# Simultaneous Graph Representation Problems

by

Krishnam Raju Jampani

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2011

## Abstract

Many graphs arising in practice can be represented in a concise and intuitive way that conveys their structure. For example: A planar graph can be represented in the plane with points for vertices and non-crossing curves for edges. An interval graph can be represented on the real line with intervals for vertices and intersection of intervals representing edges. The concept of "simultaneity" applies for several types of graphs: the idea is to find representations for two graphs that share some common vertices and edges, and ensure that the common vertices and edges are represented the same way. Simultaneous representation problems arise in any situation where two related graphs should be represented consistently. A main instance is for temporal relationships, where an old graph and a new graph share some common parts. Pairs of related graphs arise in many other situations. For example, two social networks that share some members; two schedules that share some events, overlap graphs of DNA fragments of two similar organisms, circuit graphs of two adjacent layers on a computer chip etc. In this thesis, we study the simultaneous representation problem for several graph classes.

For planar graphs the problem is defined as follows. Let $G_1$ and $G_2$ be two graphs sharing some vertices and edges. The simultaneous planar embedding problem asks whether there exist planar embeddings (or drawings) for $G_1$ and $G_2$ such that every vertex shared by the two graphs is mapped to the same point and every shared edge is mapped to the same curve in both embeddings. Over the last few years there has been a lot of work on simultaneous planar embeddings, which have been called 'simultaneous embeddings with fixed edges'. A major open question is whether simultaneous planarity for two graphs can be tested in polynomial time. We give a linear-time algorithm for testing the simultaneous planarity of any two graphs that share a 2-connected subgraph. Our algorithm also extends to the case of $k$ planar graphs, where each vertex [edge] is either common to all graphs or belongs to exactly one of them.

Next we introduce a new notion of simultaneity for intersection graph classes (interval graphs, chordal graphs etc.) and for comparability graphs. For interval graphs, the problem is defined as follows. Let $G_1$ and $G_2$ be two interval graphs sharing some vertices $I$ and the edges induced by $I$. $G_1$ and $G_2$ are said to be *simultaneous interval graphs* if there exist interval representations of $G_1$ and $G_2$ such that any vertex of $I$ is assigned to the same interval in both the representations. The *simultaneous representation problem* for interval graphs asks whether $G_1$ and $G_2$ are simultaneous interval graphs. The problem is defined in a similar way for other intersection graph classes.

For comparability graphs and any intersection graph class, we show that the simultaneous representation problem for the graph class is equivalent to a graph augmentation problem: given graphs $G_1$ and $G_2$, sharing vertices $I$ and the corresponding induced edges, do there exist edges $E'$ between $G_1 - I$ and $G_2 - I$ such that the graph $G_1 \cup G_2 \cup E'$ belongs to the graph class. This equivalence implies that the simultaneous representation problem is closely related to other well-studied classes in the literature, namely, sandwich graphs and probe graphs.

We give efficient algorithms for solving the simultaneous representation problem for interval graphs, chordal graphs, comparability graphs and permutation graphs. Further, our algorithms for comparability and permutation graphs solve a more general version of the problem when there are multiple graphs, any two of which share the same common graph. This version of the problem also generalizes probe graphs.

## Acknowledgements

I have been extremely fortunate to have Prof. Anna Lubiw as my supervisor. This thesis would never have been possible without her guidance and support. The weekly discussions we have had were crucial in defining and shaping the contents of the thesis. She gave me full freedom to explore research ideas and to collaborate with others on problems that are often not related to the thesis! She has also supported and encouraged me to travel to many conferences and workshops, even when I had no paper to present! I acknowledge her for painstakingly reading, correcting and greatly improving the quality of our research articles and reports, including this thesis. I also thank Anna and Jeff for kindly hosting me at their house in Boston, when I visited MIT.

I thank my thesis committee members Prof. Martin Charles Golumbic, Prof. Joseph Cheriyan, Prof. Therese Biedl and Prof. Timothy Chan. In particular, I thank them for carefully reading the thesis and providing me with several suggestions for improving the presentation. I thank Prof. Joseph Cheriyan and Dr. Ashkan Aazami with whom I have had numerous discussions and a fruitful collaboration. Working with them has been a great learning experience and has given me a new perspective about doing research. I also thank Prof. Therese Biedl and Elad Cohen with whom I had discussed several research problems.

I was very fortunate to have met Bernhard Haeupler during my trip to MIT (thanks to Anna for introducing him). His insight into PQ-trees and planarity testing algorithms was critical in obtaining the results of Chapter 8. I also thank my friend Debmalya Panigrahi for hosting me when I visited MIT on two occasions. Early in my PhD program, I was lucky to get a chance to work with Dr. Stephane Durocher. I thank Steph and Anna for the discussions, we have had and I especially thank Steph for doing most of the write-up for a research paper.

I thank all the faculty and staff of the Cheriton school of computer science for providing a stimulating experience. Numerous friends have made my stay at Waterloo worthwhile. I thank all of my friends in CS department, particularly: Aniket, Anup, Arash, Bob, Cathy, David, Elena, Hamid, Hrushikesh, Jalaj, Kamran, Maheedhar, Mustaq, Omer, Reza, Robin, Ruth, Tian, Vinayak, Sarvagya, Somit and Shahin. I would also like to thank my other friends in Waterloo, who have helped and supported me in various ways. In particular thanks to Arun, Murthy, Prakash, Prasad, Ramesh and Sukhpreet for making my stay in Waterloo more pleasant.

Special thanks to my Masters advisor Prof. Sunil Chandran, who influenced me to work in Graph Algorithms in the first place. Without him, I probably would not have worked in this area.

Finally, I would like to thank my parents, sister and brother for their never ending love and support. I owe everything I have achieved to them!

- Chapters 4 to 7 are joint work with Anna Lubiw and appeared in WADS09 and ISAAC10 conferences, published by Springer.
- Chapter 8 is joint work with Bernhard Haeupler and Anna Lubiw and appeared in ISAAC10 conference, published by Springer.

I thank Springer for allowing me to reuse my published articles for this thesis.

## Dedication

I dedicate this thesis to my parents.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

A graph is a combinatorial structure that models the pairwise relations between a set of objects. The vertices or nodes of the graph denote the set of objects and the edges represent the pairwise relations between the vertices. An edge consists of a pair of vertices that are said to be adjacent. Graphs arise naturally in numerous settings: road networks, where vertices denote cities and edges may represent how the cities are connected by highways; social networks, where vertices denote people and edges may represent the friendship relation between them; scheduling, where vertices denote events and edges represent the overlap information between events, etc.

Adjacency lists and adjacency matrices are the most common ways to represent graphs. An adjacency list consists of a list of sets, one for each vertex $v$, consisting of vertices adjacent to $v$. This representation is usually good for sparse graphs, i.e. graphs with a few edges. An adjacency matrix is an $n \times n$ matrix, where $n$ is the number of vertices, and where the $(i, j)$ entry of the matrix represents whether vertices $i$ and $j$ are adjacent. This representation is usually good for dense graphs, i.e. graphs with many edges.

Although adjacency lists and adjacency matrices are appropriate for representing general graphs they are not always the best at representing special graphs. Many graphs arising in practice can be represented in a more concise and intuitive way that conveys their structure. For example: A planar graph can be represented in the plane with points for vertices and non-crossing curves representing edges. An interval graph can be represented in the real line with intervals for vertices and intersection of intervals representing edges. Such representations are desirable for several reasons including the following:

1. They make it easier for humans to understand the graphs. For example, in graph drawing and graph visualization it is preferable to find a planar drawing or a near-planar drawing of a graph [71].

2. They make it easier for computers to store the graphs. For example, although an interval graph can be dense, it can be stored in $O(n \log n)$ bits ($n$ is the number of vertices), by computing an interval representation and storing the order of the interval end points [77].

3. They are a useful tool in designing and analyzing algorithms, for decision and optimization problems on the graph. For example, finding a maximum independent set (maximum number of vertices that are pairwise non-adjacent) is a hard problem for general graphs.

However for planar graphs, Baker's algorithm [3] can be used to obtain a polynomial-time approximation scheme (an algorithm whose solution is guaranteed to be 'close' to the optimum). Baker's algorithm [3] and the analysis critically use the recursive structure of a planar embedding. The algorithm can be used to obtain polynomial-time approximation schemes for several other NP-hard problems on planar graphs including minimum vertex cover, minimum dominating set and minimum edge dominating set.

Spinrad's book [77] on "Efficient Graph Representations" presents elegant representations for many types of graphs. It also contains algorithms that use representations for recognizing graph classes or for solving certain optimization problems on them.

In this thesis, we study the concept of "simultaneous representation" for several types of graphs. The idea is to find representations for two graphs that share some common vertices and edges, and ensure that the common vertices and edges are represented in the same way. Simultaneous representation problems arise in any situation where two related graphs should be represented consistently. A main instance is for temporal relationships, where an old graph and a new graph share some common parts. Pairs of related graphs arise in many other situations, for example, two social networks that share some members, two conference schedules that share some (plenary) talks, overlap graphs of DNA fragments of two similar organisms, floor plans of two adjacent floors sharing some rooms, circuit graphs of two adjacent layers on a computer chip, etc. In this thesis we study the simultaneous representation problem for several graph classes including planar graphs, interval graphs, chordal graphs, comparability graphs and permutation graphs. These classes of graphs are of enduring interest because of their many applications. The famous books of Roberts [74] and Golumbic [44] give an excellent introduction and present many applications of these graph classes. We now explain how the simultaneous representation problem is defined on each of these classes starting with planar graphs.

Planar graphs are one of the most fundamental class of graphs and their study has generated many deep results [70, 69, 71]. Many classical and contemporary problems and theorems in graph theory involve planar graphs. For example Euler's formula connecting the number of vertices, faces and edges of a convex polyhedron is one of the earliest results on planar graphs (the graph formed by the edges and vertices of three-dimensional convex polyhedron is planar). The famous four-color theorem states that a planar graph can be colored with 4 colors, i.e. each vertex can be assigned one of 4 colors in such a way that no two adjacent vertices have the same color. Several geometric representations are known for planar graphs. Fáry's theorem [54] states that any planar graph can be drawn with non-crossing straight edges. The *circle packing theorem* [60] says that a graph is planar if and only if each vertex can be represented with a disk on the plane so that two vertices are adjacent in the graph if and only if their corresponding disks touch each other. Planar graphs arise in many settings including road/rail networks, maps, VLSI circuits, etc.

The simultaneous representation problem for planar graphs is defined as follows. Two planar graphs $G_1$ and $G_2$ sharing some vertices and edges are said to have *simultaneous planar embeddings* or to be *simultaneously planar* if they have planar drawings, such that a shared vertex [edge] is represented by the same point [curve] in both drawings. Note that edges belonging to a graph are not allowed to cross in the drawing. Figure 1.1 shows two graphs that are simultaneously planar. Simultaneous planar representations are desirable for representing pairs of related planar graphs and have applications in graph visualization and graph drawing, for example, in representing two related VLSI circuits that share some components or in displaying two local portions of a large graph that share some vertices. The notion of simultaneous representation or drawing is also desirable for representing two related non-planar graphs, though finding such a representation is more difficult for them.

The problem of finding simultaneous planar embeddings has been introduced by Braß et al. [13] in 2003 and since then the problem and its variants have been studied extensively in the literature. A major open question in the literature is whether simultaneous planarity for two graphs can be tested in polynomial time. One of our main results in this thesis is a linear-time algorithm for testing the simultaneous planarity of any two graphs that share a 2-connected subgraph. Our algorithm also extends to the case of $k$ planar graphs, where each vertex [edge] is either common to all graphs or belongs to exactly one of them.

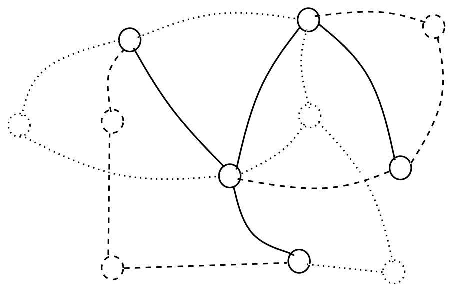


Figure 1.1: Two graphs with a simultaneous planar embedding. Solid nodes and edges are common to both graphs, dashed nodes and edges belong to the first graph, dotted nodes and edges belong to the second graph. Note that edges of different graphs are allowed to cross.

Next we introduce the notion of simultaneity for intersection graph classes and comparability graphs. An intersection graph is a graph where each vertex can be associated with a set in such a way that two vertices are adjacent in the graph if and only if their associated sets overlap. The intersection representation of such a graph is defined as the sets associated with all the vertices and the graph is called as the intersection graph of these sets. Any graph can be viewed as an intersection graph by associating each vertex with the set of edges incident to it. In this case the intersection representation is the same as the adjacency list representation of the graph. However when the sets associated with vertices are restricted to be special, e.g. intervals, disks, line segments, subtrees etc., we get interesting special graphs.

Note that interval graphs are defined as the intersection graphs of intervals on the real line. The earliest known application of interval graphs is in showing that genes are arranged in a linear fashion in a chromosome [77]. Interval graphs arise naturally in many scheduling problems: the schedule of courses at a university, the schedule of jobs to machines, the schedule of meetings at a congress etc. For example, in a course schedule, each course corresponds to a vertex and is assigned a time interval and two vertices are adjacent if and only if the corresponding time intervals overlap. Interval graphs also have an application in physical mapping of DNA. One of the techniques to find the structure of DNA involves cutting it into small fragments that are easier to analyze. Pairs of fragments are then tested for overlap using chemical experiments. If all pairs of fragments are tested for overlap then the problem is equivalent to finding an interval representation from the graph generated by fragment overlaps. We give more details about the application in chapter 3.2.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two interval graphs sharing some vertices $I$ and the edges induced by $I$ (the set of edges whose endpoints are in $I$ are the same in both the graphs).

$G_1$ and $G_2$ are said to be *simultaneous interval graphs* if there exist interval representations $R_1$ and $R_2$ of $G_1$ and $G_2$ respectively, such that any vertex of $I$ is represented by the same interval in both $R_1$ and $R_2$. For example, Figures 1.2(a) and 1.2(b) show two simultaneous interval graphs and their interval representations with the property that vertices common to both graphs are assigned to the same interval. Figure 1.2(c) shows two interval graphs that are not simultaneous interval graphs, since a consistent representation doesn't exist. This is because in any interval representation of $G_1$, the right end point of interval $b$ should appear between the right end points of $a$ and $c$, because of the path $a, b, d, c$. On the other hand, in any interval representation of $G_2$, the right end point of interval $a$ should appear between the right end points of $a$ and $c$, because of the path $b, a, f, c$.

Simultaneous interval graphs arise in several settings, for example, DNA structures of two organisms sharing some fragments, two conference schedules that share some plenary talks and job schedules on two machines in which certain jobs are processed synchronously. Simultaneous interval graphs may also have an application in physical mapping of DNA as a special case of the interval graph sandwich problem (see chapter 3.2 for details). Our second main result is an efficient algorithm for recognizing simultaneous interval graphs.

The simultaneous representation problem can be defined in a similar way for any other intersection class. Let $\mathcal{C}$ be any intersection graph class and let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs in $\mathcal{C}$, sharing some vertices $I$ and the edges induced by $I$. $G_1$ and $G_2$ are said to be *simultaneous $\mathcal{C}$-representable graphs* or *simultaneous $\mathcal{C}$ graphs* if there exist intersection representations $R_1$ and $R_2$ of $G_1$ and $G_2$ such that any vertex of $I$ is represented by the same object in both $R_1$ and $R_2$. The *simultaneous $\mathcal{C}$ representation problem* asks whether $G_1$ and $G_2$ are simultaneous $\mathcal{C}$ graphs.

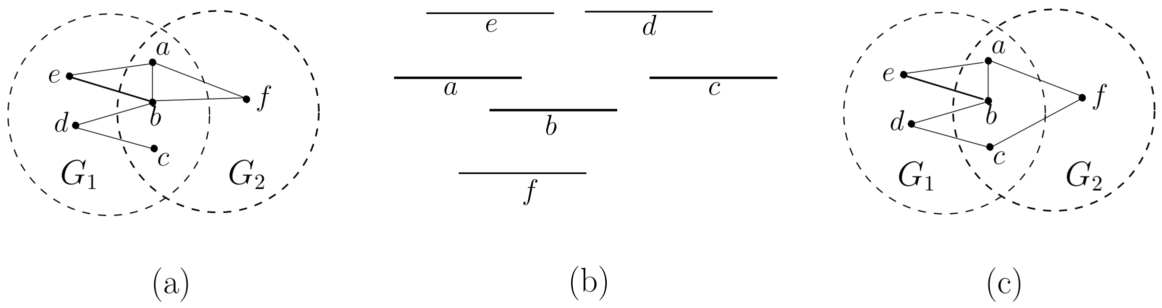


Figure 1.2: The graphs in (a) are simultaneous interval graphs since they have a consistent representation as shown in (b). Graphs in (c) are not simultaneous interval graphs.

Chordal graphs and permutation graphs are both intersection graphs and are well-studied in the literature. Chordal graphs are graphs with no induced cycles of length greater than 3. They are a generalization of interval graphs and can be characterized as the intersection graphs of subtrees of a tree (interval graphs are the intersection graphs of subtrees of a path). Permutation graphs are the intersection graphs of a family of line segments that connect two parallel lines. We study the simultaneous representation problem for both of them and give efficient recognition algorithms. Simultaneous chordal graphs, when defined for multiple graphs, have an application in constructing perfect phylogenies. A phylogeny is a tree that represents the evolutionary branching of a set of species (see section 3.2 for more details).

We now define comparability graphs and explain how the simultaneous representation problem is defined on them. A directed graph or a digraph is a graph where each edge has a direction. A digraph is said to be transitive if for any three vertices $a, b, c$, the existence of (directed) edges, from $a$ to $b$ and from $b$ to $c$ implies the existence of an edge from $a$ to $c$. A transitive orientation of an undirected graph is an assignment of a direction (or orientation) to each edge such that the resulting digraph is transitive. Not all graphs have a transitive orientation (e.g. a cycle of length 5 does not). Comparability graphs are defined as the graphs that have a transitive orientation.

Comparability graphs are not known to be intersection graphs, though their complements are intersection graphs. However, we can define the simultaneous representation problem on comparability graphs as follows. Two comparability graphs $G_1$ and $G_2$ sharing some common vertices $I$ and the edges induced by $I$ are said to be simultaneous comparability graphs if there exist transitive orientations $T_1$ and $T_2$ of $G_1$ and $G_2$ (respectively) such that any common edge is oriented in the same way in both $T_1$ and $T_2$. For example, the graphs in figure 1.3.1 are simultaneous comparability graphs as demonstrated by the orientation in figure 1.3.2. On the other hand the graphs in figure 1.3.3 are not simultaneous comparability graphs, as orienting edge say $ab$, say from $a$ to $b$, forces the orientation of all other edges and in particular the edge $gd$ is forced to go from $g$ to $d$ in $G_1$ and from $d$ to $g$ in $G_2$, as shown in figure 1.3.4. Note that orienting $ab$ from $b$ to $a$ instead would have a similar problem. Our third main result is an efficient algorithm for recognizing simultaneous comparability graphs.

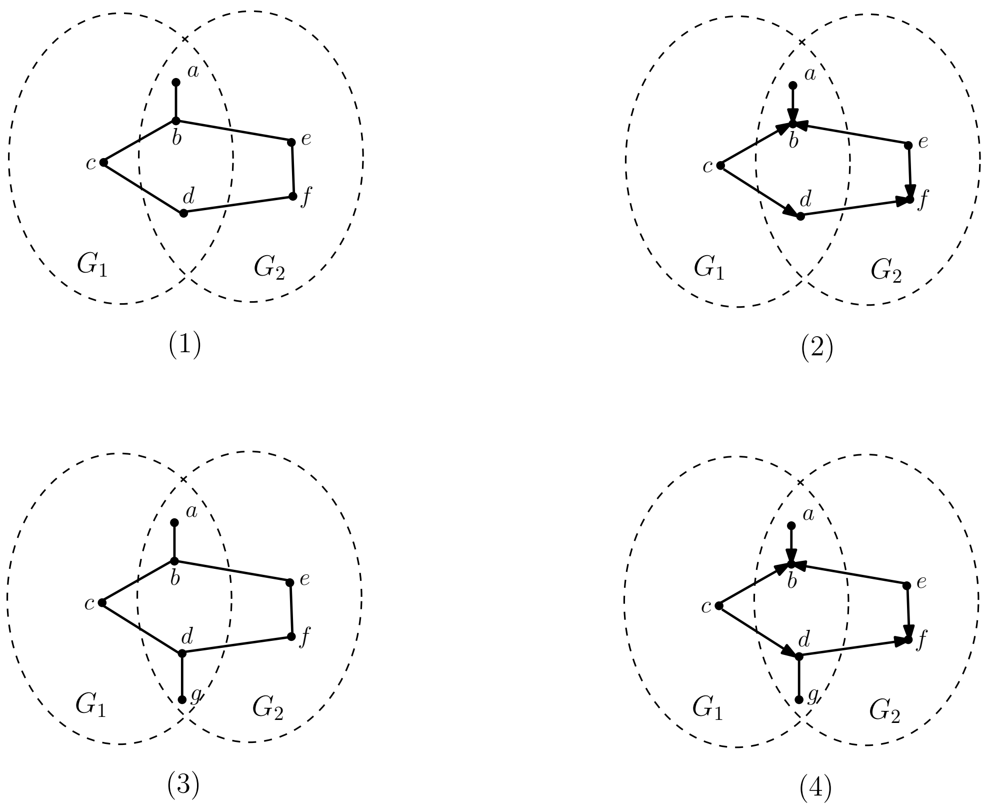


Figure 1.3: The graphs in (1) are simultaneous comparability graphs as shown by the orientation in (2). The graphs in (3) are not simultaneous comparability graphs.

Our results on simultaneous intersection graphs and simultaneous comparability graphs rely on a fundamental result that shows that the problem is equivalent to a certain graph augmentation problem. Let $\mathcal{C}$ be any intersection graph class or the class of comparability graphs. We show that the simultaneous $\mathcal{C}$ representation problem for $G_1$ and $G_2$ is equivalent to the following graph augmentation problem: Do there exist edges $E' \subseteq V_1 - I \times V_2 - I$ such that the augmented graph $(V_1 \cup V_2, E_1 \cup E_2 \cup E')$ belongs to $\mathcal{C}$. Consider the graphs in figure 1.2(a) and their simultaneous interval representation in figure 1.2(b). The interval representation contains all the intersections corresponding to the edges of $G_1 \cup G_2$ and some additional edges namely, $ef$ and $fd$. The additional edges must indeed come from $V_1 - I \times V_2 - I$, and adding these edges to $G_1 \cup G_2$ makes it an interval graph. Conversely, the existence of such augmenting edges that can make $G_1 \cup G_2$ an interval graph implies that $G_1$ and $G_2$ are simultaneous interval graphs. Theorem 2.1 shows this formally for intersection classes and Theorem 6.1 proves the equivalence for comparability graphs.

This equivalence implies that simultaneous representation problems are closely related to graph sandwich problems and probe graphs. In the *graph sandwich problem* for $\mathcal{C}$, given a graph $H = (V, E)$ and a set of 'optional edges' $E_o$ incident on $V$, where $E \cap E_o = \emptyset$, we have to determine whether $H$ can be converted into a class $\mathcal{C}$ graph by adding certain edges from $E_o$. Thus the simultaneous representation problem for $\mathcal{C}$, is a special case of graph sandwich problem for $\mathcal{C}$, in which the set of optional edges induce a complete bipartite graph. *Probe graphs* are another special case, where the set of optional edges induce a clique. However the graph sandwich problem is NP-hard for many interesting classes of graphs including interval graphs, chordal graphs, comparability graphs and permutation graphs. More details about sandwich problems and probe graphs are presented in chapter 3.

Simultaneous representation problem can be defined for multiple graphs, where any two graphs share some common vertices and the edges induced by them. This is equivalent to a graph sandwich problem where the optional edges join pairs of vertices from different graphs. In this thesis, we consider a special case of the problem, where there are multiple graphs (say $r$ of them), any two graphs share the same the same common graph. In other words, every vertex or edge is either common to all the graphs or belongs to exactly one of the graphs. This version of the problem is equivalent to a graph sandwich problem where the set of optional edges induce a complete $r$-partite graph, for some $r$. Thus this definition also generalizes probe graphs.

In this thesis, we give efficient algorithms for recognizing simultaneous interval graphs, simultaneous chordal graphs, simultaneous comparability graphs and simultaneous permutation graphs. More specifically, given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ sharing a set $I$ of vertices and the edges induced by $I$, with $n = |V_1 \cup V_2|$ and $m = |E_1 \cup E_2|$, we give:

**1.** An $O(n^3)$ algorithm to determine whether $G_1$ and $G_2$ are simultaneous chordal graphs.

**2.** An $O(n^2 \log n)$ algorithm to determine whether $G_1$ and $G_2$ are simultaneous interval graphs.

**3.** An $O(nm)$ algorithm to determine whether $G_1$ and $G_2$ are simultaneous comparability graphs.

**4.** An $O(n^3)$ algorithm to determine whether $G_1$ and $G_2$ are simultaneous permutation graphs.

Further, for comparability graphs and permutation graphs, our algorithm can solve the version of problem for multiple graphs where the intersection of any two graphs is the same. The techniques used to obtain the above results are independent of each other, except that our algorithm for simultaneous permutation graphs uses the algorithm for simultaneous comparability graphs as a black box.

The rest of the thesis is organized as follows. Chapter 2 gives notation, definitions and preliminaries. It also contains some crucial results that will be used in the rest of the thesis. In chapter 3, we discuss the background on several related problems. The remaining chapters are independent of each other and can be read in any order. In chapters 4, 5, 6 and 7, we study the simultaneous representation problem for chordal, interval, comparability and permutation graphs respectively. In chapter 8, we study the problem of testing simultaneous planarity, when the common graph is 2-connected. Finally we present conclusions and open problems in chapter 9.

# Chapter 2

# Preliminaries

In this chapter, we give notation and basic definitions and prove some preliminary theorems that will be used in the remainder of the thesis. The chapter is organized as follows. In section 2.1, we define the graph classes used or mentioned in this thesis. We also define the modular decomposition of a graph in this section, as we use it in explaining related work. Section 2.2 defines PQ-trees and their properties. PQ-trees play a major role in this thesis and are used in recognizing simultaneous interval graphs and in testing simultaneous planarity. In section 2.3, we define the simultaneous representation problem for intersection graphs and comparability graphs, on a set of graphs, where any two graphs may share some vertices. We also define a special arrangement of graphs called 'sunflower graphs' which are the main focus of this thesis. Our results on simultaneous interval, chordal, comparability and permutation representation problems assume that the input graphs are sunflower graphs. We also show in section 2.3 that the simultaneous representation problem for any intersection graph class is equivalent to a graph augmentation problem.

Throughout the thesis we only consider simple graphs, with no self loops. The graphs will be undirected unless otherwise specified. For a graph $H$, we use $V(H)$ and $E(H)$ to denote its vertex set and edge set respectively. An edge between vertices $u$ and $v$ is denoted by $(u, v)$ or $uv$. A directed edge from $u$ to $v$ is denoted by $\overrightarrow{uv}$. Given a set $S \subseteq V(H)$ of vertices, we use $H[S]$ to denote the graph induced by $S$. We use $E(S)$ as a shorthand for $E(H[S])$, when the graph $H$ is clear from the context. Given a vertex $v$ and a set of edges $A$, we use $N_A(v)$ to denote the *neighbors of* $v$ w.r.t $A$ i.e., the vertex set $\{u : (u, v) \in A\}$. Also $N(v)$ denotes all the neighboring vertices of $v$ and $N[v] = N(v) \cup \{v\}$. We use $E_H(v)$ to denote the edges incident to $v$ i.e., the edge set $\{(u, v) : u \in V(H), (u, v) \in E(H)\}$. We use $H - v$ to denote the graph obtained by removing $v$ and its incident edges from $H$.

Given a graph $G = (V, E)$, its complement is defined as the graph $\bar{G} = (V, \bar{E})$, where $\bar{E}$ consists of all edges between vertices in $V$ which are not present in $E$. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, their *union* is defined as the graph $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. If $E'$ is a set of edges incident on $V_1$, then $G_1 \cup E'$ is the graph $G_1 \cup (V_1, E')$.

## 2.1 Definitions

In this section, we define the main graph classes used or mentioned in this thesis. We also define modular decomposition, as we need it in explaining related work.

**Intersection graph.** An *intersection graph* is one that has an intersection representation consisting of an object for each vertex such that there is an edge between two vertices if and only if the corresponding objects intersect. Every graph is an intersection graph because we can define the object associated with each vertex to be the set of edges incident to it. However, when the objects are restricted to be special sets (e.g., intervals, subtrees, disks, line segments, etc) we get interesting classes.

**Chordal graph.** A *chord* in a cycle is an edge that connects two non-consecutive vertices in the cycle. Chordal graphs are defined as the graphs in which every cycle of length greater than 3 has a chord. Chordal graphs have several applications including in sparse matrix computations [7] and in constructing perfect phylogeny [77]. A number of characterizations are known for chordal graphs. We list a few of them here.

Chordal graphs can be characterized as the intersection graphs of subtrees of a host tree [44]. In other words, given a chordal graph, we can find a host tree and a family of subtrees of the host tree, one for each vertex, in such a way that two vertices are adjacent in the graph if and only if their corresponding subtrees share a vertex. There exists a host tree in which the nodes correspond to the maximal cliques of the graph and for each vertex the maximal cliques containing the vertex form a (connected) subtree [7]. Such a tree is called a *clique tree* of the graph.

A vertex $v$ is said to be *simplicial* if $N(v)$ induces a clique. A *perfect elimination ordering* is an ordering $v_1, v_2, \ldots, v_n$ of the vertices of the graph such that each $v_i$ is simplicial in the subgraph induced by $\{v_i, \ldots, v_n\}$. Chordal graphs are characterized by the existence of a perfect elimination ordering [44].

A *minimal separator* of a graph $G = (V, E)$ is a minimal set $S$ of vertices whose removal separates a pair of vertices $u, v \in V - S$. A vertex $x$ is *LB-simplicial* if every minimal separator contained in $N(x)$ is a clique. Chordal graphs can be characterized as graphs whose vertices are all LB-Simplicial [63].

**Comparability graph.** A graph is a *comparability graph* if its edges can be *transitively oriented*, i.e. for any three vertices $a, b, c$ in the graph whenever the directed edges from $a$ to $b$ and from $b$ to $c$ exist in the orientation, the edge $ac$ must be present in the graph and must be oriented from $a$ to $c$. Note that a transitive orientation is acyclic by definition, as we do not allow self-loops. Comparability graphs are not known to have a characterization as intersection graphs. However co-comparability graphs (i.e., complements of comparability graphs) can be characterized as the intersection graphs of continuous real-valued functions over some interval [47].

**Permutation graph.** A graph $G = (V, E)$ on vertices $V = \{1, \ldots, n\}$ is a *permutation graph* if there exists a permutation $\pi$ of the numbers $1, 2, \ldots, n$ such that for all $1 \leq i \leq j \leq n$, $(i, j) \in E$ if and only if $\pi(i) > \pi(j)$. Equivalently, $G = (V, E)$ is a permutation graph if and only if there are two parallel lines $l$ and $p$ and a set of line segments each connecting a distinct point on $l$ with a distinct point on $p$ such that $G$ is the intersection graph of the line segments [44].

**Interval graph.** A graph is an *interval graph* if it is the intersection graph of intervals on the real line. Equivalently, a graph is an interval graph if it is chordal and if its complement is a comparability graph [44]. Thus interval graphs are a subclass of chordal graphs and also a subclass of co-comparability graphs.

The *clique matrix* of a graph is a 0-1 matrix whose columns correspond to the maximal cliques and whose rows correspond to the vertices. An entry at row $i$ and column $j$ is 1 if and only if the clique at column $j$ contains the vertex that corresponds to $i$. It is well known that a graph is an interval graph if and only if its columns can be permuted so that in each row all the 1s appear consecutively [44]. The latter property is called the *consecutive 1s property*.

An interval representation of a graph is a set of intervals whose intersection defines the graph. We assume without loss of generality that no two interval end points are the same. A *combinatorial interval representation* or an *interval realizer* is the ordering of the end points of an interval representation.

**Planar graphs.** A graph is a *planar graph* if it can be embedded (or drawn) in the plane with points for vertices and curves for edges in such a way that no two edges cross each other. Planar graphs cannot be characterized as intersection graphs, as any intersection class of graphs must contain the class of complete graphs. However, Scheinerman's conjecture [75], which was proved recently by Chalopin and Goncalves [15], states that planar graphs are a subclass of the intersection graphs of sets of line segments in the plane.

**Weakly chordal graphs.** A graph is *weakly chordal* if neither the graph nor its complement contain an induced cycle of length greater than 4 [53]. Thus the graphs are closed under taking complement and properly contain the class of chordal graphs.

**Strongly chordal graphs.** A graph is *strongly chordal* if it is chordal and every even cycle of size at least 6 has a chord that divides the cycle into two odd paths [36].

**Proper interval graphs.** An interval graph is said to be a *proper interval graph* if it has an interval representation in which no interval is contained inside any other [44].

**Unit interval graphs.** An interval graph is said to be a *unit interval graph* if it has an interval representation in which every interval has the same (unit) length [44].

**Distance hereditary graphs.** A graph is *distance hereditary* if it is connected and for every pair of vertices in the graph, the distance between them is the same in every induced subgraph of the graph [4].

**Split graphs.** A graph is a *split graph* if it can be partitioned into a clique and an independent set. Split graphs are a subclass of chordal graphs [44].

**Cographs.** A graph is a *cograph* if it doesn't contain an induced path of length 4. Connected cographs are known to be distance hereditary graphs [44].

**Chordal bipartite graphs.** A graph is *chordal bipartite* if it is bipartite and doesn't contain induced cycles of length greater than 4 [45].

**Bipartite distance hereditary graphs.** *Bipartite distance hereditary graphs* are distance hereditary graphs that are also bipartite.

**Perfect graphs.** A graph $G$ is *perfect* if for every induced subgraph $H$, the chromatic number of $H$ is the same as the size of the maximum clique in $H$ [44]. The chromatic number of a graph is defined as the minimum number of colors that can be assigned to the graph such that no two adjacent vertices have the same color. Perfect graphs contain many other graph classes including weakly chordal graphs, comparability and co-comparability graphs.

**Trivially perfect graphs.** A graph is *trivially perfect* if it is chordal and is a cograph. Thus trivially perfect graphs do not have an induced cycle of length greater than 3 and also do not have an induced path of length 4 [44].

**Circular arc graphs.** Circular arc graphs are defined as the intersection graphs of a family of closed arcs of a circle [44]. They are a generalization of interval graphs. Proper circular arc graphs and unit circular arc graphs are defined analogous to proper interval graphs and unit interval graphs.

**Circle graphs.** A graph is said to be a *circle graph* if it is the intersection graph of chords of a circle [77].

**Ptolemaic graphs.** Let $d(u, v)$ denote the distance between vertices $u$ and $v$ in the graph. A graph is said to be *ptolemaic* if it is connected and every four vertices $u, v, w, x$ satisfy the inequality: $d(u, v)d(w, x) \leq d(u, w)d(v, x) + d(u, x)d(v, w)$ [59].

**Threshold graphs.** A graph is a *threshold graph* if there exists a real number $t$ and a function $w$ that assigns a value $w(v)$ to $v$, for all $v$, such that for any two vertices $u, v$ in the graph, $uv$ is an edge if and only if $w(u) + w(v) \geq t$. Threshold graphs are a special case of many other graph classes including cographs, split graphs, trivially perfect graphs and interval graphs [44].

**Modular decomposition.** The *modular decomposition* of a graph is a tree that represents a 'structure' of the graph. It is introduced by Gallai [40, 77] and has several applications, for example, in obtaining a linear-time algorithm for transitively orienting a comparability graph. We don't use modular decomposition in our thesis, except in explaining related work in chapter 3.

Let $G = (V, E)$ be a graph. A set $I \subseteq V$ is said to be a *module* of $G$, if for each vertex $y \in V - I$, either $y$ is adjacent to all the vertices of $I$, or to none of the vertices of $I$. A module $M$ is said to be a *strong module*, if for every other module $M'$, either $M \supseteq M'$ or $M \subseteq M'$ or $M \cap M' = \emptyset$. The strong modules of $G$ form a *modular decomposition* tree $\mathcal{T}$ by inclusion order, with the root node representing $V$, and the leaf nodes representing the singleton modules (vertices) of $G$. It turns out that $\mathcal{T}$ is a compact (linear-space) representation of all modules of $G$.

Each node of $\mathcal{T}$ is of one of three types: *series*, *parallel* or *prime*. Let $v_M$ denote the node of $\mathcal{T}$ that represents the module $M$. If the graph $G[M]$ is disconnected then $v_M$ is said to be a series node and its children correspond to the connected components of $G[M]$. If the complement of $G[M]$ is disconnected then $v_M$ is said to be a parallel node and its children correspond to the connected components of $\bar{G}[M]$. If $G[M]$ and its complement are both connected then $v_M$ is said to be a prime node and it children are the maximal proper modules of $G[M]$. (The maximal modules of a prime module are always unique).

## 2.2 PQ-Trees

PQ-trees play an important role in this thesis. We use them heavily in chapters 5 and 8. PQ-trees were discovered by Booth and Lueker [10] in 1976 and have applications in recognizing interval graphs, testing for planarity and testing whether a matrix has the consecutive-ones property. We first review PQ-trees and then in subsection 2.2.1 we define a few operations on them.

A *PQ-tree* represents the permutations of a set of elements satisfying a family of constraints. Each constraint specifies that a certain subset of elements must appear consecutively in any permutation. The leaves of a PQ-tree correspond to the elements of the set, and internal nodes are labeled 'P' or 'Q', and are drawn using a circle or a rectangle, respectively. PQ-trees are equivalent under arbitrary reordering of the children of a P-node and reversals of the order of children of a Q-node. We consider a node with two children to be a Q-node. A leaf-order of a PQ-tree is the order in which its leaves are visited, in an in-order traversal of the tree. The set of permutations represented by a PQ-tree is the set of leaf-orders of equivalent PQ-trees. Given a PQ-tree $T$ on a set $U$ of elements, adding a consecutivity constraint for a set $S \subseteq U$ *reduces* $T$ to a PQ-tree $T'$, such that the leaf-orders of $T'$ are precisely the leaf-orders of $T$ in which the elements of $S$ appear consecutively. Booth and Lueker [10] gave an efficient implementation of PQ-trees that supports this operation in amortized $O(|S|)$ time.

PQ-trees can capture all possible combinatorial interval representations of an interval graph and all possible combinatorial planar embeddings of a planar graph. In interval graph recognition, a PQ-tree is used to capture the set of linear orderings of the maximal-cliques of the graph. The leaves of the PQ-tree correspond to maximal-cliques of the graph. In planarity testing, a PQ-tree is used to capture the set of circular orderings of unembedded edges around a partially-embedded component (see chapter 8 for details). Although PQ-trees were invented to represent linear orders, they can be reinterpreted to represent circular orders as well [51]. We explain this in chapter 8.

### 2.2.1 Intersection and Projection of PQ-Trees

In this subsection, we define a few basic operations on PQ-trees. These operations are used in chapters 5 and 8.

The *projection* of a PQ-tree on a subset of its leaves $S$ is a PQ-tree obtained by deleting all elements not in $S$ and simplifying the resulting tree. Simplifying a tree means that we (recursively) delete any internal node that has no children, and delete any node that has a single child by making the child's grandparent become its parent. This can easily be implemented in linear time.

Given two PQ-trees on the same set of leaves (elements), we define their *intersection* to be the PQ-tree $T$ that represents exactly all orders that are leaf-orders in both trees. This intersection can be computed as follows.

1. Initialize $T$ to be the first PQ-tree.

2. For each P-node in the second PQ-tree, reduce $T$ by adding a consecutivity constraint on all its descendant leaves.

3. For each Q-node in the second tree, and for each pair of adjacent children of it, reduce $T$ by adding a consecutivity constraint on all the descendant leaves of the two children.

Using the efficient PQ-tree implementation such an intersection can be computed in time linear in the size of the two PQ-trees (see Booth's thesis [9]).

## 2.3 Simultaneous Intersection Graphs and Sunflower Graphs

In this section, we formally introduce the simultaneous representation problem for intersection graphs and comparability graphs. We define an arrangement of graphs called the "sunflower graphs" and study the simultaneous representation problem on them. We also present some basic results on these graphs.

We first define the general version of simultaneous representation problem for intersection graph classes and comparability graphs. Let $G_1, G_2, \ldots, G_r$ be $r$ graphs, where a vertex/edge may be present in multiple graphs. We then say that the graphs *share* the vertex/edge.

Let $\mathcal{C}$ be an intersection graph class. Then $G_1, G_2, \ldots, G_r$ are said to be *simultaneous* $\mathcal{C}$ *graphs* if for each $i \in \{1, \ldots, r\}$, there exists an intersection representation $R_i$ for $G_i$ (in $\mathcal{C}$), such that any vertex $v$ that appears in multiple graphs is assigned to the same object in all the representations of the graphs, i.e. $v$ is assigned to the same object in all the representations that contain $v$. The *simultaneous* $\mathcal{C}$ *representation problem* for $G_1, G_2, \ldots, G_r$ asks whether $G_1, G_2, \ldots, G_r$ are simultaneous $\mathcal{C}$ graphs. Observe that, as $\mathcal{C}$ is an intersection class, if $G_1, G_2, \ldots, G_r$ are simultaneous $\mathcal{C}$ graphs, then each of the individual graphs must belong to $\mathcal{C}$ and further they must satisfy the following necessary condition: If an edge $uv$ is present in some graph, then any other graph that contains vertices $u$ and $v$ must also contain the edge $uv$. Hence when considering the simultaneous representation problem for interval graphs, chordal graphs and permutation graphs we may assume that the input graphs satisfy this condition. We also make this assumption for simultaneous comparability graphs (defined below), though they are not known to be intersection graphs.

In the case where $r = 2$, which is our starting point, this necessary condition can be stated very simply: the edges induced by the common vertices must be the same in both graphs.

$G_1, G_2, \ldots, G_r$ are said to be *simultaneous comparability graphs* if for each $i \in \{1, \ldots, r\}$, there exists a transitive orientation $W_i$ of $G_i$ such that every edge $e$ is oriented the same way in all the orientations that contain $e$.

The simultaneous representation problem is interesting in the general case, but in this thesis, we mainly concentrate on the problem for 2 graphs. In some cases, our techniques extend to multiple graphs with a special kind of intersection structure. To capture this intersection structure, we define a family of graphs called the 'sunflower graphs'. Formally, we define the *$r$-sunflower graphs* to be a family of $r$ graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$, sharing some vertices $I$ and the edges induced by $I$, i.e., for any two distinct $i, j \in \{1, \ldots, r\}$, $V_i \cap V_j = I$ and $G_i[I] = G_j[I]$. Additionally if $G_1, G_2, \ldots G_r$ all belong to a class $\mathcal{C}$, then we use the term *$r$-sunflower* $\mathcal{C}$ *graphs* to denote them. Figure 2.1 shows the structure of 5-sunflower graphs.

An *augmenting edge* in $r$-sunflower graphs is an edge whose end points appear in distinct graphs. In other words, an augmenting edge belongs to the set $\{\bigcup(V_i - I) \times (V_j - I) \mid i, j \in \{1, \ldots, r\} \wedge i \neq j\}$. Let $\mathcal{C}$ be any graph class. The *sunflower $\mathcal{C}$ augmentation problem* for $G_1, G_2, \ldots, G_r$ asks whether there exists a set $A$ of augmenting edges such that $G_1 \cup G_2 \cup \cdots G_r \cup A$ is a class $\mathcal{C}$ graph. $G_i, i \in \{1, \ldots, r\}$ are said to be *augmentable to a $\mathcal{C}$ graph* if they satisfy the sunflower $\mathcal{C}$ augmentation problem.

This problem can be defined for any graph class $\mathcal{C}$ (not just for intersection graph classes) and is closely related to probe graphs and graph sandwich problems (see chapter 3).

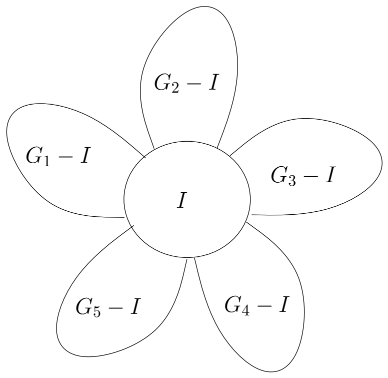


Figure 2.1: The structure of 5-sunflower graphs in which the graphs $G_1, \ldots, G_5$ share the vertices of $I$ and its induced edges.

We now give an alternative characterization of simultaneous $\mathcal{C}$ representation problem for sunflower graphs in terms of the sunflower $\mathcal{C}$ augmentation problem. The alternative formulation will be useful in relating the simultaneous representation problem to other well-studied problems in the literature. It will also be heavily used in our algorithmic approaches. For intuition, consider the interval graphs $G_1, G_2$ in Figure 1.2(a). The intersection graph of the intervals in Figure 1.2(b) is an interval graph that contains all the edges of $G_1 \cup G_2$ and certain augmenting edges. Thus $G_1, G_2$ can be augmented to an interval graph. In fact (as we formalize below) the existence of such an augmenting set of edges that can create an interval graph is equivalent to the simultaneous interval representation problem for $G_1$ and $G_2$. The following theorem proves this for any intersection graph class.

**Theorem 2.1.** *Let $\mathcal{C}$ be an intersection graph class. Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower $\mathcal{C}$ graphs (sharing some vertices $I$ and the edges induced by $I$). Then $G_i, i \in \{1, \ldots, r\}$ are simultaneous $\mathcal{C}$ graphs if and only if they are augmentable to a $\mathcal{C}$ graph.*

*Proof.* Let $I$ be the set of vertices common to $G_1, \ldots, G_r$ and let $G = G_1 \cup G_2 \cup \cdots \cup G_r$.

Let $G_1, \ldots, G_r$ be simultaneous $\mathcal{C}$ graphs. For $i \in \{1, \ldots, r\}$, let $R_i$ be the intersection representation of $G_i$, such that all $R_i$ are consistent on $I$ (i.e. each vertex in $I$ is assigned to the same object in all $R_i$). In this representation, let vertex $v \in V(G)$ be mapped to object $T_v$. Now consider the intersection graph $G_a$ of $\{T_v : v \in V(G)\}$. Clearly $G_a$ belongs to class $\mathcal{C}$. Further $V(G_a) = V(G)$ and $E(G_a) = E(G) \cup A$, where $A$ is a set of augmenting edges. Thus $G_1, \ldots, G_r$ are augmentable to a $\mathcal{C}$ graph.

For the other direction, suppose $G_1, \ldots, G_r$ are augmentable to a $\mathcal{C}$ graph. Then there exists a set $A$ of augmenting edges such that the graph $G_a = G \cup A$ belongs to class $\mathcal{C}$. Now consider the intersection representation $R$ of $G_a$. Then $R$ maps each vertex $v \in V(G)$ to a set $T_v$. For $i \in \{1, \dots, r\}$, obtain a representation $R_i$ of $G_i$ by restricting the domain of $R$ to $V_i$. Note that $R_i$ is an intersection representation of $G_i$, since $G_i$ is the subgraph of $G$ induced by $V_i$. Now any vertex $v$ in $I$ is mapped to the same set $(T_v)$ in all $R_i$. Thus $G_1, \dots, G_r$ are simultaneous $\mathcal{C}$ graphs. □

Theorem 2.1 also holds for comparability graphs and we prove this in Theorem 6.1.

Let $G_1, G_2, \dots, G_r$ be $r$-sunflower graphs, sharing some vertices $I$. Then observe that their complements $\bar{G_1}, \bar{G_2}, \dots, \bar{G_r}$ are also $r$-sunflower graphs that share $I$. The following theorem shows the relationship between sunflower graphs and their complements.

**Theorem 2.2.** *Let $\mathcal{C}$ be any graph class and let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower $\mathcal{C}$ graphs. Then $G_1, \dots, G_r$ are augmentable to a $\mathcal{C}$ graph if and only if $\bar{G_1}, \dots, \bar{G_r}$ are augmentable to a co-$\mathcal{C}$ graph.*

*Proof.* Let $I$ be the set of vertices shared by the graphs $G_1, \dots, G_r$. Let $G_1, \dots, G_r$ be augmentable to a $\mathcal{C}$ graph. Then there exists a set $A$ of augmenting edges such that the graph $G_a = G_1 \cup G_2 \cup \cdots \cup G_r \cup A$ belongs to $\mathcal{C}$. Let $A' = \{\bigcup (V_i - I) \times V_j - I) \mid i, j \in \{1, \dots, r\} \wedge i \neq j\} - A$. Observe that $A'$ is a set of augmenting edges and the graph $G'_a = \bar{G_1} \cup \bar{G_2} \cup \cdots \cup \bar{G_r} \cup A' = \bar{G_a}$ belongs to co-$\mathcal{C}$. Hence $\bar{G_1}, \dots, \bar{G_r}$ are augmentable to a co-$\mathcal{C}$ graph.

The proof of the converse is symmetric and hence the theorem holds. □

Theorems 2.1 and 2.2 imply that for any intersection graph class $\mathcal{C}$, solving the simultaneous $\mathcal{C}$ representation problem for $r$-sunflower graphs efficiently implies that both sunflower $\mathcal{C}$ augmentation problem for $r$ graphs and sunflower co-$\mathcal{C}$ augmentation problem for $r$ graphs can be solved efficiently. Similarly, Theorems 6.1 and 2.2 imply that solving the simultaneous comparability representation problem for $r$-sunflower graphs efficiently implies that both sunflower comparability augmentation problem and sunflower co-comparability augmentation problem can be solved efficiently for $r$ graphs.

# Chapter 3

# Related Work

The concept of 'simultaneity' has a substantial history for planar graphs. This thesis extends this concept to intersection graphs, where the problem is closely related to some well known problems in the literature: the graph sandwich problem and the probe graph recognition problem. In this chapter we discuss these related problems and the results known for them.

## 3.1 Simultaneous Planar Embeddings

Versions of simultaneous planarity have received much attention in recent years. Brass et al. [13] introduced the concept of *simultaneous geometric embeddings* of a pair of graphs—these are planar straight-line drawings such that any common vertex is represented by the same point. Note that a common edge will necessarily be represented by the same line segment. Estrella-Balderrama et al. [33] showed that it is NP-hard to test if two graphs have simultaneous geometric embeddings. Several results are known for pairs of restricted graphs. Brass et al. [13] proved that simultaneous geometric embeddings always exist for pairs of paths, pairs of cycles and pairs of caterpillars. (A *caterpillar* is a tree such that deleting its leaves results in a path.) Erten and Koburov [32] showed the existence of a planar graph and a path with no simultaneous geometric embedding. Brass et al. [13] found a pair of outer planar graphs that do not have a simultaneous geometric embedding. This was improved by Geyer et al. [42] who found a pair of trees that do not admit a simultaneous geometric embedding. This in turn was improved recently by Angelini et al. [2] who found a tree and a path that do not have a simultaneous geometric embedding. Additionally the tree and the path do not share an edge.

The generalization to planar drawings where edges are not necessarily drawn as straight line segments but any common edge must be represented by the same curve was introduced by Erten and Kobourov [32] and was called *simultaneous embedding with consistent edges*. Most other papers follow the conference version of Erten and Kobourov's paper and use the term *simultaneous embedding with fixed edges (SEFE)*. In our paper we use the more self-explanatory term "simultaneous planar embeddings." A further justification for this nomenclature is that there are combinatorial conditions on a pair of planar embeddings that are equivalent to simultaneous planarity. Specifically, Jünger and Schultz give a characterization in terms of "compatible embeddings" [Theorem 4 in [57]]. Specialized to the case where the common graph is connected, their result says that two planar embeddings are simultaneously planar if and only if the cyclic orderings of common edges around common vertices are the same in both embeddings.

Finding simultaneous planar embeddings is an open problem for 2 graphs. For 3 graphs, Gassner et al. [41] showed that the problem is NP-complete. Several papers [32, 43, 39] show that pairs of graphs from certain restricted classes always have simultaneous planar embeddings, the most general result being that any planar graph has a simultaneous planar embedding with any tree [39]. On the other hand, there is an example of two outerplanar graphs that have no simultaneous planar embedding [39].

The graphs that have simultaneous planar embeddings when paired with any other planar graph have been characterized [38]. In addition, Jünger and Schultz [57] characterize the common graphs that permit simultaneous planar embeddings no matter what pairs of planar graphs they occur in. There are efficient algorithms to test simultaneous planarity for biconnected outerplanar graphs [38] and for a pair consisting of a planar graph and a graph with at most one cycle [37].

At the same time and independent of our work Angelini et al. [1] showed how to test simultaneous planarity of two graphs when the common graph is 2-connected. Their algorithm is based on SPQR-trees, takes $O(n^3)$ time and is restricted to the case where the two graphs have the same vertex set. In comparison, our algorithm for testing simultaneous planarity of two graphs sharing a 2-connected subgraph runs in linear time and doesn't require the two graphs to have the same vertex set. Further, our algorithm can also solve the problem for multiple graphs, where the shared graph is the same for every pair of graphs.

There is another, even weaker form of simultaneous planarity, where common vertices must be represented by common points, but the planar drawings are otherwise completely independent, with edges drawn as Jordan curves. Any set of planar graphs can be represented this way by virtue of the result that a planar graph can be drawn with any fixed vertex locations [52, 72].

## 3.2 Graph Sandwich Problems

Graph sandwich problems were introduced by Golumbic, Kaplan and Shamir [46] in a seminal paper and are defined as follows: For any graph property $\pi$, given graphs $H_1 = (V, E_1)$ and $H_2 = (V, E_2)$ defined on the same set of vertices with $H_1 \subset H_2$, the $\pi$-sandwich problem asks whether there exists a graph $H'$ with property $\pi$ such that $H_1 \subseteq H' \subseteq H_2$. The edges in $H_2 - H_1$ are said to be the *optional* edges. Thus the $\pi$-sandwich problem asks whether we can add some optional edges to $H_1$ so that the resulting graph satisfies $\pi$.

Thus the sunflower $\pi$ augmentation problem for $r$ graphs (see section 2.3) is a special case of the $\pi$-sandwich problem in which the set of optional edges induce a complete $r$-partite graph.

If property $\pi$ is closed under edge addition (e.g. connectivity), then the $\pi$-sandwich problem is equivalent to testing whether $H_2$ satisfies $\pi$. Similarly, if $\pi$ is closed under edge deletion (e.g. planarity), then the problem is equivalent to testing whether $H_1$ satisfies $\pi$. Thus graph sandwich problems are interesting only for properties $\pi$ that are not closed under edge additions or deletions. Note that the $\pi$-sandwich problem for $H_1$ and $H_2$ is equivalent to the co-$\bar{\pi}$ sandwich problem for $\bar{H}_2$ and $\bar{H}_1$.

Golumbic, Kaplan and Shamir [46] mention several applications of graph sandwich problems including the following:

1. **Physical Mapping of DNA:** One of the techniques for DNA sequencing involves cutting the DNA into small fragments that are easier to analyze and to experimentally test for

the overlap between pairs of fragments. The problem then is to arrange the fragments as intervals along a line, so that their pairwise intersections match the experimental data. If the tests are run on all pairs of fragments, then the problem is equivalent to an interval graph problem, where vertices correspond to fragments and two fragments overlap if and only if the vertices are adjacent. However in practice, experiments are inconclusive, incomplete or may be expensive to test for all pairs of fragments. This ambiguity introduces the optional edges. Thus the problem is equivalent to the graph sandwich problem for interval graphs.

2. **Temporal Reasoning:** Given a set of events and a specification that some pairs of events intersect and some do not, the problem is to determine whether this information is consistent. In other words, can we assign an interval for each event that satisfies the specification? Clearly, this problem is equivalent to the graph sandwich problem for interval graphs, where the set of optional edges corresponds to the pairs of events for which no specification is given.

3. **Phylogenetic Trees:** In evolutionary biology, the genealogical relationships between species can be represented in a rooted tree-structure called the *phylogeny.* The leaves of the tree correspond to the species and internal nodes correspond to ancestral species. In particular, the least common ancestor of two leaves represents the nearest common ancestor species of the two leaf species. There are many ways to construct phylogeny trees and the perfect phylogeny problem is to find the best possible tree given the characteristics of the species. More specifically, suppose each species has a set of characteristics, where each characteristic can take one of several values (e.g., a characteristic can be 'skull size', whose value can be 'small', 'medium' and 'large'). The perfect phylogeny problem asks whether there is a rooted tree whose leaves are the species, whose internal nodes have some value assigned to each characteristic, in such a way that nodes with the same characteristic value form a connected subtree.

   Thus the problem involves constructing subtrees of a host tree and is thus connected to chordal graphs. Buneman [14] discovered this connection and showed that the phylogeny problem is equivalent to the problem of triangulating colored graphs (TCG) (see [77] for more details), which in turn is a restriction of the graph sandwich problem for chordal graphs. In section 4.2, we show that TCG is also a restriction of the simultaneous chordal representation problem for $r$-sunflower graphs.

Golumbic, Kaplan and Shamir [46] studied the graph sandwich problem for various properties $\pi$ (i.e. for various graph classes). They gave polynomial-time algorithms for threshold, split and cographs and showed that for comparability, permutation, chordal, circle and circular-arc graphs the problem is NP-complete. Golumbic and Shamir [48] showed that the interval graph sandwich problem is NP-complete. Kaplan and Shamir [58] proved that interval sandwich problem can be solved in polynomial time when either: (1) the degrees of the input graphs and the clique size of the solution graph are bounded; or (2) the degree of the solution graph is bounded.

Subsequently, graph sandwich problems have been studied for several properties by many authors. A graph is said to be $P_4$*-sparse* if every induced subgraph on 5 vertices contains at most one path of length 4. Dantas et.al. [27] showed that the $P_4$-sparse graph sandwich problem can be solved in polynomial time. A graph $G$ is $(k, l)$ if it can be partitioned into at most $k$ independent sets and $l$ cliques. Dantas et al. [26] showed that the $(k, l)$ sandwich problem is NP-complete when $k + l > 2$ and is polynomial-time solvable otherwise. A homogeneous set is a non-trivial module of a graph (see section 2.1). Figueiredo et al. [28] gave an efficient algorithm for the homogeneous set sandwich problem.

The graph sandwich problem is known to be NP-complete for several other graph properties: strongly chordal graphs [29], chordal bipartite graphs [29], skew-partitions [79], and 1-join compositions [30].

Habib et al. [49] studied variations of the graph sandwich problem in which the input graphs are directed graphs or posets. They gave polynomial time algorithms for the interval poset sandwich problem and the series-parallel poset sandwich problem.

## 3.3 Probe Graphs

Another graph concept closely related to that of simultaneous graphs is the concept of probe graphs defined as follows: For any graph class $\mathcal{C}$, given a graph $G = (P \cup N, E)$, where $N$ is an independent set, $G$ is said to be a *probe $\mathcal{C}$ graph*, if it can be transformed into a $\mathcal{C}$ graph by only adding edges between the vertices of $N$. Thus the probe graph recognition problem is a special case of the graph sandwich problem in which the set of optional edges induces a clique. Further, since a clique on $k$ vertices can be viewed as a complete $k$-partite graph, the problem of recognizing probe $\mathcal{C}$ graphs is a special case of the sunflower $\mathcal{C}$ augmentation problem for $r$ graphs, for appropriate $r$.

The problem of recognizing probe graphs was first studied for interval graphs [81, 68], where it has an application in molecular biology, as a tool for physical mapping of DNA. As explained in the previous section, one of the techniques for DNA sequencing involves cutting the DNA into smaller fragments and testing the overlap between pairs of fragments. One approach to reduce the number of overlap tests is to select a set of fragments called the 'probes' and test the overlap information between two fragments if and only if at least one of them is a probe. Thus probe interval graphs were invented to model this scenario.

Johnson and Spinrad [57] gave an $O(n^2)$ algorithm for recognizing probe interval graphs. They first represent the interval graph induced by probes (i.e. $G[P]$) as a Modular Decomposition-PQ-Tree (MD-PQ-Tree). This is a variant of PQ-tree in which the leaves correspond to endpoints of intervals, instead of maximal cliques. The tree is constructed from the modular decomposition of the graph. The leaf-orders of the MD-PQ-tree correspond to all possible interval representations of the graph. Next, they add non-probes one at a time and modify the MD-PQ-Tree by adding the constraints imposed by each non-probe. This requires an extensive case-analysis and unfortunately only a few cases are provided in [57], because of the page-limit. If all non-probes can be successfully added then the graph is a probe interval graph. If the graph is a probe interval graph, their algorithm constructs one possible interval realizer. However they claim that they can modify the algorithm to represent all possible interval realizers.

Later, McConnell and Spinrad [67] gave an $O(n + m \log n)$ algorithm. They first extend the modular decomposition by adding extra constraints on the modules. The resulting tree is called the $\Delta$-tree and it represents all possible realizers of an interval graph. McConnell and Spinrad [67] show how to extend the $\Delta$-tree to represent the realizers of a probe interval graph. Although their algorithm is faster, its description and the correctness proof seem to be complicated.

Recently, a linear time algorithm was discovered by McConnell and Nussbaum [66]. Their algorithm is also surprisingly simpler than the previous methods. They begin with a clique-matrix of the graph (see section 2.1) induced by the probes and extend it by adding more constraints (rows) and cliques (columns). They partition the non-probes into the following categories: (1) Non-probes $x$ such that $N(x)$ contains one or more maximal cliques of $G[P]$; (2) Non-probes $x$

such that $N(x)$ contains zero maximal cliques of $G[P]$; and (3) Non-probes that are simplicial vertices. They show that each of the non-probes in categories 1 and 2 introduces constraints into the clique matrix and each non-probe in category 3 introduces a new column (clique) in the clique-matrix. Finally, they prove that the given graph is a probe interval graph if and only if the new matrix satisfies the consecutive 1's property.

Several problems have been studied on probe interval graphs. Brandstadt et al. [12] showed that probe interval graphs always have a tree 7-spanner: there exists a subtree of the probe interval graph such that the distance between every pair of vertices in the tree is at most 7 times the distance in the graph. They also gave an $O(m \log n)$ algorithm to find a tree 7-spanner. Uehara [80] gives an $O(n^2)$ algorithm to solve the graph isomorphism problem for probe interval graphs, using a variation of PQ-trees. Przulj and Corneil [73] showed that there are at least 62 forbidden subgraphs for 2-tree probe interval graphs.

Probe chordal graphs have been studied by Berry et al. [6], who gave an $O(nm)$ time recognition algorithm. Recall that chordal graphs can be characterized as graphs whose vertices are all LB-Simplicial [63]. Berry et.al [6] extend this characterization to probe chordal graphs. Given a probe graph $G = (P + N, E)$, they define a vertex $x$ to be *quasi LB-simplicial* if every minimal separator contained in $N(x)$ is a *probe clique* (a clique if all the non-probes were to induce a clique). They show that $G$ is a probe graph if and only if each of its vertices is quasi LB-simplicial.

Chandler et al. [19] studied the probe problem for comparability and permutation graphs and gave $O(nm)$ and $O(n^3)$ algorithms respectively. Their algorithm for probe comparability graphs is a straightforward extension of Golumbic's algorithm for recognizing comparability graphs, though the correctness proof is slightly complex. Using this they obtain an algorithm for recognizing probe permutation graphs.

The probe graph recognition problem is also studied for several other classes including strongly chordal graphs [22], chordal bipartite graphs [22], distance-hereditary graphs [18], bipartite distance-hereditary graphs [18], threshold graphs [5], trivially perfect graphs [5], ptolemaic graphs [16], split graphs [62] and cographs [61]. For all of these graphs, the problem is polynomial-time solvable. It is easy to see that for any graph class $\mathcal{C}$, an algorithm for recognizing probe $\mathcal{C}$ graphs can be used to obtain an algorithm for recognizing co-$\mathcal{C}$ graphs. Le and Ridder [62] have an interesting conjecture that for any graph class $\mathcal{C}$, the probe $\mathcal{C}$ graph recognition problem is polynomial time solvable if and only if the recognition problem for class $\mathcal{C}$ is polynomial time solvable. This was initially conjectured for perfect graphs and has been subsequently extended for all graph classes [65].

Probe graphs have also been studied when the partition of vertices into probes and non-probes is not explicitly given. Several graph classes can also be recognized under this restricted setting including: chordal graphs [6], interval graphs [23], distance-hereditary graphs [24] and cographs [62]. Recently, Chandler, Chang, Kloks and Peng have written a book on probe graphs that contains most of these results and many open problems [21].

Our algorithms for simultaneous representation problem for chordal, comparability and permutation graphs are a simple extension of the recognition algorithms for these classes, though the proofs are a bit more involved. Hence our algorithms are also similar to the recognition algorithm for probe graphs for these classes. However for comparability and permutation graphs we can solve the version of the problem with multiple graphs, i.e., we can solve the simultaneous comparability representation problem and simultaneous permutation representation problem for $r$-sunflower graphs, for arbitrary $r$. Thus we generalize the recognition algorithms for probe comparability and probe permutation graphs. Moreover our algorithm for comparability graphs has the same time complexity as the best known algorithm for recognizing probe comparability graphs. Our algorithm for determining whether 2 graphs are simultaneous interval graphs is a non-trivial extension of the interval graph recognition algorithm and is not similar to any of the algorithms proposed for recognizing probe interval graphs. We explain why this problem seems to be 'harder' than the probe interval recognition problem in the beginning of chapter 5.

# Chapter 4

# Simultaneous Chordal Graphs

In this chapter[1], we give an efficient algorithm for determining whether two chordal graphs sharing some vertices are simultaneous chordal graphs. In other words, we solve the simultaneous chordal representation problem for 2-sunflower graphs. This implies that the sunflower chordal augmentation problem and the sunflower co-chordal augmentation problem can be solved efficiently for 2-sunflower graphs. We also show that the problem is NP-complete for $r$-sunflower graphs.

Recall from section 2.1 that chordal graphs are characterized by the existence of a perfect elimination ordering, i.e. there exists an ordering $v_1, \ldots, v_n$ of the vertices such that each $v_i$ is simplicial in the subgraph induced by $\{v_i, \ldots, v_n\}$.

## 4.1 Algorithm for 2-Sunflower Graphs

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be 2-sunflower chordal graphs, sharing some vertices $I$ (and the edges induced by $I$). As mentioned in chapter 2, we define an *augmenting edge* to be an edge between $V_1 - I$ and $V_2 - I$. Given $G_1, G_2$, and a set $A$ of augmenting edges between $V_1 - I$ and $V_2 - I$, we use $(G_1, G_2, A)$ to denote the graph whose vertex set is $V(G_1) \cup V(G_2)$ and whose edge set is $E(G_1) \cup E(G_2) \cup A$. Note that by Theorem 2.1, the simultaneous chordal representation problem for $G_1, G_2$ is equivalent to asking whether there exists a set $A$ of augmenting edges such that the graph $(G_1, G_2, A)$ is chordal. We solve the following generalized problem: Given $G_1$, $G_2$ and $I$ (as above), and a set $F$ of *forced* augmenting edges, does there exist a set $A$ of augmenting edges such that the graph $(G_1, G_2, F \cup A)$ is chordal?

For a vertex $v$ in $G = (G_1, G_2, F)$, we use $N_1(v)$ and $N_2(v)$ to denote the sets $N_{E(G)}(v) \cap V(G_1)$ and $N_{E(G)}(v) \cap V(G_2)$ respectively. In other words, $N_1(v)$ (resp. $N_2(v)$) denotes the neighbors of $v$ in graph $G$ that belong to $G_1$ (resp. $G_2$). Note that if $v \in V_1 - I$, then $N_2(v)$ may be non-empty because of $F$. Finally, we use $C(v)$ to denote the set of augmenting edges with both endpoints adjacent to $v$, i.e., $C(v) = \{(x, y) : x \in N_1(v) - I, y \in N_2(v) - I\}$. A vertex $v$ in $G = (G_1, G_2, F)$ is said to be *$S$-simplicial* (where $S$ stands for 'simultaneous'), if $N_1(v)$ and $N_2(v)$ induce cliques in $G_1$ and $G_2$ respectively.

**Lemma 4.1.** *If $G = (G_1, G_2, F)$ is augmentable to a chordal graph, then there exists an $S$-simplicial vertex $v$ of $G$.*

[1]The results presented in this chapter are joint work with Anna Lubiw [55].

*Proof.* Let $A$ be a set of augmenting edges such that the graph $G' = (G_1, G_2, F \cup A)$ is chordal. Because $G'$ is chordal it has a simplicial vertex, i.e. a vertex $v$ such that $N_{E(G')}(v)$ induces a clique in $G'$. This in turn implies that $N_1(v)$ and $N_2(v)$ induce cliques. $\square$

**Theorem 4.1.** *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be 2-sunflower chordal graphs. Let $G = (G_1, G_2, F)$ and let $v$ be any $S$-simplicial vertex of $G$. Then $G$ is augmentable to a chordal graph if and only if the graph $G_v = (G_1, G_2, F \cup C(v)) - v$ is augmentable to a chordal graph.*

*Proof.* Let $I$ be the set of vertices common to $G_1$ and $G_2$.

If $G_v$ is augmentable to a chordal graph, then there exists a set $A$ of augmenting edges such that $G'_v = (G_1, G_2, F \cup C(v) \cup A) - v$ is chordal. We claim that $G' = (G_1, G_2, F \cup C(v) \cup A)$ is chordal. Note that $N_{E(G')}(v) = N_1(v) \cup N_2(v)$, which forms a clique in $G'$. Thus $v$ is simplicial in $G'$. Furthermore, $G' - v = G'_v$ is chordal. This proves the claim. Thus $G$ can be augmented to a chordal graph by adding the edges $C(v) \cup A$.

To prove the other direction, assume without loss of generality that $v \in V_1$. Let $A$ be a set of augmenting edges of $G$ such that the graph $G' = (G_1, G_2, F \cup A)$ is chordal. Consider a subtree representation of $G'$. In this representation, each node $x \in V_1 \cup V_2$ is associated with a subtree $T_x$ and two nodes are adjacent in $G'$ if and only if the corresponding subtrees intersect. We now alter the subtrees as follows.

For each node $x \in N_1(v) - I$ we replace $T_x$ with $T'_x = T_x \cup T_v$. Note that $T'_x$ is a (connected) tree since $T_x$ and $T_v$ intersect. Consider the chordal graph $G''$ defined by the (intersections of) the resulting subtrees. Our goal is to show that the chordal graph $G'' - v$ is an augmentation of $G_v$, which will complete our proof. First note that $E(G'') \supseteq C(v)$ because for every $x \in N_1(v) - I$, subtree $T'_x$ intersects every subtree $T_y$ for $y \in N_{E(G')}(v)$. The only remaining thing is to show that the edges that are in $G''$ but not in $G'$ are augmenting edges, i.e. edges from $V_1 - I$ to $V_2 - I$. By construction, any edge added to $G''$ goes from some $x \in N_1(v) - I$ to some $y \in N_{E(G')}(v)$. Thus $x \in V_1 - I$, and we only need to show that $y \in V_2 - I$. Note that $(y, v)$ is an edge of $E(G')$. Now if $y \in V_1$ then $(y, v) \in E_1$ and thus $x, y$ are both in the clique $N_1(v)$ and are already joined by an edge in $G$ (and hence $G'$). Therefore $y \in V_2 - I$ and we are done. $\square$

Theorem 4.1 leads to the following algorithm for recognizing where $G_1$ and $G_2$ are simultaneous chordal graphs.

```
**Algorithm**
1. Let $G_1$ and $G_2$ be the input graphs and let $F = \emptyset$.
2. While there exists an $S$-simplicial vertex $v$ of $G = (G_1, G_2, F)$ Do
3.    $F \leftarrow F \cup C(v)$
4.    Remove $v$ and its incident edges from $G_1$, $G_2$, $F$.
5. End
6. If $G$ is empty return YES else return NO
```

Note that if $G_1$ and $G_2$ are simultaneous chordal graphs then the above algorithm can also generate an augmented chordal graph $G_a$. $G_a$ can be represented as the intersection graph of subtrees in a tree. This representation is also a simultaneous subtree representation for $G_1$ and $G_2$. We now show that the above algorithm can be implemented to run in time $O(n^3)$.

Determining whether a vertex $v$ is $S$-simplicial is a key step of the algorithm. For this we have to check whether $N_1(v)$ and $N_2(v)$ induce cliques in $G_1$ and $G_2$ respectively. Note that although the sets $N_1(x)$ and $N_2(x)$ change as we add to the edge-set $F$, the graphs $G_1$ and $G_2$ are unchanged. The straightforward implementation takes $O(n^2)$ time for this step. However we can improve this to $O(n)$ as explained below.

In a chordal graph $H$ on $n$ nodes, given a set $X \subseteq V(H)$ of vertices, we can test whether $X$ induces a clique in $O(n)$ time as follows. Let $v_1, \ldots, v_n$ be a perfect elimination order of $H$ and let $v_i$ be the first vertex in this order that is present in $X$. Then $X$ induces a clique if and only if $N(v_i) \supseteq X$. Note that computing a perfect elimination order takes linear-time [44] and hence the test takes $O(n)$ time using adjacency matrices.

Thus determining whether $v$ is $S$-simplicial takes $O(n)$ time. Since we may have to check $O(n)$ vertices before finding an $S$-simplicial vertex and since the number of iterations is $O(n)$, the algorithm runs is $O(n^3)$ time.

## 4.2 NP-Completeness for $r$-Sunflower Graphs

Since the simultaneous chordal representation problem for $r$-sunflower graphs is equivalent to the sunflower chordal augmentation problem for $r$ graphs, the problem is clearly in NP. Thus it enough to show that the problem is NP-hard. We reduce the problem of triangulating colored graphs (TCG) to our problem.

In the TCG problem, given a graph $H$ and a proper coloring function $c$ of $H$, we have to determine whether there exists a supergraph $H' \supseteq H$ such that $H'$ is chordal and $c$ is a proper coloring of $H'$. Bodlaender et al. [8] and Steel [78] have independently shown that TCG is NP-hard.

**Theorem 4.2.** *The simultaneous chordal representation problem for $r$-sunflower graphs is NP-hard, even when the common vertices induce an independent set.*

*Proof.* Let $(H, c)$ be an instance of TCG, where $H$ is a graph and $c$ is a proper coloring of $H$. Let the number of colors of $c$ be $r$, and let $C_1, C_2, \ldots, C_r$ be the color classes defined by $c$. In other words, for $i \in \{1, \ldots, r\}$ $C_i$ denotes the set of vertices in $H$ that are colored $i$. Note that $C_i$ induces an independent set.

We now create an instance of simultaneous chordal representation problem, by defining a family of $r$-sunflower graphs $G_1, \ldots, G_r$, that share a set $I$ of vertices (to be defined). For $i \in \{1, \ldots, r\}$, we define $G_i - I$ to be $C_i$. For each edge $uv$ in $H$, we create vertices $x_{uv}$ and $y_{uv}$ in $I$ and add the edges $ux_{uv}, uy_{uv}, vx_{uv}, vy_{uv}$. This completes the construction. See figure 4.1 for an example. We now show that the TCG problem for $(H, c)$ has a solution if and only if the sunflower chordal augmentation problem for $G_1, \ldots, G_r$ has a solution.

Suppose the TCG problem on $(H, c)$ has a solution. Then there exists a supergraph $H'$ of $H$, such that $H'$ is chordal and $c$ is a proper coloring of $H'$. Note that every edge $uv$ of $H'$ is an augmenting edge between vertices $u$ and $v$ of $G_1 \cup \cdots \cup G_r$, as $u$ and $v$ are colored differently by $c$. We now construct a graph $G'$ as follows. Initially, $G' = G_1 \cup \cdots \cup G_r$. For each edge $uv$ in $H'$, we add the corresponding edge between vertices $u$ and $v$ in $G'$. Note that $G' - I$ is same as $H'$ and is hence chordal. Further, every vertex of $I$ in $G'$ is a simplicial vertex, because it has two

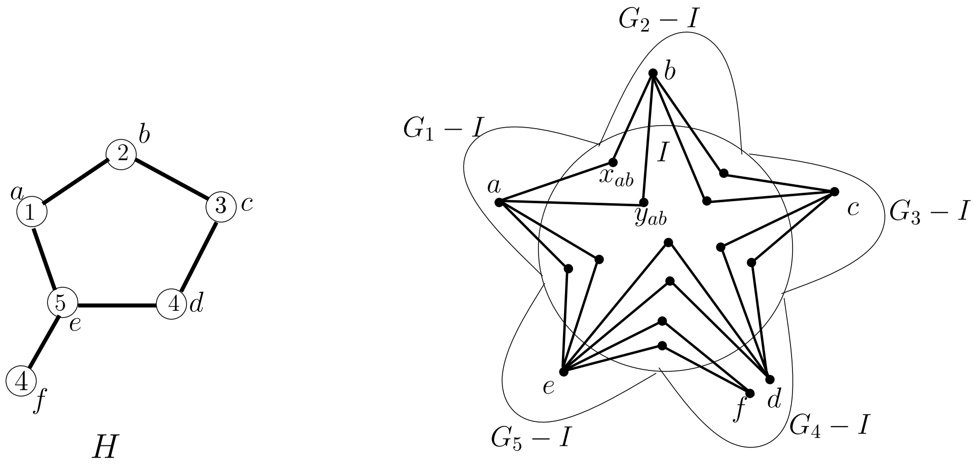


Figure 4.1: The graph $H$ to the left is an instance of TCG problem with vertices $\{a, b, c, d, e, f\}$ and a 5 coloring of the vertices. Note that vertices $f$ and $d$ are both colored 4. The graphs to the right are 5-sunflower graphs constructed from $H$.

neighbors that are joined by an edge. Thus $G'$ is a chordal graph that is obtained by augmenting $G_1, \ldots, G_r$. Thus the sunflower chordal augmentation problem for $G_1, \ldots, G_r$ has a solution.

For the other direction, assume that $G_1, \ldots, G_r$ are augmentable to a chordal graph $G'$ by adding a set $A'$ of augmenting edges. For every edge $uv$ in $H$, $G'$ contains the 4-cycle $ux_{uv}, x_{uv}v, vy_{uv}, y_{uv}u$ and hence it must contain the chord edge $uv$ ($x_{uv}y_{uv}$ is not an augmenting edge and cannot be present in $G'$). Thus every edge of $H$ is also present in $G'$ and hence $A' \supseteq E(H)$. Let $H' = G' - I$. Note that $E(H') = A'$ and hence $H'$ is a supergraph of $H$. Further every edge of $A'$ joins vertices of distinct colors and hence $c$ is a proper coloring of $H'$. Thus $H'$ is a solution to the TCG problem on $(H, c)$. This completes the proof. □

## 4.3 Open Problems

There are a few natural open problems related to this chapter: (1) Can we improve the running time of the algorithm presented in this chapter for 2-sunflower graphs to $O(n^{3-\epsilon})$ for some $\epsilon > 0$ ? or develop a faster algorithm ? (2) Can we recognize simultaneous chordal graphs when there are more than 2 graphs? In other words can we solve the simultaneous chordal representation problem for $r$-sunflower graphs for some constant $r \geq 3$? The techniques used in this chapter for 2 graphs do not seem to work for 3 graphs. In particular, the proof of Theorem 4.1 doesn't extend to three graphs.

# Chapter 5

# Simultaneous Interval Graphs

In this chapter[1], we study the simultaneous representation problem for interval graphs. Given two interval graphs sharing some vertices, we give an $O(n^2 \log n)$ algorithm to determine whether the graphs are simultaneous interval graphs. In other words, we give an efficient algorithm for solving the simultaneous interval representation for 2-sunflower graphs. This implies that the sunflower interval augmentation problem and the sunflower co-interval augmentation problem can be solved efficiently for 2-sunflower graphs. Throughout this chapter, the term "simultaneous interval graphs" refers to the version of the definition with 2 graphs.

Recall that an interval graph is defined to be the intersection graph of intervals on the real line. They are also characterized as graphs whose maximal cliques can be ordered in such a way that the cliques containing any vertex appear consecutively. The PQ-tree-based interval graph recognition algorithm (see [10]) attempts to find such an ordering of maximal cliques, by making the maximal cliques into leaves of a PQ-tree, and imposing PQ-tree constraints to ensure that the cliques containing each vertex $v$ appear consecutively. The resulting tree is called *the* PQ-tree of the graph. Note that the children of a P-node may be reordered arbitrarily and the children of a Q-node may only be reversed. We consider a node with 2 children to be a Q-node. In the figures, we use a circle to denote a P-node and a rectangle to denote a Q-node. A *leaf-order* of a PQ-tree is the order in which its leaves are visited in an in-order traversal of the tree, after children of P and Q-nodes are re-ordered as just described.

Simultaneous interval graphs arise in several settings. Some potential scenarios include: DNA structures of two organisms sharing some fragments, two conference schedules that share some plenary talks and job schedules on two machines in which certain jobs are processed synchronously. Simultaneous interval graphs may also have an application in physical mapping of DNA as a special case of the interval graph sandwich problem (see section 3.2 for details). Also our algorithm for solving the simultaneous interval representation problem for 2-sunflower graphs implies that the interval sandwich problem can be solved efficiently when the set of optional edges induce a complete bipartite graph.

We note that in the PQ-tree based solutions to probe interval graphs (see section 3.3), there is a single PQ-tree (of the graph induced by the probes) and a set of constraints imposed by the non-probes. However in our situation we have two PQ-trees, one for each graph, that we want to re-order to "match" on the common vertex set $I$. We begin by "reducing" each PQ-tree to contain only vertices from $I$. This results in PQ-trees that store non-maximal cliques, and

[1]The results presented in this chapter are joint work with Anna Lubiw [56].

our task is to modify each PQ-tree by inserting non-maximal cliques from the other tree while re-ordering the trees to make them the same.

The rest of the chapter is organized as follows. When the common graph induces an independent set, our algorithm is simple to describe and runs in linear time. We explain this special case in section 5.1. Section 5.2 gives an overview of our main algorithm. In sections 5.3, 5.4, and 5.5 we present our algorithm along with the justification of its correctness and running-time.

## 5.1 Special Case: Common Graph Induces an Independent Set

In this subsection, we present a solution for a simpler case of the simultaneous interval representation problem for sunflower graphs when the common vertices induce an independent set. We present the algorithm for 2-sunflower graphs, and later explain how it can be extended in a straightforward way to $r$-sunflower graphs (whose common vertices induce an independent set). Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be 2-sunflower graphs sharing a set $I$ of vertices (and the edges induced by $I$). Let $I$ induce an independent set.

The following is a high level overview of our approach. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the PQ-trees of $G_1$ and $G_2$ respectively. Recall the definition of PQ-tree from section 2.2. We first "restrict" the PQ-trees of $\mathcal{T}_1$ and $\mathcal{T}_2$ to $I$ and simplify them to obtain PQ-trees $\mathcal{T}_1'$ and $\mathcal{T}_2'$ respectively. Each leaf node of $\mathcal{T}_1'$ and $\mathcal{T}_2'$ corresponds to a (possibly non-maximal) clique in $I$. If $I$ doesn't induce an independent set, the leaves of $\mathcal{T}_1'$ may be different from the leaves of $\mathcal{T}_2'$ and furthermore multiple leaf nodes may correspond to the same clique. This makes the PQ-tree reductions complicated (see the next section for more details). However when $I$ induces an independent set, the cliques of $I$ are the vertices of $I$. Using this we show that $\mathcal{T}_1'$ and $\mathcal{T}_2'$ are both PQ-trees whose leaves are in 1-1 correspondence with the vertices of $I$. Then we prove that $G_1$ and $G_2$ are simultaneous interval graphs if and only if there exists a common leaf ordering in $\mathcal{T}_1'$ and $\mathcal{T}_2'$.

We now present the above approach formally. For $j = 1, 2$, given a max-clique ordering $\mathcal{S} = Q_1, Q_2, \ldots, Q_l$ of $G_j$ we can generate a realizer of $G_j$ as follows.

1. For each vertex $v$, let $Q_{l_v}$ and $Q_{r_v}$ be the first and last cliques in $\mathcal{S}$ (appearing in positions $l_v$ and $r_v$ respectively) that contain $v$. Assign the left and right end points of $v$ to $l_v$ and $r_v$ respectively.

2. Now at each index $i \in \{1, \ldots, l\}$, locally order the end points assigned to $i$ (to make them distinct) as follows. Arbitrarily permute the end points assigned to $i$ subject to the condition that all the left end points appear before the right end points.

Further, any realizer of $G_j$ can be obtained in this way by starting with an appropriate max-clique ordering. Let $\mathcal{T}_j$ be the PQ-tree of $G_j$. Note that the leaf orderings of $\mathcal{T}_j$ are the same as the max-clique orderings of $G_j$. Now let $\mathcal{S} = Q_1, Q_2, \ldots, Q_l$ be a max-clique ordering of $G_j$. Note that since $I$ is an independent set, no two vertices of $I$ are present in the same clique of $\mathcal{S}$. An ordering $\mathcal{S}_I = v_1, v_2, \ldots, v_k$ of the vertices of $I$ is said to be an *$I$-induced ordering of $\mathcal{S}$*, if for any two vertices $v_a, v_b \in I$, $v_a$ appears before $v_b$ in $\mathcal{S}_I$ if and only if all the cliques containing $v_a$ appear before all the cliques containing $v_b$ in $\mathcal{S}$.

An ordering $\mathcal{X}$ of the vertices of $I$ is said to be a *$G_j$-expandable $I$-ordering* if there exists some max-clique ordering $\mathcal{S}$ of $G_j$ such that $\mathcal{X}$ is an $I$-induced ordering of $\mathcal{S}$. The following lemma is a consequence of the above definitions.

**Lemma 5.1.** *In the case when $I$ is an independent set, $G_1$ and $G_2$ are simultaneous interval graphs if and only if there exists an ordering $\mathcal{X}$ of the vertices of $I$ such that $\mathcal{X}$ is a $G_j$-expandable $I$-ordering for $j = 1, 2$.*

Since all max-clique orderings of $G_j$ can be obtained from $\mathcal{T}_j$, all $G_j$-expandable $I$-orderings can also be obtained from $\mathcal{T}_j$. We now construct a PQ-tree $\mathcal{T}'_j$ that precisely represents all $G_j$-expandable $I$-orderings. The construction of $\mathcal{T}'_j$ from $\mathcal{T}_j$ is similar to the projection (and simplification) operation discussed in section 2.2.1. However it's a bit more complicated because in addition to deleting leaves of the PQ-tree, we also delete vertices from the cliques stored in (the leaves of) the PQ-tree.

1. Initialize $\mathcal{T}'_j$ to $\mathcal{T}_j$.
2. For each leaf $l$ of $\mathcal{T}_j$, we replace the (maximal) clique $Q_l$ that it represents with $Q_l \cap I$. Note that because $I$ is an independent set, $Q_l \cap I$ either contains a single vertex or is empty. If $Q \cap I$ is empty, we delete $v_l$. Otherwise, $l$ now represents the single vertex contained in $Q_l \cap I$.
3. We iteratively do the following. If there is a non-leaf node $n_1$ in $\mathcal{T}'_j$ such that all the children of $n_1$ are leaves and represent the same vertex, say $v$, then we replace $n_1$ with a leaf node representing $v$. Note that the parent of $n_1$ remains the same. We go to the next step if we cannot find such a non-leaf node.
4. We iteratively do the following. If there is a (non-leaf) Q-node $n_1$ in $\mathcal{T}'_j$ with two consecutive child nodes $n_a$ and $n_b$ (among others) such that $n_a$ and $n_b$ are both leaves and represent the same vertex, say $v$, then we replace $n_a$ and $n_b$ with a single leaf node representing the vertex $v$. Note that the parent of the new leaf node is $n_1$. We go to the next step if we cannot find such a non-leaf node.
5. We iteratively delete any internal node $n_1$ whose children are all empty.
6. We iteratively replace any node $n_1$ in $\mathcal{T}'_j$ with a single child with the child.

It is easy to see that the resulting tree $\mathcal{T}'_j$ is unique and can be constructed in linear time, from a bottom-up traversal of $\mathcal{T}_j$. We refer to $\mathcal{T}'_j$ as the *$I$-reduced PQ-tree* of $G_j$. Figure 5.1 gives an example of two interval graphs, their PQ-trees and $I$-reduced PQ-trees.

Note that for any leaf-ordering $\mathcal{S}$ of $\mathcal{T}_j$, there exists a leaf-ordering $\mathcal{X}$ of $\mathcal{T}'_j$ such that $\mathcal{X}$ is an $I$-induced ordering of $\mathcal{S}$. On the other hand, for any leaf-ordering $\mathcal{X}$ of $\mathcal{T}'_j$, there exists a leaf-ordering $\mathcal{S}$ of $\mathcal{T}_j$ such that $\mathcal{X}$ is an $I$-induced ordering of $\mathcal{S}$. Thus we have the following

**Lemma 5.2.** *For $j = 1, 2$, $\mathcal{T}'_j$ represents all the $G_j$-expandable $I$-orderings.*

Further, we claim that each vertex of $I$ is represented by a unique node in $\mathcal{T}'_j$. Suppose not. Then let $l_1$ and $l_2$ be two leaves of $\mathcal{T}'_j$, representing a vertex $v \in I$. Let $y$ be the least common ancestor of $l_1$ and $l_2$, and let $c_1$, $c_2$ be the child nodes of $y$ such that $c_1$ is an ancestor of $l_1$ and $c_2$ is an ancestor of $l_2$ ($c_1$ or $c_2$ could be the same as $l_1$ or $l_2$). Note that in any leaf-order of $\mathcal{T}'_j$ all leaves representing $v$ must appear consecutively. This implies that if $y$ is a P-node, then all the leaf-descendants of $y$ must represent the vertex $v$. Otherwise, it is possible to permute the children of $y$ such that in the leaf-order of the resultant tree, the nodes representing $v$ do not

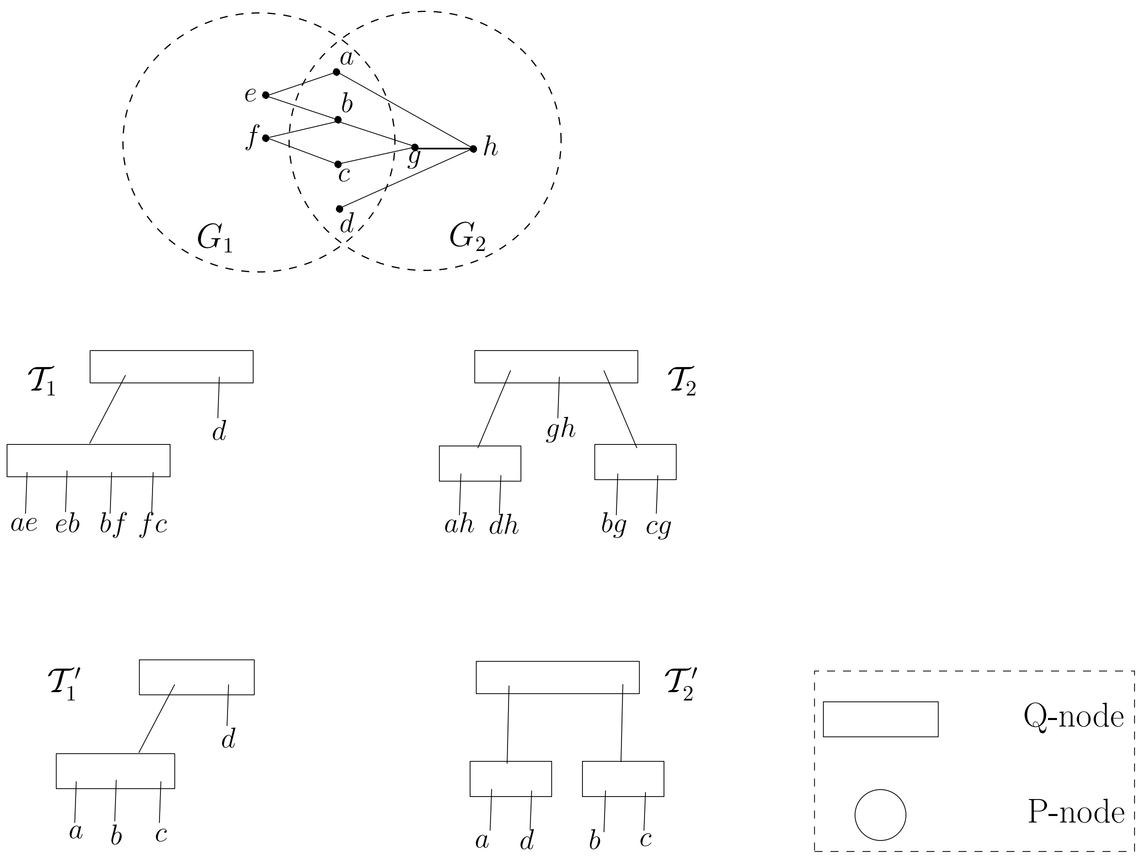


Figure 5.1: Two graphs whose common vertex set induces an independent set, their PQ-trees and $I$-reduced PQ trees.

appear contiguously. Similarly, if $y$ is a Q-node, then the leaf-descendants of $c_1$, $c_2$ and all the nodes between them must represent the single vertex $v$. Neither of these cases can happen by the construction of $\mathcal{T}_j'$. Thus each vertex of $I$ is represented by a unique node of $\mathcal{T}_j'$.

Now Lemma 5.1 and Lemma 5.2 imply that $G_1$ and $G_2$ are simultaneous interval graphs if and only if there exists a common leaf-ordering in $\mathcal{T}_1'$ and $\mathcal{T}_2'$. Note that the intersection tree $\mathcal{T}_{12}'$ of $\mathcal{T}_1'$ and $\mathcal{T}_2'$ (defined in section 2.2.1) satisfies the constraints of both the trees and hence represents all possible common leaf-orderings of $\mathcal{T}_1'$ and $\mathcal{T}_2'$. Thus testing whether $G_1$ and $G_2$ are simultaneous interval graphs is equivalent to testing whether $\mathcal{T}_{12}'$ is non-empty. As mentioned in section 2.2.1, the intersection tree can be computed in linear time. Hence we have the following theorem.

**Theorem 5.1.** *Let $G_1$ and $G_2$ be 2-sunflower interval graphs, whose common vertices induce an independent set. The problem of testing whether $G_1$ and $G_2$ are simultaneous interval graphs can be solved in linear time.*

Further given a leaf-ordering $\mathcal{X}$ of $\mathcal{T}_{12}'$, for $j = 1, 2$ it is easy to generate a leaf-ordering $\mathcal{S}_j$ of $\mathcal{T}_j$, such that $\mathcal{X}$ is a $G_j$-expandable $I$-ordering, in linear time, as follows. Travel through the nodes of $\mathcal{T}_j$ recursively in a depth-first manner and when visiting a node $d$, order the children of $d$ according to the way in which their descendants (restricted to $I$) appear in $\mathcal{X}$. From $\mathcal{S}_1$ and

$\mathcal{S}_2$, we can get interval realizers of $G_1$ and $G_2$ such that any vertex in $I$ is assigned to the same interval in both realizers.

### 5.1.1 Extension to Multiple Graphs

The algorithm described in section 5.1 can be extended in a straightforward way to solve the simultaneous interval representation problem for $r$-sunflower graphs (when they share an independent set), for arbitrary $r$. This is interesting, considering that the simultaneous chordal representation problem is NP-hard for $r$-sunflower graphs, even when the common vertices induce an independent set (see section 4.2).

Let $G_i, i \in \{1, \ldots, r\}$ be $r$-sunflower interval graphs, that share a vertex set $I$ and let $I$ induce an independent set. As before, for $i \in \{1, \ldots, r\}$, we construct the $I$-induced PQ-tree $\mathcal{T}_i'$ of $G_i$. Now the problem of testing whether $G_i, i \in \{1, \ldots, r\}$ are simultaneous interval graphs, is equivalent to testing whether there is a common leaf-ordering in all $\mathcal{T}_i', i \in \{1, \ldots, r\}$. This in turn is equivalent to testing whether the intersection tree of $\mathcal{T}_i', i \in \{1, \ldots, r\}$ is empty. The intersection PQ tree can be computed in time proportional to the sum of the sizes of the input trees. Note that the size of $\mathcal{T}_i'$ is at most the size of $G_i$. Thus we have the following theorem.

**Theorem 5.2.** *Let $G_1, G_2, \ldots, G_r$ be $r$-sunflower interval graphs whose common vertices induce an independent set. The problem of testing whether $G_i, i \in \{1, \ldots, r\}$ are simultaneous interval graphs can be solved in time proportional to the sum of the sizes of $G_i, i \in \{1, \ldots, r\}$.*

## 5.2 Overview of the General Algorithm

Our strategy for recognizing simultaneous interval graphs is similar to that of section 5.1. However the algorithm and analysis are more complicated. As before, we restrict the PQ-trees of each graph to $I$, simplify them and show that $G_1$ and $G_2$ are simultaneous interval graphs if and only if the restricted PQ-trees have "compatible" leaf orderings. But now the leaves of the PQ-trees contain (not necessarily maximal) cliques of $I$ and certain cliques can appear in multiple leaves of the tree. Furthermore, cliques that appear in one tree may not be present in the other tree. Thus we cannot use the intersection algorithm of section 2.2.1. In fact, a PQ-tree that represents all possible compatible orderings of the two PQ-trees cannot be represented as a PQ-tree.

We address this by designing an algorithm that constructs a "quasi-intersection tree" $\mathcal{T}_I$ of the two restricted PQ-trees $\mathcal{T}_1'$ and $\mathcal{T}_2'$, with the property that $\mathcal{T}_1'$ and $\mathcal{T}_2'$ have a common compatible ordering if and only if $\mathcal{T}_I'$ is non-empty. To do this, we first label the nodes of each PQ-tree based on the properties of its descendants. Our algorithm does a bottom-up traversal of the trees based on the labels and iteratively modifies each tree until they become identical. We call the resulting tree a *quasi-intersection tree* of $\mathcal{T}_1'$ and $\mathcal{T}_2'$. In each iteration we choose "unmatched" nodes $n_1$ and $n_2$ from the two trees and we either match $n_1$ with $n_2$, or reduce one of the trees. At the end of the algorithm $\mathcal{T}_1'$ and $\mathcal{T}_2'$ would either be modified to a quasi-intersection tree or we infer that $\mathcal{T}_1'$ is not compatible with $\mathcal{T}_2'$.

The following three sections contain the details of the algorithm. Section 5.3 defines the notation of compatibility and reduces the simultaneous interval representation problem to a problem on finding compatible leaf orderings. Section 5.4 labels the nodes of the PQ-tree and further simplifies them. Our algorithm and analysis are presented in section 5.5.

## 5.3 Reduction to PQ-trees

In this section we transform the simultaneous interval representation problem for 2-sunflower interval graphs to a problem about "compatibility" of two PQ-trees arising from the two graphs.

Consider an interval representation of an interval graph. For any point on the line, the intervals containing the point form a clique in the graph. This leads to the fundamental one-to-one correspondence between the interval representations of an interval graph and its *clique orderings*, defined as follows: A *clique ordering* of $G$ is a sequence of (possibly empty) cliques $\mathcal{S} = Q_1, Q_2, \ldots, Q_l$ that contains all the maximal cliques of $G$ and has the property that for each vertex $v$, the cliques in $\mathcal{S}$ that contain $v$ appear consecutively. Note that a clique can appear multiple times in a clique ordering.

Note that ignoring non-maximal cliques is fine for recognizing interval graphs; for our purposes, however, we want to consider clique orders and PQ-trees that may include non-maximal cliques. We say that a PQ-tree whose leaves correspond to cliques of a graph is *valid* if for each of its leaf orderings and for each vertex $v$, the cliques containing $v$ appear consecutively.

Let $\mathcal{S} = Q_1, Q_2, \ldots, Q_l$ be a clique ordering of interval graph $G$ and let the maximal cliques of $G$ be $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_m}$ (appearing in positions $i_1 < i_2 < \cdots < i_m$ respectively). Note that all the cliques in $\mathcal{S}$ between $Q_{i_j}$ and $Q_{i_{j+1}}$ contain $B = Q_{i_j} \cap Q_{i_{j+1}}$. We say that $B$ is the *boundary clique* or *boundary* between $Q_{i_j}$ and $Q_{i_{j+1}}$. Note that $B$ may not necessarily be present in $\mathcal{S}$. The sequence of cliques between $Q_{i_j}$ and $Q_{i_{j+1}}$ that are subsets of $Q_{i_j}$ is said to be the *right tail* of $Q_{i_j}$. The *left tail* of $Q_{i_{j+1}}$ is defined analogously. Any clique between $Q_{i_j}$ and $Q_{i_{j+1}}$ must belong to one of the two tails. Observe that the left tail of a clique forms an increasing sequence and the right tail forms a decreasing sequence (w.r.t. set inclusion). Also note that all the cliques that precede $Q_{i_1}$ are subsets of $Q_{i_1}$ and this sequence is called the left tail of $Q_{i_1}$ and all the cliques that succeed $Q_{i_m}$ are subsets of $Q_{i_m}$ and this sequence is called the right tail of $Q_{i_m}$. Thus any clique ordering of $G$ consists of a sequence of maximal cliques, with each maximal clique containing a (possibly empty) left and right tail of subcliques.

Let $Q_0$ and $Q_{l+1}$ be defined to be empty sets. An insertion of clique $Q'$ between $Q_i$ and $Q_{i+1}$ (for some $i \in \{0, \ldots, l\}$) is said to be a *subclique insertion* if $Q' \supseteq Q_i \cap Q_{i+1}$ and either $Q' \subseteq Q_i$ or $Q' \subseteq Q_{i+1}$. It is clear that after a subclique insertion the resulting sequence is still a clique ordering of $G$. A clique ordering $\mathcal{S}'$ is an *extension* of $\mathcal{S}$ if $\mathcal{S}'$ can be obtained from $\mathcal{S}$ by subclique insertions. We also say that $\mathcal{S}$ extends to $\mathcal{S}'$. Furthermore, we say that a clique ordering is *generated* by a PQ-tree, if it can be obtained from a leaf order of the PQ-tree with subclique insertions. The above definitions yield the following lemma.

**Lemma 5.3.** *A sequence of cliques $\mathcal{S}$ is a clique ordering of $G$ if and only if $\mathcal{S}$ can be generated from the PQ-tree of $G$.*

Let $G_1$ and $G_2$ be 2-sunflower graphs, that share a set $I$ of vertices (and the edges induced by $I$). Note that $G_1[I]$ is isomorphic to $G_2[I]$. A clique ordering of $G_1[I]$ is said to be an *$I$-ordering.*

The *$I$-restricted* PQ-tree of $G_j$ is defined to be the tree obtained from the PQ-tree of $G_j$ by replacing each clique $Q$ (a leaf of the PQ-tree) with the clique $Q \cap I$. Thus there is a one-to-one correspondence between the two PQ-trees, and the leaves of the $I$-restricted PQ-tree are cliques of $G_1[I]$. Note that a leaf node of an *$I$-restricted* PQ-tree may represent the empty clique (allowing such leaves to exist simplifies some of our proofs).

Let $\mathcal{I} = X_1, X_2, \ldots, X_l$ be an $I$-ordering. Then $\mathcal{I}$ is said to be $G_j$-*expandable* if there exists a clique ordering $\mathcal{O} = Q_1, Q_2, \ldots, Q_l$ of $G_j$ such that $X_i \subseteq Q_i$ for $i \in \{1, \ldots, l\}$. Further, we say that $\mathcal{I}$ expands to $\mathcal{O}$. By the definition of clique-ordering it follows that if $\mathcal{I}$ is $G_j$-expandable then it remains $G_j$-expandable after a subclique insertion (i.e. any extension of $\mathcal{I}$ is also $G_j$-expandable). We first observe the following.

**Lemma 5.4.** *The set of $G_j$-expandable $I$-orderings is same as the set of orderings that can be generated from the $I$-restricted PQ-tree of $G_j$.*

*Proof.* Let $T$ be the PQ-tree of $G_j$ and $T'$ be the $I$-restricted PQ-tree of $G_j$.

Let $\mathcal{I}$ be a $G_j$-expandable $I$-ordering of $G_j$. Then there exists a clique ordering $\mathcal{O}$ of $G_j$ such that $\mathcal{I}$ expands to $\mathcal{O}$. But by Lemma 5.3, $\mathcal{O}$ can be generated from $T$ (from a leaf order with subclique insertions). This in turn implies that $\mathcal{I}$ can be generated from $T'$ (from the corresponding leaf order with the corresponding subclique insertions).

Now for the other direction, let $\mathcal{I}' = X_1, \ldots, X_l$ be any leaf order of $T'$. Then there exists a corresponding leaf order $\mathcal{O}' = Q_1, \ldots, Q_l$ of $T$ such that $X_i \subseteq Q_i$ for $i \in \{1, \ldots, l\}$. This implies that $\mathcal{I}'$ is a $G_j$-expandable $I$-ordering. Finally, observe that if $I''$ is generated from $I'$ by subclique insertions than $I''$ is also a $G_j$-expandable $I$-ordering. Thus the lemma holds. □

Two $I$-orderings $\mathcal{I}_1$ and $\mathcal{I}_2$ are said to be *compatible* if both $\mathcal{I}_1$ and $\mathcal{I}_2$ (separately) extend to a common $I$-ordering $\mathcal{I}$. For example, the ordering $\{1\}, \{1,2\}, \{1,2,3,4\}$ is compatible with the ordering $\{1\}, \{1,2,3\}, \{1,2,3,4\}$, as they both extend to the common ordering: $\{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}$. Note that the compatibility relation is not transitive. Two PQ-trees $T_1$ and $T_2$ are said to be *compatible* if there exist orderings $\mathcal{O}_1$ and $\mathcal{O}_2$ generated from $T_1$ and $T_2$ (respectively) such that $\mathcal{O}_1$ is compatible with $\mathcal{O}_2$. The following lemma is our main tool.

**Lemma 5.5.** *$G_1$ and $G_2$ are simultaneous interval graphs if and only if the $I$-restricted PQ-tree of $G_1$ is compatible with the $I$-restricted PQ-tree of $G_2$.*

*Proof.* By Lemma 5.4, it is enough to show that $G_1$ and $G_2$ are simultaneous interval graphs if and only if there exists a $G_1$-expandable $I$-ordering $\mathcal{I}_1$ and a $G_2$-expandable $I$-ordering $\mathcal{I}_2$ such that $\mathcal{I}_1$ is compatible with $\mathcal{I}_2$. We now show this claim.

Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be as defined in the hypothesis. Since $\mathcal{I}_1$ and $\mathcal{I}_2$ are compatible, they can be extended to a common $I$-ordering $\mathcal{I}$. Let $\mathcal{I}$ expand to clique orderings $\mathcal{O}_1$ and $\mathcal{O}_2$ in $G_1$ and $G_2$ respectively. Since each vertex of $I$ appears in the same positions in both $\mathcal{O}_1$ and $\mathcal{O}_2$, it is possible to obtain interval representations $R_1$ and $R_2$ of $G_1$ and $G_2$ (from $\mathcal{O}_1$ and $\mathcal{O}_2$ respectively) such that each vertex in $I$ has the same end points in both $R_1$ and $R_2$. This implies that $G_1$ and $G_2$ are simultaneous interval graphs.

For the other direction, let $G_1$ and $G_2$ be simultaneous interval graphs. Then by Theorem 2.1, there exists an augmenting set of edges $A' \subseteq V_1 - I \times V_2 - I$ such that $G = G_1 \cup G_2 \cup A'$ is an interval graph. Let $\mathcal{O} = Q_1, Q_2, \ldots, Q_l$ be a clique-ordering of $G$. For each $i \in \{1, \ldots, l\}$ and $j \in \{1, 2\}$, by restricting $Q_i$ to $V_j$ (i.e. replacing $Q_i$ with $Q_i \cap V_j$), we obtain a clique ordering $\mathcal{O}_j$ of $G_j$. Now for $j \in 1, 2$, let $\mathcal{I}_j$ be the $I$-ordering obtained from $\mathcal{O}_j$ by restricting each clique in $\mathcal{O}_j$ to $I$. It follows that $\mathcal{I}_1$ is a $G_1$-expandable $I$-ordering and $\mathcal{I}_2$ is a $G_2$-expandable $I$-ordering. Further, $\mathcal{I}_1 = \mathcal{I}_2$ and hence $\mathcal{I}_1$ and $\mathcal{I}_2$ are compatible. □

Our algorithm will decide if the $I$-restricted PQ-tree of $G_1$ is compatible with the $I$-restricted PQ-tree of $G_2$. We first show how the $I$-restricted PQ-trees can be simplified in several ways. Two $I$-orderings $\mathcal{I}_1$ and $\mathcal{I}_2$ are said to be *equivalent* if, for any $I$-ordering $\mathcal{I}'$, $\mathcal{I}_1$ and $\mathcal{I}'$ are compatible if and only if $\mathcal{I}_2$ and $\mathcal{I}'$ are compatible. Note that this is an equivalence relation. The lemma below follows directly from the definitions of equivalent orderings and subclique insertions.

**Lemma 5.6.** *Let $\mathcal{I} = X_1, X_2, \ldots, X_l$ be an $I$-ordering in which $X_i = X_{i+1}$ for some $i \in 1, \ldots, l-1$. Let $\mathcal{I}'$ be the $I$-ordering obtained from $\mathcal{I}$ by deleting $X_{i+1}$. Then $\mathcal{I}$ is equivalent to $\mathcal{I}'$.*

Further, because equivalence is transitive, Lemma 5.6 implies that an $I$-ordering $\mathcal{I}$ is equivalent to the $I$-ordering $\mathcal{I}'$ in which all consecutive duplicates are eliminated. This allows us to simplify the $I$-restricted PQ-tree of $G_j$. Let $T$ be the $I$-restricted PQ-tree of $G_j$. We obtain a PQ-tree $T'$ from $T$ as follows.

1. Initialize $T' = T$.
2. As long as there is a non-leaf node $d$ in $T'$ such that all the descendants of $d$ are the same, i.e. they are all duplicates of a single clique $X$, replace $d$ and the subtree rooted at $d$ by a leaf node representing $X$.
3. As long as there is a (non-leaf) Q-node $d$ in $T'$ with two consecutive child nodes $n_a$ and $n_b$ (among others) such that all the descendants of $n_a$ and $n_b$ are the same i.e. they are all duplicates of a single clique $X$, replace $n_a$, $n_b$ and the subtrees rooted at these vertices by a single leaf node representing the clique $X$.

Note that the resulting $T'$ is unique. We call $T'$ the $I$-reduced PQ-tree of $G_j$.

**Lemma 5.7.** *$G_1$ and $G_2$ are simultaneous interval graphs if and only if the $I$-reduced PQ-tree of $G_1$ is compatible with the $I$-reduced PQ-tree of $G_2$.*

*Proof.* For $j \in \{1, 2\}$, let $T_j$ and $T'_j$ be the $I$-restricted and $I$-reduced PQ-trees of $G_j$ respectively. Let $\mathcal{I}$ be any $I$-ordering. Observe that by Lemma 5.6, $\mathcal{I}$ is compatible with a leaf ordering of $T_j$ if and only if $\mathcal{I}$ is compatible with a leaf ordering of $T'_j$. Thus the conclusion follows from Lemma 5.5. □

## 5.4 Labeling and Further Simplification

In section 2, we transformed the simultaneous interval recognition problem to a problem of testing compatibility of two $I$-reduced PQ-trees where $I$ is the common vertex set of the two graphs. These PQ-trees may have nodes that correspond to non-maximal cliques in $I$. In this section we prove some basic properties of such $I$-reduced PQ-trees, and use them to further simplify each tree.

Let $\mathcal{T}$ be the $I$-reduced PQ-tree of $G_j$. Recall that each leaf $l$ of $\mathcal{T}$ corresponds to a clique $X$ in $G_j[I]$. If $X$ is maximal in $I$, then $X$ is said to be a *max-clique* and $l$ is said to be a *max-clique node*, otherwise $X$ is said to be a *subclique* and $l$ is said to be a *subclique node*. When the association is clear from the context, we will sometimes refer to a leaf $l$ and its corresponding clique $X$ interchangeably, or interchange the terms "max-clique" and "max-clique node" [resp. subclique and subclique node]. A node of $\mathcal{T}$ is said to be an *essential node* if it is a non-leaf node or if it is a leaf node representing a max-clique.

Given a node $d$ of $\mathcal{T}$, the *descendant cliques* of $d$ are the set of cliques that correspond to the leaf-descendants of $d$. Because our algorithm operates by inserting subcliques from one tree into the other, we must take care to preserve the validity of a PQ-tree. For this we need to re-structure the tree when we do subclique insertions. The required restructuring will be determined based on the label $U(d)$ that we assign to each node $d$ as follows.

$U(d)$ or the *Universal set* of $d$ is defined as the set of vertices $v$ such that $v$ appears in all descendant cliques of $d$.

Note that for a leaf node $l$ representing a clique $X$, $U(l) = X$ by definition. Also note that along any path up the tree, the universal sets decrease. The following lemma gives some useful properties of the $I$-reduced PQ-tree.

**Lemma 5.8.** *Let $\mathcal{T}$ be the $I$-reduced PQ-tree of $G_j$. Let $d$ be a non-leaf node of $\mathcal{T}$ ($d$ is used in properties 2–6). Then we have:*
**0.** *Let $l_1$ and $l_2$ be two distinct leaf nodes of $\mathcal{T}$, containing a vertex $t \in I$. Let $y$ be the least common ancestor of $l_1$ and $l_2$. Then: (a) If $y$ is a P-node then all of its descendant cliques contain $t$. (b) If $y$ is a Q-node then $t$ is contained in all the descendant cliques of all children of $y$ between (and including) the child of $y$ that is the ancestor of $l_1$ and the child that is the ancestor of $l_2$.*
**1.** *Each max-clique is represented by a unique node of $\mathcal{T}$.*
**2.** *A vertex $u$ is in $U(d)$ if and only if for every child $n_1$ of $d$, $u \in U(n_1)$.*
**3.** *$d$ contains a max-clique as a descendant.*
**4.** *If $d$ is a P-node, then for any two child nodes $n_1$ and $n_2$ of $d$, we have $U(n) = U(n_1) \cap U(n_2)$.*
**5.** *If $d$ is a P-node, then any child of $d$ that is a subclique node represents the clique $U(d)$.*
**6.** *If $d$ is a Q-node and $n_1$ and $n_2$ are the first and last child nodes of $d$ then $U(n) = U(n_1) \cap U(n_2)$.*

*Proof.* (0) Observe that in any leaf ordering of $\mathcal{T}$, all the nodes that appear between $l_1$ and $l_2$ must also contain the vertex $t$, otherwise $\mathcal{T}$ would be invalid. Now let $l_3$ be a leaf descendant of $y$ that doesn't contain $t$.

If $y$ is a P-node, then we can reorder the children of $y$ in such a way that in the leaf-ordering of the resulting tree $l_3$ appears between $l_1$ and $l_2$. But this contradicts the validity of $\mathcal{T}$. This proves (a). Similarly, if $y$ is a Q-node, then $l_3$ cannot be equal to $l_1$ or $l_2$ or any node between them. Thus (b) also holds.

(1) Note that by definition of $I$-reduced PQ-tree of $G_j$, each max-clique must be present in $\mathcal{T}$. Now assume for the sake of contradiction that a max-clique $X$ is represented by two leaf nodes, say $l_1$ and $l_2$. Let $y$ be the least common ancestor of $l_1$ and $l_2$. Let $c_1$ and $c_2$ be the child nodes of $y$ that contain $n_1$ and $n_2$ (respectively) as descendants. Now by (0), if $y$ is a P-node then all of its descendant cliques must contain all the vertices of $X$. But as $X$ is maximal, all these cliques must be precisely $X$. However this is not possible, as we would have replaced $y$ with a leaf node representing $X$ in the construction of $\mathcal{T}$. Similarly, if $y$ is a Q-node then the descendant cliques

of $c_1$, $c_2$ and all the nodes between them must represent the max-clique $X$. But then we would have replaced these nodes with a leaf node representing $X$ in the construction of $\mathcal{T}$. This proves (1).

(2) If $u \in U(n)$, then all the descendant cliques of $d$ contain $u$. This implies that for any child $n_1$ of $d$, all the descendant cliques of $n_1$ contain $u$. Hence $u \in U(n_1)$. On the other hand, if each child $n_1$ of $d$ contains a vertex $u$ in its universal set, then $u$ is present in all the descendant cliques of $d$ and thus $u \in U(n)$.

(3) Note that if each descendant clique of $d$ contains precisely $U(d)$ (and no other vertex), then we would have replaced the subtree rooted at $d$ with a leaf node corresponding to the clique $U(d)$, when constructing $\mathcal{T}$. Thus there exists a clique $Q_2$ that is a descendant of $d$, such that $Q_2 - U(n)$ is non-empty. If $Q_2$ is a max-clique then we are done. Otherwise let $t \in Q_2 - U(n)$ and let $Q_1$ be a max-clique containing $t$. Suppose $Q_1$ is a not a descendant of $n_1$. Applying (0) on $Q_1$ and $Q_2$, we infer that irrespective of whether $d$ is a P-node or a Q-node, all the descendant cliques of $d$ must contain $t$. But then $t \in U(n)$, a contradiction. Thus $Q_1$ is a descendant of $n_1$.

(4) By (2) we observe that $U(n) \subseteq U(n_1) \cap U(n_2)$. Thus it is enough to show that $U(n_1) \cap U(n_2) \subseteq U(n)$. Let $u \in U(n_1) \cap U(n_2)$, then $u$ is present in all the descendant cliques of $n_1$ and $n_2$. By (0), $u$ must be present in all the descendant cliques of $d$ and hence $u \in U(n)$. Therefore $U(n_1) \cap U(n_2) \subseteq U(n)$.

(5) Consider any child $n_1$ of $d$. Suppose $n_1$ is a leaf-node and is not a max-clique. It is enough to show that $n_1$ represents the clique $U(d)$, i.e., $U(n_1) = U(n)$. Suppose not. Then there exists a vertex $t \in U(n_1) - U(n)$. Let $Q_1$ be a max-clique containing $t$. Note that the common ancestor of $Q_1$ and $n_1$ is either $d$ or an ancestor of $d$. Applying (0) on $Q_1$ and $n_1$, we infer that all the descendant cliques of $d$ must contain $t$. But then $t \in U(n)$, a contradiction.

(6) This follows from (2) and (0). □

Let $\mathcal{T}$ be the $I$-reduced PQ-tree of $G_j$. Recall that an essential node is a non-leaf node or a leaf node representing a maximal clique. Equivalently (by Lemma 5.8.3), an essential node is a node which contains a max-clique as a descendant. The following lemma shows that in some situations we can obtain an equivalent tree by deleting subclique child nodes of a P-node $d$. Recall that by Lemma 5.8.5, such subclique nodes represent the clique $U(d)$.

**Lemma 5.9.** *Let $\mathcal{T}$ be the $I$-reduced PQ-tree of $G_j$ and $d$ be a P-node in $\mathcal{T}$. Then*
**1.** *If $d$ has at least two essential child nodes, then $\mathcal{T}$ is equivalent to the tree $\mathcal{T}'$, obtained from $\mathcal{T}$ by deleting all the subclique children of $d$.*
**2.** *If $d$ has at least two subclique child nodes, then $\mathcal{T}$ is equivalent to the tree $\mathcal{T}'$, obtained from $\mathcal{T}$ by deleting all except one of the subclique children of $d$.*

*Proof.* We give the proof of (1) below. The proof of (2) is very similar and hence omitted.

Let $\mathcal{O}_1$ be any $I$-ordering. It is enough to show that there exists a leaf ordering $\mathcal{O}$ of $\mathcal{T}$ that is compatible with $\mathcal{O}_1$ if and only if there exists a leaf ordering $\mathcal{O}'$ of $\mathcal{T}'$ that is compatible with $\mathcal{O}_1$.

Let $\mathcal{O}$ be any leaf ordering of $\mathcal{T}$, compatible with $\mathcal{O}_1$. Consider the ordering $\mathcal{O}'$ obtained from $\mathcal{O}$ by deleting the cliques $U(d)$ that correspond to the (subclique) child nodes of $d$ in $\mathcal{T}$. Clearly $\mathcal{O}'$ is a leaf ordering of $\mathcal{T}'$. Further $\mathcal{O}'$ can be extended to $\mathcal{O}$ by adding copies of the cliques $U(d)$ at appropriate positions. Thus $\mathcal{O}'$ is compatible with $\mathcal{O}_1$.

Now for the other direction, let $\mathcal{O}'$ be a leaf order of $\mathcal{T}'$, compatible with $\mathcal{O}_1$ and let $\mathcal{O}'$ and $\mathcal{O}_1$ extend to a common ordering $\mathcal{O}_F$. From the hypothesis, we can assume that there exist two essential child nodes $n_1$ and $n_2$ of $d$ in $\mathcal{T}'$ such that the clique descendants of $n_1$ appear immediately before the clique descendants of $n_2$ in $\mathcal{O}'$. Also let $S(n_1)$ and $S(n_2)$ be the two subsequences of $\mathcal{O}'$ containing the clique descendants of $n_1$ and $n_2$ respectively. Since $n_1$ and $n_2$ are essential nodes, $S(n_1)$ and $S(n_2)$ each contain at least one max-clique. Let $Q_1$ be the last max-clique in $S(n_1)$ and $Q_2$ be the first max-clique in $S(n_2)$. By Lemma 5.8.0, $Q_1 \cap Q_2 = U(n_1) \cap U(n_2) = U(n)$. Since $\mathcal{O}'$ is compatible with $\mathcal{O}_1$, in each of the two orderings $\mathcal{O}'$ and $\mathcal{O}_1$, $Q_2$ occurs after $Q_1$ and no other max-clique appears between them. Further the same holds for $\mathcal{O}_F$ (as it is an extension of $\mathcal{O}'$). Let $k$ be the number of subclique children of $d$ (that represent the clique $U(d)$). Then obtain a leaf ordering $\mathcal{O}$ of $\mathcal{T}$, from $\mathcal{O}'$, by inserting $k$ copies of $U(d)$ between $S(n_1)$ and $S(n_2)$. Now extend $\mathcal{O}_F$ to $\mathcal{O}'_F$ by inserting $k$ copies of $U(d)$ between $Q_1$ and $Q_2$ (there is a unique way of adding a subclique between two max-cliques). It is clear that $\mathcal{O}'_F$ is an extension of both $\mathcal{O}$ and $\mathcal{O}_1$. Therefore $\mathcal{O}$ is compatible with $\mathcal{O}_1$. This proves (1). $\square$

We will simplify $\mathcal{T}$ as much as possible by applying Lemma 5.9 and by converting nodes with two children into Q-nodes. We call the end result a *simplified* $I$-reduced PQ-tree, but continue to use the term "$I$-reduced PQ-tree" to refer to it. Note that the simplification process does not change the universal sets and preserves the validity of the PQ-tree, so Lemma 5.7 and all the properties given in Lemma 5.8 still hold. Because we consider nodes with 2 children as Q-nodes Lemma 5.9 implies:

**Corollary 5.1.** *In a [simplified] $I$-reduced PQ-tree, any P-node has at least 3 children, and all the children are essential nodes.*

## 5.5 Algorithm

For $k \in \{1, 2\}$, let $\mathcal{T}_k$ be the [simplified] $I$-reduced PQ-tree of $G_k$. By Lemma 5.7, testing whether $G_1$ and $G_2$ are simultaneous interval graphs is equivalent to testing whether $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible. We test this by modifying $\mathcal{T}_1$ and $\mathcal{T}_2$ (e.g. inserting the sub-clique nodes from one tree into the other) so as to make them identical, without losing their compatibility. The following is a high level overview of our approach for checking whether $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible.

Our algorithm is iterative and tries to *match* essential nodes of $\mathcal{T}_1$ with essential nodes of $\mathcal{T}_2$ in a bottom-up fashion. An essential node $n_1$ of $\mathcal{T}_1$ is matched with an essential node $n_2$ of $\mathcal{T}_2$ if and only if the subtrees rooted at $n_1$ and $n_2$ are the same, i.e., their essential children are matched, their subclique children are the same and furthermore (in the case of Q-nodes) their child nodes appear in the same order. If $n_1$ is matched with $n_2$ then we consider $n_1$ and $n_2$ to be identical and use the same name (say $n_1$) to refer to either of them. Initially, we match each max-clique node of $\mathcal{T}_1$ with the corresponding max-clique node of $\mathcal{T}_2$. Note that every max-clique node appears uniquely in each tree by Lemma 5.8.1. A sub-clique node may appear in only one tree in which case we must first insert it into the other tree. This is done when we consider the parent of the subclique node.

In each iteration, we either match an unmatched node $u$ of $\mathcal{T}_1$ to an unmatched node $v$ of $\mathcal{T}_2$ (which may involve inserting subclique child nodes of $v$ as child nodes of $u$ and vice versa) or we *reduce* either $\mathcal{T}_1$ or $\mathcal{T}_2$ without losing their compatibility relationship. *Reducing* a PQ-tree means restricting it to reduce the number of leaf orderings. Finally, at the end of the algorithm either we have modified $\mathcal{T}_1$ and $\mathcal{T}_2$ to a "common" tree $\mathcal{T}_I$ that establishes their compatibility or we conclude that $\mathcal{T}_1$ is not compatible with $\mathcal{T}_2$. The common tree $\mathcal{T}_I$ is said to be a *quasi-intersection tree* (of $\mathcal{T}_1$ and $\mathcal{T}_2$) and has the property that any ordering generated by $\mathcal{T}_I$ can also be generated by $\mathcal{T}_1$ and $\mathcal{T}_2$. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, there may be several quasi-intersection trees of $\mathcal{T}_1$ and $\mathcal{T}_2$, but our algorithm finds only one of them.

We need the following additional notation for the rest of this Chapter. A sequence of subcliques $\mathcal{S} = X_1, X_2, \ldots, X_l$ is said to satisfy the *subset property* if $X_i \subseteq X_{i+1}$ for $i \in \{1, \ldots, l-1\}$. $\mathcal{S}$ is said to satisfy the *superset property* if $X_i \supseteq X_{i+1}$ for each $i$. Note that $S$ satisfies the subset property if and only if $\bar{\mathcal{S}} = X_l, \ldots, X_2, X_1$ satisfies the superset property.

Let $d$ be an essential child node of a Q-node in $\mathcal{T}_k$. We will overload the term "tail" (previously defined for a max clique in a clique ordering) and define the *tails* of $d$ as follows. The left tail (resp. right tail) of $d$ is defined as the sequence of subcliques that appear as siblings of $d$, to the immediate left (resp. right) of $d$, such that each subclique is a subset of $U(d)$. Note that the left tail of $d$ should satisfy the subset property and the right tail of $d$ should satisfy the superset property (otherwise $\mathcal{T}_k$ will not be valid). Also note that since the children of a Q-node can be reversed in order, "left" and "right" are relative to the child ordering of the Q-node. We will be careful to use "left tail" and "right tail" in such a way that this ambiguity does not matter. Now suppose $d$ is a matched node. Then, in order to match the parent of $d$ in $\mathcal{T}_1$ with the parent of $d$ in $\mathcal{T}_2$, our algorithm has to "merge" the tails of $d$.

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two subclique sequences that satisfy the subset property. Then $\mathcal{L}_1$ is said to be *mergable* with $\mathcal{L}_2$ if the union of subcliques in $\mathcal{L}_1$ and $\mathcal{L}_2$ can be arranged into an ordering $\mathcal{L}'$ that satisfies the subset property. Analogously, if $\mathcal{L}_1$ and $\mathcal{L}_2$ satisfy the superset property, then they are said to be mergable if the union of their subcliques can be arranged into an ordering $\mathcal{L}'$ that satisfies the superset property. In both cases, $\mathcal{L}'$ is said to be the *merge* of $\mathcal{L}_1$ and $\mathcal{L}_2$ and is denoted by $\mathcal{L}_1 + \mathcal{L}_2$.

A *maximal matched node* is a node that is matched but whose parent is not matched. For an unmatched essential node $x$, the *MM-descendants* of $x$, denoted by $MMD(x)$ are its descendants that are maximal matched nodes. If $x$ is matched then we define $MMD(x)$ to be the singleton set containing $x$. Note that the set of MM-descendants of an essential node is non-empty (since every essential node has a max-clique descendant).

Our algorithm matches nodes from the leaves up, and starts by matching the leaves that are max-cliques. As the next node $n_1$ that we try to match, we want an unmatched node whose essential children are already matched. To help us choose between $\mathcal{T}_1$ and $\mathcal{T}_2$, and also to break ties, we prefer a node with larger $U$ set. Then, as a candidate to match $n_1$ to, we want an unmatched node in the other tree that has some matched children in common with $n_1$. With this intuition in mind, our specific rule is as follows.

Among all the unmatched essential nodes of $\mathcal{T}_1 \cup \mathcal{T}_2$ choose $n_1$ with maximal $U(n_1)$, minimal $MMD(n_1)$, and maximal depth, in that preference order. Assume without loss of generality that $n_1 \in \mathcal{T}_1$. Select an unmatched node $n_2$ from $\mathcal{T}_2$ with maximal $U(n_2)$, minimal $MMD(n_2)$ and maximal depth (in that order) satisfying the property that $MMD(n_1) \cap MMD(n_2) \neq \emptyset$. The following lemma captures certain properties of $n_1$ and $n_2$, including why these rules match our intuitive justification.

**Lemma 5.10.** *For $n_1$ and $n_2$ chosen as described above, let $M_1 = MMD(n_1)$, $M_2 = MMD(n_2)$ and $X = M_1 \cap M_2$. Also let $C_1$ and $C_2$ be the essential child nodes of $n_1$ and $n_2$ respectively. Then we have:*
**1.** *$M_1 = C_1$ and $X \subseteq C_2$.*
*Further when $\mathcal{T}_1$ is compatible with $\mathcal{T}_2$, we have:*
**2.** *For every (matched) node $l$ in $M_1 - X$ of $\mathcal{T}_1$, its corresponding matched node $l'$ in $\mathcal{T}_2$ is present outside the subtree rooted at $n_2$. Analogously, for every (matched) node $r'$ in $M_2 - X$ of $\mathcal{T}_2$, its corresponding matched node $r$ in $\mathcal{T}_1$ is present outside the subtree rooted at $n_1$.*
**3.** *If $n_1$ [resp. $n_2$] is a Q-node, then in its child ordering, no node of $C_1 - X$ [resp. $C_2 - X$] can be present between two nodes of $X$.*
**4.** *If $n_1$ and $n_2$ are Q-nodes, then in the child ordering of $n_1$ and $n_2$, nodes of $X$ appear in the same relative order i.e. for any three nodes $x_1, x_2, x_3 \in X$, $x_1$ appears between $x_2$ and $x_3$ in the child ordering of $n_1$ if and only if $x_1$ also appears between $x_2$ and $x_3$ in the child ordering of $n_2$.*
**5.** *If $C_1 - X$ [resp. $C_2 - X$] is non-empty then $U(n_1) \subseteq U(n_2)$ [resp. $U(n_2) \subseteq U(n_1)$]. Further, if $C_1 - X$ is non-empty then so is $C_2 - X$ and hence $U(n_1) = U(n_2)$.*
**6.** *Let $C_1 - X$ be non-empty. If $n_1$ [resp. $n_2$] is a Q-node, then in its child-ordering either all nodes of $C_1 - X$ [resp. $C_2 - X$] appear before the nodes of $X$ or they all appear after the nodes of $X$.*

*Proof.* (1) If there exists an unmatched child $c$ of $n_1$, then as $U(c) \supseteq U(n_1)$, $MMD(c) \subseteq MMD(n_1)$ and $c$ has a greater depth than $n_1$, we would have chosen $c$ over $n_1$. Thus every node in $C_1$ is matched and hence by the definition of MM-descendants $C_1 = M_1$.

For the second part, suppose there exists a node $x \in X$ that is not a child of $n_2$. Let $c_2$ be the child of $n_2$ that contains $x$ as a descendant. $c_2$ must be an unmatched node. (Otherwise $MMD(n_2)$ would have contained $c_2$ and not $x$). But then we would have picked $c_2$ over $n_2$.

(2) Let $l$ be a (matched) node in $M_1 - X$ (in $\mathcal{T}_1$) such that the corresponding matched node $l'$ in $\mathcal{T}_2$ is a descendant of $n_2$. Note that $l'$ cannot be a child of $n_2$. Otherwise $l' \in M_2$ and thus $l = l'$ is in $X$. Let $p'$ be the parent of $l'$. Now $p'$ cannot be a matched node. (Otherwise $p'$ would have been matched to $n_1$, a contradiction since $n_1$ is unmatched). Also $p'$ is a descendant of $n_2$ and hence $U(p') \supseteq U(n_2)$, $MMD(p') \subseteq MMD(n_2)$ and $p'$ has greater depth than $n_2$. Further $l = l'$ is a common MM-descendant of $n_1$ and $p'$. This contradicts the choice of $n_2$.

Now let $r'$ be a (matched) node in $M_2 - X$ (in $\mathcal{T}_2$), such that the corresponding matched node $r$ is a descendant of $n_1$. Note that $r$ is not a child of $n_1$, otherwise $r = r'$ is a common MM-descendant of $n_1$ and $n_2$ and hence $r' = r \in X$. Let $p$ be the parent of $r$ in $\mathcal{T}_1$. Since $p$ is a proper descendant of $n_1$, $p$ is a matched node. Let $p$ be matched to a node $p'$ in $\mathcal{T}_2$. Now $p'$ is a parent of $r'$ and a descendant of $n_2$. But then the MM-descendants of $n_2$ should not have contained $r'$.

(3) Suppose in the child ordering of $n_1$, node $y \in C_1 - X$ is present between nodes $x_a \in X$ and $x_b \in X$. Let $Y, X_a$ and $X_b$ be any max-cliques that are descendants of $y, x_a$ and $x_b$ respectively. Then in any ordering of $\mathcal{T}_1$, $Y$ appears between $X_a$ and $X_b$. But by (2), the corresponding matched node $y'$ of $y$ in $\mathcal{T}_2$ appears outside the subtree rooted at $n_2$. Thus in any ordering of $\mathcal{T}_2$, $Y$ appears either before or after both $X_a$ and $X_b$. Thus $\mathcal{T}_1$ and $\mathcal{T}_2$ are not compatible. This shows the claim for $n_1$. The proof for $n_2$ is similar.

(4) This follows from the fact that $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible and by observing that each matched node (in particular any node in $X$) contains a max-clique as a descendant.

(5) Let $x_a \in X$ be a common child of $n_1$ and $n_2$. Let $X_a$ be a max-clique descendant of $x_a$. Suppose $C_1 - X$ is non-empty. Then let $Y_a$ be any max-clique descendant of a node in $C_1 - X$. Note that by (2), $Y_a$ is present outside the subtree rooted at $n_2$. Now by Lemma 5.8.0 and observing that the least common ancestor of $X_a$ and $Y_a$ (in $\mathcal{T}_2$) is an ancestor of $n_2$, we get $U(n_2) \supseteq X_a \cap Y_a \supseteq U(n_1)$. Thus $U(n_1) \subseteq U(n_2)$. Using an analogous argument we can show that if $C_2 - X$ is non-empty then $U(n_2) \subseteq U(n_1)$. This proves the first part of the property.

For the second part we once again assume that $C_1 - X$ is non-empty and hence $U(n_1) \subseteq U(n_2)$. Now if $C_2 - X$ is empty then $MMD(n_2) = X \subset MMD(n_1)$. But this contradicts the choice of $n_1$ (we would have selected $n_2$ instead).

(6) By (5), $C_2 - X$ is non-empty and $U(n_1) = U(n_2)$. Let $x_a \in X$ and suppose $y_a, y_b \in C_1 - X$ are any two nodes on different sides of $X$. Let $z_a \in C_2 - X$. Note that by (2), the matched nodes of $y_a, y_b$ in $\mathcal{T}_2$ appear outside the subtree rooted at $n_2$ and the matched node of $z_a$ in $\mathcal{T}_1$ appears outside the subtree rooted at $n_1$. Now let $X_a, Y_a, Y_b$ and $Z_a$ be any descendant max-cliques of $x_a, y_a, y_b$ and $z_a$ respectively. In any leaf-ordering of $\mathcal{T}_1$, $X_a$ appears between $Y_a$ and $Y_b$, and $Z_a$ doesn't appear between $Y_a$ and $Y_b$. But in any leaf-ordering of $\mathcal{T}_2$, either $Z_a$ and $X_a$ both appear between $Y_a$ and $Y_b$ or they both appear before or after $Y_a$ and $Y_b$. This contradicts that $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible. Therefore all nodes of $X$ appear before or after all nodes of $C_1 - X$ in the child ordering of $n_1$. Similarly, the claim also holds for the child ordering of $n_2$ in $\mathcal{T}_2$.

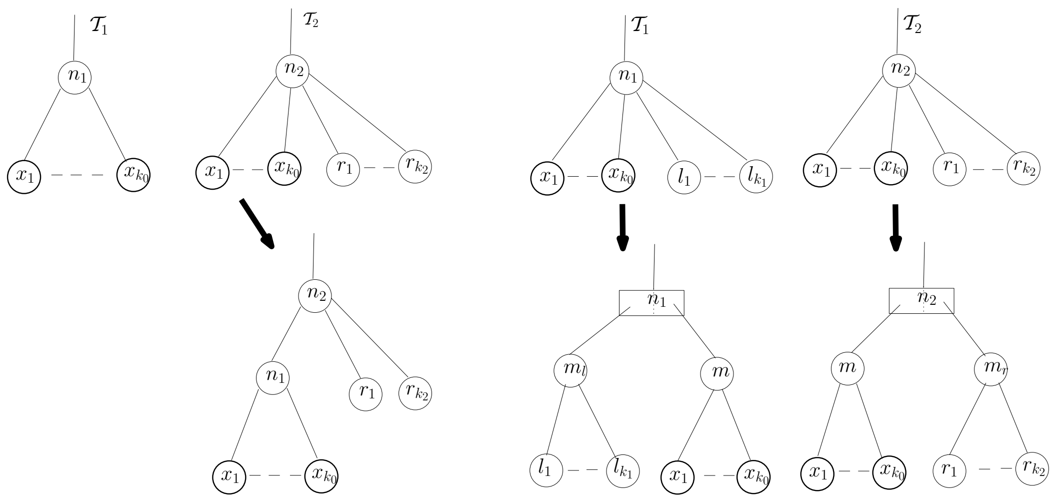
$\square$

We now describe the main step of the algorithm. Let $n_1, n_2, M_1, M_2, C_1, C_2$ and $X$ be as defined in the above lemma. We have four cases depending on whether $n_1$ and $n_2$ are P or Q-nodes. In each of these cases, we make progress by either matching two previously unmatched essential nodes of $\mathcal{T}_1$ and $\mathcal{T}_2$ or by reducing $\mathcal{T}_1$ and/or $\mathcal{T}_2$ at $n_1$ or $n_2$ while preserving their compatibility. We show that our algorithm requires at most $O(n \log n)$ iterations and each iteration takes $O(n)$ time. Thus our algorithm runs in $O(n^2 \log n)$ time.

During the course of the algorithm we may also insert subcliques into a Q-node when we are trying to match it to another Q-node. This is potentially dangerous as it may destroy the validity of the PQ-tree. When the Q-nodes have the same universal set, this trouble does not arise. However, in case the two Q-nodes have different universal sets, we need to re-structure the trees. Case 4, when $n_1$ and $n_2$ are both Q-nodes, has subcases to deal with these complications.

## Case 1: $n_1$ and $n_2$ are both P-nodes.

By Corollary 5.1, the children of $n_1$ and $n_2$ are essential nodes, so $C_1$ and $C_2$ are precisely the children of $n_1$ and $n_2$ respectively. Let $X$ consist of nodes $\{x_1, \ldots, x_{k_0}\}$. If $C_2 - X$ is empty, then by Lemma 5.10.5, $C_1 - X$ is also empty and hence $n_1$ and $n_2$ are the same. So we match $n_1$ with $n_2$ and go to the next iteration. Suppose now that $C_2 - X$ is non empty. Let $C_2 - X = \{r_1, \ldots, r_{k_2}\}$. If $C_1 - X$ is empty, then we use the reduction template of Figure 5.2(a) to modify $\mathcal{T}_2$, matching the new parent of $X$ in $\mathcal{T}_2$ to $n_1$. It is easy to see that $\mathcal{T}_1$ is compatible with $\mathcal{T}_2$ if and only if $\mathcal{T}_1$ is compatible with the modified $\mathcal{T}_2$.



(a) $C_1 - X$ is empty

(b) $C_1 - X$ is non-empty

Figure 5.2: Reduction templates for Case 1.

Now let $C_1 - X = \{l_1, \ldots, l_{k_1}\}$ be non-empty. In this case we use the reduction template of Figure 5.2(b) to modify $\mathcal{T}_1$ and $\mathcal{T}_2$ to $\mathcal{T}_1'$ and $\mathcal{T}_2'$ respectively. Note that it is possible to have $k_i = 1$ for some $i$'s, in which case the template is slightly different because we do not make a node with one child, however, the reduction always makes progress as each $n_i$ has at least 3 children.

We now claim that $\mathcal{T}_1$ is compatible with $\mathcal{T}_2$ if and only if $\mathcal{T}_1'$ is compatible with $\mathcal{T}_2'$. The reverse direction is trivial. For the forward direction, let $\mathcal{O}_1$ and $\mathcal{O}_2$ be two compatible leaf orderings of $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively. Recall that by Lemma 5.10.2, for every [matched] node of $C_1 - X$ in $\mathcal{T}_1$, the corresponding matched node in $\mathcal{T}_2$ appears outside the subtree rooted at $n_2$. This implies that the descendant nodes of $\{x_1, x_2, \ldots, x_{k_0}\}$ all appear consecutively in $\mathcal{O}_1$. Hence the descendant nodes of $\{x_1, x_2, \ldots, x_{k_0}\}$ also appear consecutively in $\mathcal{O}_2$. Thus we conclude that $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible if and only if the reduced trees $\mathcal{T}_1'$ and $\mathcal{T}_2'$ are also compatible. Note that both the template reductions take at most $O(n)$ time.

## Case 2: $n_1$ is a P-node and $n_2$ is a Q-node.

If $C_1 - X = \emptyset$, we reduce $\mathcal{T}_1$ by ordering the children of $n_1$ as they appear in the child ordering of $n_2$, and changing $n_1$ into a Q-node (and leading to Case 4). This reduction preserves the compatibility of the two trees.

Now suppose $C_1 - X \neq \emptyset$. Lemma 5.10.5 implies that $C_2 - X \neq \emptyset$ and $U(n_1) = U(n_2)$. By Lemma 5.10.6, we can assume that the nodes in $X$ appear before the nodes in $C_2 - X$ in the child ordering of $n_2$. Now let $X = x_1, \ldots, x_{k_0}$, $C_1 - X = l_1, \ldots, l_{k_1}$ and $C_2 - X = r_1, \ldots, r_{k_2}$. For $i \in 2, \ldots, k_0$, let $\mathcal{S}_i$ be the sequence of subcliques that appear between $x_{i-1}$ and $x_i$ in the child ordering of $n_2$. Note that $\mathcal{S}_i$ consists of the right tail of $x_{i-1}$ followed by the left tail of $x_i$. We let $\mathcal{S}_1$ and $\mathcal{S}_{k_0+1}$ denote the left and right tails of $x_1$ and $x_{k_0}$ respectively. We now reduce the subtree rooted at $n_1$ as shown in Figure 5.3, changing it into a Q-node. Clearly $U(n_1)$ is preserved in this operation. The correctness of this operation follows by Lemma 5.10.2. It is easy to see that both the template reductions run in $O(n)$ time.

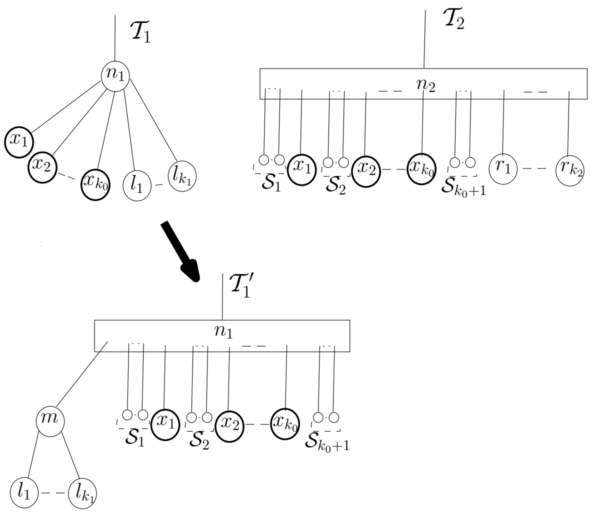


Figure 5.3: Reduction template for Case 2, when $C_1 - X \neq \emptyset$.

## Case 3: $n_1$ is a Q-node and $n_2$ is a P-node.

If $C_2 - X$ is empty, then we reduce $\mathcal{T}_2$ by ordering the child nodes of $n_2$ (i.e. $X$) as they appear in the child ordering of $n_1$, and changing $n_2$ into a Q-node.

Now let $C_2 - X$ be nonempty. By Lemma 5.10.5, $U(n_2) \subseteq U(n_1)$. Let $X = \{x_1, x_2, \ldots, x_{k_0}\}$, $C_2 - X = \{r_1, \ldots, r_{k_2}\}$ and $\mathcal{S}_1, \ldots, \mathcal{S}_{k_0+1}$ be defined as in the previous case: $\mathcal{S}_1$ is the left tail of $x_1$ (in $\mathcal{T}_1$), $\mathcal{S}_i$ is the concatenation of the right tail of $x_{i-1}$ and the left tail of $x_i$, for $i \in \{2, \ldots, k_0\}$ and $\mathcal{S}_{k_0+1}$ is the right tail of $x_{k_0}$.

Now if $C_1 - X$ is empty, then we use the template of Figure 5.4 to reduce $\mathcal{T}_2$, grouping all nodes of $X$ into a new Q-node $w$, ordering them in the way they appear in $\mathcal{T}_1$ and inserting the subclique children of $n_1$ into $w$. Note that since $U(n_2) \subseteq U(n_1)$, this operation doesn't change $U(n_2)$ and hence it preserves the validity of $\mathcal{T}_2$. Further $n_1$ is identical to $w$ and hence we match these nodes. Thus we make progress (by either reducing one of the trees or matching a node) even when $|X| = 1$.

If $C_1 - X$ is non-empty, we use the template similar to Figure 5.3 (to reduce $\mathcal{T}_2$) in which the roles of $n_1$ and $n_2$ have been switched. Note that the template reductions of this case run in $O(n)$ time.

## Case 4: $n_1$ and $n_2$ are both Q-nodes.

Let $X = \{x_1, \ldots, x_{k_0}\}$ appear in that order in the child ordering of $n_1$ and $n_2$. (They appear in the same order because of Lemma 5.10.4.) Let $p_1$ and $p_2$ be the parents of $n_1$ and $n_2$ respectively.

If $n_1$ and $n_2$ have no other children than $X$, we match $n_1$ with $n_2$ and proceed to the next iteration. More typically, they have other children. These may be essential nodes to one side or the other of $X$ (by Lemma 5.10.6) or subclique nodes interspersed in $X$ as tails of the nodes of $X$. We give a high-level outline of Case 4, beginning with a discussion of subclique nodes.

For $i \in \{1, \ldots, k_0\}$, let $\mathcal{L}_i$ and $\mathcal{R}_i$ be the left and right tails of $x_i$ in $\mathcal{T}_1$ and, $\mathcal{L}_i'$ and $\mathcal{R}_i'$ be the left and right tails of $x_i$ in $\mathcal{T}_2$. The only way to deal with the subclique nodes is to do subclique insertions in both trees to merge the tails. This is because in any quasi-intersection

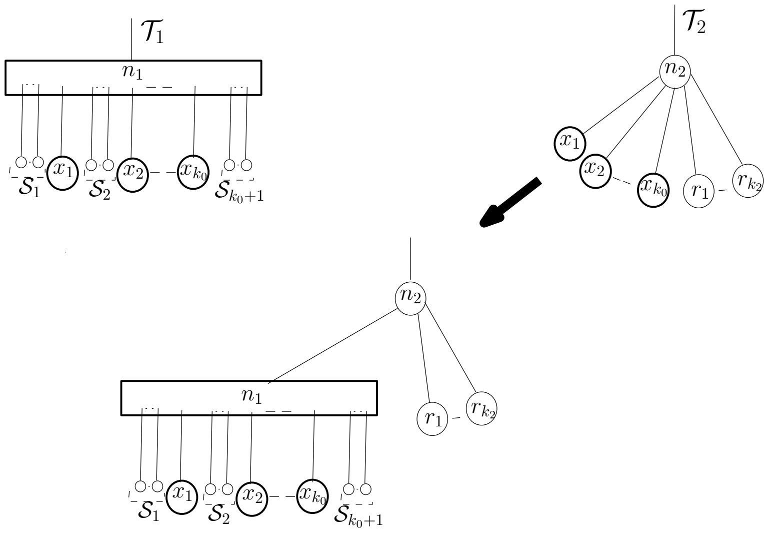


Figure 5.4: Reduction template for Case 3, when $n_1$ is a Q-node, $n_2$ is a P-node and $C_1 - X$ is empty.

tree $\mathcal{T}_I$ obtained from $\mathcal{T}_1$ and $\mathcal{T}_2$, the tails of $x_i$ in $\mathcal{T}_I$ must contain the merge of the tails of $x_i$ in $\mathcal{T}_1$ and $\mathcal{T}_2$. So long as $|X| \geq 2$, the ordering $x_1, \ldots, x_{k_0}$ completely determines which pairs of tails must merge: $\mathcal{L}_i$ must merge with $\mathcal{L}'_i$ and $\mathcal{R}_i$ must merge with $\mathcal{R}'_i$.

The case $|X| = 1$ is more complicated because the quasi-intersection tree may merge $\mathcal{L}_1$ with $\mathcal{L}'_1$ and $\mathcal{R}_1$ with $\mathcal{R}'_1$ or merge $\mathcal{L}_1$ with $\bar{\mathcal{R}}_1$ and $\mathcal{R}_1$ with $\bar{\mathcal{L}}_1$. This decision problem is referred to as the *alignment problem.* We prove (at the beginning of Case 4.3) that in case both choices give mergable pairs, then either choice yields an quasi-intersection tree, if a quasi-intersection tree exists.

This completes our high-level discussion of subclique nodes. We continue with a high-level description of the subcase structure for Case 4. We have subcases depending on whether $U(n_1) = U(n_2)$ and whether $n_1$ and $n_2$ have the same essential children. If both these conditions hold, then we merge the tails of the nodes of $X$ and match $n_1$ with $n_2$. (In other words we replace $\mathcal{L}_i$ and $\mathcal{L}'_i$ with $\mathcal{L}_i + \mathcal{L}'_i$, and replace $\mathcal{R}_i$ and $\mathcal{R}'_i$ with $\mathcal{R}_i + \mathcal{R}'_i$). The cost of matching any two nodes $x$ and $y$ is $(m_x + m_y)|I|$, where $m_x$ and $m_y$ are the number of children of $x$ and $y$ respectively. Once a node is matched, its children will not change. Hence the total amortized cost of matching all the nodes is $O(n \cdot |I|) = O(n^2)$.

When $U(n_1) \neq U(n_2)$ or when $n_1$ and $n_2$ do not have the same essential children then we have three subcases. Case 4.1 handles the situation when $U(n_1) \not\supseteq U(n_2)$. In this case we either insert subcliques of one tree into another and match $n_1$ with $n_2$ or we do some subclique insertions that will take us to the case when $U(n_1) \supseteq U(n_2)$. The remaining cases handle the situation when $U(n_1) \supseteq U(n_2)$, Case 4.2 when $C_1 - X$ is non-empty and Case 4.3 when it is empty. In both cases, we reduce $\mathcal{T}_1$ but the details vary. However in both cases our reduction templates depend on whether $p_1$ is a P-node or a Q-node. If $p_1$ is a P-node, we reduce $\mathcal{T}_1$ by grouping some of the child nodes of $p_1$ into a single node, deleting them and adding the node as a first or last child of $n_1$. If $p_1$ is a Q-node then there are two ways of reducing: delete $n_1$ and reassign its children as children of $p_1$ or reverse the children of $n_1$, delete $n_1$ and reassign its children as children of $p_1$. We refer to this operation as a *collapse.* We now give the details of each case.

**Case 4.1:** $U(n_1) \not\supseteq U(n_2)$**.**
Since $n_1$ was chosen so that $U(n_1)$ is maximal, we also have $U(n_2) \not\supseteq U(n_1)$. Now using Lemma 5.10.5, we infer that $C_1 - X$ is empty and $C_2 - X$ is empty. Thus the difference between $U(n_1)$ and $U(n_2)$ arises due to the subcliques. Let $L$ be a subclique that is either the first or the last child of $n_1$, with the property that $L \not\subseteq U(n_2)$. Such a subclique exists since by Lemma 5.8.6, the intersection of the universal sets of the first and last child nodes of $n_1$ is $U(n_1)$. Also let $R$ be a subclique that is either the first or the last child of $n_2$, with the property that $R \not\subseteq U(n_1)$.

Note that even if $|X| = 1$, the alignment is unique since $L$ and $R$ cannot appear in the same tail of $x_1$ in any quasi-intersection tree. Further, we can assume without loss of generality that $L$ is present in the left tail of $x_1$ in $\mathcal{T}_1$ and $R$ is present in the right tail of $x_{k_0}$ in $\mathcal{T}_2$.

Let $X_1$ be any max-clique descendant of $x_1$. If $p_1$ is a P-node then we claim that $U(p_1) \subseteq U(n_2)$. To see this, let $Z$ be a max-clique descendant of $p_1$ that is not a descendant of $n_1$. In $\mathcal{T}_2$, $Z$ appears outside the subtree rooted at $n_2$. Now by applying Lemma 5.8.0 on $X_1$ and $Z$, we conclude that every descendant of $n_2$ must contain the vertex set $Z \cap X_1$. Thus we have $U(p_1) \subseteq (Z \cap X_1) \subseteq U(n_2)$. Note that if $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, then in any quasi-intersection tree of $\mathcal{T}_1$ and $\mathcal{T}_2$, the nodes of $\mathcal{L}_1$ and $\mathcal{L}'_1$ appear in the left tail of $x_1$ and the nodes of $\mathcal{R}_{k_0}$ and $\mathcal{R}'_{k_0}$ appear in the right tail of $x_{k_0}$. Now if $\mathcal{L}'_1$ is non-empty, then we insert the left most subclique of $\mathcal{L}'_1$ into $\mathcal{L}_1$ (at the appropriate location so that the resulting sequence is still a subclique ordering), as a child of $n_1$. Also if $\mathcal{R}'_{k_0}$ is non-empty, then we insert the right most subclique of $\mathcal{R}'_{k_0}$ into $\mathcal{R}_{k_0}$, as a child of $n_1$. These insertions change $U(n_1)$ to $U(n_1) \cap U(n_2) \supseteq U(p_1)$ and we would be in case 4.3 with the roles of $n_1$ and $n_2$ being reversed. (Note that since the universal set of the modified $n_1$ is a superset of the universal set of $p_1$, the resulting reduced tree of $\mathcal{T}_1$ is valid.) Although this doesn't constitute a progress step since the number of leaf orderings of $n_1$ doesn't change, we will make progress in the Case 4.3.

Similarly if $p_2$ is a P-node then we insert the first subclique of $\mathcal{L}_1$ (if it exists) into $\mathcal{L}'_1$ and the last subclique of $\mathcal{R}_{k_0}$ (if it exists) into $\mathcal{R}'_{k_0}$. After this we would be in Case 4.3.

Now if the parents of $n_1$ and $n_2$ are both Q-nodes then we look at the tails of $n_1$ and $n_2$. If all the subcliques in these tails are subsets of $U(n_1) \cap U(n_2)$, then we replace $\mathcal{L}_i$ and $\mathcal{L}'_i$ with $\mathcal{L}_i + \mathcal{L}'_i$ and $\mathcal{R}_i$ and $\mathcal{R}'_i$ with $\mathcal{R}_i + \mathcal{R}'_i$. This changes $U(n_1)$ and $U(n_2)$ to $U(n_1) \cap U(n_2)$ and makes $n_1$ identical to $n_2$. Thus we match $n_1$ with $n_2$ and iterate.

Otherwise without loss of generality let the subclique $S \not\subseteq U(n_2)$ be present in the (say left) tail of $n_1$. Observe that in any quasi-intersection tree $S$ and $R$ cannot be present in the same tail of $x_{k_0}$ (since neither is a subset of the other). This implies that we can reduce the tree $\mathcal{T}_1$ by collapsing $n_1$ i.e. by removing $n_1$, inserting the sequence of child nodes of $n_1$ after $S$ ($S$ and $L$ are now in the left tail of $x_1$), and assigning $p_1$ as their parent. This completes case 4.1. Note that all the steps in this case take $O(n)$ time, except the matching step (recall that all the matching steps take $O(n^2)$ amortized time).

**Case 4.2:** $U(n_1) \supseteq U(n_2)$ **and** $C_1 - X$ **is non-empty.**
By Lemma 5.10.5, $C_2 - X$ is also non-empty and further $U(n_1)$ is equal to $U(n_2)$. In this case we will reduce $\mathcal{T}_1$ depending on whether $p_1$ is a P-node or a Q-node. Further when $p_1$ is a Q-node, our reduction template also depends on whether $n_1$ has sibling essential nodes.

Let $l_1, l_2, \ldots, l_{k_1}$ be the essential nodes in $C_1 - X$ appearing in that order and appearing (without loss of generality) before the nodes of $X$ in $\mathcal{T}_1$. Note that by Lemma 5.10.2, for each node in $C_1 - X$, the corresponding matched node in $\mathcal{T}_2$ appears outside the subtree rooted at $n_2$. Thus if $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, then all the nodes of $C_2 - X$ must appear after the nodes of $X$ in the child ordering of $n_2$. Let these nodes be $r_1, r_2, \ldots, r_{k_2}$.

**Case 4.2.1: $p_1$ is a P-node.**
Let $Y = \{y_1, y_2, \ldots, y_{k_3}\}$ be the child nodes of $p_1$ other than $n_1$. Also, let $\mathcal{T}_I$ be any quasi-intersection tree of $\mathcal{T}_1$ and $\mathcal{T}_2$. We first observe that for $i \in \{1, \ldots, k_0\}$ and $j \in \{1, \ldots, k_2\}$, $MMD(y_i) \cap MMD(r_j) \neq \emptyset$, if and only if $y_i$ and $r_j$ have a max-clique descendant.

For any such pair $y_i$ and $r_j$, let $MMD(y_i) \cap MMD(r_j) \neq \emptyset$ and let $Y$ be a common max-clique descendant of $y_i$ and $r_j$. Then note that because of the constraints imposed by the child ordering of $n_2$, in any leaf ordering of $\mathcal{T}_I$, the descendant cliques of $l_1$ do not appear between the descendant cliques of $x_1$ and $Y$. Thus $y_i$ must appear after $x_{k_1}$, and so we reduce $\mathcal{T}_1$, by grouping all nodes $y_i$ satisfying $MMD(y_i) \cap MMD(r_j) \neq \emptyset$ for some $r_j$ into a P-node and adding it as a child node of $n_1$ to the (immediate) right of $\mathcal{R}_{k_0}$ as shown in Figure 5.5(top).

Now if $MMD(y_i) \cap MMD(r_j) = \emptyset$ for all $y_i$ and $r_j$, then the above reduction doesn't apply. But in this case (because of the constraints on $n_2$), for every $y_i$ and every leaf ordering of $\mathcal{T}_I$, no max-clique descendant of $y_i$ appears between the max-clique descendants of $n_2$. Thus we group all the nodes of $Y$ into a P-node and add it as a child of $n_1$ to the left of $l_1$ as shown in Figure 5.5(bottom).

Note that for any two distinct nodes $y_a, y_b \in \{y_1, \ldots, y_{k_3}\}$ we have: $MMD(y_a) \cap MMD(y_b) = \emptyset$. Similarly, for any two distinct nodes $r_a, r_b \in \{r_1, \ldots, r_{k_2}\}$ we have: $MMD(r_a) \cap MMD(r_b) = \emptyset$. This implies that we can first compute the MM-descendants of all $y_i$ and $r_j$ in $O(n)$ time and further we can compute all nodes $y_i$ that satisfy $MMD(y_i) \cap MMD(r_j) \neq \emptyset$ for some $r_j$, in $O(n)$ time. Thus the template reductions of Figure 5.5 run in $O(n)$ time.

**Case 4.2.2: $p_1$ is a Q-node and $n_1$ is its only essential child.**
Since the only essential child of $p_1$ is $n_1$, all of its remaining children are subcliques that are present as tails of $n_1$. Thus each of these subcliques is a subset of $U(n_1)$. Now let $Z$ and $R$ be any two max-clique descendants of $x_{k_0}$ and $r_1$ respectively. By Lemma 5.10.2, $R$ appears outside the subtree rooted at $n_1$ (in $\mathcal{T}_1$) and hence outside the subtree rooted at $p_1$. By Lemma 5.8.0, we conclude that each descendant clique of $p_1$ must contain $Z \cap R$. Thus we have $U(p_1) \supseteq Z \cap R \supseteq U(n_2) = U(n_1) \supseteq U(p_1)$. Hence all of these sets must be equal and hence we infer the following: $Z \cap R = U(n_1)$ and hence $U(x_{k_0}) \cap U(r_1) = U(n_1)$. Further, each subclique child of $p_1$ must precisely be the clique $U(n_1)$.

Since we have eliminated adjacent duplicates from all Q-nodes, there can be at most one such subclique in each tail of $n_1$. Now if the subclique $(U(n_1))$ appears on both sides of $n_1$, then there is a unique way of collapsing $n_1$ (see Figure 5.6(top)). Otherwise we collapse $n_1$ in such a way that $U(n_1)$ is present in the tail of $x_{k_0}$ as shown in Figure 5.6(bottom). This is justified (i.e. it preserves compatibility between $\mathcal{T}_1$ and $\mathcal{T}_2$) because $U(n_1)$ can be inserted into the right tail of $x_{k_0}$ in both $\mathcal{T}_1$ and $\mathcal{T}_2$. In other words, if $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, then there exists a quasi-intersection tree in which $U(n_1)$ is present in the right tail of $x_{k_0}$. The template reductions of this case clearly run in $O(n)$ time.

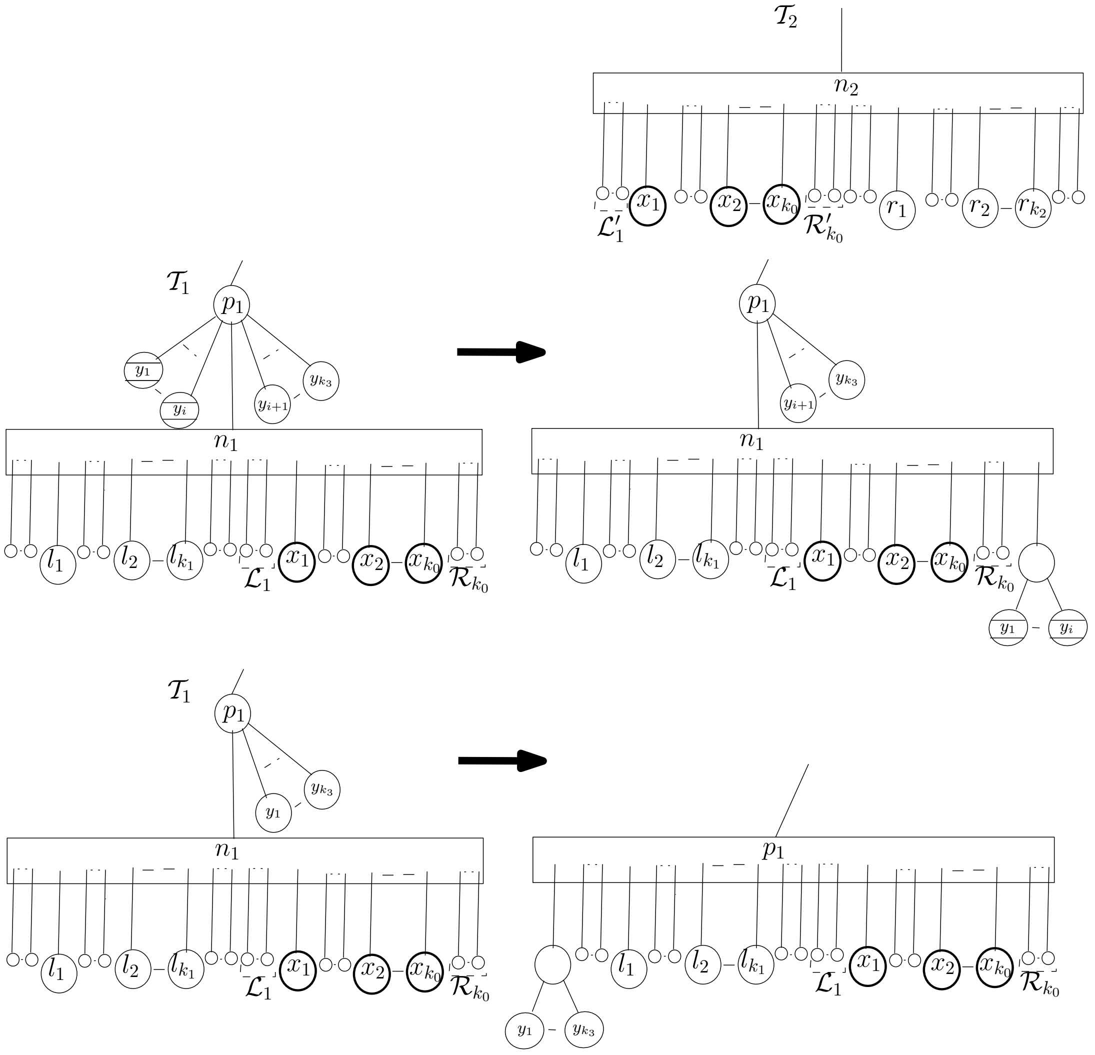


Figure 5.5: Reduction template of $\mathcal{T}_1$ for Case 4.2.1. A node $y_a$ has horizontal stripes if $MMD(y_a) \cap MMD(r_b) \neq \emptyset$ for some $r_b$ and no stripes otherwise.

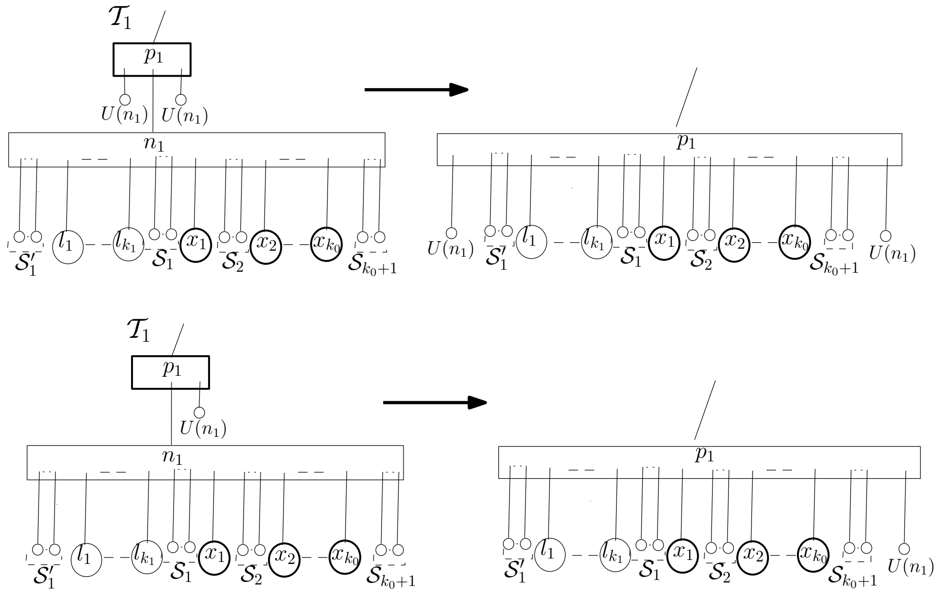


Figure 5.6: Reduction templates of $\mathcal{T}_1$ for Case 4.2.2.

**Case 4.2.3: $p_1$ is a Q-node and has more than one essential child.**
Let $y$ be an essential child of $p_1$, such that all the nodes between $n_1$ and $y$ are subcliques. Without loss of generality, we assume that $y$ appears to the right of $n_1$. We collapse $n_1$, depending on whether $MMD(y) \cap MMD(r_1)$ is empty or not, as shown in Figure 5.7. Thus the template reduction runs in $O(n)$ time.

If $MMD(y) \cap MMD(r_1)$ is non-empty, there exists a max-clique $Y$ that is a descendant of both $r_1$ and $y$. Now if $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, then in the leaf ordering of any quasi-intersection tree, the max-clique descendants of $x_{k_0}$ appear in between the max-clique descendants of $l_1$ and $Y$. Thus we collapse the node $n_1$, by deleting $n_1$, and reassigning $p_1$ as the parent of all the children of $n_1$. (Thus no essential node appears between $x_{k_0}$ and $y$.)

On the other hand if $MMD(y) \cap MMD(r_1)$ is empty, we observe the following: In the leaf ordering of any quasi-intersection tree $\mathcal{T}_I$ no max-clique appears in between the max-clique descendants of $x_{k_0}$ and the max-clique descendants of $r_1$. Therefore, in this case we collapse $n_1$, by reversing its children, deleting it, and reassigning $p_1$ as the parent of all the children of $n_1$. (Thus no essential node appears between $l_1$ and $y$.)

**Case 4.3: $U(n_1) \supsetneq U(n_2)$ and $C_1 - X$ is empty.**
As before we have three cases depending on whether $p_1$ is a P-node or a Q-node and whether $p_1$ has more than one essential child. In each of these cases, when $|X| = 1$, we need to first solve the alignment problem (as a first step). Also when $p_1$ is a Q-node, both ways of collapsing $n_1$ may lead to a valid quasi-intersection tree.

## Alignment Problem

Recall that when $|X| = 1$ (and $C_1 - X = \emptyset$), the alignment may not be unique i.e. one of the following might happen in the quasi-intersection tree $\mathcal{T}_I$.

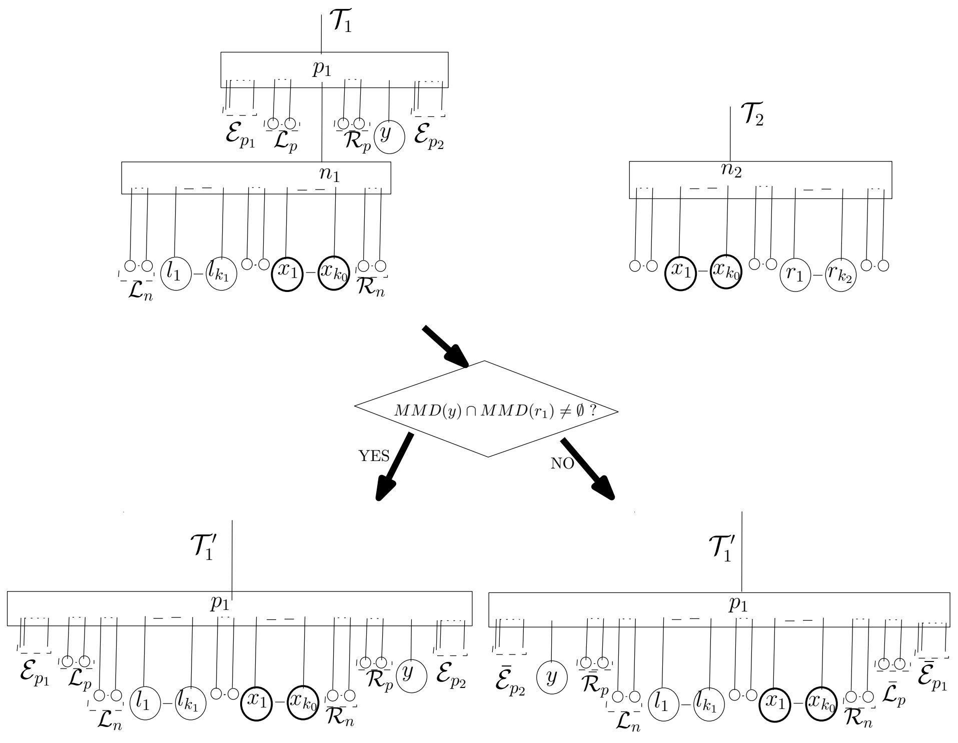


Figure 5.7: Reduction template of $\mathcal{T}_1$ for Case 4.2.3.

1. Left and right tails of $x_1$ (in $\mathcal{T}_I$) contain $\mathcal{L}_1 + \mathcal{L}_1'$ and $\mathcal{R}_1 + \mathcal{R}_1'$ respectively.

2. Left and right tails of $x_1$ (in $\mathcal{T}_I$) contain $\bar{\mathcal{R}}_1 + \mathcal{L}_1'$ and $\bar{\mathcal{L}}_1 + \mathcal{R}_1'$ respectively.

If one of the merges in (1) or (2) is invalid, then there is only a single way of aligning the tails, otherwise we show in the following lemma that if $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, then choosing either one of the two alignments will work.

**Lemma 5.11.** *Let $U(n_1) \supseteq U(n_2)$, $|X| = 1$ and $C_1 - X$ be empty. Let $\mathcal{L}_1$, $\mathcal{R}_1$ be the left and right tails of $x_1$ in $\mathcal{T}_1$ and $\mathcal{L}_1'$, $\mathcal{R}_1'$ be the left and right tails of $x_1$ in $\mathcal{T}_2$. If both ways of alignment are mergable i.e. (a) $\mathcal{L}_1 + \mathcal{L}_1'$, $\mathcal{R}_1 + \mathcal{R}_1'$ are valid and (b) $\bar{\mathcal{R}}_1 + \mathcal{L}_1'$, $\bar{\mathcal{L}}_1 + \mathcal{R}_1'$ are valid, then there exists a quasi-intersection tree $\mathcal{T}_I$ (of $\mathcal{T}_1$ and $\mathcal{T}_2$) with $\mathcal{L}_1 + \mathcal{L}_1'$ and $\mathcal{R}_1 + \mathcal{R}_1'$ contained in the left and right tails of $x_1$ (respectively) if and only if there exists a quasi-intersection tree $\mathcal{T}_I'$ with $\bar{\mathcal{R}}_1 + \mathcal{L}_1'$ and $\bar{\mathcal{L}}_1 + \mathcal{R}_1'$ contained in the left and right tails of $x_1$ (respectively).*

*Proof.* Let $\mathcal{L}$, $\mathcal{R}$ be the left and right tails of $x_1$ in a quasi-intersection tree $\mathcal{T}_I$. Each subclique $S$ in $\mathcal{L}$ or $\mathcal{R}$ appears as a subclique in $\mathcal{T}_1$ or $\mathcal{T}_2$. In particular we observe the following:

*Property 1:* If $S$ is a subclique in $\mathcal{L}$ or $\mathcal{R}$ then in $\mathcal{T}_1$ or $\mathcal{T}_2$, $S$ is present in a tail of $x_1$ or in a tail of an ancestor of $x_1$.

Note that since the parent of $x_1$ in $\mathcal{T}_I$ contains at least two children, $\mathcal{L}$ and $\mathcal{R}$ both cannot be empty. If one of them, say $\mathcal{L}$ is empty then the last clique in $\mathcal{R}$ must be $U(n_1)$. If both $\mathcal{L}$ and

$\mathcal{R}$ are non-empty then the intersection of the first subclique of $\mathcal{L}_1$ with the last subclique of $\mathcal{R}_1$ is $U(n_1)$. In either case we observe that, since $\mathcal{L}_1 + \mathcal{L}'_1$ and $\bar{\mathcal{R}}_1 + \mathcal{L}'_1$ are both valid (subclique orderings), each subclique in $\mathcal{L}'_1$ is either a superset of $U(n_1)$ or a subset of $U(n_1)$. Similarly, since $\mathcal{R}_1 + \mathcal{R}'_1$ and $\bar{\mathcal{L}}_1 + \mathcal{R}'_1$ are both valid (superclique orderings), each subclique in $\mathcal{R}'_1$ is either a superset of $U(n_1)$ or a subset of $U(n_1)$. Further for any ancestor $n_a$ of $n_2$, $U(n_a) \subseteq U(n_2) \subseteq U(n_1)$ and hence the tails of any such $n_a$ would consist of subcliques that are subsets of $U(n_1)$. Note that this condition also holds for any ancestor of $n_1$ in $\mathcal{T}_1$.

By above conditions and (1) we infer that for any subclique $S$ in $\mathcal{L}$ or $\mathcal{R}$, $S$ is either a superset of $U(n_1)$ or a subset of $U(n_1)$. Furthermore, if $S$ is a superset of $U(n_1)$ then it is present in one of $\mathcal{L}_1, \mathcal{R}_1, \mathcal{L}'_1$ or $\mathcal{R}'_1$. This implies that if there exists a quasi-intersection tree $\mathcal{T}_I$ in which $\mathcal{L}$ contains $\mathcal{L}_1 + \mathcal{L}'_1$ and $R$ contains $\mathcal{R}_1 + \mathcal{R}'_1$, then replacing $\mathcal{L}$ with $\mathcal{L} - \mathcal{L}_1 + \bar{\mathcal{R}}_1$ and $\mathcal{R}$ with $\mathcal{R} - \mathcal{R}_1 + \bar{\mathcal{L}}_1$ also results in a valid quasi-intersection tree. □

Note that the amortized cost of doing the mergability checks (a) and (b) of Lemma 5.11 (over all iterations of the algorithm) is $O(n \cdot |I|) = O(n^2)$. For the rest of the cases, we can assume that $\mathcal{L}_1$ is aligned with $\mathcal{L}_2$ and $\mathcal{R}_1$ is aligned with $\mathcal{R}_2$. In other words if $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, then there exists a quasi-intersection tree that contains $\mathcal{L}_1+\mathcal{L}_2$ and $\mathcal{R}_1+\mathcal{R}_2$ as the tails of $x_1$.

### Case 4.3.1: $p_1$ is a P-node.

If $C_2 - X = \emptyset$, then using the same argument as before (Lemma 5.8.0), we get $U(p_1) \subseteq U(n_2)$. Hence we replace $\mathcal{L}_i$ and $\mathcal{L}'_i$ with $\mathcal{L}_i + \mathcal{L}'_i$ in $\mathcal{T}_1$ and $\mathcal{T}_2$ changing $U(n_1)$ to $U(n_1) \cap U(n_2) \supseteq U(p_1)$, and we match $n_1$ with $n_2$.

Now we look at the case when $C_2 - X$ is non-empty. Let $L = \{l_1, l_2, \ldots, l_{k_1}\}$ be the set of essential nodes appearing to the left of $X$ and $R = \{r_1, \ldots, r_{k_2}\}$ be the set of essential nodes appearing to the right of $X$ in $\mathcal{T}_2$. Let $Y = \{y_1, \ldots, y_{k_3}\}$ be all the remaining child nodes of $p_1$ other than $n_1$. For all $i \in \{1, \ldots, k_3\}$, if $MMD(y_i) \cap MMD(l_j) \neq \emptyset$ for some $j \in \{1, \ldots, k_1\}$, then $y_i$ and $l_j$ both have a common max-clique descendant say $Y$, and further in any leaf order of a quasi-intersection tree $\mathcal{L}_1 + \mathcal{L}'_1$ must appear between $Y$ and the descendants of $x_1$.

Thus we group all $y_i$ such that $MMD(y_i) \cap MMD(l_j) \neq \emptyset$ into a new P-node and add it to the (immediate) left of $\mathcal{L}_1$ (see Figure 5.8). Similarly, we group all $y_i$ such that $MMD(y_i) \cap MMD(r_j) \neq \emptyset$, for some $j \in \{1, \ldots, k_2\}$ into a new P-node and add it to the (immediate) right of $\mathcal{R}_{k_0}$.

Note that if for some $y \in Y$, there exist $l_i$ and $r_j$ such that both $MMD(y) \cap MMD(l_i) \neq \emptyset$ and $MMD(y) \cap MMD(r_j) \neq \emptyset$, then we can conclude that $\mathcal{T}_1$ and $\mathcal{T}_2$ are incompatible.

Also if $L$ and $R$ are both non-empty and for all $y \in Y$, $MMD(y)$ doesn't intersect with any $MMD(l_i)$ for $i \in \{1, \ldots, k_1\}$ and with any $MMD(r_j)$ for $j \in \{1, \ldots, k_2\}$ then once again we conclude that $\mathcal{T}_1$ and $\mathcal{T}_2$ are incompatible.

On the other hand if one of $L$ or $R$ is empty, say $L$, and $MMD(y_i) \cap MMD(r_j)$ is empty for all $y_i \in Y$ and $r_j \in R$, then the above template would not reduce $\mathcal{T}_1$. But then note that in any leaf-ordering of any quasi-intersection tree, $\mathcal{L}_1 + \mathcal{L}'_1$ should appear between the descendants of $y_i$ and $x_1$ for all $y_i \in Y$ (because of the constraints imposed by $\mathcal{T}_1$ and $\mathcal{T}_2$). Hence in this case we group all the nodes of $Y$ into a P-node and add it a child node of $p_1$ to the (immediate) left of $\mathcal{L}_1$ as shown in Figure 5.9.

Note that since the MM-descendents of any two sibling nodes are disjoint, both of the above templates can be implemented in $O(n)$ time.

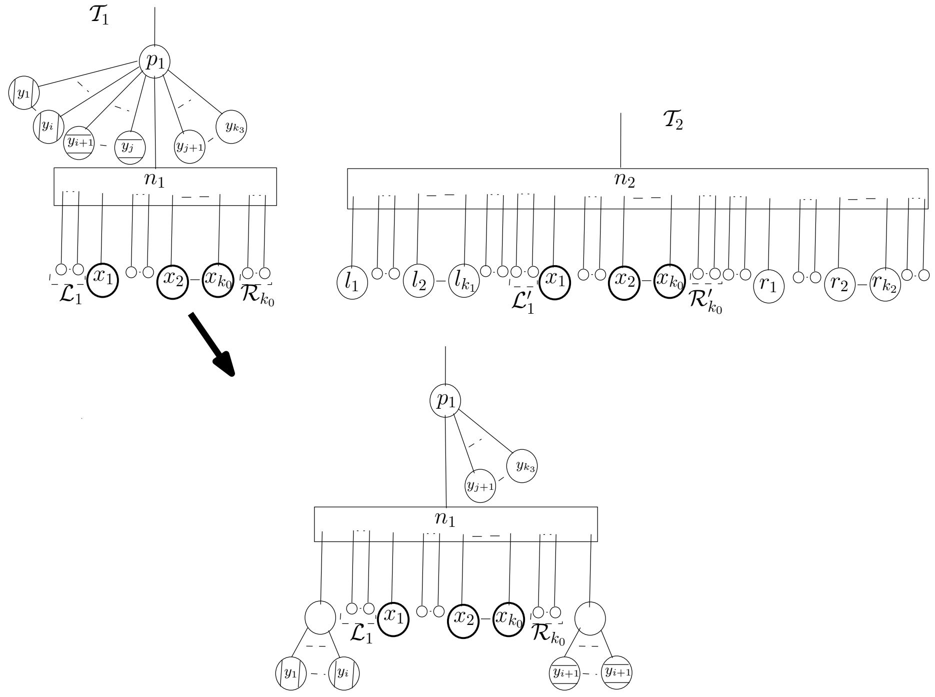


Figure 5.8: First reduction template of $\mathcal{T}_1$ for Case 4.3.1. A node $y_a$ has vertical stripes if $MMD(y_a) \cap MMD(l_b) \neq \emptyset$ for some $l_b$, horizontal stripes if $MMD(y_a) \cap MMD(r_b) \neq \emptyset$ for some $r_b$ and no stripes otherwise.

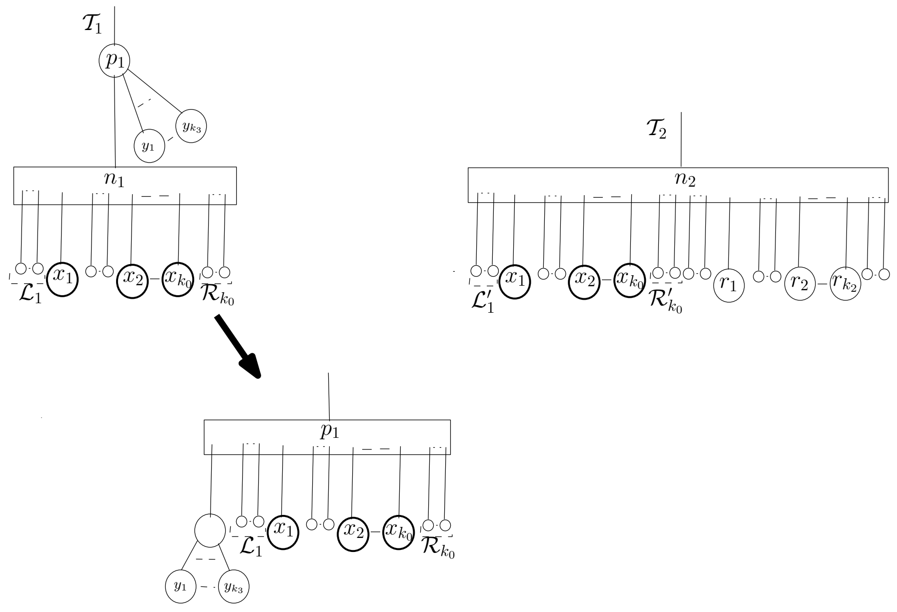


Figure 5.9: Second reduction template of $\mathcal{T}_1$ for Case 4.3.1. The stripes on the $y$ nodes are defined as before.

**Case 4.3.2: $p_1$ is a Q-node and $n_1$ is its only essential child.**
Let $\mathcal{L}_p$ and $\mathcal{R}_p$ be the left and right tails of $p_1$. Note that in this case all the siblings of $n_1$ are subcliques that are present in its tails. We have three subcases depending on how $U(p_1)$ intersects $U(n_2)$.

Suppose $U(p_1)$ properly intersects $U(n_2)$. We have $U(p_1)-U(n_2) \neq \emptyset$ and $U(n_2)-U(p_1) \neq \emptyset$. We first claim that $C_2 - X$ is empty. Suppose not. Let $Z$ be a max-clique descendant of a node in $C_2 - X$ and $X_1$ be a max-clique descendant of $x_1$. By Lemma 5.8.2, in $\mathcal{T}_1$, $Z$ appears outside the subtree rooted at $n_1$, and hence outside the subtree rooted at $p_1$. Thus using Lemma 5.8.0, we conclude that each descendant of $p_1$ must contain all the vertices in $Z \cap X_1 \supseteq U(n_2)$. A contradiction. Hence $C_2 - X$ is empty.

Now by Lemma 5.8.6, there exists a subclique $S_1 \not\supseteq U(n_2)$ such that $S_1$ is the first clique of $\mathcal{L}_p$ or the last clique of $\mathcal{R}_p$. Similarly there exists a subclique $S_2 \not\supseteq U(p_1)$ such that $S_1$ is the first clique of $\mathcal{L}'_1$ or the last clique of $\mathcal{R}'_{k_0}$. Without loss of generality let $S_1$ be the first clique of $\mathcal{L}_p$ and $S_2$ be the last clique of $\mathcal{R}'_{k_0}$. Observe that $S_1 \supseteq U(p_1)$ and $S_2 \supseteq U(n_2)$. This implies that $S_1$ and $S_2$ cannot be in the same tail (of $x_1$ or $x_{k_0}$) in any quasi-intersection tree of $\mathcal{T}_1$ and $\mathcal{T}_2$. Thus we reduce $\mathcal{T}_1$ by collapsing $n_1$ i.e. by deleting $n_1$, changing the parent of child nodes of $n_1$ to $p_1$ and arranging the child nodes such that $\mathcal{L}_p$ appears to the left of $\mathcal{L}_1$ and $\mathcal{R}_p$ appears to the right of $\mathcal{R}_{k_0}$. Clearly, this reduction can be done in $O(n)$ time.

Now we have to deal with the case when either $U(n_2) \subseteq U(p_1)$ or $U(p_1) \subseteq U(n_2)$. Note that in any quasi-intersection tree $\mathcal{T}_I$ (of $\mathcal{T}_1$ and $\mathcal{T}_2$), the cliques of $\mathcal{L}_1 + \mathcal{L}'_1$ appear in the left tail of $x_1$ and the cliques of $\mathcal{R}_{k_0} + \mathcal{R}'_{k_0}$ appear in the right tail of $x_{k_0}$. Further either (a) the cliques of $\mathcal{L}_p$ appear in the left tail of $x_1$ and the cliques of $\mathcal{R}_p$ appear in the right tail of $x_{k_0}$ or (b) the cliques of $\mathcal{L}_p$ appear in the right tail of $x_{k_0}$ and the cliques of $\mathcal{R}_p$ appear in the left tail of $x_1$. In the first case $\mathcal{L}_1 + \mathcal{L}'_1 + \mathcal{L}_p$ and $\mathcal{R}_{k_0} + \mathcal{R}'_{k_0} + \mathcal{R}_p$ are both valid and in the second case $\mathcal{L}_1 + \mathcal{L}'_1 + \bar{\mathcal{R}}_p$ and $\mathcal{R}_{k_0} + \mathcal{R}'_{k_0} + \bar{\mathcal{L}}_p$ are both valid. If neither of these is valid then we conclude that $\mathcal{T}_1$ and $\mathcal{T}_2$ are incompatible. If exactly one of the above merges is valid, then there is a unique way of collapsing $n_1$. When both of the above merge pairs are valid, we use the reduction template shown in Figure 5.10. The justification (given below) depends on whether $U(n_2) \subseteq U(p_1)$ or $U(p_1) \subseteq U(n_2)$.

Let $U(n_2) \subseteq U(p_1)$. Note that by Lemma 5.8.6, the intersection of the universal nodes of the first and last child nodes of $p_1$ is $U(p_1)$. Hence if $\mathcal{L}_1 + \mathcal{L}'_1 + \mathcal{L}_p$, $\mathcal{R}_{k_0} + \mathcal{R}'_{k_0} + \mathcal{R}_p$, $\mathcal{L}_1 + \mathcal{L}'_1 + \bar{\mathcal{R}}_p$ and $\mathcal{R}_{k_0} + \mathcal{R}'_{k_0} + \bar{\mathcal{L}}_p$ are all valid then any subclique in $\mathcal{L}'_1$ or $\mathcal{R}'_{k_0}$ is either a superset or a subset of $U(p_1)$. Thus in any quasi-intersection tree $\mathcal{T}_I$, any subclique $S$ in the left tail of $x_1$ or the right tail of $x_{k_0}$ is either a superset or a subset of $U(p_1)$. Further if $S \supseteq U(p_1)$, then $S$ must appear in one of $\{\mathcal{L}'_1, \mathcal{R}'_{k_0}, \mathcal{L}_p, \mathcal{R}_p, \mathcal{L}_1, \mathcal{R}_{k_0}\}$. This implies that a quasi-intersection tree satisfying condition (a) exists if and only if a quasi-intersection tree satisfying condition (b) exists. This justifies the use of our template in Figure 5.10, for reducing $\mathcal{T}_1$.

Similarly, if $U(p_1) \subseteq U(n_2)$, we infer that any clique in $\mathcal{L}_p$ or $\mathcal{R}_p$ is either a subset of $U(n_2)$ or a superset of $U(n_2)$. This in turn implies that in $\mathcal{T}_I$, any subclique $S$ in the left tail of $x_1$ or the right tail of $x_1$, is either a subset or a superset of $U(n_2)$. Further, if $S \supseteq U(n_2)$ then it must appear in $\{\mathcal{L}_1, \mathcal{R}_{k_0}, \mathcal{L}'_1, \mathcal{R}'_{k_0}, \mathcal{L}_p, \mathcal{R}_p\}$. This implies that a quasi-intersection tree satisfying condition (a) exists if and only if a quasi-intersection tree satisfying (b) exists. This justifies the use of our template in Figure 5.10, for reducing $\mathcal{T}_1$.

We now show that the amortized cost of executing the reduction template in Figure 5.10, over all instances of the algorithm takes $O(n^2)$ time. Note that we use the same template for Case

4.3.3 when $C_2 - X$ is empty. It is enough to show that the amortized time of all the mergability checks: (whether $\mathcal{L}'_1 + \mathcal{L}_p$ and $\mathcal{R}'_{k_0} + \mathcal{R}_p$ are both valid) take $O(n^2)$ time.

Let $c(\mathcal{L}'_1)$ and $c(\mathcal{R}'_{k_0})$ be the (consecutive) subsequences of $\mathcal{L}'_1$ and $\mathcal{R}'_{k_0}$ (respectively) such that each subclique in $c(\mathcal{L}'_1)$ and $c(\mathcal{R}'_{k_0})$ contains $U(p_1)$ but not $U(n_1)$. $c(\mathcal{L}'_1)$ and $c(\mathcal{R}'_{k_0})$ are said to be the *core* tails of $n_2$.

Similarly let $c(\mathcal{L}_p)$ and $c(\mathcal{R}_p)$ be the (consecutive) subsequences of $\mathcal{L}_p$ and $\mathcal{R}_p$ (respectively) such that each subclique in $c(\mathcal{L}_p)$ and $c(\mathcal{R}_p)$ contains $U(n_2)$. Then $c(\mathcal{L}_p)$ and $c(\mathcal{R}_p)$ are said to be the *core* tails of $p_1$.

Note that the core tails are only defined for $p_1$ and $n_2$, for the current case and Case 4.3.3, when $C_2 - X$ is empty. We define the core tails of all other nodes to be empty. Observe that when $n_1$ is collapsed, the (new) core tails of any node in $\mathcal{T}_1$ (resp. $\mathcal{T}_2$) are disjoint from the core tails of $p_1$ (resp. $n_2$) before the collapse.

We observe that checking the validity of $\mathcal{L}'_1 + \mathcal{L}_p$ reduces to checking the validity of $c(\mathcal{L}'_1) + c(\mathcal{L}_p)$. Similarly, checking the validity of $\mathcal{R}'_{k_0} + \mathcal{R}_p$ reduces to checking the validity of $c(\mathcal{R}'_{k_0}) + c(\mathcal{R}_p)$.

Now computing the cores over all executions of this template takes $O(n \cdot |I|) = O(n^2)$ amortized time. Also, computing the mergability of the cores, over all executions of the template takes $\sum_i (m_i + t_i)|I|$, where $m_i$, $t_i$ are the number of subcliques in the cores of $n_2$ and $p_1$ (respectively), in the $i$th execution of the template. Since $\sum_i m_i = O(n)$ and $\sum_i t_i = O(n)$, the total running time of the template in Figure 5.10 over all executions is $O(n^2)$.

### Case 4.3.3: $p_1$ is a Q-node with more than one essential child.

Now let $y$ be an essential child of $p_1$, such that all the nodes between $n_1$ and $y$ are subcliques. Without loss of generality, we assume that $y$ appears to the right of $n_1$.

We first consider the subcase when $C_2 - X$ is empty. In this case observe that all the max-clique descendants of $y$ appear outside the subtree rooted at $n_1$ in $\mathcal{T}_1$. Applying Lemma 5.8.0 on a max-clique descendant of $y$ and a max-clique descendant of $x_1$, we infer that each descendant clique of $n_2$ must contain all the vertices in $U(p_1)$. In other words we get $U(p_1) \subseteq U(n_2)$. Now the template (and the argument) in this case is analogous to case 4.3.2, when $U(p_1) \subseteq U(n_2)$ (Figure 5.10).

Now suppose $C_2 - X$ is non empty. Let $r_1$ be the first essential child to the right of $x_{k_0}$ in $\mathcal{T}_2$. We use the templates in Figure 5.11, depending on whether $MMD(y) \cap MMD(r_1)$ is empty or not. If $MMD(y) \cap MMD(r_1)$ is non-empty, then by Lemma 5.8.2, there exists a max-clique $Y$ that is a descendant of both $r_1$ and $y$. Now if $\mathcal{T}_1$ and $\mathcal{T}_2$ are compatible, then in any leaf ordering of a quasi-intersection tree $\mathcal{T}_I$, the subcliques of $\mathcal{R}_{k_0} + \mathcal{R}'_{k_0}$ appear in between the descendants of $x_k$ and $Y$ (because of $\mathcal{T}_2$). This justifies the reduction template of $\mathcal{T}_1$ in Figure 5.11.

On the other hand, if $MMD(r_1) \cap MMD(y) = \emptyset$, then by the constraints of $\mathcal{T}_2$, in any leaf ordering of $\mathcal{T}_I$, the subcliques of $\mathcal{R}_{k_0} + \mathcal{R}'_{k_0}$ appear between the descendants of $x_{k_0}$ and $l_1$ and further no max-clique appears between them. This justifies the reduction template of $\mathcal{T}_1$ in Figure 5.11. Moreover the template reduction takes $O(n)$ time.

### Run time of the Algorithm.

In this section, we show that the run time of our algorithm is $O(n^2 \log n)$, where $n$ is the total number of vertices in $G_1 \cup G_2$.

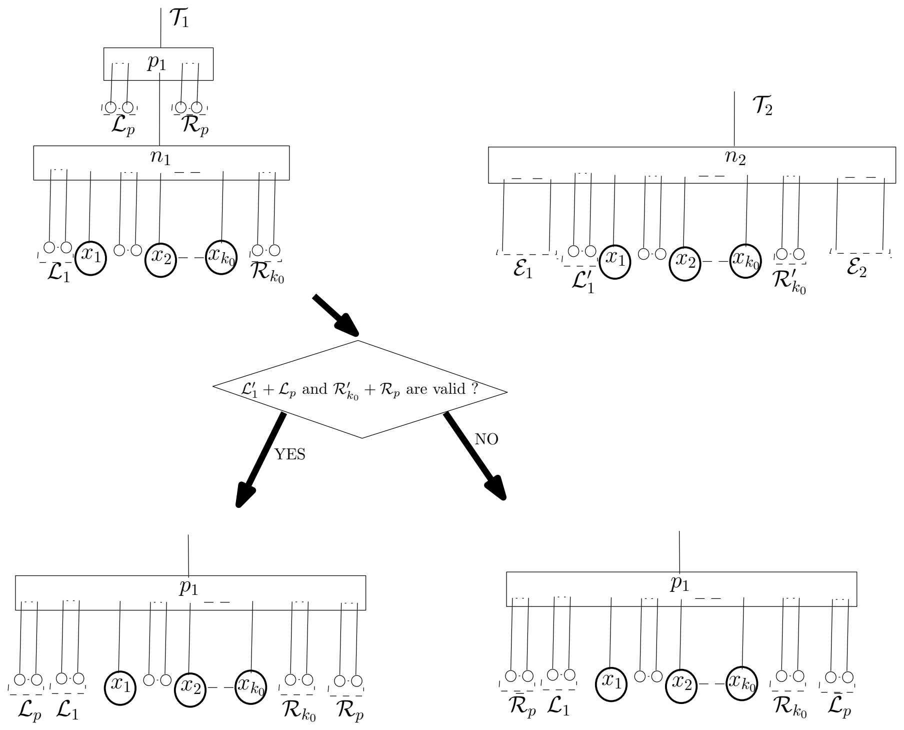

$\mathcal{T}_1$
$p_1$
$\mathcal{L}_p$
$\mathcal{R}_p$
$n_1$
$\mathcal{L}_1$
$x_1$
$x_2$
$x_{k_0}$
$\mathcal{R}_{k_0}$
$\mathcal{T}_2$
$n_2$
$\mathcal{E}_1$
$\mathcal{L}'_1$
$\mathcal{R}'_{k_0}$
$\mathcal{E}_2$
$\mathcal{L}'_1 + \mathcal{L}_p$ and $\mathcal{R}'_{k_0} + \mathcal{R}_p$ are valid ?
YES
NO
$p_1$
$\mathcal{L}_p$ $\mathcal{L}_1$ $x_1$ $x_2$ $x_{k_0}$ $\mathcal{R}_{k_0}$ $\mathcal{R}_p$
$p_1$
$\mathcal{R}_p$ $\mathcal{L}_1$ $x_1$ $x_2$ $x_{k_0}$ $\mathcal{R}_{k_0}$ $\mathcal{L}_p$

Figure 5.10: Reduction template of $\mathcal{T}_1$ for Case 4.3.2. Note that $\mathcal{E}_1$ and $\mathcal{E}_2$ denote (possibly empty) sequences of cliques.

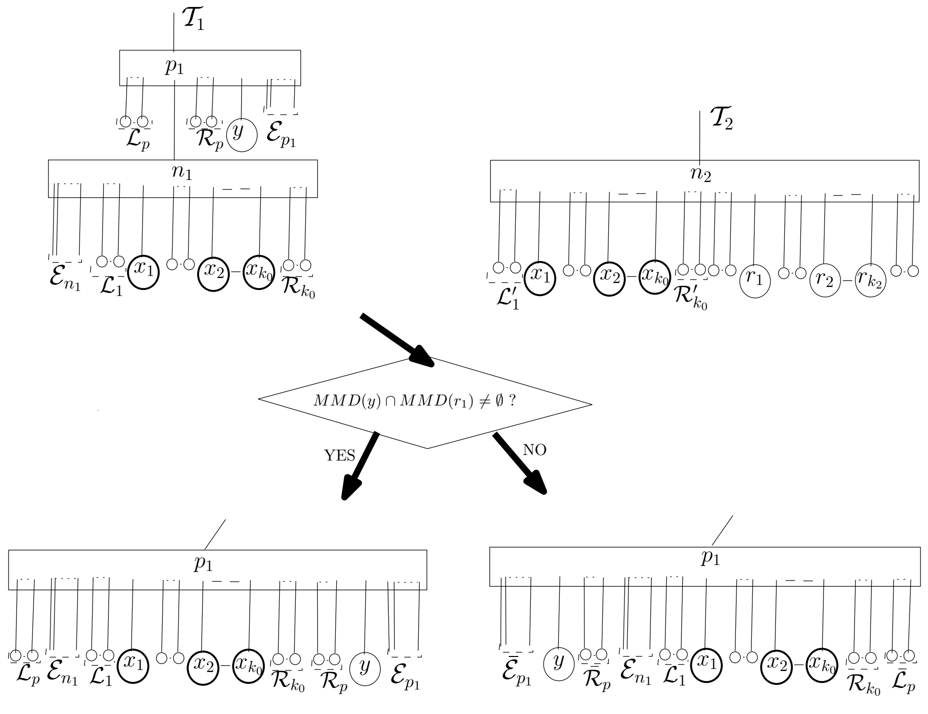

$\mathcal{T}_1$
$p_1$
$\mathcal{L}_p$
$\mathcal{R}_p$
$y$
$\mathcal{E}_{p_1}$
$n_1$
$\mathcal{E}_{n_1}$ $\mathcal{L}_1$ $x_1$ $x_2$ $x_{k_0}$ $\mathcal{R}_{k_0}$
$\mathcal{T}_2$
$n_2$
$\mathcal{L}'_1$ $x_1$ $x_2$ $x_{k_0}$ $\mathcal{R}'_{k_0}$ $r_1$ $r_2$ $r_{k_2}$
$MMD(y) \cap MMD(r_1) \neq \emptyset$ ?
YES
NO
$p_1$
$\mathcal{L}_p$ $\mathcal{E}_{n_1}$ $\mathcal{L}_1$ $x_1$ $x_2$ $x_{k_0}$ $\mathcal{R}_{k_0}$ $\mathcal{R}_p$ $y$ $\mathcal{E}_{p_1}$
$p_1$
$\bar{\mathcal{E}}_{p_1}$ $y$ $\bar{\mathcal{R}}_p$ $\mathcal{E}_{n_1}$ $\mathcal{L}_1$ $x_1$ $x_2$ $x_{k_0}$ $\mathcal{R}_{k_0}$ $\bar{\mathcal{L}}_p$

Figure 5.11: Reduction template of $\mathcal{T}_1$ for Case 4.3.3.

Observe that reducing $\mathcal{T}_1$ [resp. $\mathcal{T}_2$] decreases the number of leaf orderings of $\mathcal{T}_1$ [resp. $\mathcal{T}_2$] by at least half. Moreover the total number of nodes in $\mathcal{T}_1$ and $\mathcal{T}_2$ is at most $n$. Thus the number of leaf orderings of $\mathcal{T}_1$ and $\mathcal{T}_2$ is at most $n!$ and hence the algorithm requires at most $n \log n$ reductions.

We begin by showing that selecting the nodes $n_1$ and $n_2$ takes $O(n)$ time in any iteration. We first note that computing the number of MM-descendants for all the nodes takes $O(n)$ time (they can be computed in a bottom-up fashion). With each node $x$, we store $U(x)$ and the cardinality of $MMD(x)$.

Recall that as we go down the tree the universal sets increase and the MM-descendants sets decrease. Thus $n_1$ must have maximal depth among all unmatched essential nodes. Hence we can select $n_1$ in $O(n)$ time by looking at the unmatched essential nodes of maximal depth in $\mathcal{T}_1$ and $\mathcal{T}_2$, and selecting a node with the greatest universal set size and the least number of MM-descendants in that order. Note that by property 1 of Lemma 5.10, the MM-descendants of $n_1$ are same as the essential child nodes of $n_1$. Now we can select $n_2$ from the other tree $\mathcal{T}_2$ in $O(n)$ time as follows: Let $S$ be the set of [matched] essential children of $n_1$ and $S'$ be the corresponding set of matched nodes in $\mathcal{T}_2$. Let $p(S')$ be the set of parent nodes of nodes in $S'$. We select $n_2$ to be a node of maximum depth among $p(S')$.

Also recall that at the high level our algorithm has 4 cases depending on whether $n_1$, $n_2$ are P-nodes or Q-nodes. We have shown that each step in cases 1,2 or 3 takes $O(n)$ time. For case 4, we have also shown that each of reduction steps, excluding the mergability checks in Cases 4.3.2 and 4.3.3 take $O(n)$ time. We have also shown that the mergability checks of Cases 4.3.2 and 4.3.3 take $O(n^2)$ amortized time over all steps of the algorithm. Further, in the beginning of Case 4, we have shown that the matching steps, which involve inserting the subcliques of one tree into the other take at most $O(n^2)$ amortized time. Thus the total time taken by our algorithm is $O(n^2 \log n + n^2 + n^2) = O(n^2 \log n)$. At each node $y$ of $\mathcal{T}_1$ (resp. $\mathcal{T}_2$) we explicitly store the set $U(y)$ and the cardinality of $MMD(y)$. Since the number of internal nodes is less than the number of leaf nodes, this additional storage still takes $O(n+m)$. Thus the space complexity of our algorithm is $O(n+m)$.

## 5.6 Conclusion and Open Problem

We gave an $O(n^2 \log n)$ algorithm to recognize whether two graphs are simultaneous interval graphs, where $n$ is the total number of vertices in both the graphs. This is equivalent to solving the sunflower interval augmentation problem for two graphs, and the interval sandwich problem when the set of optional edges induce a complete bipartite graph.

A natural open question is whether we can solve the problem for $r$-sunflower graphs for $r > 2$. We conjecture that this can be solved efficiently. Note that when the common vertices induce an independent set, this problem can be solved efficiently for interval graphs, by Theorem 5.2. In contrast, the simultaneous chordal representation problem for $r$-sunflower graphs is NP-hard, even when the common vertices induce an independent set (see Theorem 4.2).

# Chapter 6

# Simultaneous Comparability Graphs

In this chapter[1], we study the simultaneous comparability representation problem for multiple graphs and give an efficient algorithm for solving the problem.

A *comparability graph*, as defined in chapter 2, is a graph whose edges can be transitively oriented. Golumbic [44] gave an $O(nm)$ time algorithm ($n$ and $m$ are the number of vertices and number of edges respectively) for recognizing comparability graphs and constructing a transitive orientation if it exists. In this section we extend Golumbic's [44] results to simultaneous comparability graphs and show that the simultaneous comparability representation problem for $r$-sunflower graphs can also be solved in $O(nm)$ time, where $n$ and $m$ are the total number of vertices and the total number of edges.

We use the following notation in this chapter. If $A$ and $B$ are disjoint sets, we use $A + B$ to denote the disjoint-union of $A$ and $B$. A directed edge from $u$ to $v$ is denoted by $\overrightarrow{uv}$. If $A$ is a set of directed edges, then we use $A^{-1}$ to denote the set of edges obtained by reversing the direction of each edge in $A$. We use $\hat{A}$ to denote the union of $A$ and $A^{-1}$. $A$ is said to be *transitive* if for any three vertices $a, b, c$, we have $\overrightarrow{ab} \in A$ and $\overrightarrow{bc} \in A$ implies that $\overrightarrow{ac} \in A$. Our edge sets never include loops, so note that if $A$ is transitive then it cannot contain a directed cycle and must satisfy $A \cap A^{-1} = \emptyset$ (because if it contained both $\overrightarrow{ab}$ and $\overrightarrow{ba}$ it would contain $\overrightarrow{aa}$). An *orientation* of a graph is an assignment of directions to all its edges. A *transitive orientation* assigns a direction to each edge in such a way that the resulting set of directed edges is transitive. We use $G - A$ to denote the graph obtained by undirecting $A$ and removing it from graph $G$.

For the rest of this chapter we let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower comparability graphs sharing some vertices $I$ (and the edges induced by $I$). We use $E(I)$ to denote the set of edges induced by $I$ in any graph, say $G_1$. As mentioned in section 2.3, the simultaneous comparability representation problem for $G_i, i \in \{1, \ldots, r\}$, asks whether $G_1, G_2, \ldots, G_r$ can be transitively oriented in such a way that any edge in $E(I)$ (i.e. any common edge) is oriented in the same way in all the graphs. Also note that an *augmenting edge* is an edge whose end points appear in different graphs. In other words an augmenting edge belongs to the set $\{\bigcup (V_i - I) \times (V_j - I) \mid i, j \in \{1, \ldots, r\} \wedge i \neq j\}$. Let $G = G_1 \cup G_2 \cup \cdots \cup G_r$, $n = |V(G)|$ and $m = |E(G)|$. If $W \subseteq \widehat{E(G)}$, then $W$ is said to be *pseudo-transitive* if $W \cap \hat{E}_i$ is transitive for all $i \in \{1, \ldots, r\}$. $W$ is said to be a *pseudo-transitive orientation* if it is an orientation of $G$ and is pseudo-transitive. Thus by the definition of simultaneous comparability, $G_1, G_2, \ldots, G_r$ are

[1]The results presented in this chapter are joint work with Anna Lubiw [55].

simultaneous comparability graphs if and only if $G = G_1 \cup G_2 \cup \cdots \cup G_r$ has a pseudo-transitive orientation.

## 6.1 Equivalence with Sunflower Comparability Graphs

We now prove an alternative characterization of simultaneous comparability for sunflower graphs, in terms of the sunflower comparability augmentation problem. This is analogous to Theorem 2.1, which only applied for intersection classes. The main ingredient of the proof is to show that any pseudo-transitive orientation of $G$ can be augmented to a transitive orientation.

**Theorem 6.1.** *Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), G_r = (V_r, E_r)$ be $r$-sunflower comparability graphs sharing some vertices $I$ (and the edges induced by $I$). $G_i, i \in \{1, \ldots, r\}$ are augmentable to a comparability graph if and only if $G_1 \cup G_2 \ldots G_r$ has a pseudo-transitive orientation (or equivalently, $G_i, i \in \{1, \ldots, r\}$ are simultaneous comparability graphs).*

*Proof.* Let $G = G_1 \cup G_2 \cdots \cup G_r$.

Let $G_i, i \in \{1, \ldots, r\}$ be augmentable to a comparability graph. Then there exists a set $A \subseteq \{\bigcup (V_i - I) \times (V_j - I) \mid i, j \in \{1, \ldots, r\} \wedge i \neq j\}$ of augmenting edges such that the graph $G_A = G \cup A$ is a comparability graph. Let $T$ be a transitive orientation of $G_A$. For $i \in \{1, \ldots, r\}$, let $T_i$ be a (directed) subgraph of $T$ induced by $V_i$. Clearly $T_i$ is a transitive orientation of $G_i$. Further any edge in $E(I)$ gets the same orientation in $T_i$, for all $i$. Now it is easy to see that the orientation $T - \hat{A} = T_1 \cup T_2 \cup \cdots \cup T_r$ is a pseudo-transitive orientation of $G$.

For the other direction let $T$ be a pseudo-transitive orientation of $G$. We now extend $T$ to a transitive orientation $T'$ by adding a set $A'$ of (directed) augmenting edges. This is enough to show that $G_i, i \in \{1, \ldots, r\}$ are augmentable to a comparability graph. We define $A'$ as follows:

> For all distinct $i, j \in \{1, \ldots, r\}$ and all vertex triples $a, b, c$ with $a \in V_i - I$, $b \in I$ and $c \in V_j - I$, $\overrightarrow{ab} \in T$ and $\overrightarrow{bc} \in T \Rightarrow \overrightarrow{ac} \in A'$.

Now it is sufficient to prove that $T' = T \cup A'$ is transitive. We first show that for any two vertices $a, c \in V(G)$ exactly one of $\overrightarrow{ac}$ and $\overrightarrow{ca}$ can be in $T'$. Suppose both $\overrightarrow{ac} \in T'$ and $\overrightarrow{ca} \in T'$ hold. Since $T$ is pseudo-transitive, $\overrightarrow{ac}$ and $\overrightarrow{ca}$ cannot both belong to $T$. Suppose $\overrightarrow{ac} \in A'$ with $a \in V_i - I$ and $c \in V_j - I$. Then $\overrightarrow{ca}$ must be in $A'$ as well (not in $T$). Thus (by definition of $A'$) there exist vertices $b, d \in I$ such that $\overrightarrow{ab} \in T$, $\overrightarrow{bc} \in T$, $\overrightarrow{cd} \in T$ and $\overrightarrow{da} \in T$. Now $b, a, d \in V_i$, therefore $\overrightarrow{da} \in T$ and $\overrightarrow{ab} \in T$ implies that $\overrightarrow{db} \in T$. Similarly $b, c, d \in V_j$, therefore $\overrightarrow{bc} \in T$ and $\overrightarrow{cd} \in T$ implies that $\overrightarrow{bd} \in T$. Thus $T$ contains both $\overrightarrow{bd}$ and $\overrightarrow{db}$ which contradicts that $T$ is pseudo-transitive.

Now let $\overrightarrow{ab}$ and $\overrightarrow{bc}$ belong to $T'$. It is enough to show that $\overrightarrow{ac} \in T'$.
Case 1: $\overrightarrow{ab} \in T$ and $\overrightarrow{bc} \in T$
Assume without loss of generality that $a, b \in V_1$. If $c \in V_1$ then by transitivity of $T_1$, $\overrightarrow{ac} \in T_1 \subseteq T'$. Otherwise $c \in V_j - I$, for some $j$, which forces $b \in I$, so by definition of $A'$, $\overrightarrow{ac} \in A' \subseteq T'$.
Case 2: $\overrightarrow{ab} \in T$ and $\overrightarrow{bc} \in A'$
Since $\overrightarrow{bc} \in A'$, we can assume without loss of generality that $b \in V_1 - I$ and $c \in V_2 - I$. Also by definition of $A'$, there exists a vertex $d \in I$ such that $\overrightarrow{bd} \in T$ and $\overrightarrow{dc} \in T$. Now $\overrightarrow{ab} \in T$ implies

that $a \in V_1$ (since $b \in V_1$) and thus $a, b, d$ all belong to $V_1$. Since $\overrightarrow{ab} \in T$ and $\overrightarrow{bd} \in T$ we must have $\overrightarrow{ad} \in T$. Now if $a \in V_1 - I$ then $\overrightarrow{ac} \in A' \subseteq T'$ and if $a \in I$ then $ac \in T \subseteq T'$ (since $\{a, d, c\} \subseteq V_2$)
Case 3: $\overrightarrow{ab} \in A'$ and $\overrightarrow{bc} \in T$
This case is identical to case 3.
Case 4: $\overrightarrow{ab} \in A'$ and $\overrightarrow{bc} \in A'$
We can assume without loss of generality that $a \in V_1 - I$, $b \in V_2 - I$ and $c \in V_j - I$ for some $j \neq 2$. Now $\overrightarrow{ab} \in A'$ implies that there exists a vertex $d \in I$ such that $\overrightarrow{ad} \in T$ and $\overrightarrow{db} \in T$
Similarly $\overrightarrow{bc} \in A'$ implies that there exists a vertex $e \in I$ such that $\overrightarrow{be} \in T$ and $\overrightarrow{ec} \in T$
Since $\overrightarrow{db}, \overrightarrow{be} \in T$ and $\{d, b, e\} \in V_2$, $\overrightarrow{de} \in T$. Now $a, d, e$ all belong to $V_1$, hence $\{\overrightarrow{ad}, \overrightarrow{de}\} \subseteq T$ implies $\overrightarrow{ae} \in T \subseteq T'$. Now if $c \in V_1$ then $\overrightarrow{ac} \in T \subseteq T'$, else $\overrightarrow{ac} \in A' \subseteq T'$.

Thus in all cases $\overrightarrow{ac} \in T'$. Hence we conclude that $T'$ is transitive. This completes the proof. $\square$

## 6.2 Overview

We now sketch a high level overview of Golumbic's algorithm for recognizing comparability graphs and compare it with our approach. Golumbic's recognition algorithm is conceptually quite simple: orient one edge (call it a "seed" edge), and follow implications to orient further edges. If this process results in an edge being oriented both forwards and backwards, the input graph is rejected. Otherwise, when there are no further implications, the set of oriented edges (called an "implication class") is removed and the process repeats with the remaining graph. The correctness proof is not so simple, requiring an analysis of implication classes, and of how deleting one implication class changes other implication classes. Golumbic proves the following theorem.

**Theorem 6.2.** *(Golumbic [44]) Let $H = (V, E)$ be an undirected graph and let $\widehat{E(H)} = \hat{B}_1 + \hat{B}_2 + \cdots + \hat{B}_j$ be any decomposition of $H$, where for each $k \in \{1, \ldots, j\}$, $B_k$ is an "implication class" of $H - \cup_{1 \leq l < k} \hat{B}_l$. The following statements are equivalent:*

1. *$H$ is a comparability graph.*
2. *$A \cap A^{-1} = \emptyset$ for all implication classes $A$ of $H$.*
3. *$B_k \cap B_k^{-1} = \emptyset$ for $k = 1, \ldots, j$.*

We follow a similar strategy except that the "seed" edges must be chosen carefully for our proof to work. In the next section we will define the concept of a "composite class" which is analogous to an implication class. We further classify a composite class as a "base class" or a "super class" depending on whether it is disjoint from $E(I)$ or not. Our algorithm works as follows: As long as there is a base class, remove it and recursively orient the remaining graph. Otherwise (when there are no base classes left) as long as there is a super class, remove it and recursively orient the remaining graph.

We define an *S-decomposition* of $G_1 \cup G_2 \cup \cdots \cup G_r$, in the next section, and prove the following theorem.

**Theorem 6.3**. *Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower comparability graphs, sharing some vertices $I$ (and the edges induced by $I$). Let $G = G_1 \cup G_2 \cdots \cup G_r$ and let $\widehat{E(G)} = \hat{B}_1 + \hat{B}_2 + \cdots + \hat{B}_i + \hat{S}_{i+1} + \hat{S}_{i+2} + \cdots + \hat{S}_j$ be an "S-decomposition" of $G$. The following statements are equivalent.*

1. *$G_1, G_2, \ldots, G_r$ are simultaneous comparability graphs.*

2. *Every composite class of $G$ is pseudo-transitive, i.e. $C \cap C^{-1} = \emptyset$ for all composite classes $C$ of $G$.*

3. *Every part of the $S$-decomposition is pseudo-transitive, i.e. $B_k \cap B_k^{-1} = \emptyset$ for $k = 1, \ldots, i$ and $S_k \cap S_k^{-1} = \emptyset$ for $k = i+1, \ldots, j$.*

## 6.3 Algorithm for Simultaneous Comparability Graphs

We now formalize and justify the above defined notions. Given an undirected graph $H$, we can replace each undirected edge $(u, v)$ by two directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$ and define a relation $\Gamma$ on the directed edges of $H$ as explained below. Whenever there are edges $ab$ and $bc$ and no edge $ac$, then any transitive orientation of $H$ cannot use directions $\overrightarrow{ab}$ and $\overrightarrow{bc}$ nor can it use directions $\overrightarrow{cb}$ and $\overrightarrow{ba}$. This can be expressed as the constraint that we use $\overrightarrow{ab}$ iff $\overrightarrow{cb}$ and we use $\overrightarrow{ba}$ iff $\overrightarrow{bc}$. We define $\Gamma$ on the directed edges of $H$ to capture this:

$\overrightarrow{ab}\Gamma\overrightarrow{a'b'}$ if ($a = a'$ and $bb' \notin E(H)$) or ($b = b'$ and $aa' \notin E(H)$).

$\Gamma$ can be viewed as a constraint that directs the $ab$ edge from $a$ to $b$ if and only if the edge $a'b'$ is directed from $a'$ to $b'$. It is easy to see that the transitive closure of $\Gamma$, denoted by $\Gamma_t$, is an equivalence relation. We refer to the equivalence classes of $\Gamma_t$ as *implication classes*. The following lemmas capture some of the fundamental properties of implication classes.

**Lemma 6.1.** *([44]) Let $A$ be an implication class of a graph $H$. If $H$ has a transitive orientation $F$, then either $F \cap \hat{A} = A$ or $F \cap \hat{A} = A^{-1}$ and in either case, $A \cap A^{-1} = \emptyset$.*

**Lemma 6.2.** *([44]) Let the vertices $a, b, c$ induce a triangle in $H$ and let $\overrightarrow{bc}$, $\overrightarrow{ca}$ and $\overrightarrow{ba}$ belong to implication classes $A, B$ and $C$ respectively (see Figure 6.1). If $A \neq C$ and $A \neq B^{-1}$, then*

1. *If $\overrightarrow{b'c'} \in A$ then $\overrightarrow{b'a} \in C$ and $\overrightarrow{c'a} \in B$*

2. *No edge of $A$ is incident with $a$.*

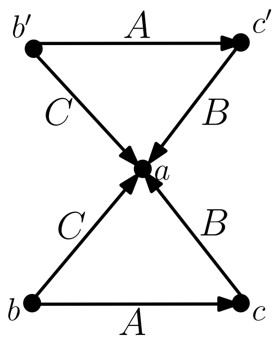


Figure 6.1: An example to illustrate Lemma 6.2.

**Lemma 6.3.** *([44]) Let $A$ be an implication class of a graph $H$. If $A \cap A^{-1} = \emptyset$, then $A$ is transitive.*

Note that in Lemma 6.2, if the directions of one or more edges of triangle $abc$ are reversed, then the lemma can still be applied by inversing the corresponding implication classes. For e.g when $\overrightarrow{ab} \in C$, $\overrightarrow{ac} \in B$ and $\overrightarrow{bc} \in A$, if $A \neq C^{-1}$ and $A \neq B$, then condition (1) becomes: If $\overrightarrow{b'c'} \in A$ then $\overrightarrow{ab'} \in C$ and $\overrightarrow{ac'} \in B$.

Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower comparability graphs, sharing some vertices $I$ (and the edges induced by $I$). Let $G = G_1 \cup G_2 \cup \cdots \cup G_r$. We define a relation $\Gamma'$ on the (directed) edges of $G$ as follows: $\overrightarrow{e}\Gamma'\overrightarrow{f}$ if $\overrightarrow{e}\Gamma\overrightarrow{f}$ and $e$ and $f$ belong to $E_i$ for some $i \in \{1, \ldots, r\}$. It is easy to see that the transitive closure of $\Gamma'$ denoted by $\Gamma'_t$ is an equivalence relation. We refer to the equivalence classes of $\Gamma'_t$ as *composite classes.*

From the definition, it follows that each composite class is a union of zero or more implication classes of $G_i$ for all $i \in \{1, \ldots, r\}$. If a composite class $C$ of $G$ has an edge that belongs to $E(I)$, then we use the term “super class” to refer to $C$. Otherwise $C$ is said to be a “base class”. Thus any base class is a single implication class of $G_i$ for some $i \in \{1, \ldots, r\}$ and is contained in $\hat{E}_i - \hat{E}(I)$. Figure 6.2 shows a pair of comparability graphs and their composite classes.

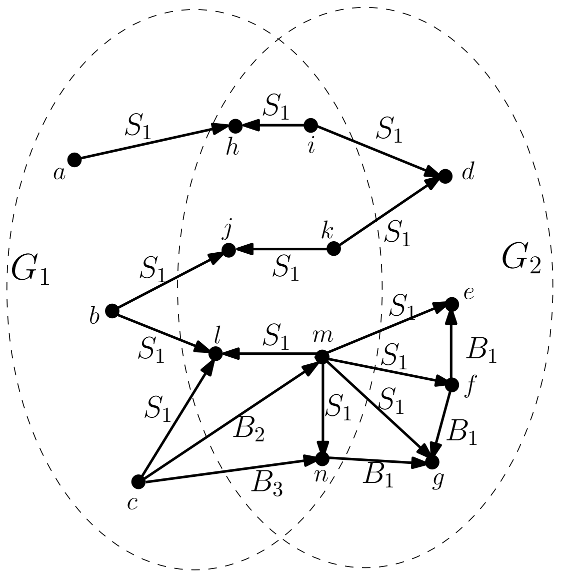


Figure 6.2: An instance of simultaneous comparability representation problem with two graphs $G_1$ and $G_2$ sharing vertices $\{h, i, j, k, l, m, n\}$. $S_1$ and its inverse are the super classes of $G_1 \cup G_2$ and $B_1, B_2, B_3$ and their inverses are the base classes. Thus there are 8 composite classes in total.

**Observation:** *Note that every implication class of a super class contains an edge $\overrightarrow{e} \in \hat{E}(I)$.*

The following lemmas for composite classes are analogous to Lemmas 6.1, 6.2 and 6.3.

**Lemma 6.4.** *Let $A$ be a composite class of $G$. If $F$ is a pseudo-transitive orientation of $G$ then either $F \cap \hat{A} = A$ or $F \cap \hat{A} = A^{-1}$ and in either case, $A \cap A^{-1} = \emptyset$.*

*Proof.* Let $Y$ be a set of directed augmenting edges of $G = G_1 \cup G_2 \cup \cdots \cup G_r$ such that $F' = F \cup Y$ is a transitive orientation. Let $G'$ be the graph obtained by undirecting $F'$. Thus $G'$ is an augmentation of $G$. Now any composite class of $G$ is contained in some implication class of $G'$, as any two edges that are related by $\Gamma'$ in $G$ are related by $\Gamma$ in $G'$. Let $A'$ be the implication class of $G'$ that contains $A$. Note that $A \subseteq A' - Y$. Now the lemma follows by applying Lemma 6.1 on $A'$ and $G'$. □

**Lemma 6.5.** *Let the vertices $a \in I$, $b$ and $c$ induce a triangle in $G$, such that $\overrightarrow{bc}$, $\overrightarrow{ca}$ and $\overrightarrow{ba}$ belong to composite classes $A, B$ and $C$ respectively. If $A \neq C$ and $A \neq B^{-1}$, then*

1. *If $\overrightarrow{b'c'} \in A$ then $\overrightarrow{b'a} \in C$ and $\overrightarrow{c'a} \in B$.*
2. *No edge of $A$ is incident with $a$.*

*Proof.* We cannot appeal immediately to Lemma 6.2 because although $\overrightarrow{bc}$ and $\overrightarrow{b'c'}$ belong to the same composite class $A$, they need not be in the same implication class.

Since $\overrightarrow{bc}, \overrightarrow{b'c'} \in A$, there exist a sequence of edges $\overrightarrow{b_1c_1}, \cdots, \overrightarrow{b_kc_k}$, from $A$, such that $\overrightarrow{bc}\Gamma'\overrightarrow{b_1c_1}\Gamma' \cdots \Gamma'$ $\overrightarrow{b_kc_k}\Gamma'\overrightarrow{b'c'}$. Assume inductively that (1) holds for $\overrightarrow{b_kc_k}$, i.e. $\overrightarrow{b_ka} \in C$ and $\overrightarrow{c_ka} \in B$. Now since $\overrightarrow{b_kc_k}\Gamma'\overrightarrow{b'c'}$, both $(b_k, c_k)$ and $(b', c')$ belong to $E_i$ for some $i \in \{1, \ldots, r\}$. Further $\overrightarrow{b_kc_k}\Gamma\overrightarrow{b'c'}$. Assume without loss of generality that $(b_k, c_k), (b', c') \in E_1$. Since $a \in I \subseteq V_1$, we have $\{b_k, b', c_k, c', a\} \subseteq V_1$. Let $A_1$, $B_1$ and $C_1$ be the implication classes of $G_1$ such that $\overrightarrow{b_kc_k} \in A_1$, $\overrightarrow{c_ka} \in B_1$, and $\overrightarrow{b_ka} \in C_1$. Note that $A_1 \subseteq A$, $B_1 \subseteq B$ and $C_1 \subseteq C$. Since $\overrightarrow{b_kc_k}\Gamma\overrightarrow{b'c'}$, we have $\overrightarrow{b'c'} \in A_1$. Now applying Lemma 6.2 on triangle $ab_kc_k$ and the edge $\overrightarrow{b'c'}$, we conclude that $\overrightarrow{c'a} \in B_1$ and $\overrightarrow{b'a} \in C_1$. Therefore $\overrightarrow{c'a} \in B$ and $\overrightarrow{b'a} \in C$.

For the second part, assume for the sake of contradiction that an edge of $A$ is incident with $a$. Then there exists a vertex $d$ such that either $\overrightarrow{ad} \in A$ or $\overrightarrow{da} \in A$. If $\overrightarrow{ad} \in A$, then by the first part of this theorem, $\overrightarrow{da} \in B$, and thus $A = B^{-1}$, a contradiction. Similarly, if $\overrightarrow{da} \in A$, then by the first part of this theorem, $\overrightarrow{da} \in C$, and thus $A = C$, a contradiction. This shows the second part. □

**Lemma 6.6.** *Let the vertices $a, b, c$ form a triangle in $G$ and let the edges $\overrightarrow{bc}, \overrightarrow{ca}$ and $\overrightarrow{ba}$ belong to composite classes $A, B$ and $C$ respectively with $A \neq C$, $A \neq B^{-1}$ and $B \neq C$. If $B$ and $C$ are both super classes then there exists a triangle $a', b', c'$ in $I$ with $\overrightarrow{b'c'} \in A$, $\overrightarrow{c'a'} \in B$ and $\overrightarrow{b'a'} \in C$ and hence $A$ is a super class.*

*Proof.* We can assume without loss of generality that $a, b, c \in V_1$. Let the edges $\overrightarrow{ca}$ and $\overrightarrow{ba}$ belong to implication classes $I_b$ and $I_c$ (respectively) of $G_1$ (thus $I_b \subseteq B$, $I_c \subseteq C$). Hence $I_b \cap E\hat{(}I) \neq \emptyset$, $I_c \cap E\hat{(}I) \neq \emptyset$. Let $\overrightarrow{b'a''} \in I_c \cap E\hat{(}I)$ and $\overrightarrow{c'a'} \in I_b \cap E\hat{(}I)$. Applying Lemma 6.2 on triangle $c, a, b$ and the edge $\overrightarrow{b'a''}$ (and noting that $A \neq C$, $A \neq B^{-1}$), we infer that $\overrightarrow{ca''} \in I_b$ and $\overrightarrow{b'c} \in A$. Now applying Lemma 6.2 again on triangle $b', c, a''$ and the edge $\overrightarrow{c'a'}$ (and noting that $B \neq C$ and $B \neq A^{-1}$), we infer that $\overrightarrow{b'a'} \in I_c$ and $\overrightarrow{b'c'} \in A$. This in turn implies that $A$ is a super class (since $b', c' \in I$). □

**Lemma 6.7.** *Let $A$ be a composite class of $G = G_1 \cup G_2 \cup \cdots \cup G_r$. If $A \cap A^{-1} = \emptyset$, then for all $i \in \{1, \ldots, r\}$, $A \cap \hat{E}_i$ is transitive and hence $A$ is pseudo-transitive.*

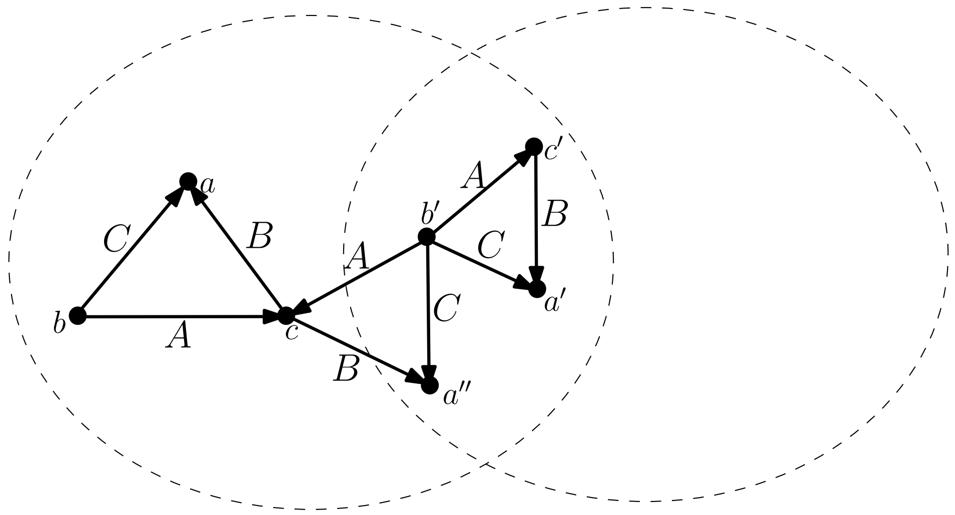


Figure 6.3: An example to illustrate Lemma 6.6.

*Proof.* If $A$ is a base class then $A$ is transitive by Lemma 6.3. Thus we can assume that $A$ is a super class. If $A$ doesn't satisfy the conclusion of the lemma then we can assume without loss of generality that there exist vertices $a, b, c \in V_1$ such that $\overrightarrow{ba} \in A$, $\overrightarrow{ac} \in A$ and $\overrightarrow{bc} \notin A$. If the edge $(b, c)$ is not present in $E_1$, then $\overrightarrow{ba}\Gamma'\overrightarrow{ca}$ and thus $\overrightarrow{ca} \in A \cap A^{-1}$. Therefore we can assume that $(b, c) \in E_1$. Let $C_a$ be the composite class that contains $\overrightarrow{bc}$. Now in triangle $abc$, we have $\overrightarrow{ba} \in A$ and $\overrightarrow{ca} \in A^{-1}$. Also both $A$ and $A^{-1}$ are super classes with $A \neq C_a$. Therefore by Lemma 6.6, $C_a$ must be a super class. Further there exists a triangle $a'b'c'$ in $I$ with $\overrightarrow{b'a'} \in A$, $\overrightarrow{a'c'} \in A$ and $\overrightarrow{b'c'} \in C_a$. But by the second condition of Lemma 6.5 (on triangle $b'c'a'$), $b'$ cannot be incident with an $A$ edge, a contradiction. Thus $C_a = A$ and we conclude that $A$ is pseudo-transitive. □

The following Corollary is a consequence of Lemma 6.7.

**Corollary 6.1.** *Let $A$ be a composite class of $G$. Then $A$ is pseudo-transitive iff $A \cap A^{-1} = \emptyset$.*

Recall that our approach involves deleting a composite class $A$ from $G$. Any composite class of $G - A$ is a union of composite classes of $G$ formed by successive "merging". Two composite classes $B$ and $C$ of $G$ are said to be *merged* by the deletion of class $A$, if deleting $A$ creates a $(\Gamma')$ relation between a $B$-edge and a $C$-edge. Note that for this to happen there must exist a triangle $a, b, c$ in $G$ with $(b, c) \in \hat{A}$ and either $\overrightarrow{ba} \in C$ and $\overrightarrow{ca} \in B$ or $\overrightarrow{ab} \in C$ and $\overrightarrow{ac} \in B$. We first iteratively delete all the base classes followed by the (remaining) super classes. The following lemmas examine what happens when a base or super class gets deleted by the algorithm.

**Lemma 6.8.** *If the composite classes of $G$ are all pseudo-transitive and $A$ is a base class of $G$ then the composite classes of $G - A$ are also pseudo-transitive.*

*Proof.* Let $C$ be any (composite) class of $G - A$. If $C$ is also a composite class of $G$ then it is pseudo-transitive by assumption. So assume that $C$ is formed by merging two or more composite classes of $G$.

**Claim 1.** *Let $B_1$ be a base class contained in $C$. If $B_1$ merges with another composite class $M$ then $B_1 \neq M^{-1}$ and $B_1$ does not merge with any other class.*

*Proof.* Since $C$ contains a merge of $B_1$ and $M$, there exists a triangle $a, b, c$ in $G$ such that one of the following conditions hold:

1. $\overrightarrow{ba} \in M$, $\overrightarrow{ca} \in B_1$ and $\overrightarrow{bc} \in A$.
2. $\overrightarrow{ba} \in M$, $\overrightarrow{ca} \in B_1$ and $\overrightarrow{cb} \in A$.
3. $\overrightarrow{ab} \in M$, $\overrightarrow{ac} \in B_1$ and $\overrightarrow{bc} \in A$.
4. $\overrightarrow{ab} \in M$, $\overrightarrow{ac} \in B_1$ and $\overrightarrow{cb} \in A$.

All these cases are symmetric and hence we assume without loss of generality that condition (1) holds. If $B_1 = M^{-1}$ then by Lemma 6.7, $A = M$, a contradiction. Let $B_2 \subseteq M$ be the implication class containing the edge $\overrightarrow{ba}$. Now it is enough to show that deleting $A$ wouldn't merge $B_1$ with some other implication class $D \neq B_2$ of $G$. To see this, suppose deleting $A$ merges $B_1$ with $D$. Then there exists an edge $\overrightarrow{b'c'} \in A$ which together with a $B_1$ edge and a $D$ edge forms a triangle $T$ in $G$. Let $a'$ be the other vertex of $T$.

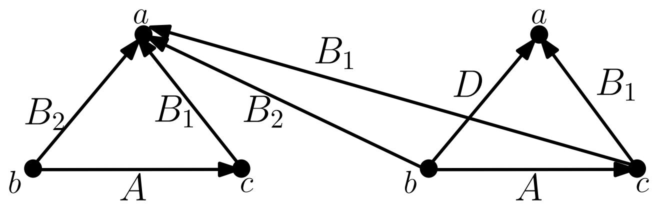


Figure 6.4: An example to illustrate Claim 1.

We claim that the $B_1$ edge of $T$ is $\overrightarrow{c'a'}$. To see this we first note that by Lemma 6.2, $\overrightarrow{b'a} \in B_2$ and $\overrightarrow{c'a} \in B_1$. Applying the second part of Lemma 6.2 on $b'c'a$ we infer that $b'$ cannot be adjacent to a $B_1$ edge. Also by the same lemma, applied on $b'c'a$ and $\overrightarrow{a'c'}$, if $\overrightarrow{a'c'} \in B_1$ then $\overrightarrow{b'c'} \in B_2$, a contradiction. Thus $\overrightarrow{c'a'} \in B_1$ and hence $\overrightarrow{b'a'} \in D$.

Now applying Lemma 6.2 again on $b'c'a$ and edge $\overrightarrow{c'a'}$, we infer that $D = B_2$. $\square$

**Claim 2.** *Two super classes don't merge in $C$.*

*Proof.* Assume (for the sake of contradiction) that two super classes $S_y$ and $S_z$ merge in $C$. Thus we can assume without loss of generality that there exists a triangle $a, y, z$ in $G$ with $\overrightarrow{yz} \in A$, $\overrightarrow{za} \in S_y$ and $\overrightarrow{ya} \in S_z$. (The other cases are symmetric as observed in Claim 1). Since $A$ is disjoint from $\hat{S}_y$ and $\hat{S}_z$ and $S_y \neq S_z$, the conditions of Lemma 6.6 are satisfied. Thus applying Lemma 6.6 on triangle $ayz$, we conclude that $A$ is a super class, a contradiction. $\square$

**Claim 3.** *If $C$ contains a super class, say $S$ (of $G$), then $C$ is of the form $C = S \cup B_1 \cup B_2 \cup \cdots \cup B_k$, where $B_1, \ldots, B_k$ are base classes of $G$ and $B_i \neq B_j^{-1}$ for $1 \leq i, j \leq k$. Otherwise $C$ is the union of two base classes $B_1$ and $B_2$ with $B_1 \neq B_2^{-1}$.*

*Proof.* This follows directly as a consequence of Claims 1 and 2. □

Claim 3 implies that $C \cap C^{-1} = \emptyset$ and hence $C$ is pseudo-transitive (by Lemma 6.7). □

**Lemma 6.9.** *Let each of the composite classes of $G = G_1 \cup G_2 \cup \cdots \cup G_r$ be super and pseudo-transitive. If $A$ is any super class of $G$ then each of the composite classes of $G - A$ is pseudo-transitive.*

*Proof.* Let $L$ be a composite class of $G - A$. If $L$ is also a composite class of $G$ then $L$ is pseudo-transitive by assumption. So assume that $L$ is not a composite class of $G$. We claim that $L$ consists of precisely a merge of two super classes. To see this let $L$ contains the merge of super classes $B$ and $C$. We can assume (without loss of generality) that there exists a triangle $abc$ in $G$ with $\overrightarrow{bc} \in A$, $\overrightarrow{ca} \in B$ and $\overrightarrow{ba} \in C$. Further by Lemma 6.6, we can assume that $\{a, b, c\} \in I$.

We now show that deleting $A$ wouldn't merge $B$ with some other super class $D \neq C$. The proof is parallel to that of Claim 1, though we can't appeal to Lemma 6.2 as we do there, and must use Lemma 6.5 instead. If $L$ contains a merge of $B$ with some other super class $D$, then there exists a triangle $T = a'b'c'$ in $G$ with $\overrightarrow{b'c'} \in A$ and the other two edges in $B$ and $D$. Further by Lemma 6.6, we can assume that $\{a', b', c'\} \subseteq I$.

We claim that the $B$ edge of $T$ must be $\overrightarrow{c'a'}$. To see this we first note that by Lemma 6.5, $\overrightarrow{b'a} \in C$ and $\overrightarrow{c'a} \in B$. Applying the second part of Lemma 6.5 on $b'c'a$ we infer that $b'$ cannot be adjacent to a $B$ edge. Also by the same lemma applied on $b'c'a$ and $\overrightarrow{a'c'}$, if $\overrightarrow{a'c'} \in B$ then $\overrightarrow{b'c'} \in C$, a contradiction. Thus $\overrightarrow{c'a'} \in B$ and $\overrightarrow{b'a'} \in D$.

Now applying Lemma 6.5 again on $b'c'a$ and the edge $\overrightarrow{c'a'}$, we infer that $D = C$. Therefore $B$ doesn't merge with any class other than $C$ and similarly $C$ doesn't merge with any class other than $B$. Hence $L$ consists of precisely a merge of two super classes $B$ and $C$ and therefore $L$ is pseudo-transitive (since $B \neq C^{-1}$). □

A partition of the edge set $E(\hat{G}) = \hat{B}_1 + \hat{B}_2 + \cdots + \hat{B}_i + \hat{S}_{i+1} + \hat{S}_{i+2} + \cdots + \hat{S}_j$ is said to be an *$S$-decomposition* of $G = G_1 \cup G_2 \cup \cdots \cup G_r$, if for each $k \in \{1, \ldots, i\}$, $B_k$ is a base class of $G - \cup_{1 \leq l < k} \hat{B}_l$ and for each $k \in \{i+1, \ldots, j\}$, $S_k$ is a super class of $G - \cup_{1 \leq l \leq i} \hat{B}_l - \cup_{i+1 \leq l < k} \hat{S}_l$

We are now ready to prove the main theorem.

**Theorem 6.3.** *Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower comparability graphs, sharing some vertices $I$ (and the edges induced by $I$). Let $G = G_1 \cup G_2 \cdots \cup G_r$ and $E(\hat{G}) = \hat{B}_1 + \hat{B}_2 + \cdots + \hat{B}_i + \hat{S}_{i+1} + \hat{S}_{i+2} + \cdots + \hat{S}_j$ be an $S$-decomposition of $G$. The following statements are equivalent.*

1. *$G_1, G_2, \ldots, G_r$ are simultaneous comparability graphs.*

2. *Every composite class of $G$ is pseudo-transitive, i.e. $C \cap C^{-1} = \emptyset$ for all composite classes $C$ of $G$.*

3. *Every part of the $S$-decomposition is pseudo-transitive, i.e. $B_k \cap B_k^{-1} = \emptyset$ for $k = 1, \ldots, i$ and $S_k \cap S_k^{-1} = \emptyset$ for $k = i+1, \ldots, j$.*

*Proof.* (1) $\Rightarrow$ (2) By Theorem 6.1 $G$ has a pseudo-transitive orientation. Thus the claim follows from Lemmas 6.4 and 6.7.

(2) $\Rightarrow$ (3) is a direct consequence of Lemmas 6.8 and 6.9.

(3) $\Rightarrow$ (1)

Let $T = B_1 + B_2 + \cdots + B_i + \cdots S_{i+1} + S_{i+2} + \cdots + S_j$. We now claim that $T$ is pseudo-transitive. For $k = 1, \ldots, j$, define $C_k$ as $C_k = B_k$ if $k \leq i$ and $C_k = S_k$ otherwise. Thus $T = C_1 + \cdots + C_j$.

For $k = 1 \ldots j$, let $T_k = C_k + \cdots + C_j$. (Thus $T_1 = T$) and $H_k = \hat{C}_k + \cdots \hat{C}_j$. Thus $C_k$ is a composite class of $H_k$. Now it is enough to show that $T_k$ is pseudo-transitive for any $k$. Assume inductively that $T_{k+1} = T_k - C_k$ is pseudo-transitive. Note that $\hat{T}_{k+1} \cap \hat{C}_k = \emptyset$. Now we claim that $T_k = T_{k+1} \cup C_k$ is also pseudo-transitive.

Suppose not. We can assume without loss of generality that there exist vertices $a, b, c$ all in $G_1$ such that $\overrightarrow{ab} \in T_k$, $\overrightarrow{bc} \in T_k$ and $\overrightarrow{ac} \notin T_k$. Since $T_{k+1}$ and $C_k$ are pseudo-transitive we only have to consider the case when $\overrightarrow{ab} \in T_{k+1}$ and $\overrightarrow{bc} \in C_k$ (the other case $\overrightarrow{ab} \in C_k$ and $\overrightarrow{bc} \in T_{k+1}$ is symmetric).

Now if the edge $(a, c)$ is not present in $H_k$ then $\overrightarrow{bc}\Gamma'\overrightarrow{ba}$ and thus $\overrightarrow{ba} \in C_k$, contradicting that $\hat{T}_{k+1} \cap \hat{C}_k = \emptyset$. So either $\overrightarrow{ca} \in T_{k+1}$ or $\overrightarrow{ca} \in C_k$. This implies (by the pseudo-transitivity of $T_{k+1}$ and $C_k$) that $\overrightarrow{cb} \in T_{k+1}$ or $\overrightarrow{ba} \in C_k$. In both cases we get a contradiction to $\hat{T}_{k+1} \cap \hat{C}_k = \emptyset$.

Hence we conclude that $T_k$ is pseudo-transitive. □

Theorem 6.3 gives rise to the following $O(nm)$ algorithm for determining whether a family of $r$-sunflower graphs are simultaneous comparability graphs: Given graphs $G_1, G_2, \ldots, G_r$ check whether all composite classes of $G = G_1 \cup G_2 \cup \cdots \cup G_r$ are pseudo-transitive. If so return YES otherwise return NO. Further, if $G_1, G_2, \ldots, G_r$ are simultaneous comparability graphs then the following algorithm computes an $S$-decomposition of $G$. As shown in the proof of Theorem 6.3, this immediately gives a pseudo-transitive orientation. In fact it gives $2^j$ pseudo-transitive orientations.

To compute a pseudo-transitive orientation (because of Theorem 6.3), we have to first iteratively select and delete base classes from $G_1 \cup G_2 \cup \cdots \cup G_r$, before selecting and deleting super classes. We compute a pseudo-transitive orientation as follows.

**Algorithm 2**

```
1. Initialize $T = \emptyset$ and $G' = G$.
2. Compute all base-classes $B_1$, $B_1^{-1}, \ldots, B_b, B_b^{-1}$.
3. For $i = 1$ to $b$, place all the edges of $B_i$ (resp. $B_i^{-1}$) in a separate set labelled $i$ (resp. $-i$).
4. Place all the remaining edges in one set and assign a label 0.
5. Let $S_{\overrightarrow{uv}}$ denote the set containing $\overrightarrow{uv}$.
6. While there exists a set $S$ with non-zero label Do:
7.    Add all (directed) edges of $S$ to $T$.
8.    Assign label 0 to $S$ and $S^{-1}$.
9.    For each edge $\overrightarrow{bc}$ in $S$ and each vertex $a$ in $G$ such that $abc$ forms a triangle Do:
10.      If labels of $S_{\overrightarrow{ab}}$ and $S_{\overrightarrow{ac}}$ are not equal:
11.         Let $l$ be a label defined as: $l = 0$ if labels of $S_{\overrightarrow{ab}}$ or $S_{\overrightarrow{ac}}$ is 0, otherwise $l =$ label of $S_{\overrightarrow{ab}}$.
12.         Merge $S_{\overrightarrow{ab}}$ and $S_{\overrightarrow{ac}}$ and assign label $l$ to the union.
13.         Merge $S_{\overrightarrow{ba}}$ and $S_{\overrightarrow{ca}}$ and assign label $-l$ to the union.
14.   End
```

```
15. End
16.Let $G' = G - \hat{T}$. /* Now each composite class of $G'$ is a super class. */
17. While $G'$ is non-empty Do:
18.     Let $C$ be a super class of $G'$.
19.     $T = T \cup C$ and $G' = G' - \hat{C}$.
20. End
21. Return $T$.
```

Given simultaneous graphs $G_1, G_2, \ldots, G_r$, Algorithm 2 computes the pseudo-transitive orientation of $G = G_1 \cup G_2 \cdots \cup G_r$ as follows. We first compute all the base classes and distinguish them from super classes using labels (lines 2,3 and 4). The algorithm iteratively orients all the base classes before orienting the super classes. In each iteration of the first while loop (line 6), we use the labels to find a base class, add it to the solution and delete it (lines 6–8). By Lemma 6.8 deletion of a base class may leave other composite classes unchanged, or merge two base classes or merge a super class with a set of base classes. We handle these cases in lines 9 to 13 and update the labels. Note that we label the super classes and the base classes that get deleted with '0'. After the while loop terminates, we are only left with super classes. These are handled in lines 17 to 21.

Algorithm 2 can be implemented to run in $O(nm)$ time using a disjoint-set data structure. Using a linked-list representation and a weighted-union heuristic (see [25]), we obtain $O(1)$ amortized time for the find operation and $O(\log n)$ time for the union operation. Since the number of find operations in the algorithm are greater than the number of union operations by a polynomial factor, we may assume that each set operation takes $O(1)$ amortized time. Now consider the run time of each of the steps in the algorithm: Computing all the composite classes (base and super) takes $O(nm)$ time. Thus line 2 takes $O(nm)$ time. Lines 3 and 4 take $O(m)$ time. In line 6, finding a set with non-zero label takes at most $O(m)$ time. In each iteration of the while loop, if the chosen set has $m_1$ elements, then the For loop (lines 9–14) takes $O(m_1 n)$ time. Hence the total run time of the while loop is $O((m_1 + m_2 + \cdots m_i)n)$ where $m_i$ is the size of the set chosen in the $i$th iteration of the algorithm. This in turn is at most $O(nm)$. Lines 16–21 also run in $O(nm)$ time. Hence the run time of Algorithm 2 is $O(nm)$.

Algorithm 2 can be improved to run faster for sparse graphs as follows. In the for loop of line 9, instead of visiting each vertex $a$ to check whether it forms a triangle with the edge $bc$, we can take the minimum degree vertex among $b$ and $c$, and visit each of its neighbors to check whether it forms a triangle with $bc$. Thus if each vertex has degree at most $d$, then the algorithm takes $O(md)$ time. Even if the vertex degrees are not bounded, this algorithm can be shown to have a better running time for sparse graphs, as follows.

Let $d > 0$ be any constant. For each edge $bc$, if one of the end vertices has degree at most $d$, then we spend $O(d)$ time for the edge, otherwise we may spend at most $O(n)$. The number of vertices with degree greater than $d$ is at most $m/d$. Thus the number of edges, for which we need to do more than $O(d)$ work is at most $(m/d)^2$. Hence the total running time of the algorithm is at most $O(md) + O(n(m/d)^2)$. By choosing $d$ to be $O((mn)^{1/3})$, we get a running time of $O(m^{4/3}n^{1/3})$.

**Remark:** Note that if $T$ is a pseudo-transitive orientation of $G$, then $T$ can be augmented to a transitive orientation by computing $T' = T \cup T^2$ (as shown in the proof of Theorem 6.1). This can be easily done in $O(nm)$ time as follows: Initialize $T'$ to $T$. For each edge $\overrightarrow{ab} \in T$ and each vertex

$c \in G$, if $\overrightarrow{ab} \in T$ and $\overrightarrow{bc} \in T$ then add $\overrightarrow{ac}$ to $T'$. Hence computing an augmented comparability graph takes $O(nm)$ steps.

## 6.4 Summary

We have given an $O(nm)$ algorithm for solving the simultaneous comparability representation problem for $r$-sunflower graphs, for arbitrary $r$. This is the same as solving the sunflower comparability augmentation problem for $r$-sunflower graphs and is equivalent to solving the comparability graph sandwich problem when the set of optional edges induces a complete $r$-partite graph. Hence our algorithm strictly generalizes the recognition algorithm for probe comparability graphs. Furthermore the currently known algorithm for recognizing probe comparability graphs also runs in $O(nm)$ time [19]. Also our algorithm implies that the sunflower co-comparability augmentation problem for $r$-sunflower graphs can be solved in $O(n^3)$ time, by taking the complements of the $r$-sunflower graphs and testing whether they are simultaneous comparability graphs. Since co-comparability graphs are intersection graphs, we then have an $O(n^3)$ algorithm to solve the simultaneous co-comparability representation problem for $r$-sunflower graphs.

In the next chapter, we use the algorithm for simultaneous comparability graphs presented in this chapter to obtain an efficient algorithm for solving the simultaneous permutation representation problem for $r$-sunflower graphs.

Finally, we note that a more general version of simultaneity can be studied for comparability graphs. Let $G_1$ and $G_2$ be two graphs that share some vertices $I$. If $G_1[I]$ is the same as $G_2[I]$ then $G_1$ and $G_2$ are 2-sunflower graphs. However, if we allow $G_1[I]$ to be different from $G_2[I]$, then the problem of testing whether $G_1$ and $G_2$ are simultaneous comparability graphs is an open problem.

# Chapter 7

# Simultaneous Permutation Graphs

In this chapter[1], we use the results on the simultaneous comparability representation problem for $r$-sunflower graphs to obtain an efficient algorithm for solving the simultaneous permutation representation problem for $r$-sunflower graphs. This implies that the sunflower permutation augmentation problem and the sunflower co-permutation augmentation problem can be solved efficiently for $r$-sunflower graphs.

Recall that a graph $H = (V, E)$ on vertices $V = \{1, \ldots, n\}$ is said to be a *permutation graph* if there exists a permutation $\pi$ of the numbers $1, 2, \ldots, n$ such that for all $1 \leq i < j \leq n$, $(i, j) \in E$ if and only if $\pi(i) > \pi(j)$. Equivalently, $H = (V, E)$ is a permutation graph if and only if there are two orderings $L$ and $P$ of $V$ such that $(u, v) \in E$ iff $u$ and $v$ appear in the opposite order in $L$ and in $P$. We call $\langle L, P \rangle$ an *order-pair* for $G$. The intersection representation for permutation graphs follows immediately: $H = (V, E)$ is a permutation graph iff there are two parallel lines $l$ and $p$ and a set of line segments each connecting a distinct point on $l$ with a distinct point on $p$ such that $H$ is the intersection graph of the line segments. Observe that $L$ and $P$ correspond to the ordering of the endpoints of the line segments on $l$ and $p$ respectively. Figure 7.1 shows a permutation graph and its intersection representation. Since permutation graphs are a class of intersection graphs, the equivalence Theorem 2.1 is applicable for this class.

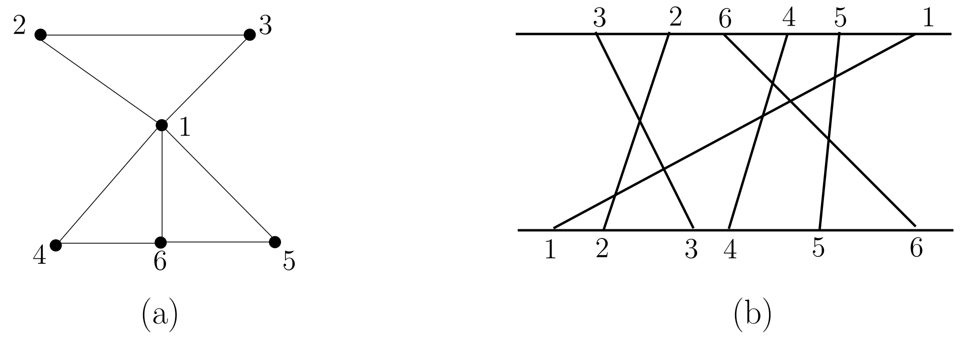


Figure 7.1: A permutation graph and its intersection representation.

Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower permutation graphs, sharing some vertices $I$ (and the edges induced by $I$). We begin with a "relaxed" characterization

[1]This results presented in this chapter are joint work with Anna Lubiw [55].

of simultaneous permutation graphs in terms of order-pairs.

**Lemma 7.1.** *$G_1, G_2, \ldots, G_r$ are simultaneous permutation graphs iff for every $i \in \{1, \ldots, r\}$ there exists an order-pair $\langle L_i, P_i \rangle$ of $G_i$, such that every pair of vertices $u, v \in I$ appear in the same order in all $L_i$ AND appear in the same order in all $P_i$.*

*Proof.* Let $G_1, G_2, \ldots, G_r$ be simultaneous permutation graphs. By Theorem 2.1, there exists a set $A$ of augmenting edges such that the graph $G_A = G_1 \cup G_2 \cdots \cup G_r \cup A$ is a permutation graph. Let $\langle L, P \rangle$ be an order pair of $G_A$ and for $i \in \{1, \ldots, r\}$ let $\langle L_i, P_i \rangle$ be an order-pair obtained from $\langle L, P \rangle$ by only considering the vertices of $G_i$. Clearly $\langle L_i, P_i \rangle$ is an order-pair of $G_i$ and further every pair of vertices $v, u \in I$ appear in the same order in all $L_i$ and appear in the same order in all $P_i$.

For the reverse direction, we create a total order $L$ on $V_1 \cup V_2 \cup \cdots \cup V_r$ consistent with all $L_i$, where $i \in \{1, \ldots, r\}$. This is possible because $L_i$ are consistent on $I$. Similarly we create a total order $P$ on $V_1 \cup V_2 \cup \cdots \cup V_r$ consistent with all $P_i$. The orderings $L$ and $P$ provide the endpoints of line segments for the simultaneous intersection representations of $G_1, G_2, \ldots, G_r$. □

It is known that a graph $H$ is a permutation graph if and only if $H$ and $\bar{H}$ are both comparability graphs [34]. Using this we can prove the following analogous result for simultaneous permutation graphs.

**Theorem 7.1.** *Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$-sunflower permutation graphs, sharing some vertices $I$ (and the edges induced by $I$). $G_i$ for $i = \{1, \ldots, r\}$ are simultaneous permutation graphs if and only if they are simultaneous comparability graphs and simultaneous co-comparability graphs.*

*Proof.* Let $G = G_1 \cup G_2 \cup \cdots \cup G_r$.

Let $G_i, i = \{1, \ldots, r\}$ be simultaneous permutation graphs. By Theorem 2.1 there exists an augmenting set of edges $A \subseteq \{\bigcup (V_i - I) \times (V_j - I) \mid i, j \in \{1, \ldots, r\} \wedge i \neq j\}$, such that $G_A = G \cup A$ is a permutation graph. Thus $G_A$ and $\bar{G_A}$ are comparability graphs and hence by Theorem 6.1, $G_i, i \in \{1, \ldots, r\}$ are simultaneous comparability graphs and $\bar{G_i}, i \in \{1, \ldots, r\}$ are simultaneous comparability graphs.

For the other direction, let $G_i, i \in \{1, \ldots, r\}$ be simultaneous comparability graphs and $\bar{G_i}, i \in \{1, \ldots, r\}$ also be simultaneous comparability graphs. For $i \in \{1, \ldots, r\}$, let $F_i$ be the transitive orientations of $G_i$ such that $F_i$ are consistent on the edges induced by $I$. Also let $R_i$ be the transitive orientations of $\bar{G_i}$, such that $R_i$ are consistent on the edges induced by $I$. As shown in [34], $F_i + R_i$ and $F_i^{-1} + R_i$ are both acyclic transitive orientations of $G_i$. Following the original idea of Pneuli et al. [34], we define an order-pair $\langle L_i, P_i \rangle$ on $V_i$ as follows: let $L_i$ be a total order of $V_i$ consistent with the partial order $F_i + R_i$; and let $P_i$ be a total order of $V_i$ consistent with the partial order $F_i^{-1} + R_i$.

We now show that any two vertices $u, v \in I$ satisfy the conditions of Lemma 7.1.
*Case 1.* $(u, v) \in E(G)$: Note that $(u, v) \in E_i$ for all $i \in \{1, \ldots, r\}$. Without loss of generality assume that the edge is directed from $u$ to $v$ in all $F_i$. Now for any $i \in \{1, \ldots, r\}$, $L_i(u) < L_i(v)$ (since $F_i + R_i$ is a transitive orientation). Similarly $P_i(u) > P_i(v)$ for all $i$.

*Case 2.* $(u, v) \in E(\bar{G})$: Note that $(u, v) \in \bar{E}_i$ for all $i \in \{1, \ldots, r\}$. Without loss of generality assume that the edge is directed from $u$ to $v$ in all $R_i$. We have $L_i(u) < L_i(v)$, and $P_i(u) < P_i(v)$ for all $i$.

From the above two cases, the conditions of Lemma 7.1 hold true for $G_i, i \in \{1, \ldots, r\}$ and hence we conclude that $G_1, G_2 \ldots, G_r$ are simultaneous permutation graphs. $\square$

Since simultaneous comparability representation problem for $r$-sunflower graphs can be solved $O(nm)$ time, Theorem 7.1 implies that simultaneous permutation representation problem for $r$-sunflower graphs can be solved in $O(n^3)$ time. We also note that a similar approach was used in [19] to obtain an $O(n^3)$ algorithm for probe permutation graphs. Since our result is equivalent to solving the sunflower permutation augmentation problem for $r$ graphs, for arbitrary $r$, it is more general than recognizing probe permutation graphs.

The best known algorithm for recognizing probe permutation graphs runs in $O(n^2)$ time [17, 20] and it is an open problem to determine whether we can solve the simultaneous permutation representation problem for $r$-sunflower graphs in $O(n^2)$ time.

# Chapter 8

# Simultaneous Planar Graphs

In this chapter[1] we study simultaneous planar graphs. As mentioned in section 3.1, the problem of testing simultaneous planarity for two graphs seems to be right on the feasibility boundary. The problem is NP-complete for three graphs [41] and the version where the planar drawings are required to be straight-line is already NP-hard for two graphs and only known to lie in PSPACE [33]. On the other hand several classes of (pairs of) graphs are known to always have simultaneous planar embeddings [32, 43, 39, 38, 57] and there are efficient algorithms to test simultaneous planarity for some very restricted graph classes: biconnected outerplanar graphs [38], and the case where one graph has at most one cycle [37].

This chapter shows how to efficiently test simultaneous planarity of any two graphs that share a 2-connected subgraph and thus greatly extends the classes of graph pairs for which a testing algorithm is known. Note that unlike in the previous chapters, the edges induced by the common vertices need not be the same in both the graphs. Our algorithm builds on the planarity testing algorithm of Haeupler and Tarjan [51], which in turn unifies the planarity testing algorithms of Lempel-Even-Cederbaum [64], Shih-Hsu [76] and Boyer-Myrvold [11].

We note that, at the same time and independent of our work Angelini et al. [1] showed how to test simultaneous planarity of two graphs when the common graph is 2-connected. Their algorithm is based on SPQR-trees, takes $O(n^3)$ time and is restricted to the case where the two graphs have the same vertex set. In comparison, our algorithm for testing simultaneous planarity of two graphs sharing a 2-connected subgraph runs in linear time and doesn't require the two graphs to have the same vertex set. Further, our algorithm can also solve the problem for multiple graphs, any two of which share the same 2-connected subgraph.

The rest of the chapter is organized as follows: We first review the methods used for testing planarity in section 8.1 and extend them in section 8.2 to obtain an algorithm for simultaneous planarity when the common graph is 2-connected. We then show that our algorithm can be extended to handle multiple graphs.

## 8.1 Planarity Testing Using PQ-trees

In this section, we review the recent algorithm of Haeupler and Tarjan [51] for testing the planarity of a graph. We begin with some basic definitions.

[1]The results in this chapter are joint work with Bernhard Haeupler and Anna Lubiw [50].

Let $G = (V, E)$ be a graph on vertex set $V = \{v_1, \dots, v_n\}$ and let $\mathcal{O}$ be an ordering of the vertices of $V$. An edge $v_iv_j$ is an *in-edge* of $v_i$ (in $\mathcal{O}$) if $v_j$ appears before $v_i$ in $\mathcal{O}$, and $v_iv_j$ is an *out-edge* of $v_i$ if $v_j$ appears after $v_i$ in $\mathcal{O}$.

An st-ordering of $G$ is an ordering $\mathcal{O}$ of the vertices of $G$, such that the first vertex of $\mathcal{O}$ is adjacent to the last vertex of $\mathcal{O}$ and every intermediate vertex has an in-edge and an out-edge. It is well-known that $G$ has an st-ordering if and only if it is 2-connected. Further, an st-ordering can be computed in linear time [35].

A *combinatorial planar embedding* of $G$, denoted by $\mathcal{C}(G)$, is defined as a clockwise circular ordering of the incident edges of $v_i$, for each $i \in \{1, \dots, n\}$, consistent with a planar drawing of $G$. Given a set of circular orderings, we can determine whether it is a combinatorial planar embedding by Euler's formula [31]. If $\mathcal{C}$ is a combinatorial planar embedding of $G$, we use $\mathcal{C}(v_i)$ to denote the clockwise circular ordering of edges incident with $v_i$ in $\mathcal{C}$.

Recall the definition of PQ-trees from section 2.2. Although PQ-trees were invented to represent linear orders, they can be reinterpreted to represent circular orders as well [51]: Given a PQ-tree we imagine that there is a new special leaf $s$ attached as the "parent" of the root. A circular leaf order of the augmented tree is a circular order that begins at the special leaf, followed by a linear order of the remaining PQ-tree and ending at the special leaf. Again, a PQ-tree represents all circular leaf-orders of equivalent PQ-trees. It is easy to see that a consecutivity constraint on such a set of circular orders directly corresponds to a consecutivity constraint on the original set of linear leaf-orders. Note that using PQ-trees for circular orders requires solely this different view on PQ-trees but does not need any change in their implementation.

Let $G = (V, E)$ be a connected graph. The planarity testing algorithm of Haeupler and Tarjan embeds vertices (and their edges) one at a time and maintains all possible partial embeddings of the embedded subgraph at each stage. For the correctness of the algorithm it is crucial that the vertices are added in a leaf-to-root order of a spanning tree. This guarantees that the remaining vertices induce a connected graph and hence lie in a single face of the partial embedding at any time. Without loss of generality we assume that the remaining vertices lie in the outer face of the partial embedding. We concentrate on two leaf-to-root orders: an st-order and a leaf-to-root order of a spanning tree. Using any one of these orders leads to particularly simple implementations that run in linear time. Indeed these two orders are essentially the only known orders in which the algorithm runs in linear time using the standard PQ-tree implementation. Our algorithm uses a mixture of the two orders: We first add the vertices that are contained in only one of the graphs using a depth-first search order and then add the common vertices using an st-ordering. We now give an overview of how the simple planarity test works for each of these orderings.

**st-order:**
Let $v_1, v_2, \dots, v_n$ be an st-order of $G$. At any stage $i \in \{1, \dots, n-1\}$ the vertices $\{v_1, \dots, v_i\}$ form a connected component and the algorithm maintains all possible circular orderings of out-edges around this component using a PQ-tree $T_i$. Since $v_1v_n$ is an out-edge at every stage, it can stay as the special leaf of $T_i$ for all $i$. At stage 1, the tree $T_1$ consists of the special leaf $v_1v_n$ and a P-node whose children are all other out-edges of $v_1$.

Now suppose we are at a stage $i \in \{1, \dots, n-2\}$. We call the set of leaves of $T_i$ that correspond to edges incident to $v_{i+1}$, the *black* leaves or the *black* edges. To go to the next stage, we first reduce $T_i$ so that all the black edges appear together. A non-leaf node in the reduced PQ-tree is said to be black if all its descendants are black edges. We next create a new P-node $p_{i+1}$ and add all the out-edges of $v_{i+1}$ as its children. Now $T_{i+1}$ is constructed from $T_i$ as follows:

**Case 1:** *$T_i$ contains a black node $x$ that is an ancestor of all the black leaves.* We obtain $T_{i+1}$ from $T_i$ by replacing $x$ and all its descendants with $p_{i+1}$.

**Case 2:** *$T_i$ contains a (non-black) Q-node containing a (consecutive) sequence of black children.* We obtain $T_{i+1}$ from $T_i$ by replacing these black children (and their descendants) with $p_{i+1}$.

Note that if the reduction step fails at any stage then the graph must be non-planar. Otherwise the algorithm concludes that the graph is planar.

**Leaf-to-root order of a depth-first spanning tree:**
Let $v_1, v_2, \ldots, v_n$ be a leaf-to-root order of a depth-first spanning tree of $G$. Note that at stage $i$, the vertices $\{v_1, \ldots, v_i\}$ may induce several components. We maintain a PQ-tree for each component representing the set of circular orderings of its out-edges. Using a depth-first spanning tree, in contrast to an arbitrary spanning tree, has the advantage that we can easily maintain the invariant that the spanning tree edge incident to the smallest node greater than $i$ will be the special leaf. This allows us to construct the PQ-trees of the next stage, without having to rotate the PQ-trees of the current stage. Adding $v_{i+1}$ can lead to merging several components into one.

To go to the next stage, we first reduce each PQ-tree corresponding to such a component by adding a consecutivity constraint that requires the set of out-edges that are incident to $v_{i+1}$ to be consecutive and then deleting these edges. By the invariant stated above the special leaf is among these edges. Note that the resulting PQ-tree for a component now represents the set of linear orders of the out-edges that are not incident to $v_{i+1}$. Now we construct the PQ-tree for the new merged component including $v_{i+1}$ as follows: Let $v_l$ be the parent of $v_{i+1}$ in the depth-first spanning tree. The PQ-tree for the new component consists of the edge $v_{i+1}v_l$ as the special leaf and a new P-node as a root and whose children are all the remaining out-edges of $v_{i+1}$ and the roots of the PQ-trees of the reduced components (similar to the picture in Figure 8.1, in which we replace the edge $v_1v_n$ with $v_{i+1}v_l$). Note that by choosing the edge $v_{i+1}v_l$ as the special leaf we again maintain the above-mentioned invariant.

As before, if the reduction step fails for any component, then the graph is non-planar. Otherwise the algorithm concludes that the graph is planar.

## 8.2 Simultaneous Planarity Testing When the Common Graph is 2-Connected

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two planar connected graphs with $|V_1| = n_1$ and $|V_2| = n_2$. Note that $G_1[V_1 \cap V_2]$ need not be the same as $G_2[V_1 \cap V_2]$. Let $G = (V_1 \cap V_2, E_1 \cap E_2)$ be 2-connected and $n = |V_1 \cap V_2|$. Let $v_1, v_2, \ldots, v_n$ be an st-ordering of $V_1 \cap V_2$. We call the edges and vertices of $G$ *common* and all other vertices and edges *private*.

We say two linear or circular orderings of elements with some common elements are *compatible* if the common elements appear in the same relative order in both orderings. Similarly we say two combinatorial planar embeddings of $G_1$ and $G_2$ respectively are compatible if for each common vertex the two circular orderings of edges incident to it are compatible.

If $G_1$ and $G_2$ have simultaneous planar embeddings, then clearly they have combinatorial planar embeddings that are compatible with each other. The converse also turns out to be true, if the common edges form a connected graph. This can be easily proved as follows. Let $\mathcal{E}_1$

and $\mathcal{E}_2$ be the compatible combinatorial planar embeddings of $G_1$ and $G_2$ respectively. Let $\mathcal{E}_p$ be the partial embedding of $\mathcal{E}_1$ [or $\mathcal{E}_2$] obtained by restricting $G_1$ [resp. $G_2$] to the common subgraph. (Note that since $\mathcal{E}_1$ and $\mathcal{E}_2$ are compatible, the partial embedding of $\mathcal{E}_1$ restricted to the common subgraph is the same as the partial embedding of $\mathcal{E}_2$ restricted to the common subgraph.) Now we can find a planar embedding of $\mathcal{E}_p$ and iteratively extend it to an embedding of $\mathcal{E}_1$ and an embedding of $\mathcal{E}_2$ (see Lemma 2 of Jünger and Schultz [57] for a proof). The two planar embeddings thus obtained are simultaneous planar embeddings and thus it is enough to compute a pair of compatible combinatorial planar embeddings.

We will find compatible combinatorial planar embeddings by adding vertices one by one, iteratively constructing two sets of PQ-trees, representing the partial planar embeddings of $G_1$ and of $G_2$ respectively. Each PQ-tree represents one connected component of $G_1$ or $G_2$. In the first phase we will add all private vertices of $G_1$ and $G_2$, and in the second phase we will add the common vertices in an st-order. When a common vertex is added, it will appear in two PQ-trees, one for $G_1$ and one for $G_2$ and we must take care to maintain compatibility.

Before describing the two phases, we give the main idea of maintaining compatibility between two PQ-trees. Given two PQ-trees $T_1$ and $T_2$ sharing some common elements, we can obtain compatible leaf-orderings of $T_1$ and $T_2$ as follows. We project $T_1$ and $T_2$ to the common elements and compute their intersection tree $T$. Now any leaf-ordering $\mathcal{S}$ of $T$ can easily be "lifted" back to leaf-orderings of $T_1$ and $T_2$ that respect the chosen ordering of $\mathcal{S}$ and hence are compatible. Furthermore any two compatible leaf-orderings of $T_1$ and $T_2$ can be obtained this way. However we cannot represent the set of compatible orderings with a PQ-tree. This is because a PQ-tree can only represent a set of consecutivity constraints among the elements of a ground set. A consecutivity constraint that only applies to a proper subset of the ground set may not be incorporated into the tree. For example, consider the ground set $\{1, 2, 3, 4\}$ and the constraint that says that $\{2, 3\}$ must be consecutive among $\{1, 2, 3\}$. In other words we cannot add the constraints of $T$ to $T_1$ (or $T_2$), in order to reduce it to a new PQ-tree. This can be a problem, since we need to compute compatible leaf-orderings for a sequence of pairs of PQ-trees.

To address this issue we introduce a boolean "orientation" variable attached to each Q-node to encode whether it is ordered forward or backward. Compatibility is captured by equations relating orientation variables. At the conclusion of the algorithm, it is a simple matter to see if the resulting set of Boolean equations has a solution. If it does, we use the solution to create compatible orderings of the Q-nodes of the two PQ-trees. Otherwise the graphs do not have simultaneous planar embeddings.

In more detail, we create a Boolean orientation variable $f(q)$ for each Q-node $q$, with the interpretation that $f(q) =$ true iff $q$ has a "forward" ordering. We record the initial ordering of each Q-node in order to distinguish "forward" from "backward". During PQ-tree operations, Q-nodes may merge, and during planarity testing, parts of PQ-trees may be deleted. We handle these modifications to Q-nodes by the simple expedient of having an orientation variable for each Q-node, and equating the variables as needed. When Q-nodes $q_1$ and $q_2$ merge, we add the equation $f(q_1) = f(q_2)$ if $q_1$ and $q_2$ are merged in the same order (both forward or both backward), or $f(q_1) = \neg f(q_2)$ otherwise.

We now describe the two phases of our simultaneous planarity testing algorithm. To process the private vertices of $G_1$ and $G_2$ in the first phase we compute for each graph a reverse depth-first ordering by contracting $G$ into a single vertex and then running a depth-first search from this vertex. With these orderings we can now run the algorithm of Haeupler and Tarjan for all private vertices as described in section 8.1.

Now the processed vertices induce a collection of components, such that each component has an out-edge to a common vertex. Further, the planarity test provides us for each component with an associated PQ-tree representing all possible cyclic orderings of out-edges for that component. For each component we look at the out-edge that goes to the first common vertex in the st-order and re-root the PQ-tree for this component to have this edge represented by the special leaf. This completes the first phase.

For the second phase we insert the common vertices in an st-order. The algorithm is similar to that described in section 8.1 for an st-order but in addition has to take care of merging in the private components as well. We first examine the procedure for a single graph. Adding the first common vertex $v_1$ is a special set-up phase; we will describe the general addition below. Adding $v_1$ joins some of the private components into a new component $C_1$ containing $v_1$.

For each of these private components we reduce the corresponding PQ-tree so that all the out-edges to $v_1$ appear together, and then delete those edges. Note that due to the re-rooting at the end of the first phase the special leaf is among those edges. Thus the resulting PQ-tree represents the linear orderings of the remaining edges. We now build a PQ-tree representing the circular orderings around the new component $C_1$ as follows: we take $v_1v_n$ as the special leaf, create a new P-node as a root and add all the out-edges of $v_1$ and the roots of the PQ-trees of the merged private components as children of the root (see Figure 8.1).

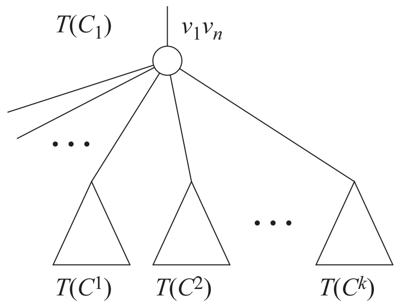


Figure 8.1: Setting up $T(C_1)$. The P-node's children are the outgoing edges of $v_1$ and the PQ-trees for the components that are joined together by $v_1$.

Now consider the situation when we are about to add the common vertex $v_i$, $i \geq 2$. The graph so far may have many connected components but because of the choice of an st-ordering all common vertices embedded so far are in one component $C_{i-1}$, which we call the *main* component. When we add $v_i$, all components with out-edges to $v_i$ join together to form the new main component $C_i$. This includes $C_{i-1}$ and possibly some private components. The other private components do not change, nor do their associated PQ-trees.

We now describe how to update the PQ-tree $T_{i-1}$ associated with $C_{i-1}$ to form the PQ-tree $T_i$ associated with $C_i$. This is similar to the approach described in section 8.1. We first reduce $T_{i-1}$ so that all the *black* edges (the ones incident to $v_i$) appear together. As before, we call a non-leaf node in the reduced PQ-tree *black* if all its descendants are black leaves. For any private component with an out-edge to $v_i$, we reduce the corresponding PQ-tree so that all the out-going edges to $v_i$ appear together, and then delete those edges. We make all the roots of the resulting PQ-trees into children of a new P-node $p_i$, and also add all the out-going edges of $v_i$ as children of $p_i$. It remains to add $p_i$ to $T_{i-1}$ which we do as described below. In the process we also create a *black tree* $J_i$ that represents the set of linear orderings of the black edges.

**Case 1:** *$T_{i-1}$ contains a black node $x$ such that all black edges are descendants of $x$.* Let $J_i$ be the subtree rooted at $x$. We obtain $T_i$ from $T_{i-1}$ by replacing $x$ and all its descendants with $p_i$.

**Case 2:** *$T_{i-1}$ contains a non-black Q-node $x$ that has a sequence of adjacent black children.* We group all the black children of $x$ and add them as children (in the same order) of a new Q-node $x'$. Let $J_i$ be defined as the subtree rooted at $x'$. We add an equation relating the orientation variables of $x$ and $x'$. We obtain $T_i$ from $T_{i-1}$ by replacing the sequence of black children of $x$ (and their descendants) with $p_i$ (see Figure 8.2).

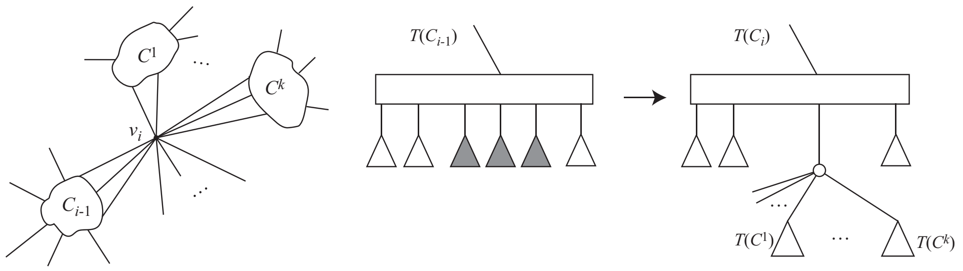


Figure 8.2: (*left*) Adding vertex $v_i$ which is connected to main component $C_{i-1}$ and to private components $C^1, \ldots, C^k$. (*right*) Creating $T_i$ from $T_{i-1}$ by replacing the black subtree by a P-node whose children are the outgoing edges of $v_i$ and the PQ-trees for the newly joined private components.

Note that we use orientation variables above for a purpose other than compatibility. (We are only working with one graph so far.) Standard planarity tests would simply keep track of the order of the deleted subtree $J_i$ in relation to its parent. Since we have orientation variables anyway, we use them for this purpose.

We perform a similar procedure on graph $G_2$. We will distinguish the black trees of $G_1$ and $G_2$ using superscripts. Thus after adding $v_i$ we have black trees $J_i^1$ and $J_i^2$. It remains to deal with compatibility. We claim that it suffices to enforce compatibility between each pair $J_i^1$ and $J_i^2$.

To do so, we perform a *unification step* in which we add equations between orientation variables for Q-nodes in the two trees.

### Unification step for stage $i$

We first project $J_i^1$ and $J_i^2$ to the common edges, as described in section 2.2.1, carrying over orientation variables from each original node to its copy in the projection (if it exists). Next we create the PQ-tree $R_i$ that is the intersection of these two projected trees as described in section 2.2.1. Initially $R_i$ is equal to the first tree. The step dealing with Q-nodes (Step 3) is enhanced as follows:

3. For each Q-node $q$ of the second tree, and for each pair $a_1$, $a_2$ of adjacent children of $q$ do the following: Reduce $R_i$ by adding a consecutivity constraint on all the descendant leaves of $a_1$ and $a_2$. Find the Q-node that is the least common ancestor of the descendants of $a_1$ and $a_2$ in $R_i$. Add an equation relating the orientation variable of this ancestor with the orientation variable of $q$ (using a negation if needed to match the orderings of the descendants).

Observe that any equations added during the unification step are necessary. Thus if the system of Boolean equations is inconsistent at the end of the algorithm, we conclude that $G_1$ and $G_2$ do not have a compatible combinatorial planar embedding. Finally, if the system of Boolean equations has a solution, then we obtain compatible leaf-orders for each pair $J_i^1$ and $J_i^2$ as follows: Pick an arbitrary solution to the system of Boolean equations. This fixes the truth values of all orientation variables and thus the orientations of all Q-nodes in all the trees. Subject to this, choose a leaf ordering $I$ of $R_i$ (by choosing the ordering of any P-nodes). $I$ can then be lifted back to (compatible) leaf-orders of $J_i^1$ and $J_i^2$ that respect the ordering of $I$. The following lemma shows that this is sufficient to obtain compatible combinatorial planar embeddings of $G_1$ and $G_2$

**Lemma 8.1.** *If the system of Boolean equations has a solution then $G_1$ and $G_2$ have compatible combinatorial planar embeddings.*

*Proof.* The procedure described above produces compatible leaf orders for all pairs of black trees $J_i^1$ and $J_i^2$. Recall that the leaves of $J_i^1$ (resp. $J_i^2$) are the out-edges of the component $C_{i-1}$ in $G_1$ (resp. $G_2$) and contain all the common in-edges of $v_i$. Focussing on $G_1$ individually, its planarity test has succeeded, and we have a combinatorial planar embedding such that the ordering of edges around $v_i$ contains the leaf order of $J_i^1$. Also, we have a combinatorial planar embedding of $G_2$ such that the ordering of edges around $v_i$ contains the leaf order of $J_i^2$.

The embedding of a graph imposes an ordering of the out-edges around every main component. We can show inductively, starting from $i = n$, that the ordering of the out-edges around the main component $C_{i-1}$ in $G_1$ is compatible with the ordering of the out-edges in the corresponding main component in $G_2$. Moreover all the common edges incident to $v_i$ belong to either $C_{i-1}$ or $C_i$. This implies that in both embeddings, the orderings of edges around any common vertex are compatible. Therefore $G_1$ and $G_2$ have compatible combinatorial planar embeddings. □

### 8.2.1 A Generalization of Simultaneous Planarity to $r$ Graphs

In this subsection we consider a generalization of simultaneous planarity for $r$ graphs, when each vertex [edge] is either present in all the graphs or present in exactly one of them. Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2), \cdots, G_r = (V_r, E_r)$ be $r$ planar graphs such that $V = V_i \cap V_j$ for all distinct $i, j$ and $E = E_i \cap E_j$ for all distinct $i, j$. As before, we call the edges and vertices of $G = (V, E)$ common and all other edges and vertices private. We show that the algorithm of section 8.2 can be readily extended to solve this generalized version, when the common graph $G$ is 2-connected.

If $G_1, G_2, \ldots, G_r$ have simultaneous planar embeddings then they clearly have mutually compatible combinatorial planar embeddings. Conversely if $G_1, G_2, \ldots, G_r$ have combinatorial planar embeddings that are mutually compatible, then, as before (see the beginning of section 8.2), we can first find the planar embedding of the common subgraph and extend it to the planar embeddings of $G_1, G_2, \ldots, G_r$. Thus once again, the problem is equivalent to finding combinatorial planar embeddings for $G_1, G_2, \ldots G_r$, that are mutually compatible.

Our algorithm for finding such an embedding works as before, inserting private vertices first followed by common vertices. The only difference comes in the unification step, where we have to take the intersection of $r$ projected trees instead of 2. Doing this is straightforward: We initialize the intersection tree to be the first projected tree, and then insert the constraints of all the other trees into the intersection tree. Finally, Lemma 8.1 and its proof can be trivially extended to multiple graphs.

### 8.2.2 Running Time

We show that our algorithm can be implemented to run in linear time. (In the generalization to $r$ graphs, the run time is linear in the sum of the sizes of the graphs—in other words, a common vertex counts $r$ times.) Computing the reverse depth-first ordering and the st-ordering are known to be feasible in linear time [35]. The first phase of our algorithm uses PQ-tree based planarity testing with a reverse depth-first search order [51], which runs in linear time using the efficient PQ-tree implementation of Booth and Lueker [9, 10]. The re-rooting between the two phases needs to be done only once and can easily be done in linear time. The second phase of our algorithm uses PQ-tree based planarity testing with an st-order, as discussed in section 8.1. This avoids re-rooting of PQ-trees, and thus also runs in linear time [51, 10, 64]. The other part of the second phase is the unification step, which is only performed on the black trees, i.e. the edges connecting to the current vertex. Note that these edges will get deleted and will not appear in subsequent stages. Thus we can explicitly store the black trees and the intersection tree at every stage and allow the unification step to take time linear in the complete size of both black trees. The intersection algorithm can be implemented in linear time, as mentioned in section 2.2.1. The last thing that needs to be implemented efficiently is the handling of the orientation variables. It is easy to see that once the equations are generated, they can be solved in linear time, by repeatedly fixing the value of a free variable to be true or false, finding all the equations that contain the variable and recursively (say in a depth-first way) fixing all the variables so as to satisfy the equations. In the following sub-section we explain how to also generate the variable equations in linear time. With this we conclude that the algorithm runs in linear time.

#### Generating variable equations in linear time

Note that in the implementation of PQ-trees (see Booth and Lueker [10]) the children of a Q-node are stored in a doubly-linked list and only the left most and right most children have parent pointers. Thus when two Q-nodes one a child of the other merge, we may not know the variable and the orientation of the parent Q-node. To address this problem, we use labels on certain links of the doubly-linked list as explained below and compute all the equations generated by the reductions of a unification step at the end of the step.

For any two adjacent child nodes $c_i$ and $c_{i+1}$ of a Q-node $q$, either the links $c_i \rightarrow c_{i+1}$ and $c_{i+1} \rightarrow c_i$ are labelled with $l$ and $\neg l$ (respectively), for some literal $l$, or they are both unlabeled. The underlying interpretation is that $c_i$ appears before $c_{i+1}$ in the child ordering of $q$ iff $l$ is true. Thus the literals that we encounter when travelling from one end to the other of the doubly-linked list, are all (implicitly) equal. When a Q-node is first created with two child nodes, say $x$ and $y$, we create a variable associated with it and label the link from $x$ to $y$ with the variable and the link from $y$ to $x$ with the negation of the variable.

During the algorithm, there are two types of equations: (1) Equations consisting of literals appearing in Q-nodes of distinct trees. (These can be PQ-trees of the same graph, as happens in Case 2 of section 8.2 or PQ-trees of different graphs, as happens in step 3 of Unification.) (2) Equations consisting of literals appearing in Q-nodes of a single tree (created during PQ-tree reductions).

Note that type 1 equations are essentially equations that constrain the ordering of child nodes across the two Q-nodes, and we handle them as follows. Let $c_1, c_2$ be any two adjacent child nodes of the first Q-node that are constrained to appear in the same order as child nodes $c'_1, c'_2$

of the second Q-node. If the links between $c_1$ and $c_2$ are unlabeled, we create a new variable $x$ and label the links $c_1 \to c_2$ and $c_2 \to c_1$ with $x$ and $\neg x$ respectively. Similarly, we label the links between $c'_1$ and $c'_2$, if they are unlabeled. Now we create the equation that equates the literal associated with $c_1 \to c_2$ with the literal associated with $c'_1 \to c'_2$.

Type 2 equations happen when two Q-nodes merge. In this case the merged node contains literals from both the Q-nodes. Finding an equation after each merge is costly, as we need to scan through each Q-node until we find a literal. However we can compute the equations in a lazy fashion at the end of each unification step as follows. For every Q-node of the two black trees and the intersection tree obtained from their projections (i.e. the output tree of the unification step on the black trees), we pass from the first child to the last child and equate all the literals encountered in the labels of the links. This clearly takes linear time in the size of the black trees.

## 8.3 Summary and Open Problems

We have given a linear-time algorithm for testing simultaneous planarity of two graphs that share a 2-connected subgraph. Our algorithm doesn't require the two graphs to have the same vertex set. Further our algorithm works for the more general case when there are $r$ graphs, any two of which share the same 2-connected subgraph. In our algorithm, 2-connectivity restricts each graph to have at most one main component at any stage, thus making the unification step simpler.

A big open question is to determine whether simultaneous planarity of two graphs can be tested in polynomial time. We conjecture that the problem can be solved in polynomial time when the common graph is 1-connected. Note that in this case the problem is still equivalent to finding compatible combinatorial planar embeddings for the two graphs.

We note that the NP-hardness result for testing simultaneous planarity of three graphs [41], uses edges in the following categories: (1) private to one of the graphs; (2) common to all the graphs; and (3) common to a pair of graphs but not the third, for every pair of graphs. In this chapter, we solved the case when there are no edges in category (3) and when the common graph is 2-connected. An open problem is whether simultaneous planarity for three graphs is NP-hard when every edge [vertex] is either private or common to all the graphs and the common graph is not necessarily 2-connected.

Another way to generalize simultaneous planarity (for 2 graphs) is to allow certain edges that are common to be drawn in different ways in both the graphs. Such an edge belongs to both the graphs, but is considered private. The complexity of this problem is open even when the common graph is 2-connected. We note if all the common edges are allowed to be drawn in different ways in both the graphs, then the two graphs can always be drawn simultaneously, by arbitrary fixing the vertex locations and drawing each graph independently (see Pach and Wenger [72]).

# Chapter 9

# Conclusions and Discussion

One of the main results of this thesis is a linear-time algorithm for testing simultaneous planarity when the common graph is 2-connected. The algorithm can be extended to multiple graphs that share a common 2-connected subgraph. Our algorithm is simple and is an extension of the PQ-tree based planarity testing algorithm of Haeupler and Tarjan [51].

The other main contribution of the thesis is to initiate the study of simultaneous representation problem for intersection graphs and comparability graphs. For a positive integer $r$, we have defined a family of $r$ graphs called the $r$-sunflower graphs, and studied the simultaneous representation problem on them. For 2-sunflower graphs, we have given efficient algorithms to recognize simultaneous chordal graphs, simultaneous interval graphs, simultaneous comparability graphs and simultaneous permutation graphs. Our algorithms for these classes run in $O(n^3)$, $O(n^2 \log n)$, $O(nm)$ and $O(n^3)$ time respectively. Also, our algorithms for simultaneous comparability and simultaneous permutation graphs extend to $r$-sunflower graphs, for arbitrary $r$. On the other hand, we have proved that recognizing whether a family of $r$-sunflower graphs are simultaneous chordal graphs, is NP-hard, even when the common vertices induce an independent set. Table 9.1 gives a summary of our results for comparability and intersection graph classes.

For a graph class $\mathcal{C}$ and integer $r$, we have defined the sunflower $\mathcal{C}$ augmentation problem for $r$-sunflower graphs, which is equivalent to the simultaneous $\mathcal{C}$ representation problem for $r$-sunflower graphs. The sunflower $\mathcal{C}$ augmentation problem for $r$-sunflower graphs is a generalization of the probe $\mathcal{C}$ recognition problem and is equivalent to the $\mathcal{C}$ sandwich problem, where the set of optional edges induce a complete $r$-partite graph.

Our results for simultaneous chordal graphs and simultaneous interval graphs imply that the sunflower chordal augmentation problem, the sunflower co-chordal augmentation problem, the

| **Summary of our results for comparability and intersection graphs** | | |
|---|---|---|
| **Graph Class** | **2-sunflower graphs** | **$r$-sunflower graphs** |
| Simultaneous Chordal Graphs | $O(n^3)$ | NP-hard |
| Simultaneous Interval Graphs | $O(n^2 \log n)$ | Open |
| Simultaneous Comparability Graphs | $O(nm)$ | $O(nm)$ |
| Simultaneous Permutation Graphs | $O(n^3)$ | $O(n^3)$ |

Table 9.1: Summary of our algorithmic and complexity results for simultaneous comparability graphs and simultaneous intersection graphs. Note that $r$ is a variable.

sunflower interval augmentation problem and the sunflower co-interval augmentation problem can all be solved efficiently for 2-sunflower graphs.

Similarly our results for simultaneous comparability and simultaneous permutation graphs imply that the sunflower comparability augmentation problem, the sunflower co-comparability augmentation problem, the sunflower permutation augmentation problem and the sunflower co-permutation augmentation problem can all be solved efficiently for $r$-sunflower graphs, for arbitrary $r$.

Many open questions arise from this thesis. For simultaneous planarity, a natural open question is, whether simultaneous planarity for 2 graphs can be tested in polynomial time, when the common graph is 1-connected. Recall that when the common graph is 1-connected, the problem reduces to finding compatible combinatorial embeddings of the two graphs. The approach taken in chapter 8 doesn't seem to work for this general case, as adding common vertices one after another will create different components in the two graphs that do not have a one-to-one correspondence. Keeping track of the constraints generated by these components seem to be complicated. However we conjecture that this problem can be solved in polynomial time.

For simultaneous intersection and simultaneous comparability representation problems, although we focus on $r$-sunflower graphs, the problems are interesting for graphs that intersect in more general way. One possibly tractable and interesting case is when there are a sequence of graphs, one in each layer, such that each vertex is present in a contiguous set of layers.

Another major question arising from the thesis is to determine whether the simultaneous interval representation problem for $r$-sunflower graphs can be solved efficiently. Solving this for arbitrary $r$, implies that the interval graph sandwich problem can be efficiently solved when the set of optional edges induce a complete $r$-partite graph and hence this would generalize probe interval graphs. We conjecture that this problem has a poly-time algorithm. We believe that our algorithm for simultaneous interval graphs may be extendable to more than 2 graphs. However to keep the number of cases small, it has to be simplified.

For chordal graphs, the simultaneous chordal representation problem for $r$-sunflower graphs is open, when $r$ is a constant. (It is NP-hard when $r$ is a variable.) Our techniques for solving the problem for 2-sunflower graphs do not seem to be extendable to 3-sunflower graphs.

Another obvious open problem is to improve the running time of the existing algorithms. Our algorithm for recognizing simultaneous comparability graphs matches the running time of the best known algorithm for recognizing probe comparability graphs. However the best known algorithms for recognizing probe chordal, probe interval and probe permutation graphs run in $O(nm)$, $O(n+m)$ and $O(n^2)$ time respectively. It is an open problem to determine whether simultaneous chordal, simultaneous interval and simultaneous permutation graphs can also be recognized in these times respectively.

Finally, it would be interesting to study the complexity of the sunflower augmentation problem for other classes of graphs including proper interval graphs, circular arc graphs, perfect graphs and strongly chordal graphs.

# References

[1] Patrizio Angelini, Giuseppe Di Battista, Fabrizio Frati, Maurizio Patrignani, and Ignaz Rutter. Testing the simultaneous embeddability of two graphs whose intersection is a biconnected graph or a tree. In *IWOCA 10: International Workshop on Combinatorial Algorithms*, LNCS, 2010. 17, 69

[2] Patrizio Angelini, Markus Geyer, Michael Kaufmann, and Daniel Neuwirth. On a tree and a path with no geometric simultaneous embedding. *CoRR*, abs/1001.0555, 2010. 16

[3] Brenda S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM*, 41(1):153–180, 1994. 2

[4] Hans-Jürgen Bandelt and Henry Martyn Mulder. Distance-hereditary graphs. *Journal of Combinatorial Theory Series B*, 41(2):182–208, 1986. 10

[5] Daniel Bayer, Van Bang Le, and H. N. de Ridder. Probe threshold and probe trivially perfect graphs. *Theoretical Computer Science*, 410(47-49):4812–4822, 2009. 20

[6] Anne Berry, Martin Charles Golumbic, and Marina Lipshteyn. Two tricks to triangulate chordal probe graphs in polynomial time. In *SODA 04: 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 992–969, Philadelphia, PA, USA, 2004. SIAM. 20

[7] Jean R. S. Blair and Barry W. Peyton. An introduction to chordal graphs and clique trees. *IMA Volumes in Mathematics and its Applications*, 56:1–30, 1993. 9

[8] Hans L. Bodlaender, Michael R. Fellows, and Tandy Warnow. Two strikes against perfect phylogeny. In *ICALP 92: 19th International Colloquium on Automata, Languages and Programming*, volume 623 of *LNCS*, pages 273–283. Springer, 1992. 24

[9] Kellogg Booth. *PQ Tree Algorithms*. PhD thesis, University of California, Berkeley, 1975. 13, 76

[10] Kellogg Booth and George Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976. 12, 26, 76

[11] John M. Boyer and Wendy J. Myrvold. On the cutting edge: Simplified $o(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004. 69

[12] Andreas Brandstädt, Feodor F. Dragan, Hoàng-Oanh Le, Van Bang Le, and Ryuhei Uehara. Tree spanners for bipartite graphs and probe interval graphs. *Algorithmica*, 47(1):27–51, 2007. 20

[13] Peter Braß, Eowyn Cenek, Christian A. Duncan, Alon Efrat, Cesim Erten, Dan P. Ismailescu, Stephen G. Kobourov, Anna Lubiw, and Joseph S. B. Mitchell. On simultaneous planar graph embeddings. *Computational Geometry Theory and Applications*, 36(2):117–130, 2007. 3, 16

[14] Peter Buneman. A characterization of rigid circuit graphs. *Discrete Math*, 9:205–212, 1974. 18

[15] Jérémie Chalopin and Daniel Gonçalves. Every planar graph is the intersection graph of segments in the plane. In *STOC 09: 41st Annual ACM Symposium on Theory of Computing*, pages 631–638. ACM, 2009. 10

[16] David B. Chandler, Maw-Shang Chang, Ton Kloks, Van Bang Le, and Sheng-Lung Peng. Probe ptolemaic graphs. In *COCOON 08: 14th Annual International Computing and Combinatorics Conference*, volume 5092 of *LNCS*, pages 468–477. Springer, 2008. 20

[17] David B. Chandler, Maw-Shang Chang, Ton Kloks, Jiping Liu, and Sheng-Lung Peng. On probe permutation graphs (extended abstract). *TAMC 06: 6th Annual Conference on Theory and Applications of Models of Computation*, pages 494–504, 2006. 68

[18] David B. Chandler, Maw-Shang Chang, Ton Kloks, Jiping Liu, and Sheng-Lung Peng. Recognition of probe cographs and partitioned probe distance hereditary graphs. In *AAIM 06: 2nd International Conference on Algorithmic Aspects in Information and Management*, volume 4041 of *LNCS*, pages 267–278. Springer, 2006. 20

[19] David B. Chandler, Maw-Shang Chang, Ton Kloks, Jiping Liu, and Sheng-Lung Peng. Partitioned probe comparability graphs. *Theoretical Computer Science*, 396(1-3):212–222, 2008. 20, 65, 68

[20] David B. Chandler, Maw-Shang Chang, Ton Kloks, Jiping Liu, and Sheng-Lung Peng. On probe permutation graphs. *Discrete Applied Mathematics*, 157(12):2611–2619, 2009. 68

[21] David B. Chandler, Maw-Shang Chang, Ton Kloks, and Sheng-Lung Peng. *Probe Graphs*. 2009. 20

[22] David B. Chandler, Jiong Guo, Ton Kloks, and Rolf Niedermeier. Probe matrix problems: Totally balanced matrices. In *AAIM 07: 3rd International Conference on Algorithmic Aspects in Information and Management*, volume 4508 of *LNCS*, pages 368–377. Springer, 2007. 20

[23] Gerard Jennhwa Chang, Ton Kloks, Jiping Liu, and Sheng-Lung Peng. The pigs full monty - a floor show of minimal separators. In *STACS 05: 22nd Symposium on Theoretical Aspects of Computer Science*, volume 3404 of *LNCS*, pages 521–532. Springer, 2005. 20

[24] Maw-Shang Chang, Ling-Ju Hung, and Peter Rossmanith. Probe distance-hereditary graphs. In *CATS 10: The 16th Computing: the Australasian Theory Symposium*, pages 55–65, 2010. 20

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (third edition)*. MIT Press, 2009. 64

[26] Simone Dantas, Celina M. Herrera de Figueiredo, and L. Faria. On decision and optimization $(k, l)$-graph sandwich problems. *Discrete Applied Mathematics*, 143(1-3):155–165, 2004. 18

[27] Simone Dantas, Sulamita Klein, Célia P. de Mello, and Aurora Morgana. The graph sandwich problem for $P_4$-sparse graphs. *Discrete Mathematics*, 309(11):3664–3673, 2009. 18

[28] Celina M. Herrera de Figueiredo, Guilherme Dias da Fonseca, Vinícius G. P. de Sá, and Jeremy Spinrad. Algorithms for the homogeneous set sandwich problem. *Algorithmica*, 46(2):149–180, 2006. 18

[29] Celina M. Herrera de Figueiredo, L. Faria, Sulamita Klein, and R. Sritharan. On the complexity of the sandwich problems for strongly chordal graphs and chordal bipartite graphs. *Theoretical Computer Science*, 381(1-3):57–67, 2007. 19

[30] Celina M. Herrera de Figueiredo, Sulamita Klein, and Kristina Vuskovic. The graph sandwich problem for 1-join composition is NP-complete. *Discrete Applied Mathematics*, 121(1-3):73–82, 2002. 19

[31] Jack R. Edmonds. A combinatorial representation for polyhedral surfaces. *Notices of the American Mathematical Society*, 7, 1960. 70

[32] Cesim Erten and Stephen G. Kobourov. Simultaneous embedding of planar graphs with few bends. *Journal of Graph Algorithms and Applications*, 9(3):347–364, 2005. 16, 17, 69

[33] Alejandro Estrella-Balderrama, Elisabeth Gassner, Michael Jünger, Merijam Percan, Marcus Schaefer, and Michael Schulz. Simultaneous geometric graph embeddings. In *GD 07: 15th International Symposium on Graph Drawing*, volume 4875 of *LNCS*, pages 280–290. Springer, 2008. 16, 69

[34] Shimon Even, Amir Pnueli, and Abraham Lempel. Permutation graphs and transitive graphs. *Journal of the ACM*, 19:400–410, 1972. 67

[35] Shimon Even and Robert E. Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2(3):339–344, 1976. 70, 76

[36] Martin Farber. Characterizations of strongly chordal graphs. *Discrete Mathematics*, 43(2-3):173–189, 1983. 10

[37] J. Joseph Fowler, Carsten Gutwenger, Michael Jünger, Petra Mutzel, and Michael Schulz. An SPQR-tree approach to decide special cases of simultaneous embedding with fixed edges. In *GD 08: 16th International Symposium on Graph Drawing*, volume 5417 of *LNCS*, pages 157–168. Springer, 2009. 17, 69

[38] J. Joseph Fowler, Michael Jünger, Stephen G. Kobourov, and Michael Schulz. Characterizations of restricted pairs of planar graphs allowing simultaneous embedding with fixed edges. In *WG 08: 33rd International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 5344 of *LNCS*, pages 146–158. Springer, 2008. 17, 69

[39] Fabrizio Frati. Embedding graphs simultaneously with fixed edges. In *GD 06: 14th International Symposium on Graph Drawing*, volume 3472 of *LNCS*, pages 108–113. Springer, 2007. 17, 69

[40] Tibor Gallai. Transitiv orientierbare Graphen. *Acta Mathematica Academiae Scientiarum Hungaricae*, 18:25–66, 1967. 11

[41] Elisabeth Gassner, Michael Jünger, Merijam Percan, Marcus Schaefer, and Michael Schulz. Simultaneous graph embeddings with fixed edges. In *WG 06: 32nd International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 4271 of *LNCS*, pages 325–335. Springer, 2006. 17, 69, 77

[42] Markus Geyer, Michael Kaufmann, and Imrich Vrto. Two trees which are self-intersecting when drawn simultaneously. *Discrete Mathematics*, 309(7):1909–1916, 2009. 16

[43] Emilio Di Giacomo and Giuseppe Liotta. Simultaneous embedding of outerplanar graphs, paths, and cycles. *International Journal of Computational Geometry and Applications*, 17(2):139–160, 2007. 17, 69

[44] Martin Charles Golumbic. *Algorithmic Graph Theory And Perfect Graphs*. Academic Press, New York, 1980. 2, 9, 10, 11, 24, 54, 56, 57, 58

[45] Martin Charles Golumbic and Clinton F. Goss. Perfect elimination and chordal bipartite graphs. *Journal of Graph Theory*, 2:155–193, 1978. 10

[46] Martin Charles Golumbic, Haim Kaplan, and Ron Shamir. Graph sandwich problems. *Journal of Algorithms*, 19(3):449–473, 1995. 17, 18

[47] Martin Charles Golumbic, Doron Rotem, and Jorge Urrutia. Comparability graphs and intersection graphs. *Discrete Mathematics*, 43(1):37–46, 1983. 9

[48] Martin Charles Golumbic and Ron Shamir. Complexity and algorithms for reasoning about time: A graph-theoretic approach. *Journal of ACM*, 40(5):1108–1133, 1993. 18

[49] Michel Habib, David Kelly, Emmanuelle Lebhar, and Christophe Paul. Can transitive orientation make sandwich problems easier? *Discrete Mathematics*, 307:2030–2041, 2007. 19

[50] Bernhard Haeupler, Krishnam Raju Jampani, and Anna Lubiw. Testing simultaneous planarity when the common graph is 2-connected. In *ISAAC 10: The 21st International Symposium on Algorithms and Computation*, LNCS. Springer, 2010. 69

[51] Bernhard Haeupler and Robert Endre Tarjan. Planarity algorithms via PQ-trees (extended abstract). *Electronic Notes in Discrete Mathematics*, 31:143–149, 2008. 12, 69, 70, 76, 78

[52] John H. Halton. On the thickness of graphs of given degree. *Inf. Sci.*, 54(3):219–238, 1991. 17

[53] Ryan B. Hayward. Weakly triangulated graphs. *Journal of Combinatorial Theory, Series B*, 39(3):200–208, 1985. 10

[54] Fáry István. On straight-line representation of planar graphs. *Acta Scientiarum Mathematicarum*, 11:229–233, 1948. 2

[55] Krishnam Raju Jampani and Anna Lubiw. The simultaneous representation problem for chordal, comparability and permutation graphs. In *WADS 09: 11th Algorithms and Data Structures Symposium*, volume 5664 of *LNCS*, pages 387–398. Springer, 2009. 22, 54, 66

[56] Krishnam Raju Jampani and Anna Lubiw. Simultaneous interval graphs. In *ISAAC 10: The 21st International Symposium on Algorithms and Computation*, LNCS. Springer, 2010. 26

[57] Michael Jünger and Michael Schulz. Intersection graphs in simultaneous embedding with fixed edges. *Journal of Graph Algorithms and Applications*, 13(2):205–218, 2009. 16, 17, 19, 69, 72

[58] Haim Kaplan and Ron Shamir. Bounded degree interval sandwich problems. *Algorithmica*, 24(2):96–104, 1999. 18

[59] David C. Kay and Gary Chartrand. A characterization of certain ptolemaic graphs. *Canadian Journal of Mathematics*, 17:342–346, 1996. 11

[60] Paul Koebe. Kontaktprobleme der konformen Abbildung. *Ber. Sächs. Akad. Wiss. Leipzig, Math.-Phys. KI.*, 88:141–164, 1936. 2

[61] Van Bang Le and H. N. de Ridder. Probe split graphs. *Discrete Mathematics & Theoretical Computer Science*, 9(1), 2007. 20

[62] Van Bang Le and H.N. de Ridder. Characterisations and linear-time recognition of probe cographs. In *WG 07: 33rd International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 4769 of *LNCS*, pages 226–237. Springer, 2007. 20

[63] Cornelis G. Lekkerkerker and J. Ch. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math*, 51:45–64, 1962. 9, 20

[64] Abraham Lempel, Shimon Even, and Israel Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs: International Symposium*, pages 215–232, 1967. 69, 76

[65] Federico Mancini. *Graph Modificiation Problems Related to Graph Classes*. PhD thesis, University of Bergen, Norway, 2008. 20

[66] Ross M. McConnell and Yahav Nussbaum. Linear-time recognition of probe interval graphs. In *ESA 09: 17th Annual European Symposium on Algorithms*, volume 5757 of *LNCS*, pages 349–360. Springer, 2009. 19

[67] Ross M. McConnell and Jeremy P. Spinrad. Construction of probe interval models. In *SODA 02: 13th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 866–875, Philadelphia, PA, USA, 2002. SIAM. 19

[68] Fred R. McMorris, Chi Wang, and Peisen Zhang. On probe interval graphs. *Discrete Appl. Math.*, 88(1-3):315–324, 1998. 19

[69] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins University Press, 2001. 2

[70] Takao Nishizeki and Norishige Chiba. *Planar graphs: Theory and Algorithms*. Elsevier, 1988. 2

[71] Takao Nishizeki and Md. Saidur Rahman. *Planar Graph Drawing*. World Scientific, 2004. 1, 2

[72] János Pach and Rephael Wenger. Embedding planar graphs at fixed vertex locations. *Graphs and Combinatorics*, 17(4):717–728, 2001. 17, 77

[73] Natasa Przulj and Derek G. Corneil. 2-tree probe interval graphs have a large obstruction set. *Discrete Applied Mathematics*, 150(1-3):216–231, 2005. 20

[74] Fred S. Roberts. *Graph Theory and its Applications to Problems of Society*. SIAM, 1978. 2

[75] Edward R. Scheinerman. *Intersection classes and multiple intersection parameters of graphs*. PhD thesis, Princeton University, NJ, USA, 1984. 10

[76] Wei Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theoretical Computer Science*, 223(1-2):179–191, 1999. 69

[77] Jeremy Spinrad. *Efficient Graph Representations*. Fields Institute Monographs. American Mathematical Society, 2003. 1, 2, 3, 9, 11, 18

[78] Michael A. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9(1):91–116, 1992. 24

[79] Rafael B. Teixeira, Simone Dantas, and Celina M. Herrera de Figueiredo. Skew partition sandwich problem is NP-complete. *Electronic Notes in Discrete Mathematics*, 35:9–14, 2009. 19

[80] Ryuhei Uehara. Canonical data structure for interval probe graphs. In *ISAAC 04: The 15th International Symposium on Algorithms and Computation*, volume 3341 of *LNCS*, pages 859–870. Springer, 2004. 20

[81] Peisen Zhang, Eric A. Schon, Stuart G. Fischer, Eftihia Cayanis, Janie Weiss, Susan Kistler, and Philip E. Bourne. An algorithm based on graph theory for the assembly of contigs in physical mapping of DNA. *CABIOS*, 10:309–317, 1994. 19