elements of ic_index_combos and let each of them dictate how we combine proper and inverted versions of entire color/implication classes within $multiplex$. More specifically, we begin a *for* loop where for each $choice$ in ic_index_combos we initialize an empty list $temp\_F$ in which we gradually construct the orientation we produce before examining if it is transitive, and also a variable i equal to 0 which is going to be used as the index through which all of $multiplex$'s elements will be accessed. Everything is now ready

for a *for* loop to start (in line 143), in which we go through every $index$ in $choice$ and appending every edge of $multiplex[i]$, i.e. the $(i+1)$-th color/implication class that makes up $multiplex$, to $temp\_F$ if $index$'s value is 0, but appending the inverse of every edge in $multiplex[i]$ to $temp\_F$ if $index$ is 1. In either case, we increase i by 1 after appending the appropriate edges to $temp\_F$. After this *for* loop ends, $temp\_F$ is an orientation of the edges of $multiplex$ and we shall now test if it is transitive. To this end, we initialize another boolean variable named $orientationVerifier$ as $True$, planning to change its truth value to $False$ upon realizing that $temp\_F$ is not transitive. Since a transitive orientation's definition demands that the transitive property holds for every pair of edges of the type $\overrightarrow{ab}, \overrightarrow{bc}$ in the orientation, we can consider $temp\_F$ to be transitive until proven not to be, and the only way to prove this is to find a pair of edges $\overrightarrow{ab}, \overrightarrow{bc}$ of $temp\_F$ such that $\overrightarrow{ac}$ is not in $temp\_F$. The double *for* loop that begins in line 153 is designed to do just that. By going through every possible pair of edges in $temp\_F$, we can, whenever we find a pair in which the tail of the first edge is identical to the head of the second edge, check whether the head of the first edge and the tail of the second one are also connected by a directed edge in $temp\_F$ in this manner. If we find a single pair for which this is not the case, we turn $orientationVerifier$'s truth value to $False$ and break the double *for* loop. After this double *for* loop ends, either normally or by being broken, if $orientationVerifier$ is $True$ then we have found a transitive orientation of $multiplex$, namely $temp\_F$, so we change $transitiveOrientation$ to $True$ so that the *while* loop that we are in will terminate and add all of $temp\_F$'s edges to F. If $orientationVerifier$ is $False$ then we go back to the beginning of the *while* loop and continue to search for a transitive orientation of $multiplex$. The code that corresponds to the case of c being equal to 1 is provided in Figure 2.8.

```
135     if (c==0):
136         ic_index_combos=list(itertools.product([0,1],repeat=len(multiplex)))
137         transitiveOrientation=False
138         while (transitiveOrientation==False):
139             for choice in ic_index_combos:
140                 temp_F=[]
141                 i=0
142                 for index in choice:
143                     if (index==0):
144                         for edge in multiplex[i]:
145                             temp_F.append(edge)
146                         i+=1
147                     else:
148                         for edge in multiplex[i]:
149                             temp_F.append((edge[1],edge[0]))
150                         i+=1
151             orientationVerifier=True
152             for edge_1 in temp_F:
153                 for edge_2 in temp_F:
154                     if (edge_1[1]==edge_2[0]):
155                         if ((edge_1[0],edge_2[1]) not in temp_F):
156                             orientationVerifier=False
157                             break
158                 if (not orientationVerifier):
159                     break
160             if orientationVerifier:
161                 transitiveOrientation=True
162                 for edge in temp_F:
163                     F.append(edge)
```

Figure 2.8: A transitive orientation for multiplices that do not contain v_i

Let us now examine what happens when $c == 1$, i.e. when $multiplex$ contains an edge where v_i is an endpoint, and how our handling of such multiplices differs from the other case we examined. We begin by initializing ic_index_combos exactly like the other case and then we also initialize max as 0, which is a variable that will hold, at any point of the execution, the maximum in-degree that v_i has had in all of the transitive orientations of $multiplex$ that we will have examined up until that point. Since we, in this case, do not just want to calculate a transitive orientation of $multiplex$ but rather locate a transitive orientation of $multiplex$ in which v_i has minimum out-degree (or, equivalently, maximum in-degree), we do not enter a *while* loop that stops upon the discovery of a transitive orientation of $multiplex$ but on the contrary, we enter a *for* loop that goes through all elements of ic_index_combos, similar to the one we implemented in the case where $c == 0$, but with a few key differences. The code inside this *for* loop for the two cases is identical up until the point where a transitive orientation of $multiplex$ is discovered. There, we initialize $inDegree$ as 0, which is a variable in which v_i's in-degree in $temp\_F$, which is calculated in the following two-line *for* loop, is saved. $inDegree$ is then compared with max, and if the former is no less than the latter we change max's value to $inDegree$ and let $best\_F$, a new variable which saves the best orientation we

have so far found in terms of v_i's in-degree, be equal to $temp\_F$. After the *for* loop that begins in line 168 is over, we append the edges of $best\_F$ to F. The differences between this case and the one in which c is equal to 0 can be viewed in detail in Figure 2.9.

```
164     else:
165         ic_index_combos=list(itertools.product([0,1],repeat=len(multiplex)))
166         max=0
167         for choice in ic_index_combos:
168             temp_F=[]
169             i=0
170             for index in choice:
171                     if (index==0):
172                         for edge in multiplex[i]:
173                             temp_F.append(edge)
174                         i+=1
175                     else:
176                         for edge in multiplex[i]:
177                             temp_F.append((edge[1],edge[0]))
178                         i+=1
179             orientationVerifier=True
180             for edge_1 in temp_F:
181                 for edge_2 in temp_F:
182                     if (edge_1[1]==edge_2[0]):
183                         if ((edge_1[0],edge_2[1]) not in temp_F):
184                             orientationVerifier=False
185                             break
186                 if (not orientationVerifier):
187                     break
188             if orientationVerifier:
189                 inDegree=0
190                 for edge in temp_F:
191                     if (edge[1]==vi):
192                         inDegree+=1
193                 if inDegree>=max:
194                     max=inDegree
195                     best_F=temp_F
196         for edge in best_F:
197             F.append(edge)
```

Figure 2.9: A transitive orientation of $multiplex$ in which v_i has maximum in degree

We have now fully described the process through which F becomes a transitive orientation in which v_i has maximum $inDegree$ among all other transitive orientations of G. So, as we have already proven, v_i's out-degree in F is going to be equal to the smallest possible number of edges incident on v that can be added to $G + v_i v$ for the resulting graph $G_v(V_{new}, E_v)$ to be a comparability graph. We calculate this out-degree indirectly, by first initializing an empty list $nodes\_to\_connect$, to which we append all nodes that are tails in edges of F in which v_i is the head. Now $len(nodes\_to\_connect)$ is precisely v_i's out-degree in F. We finish off the code by printing suitable messages depending on whether $len(nodes\_to\_connect)$ is 0 or not, and then printing out F for the user to see a transitive orientation of G that is evidence of how the edge additions that we suggest indeed produce a

comparability graph. Of course, the printing of F alone does not prove the optimal nature of our solution. For that the user will have to look at the theoretical results, algorithm, code implementation and explanations that we have provided. What printing F undoubtedly proves though is that we can make a comparability graph G_v by adding no more than $len(nodes\_to\_connect)$ incident on v to $G + v_i v$.

CHAPTER 3

ON PERMUTATION GRAPHS

3.1 Main Idea

Despite this chapter being dedicated entirely to our approach for solving our problem for permutation graphs, it would be wrong to say that the focus is shifted away from comparability graphs and the method that we developed for them entirely, since the inherent connection of these two graph classes unsurprisingly makes its way into our algorithm for permutation graphs.

Let us briefly restate the question. Let $G(V, E)$ be a permutation graph with $V = \{v_1, \dots, v_n\}$, let v be a vertex not in V and $i \in T_n$. If $G + v_i v(V_{new}, E_{new})$, where $V_{new} = V \cup \{v\}$ and $E_{new} = E \cup \{vv_i\}$, then we want to know if $G + v_i v$ is a permutation graph and if it is not, we need to figure out a way to turn it into one by adding to it the smallest number possible of edges incident on v.

Since an undirected graph is a permutation graph if and only if both it and its complement are comparability graphs [12], we can actually modify our method for comparability graphs to answer this question too. Let $M_1, \dots, M_k$ be the k maximal multiplices of G such that $\forall a, b \in T_k$: $M_a \cap M_b = \emptyset$ and $\bigcup_{a \in T_k} M_a = E$. Let $\{A, B\}$ be a partition of T_k such that $\forall a \in$

A $\forall y \in V: v_i y \notin M_a$ and $\forall b \in B$ $\exists y \in V: v_i y \in M_b$. Then, for every $a \in A$ we, similarly to the method for comparability graphs, calculate a transitive orientation of M_a and for every $b \in B$ calculate every single transitive orientation of M_b. Let $\{\overrightarrow{F_x}\}_{x \in T_k}$ be a family of sets where for every $x \in T_k$, $\overrightarrow{F_x}$ is the set of transitive orientations of M_x calculated in the aforementioned manner. Of course, $\forall a \in A: |\overrightarrow{F_a}| = 1$ and $\forall b \in B: |\overrightarrow{F_b}| \geq 2$ since the inverse of every transitive orientation of a multiplex is also its transitive orientation. Then $A, B_G = \overrightarrow{F_1} \times \ldots \times \overrightarrow{F_k}$ is a set of transitive orientations of G. While the set of all possible transitive orientations of G properly includes A, B_G (since $\forall a \in A: \overrightarrow{F_a}$ does not contain all transitive orientations of M_a), the latter's transitive orientations are sufficient to dictate all the possible different ways to add any amount of edges incident on v to $G + v_i v$ and produce a comparability graph through the method described in *Lemma 2.2*. Indeed, if $\vec{F}$ is a transitive orientation of G not in A, B_G, then $\exists \vec{T} \in A, B_G: \forall b \in B$ the b-th index of $\vec{F}$ and $\vec{T}$ are equal. This means that the two transitive orientations will only differ on how multiplices M_a, $a \in A_F \subseteq A$ (of course $A_F \neq \emptyset$) are oriented, but, by definition, these multiplices do not include edges incident on v_i and thus do not impact the number of edges that the process described in *Lemma 2.2* adds to $G + v_i v$.

After doing that, we can sort the elements of A, B_G in ascending order with respect to the out-degree of v_i in them. Then, we can examine the transitive orientations in this order, checking whether the complement $\overline{G_v}$ of the graph G_v that results from adding the edges that *Lemma 2.2* and each orientation $\vec{F}$ dictate is a comparability graph. This process shall continue until the first occurrence of $\overline{G_v}$ being comparability is met, because, since G_v is also a comparability graph, G_v is a permutation graph. The order in which we examine A, B_G's elements guarantees that when we first discover a certain G_v that is a permutation graph, the number of edges that was added to $G + v_i v$ for the construction of G_v is the smallest possible number of edges incident on v that can be added to $G + v_i v$ for the result to be a permutation graph.

The correctness of this process needs no further arguments for its establishment. What shall be stated though is that it also is guaranteed to produce a result, i.e. there is always a way to add edges incident on v in $G + v_i v$ resulting in a permutation graph. Let us provide a more formal version of this statement with a proof.

Proposition 3.1. Let $G(V, E)$ be a permutation graph and $v \notin V$. Then $G_v(V_{new}, E_v)$ where $V_{new} = V \cup \{v\}$, $E_v = E \cup \{vx \mid x \in V\}$ is a permutation graph.

Proof. Let $\pi = [\pi_1, \dots, \pi_n]$ be a permutation of T_n that represents G, i.e. $G \cong G[\pi]$. This suggests that $n = |V|$. Let $\pi' = [n+1, \pi_1, \dots, \pi_n]$, a permutation of T_{n+1} where $n+1$ appears first and the elements of T_n appear after it in the exact order that they appear in π. Let $f: T_n \to V$ be a bijection such that $\forall i, j \in T_n: (i-j)\left(\pi_i^{-1} - \pi_j^{-1}\right) < 0 \Leftrightarrow f(i)f(j) \in E$. The existence of f is guaranteed by the fact that G is a permutation graph represented by π. Let $f': T_{n+1} \to V_{new}$ such that $\forall x \in T_n: f'(x) = f(x)$ and $f'(n+1) = v$. Let $x \in V_{new}$. Then $x \in V$ or $x = v$. If $x \in V$ then, since f is onto, $\exists i \in T_n: f'(i) = f(i) = x$. If $x = v$ then $f'(n+1) = x$. Therefore f' is onto V_{new}. Now let $a, b \in T_{n+1}$ such that $f'(a) = f'(b)$. If $\{a, b\} \subseteq T_n$ then $f(a) = f'(a) = f'(b) = f(b)$ therefore $a = b$ because f is one-to-one. If $\{a, b\} \nsubseteq T_n$ then let us assume without loss of generality that $a \notin T_n$, therefore $a = n+1$ so $f'(a) = v = f'(b)$. But since $\forall x \in T_n: f'(x) \in V$ and $f'(n+1) = v$ we deduce that $b = n+1$, so $a = b$ which proves that f' is one-to-one. So, f' is a bijection. Now let $G[\pi']$ be the inversion graph of π' and $i, j \in T_{n+1}$ such that $ij \in E_{\pi'}$ which shall denote $G[\pi']$'s edge set. We will prove that $f'(i)f'(j) \in E_v$. If $\{i, j\} \subseteq T_n$ then, due to the fact that $ij \in E_{\pi'} \Leftrightarrow (i-j)\left(\pi'^{-1}_i - \pi'^{-1}_j\right) < 0 \Leftrightarrow (i-j)\left((\pi_i^{-1}+1) - (\pi_j^{-1}+1)\right) < 0 \Leftrightarrow (i-j)\left(\pi_i^{-1} - \pi_j^{-1}\right) < 0$, we have that $f(i)f(j) \in E \Longrightarrow f'(i)f'(j) \in E_v$. If, $\{i, j\} \nsubseteq T_n$ then let us assume without loss of generality that $i \notin T_n \Leftrightarrow i = n+1$. Then, since $f'(i) = f'(n+1) = v$ and $\forall x \in V: xv \in E_v$ we have that $f'(i)f'(j) \in E_v$. Now let $i, j \in T_{n+1}$ such that $f'(i)f'(j) \in E_v$. We will prove that $ij \in E_{\pi'}$. If $\{i, j\} \subseteq T_n$ then $f'(i)f'(j) \in E_v \Leftrightarrow f(i)f(j) \in E \Leftrightarrow (i-j)\left(\pi_i^{-1} - \pi_j^{-1}\right) < 0 \Leftrightarrow ij \in E_\pi \subseteq E_{\pi'}$. If $\{i, j\} \nsubseteq T_n$ then let us assume without loss of generality that $i \notin T_n \Leftrightarrow i = n+1$. Then, since $\pi'^{-1}_{n+1} = 1$, $\forall x \in T_{n+1}: ix \in E_{\pi'}$. This proves that G_v is isomorphic to $G[\pi']$ and thus G_v is a permutation graph. □

Before the ending of this section, which precedes a more formal description of the algorithm that we implemented, it is worth noting that *Proposition 3.1* could also be proven by showing that G_v is a comparability graph (by combining a transitive orientation of G with $\{\overrightarrow{vx} \mid x \in V\}$) and then showing that the edge set of the complement $\overline{G_v}$ of G_v is identical to the one of $\bar{G}$ which proves that $\overline{G_v}$ is a comparability graph and thus G_v is a permutation graph.

3.2 Algorithm-Permutation

In this section we will discuss the suggested algorithm for permutation graphs in more detail, provide its steps one by one and explain them. Since both *Algorithm-Permutation* and *Algorithm-Multiplices* perform the exact same steps and use the same implementation up until the point where all unique maximal multiplices of G are the only elements of $multiplices$ in *Algorithm-Multiplices*, we will only present and comment on the following steps of *Algorithm-Permutation*.

Once we reach the final point up until which these two algorithms are identical, we now go through all unique maximal multiplices of G and for each of them, we calculate just one transitive orientation of the multiplex if it does not contain an edge that is incident on v_i, otherwise we calculate all its transitive orientations, and save our results. These results are then combined in all possible ways (as described in the previous section of this chapter) to produce A, B_G, the out-degree of v_i in every transitive orientation of A, B_G is computed, and these transitive orientations are then sorted with respect to this out-degree in an ascending order. Let SF be the set of transitive orientations of A, B_G ordered using this criterion.

We then go through each orientation of SF in order and produce a comparability graph G_v out of $G + v_i v$ in the way that *Lemma 2.2* provides. We then check whether the complement $\overline{G_v}$ of G_v is a comparability graph using the TRO Algorithm [4]. If it is not, we continue with the loop but if it is, we deduce that the number of edges added by *Lemma 2.2*'s method is the smallest possible one such that G_v is a permutation graph and break the loop after printing the relevant messages.

Before moving on to an examination of the algorithm's complexity, here is a pseudocode version of the algorithm.

Algorithm 3.2 Algorithm-Permutation

Require: $G(V, E)$ a permutation graph, $v_i \in V$

1: **enumerate** all cliques of G
2: **enumerate** the implication/color classes of G
3: **for** c **in** cliques of G **do**
4: **if** c is a simplex **do**
5: **append** c to a list named $simplices$
6: **for** c **in** $simplices$ **do**
7: **if** c is not maximal **do**

8: **delete** c **from** $simplices$

9: $multiplices \leftarrow$ the multiplices generated by the simplices in $simplices$

10: **delete** all duplicates **from** $multiplices$

11: **for** m **in** $multiplices$ **do**

12: **if** m contains v_i **do**

13: **compute** a transitive orientation of m in which v_i has minimum out degree

14: **else do**

15: **compute** all transitive orientations of m

16: $all\_Fs \leftarrow$ all transitive orientations of G that result from combining these transitive orientations in every possible way

17: $SF \leftarrow$ the elements of all_Fs sorted with respect to v_i's out-degree

18: **for** F **in** SF **do**

19: **if** $\overline{G_v}$ is a comparability graph **do**

20: **print** v_i's out-degree in F and the nodes in $\{x \in V \mid \overrightarrow{v_i x} \in F\}$

21: **end**

We are only going to examine *Algorithm-Permutation*'s time complexity after it becomes different to *Algorithm-Multiplices* since we have already examined the latter algorithm's complexity in detail in the previous chapter. Even though we now require different transitive orientations of the maximal multiplices that contain v_i, the computational complexity of the *for* loop corresponding to the one of *Algorithm-Multiplices* remains the same, i.e. $O(k^3 2^h)$ for every multiplex, where k is the number of edges in the multiplex and h is th number of transitive orientations that comprise it. Computing all_Fs requires time proportionate to the number of transitive orientations that we enumerated (which are provided by *Theorem 2.11, (iv)*), while sorting its elements with respect to v_i's out-degree in SF requires $O(n \log n)$ time (e.g. if we use the Merge Sort algorithm). Finally, for each element of SF, we need to construct $\overline{G_v}$ (which requires $O(|E|)$ time for G_v's computation and then $O(n^2)$ for evaluating its complement's edges). Recognizing whether $\overline{G_v}$ is a comparability graph can be done in $O(\delta m)$ where δ is the maximum degree of a node in $\overline{G_v}$ and m is the same graph's edge set cardinality [4]. If $\overline{G_v}$ is indeed a comparability graph, outputting the result of our search requires $O(m)$ time.

Let us now present an example of the algorithm's execution. Consider $G(V, E)$ to be the graph pictured below and let v_4 be the node to which we want to connect the new node v.

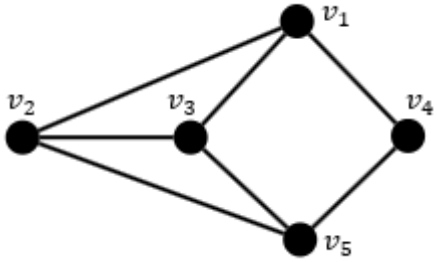

Figure 3.1: Example's permutation graph

We can observe that this graph's color classes are $C_1 = E\backslash\{v_1v_3\}$ and $C_2 = \{v_1v_3\}$. Therefore, the only simplices of G are those induced by any two neighboring nodes and are all maximal. So, the maximal multiplices of G are $M_1 = C_1$ and $M_2 = C_2$, and only M_1 involves v_4. Thus, our algorithm will compute just one of the two existent transitive orientations of M_2 and the two existent transitive orientations of M_1 and combine them into the two transitive orientations of G showcased below.

Figure 3.2: Transitive orientations in A, B_G

The node v_4 has the minimum out-degree of 0 in the leftmost of the two orientations so this orientation is going to be placed in the first position of the list in which orientations are sorted with respect to the out-degree of v_i in them. By connecting a new node v only to v_4 (since this is what the transitive orientation dictates) and producing the complement of the resulting graph we observe that it is comparability due to it having the transitive orientation of the figure below:

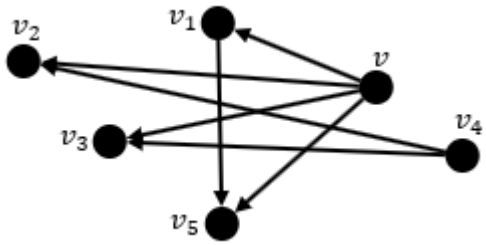

Figure 3.3: A transitive orientation of $\overline{G_v}$

Hence, we deduce that connecting v to v_4 alone is the optimal solution to our problem. Had $\overline{G_v}$ not been comparability, we would have to form another G_v be considering the other transitive orientation of Figure 3.2.

3.3 Code Implementation & Explanations

We have already discussed that *Algorithm-Multiplices* and *Algorithm-Permutation* are not only, up to a certain stage, identical as algorithms but are also implemented in the same way. Thus, the description of the implementation of *Algorithm-Permutation* will begin right after the calculation of the final form of $multiplices$ which, exactly like in the implementation of *Algorithm-Multiplices,* contains all unique maximal multiplices of G.

We now enter the phase where we want to compute a transitive orientation of every element of $multiplices$ that does not contain any edges in which v_i is an endpoint and all the transitive orientations of the elements of $multiplices$ that do contain such an edge. To save the results of our calculations, we initialize a list named $all\_multiplex\_orientations$. After the completion of the upcoming *for* loop, which is very similar to the final *for* loop of the implementation discussed in the previous chapter as we will soon see, the k-th index of $all\_multiplex\_orientations$ will contain a list of one element which will be a transitive orientation of $multiplices[k]$ if $multiplices[k]$ does not contain v_i, and a list of all possible transitive orientations of $multiplices[k]$ if $multiplices[k]$ does contain v_i. Note that transitive orientations of graphs and multiplices are also lists, so $all\_multiplex\_orientations$ will eventually be a list that contains lists that contain either one or more lists. Despite this usage of the list structure possibly seeming unnecessary to some, it will prove very practical later. The *for* loop that follows accomplishes precisely that by doing the following: we go through each $multiplex$ in $multiplices$ and initially perform the same check we implemented in our

other algorithm to find out whether $multiplex$ contains an edge that features v_i or not. In case c is equal to 0 (where c, exactly like in the implementation of *Algorithm-Multiplices*, indicates whether $multiplex$ contains v_i or not) after the check is performed, we follow the exact same process we followed for *Algorithm-Multiplices,* but with a crucial twist. Before going into the *for* loop that goes through all elements of ic_index_combos we initialize an empty list named $specific\_multiplex\_orientation$ which will eventually hold the single transitive orientation of $multiplex$ that we want to keep. After managing to verify that $temp\_F$ is a transitive orientation of $multiplex$, we append $temp\_F$ to $specific\_multiplex\_orientation$ and then append $specific\_multiplex\_orientation$ itself to $all\_multiplex\_orientations$. The way the code has now taken shape for the case of c being equal to 0 can be seen in Figure 3.4.

```
135     if (c==0):
136         ic_index_combos=list(itertools.product([0,1],repeat=len(multiplex)))
137         specific_multiplex_orientation=[]
138         transitiveOrientation=False
139         while (transitiveOrientation==False):
140             for choice in ic_index_combos:
141                 temp_F=[]
142                 i=0
143                 for index in choice:
144                     if (index==0):
145                         for edge in multiplex[i]:
146                             temp_F.append(edge)
147                         i+=1
148                     else:
149                         for edge in multiplex[i]:
150                             temp_F.append((edge[1],edge[0]))
151                         i+=1
152             orientationVerifier=True
153             for edge_1 in temp_F:
154                 for edge_2 in temp_F:
155                     if (edge_1[1]==edge_2[0]):
156                         if ((edge_1[0],edge_2[1]) not in temp_F):
157                             orientationVerifier=False
158                             break
159                 if (not orientationVerifier):
160                     break
161             if orientationVerifier:
162                 transitiveOrientation=True
163                 specific_multiplex_orientation.append(temp_F)
164                 all_multiplex_orientations.append(specific_multiplex_orientation)
```

Figure 3.4: A transitive orientation for multiplices that do not contain v_i (Permutation)

In the case where c equals 1 after the check is performed, we again enter a loop that goes through all the elements of ic_index_combos. The only difference between this loop and the one we implemented for *Algorithm-Multiplices* is that we now are not, for the moment at least, interested in the in or out degree of v_i in any of the transitive orientations of $multiplex$ that we come across (therefore we do not initialize any of the auxiliary variables that were defined in our other code, namely $max, inDegree$ and $best\_F$). Our only concern

is to append every $temp\_F$ that is a transitive orientation of $multiplex$ to $specific\_multiplex\_orientations$. After we have gone through all elements of ic_index_combos we will have saved all transitive orientations of $multiplex$ that are relevant to our problem to $specific\_multiplex\_orientations$, and thus we can append it to $all\_multiplex\_orientations$. Figure 3.5 presents the code that corresponds to this section.

```
165     else:
166         ic_index_combos=list(itertools.product([0,1],repeat=len(multiplex)))
167         specific_multiplex_orientations=[]
168         for choice in ic_index_combos:
169             temp_F=[]
170             i=0
171             for index in choice:
172                     if (index==0):
173                         for edge in multiplex[i]:
174                             temp_F.append(edge)
175                         i+=1
176                     else:
177                         for edge in multiplex[i]:
178                             temp_F.append((edge[1],edge[0]))
179                         i+=1
180             orientationVerifier=True
181             for edge_1 in temp_F:
182                 for edge_2 in temp_F:
183                     if (edge_1[1]==edge_2[0]):
184                         if ((edge_1[0],edge_2[1]) not in temp_F):
185                             orientationVerifier=False
186                             break
187                 if (not orientationVerifier):
188                     break
189             if orientationVerifier:
190                 specific_multiplex_orientations.append(temp_F)
191         all_multiplex_orientations.append(specific_multiplex_orientations)
```

Figure 3.5: Appending all transitive orientations of $multiplex$ in a list

Since the next step of our algorithm is to calculate the out-degree of v_i in each of the transitive orientations of G that we can gather by appropriately combining the transitive orientations of $multiplices$ that we calculated, we first need to gain immediate access to these transitive orientations of G. This is implemented simply in the following code section where we take advantage of the fact that we have structured $all\_multiplex\_orientations$ in a way that the cartesian product $\prod_{k=1}^{len(all\_multiplex\_orientations)} all\_multiplex\_orientations[k-1]$ is precisely the set of transitive orientations of G that we desire. The way we structured $all\_multiplex\_orientations$ may have gifted us a convenient way to arrive at $temp\_all\_Fs$ in line 194 (pictured below in Figure 3.6), however the elements of this list are divided into the color/implication classes that comprise them. So, to extract from $temp\_all\_Fs$ a list in which each element is a list that contains edges that form a transitive orientation of G, we

execute the loop that immediately follows, producing all_Fs, a list that fulfills those desired criteria. This is realized with the help of $temp\_constructor$, an empty list initialized for every orientation in $temp\_all\_Fs$ and to which we append every directed edge of every transitive orientation of every unique maximal multiplex of G included in $temp\_all\_Fs$, before appending $temp\_constructor$ to all_Fs.

```
194  temp_all_Fs=list(itertools.product(*all_multiplex_orientations))
195  all_Fs=[]
196  for orientation in temp_all_Fs:
197      temp_constructor=[]
198      for multiplex_orientation in orientation:
199          for edge in multiplex_orientation:
200              temp_constructor.append(edge)
201      all_Fs.append(temp_constructor)
```

Figure 3.6: A, B_G calculation

Now that all_Fs holds all relevant transitive orientations of G (in other words, the transitive orientations of A, B_G) we finally compute the out-degree of v_i in every transitive orientation of all_Fs and save it in $out\_degrees$. After the computation we initialize a copy of $out\_degrees$ named $out\_degrees\_copy$ and sort $out\_degrees$'s elements in ascending order. The existence of these two lists that contain the exact same elements but one of them is ordered will prove very useful in the upcoming sorting of the elements of all_Fs.

The sorting of the elements of all_Fs occurs in the next small block of code that begins in line 213 and ends in line 219. The sorted version of all_Fs will be saved in another list named $sorted\_Fs$ which is initialized before the sorting begins. We begin the process using a double *for* loop that goes through all indices i of $out\_degrees$ for the outmost loop and j of $out\_degrees\_copy$ for the inmost loop. Our plan is to search for $out\_degrees[i]$ (which is the $(i+1)$-th largest out-degree of v_i in the orientations of A, B_G) in $out\_degrees\_copy$. The first index j such that $out\_degrees[i] == out\_degrees\_copy[j]$ is the index of all_Fs where the transitive orientation of A, B_G with the i-th smallest out-degree for v_i lies, thus we append $all\_Fs[j]$ to $sorted\_Fs$. Since there may be more than one elements of all_Fs with the same out-degree for v_i, $out\_degrees$ and $out\_degrees\_copy$ may include duplicate elements. Our wish to avoid appending the same element of all_Fs to $sorted\_Fs$ more than once is what drives our decision to change the value of an element of $out\_degrees\_copy$ to something that definitely cannot be found in $out\_degrees$ once two elements of these two out-degree lists are found to be equal. That way, whenever we examine two or more consecutive equal elements of $out\_degrees$, the indices j for which these elements are found to be equal to

$out\_degrees\_copy[j]$ always differ, resulting in all different orientations of all_Fs in which v_i has that out-degree to be eventually appended to $sorted\_Fs$. Had we not changed $out\_degrees\_copy[j]$ value to "x", or any other value guaranteed not to be equal to any element of $out\_degrees$, the same orientation of all_Fs would be appended to $sorted\_Fs$ as many or even more times than the number of orientations in all_Fs in which v_i has out-degree equal to $out\_degrees[i]$. The section that begins with the initialization of $out\_degrees$ and ends with the sorting of the elements of all_Fs in $sorted\_Fs$ can be examined in Figure 3.7.

```
202  out_degrees=[]
203  for orientation in all_Fs:
204      c=0
205      for edge in orientation:
206          if edge[0]==vi:
207              c+=1
208      out_degrees.append(c)
209  out_degrees_copy=out_degrees[:]
210  out_degrees.sort()
211
212  sorted_Fs=[]
213  for i in range(len(out_degrees)):
214      for j in range(len(out_degrees_copy)):
215          if out_degrees_copy[j]==out_degrees[i]:
216              sorted_Fs.append(all_Fs[j])
217              out_degrees_copy[j]="x"
218              break
```

Figure 3.7: Sorting the elements of all_Fs

Before entering the final section of our code implementation where we seek to provide an answer to our question, we include a small albeit significant piece of code in which we construct a list that will feature all undirected, i.e. directed in both possible directions, edges present in $K_{|V_{new}|}$. The reason why such a step is important is because the next section of code will require us to calculate the complement of the comparability graphs G_v we produce, and the edges of such complement graphs will be computed by removing the edges present in G_v from the ones in a complete graph on $|V_{new}|$ nodes. After initializing $complete\_edges$, which is the list in which $K_{|V_{new}|}$'s edges will be saved, we define a new list named new_nodes initially equal to a list that includes only G's nodes (which are the integers in $T_{|V|}$) and then we append to it $\max\{V\}+1$, which is the integer that will correspond to node v. Then for every pair i and j of non-identical nodes in new_nodes we append (i,j) to $complete\_edges$.

We are now ready to proceed to the last section of the code. As we have already suggested, we shall go through the elements of $sorted\_Fs$ in the order that they appear and for

each $orientation$ in it, examine whether $\overline{G_v}$ is a comparability graph, where G_v is the comparability graph produced by adding edges to $G + v_i v$ using *Lemma 2.2* and $orientation$. To this end, we begin a *for* loop that goes through each element called $orientation$ in $sorted\_Fs$ and immediately initialize two empty lists named new_edges and $nodes\_connected$, the former to include the undirected edges that shall be added to $G + v_i v$ in order for G_v to occur and the latter to include the nodes that will be connected to v during this process, except v_i. In the *for* loop that immediately follows we go through all edges in the orientation that is currently examined, and if one's head is v_i then the tail is added to $nodes\_connected$ and both directed versions of an edge that connects v with that tail are appended to new_edges. We then introduce $total\_edges$, a list initially identical to E, that shall represent the edges in G_v. Thus, vv_i is appended to it, as well as every edge in new_edges. Finally, we are ready to define $comp\_edges$, the list that will include the edges of $\overline{G_v}$, and we initialize it to be equal to $complete\_edges$, and then we remove from it all of its elements that can also be found in $total\_edges$. The graph G_comp that represents $\overline{G_v}$ can now be defined as a networkx graph and get its edges from $comp\_edges$.

Before producing a transitive orientation of $\overline{G_v}$ or deciding that it is not a comparability graph with the help of the TRO Algorithm, we shall initialize three auxiliary lists. The first one is $temp\_E$, which is the list which we will arbitrarily pick edges from, so it must be defined to contain exactly the edges of E. The second is F, in which we will append the edges of a transitive orientation of $\overline{G_v}$ if such an orientation exists, and $temp\_F$, which will host the implication classes enumerated during the execution of the TRO Algorithm. Both F and $temp\_E$ are initially empty. These preliminary processes that occur before the execution of the TRO Algorithm can be seen in Figure 3.8.

```
228  for orientation in sorted_Fs:
229      new_edges=[]
230      nodes_connected=[]
231      for edge in orientation:
232          if edge[0]==vi:
233              new_edges.append((max(G.nodes())+1,edge[1]))
234              new_edges.append((edge[1],max(G.nodes())+1))
235              nodes_connected.append(edge[1])
236      total_edges=E[:]
237      for edge in new_edges:
238          total_edges.append(edge)
239      total_edges.append((max(G.nodes())+1,vi))
240      total_edges.append((vi,max(G.nodes())+1))
241      comp_edges=complete_edges[:]
242      for edge in total_edges:
243          comp_edges.remove(edge)
244      G_comp=nx.Graph()
245      G_comp.add_edges_from(comp_edges)
246      temp_E=comp_edges[:]
247      F=[]
248      temp_F=[]
```

Figure 3.8: Initializations before the TRO Algorithm

The *for* loop that immediately follows is where the TRO Algorithm is executed. We use a *for* loop because we decide to always choose the first not already accounted for edge of $temp\_E$ to be the edge whose implication class we enumerate, something that the TRO Algorithm allows since the decision on which edge to choose is arbitrary. So, after some edge i in $temp\_E$ has been chosen, we need to define the graph in which we will search for i's implication class. This is dictated by the TRO Algorithm's design: when looking for an edge's implication class, we do it by having first removed from the graph that we started the algorithm with all edges that have already been enumerated in an implication class. This is precisely what we do in lines 254-255, and this will become clear later when we reach the point where the edges enumerated in some edge's implication class are removed from $temp\_E$ along with their inverse edges. Before the enumeration of i's implication class, we append i to $temp\_F$ (which, as we stated, will eventually feature i's whole implication class) and remove i from $temp\_E$ as we not only do not wish to ever re-enumerate its implication class but we are also certain that i would eventually force itself to be included in its implication class [4], which would be undesirable since we would have at least two instances of i in $temp\_F$. Then we begin the main double *for* loop inside which i's implication class is enumerated and gradually appended, edge by edge, to $temp\_F$. The outmost loop goes through all elements of $temp\_F$. Initially $temp\_F$ only contains i but as we enumerate the implication class, more and more edges will be added to it and by examining them we would trace the edges they directly force that were not directly forced by i, thus tracing the edges of $temp\_E$ that i indirectly, or eventually, forces

in G_comp_temp. The inmost loop goes through all elements of $temp\_E$, which at any point of the execution of this double *for* loop contains the edges of G_comp_temp that either do not belong to $temp\_F$ or have not yet being checked on whether they do belong to it or not. Once an edge k of $temp\_E$ is found to be directly forced by one edge j of $temp\_F$ by satisfying the Γ relation criterion, we append it to $temp\_F$ and remove it from $temp\_E$ for the same reasons why we removed i from $temp\_E$. After this double *for* loop ends the calculation of i's implication class is complete.

Where the TRO Algorithm goes next is a test of whether $temp\_F$ transitive. It is true that $temp\_F$ is transitive if and only if for every edge in $temp\_F$, its inverse is not in $temp\_F$ [13]. We implement this check by initializing a Boolean variable $flag$ as $True$ and going through each $edge$ in $temp\_F$ in a *for* loop. If, for any such $edge$, it holds that the inverse of $edge$ is in $temp\_F$, we change $flag$'s truth value to $False$ and break the *for* loop. If after the ending of said *for* loop $flag$ is still $True$ that means that $temp\_F$ is indeed transitive so we append all its edges to F and remove all the inverses of these edges from $temp\_E$ as the TRO Algorithm demands we do. After we do these tasks for all edges in $temp\_F$, we empty that list for it to hold the next implication class that we should enumerate in the potential next iteration of the loop. If, on the other hand, $flag$ is $False$ we deduce that $\overline{G_v}$ is not a comparability graph and we break the *for* loop that we are currently in, meaning that we move on to the next $orientation$ in $sorted\_Fs$. It should go without saying that we cannot use a linear time algorithm like the one presented in [14] instead of the TRO Algorithm because such an algorithm can only produce transitive orientation of $\overline{G_v}$ in linear time if it is already known that $\overline{G_v}$ is a comparability graph, which is precisely what we want to check. The code for the TRO Algorithm is presented in full in Figure 3.9.

```
    for i in temp_E:
        G_comp_temp=nx.Graph()
        G_comp_temp.add_edges_from(temp_E)
        temp_F.append(i)
        temp_E.remove(i)
        for j in temp_F:
            for k in temp_E:
                if ((j[0]==k[0] and not(G_comp_temp.has_edge(j[1],k[1]))) or (not(G_comp_temp.has_edge(j[0],k[0])) and j[1]==k[1])):
                    temp_F.append(k)
                    temp_E.remove(k)
        flag=True
        for edge in temp_F:
            if (edge[1],edge[0]) in temp_F:
                flag=False
                break
        if flag:
            for edge in temp_F:
                F.append(edge)
                if (edge[1],edge[0]) in temp_E:
                    temp_E.remove((edge[1],edge[0]))
            temp_F.clear()
        else:
            break
```

Figure 3.9: The TRO Algorithm

If, for a certain $orientation$ in $sorted\_Fs$, by the end of each enumeration of an implication class, $temp\_F$ always ends up transitive until $temp\_E$ becomes empty, $flag$ is still going to be $True$ so once we finish the *for* loop that arbitrarily picks edges from $temp\_E$, we enter the final *if* statement having in our hands a transitive orientation of $\overline{G_v}$ in the form of F. We then proceed to print out our result, which features the number of nodes we needed to connect to v (apart from v_i) in order for G_v to be a permutation graph (that number is precisely $len(nodes\_to\_connect)$), the nodes themselves and finally F, a transitive orientation of $\overline{G_v}$ which obviously proves that it is a comparability graph.

3.4 Another Approach

At this point, we would like to introduce a second method for solving the same problem for permutation graphs that is not related to the algorithm that was proposed in *Chapter 2* but, instead, only focuses on permutations that represent our graph. This new approach involves the insertion of a new number in between any two integers of any permutation that represents G and a computation of the degree of the new integer in the resulting inversion graph, which, as we will show, is exactly what we aim to minimize. The incorporation of a certain technique which we will present below is what allows for this degree computation to become more efficient and therefore make it significantly better than a brute force search.

Consider our general question as it was posed in the beginning of *Section 3.1* and let π be a permutation of $T_{|V|}$ that represents G. Let f be an isomorphism between G and $G[\pi]$ such that $f(v_i) = k$. Our algorithm is essentially going to try and "insert" every number from $k - 1/2$ down to $1/2$ (subtracting 1 in each step) in between any two elements of π that lie

to the right of π_k^{-1} and every number from $k + 1/2$ up to $|V| + 1/2$ (adding 1 in each step) in between any two elements that lie to the left of π_k^{-1}, aiming to minimize inversions of integers. After the insertion, we can turn the resulting sequence into a permutation of $T_{|V|+1}$ by increasing the inserted number by $1/2$ and all the other numbers that were larger than it by 1. The number we insert is representing v, the new node that is connected to v_i, so the minimization of inversions in the resulting permutation entails a minimization of the edges that v is connected to in G_v.

Of course, to guarantee an optimal solution to our problem in the general case we would need to complete this process for all permutations that represent G (an example that proves that this is the case will be given later), and in the cases where v_i may correspond to more than one integers of T_n through different isomorphisms between G and $G[\pi]$, we would also have to examine all these isomorphisms (again, this will be demonstrated by an example). Finally, for this to be a valid method, we need to make sure that any permutation of $T_{|V|+1}$ that represents G_v will be turned into a permutation of $T_{|V|}$ that represents G by removing the integer that corresponds to v and subtracting 1 from all integers that are larger than it. If this were not the case, then there may well have been better solutions that our algorithm would fail to detect. In the following lemma we prove the desired result.

Lemma 3.2: *Let $R(V_R, E_R)$ be a permutation graph, $v \in V_R$ and π be a permutation such that $R \cong G[\pi]$. Let $f: V_R \to T_{n+1}, n \in \mathbb{N}$ be an isomorphism between R and $G[\pi]$ such that $f(v) = k$. Let $G(V, E), |V| = n$ be the graph that results from the deletion of v from R. Let π' be the permutation that results from deleting k from π and subtracting 1 from every integer $l > k$ in π. If $\Pi = \{\pi: \pi$ is a permutation of $T_n \wedge G \cong G[\pi]\}$ then $\pi' \in \Pi$.*

Proof. Let $f': V \to T_n$ such that $f'(u) = \begin{cases} f(u), f(u) < k \\ f(u) - 1, f(u) > k \end{cases}$. We will first show that f' is a bijection. Let $x, y \in V$ such that $f'(x) = f'(y)$. If both $f(x)$ and $f(y)$ are smaller than k then $f'(x) = f'(y) \Leftrightarrow f(x) = f(y) \Leftrightarrow x = y$ because f is a bijection. Similarly, if $f(x)$ and $f(y)$ are larger than k then $f'(x) = f'(y) \Leftrightarrow f(x) - 1 = f(y) - 1 \Leftrightarrow x = y$ again due to f being a bijection. If $f(x)$ and $f(y)$ are not both larger or smaller than k, which is the only scenario left unexplored, let us suppose, without loss of generality, that $f(x) < k$ and $f(y) > k$, which, since $\{f(x), f(y)\} \subseteq T_{n+1} \subseteq \mathbb{N}$, implies that $f(x) \leq k - 1$ and $f(y) \geq k + 1$. Then $f'(x) = f'(y) \Leftrightarrow f(x) = f(y) - 1 \Leftrightarrow f(y) - f(x) = 1$ which is a contradiction because

$f(y) - f(x) \geq 2$. Therefore $f'(x)$ and $f'(y)$ cannot be equal if $(f(x) - k)(f(y) - k) < 0$. We have thus deduced that $\forall x, y \in V: f'(x) = f'(y) \Leftrightarrow x = y$ which means that f' is one-to-one. Now let $m \in T_n$. We want to show that f' is onto, or equivalently that $\exists x \in V: f'(x) = m$. We have: $T_n \subseteq T_{n+1} \wedge f: V_R \to T_{n+1} \Rightarrow \exists s \in V_R: f(x) = m$. If $m < k$, then we know that $s \neq v$ because $f(v) = k$ and f is one-to-one, therefore $s \in V$ and $f'(s) = f(s) = m$. If $m \geq k$ then, because $m + 1 \in T_{n+1}$ and f is onto, $\exists d \in V_R: f(d) = m + 1$. Again, $m + 1 \neq k$ so $d \neq v$ therefore $d \in V$ and $f'(d) = f(d) - 1 = (m + 1) - 1 = m$. So f' is onto. Now we need to show that f' is an isomorphism between G and $G[\pi']$, i.e. $\forall x, y \in V: xy \in E \Leftrightarrow f'(x)f'(y) \in E(G[\pi'])$. Since $G[\pi']$ is an inversion graph, $f'(x)f'(y) \in E(G[\pi']) \Leftrightarrow \big(f'(x) - f'(y)\big)\big(\pi'^{-1}_{f'(x)} - \pi'^{-1}_{f'(y)}\big) < 0$. Let $xy \in E$. Since $E \subseteq E_R$ (which follows from the way that G is derived from R) it holds that $\big(f(x) - f(y)\big)\big(\pi^{-1}_{f(x)} - \pi^{-1}_{f(y)}\big) < 0$. Let us suppose, without loss of generality, that $f(x) < f(y) \wedge \pi^{-1}_{f(x)} > \pi^{-1}_{f(y)}$. By the definition of f' we have that $\forall x, y \in V: f(x) < f(y) \Leftrightarrow f'(x) < f'(y)$. As for π', let $m, t \in T_{n+1} \backslash \{k\}: m < t \wedge \pi^{-1}_m > \pi^{-1}_t$ and suppose $f(x) = m$, $f(y) = t$. By removing k from π and then modifying it to get π', m and t will only be decreased by 1 if they are larger than k and their position in π' will be 1 less than their position in π only if they appear after k in π. Whatever their position in π is though, it holds that m and t are inverted in π if and only if the integers that correspond to them in π' are also inverted. But this correspondence between integers of the two permutations is exactly what f' expresses, so we have that $\forall x, y \in V: \pi^{-1}_{f(x)} > \pi^{-1}_{f(y)} \Leftrightarrow \pi'^{-1}_{f'(x)} > \pi'^{-1}_{f'(y)}$. What we have proven is that $\forall x, y \in V: f'(x)f'(y) \in E(G[\pi']) \Leftrightarrow \big(f(x) - f(y)\big)\big(\pi^{-1}_{f(x)} - \pi^{-1}_{f(y)}\big) < 0 \Leftrightarrow xy \in E_R$. So, what now needs to hold for f' to be an isomorphism between G and $G[\pi']$ is that $\forall x, y \in V: xy \in E \Leftrightarrow xy \in E_R$ which is true because $E = E_R \backslash \{ab \in E_R \mid a = v\}$. Therefore, G is a permutation graph that is represented by π', which implies that $\pi' \in \Pi$. □

To illustrate the algorithm and the technique that allows us to make it more computationally efficient than a pure exhaustive search, we are going to use an example. Let $G(V, E)$ be the following graph.

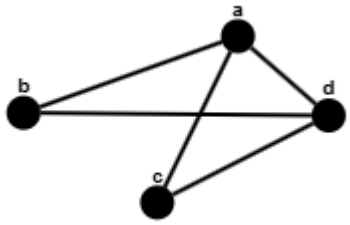

Figure 3.10: A permutation graph G

It is not hard to verify that G can be represented by $[4,2,3,1], [4,3,1,2]$ and $[3,4,2,1]$. Suppose that v_i is b and that we first want to try inserting a new number to $[4,2,3,1]$. We could construct an isomorphism f between G and $G[4,2,3,1]$ so that $b = v_i$ corresponds to either 2 or 3. Let us suppose that we map it to 2. If we do so, we must now try to insert the numbers 1.5 and 0.5 after 2 in $[4,2,3,1]$ or the numbers $2.5, 3.5$ and 4.5 before 2 in the same permutation. However, instead of computing the inversions caused by each insertion from scratch, we save time and memory through the following observation.

Table 3.1: Incomplete offsets and columns for $[4,2,3,1]$, $f(v_i) = 2$

				offset= 2		offset= 3		offset= 2
	4		**2**		3		1	
				$1.5 \rightarrow 0$		$1.5 \rightarrow 0$		$1.5 \rightarrow 0$
				$0.5 \rightarrow -1$		$0.5 \rightarrow -1$		$0.5 \rightarrow +1$
					$+1$		-1	

The table above has the numbers of the permutation $[4,2,3,1]$ in its second row, with gaps in between so that in the columns we can demonstrate the insertions that we would like to test. We have so far only completed the insertions of numbers smaller than 2 (which is written in bold to remind us that this is the integer that corresponds to v_i) and see that after inserting 1.5 (which is the largest number smaller than 2 that we insert to the permutation) exactly to the right of 2 we can compute the inversions that this insertion creates (apart from the one with 2) and call that the column's offset, as is written in the corresponding column of the table. Then, we can complete the column in such a way that if we were to add the column's

offset with the first j elements of the column, the sum would be equal to the inversions caused by the insertion of the j-th largest number. To achieve this, we need to observe that the number that corresponds to 1.5 should be 0 (since the offset is calculated with respect to this number) and then we should "map" 0.5 to 1 if 1 is to its left (since that would give exactly as many inversions as the insertion of 1.5 would plus one more new inversion due to 1 and 0.5 now being inverted) and we should map it to -1 if 1 is on its right (this is actually the case in our example and this is why the arrow next to 0.5 points to -1 in that column).

We have described how the inversions caused by each insertion in the column exactly to the right of 2 can be computed without treating each insertion as if it is not related to the others that we are interested in, i.e. the ones whose resulting inversions we have already computed. Let us now describe how we can derive the columns to this one's right. Since 3 lies exactly to the right of 2 in $[4,2,3,1]$ and 1.5 (and hence 0.5) would invert with 3 if we were to insert it to 3's right, the offset of the next column will increase by 1 and that column shall otherwise be identical to the one on its left. The $+1$ number three cells below 3 indicates precisely this column offset increase. However, when it comes to inserting after the number 1, i.e. at the end of $[4,2,3,1]$, we can see that the inversion between 1.5 and 1 that was reflected in the previous column is no longer there (therefore the column offset decreases by 1 as is indicated three cells below 1) however 0.5 now inverts with 1 so we need to demonstrate that in the new column, which shall be identical to the previous one except for the cell that corresponds to 0.5 which changes from -1 to 1. The difference between the two consecutive columns is highlighted by the green color of the sole cell which marks the difference between the two columns.

Following this train of thought, we can complete the rest of the table like this.

Table 3.2: Complete offsets and columns for $[4,2,3,1]$, $f(v_i) = 2$

offset= 1		offset= 2		offset= 2		offset= 3		offset= 2
	4		**2**		3		1	
$2.5 \to 0$		$2.5 \to 0$		$1.5 \to 0$		$1.5 \to 0$		$1.5 \to 0$
$3.5 \to +1$		$3.5 \to +1$		$0.5 \to -1$		$0.5 \to -1$		$0.5 \to +1$
$4.5 \to +1$		$4.5 \to -1$			$+1$		-1	
	-1							

To compute what the best possible insertion is for this particular permutation and integer that maps to $b = v_i$ is we would need to find, for which column and integer j, the column's offset plus the sum of the first j numbers of the column produces the minimum result. Of course, these sums shall not be calculated independently from one another. Having calculated the sum x of a column's offset plus its $(j-1)$ first elements, the calculation of the sum of the same offset plus the j first column elements shall follow immediately after x's computation and be equal to x plus the j-th element of the column. And since a sum equal to 0 is guaranteed to be an optimal solution (since it would imply that $G + v_i v$ is a permutation graph), these sums must be computed right after a column's completion to save unnecessary time of computing other columns when the optimal solution can potentially be found in the columns we have already computed. We shall maintain variables of the permutation, the integer k that corresponds to v_i, the number that we inserted and the position in which said number was inserted to, for which the minimum number of inversions (or equivalently sum of column offset plus the j first elements of said column) is achieved. Of course, if we run into a sum equal to 0 we stop the search after renewing the values of these variables. If this never happens, after examining all permutations, isomorphisms between these permutations and G, numbers and positions for possible insertion, we know that a way to turn $G + v_i v$ into a permutation graph by connecting v to as few nodes of V as possible is by connecting it to the nodes of V that correspond to the integers (of the permutation where the minimum sum was achieved) that invert with the number that we inserted in the position where the minimum sum was achieved.

Before providing pseudocode for this algorithm, we shall show why examining all permutations that represent G, as well as all possible isomorphisms between G and $G[\pi]$, is necessary to guarantee that we will find the optimal solution.

We will first tackle the need for considering multiple permutations that represent G by returning to the example through which we introduced the algorithm. It is easily verifiable by consulting Table 3.2 that the best an insertion to $[4,2,3,1]$ can achieve is the connection of v to one node of V apart from v_i for the result to be a permutation graph. However, if we consider the permutation $[3,4,2,1]$ and the fact that $b = v_i$ corresponds to the number 3, we can see that inserting 0.5 to the exact right of 3 creates no other inversions than the one

between 3 and 0.5, so $G + v_i v$ is a permutation graph, a fact that we could not detect through insertions in $[4,2,3,1]$. The only reason why this does not constitute an example of why we shall consider multiple isomorphisms between G and $G[\pi]$ is that we can observe that $b = v_i$ could also be mapped to 4 through an isomorphism, but if we were to insert 3.5 at the end of the permutation, i.e. right after 1, we would again cause no extra inversions apart from the one between 4 and 3.5, thus proving $G + v_i v$ is a permutation graph through a different isomorphism between G and $G[\pi]$.

To illustrate why all isomorphisms between G and $G[\pi]$ shall be tested we shall introduce a new example. Consider the graph displayed in Figure 3.1 and that $i = 3$, i.e. $v_i = v_3$. That graph, which we will call G, can be represented by (at least) $[4,5,2,1,3]$ and $[4,3,5,1,2]$, but we will only focus on $\pi = [4,3,5,1,2]$. We could either (through an isomorphism f between G and $G[\pi]$) map v_3 to 4 or 3. In the first mapping, the insertion of 0.5 exactly to the right of 4 proves that $G + v_i v$ is a permutation graph and no more connections between v and other nodes of V are needed. However, for the second mapping, we can observe that no insertion will produce such a result because any insertion to the right of 3 will invert with 4 (which lies in the first position of π) and every insertion to the left of 3 will invert with 1 and 2 (which lie in the penultimate and ultimate positions of π respectively).

Below, we provide a pseudocode version of the algorithm.

Algorithm 3.2 Insertion Algorithm

Require: $G(V, E)$ a permutation graph, $v_i \in V$

1: **enumerate** all permutations that represent G
2: **for** all permutations π that represent G **do**
3: **find** all isomorphisms between G and $G[\pi]$
4: **for** all π that represent G **do**
5: **for** all isomorphisms f between G and $G[\pi]$ ($f(v_i) = k$) **do**
6: **for** each possible insertion point to the right of k **do**
7: **compute** the offset, column elements and sums
8: **if** a new minimum sum is located **do**
9: **save** all parameters of the optimal insertion
10: **if** sum$= 0$ **go to 16**
11: **for** each possible insertion point to the left of k **do**

12: **compute** the offset and the column elements
13: **if** a new minimum sum is located **do**
14: **save** all parameters of the optimal insertion
15: **if** sum$= 0$ **go to 16**
16: **for** the parameters involved in the smallest sum **do**
17: **return** the optimal solution according to the parameters

When it comes to this algorithm's complexity, the most expensive steps seem to be the first two. Indeed, by *Theorem 2.11, (iv)*, G's number of transitive orientations is shown to be $\prod_{i=1}^{k}(r_i+1)!$, where r_i is the rank of a unique maximal multiplex M_i of G, and this number could be quite large for some particular graphs (and on top of that, one should consider that the different combined orientations of the implication classes that make up the multiplex shall also be checked for comprising a transitive orientation of M_i. While this check can be performed in polynomial time, it further increases the complexity). When it comes to finding all isomorphisms between G and the inversion graphs isomorphic to it, there does not seem to exist a more reliable method for the general permutation graph case than the obvious brute-force method of checking which of the $|V|!$ bijections between V and T_n are isomorphisms. After these two more expensive steps are finished, we observe that the total number of insertions that we consider for each permutation π and isomorphism f are (in the worst case where an optimal solution is not found before the computation of all columns) $k(n+1+\pi_k^{-1})+\pi_k^{-1}(n+1-k)=(k+\pi_k^{-1})(n+1)-2k\pi_k^{-1}$ where $k=f(v_i)$ and π_k^{-1} is k's position in π. The number of total insertions may be dependent on k and π_k^{-1} but these numbers are bounded by n so the insertions' number is linear with respect to n. For each insertion/column, we need $O(n)$ time for the offset and $O(1)$ for each of the other column elements if the column lies to the immediate left or right of π_k^{-1}, and we need $O(1)$ for both offset and each column element for all other columns. To compute the sum of the offset plus the j first elements of a particular column we require $O(j)$ time, with j being bounded by n.

CHAPTER 4

CONCLUDING REMARKS

Now that we have presented all our results, hopefully having succeeded in providing sufficient answers to the questions that originally motivated us to work on this thesis, it is time to provide some brief comments on our efforts and some ideas for where future research could head towards given the contributions of this thesis.

The algorithms designed for this thesis provide us with a tool to expand, potentially even ad infinitum, comparability and permutation graphs by adding one node at a time and connecting it to the fewest possible other nodes. This could prove immeasurably helpful if we, for example, wanted to model a network which, as time went on, would get nodes added to it, and its desired properties would also dictate that the graph which would model it shall be comparability or permutation (e.g. a network whose nodes should be placed in a strict order).

The chapters that preceded this one indicate directions towards which one could guide their research. On the purely mathematical front, the study of other characteristic features of these graph classes (e.g. UPOs, schemes, G-decompositions and Γ^* matroids for comparability graphs and permutation sorting for permutation graphs [4]) could potentially allow for these features' appropriation for the design of more efficient algorithms. There is also room for improvement in both our algorithm design and implementation, through potential modifications that would allow faster algorithms to partake in our design (e.g. the linear time algorithm presented in [14] instead of the TRO Algorithm) and through the use of fitting data structures upon which we could perform our desired operations in more concise code that executes

quicklier. When it comes to research that is beyond the scope of this thesis, we would like to restate the seeming lack of work on the general question we posed in the first chapter for many otherwise well-studied graph classes and encourage interested readers to try providing effective algorithms for such classes, and also note that our problem is a simpler case of the one in which the new edges that need to be added to maintain a graph's membership in class do not necessarily have to be incident on v. That problem is one of great interest too.

BIBLIOGRAPHY

[1] P.C. Tsamatos, *Θεμελιώδεις Έννοιες Μαθηματικής Ανάλυσης*, Εκδόσεις Τζιόλα, 2009.

[2] R.C. Lacher, MAD 3105. Class Lecture, Topic: "Closure of Relations," Department of Computer Science, Florida State University [Online]. Available:[Accessed Apr. 16, 2025].

[3] S. Goldberg, *Probability: An Introduction*, Dover Books on Mathematics, Courier Corporation, p.41, 1986.

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

[5] A. Mpanti, S. D. Nikolopoulos, L. Palios, "Adding a Tail in Classes of Perfect Graphs," *Algorithms*, vol. 16, no. 6, 289, 2023.

[6] P. Heggernes, F. Mancini, C. Papadopoulos, "Making arbitrary graphs transitively orientable: Minimal comparability completions," In Proc. ISAAC 2006, Kolkata, India, 2006, Springer Verlag, pp. 419-428.

[7] M. Andresen, "ON TRANSITIVE ORIENTATIONS OF $G - \hat{e}$," *Discussiones Mathematicae Graph Theory*, vol. 29, no. 1, pp. 423-467, 2009.

[8] S. Olariu, "On sources in comparability graphs, with applications," *Discrete Mathematics*, vol. 110, no. 1-3, Dec., pp. 289-292, 1992.

[9] M. Hamadé, A. Belkasri, "A NEW APPROACH TO CHARACTERISE ALL THE TRANSITIVE ORIENTATIONS FOR AN UNDIRECTED GRAPH," in *arXiv*. Cornell University, [online document], 1994. Available: arXiv, https://arxiv.org/abs/alg-geom/9411013v1 [Accessed: April 17, 2025].

[10] D. Eppstein, CS163 & CS265. Class Lecture, Topic: "Cliques and the Bron-Kerbosch Algorithm," Department of Computer Science, University of California, Irvine, Jan. 15, 2025.

[11] L. Euler, "Solutio problematis ad geometriam situs pertinentis," in *Euler Archive*, University of The Pacific, [online document], 1741, Available: Scholarly Commons, https://scholarlycommons.pacific.edu [Accessed: April 17, 2025].

[12] B. Dushnik, E.W. Miller, "Partially Ordered Sets," *American Journal of Mathematics*, vol. 63, no. 3, Jul., pp. 600-610, 1941.

[13] M.C. Golumbic, A. Trenk, *Tolerance Graphs*, Cambridge Studies in Advanced Mathematics, vol. 89, Cambridge University Press, 2004.

[14] R.M. McConnell, J.P. Spinrad, "Modular decomposition and transitive orientation," *Discrete Mathematics*, vol. 201, no. 1-3, Apr., pp. 189-241, 1999.

CV

Konstantinos Stamatis was born in June of 1999 in Ioannina. He studied in the Department of Mathematics in the University of Ioannina, in which he enrolled in October 2017 and graduated in April 2022. In October of 2022 he enrolled in the postgraduate program "Data and Computer Systems Engineering" in the Department of Computer Science & Engineering in the University of Ioannina. His research interests lie in Graph Theory.