Path Graphs and PR-trees

by

Steven Chaplick

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2012 by Steven Chaplick
The \textit{PR-tree} data structure is introduced to characterize the sets of path-tree models of path graphs. We further characterize the sets of directed path-tree models of directed path graphs with a slightly restricted form of the PR-tree called the \textit{Strong PR-tree}. Additionally, via PR-trees and Strong PR-trees, we characterize path graphs and directed path graphs by their \textit{Split Decompositions}. Two distinct approaches (\textit{Split Decomposition} and \textit{Reduction}) are presented to construct a PR-tree that captures the path-tree models of a given graph $G = (V, E)$ with $n = |V|$ and $m = |E|$. An implementation of the split decomposition approach is presented which runs in $O(nm)$ time. Similarly, an implementation of the reduction approach is presented which runs in $O(A(n + m)nm)$ time (where $A(s)$ is the inverse of Ackermann’s function arising from Union-Find [40]). Also, from a PR-tree, an algorithm to construct a corresponding Strong PR-tree is given which runs in $O(n + m)$ time. The sizes of the PR-trees and Strong PR-trees produced by these approaches are $O(n + m)$ with respect to the given graph. Furthermore, we demonstrate that an implicit form of the PR-tree and Strong PR-tree can be represented in $O(n)$ space.
I am happy to thank the people who have made the work involved with this thesis an enjoyable and successful process.

First and foremost I would like to thank my supervisor, Dr. Derek Corneil. His enthusiasm, support, insight, and expertise have made this work an excellent experience.

Many thanks also go to an anonymous referee (for the journal version of this work, which is still making its way through the review process). This referee suggested the use of Split Decomposition and this has led to many of the results contained in this thesis.

Finally, I must thank my girlfriend, Marina, who has been my muse throughout this work, providing inspiration and encouragement every step of the way.
Contents

1 Introduction 1
  1.1 Outline ................................................................. 6

2 Background 7
  2.1 The Clique Tree Theorem ........................................... 7
  2.2 The Characterization Problems ..................................... 9
    2.2.1 Path-Tree, Directed Path-Tree, and Path-Path Problems .... 9
    2.2.2 Constraint Sets and Models .................................... 11
  2.3 PQ-trees ............................................................... 14
    2.3.1 PQ-trees: Definition and Results [2] ......................... 14
    2.3.2 Constraints and PQ-trees ....................................... 16
  2.4 Forbidden Induced Subgraph Characterization (FISC) of Path Graphs .. 19

3 Weak PR-trees, PR-trees, and Strong PR-trees 22
  3.1 The Weak PR-tree ................................................... 23
    3.1.1 Defining Weak PR-trees ......................................... 23
    3.1.2 The Constraint Set ............................................. 27
    3.1.3 Relating Graphs and Weak PR-trees via Constraint Sets .... 29
  3.2 Formalizing PR-trees and Strong PR-trees ........................... 31
    3.2.1 The Consistent Set ............................................. 32
    3.2.2 Connecting $\mathcal{T}_{SD}$ and $\text{Consistent}(D)$ ............ 38
3.3 Path-Tree Constraints of a Weak PR-tree ........................................ 42
  3.3.1 Least Common Ancestors ................................................. 42
  3.3.2 The Weak PR-tree Representing a Constraint Set ...................... 49
3.4 The size of a weak PR-tree ................................................... 53
3.5 Making (strong) PR-trees from weak PR-trees ............................... 55
  3.5.1 Strong Orientation ..................................................... 55
  3.5.2 Using the Algorithm from [4] .......................................... 57

4 Weak PR-tree Construction via Split Decomposition .......................... 62
  4.1 Split Decomposition ....................................................... 63
  4.2 Joining PR-trees Across a Split ......................................... 67
    4.2.1 Notation and Context ............................................... 67
    4.2.2 The Join Operation .................................................. 68
    4.2.3 The Correctness of the Join Operation .............................. 71
  4.3 Prime Chordal Graphs and their Weak PR-trees ............................ 77
  4.4 Implementation Details ................................................... 82
    4.4.1 Modified Weak PR-trees ............................................ 82
    4.4.2 Constructing MPR-trees using Split Decomposition .................. 83
    4.4.3 Runtime Complexity ................................................. 88
  4.5 Consequences of Split Decomposition .................................... 92

5 Weak PR-tree Construction via Reduction ....................................... 97
  5.1 Connected Reduction ........................................................ 98
  5.2 Weak PR-tree Reduction .................................................... 100
  5.3 The Reduce Operation ..................................................... 104
    5.3.1 Irreducibility and Template Terminology ............................ 105
    5.3.2 The Templates ....................................................... 108
    5.3.3 Finalizing the Reduce Operation ................................... 111
Chapter 1

Introduction

The focus of this thesis is the study of two intersection families of graphs; namely, path graphs (the intersection graphs of paths in trees) and directed path graphs (the intersection graphs of directed paths in directed trees). In this work, we present a new data structure (the PR-tree) to characterize these graph classes via their corresponding intersection models.

All graphs considered here are finite, have no parallel edges, and no loops. Also, when discussing the intersection of subgraphs of a graph we mean vertex (rather than edge) set intersection. For a graph $G = (V, E)$, we use the following notation. Let $n$ and $m$ denote the number of vertices and edges of $G$ respectively. For a subset $V'$ of $V$, $G[V']$ is the subgraph of $G$ induced by $V'$. A clique $C$ of $G$ is a subset of $V$ where every pair is adjacent. Let $\mathcal{K}_G$ denote the set of maximal cliques of $G$, and for every vertex $v \in V$, we let $S_v$ denote the set of maximal cliques of $G$ that contain $v$ (i.e., $S_v = \{C : C \in \mathcal{K}_G$ and $v \in C\}$). For a vertex $v$, the neighbourhood $N(v)$ of $v$ is the set of vertices adjacent to $v$. We also use $[i, j]$ to denote the set $\{i, i+1, ..., j\}$ for integers $i$ and $j$ where $i \leq j$.

A tree is a connected graph with no cycles. Furthermore, a directed tree is a tree where each edge is treated as an ordered pair. Notice that, in a directed tree, we can have multiple vertices with no edges that end at them (i.e., multiple “source vertices”
are possible). A rooted tree $T$ is a directed tree with a single source vertex (note: since $T$ is a directed tree, there must be at least one source). Finally, a directed path is a rooted tree with a single sink (i.e., a path whose edges are all oriented in the same direction).

A graph $G = (V = \{v_0, v_1, ..., v_{n-1}\}, E)$ is an intersection graph (as defined in [34]) if there exists a collection of sets $F = \{S_0, S_1, ..., S_{n-1}\}$ such that $v_i$ is adjacent to $v_j$ iff $S_i \cap S_j \neq \emptyset$ (the set $F$ is called an intersection representation of $G$). For a collection of sets $F = \{S_0, S_1, ..., S_{n-1}\}$ we use $I(F)$ to denote the graph $G$, where $V(G) = F$ and $E(G) = \{S_iS_j : i, j \in [0, n-1], i \neq j, S_i \cap S_j \neq \emptyset\}$. It is not difficult to see that all graphs are intersection graphs (i.e., by using the incident edges of a vertex $v$ as its set $S_v$). In light of this, additional structure is imposed on the collection of sets in order to develop restricted families of graphs. A graph $H$ whose vertex set is $\bigcup_{i=0}^{n-1} S_i$ is referred to as a host graph for the collection of sets $F$. Furthermore, for each $i \in [0, n-1]$, $H[S_i]$ is referred to as a guest graph of $H$. Thus, by imposing restrictions on the host graph and guest graphs we can develop various restricted families of graphs.

A graph is chordal when it has no induced $k$-cycle, for $k \geq 4$. In [18] Gavril showed that the chordal graphs are the intersection graphs of subtrees of a tree. In particular, a graph $G$ with $V(G) = \{v_0, v_1, ..., v_{n-1}\}$, is chordal iff:

There is a tree $T$, and a collection \{t_0, t_1, ..., t_{n-1}\} of subtrees of $T$, such that: $T = \bigcup_{i=0}^{n-1} t_i$ and $G = I(\{t_0, t_1, ..., t_{n-1}\})$ (i.e., $\forall i, j \in [0, n-1], v_iv_j \in E(G)$ $\iff$ $V(t_i) \cap V(t_j) \neq \emptyset$).

We call such a tree and collection of subtrees a tree-tree model of $G$. See Fig. 1.1 (on page 4) for various chordal graphs and their tree-tree models.

An interval graph is the intersection graph of a set of intervals on the real line; equivalently, it is the intersection graph of subpaths of a path. In particular, a graph $G$ with $V(G) = \{v_0, v_1, ..., v_{n-1}\}$, is interval iff:

There is a path $P$ and a collection \{p_0, p_1, ..., p_{n-1}\} of subpaths of $P$, such that: $P = \bigcup_{i=0}^{n-1} p_i$ and $G = I(\{p_0, p_1, ..., p_{n-1}\})$. 

We call such a path and collection of subpaths a path-path model of \( G \).

In [2] the sets of path-path models of interval graphs are characterized using a data structure called PQ-trees. In particular, a PQ-tree can be used to represent the set of path-path models of a given interval graph. Additionally, a PQ-tree can be constructed for a given graph \( G \) in \( \mathcal{O}(n + m) \) time. The PQ-tree has also seen numerous applications. Some of these include linear time: isomorphism testing for interval graphs [33], recognition for interval graphs [2], and consecutive ones property testing of a matrix [2].

The path graphs – intersection graphs of subpaths of a tree – are the primary focus of this work and are an intermediate class between interval graphs and chordal graphs. In particular, a graph \( G \) with \( V(G) = \{v_0, v_1, \ldots, v_{n-1}\} \), is a path graph iff:

There exists a tree \( T \) and a collection \( \{p_0, p_1, \ldots, p_{n-1}\} \) of subpaths of \( T \), such that:

\[
T = \bigcup_{i=0}^{n-1} p_i \text{ and } G = \mathcal{I}(\{p_0, p_1, \ldots, p_{n-1}\}).
\]

We call such a tree and collection of subpaths a path-tree model of \( G \).

The directed path graphs – intersection graphs of directed paths in directed trees – are the secondary focus in this work and are characterized by having a directed path-tree model (i.e., a collection of directed paths whose union is a directed tree).

These graph classes are related by the following proper inclusions: interval \( \subset \) directed path \( \subset \) path \( \subset \) chordal [35] (see fig. 1.1 below).

A lot of work has been done to characterize path graphs, directed path graphs, and other similar chordal graph classes. The clique tree theorems (presented in detail in section 2.1) by Gavril (path graphs [20], and chordal graphs [18]), Monma and Wei (on directed path graphs [35]) and Fulkerson and Gross (on interval graphs [15]) tell us that we can greatly restrict the possibilities for the corresponding model of each of these graph classes, i.e., the model of a graph \( G \) must be a spanning subgraph of \( G \)’s clique graph\(^\dagger\). The clique tree theorems were leveraged by Monma and Wei to develop the

\(^\dagger\)The vertex set of the clique graph of a graph \( G \) is \( K_G \) and the edge set is the pairs of maximal cliques whose intersection is not empty.
clique separator theorems (see [35]), which characterize each of these graph classes by considering the local structure surrounding clique separators. There are also forbidden induced subgraph characterizations (FISC) of chordal graphs (by definition), interval graphs [30], directed path graphs (by Panda [36]), and, path graphs (by Lèvèque et al. [31]).

The recognition problem for these graph classes has also been well studied and polynomial time recognition algorithms are known for all of these graph classes. The best known recognition algorithm for path graphs is by Schaffer [38] and runs in $O(nm)$ time (this algorithm follows the approach described by Monma and Wei [35] via their clique separator theorem). Also, a proposed linear time recognition algorithm for path graphs is outlined in an extended abstract (by Dahlhaus and Bailey [12]) but, according to Dahlhaus [11], this work will not appear in a journal submission. More specifically, the algorithm in [11] is presented at a high level with no proof of correctness, and there are
neither plans for it to be formalized nor proven correct. Also, the best known recognition algorithm for directed path graphs runs in $O(nm)$ time (by Chaplick et al. [4]). Additionally, interval graphs were first shown to be recognizable in linear time by Booth and Lueker [2] (using PQ-trees) and a simple linear time recognition algorithm has been developed by Corneil et al. [8]. Chordal graphs also have a simple linear time recognition algorithm due to Rose et al. [37].

In this thesis we introduce the weak PR-tree, PR-tree, and strong PR-tree (see chapter 3) data structures. The weak PR-tree is the most general form and is a useful tool to reach the main contribution of this work. In particular, we use the weak PR-tree to show that the PR-trees and strong PR-trees characterize the sets of path-tree models of path graphs and the sets of directed path-tree models of directed path graphs respectively.

The proof of this characterization is outlined as follows. In section 3.5, we present an algorithm that, when given a weak PR-tree that represents a graph $G$, produces either a PR-tree capturing the set of all path-tree models of $G$, or a strong PR-tree capturing the set of all directed path-tree models of $G$ (as needed). We also show that this algorithm runs in $O(n+m)$ time. We then present two distinct approaches that, when given a graph $G$, construct a weak PR-tree that represents $G$ (see chapters 4 and 5). The algorithm presented in chapter 4 uses Split Decomposition (also known as Join Decomposition) [10], runs in $O(nm)$ time, and constructs a weak PR-tree representing $G$ when possible. Our presentation of this Split Decomposition algorithm also leads to new characterizations of both path graphs and directed path graphs in terms of their Split Decompositions (this is also presented in chapter 4). The algorithm presented in chapter 5: follows the reduction approach used by Booth and Lueker to construct PQ-trees [2] (the templates associated with our algorithm are given in the appendix); runs in $O(A(n + m)nm)$ time where $A(s)$ is the inverse of Ackermann’s function arising from Union-Find‡; and constructs a weak PR-tree representing $G$ when $G$ is a path graph (i.e., showing that every path graph is

‡The specific form of the Union-Find data structure used is presented in appendix A.3.1.
representable by a weak PR-tree).

A useful side effect of our PR-tree construction algorithms is that they implicitly recognize path graphs and directed path graphs. In particular, a graph $G$ is a (directed) path graph iff it can be represented by a (strong) PR-tree. Our Split Decomposition based algorithm matches the fastest known recognition algorithms for path graphs [38] and directed path graphs [4] (each of which has a runtime complexity of $O(nm)$).

Also, in section 3.4, we prove that the weak PR-tree, PR-tree, or strong PR-trees which represents a graph $G$ can be represented in $O(n + m)$ space. Furthermore, we demonstrate an implicit form of these structures which can be represented in $O(n)$ space (this is presented in section 4.4.3 and relies on a key relationship between weak PR-trees and Split Decomposition).

1.1 Outline

This thesis is outlined as follows. We begin with some background results regarding these chordal graph classes (see chapter 2). In chapter 3 we define the weak PR-tree, PR-tree, and strong PR-tree and prove some key theorems which lead to the characterizations of path graphs by PR-trees and directed path graphs by strong PR-trees. We end the chapter by demonstrating how to convert a weak PR-tree into a PR-tree or strong PR-tree. In chapter 4 we present the Split Decomposition algorithm to construct a weak PR-tree from a given graph. Similarly, in chapter 5 we present the reduction algorithm to construct a weak PR-tree from a given graph (note: the technical details of this algorithm are presented in the appendix). The final chapter (6) provides some concluding remarks and consequences of this work.
Chapter 2

Background

This chapter contains the basis from which we will establish the weak PR-tree. In section 2.1 we discuss the Clique Tree Theorem for the path, directed path, interval, and chordal graph classes. In section 2.2 we then relate the Clique Tree Theorem to characterizing the respective sets of models of path graphs, directed path graphs, and interval graphs. This section also provides the basis for our data structures (i.e., the weak PR-tree, PR-tree, and strong PR-tree presented in chapter 3) and construction algorithms (i.e., via Reduction in chapter 5 and via Split Decomposition in chapter 4). In section 2.3 we recall the definition of PQ-trees [2] and describe them in terms of the notation introduced in section 2.2. The final section (2.4) of this chapter contains the forbidden induced subgraph characterization of path graphs and directed path graphs as presented in [36] and [31] respectively. These graphs will be of interest throughout this thesis and as such are presented here.

2.1 The Clique Tree Theorem

In the definition of the path graph class there are no restrictions regarding how we can choose the path-tree model which demonstrates a graph’s membership in this class. In particular, any tree and collection of subpaths may be selected as long as they satisfy
Chapter 2. Background

the intersection conditions. This is similarly true for the directed path, interval, and chordal graph classes. In this section we present a very useful restriction on the models of these graph classes in terms of their maximal cliques. This is known as the clique tree theorem. In particular, the clique tree theorem tells us that a graph $G$ belongs to the path / directed path / interval / chordal graph class iff a respective model can be constructed from a host graph $H$ whose vertex set is the set of maximal cliques of $G$ (i.e., $\mathcal{K}_G$) and the subset of $V(H)$ corresponding to a vertex $v \in V(G)$ is the set of maximal cliques incident with $v$ (i.e., $S_v$). This is formalized in the following theorem:

**Theorem 2.1. (Clique Tree)** A graph $G = (V = \{v_0, v_1, ..., v_n\}, E)$ is:

1. [18] A chordal graph iff there exists a tree $T$ with vertex set $\mathcal{K}_G$, such that for every $i \in [0, n - 1]$, $T$ induced on $S_{v_i}$ is a tree.

2. [20] A path graph iff there exists a tree $T$ with vertex set $\mathcal{K}_G$, such that for every $i \in [0, n - 1]$, $T$ induced on $S_{v_i}$ is a path.

3. [35] A directed path graph iff there exists a directed tree $T$ with vertex set $\mathcal{K}_G$, such that for every $i \in [0, n - 1]$, $T$ induced on $S_{v_i}$ is a directed path.

4. [15] An interval graph iff there exists a path $P$ with vertex set $\mathcal{K}_G$, such that for every $i \in [0, n - 1]$, $P$ induced on $S_{v_i}$ is a path.

**Note:** We use $\mathcal{C}_G$, $\mathcal{T}_G$, $\overset{\rightarrow}{\mathcal{T}}_G$, and $\mathcal{P}_G$ to denote the set of all such tree-tree, path-tree, directed path-tree, and path-path models (as in (1), (2), (3), and (4) respectively). E.g.,

$$\mathcal{T}_G = \{T : T \text{ is a tree with } V(T) = \mathcal{K}_G, \text{ and } \forall i \in [0, n - 1], T[S_{v_i}] \text{ is a path}\}.$$

For a graph $G$ (as discussed in section 2.1), the clique versions of the tree-tree / path-tree / directed path-tree / path-path models intersection models of a graph $G$ (as in theorem 2.1 above) form a representative set of all tree-tree / path-tree / directed path-tree / path-path models models. In particular it is well known that any tree-tree / path-tree / directed path-tree / path-path model of a graph $G$ can be “shrunk” to a
clique version through the application of edge contractions and vertex deletions to the given model. Thus, by capturing all clique versions of these types models we are in fact capturing all of these types of models.

From here on we consider all tree-tree, path-tree, directed path-tree, and path-path models to be the clique versions as outlined in the Clique Tree Theorem (i.e., a \textit{path-tree model} of a graph $G$ is a tree, $T$, with vertex set $K_G$ such that, for every vertex $v$ of $G$, $S_v$ induces a path in $T$). The tree-tree model is more commonly referred to as a \textit{clique tree}. We have chosen the name tree-tree model so that it is easier to differentiate between a clique tree and the more restrictive versions of clique trees used by restricted subclasses of chordal graphs (i.e., the path, directed path, and interval graph classes). A discussion of clique trees can be found in many graph theory texts (e.g., [34, 23]).

2.2 The Characterization Problems

Using the clique tree theorem we can now formalize what we mean by having a data structure which can characterize the set of path-tree / directed path-tree / path-path models associated with any given graph. We first define the relevant problems (see subsection 2.2.1) then we discuss some properties of them (see subsection 2.2.2). Recall that, for a graph $G$ (as discussed in section 2.1), the clique versions of the path-tree / directed path-tree / path-path models intersection models of a graph $G$ form a representative set of all models (i.e., all models discussed in this section and throughout the remainder of this thesis are the clique versions).

2.2.1 Path-Tree, Directed Path-Tree, and Path-Path Problems

We use the path-tree models as our canonical case for establishing this concept. In particular, a data structure must be devised such that, for an arbitrary graph $G$, there is an instance of this data structure which captures the path-tree models of $G$ in a “concise”
manner (note: we explicitly represent all clique versions of these models as in theorem 2.1). Furthermore, an algorithm must be devised which, for a given $G$, produces such an instance of the data structure. The PR-tree and its corresponding construction algorithm accomplish this goal. Additionally, the strong PR-tree and its construction algorithm achieve this goal for directed path-tree models. Booth and Lueker [2] have already solved this for path-path models using the PQ-tree and its corresponding construction algorithm. In observation 3.4 we will relate a degenerate form of PR-trees to PQ-trees (note: this degenerate form is semantically the same as a PQ-tree).

We begin to formalize the specification of each construction algorithm by defining the path-tree, directed path-tree, and path-path problems.

**Definition 2.2.** For a graph $G$ the path-tree problem is to produce the set $T_G$ of path-tree models of $G$ (as in theorem 2.1), such that $T_G$ is captured in a “concise” manner. The directed path-tree and path-path problems are defined similarly using $\overrightarrow{T}_G$ and $P_G$ in place of $T_G$ (respectively).

*Note:* $G$ is a path/directed path/interval graph iff $T_G \neq \emptyset$, $\overrightarrow{T}_G \neq \emptyset$, $P_G \neq \emptyset$ respectively.

Notice that, the set $T_G$ is the subset of all trees over the vertex set $K_G$ in which $S_v$ is a path for all $v \in V(G)$. Additionally, $S_v$ similarly restricts the sets $C_G$, $\overrightarrow{T}_G$, and $P_G$. In this framing, each set of incident maximal cliques $S_v$ can be thought of as a constraint. In particular, we define the constraint set of a graph $G$ to be the set $S_G = \{S_v : v \in V(G)\}$. While it is possible that a graph $G$ has two vertices with the same constraint, we do not need to maintain this multiplicity since the restriction imposed is redundant; i.e., $S_G$ is truly a set rather than a multiset. Notice that we can have a constraint set $S$ independently of the existence of a graph $G$ with $S = S_G$. In particular, for a set $S$ of sets of elements, we can consider the tree-tree, path-tree, directed path-tree, and path-path models that satisfy $S$. For example the set of path-tree models that satisfy $S$, denoted $T_S$, is the set of trees $T$ where $V(T) = \bigcup_{S \in S} S$ and $T[S]$ is a path for every $S \in S$. We similarly define the tree-tree, directed path-tree, and path-path models that satisfy $S$ and
denote them by $C, \overrightarrow{T}, P$ respectively. Clearly, for a graph $G$: $C_G = C; T_G = T; \overrightarrow{T}_G = \overrightarrow{T};$ and $P_G = P$. Note: for convenience we let $U_S = \bigcup_{S \in S} S$; i.e., for a graph $G$, $U_G = K_G$. We take a closer look at constraint sets and the models that satisfy them in the next section (see section 2.2.2).

To explain what we mean by "concise", we consider a general approach (from [1]) to the problem of constructing $T_S$ (see algorithm 1 below). This is called a Reduction because it iteratively reduces the set of all trees to those which satisfy the constraints.

**Algorithm 1: Reduction(S):** Locating $T_S$ of a constraint set $S$ by Reduction [1].

pre : $S$ is a constraint set.
post: $T$ is the set of path-tree models of $S$ (i.e., $T = T_S$).

1. $T = \{T : T$ is a tree and $V(T) = U_S\}$ // all possible trees over $U_S$.
2. for each $S \in S$ do $T = T \setminus \{T : T \in T, \text{ and } T[S] \text{ is not a path}\}$
3. return $T$

If we implemented this algorithm by representing $T$ as a list, it would be easy, but also quite inefficient. Algorithm 1’s success relies entirely on having a data structure that represents the set $T$ in a concise manner and allows the introduction of new constraints (as in line 2 of Algorithm 1), to be performed efficiently. Furthermore, representing the set $T$ as a list is not a good idea since this could be exponentially large with respect to $S$ (or a graph $G$ with $S_G = S$). The data structure that we use for this is the weak PR-tree.

### 2.2.2 Constraint Sets and Models

In this section we consider some properties of the path-tree models $T_S$ arising from a constraint set $S$. The interaction amongst constraints in $S$ can imply that subsets of $U_S$ distinct from those in $S$ are guaranteed to be paths in every $T \in T_S$ (i.e., indicating “implied” constraints). In particular, we consider some ways in which such “implied constraints” can arise. In this context, we define a *path-tree constraint* as follows.
**Definition 2.3.** For a constraint set \( \mathbb{S} \), we say that \( S \) is a path-tree constraint (PT-constraint) of \( \mathbb{S} \) when \( S \subseteq \mathcal{U}_S \) and \( S \) induces a path in every path-tree model \( \mathbb{S} \) (i.e., \( T[S] \) is a path in every \( T \in \mathcal{T}_S \)). We use \( \mathbb{PT}_S \) to denote the PT-constraints of a constraint set \( \mathbb{S} \) and, for a graph \( G \), \( \mathbb{PT}_G = \mathbb{PT}_{\mathcal{S}_G} \).

*Note:* We refer to a PT-constraint \( S \) of \( \mathbb{S} \) as an implied PT-constraint of \( \mathbb{S} \) when it is not a member of \( \mathbb{S} \) (i.e., \( S \in \mathbb{PT}_S \setminus \mathbb{S} \)). Also, for a graph \( G \), when we refer to a constraint of \( G \) (i.e., omitting “PT”), we specifically mean an element of \( \mathcal{S}_G \).

Notice that the PT-constraints of a constraint set \( \mathbb{S} \) are a superset of it (i.e., \( \mathbb{PT}_S \supseteq \mathbb{S} \)). In particular, there can be many different constraint sets with the same set of PT-constraints. Additionally, by def. 2.3, it is easy to see that any pair of constraint sets \( \mathbb{S}, \mathbb{S}' \) where \( \mathbb{PT}_S = \mathbb{PT}_{S'} \), \( \mathcal{T}_S = \mathcal{T}_{S'} \). Furthermore, if \( \mathcal{T}_S = \emptyset \), then every subset of \( \mathcal{U}_S \) is a PT-constraint. Thus, PT-constraints are only interesting when considered in the context of \( \mathcal{T}_S \neq \emptyset \).

We now consider how to generate some of the implied PT-constraints of constraint set \( \mathbb{S} \). This concept arises from the following two straightforward observations on trees.

**Observation 2.4.** For a tree \( T \) with \( V_1, V_2 \subseteq V(T) \) such that \( T[V_1] \) is a tree and \( T[V_2] \) is a path: \( T[V_1 \cap V_2] \) is a path.

**Observation 2.5.** For a tree \( T \) with \( V_1, V_2 \subseteq V(T) \) such that \( T[V_1] \) is a tree and \( T[V_2] \) is a path: if \( V_1 \cap V_2 \neq \emptyset \), then \( T[V_1 \cup V_2] \) is a tree.

Notice that we can generalize these observations to PT-constraints (i.e., paths) of \( \mathcal{T}_S \) (i.e., a tree). In particular, for two PT-constraints \( S \) and \( S' \), the intersection \( S'' = S \cap S' \) is also a PT-constraint (i.e., \( S, S' \in \mathbb{PT}_S \implies S \cap S' \in \mathbb{PT}_S \)). This is clear since, for every tree \( T \in \mathcal{T}_S \), both \( T[S] \) and \( T[S'] \) are paths and thus (by observation 2.4) \( T[S''] \) is also a path. Similarly, as in observation 2.5, when two PT-constraints have a non-empty intersection, their union will be a tree in every \( T \in \mathcal{T}_S \). Consider the notion of
connectedness in def. 2.6 below. Using this definition we state a general form of the observations described in this paragraph (see observations 2.7 and 2.8).

**Definition 2.6.** A collection of sets $S = \{S_0, S_1, ..., S_{k-1}\}$ is connected when for every $j, j^* \in [0, k-1]$ there exists a sequence $j = j_0, ..., j_{\ell - 1} = j^*$ (with $j_i \in [0, k-1]$ for every $i \in [0, \ell - 1]$) such that $S_{j_i} \cap S_{j_i+1} \neq \emptyset$ for every $i \in [0, \ell - 2]$ (i.e., the intersection graph $\mathcal{I}(S)$ is connected).

**Observation 2.7.** For a constraint set $S$, if $S^* \subseteq \text{PT}_S$ is connected, then:

(a) For every $T \in \mathcal{T}_S$, $T[U_{S^*}]$ is a tree.

(b) For every PT-constraint $S$, $S \cap U_{S^*}$ is a PT-constraint.

**Observation 2.8.** For a constraint set $S$ and a constraint $S'$, if $S' \subseteq S$ and $S' \cup \{S\}$ are both connected, then $S' = S \cap U_{S'}$ is a PT-constraint of $S \cup \{S\}$.

We now consider relationships among PT-constraints of a constraint set $S$ where $\mathcal{T}_S \neq \emptyset$; i.e., a graph $G$ with $S_G = S$ would be a path graph $G$. In particular, we observe that the PT-constraints satisfy a chordality (see observation 2.9) and a helly property (see observation 2.10) when $\mathcal{T}_S \neq \emptyset$.

Notice that, if a set of PT-constraints were to indicate the formation of a cycle, this would immediately imply that $\mathcal{T}_S = \emptyset$. For example, if $S_0 = \{1, 2\}$, $S_1 = \{2, 3\}$, $S_2 = \{3, 4\}$, and $S_2 = \{4, 1\}$ were PT-constraints, then the edges $(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 1)$ would need to be present in every path-tree model satisfying these constraints (i.e., there would be no path-tree models satisfying these constraints). This generalizes to a “chordality” condition among PT-constraints (formalized in observation 2.9 below).

Consider an ordered collection of PT-constraints $X = \{S_0, S_1, ..., S_{k-1}\}$. We say that $X$ is a loop when $S_i \cap S_{i+1(\text{mod } k)} \neq \emptyset$ for $i \in [0, k-1]$. Furthermore, $X$ is an induced loop when $S_i \cap S_j = \emptyset$ for $i \in [0, k-1]$ and $j \in [0, k-1] \setminus \{i - 1(\text{mod } k), i, i + 1(\text{mod } k)\}$. This leads to the following observation.
Observation 2.9. For a constraint set $S$, every induced loop $\{S_0, ..., S_{k-1}\}$ of PT-constraints of $S$ has $k \leq 3$.

Additionally, we notice that paths in a tree satisfy the helly property [23]. In particular, every collection of pairwise intersecting paths in a tree $T$ must have a common vertex (this is formalized in observation 2.10 below).

Observation 2.10. For a constraint set $S$, if $\{S_0, S_1, ..., S_{k-1}\} \subseteq PT_S$ and $S_i \cap S_j \neq \emptyset$ for every $i, j \in [0, k - 1]$, then $S = \bigcap_{i=0}^{k-1} S_i \neq \emptyset$ and $S$ is a PT-constraint of $S$.

Proof. Let $T$ be a path-tree model satisfying a constraint set $S$ (i.e., $V(T) = U_S$ and for every $S \in S$, $T[S]$ is a path). Now, let $\{S_0, S_1, ..., S_{k-1}\} \subseteq PT_S$ such that $S_i \cap S_j \neq \emptyset$ for every $i, j \in [0, k - 1]$. Clearly, every $S_i$ ($i \in [0, k - 1]$) induces a path in $T$. Thus, $S = \bigcup_{i=0}^{k-1} S_i \neq \emptyset$ and, by observation 2.4, $T[S]$ is a path. Thus, since $T$ is an arbitrary path-tree model satisfying $S$, $S$ is a PT-constraint of $S$. \qed

The properties of PT-constraints presented in this section will be useful throughout our discussions of the various PR-trees and their construction algorithms.

2.3 PQ-trees

This section consists of two parts. First, we recall the definitions and results from [2] (see section 2.3.1). Then, in section 2.3.2, we discuss the constraints of PQ-trees.

2.3.1 PQ-trees: Definition and Results [2]

The class of PQ-trees over $U = \{u_0, u_1, ..., u_{z-1}\}$ (formally defined below) is all rooted, ordered trees$^\dagger$ whose leaves are elements of $U$ and whose internal (non-leaf) nodes are distinguished as being either P-nodes or Q-nodes. The PQ-tree is constructively defined as follows.

$^\dagger$A tree where the children of every node are ordered, i.e., there is a first child, second child, etc.
Definition 2.11. A rooted ordered tree $T$ is a PQ-tree iff it can be constructed through the following three operations.

1. **Leaves:** Every element $u_i \in U$ is a PQ-tree whose root is the element. The tree consists of a single leaf and is drawn as the element itself.

2. **P-nodes:** If $T_0, T_1, \ldots, T_{k-1}$ ($k \geq 3$) are disjoint PQ-trees then a P-node with the root of each $T_0, T_1, \ldots, T_{k-1}$ as children (in this order) is a PQ-tree. A P-node is drawn as a circle with its children drawn below it (see fig. 2.1).

3. **Q-nodes:** If $T_0, T_1, \ldots, T_{k-1}$ ($k \geq 2$) are disjoint PQ-trees then a Q-node with the root of each $T_0, T_1, \ldots, T_{k-1}$ as children (in this order) is a PQ-tree. A Q-node is drawn as a rectangle with its children drawn below it (see fig. 2.2).

![Figure 2.1: A P-node.](image1)

![Figure 2.2: A Q-node.](image2)

We now describe how P and Q nodes enforce constraints and what it means for two PQ-trees $T$ and $T^*$ to be equivalent, denoted by $T \equiv T^*$. Two PQ-trees are equivalent iff one can be transformed into the other by applying zero or more equivalence transformations, where each transformation specifies a legal reordering of the nodes within a tree. The two types of equivalence transformations are: permuting the children of a P-node, and reversing the children of a Q-node (two equivalent PQ-trees are given in fig. 2.3). Each equivalent PQ-tree represents a different ordering of the leaves because each PQ-tree is ordered. The ordering of the leaves established by a PQ-tree $T$ is called the frontier of $T$ and is denoted $\text{Frontier}(T)$ (the two PQ-trees in fig. 2.3 have frontiers $ABCDEFGHIJ$ (left) and $BIJHGEDFAC$ (right)). The set of all orderings arising from PQ-trees equivalent to $T$ is called the consistent set of $T$ and is denoted $\text{Consistent}(T) = \{\text{Frontier}(T^*) : T^* \equiv T\}$. In particular, $\text{Consistent}(T)$ is a set of
paths whose vertex set is $L(T)$ and satisfy certain constraints. At the end of this section we describe the set of constraints $S_T$ of a PQ-tree $T$ (i.e., $P_{S_T} = \text{Consistent}(T)$).

Figure 2.3: Two equivalent PQ-trees.

As a matter of convention Booth and Lueker insist that each Q-node has at least three children and each P-node has at least two. This is different from, but equivalent to, what we have presented in def. 2.11 above. Notice that, semantically, a P-node with two children and a Q-node on the same two children are the same (i.e., they allow identical equivalence transformations). In particular, an equivalent convention is that P-nodes have at least three children and Q-nodes have at least two children. Furthermore, we use the convention in def. 2.11 as it is more convenient when comparing PQ-trees and PR-trees.

In [2] Booth and Lueker demonstrate that the PQ-tree can be used to solve the path-path problem (as in the following theorem). The algorithm mentioned in this theorem uses the reduction approach (i.e., as in algorithm 1 in section 2.2.1).

Theorem 2.12. (PQ-tree Theorem) For a graph $G$, a PQ-tree $T$ with $\text{Consistent}(T) = P_G$ can be produced in $O(n + m)$. Additionally, $T \neq \emptyset$ iff $P_G \neq \emptyset$ (i.e., $G$ is an interval graph iff $T \neq \emptyset$).

2.3.2 Constraints and PQ-trees

We now discuss the constraints enforced by a given PQ-tree $T$. We use $S_T$ to denote these constraints. Also, we use $L(T)$ to denote the leaf set of a PQ-tree $T$. We consider
a set $S \subseteq L(T)$ to be a *good leaf set* of a PQ-tree $T$ when $S$ is consecutive in every path $P$ in $\text{Consistent}(T)$. In particular, $S_T$ is the collection of the good leaf sets of $T$. The purpose of this discussion is to aid in the understanding of, and simplify the corresponding discussion regarding, PR-trees (see section 3.2). Note: the concepts of good leaf sets and the constraint set of a PQ-tree were not presented in [2], though they are implicit from the results in [2].

To understand the good leaf sets, we consider the *least common ancestor* (LCA) of a good leaf set $S$ (note: it will be useful to consider the two equivalent PQ-trees given in fig. 2.3 during this discussion). In particular, this LCA is either a P-node or a Q-node. Notice that, when the LCA is a P-node the leaf set of this P-node must be precisely the set $S$ (otherwise, we can easily apply equivalence transformations to contradict the fact that $S$ is a good leaf set). Similarly, when the LCA is a Q-node $Q$ there must be a consecutive subsequence $T_i, ..., T_i+j$ of $Q$’s children such that $S = \bigcup_{\ell=i}^{i+j} L(T_{\ell})$. Thus, we can present the set $S_T$ as follows (note: when $T$ is a leaf, $S_T = \{L(T)\}$). When the root of $T$ is a P-node with children $T_0, T_1, ..., T_{k-1}$:

$$S_T = \left( \bigcup_{i=0}^{k-1} S_{T_i} \right) \cup \left\{ \bigcup_{\ell=0}^{k-1} L(T_{\ell}) \right\}.$$  (2.1)

Similarly, when the root of $T$ is a Q-node with children $T_0, T_1, ..., T_{k-1}$ (in this order):

$$S_T = \left( \bigcup_{i=0}^{k-1} S_{T_i} \right) \cup \left\{ \bigcup_{\ell=1}^{j} L(T_{\ell}) : i \in [0, k - 2], j \in [i + 1, k - 1] \right\}.$$  (2.2)

Notice that $S_T$ always includes $L(T) = \bigcup_{\ell=0}^{k-1} L(T_{\ell})$. Thus, every path-tree model satisfying $S_T$ (i.e., $T_{S_T}$) is actually a path (i.e., $T_{S_T} = P_{S_T}$). Furthermore, since $S_T$ is the collection of good leaf sets of $T$, it is not only a constraint set, but also a PT-constraint set (i.e., $\mathbb{P}T_{S_T} = S_T$).

We now show that every path $P$ satisfying $S_T$ is indeed in the consistent set of $T$. 
Lemma 2.13. For a PQ-tree $T$, a permutation $\pi$ of $L(T)$ belongs to $\text{Consistent}(T)$ iff it satisfies the consecutiveness constraints $S_T$.

Proof.

$\implies$ This is clear from our discussion of good leaf sets (above).

$\impliedby$ This is trivially true for the case when $T$ is a leaf. Thus, we can inductively assume this is true for all PQ-trees whose height is less than $T$ and prove it for $T$ itself. In particular, if the root of $T$ has $k$ children, then a permutation $\pi$ satisfying $S_T$ can be partitioned into $k$ disjoint permutations $\pi_0, \pi_1, ..., \pi_{k-1}$ such that $\pi_i$ satisfies $S_{T_i}$ for each $i \in [0, k-1]$. Furthermore, using our assumption, we apply equivalence transformations to each $T_i$ so that $\text{Frontier}(T_i) = \pi_i$. We now consider the two cases for the root of $T$:

(1): The root is a P-node $P$. In this case we simply permute the children of $P$ to match the order of $\pi_0, \pi_1, ..., \pi_{k-1}$ in $\pi$. Clearly, the frontier after this transformation is $\pi$.

(2): The root is a Q-node $Q$. Suppose w.l.o.g. that $Q$’s children are ordered $T_0, T_1, ..., T_{k-1}$. Now, due to the constraints enforced by $Q$ (i.e., $\bigcup_{\ell=i}^j L(T_\ell)$ for every $i \in [0, k-2]$ and $j \in [i+1, k-1]$), we can see that $\pi$ is either $(\pi_0, \pi_1, ..., \pi_{k-1})$ or $(\pi_{k-1}, \pi_{k-2}, ..., \pi_0)$. Thus, by reversing $Q$ as needed, the frontier becomes $\pi$. \qed

It is interesting to note that we can generate a constraint set $S_T^*$ which is smaller than $S_T$ and still has $PT_{S_T^*} = S_T$. In fact, we can select $S_T^*$ such that the number of constraints in $S_T^*$ is less than twice the number of nodes in $T$. In particular, when $T$ is a leaf $S_T^* = \{L(T)\}$, and when the root of $T$ is a P-node we use equation 2.1 (as before). However, when the root of $T$ is a Q-node consider using the following equation:

$$S_T^* = \left( \bigcup_{i=0}^{k-1} S_{T_i}^* \right) \cup \left( \bigcup_{\ell=0}^{k-1} L(T_\ell) \right) \cup \left\{ L(T_i) \cup L(T_{i+1}) : i \in [0, k-2] \right\}. \quad (2.3)$$

Now, from observation 2.8, every consecutive subsequence $T_i, ..., T_j$ ($0 \leq i < j \leq k-1$) of $T_0, T_1, ..., T_{k-1}$ is a PT-constraint of these constraints. In particular, we have a connected
set of PT-constraints \( S = \{ L(T_\ell) \cup L(T_{\ell+1}) : \ell \in [i, j-1] \} \) and we have \( \bigcup_{i=0}^{k-1} L(T_i) \) as a PT-constraint. Thus, \( \bigcup_{i=0}^{k-1} L(T_i) \cap U_S = \bigcup_{\ell=i}^{j} L(T_\ell) \) is also a PT-constraint.

To see that \( |S_T^*| \) is less than twice number of nodes in \( T \) we use the following charging argument. For each P-node \( P \), charge its constraint \( L(P) \) against \( P \)'s first child. Similarly, for each Q-node \( Q \) whose children are \( N_0, N_1, ..., N_{k-1} \) (in this order), charge the constraint \( L(Q) \) against \( N_0 \) and for each constraint \( S_{i+1} = L(N_i) \cup L(N_{i+1}) \) \( (i \in [0, k-2]) \) charge \( S_{i+1} \) against \( N_{i+1} \). Thus, the number of constraints introduced by the non-leaf nodes is at most the number of edges in \( T \) (i.e., less than the number of nodes in \( T \)). Furthermore, each leaf of \( T \) introduces exactly one constraint. Thus, the total number of constraints is less than twice the number of nodes in \( T \).

In chapter 3 we present analogous concepts to the equivalent transformations, frontier, consistent set, and constraint set of a PQ-tree for PR-trees. Additionally, in theorem 3.15 (in section 3.2.2), we will prove that a tree satisfies the “constraint set” of a PR-tree \( D \) if and only if this tree is in \( D \)'s “consistent set” (i.e., that a PR-tree represents the set of trees \( T_\mathcal{S} \) for a particular constraint set \( S \)).

### 2.4 Forbidden Induced Subgraph Characterization (FISC) of Path Graphs

In this section we provide the FISC characterizations of path graphs (see theorem 2.14 and fig. 2.4) [31] and directed path graphs (see theorem 2.15 and fig. 2.5) [36].

The FISC of path graphs was obtained by Lévêque, Maffray and Preissmann [31] (see Figure 2.4); an independent proof has been obtained by Tondato [41]:

**Theorem 2.14 ([31]).** A graph is a path graph if and only if it does not contain any of \( F_0(n)_{n \geq 4}, F_1, F_2, F_3, F_4, F_5(n)_{n \geq 7}, F_6, F_7, F_8, F_9, F_{10}(n)_{n \geq 8}, F_{11}(4k)_{k \geq 2}, F_{12}(4k)_{k \geq 2}, F_{13}(4k+1)_{k \geq 2}, F_{14}(4k+1)_{k \geq 2}, F_{15}(4k+2)_{k \geq 2}, F_{16}(4k+3)_{k \geq 2} \) as an induced subgraph.
Similarly, the FISC of directed path graphs was obtained by Panda [36] (see Figure 2.4); an independent proof has been obtained by Chaplick et. al [4] (this independent proof relies on the FISC of path graphs):

**Theorem 2.15** ([36]). A graph is a directed path graph if and only if it contains no $F_0(n)_{n \geq 4}$, $F_1$, $F_3$, $F_4$, $F_5(n)_{n \geq 7}$, $F_6$, $F_7$, $F_9$, $F_{10}(n)_{n \geq 8}$, $F_{13}(4k + 1)_{k \geq 2}$, $F_{15}(4k + 2)_{k \geq 2}$, $F_{16}(4k + 3)_{k \geq 2}$, $F_{17}(4k + 2)_{k \geq 1}$ as an induced subgraph.

Of these minimal forbidden graphs, two infinite families are of particular interest in this work. Namely, $F_{17}(4k + 2)_{k \geq 1}$ which are more commonly referred to as the **odd suns** (since they are an odd sized centre clique with an odd number of “points”) and $F_{11}(4k)_{k \geq 2}$ which we will refer to as **false twin odd suns** (since they are the odd suns after adding a pair of non-adjacent vertices universal to the centre clique which are false twins).

Notice that, from the list of minimal forbidden induced subgraphs of path graphs of theorem 2.14, if we remove every graph that contains an odd sun, namely $F_2$, $F_8$, $F_{11}(4k)_{k \geq 2}$, $F_{12}(4k)_{k \geq 2}$, $F_{14}(4k + 1)_{k \geq 2}$, and add the odd suns to the list, we obtain the
Figure 2.5: Minimal forbidden induced subgraphs for directed path graphs (the vertices in the cycle marked by bold edges form a clique).

list of minimal forbidden induced subgraphs of directed path graphs (see fig. 2.5).
Chapter 3

Weak PR-trees, PR-trees, and Strong PR-trees

In this chapter we present three distinct forms of PR-trees: weak PR-trees, PR-trees, and strong PR-trees. For a constraint set $S$, the weak PR-tree represents the PT-constraints of $S$ in a concise way such that it is easy to determine the trees and directed trees which satisfy $S$. In particular, the (strong) PR-trees of a weak PR-tree $D$ are certain special “ordered” instances of $D$ which correspond to the (directed) trees that satisfy $S$. Recall that we use $\mathcal{U}_S$ to denote the union of the constraints in $S$ (i.e., $\mathcal{U}_S = \bigcup_{S \in S} S$). Notice that, for a graph $G$, $\mathcal{U}_G = \mathcal{K}_G$.

The class of weak PR-trees / PR-trees / strong PR-trees over $\mathcal{U} = \{u_0, u_1, ..., u_{z-1}\}$ is defined to be all multi-sourced directed acyclic graphs where: each source node\(^1\) is a P-node, an R'-node, or a weak R-node / an R-node / a strong R-node; each internal (non-leaf, non-source) node is a P-node or an R'-node; and the leaves\(^2\) are the elements of $\mathcal{U}$. Since a weak PR-tree is a multi-sourced directed acyclic graph, each non-source node could have more than one parent node and we use $\text{Parents}(N)$ to denote the parents of

\(^1\)An element of $\mathcal{U}$ can be a source node, but only when $|\mathcal{U}| = 1$.

\(^2\)When we refer to the leaves of a weak PR-tree / PR-tree / strong PR-tree, we mean the nodes with out-degree zero (i.e., the sinks).
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

N. Similarly, we use Children(N) to denote the children of N and L(N) to denote the leaves of N (note: if N is a leaf, L(N) = {N}).

This chapter is outlined as follows. We first present the weak PR-tree and its corresponding constraint set (see section 3.1). In section 3.2 we establish the PR-tree and strong PR-tree by describing the trees and directed trees satisfying the constraints of a weak PR-tree (i.e., the consistent set and directed consistent set of a weak PR-tree). In section 3.3 we connect the PT-constraints of the constraint set of a weak PR-tree with the nodes of a weak PR-tree. We also formalize what it means for a weak PR-tree to represent a graph and observe that weak PR-trees can only represent chordal graphs in this section. In section 3.4 we demonstrate that the size of a weak PR-tree that represents a graph $G$ is $O(n + m)$. In the final section (3.5) we present two algorithms: one to convert a weak PR-tree into a PR-tree, and one to convert a weak PR-tree into a strong PR-tree. These algorithms both run in linear time with respect to the size of the given weak PR-tree.

3.1 The Weak PR-tree

This section consists of three subsections. In the first two subsections we define the weak PR-tree and its constraint set. In the final subsection we consider some weak PR-trees and graphs with the same set of PT-constraints.

3.1.1 Defining Weak PR-trees

The class of weak PR-trees over $\mathcal{U} = \{u_0, u_1, ..., u_{z-1}\}$ is defined to be the multi-sourced directed acyclic graphs where: the leaves are the elements of $\mathcal{U}$, each internal (non-leaf, non-source) node is a P-node or an $R'$-node, and each source node is a P-node, a weak R-node, or an $R'$-node $^\S$.

$^\S$An element of $\mathcal{U}$ can be a source node, but only when $|\mathcal{U}| = 1$. 
In section 3.2 we will refine the concept of a weak PR-tree into PR-trees and strong PR-trees. The differences amongst weak PR-trees, PR-trees, and strong PR-trees are captured by the differences amongst weak R-nodes (as in part 3 of def. 3.1 below), R-nodes (see def. 3.9), and strong R-nodes (see def. 3.11). In particular, the R-node and strong R-node are refinements of the weak R-node which describe additional conditions required to ensure the path-tree and directed path-tree models are properly represented.

Also, even though the weak PR-tree is a multi-sourced directed acyclic graph, for each node $N$, we insist that the subgraph induced by $N$ together with $N$’s descendants is a rooted tree (with $N$ as its root). We present the weak PR-tree through the following constructive definition. A drawing of a weak PR-tree is given in fig. 3.3.

**Definition 3.1.** $D$ is a weak PR-tree iff it can be constructed through the following three operations. Note: we use $\text{Sources}(D)$ to specify the source nodes of $D$ ; i.e., we say $\text{Sources}(D) = \{A_0, A_1, ..., A_{\alpha-1}, R_0, R_1, ..., R_{\gamma-1}\}$ where $A_0, A_1, ..., A_{\alpha-1}$ are the P-node and $R'$-node sources of $D$ and $R_0, R_1, ..., R_{\gamma-1}$ are the weak R-nodes of $D$, each of which must be a source node.

1. **Leaves:** Every element $u_i \in U$ is a weak PR-tree with $u_i$ as its only source. The weak PR-tree consists of only a single leaf and is drawn as the element itself.

2. **P-nodes:** If $D_0, D_1, ..., D_{k-1}$ ($k \geq 3$) are disjoint weak PR-trees and for every $i \in [0, k-1]$, $N_i$ is either a P-node or $R'$-node or leaf in $D_i$ (note: $N_i$ need not be a source of $D_i$), then we can construct a new weak PR-tree $D$ whose source nodes are: $\{P\} \cup (\bigcup_{i=0}^{k-1} \text{Sources}(D_i) \setminus \{N_0, N_1, ..., N_{k-1}\})$ where $P$ is a P-node and $N_0, N_1, ..., N_{k-1}$ are the children of $P$. A P-node is drawn as a circle with its children drawn below it. This is depicted in fig. 3.1.

3. **Weak R-nodes and $R'$-nodes:** If $D_0, D_1, ..., D_{k-1}$ ($k \geq 2$) are disjoint weak PR-trees, and for every $i \in [0, k-1]$, $N_i$ is either a P-node or $R'$-node or leaf in $D_i$ (note: $N_i$ need not be a source of $D_i$) and no parent of $N_i$ is a weak
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

Figure 3.1: Two drawings of a P-node $P$ whose children are $N_0$, $N_1$, ..., $N_{k-1}$ where $\pi$ is a permutation on $[0, k-1]$.

**R-node**, then we can construct a new weak PR-tree $D$ whose source nodes are:

$$\{R, Q_0, Q_1, ..., Q_{\ell-1}\} \cup (\bigcup_{i=0}^{k-1} \text{Sources}(D_i) \setminus \{N_0, N_1, ..., N_{k-1}\})$$

where:

- $R$ is a weak R-node and every $Q_j$ is an $R'$-node.
- $N_0, N_1, ..., N_{k-1}$ are $R$’s children and $R$ is associated with a tree called $R_T$ where $V(R_T) = \{N_0, N_1, ..., N_{k-1}\}$ (this can be any tree).
- For each $j \in [0, \ell - 1]$, the children of $Q_j$ are a subset of $\text{Children}(R)$ which forms a non-trivial path in $R_T$; i.e., $\text{Children}(Q_j) \subseteq \text{Children}(R)$, $|\text{Children}(Q_j)| \geq 2$, and $R_T[\text{Children}(Q_j)]$ is a path. Also, we say that $Q_j$ specifies the path $R_T[\text{Children}(Q_j)]$.
- For each edge $N_i, N_i^*$ of $R_T$, there is at least one $R'$-node $Q_j$ such that both $N_i$ and $N_i^*$ are children of $Q_j$ (i.e., $Q_j$ specifies the edge $N_i, N_i^*$).

We use $R_{R'}$ to denote the set of $R'$-nodes associated with $R$ (i.e., $\{Q_0, Q_1, ..., Q_{\ell-1}\}$) and $R_T$ to denote the tree. Also, we use $Q_R$ to denote the unique weak R-node to which an $R'$-node $Q$ is associated. Furthermore, we draw each $R'$-node as a rectangle with its children drawn below it in an order that is consistent with its specified path.

A weak R-node $R$ is depicted in fig. 3.2 below.

Figure 3.2: A weak R-node with $R'$-nodes $R_{R'} = \{Q_0, Q_1, Q_2\}$ (left) and tree $R_T$ (right). Note: each $R'$-node specifies the path as shown (e.g., $Q_2$ specifies the path $N_4, N_1, N_0, N_3$).
These operations always produce leaves which are elements of $\mathcal{U}$, internal nodes which are either P-nodes or R’-nodes, and source nodes which are P-nodes, R’-nodes or weak R-nodes. Notice that we insist that R’-nodes have at least two children and P-nodes have at least three children. This is to ensure that the weak PR-tree is a concise data structure. Furthermore, we do not allow P-nodes to have two children since every P-node with exactly two children can be represented as an R’-node and it is useful to insist on this convention (we justify this later in this section).

We now consider the relationship between weak R-nodes and R’-nodes. In particular, there is a one-to-many relationship from weak R-nodes to R’-nodes. More specifically, for every weak R-node $\mathcal{R}$ there is a non-empty set of R’-nodes $\mathcal{R}_{R’}$ where $\text{Children}(\mathcal{R}) = \bigcup_{Q \in \mathcal{R}_{R’}} \text{Children}(Q) \ (= V(\mathcal{R}_T))$. Similarly, for every R’-node $Q$ there is a unique weak R-node $Q_R$ such that every child of $Q$ is a child of $Q_R$ and no other weak R-node has children in common with $Q$. Also, the children of an R’-node $Q$ form a path in the tree of $Q$’s weak R-node (i.e., $(Q_R)_T$).

We provide the following observations (3.2 and 3.3) to emphasize the relationship between weak R-nodes and R’-nodes.

**Observation 3.2.** For a weak R-node $\mathcal{R}$ and an R’-node $Q$, if $\text{Children}(Q) \cap \text{Children}(\mathcal{R}) \neq \emptyset$, then $\text{Children}(Q) \subseteq \text{Children}(\mathcal{R})$, $Q_R = \mathcal{R}$ and $Q \in \mathcal{R}_{R’}$.

**Observation 3.3.** If $Q_0$ and $Q_1$ are R’-nodes and $\text{Children}(Q_0) \cap \text{Children}(Q_1) \neq \emptyset$, then $(Q_0)_R = (Q_1)_R$. 

Figure 3.3: A weak PR-tree $D$ with $\text{Sources}(D) = \{Q_0, Q_1, P_3, R\}$, where $\mathcal{R}_{R’} = \{Q_0, Q_1, Q_2\}$ and $\mathcal{R}_T$ is the tree given on the right. Note: we draw nodes with multiple parents once as with the node $P_0$. 
For some insight into the weak PR-tree, we examine the different ways a weak PR-tree can be drawn. In particular, to draw a weak PR-tree we must “choose” an ordering the children of P-nodes and R’-nodes. From fig. 3.1 (on page 25), it is clear that we can choose an arbitrary permutation on the children of a P-node when drawing a weak PR-tree. However, since an R’-node Q corresponds to a path in \((Q_R)_T\) we only have two ways to draw Q (i.e., left-to-right or right-to-left) since any other drawing would not represent a path in \((Q_R)_T\). Notice that, if we consider the set of drawings of a P-node or R’-node and its descendants, this is precisely a set of equivalent PQ-trees (with the R’-nodes acting as Q-nodes) – this is formalized as observation 3.4 below. However, as we will see in section 3.2.1, these operations alone (i.e., permuting the children of a P-node and reversing the children of an R’-node) are not enough to establish the concept of a consistent set of a weak PR-tree.

**Observation 3.4.** The weak PR-trees \(D\) over a leaf set \(U\) where Sources\((D)\)={\(A, R_0, R_1, ..., R_{\gamma-1}\)} and \(A\) is either a P-node or R’-node and \(R_0, R_1, ..., R_{\gamma-1}\) are weak R-nodes (i.e., \(D\) has only one source node that is not a weak R-node), correspond to the equivalence classes of PQ-trees over \(U\). Note: we call such a weak PR-tree a rooted PR-tree.

### 3.1.2 The Constraint Set

In this section we use the relationship between the weak PR-tree and PQ-tree (as in observation 3.4) to generalize the constraint set of a PQ-tree (as presented in section 2.3) to weak PR-trees. In particular, we define the constraint set \(S_D\) of a weak PR-tree \(D\) through the same constructive approach we used to define the weak PR-tree as follows (note: \(S_D = \{L(D)\}\) when \(D\) is a leaf).

When \(D\) is formed as in part 2 of definition 3.1 (i.e., by joining weak PR-trees \(D_0, D_1, ..., D_{k-1}\) through nodes \(N_0, N_1, ..., N_{k-1}\) via a new P-node \(P\)), \(S_D\) consists of the
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

28

constraints of each $D_i$ together with the leaf set of the new P-node $P$; i.e.,

$$S_D = \left( \bigcup_{i=0}^{k-1} S_{D_i} \right) \cup \{ L(P) \}. \quad (3.1)$$

Similarly, when $D$ is formed as in part 3 of definition 3.1 (i.e., via a weak R-node $R$), we again keep the constraints of each $D_i$, but we now have many more constraints. In particular, the leaf set $L(Q)$ of each $R'$-node $Q \in R_{R'}$ is a constraint of $D$ and the leaf set $L(N_i) \cup L(N_{i^*})$ of each edge $N_i, N_{i^*}$ of $R_T$ is a constraint of $D$; i.e.,

$$S_D = \left( \bigcup_{i=0}^{k-1} S_{D_i} \right) \cup \left\{ L(Q) : Q \in R_{R'} \right\} \cup \left\{ L(N_i) \cup L(N_{i^*}) : (N_i, N_{i^*}) \in E(R_T) \right\}. \quad (3.2)$$

We now consider some properties of the constraint set $S_D$ of a weak PR-tree $D$. Notice that, the constraint set $S_D$ nearly uniquely determines the weak PR-tree $D$. In fact, if we were to allow duplicates in $S_D$, then $S_D$ would uniquely determine $D$. In particular, since $S_D$ has the very strict hierarchical structure described by equations 3.1 and 3.2, the only ambiguity regarding a weak PR-tree $D^*$ with $S_{D^*} = S_D$ is the absence/presence of certain $R'$-node sources that specify precisely one edge in the tree of their respective weak $R$-nodes. It is easy to see that $S_D$ unambiguously determines all of $D^*$’s P-nodes, the trees of $D^*$’s weak $R$-nodes, and most of $D^*$’s $R'$-nodes (i.e., each $R'$-node $Q$ where either $Q$ is not a source, or $Q$ specifies at least two edges or some edge is exclusively specified by $Q$). Notice that these unambiguous P-nodes, trees, and $R'$-nodes are sufficient to form a weak PR-tree $D^{**}$ with $S_D = S_{D^{**}}$ (in fact $D^{**}$ is contained in every weak PR-tree whose constraint set is $S_D$). However, there are additional $R'$-nodes which could belong to $D^*$. In particular, if there is an edge in the tree of a weak $R$-node of $D^{**}$ for which there is no $R'$-node $Q$ that specifies precisely this edge, then adding $Q$ to $D^{**}$ does affect $S_{D^{**}}$. Thus, we have the following observation.

\footnote{The only duplicate constraints involve an $R'$-node $Q$ that specifies precisely one edge in $(Q_R)_T$.}
Observation 3.5. For every weak PR-tree $D$, there is a weak PR-tree $D^{**}$ such that $S_{D^{**}} = S_D$ and for every weak PR-tree $D^*$ with $S_D = S_{D^*}$, $D^*$ can be obtained by adding $R'$-node sources to $D^{**}$.

3.1.3 Relating Graphs and Weak PR-trees via Constraint Sets

In this section we consider some weak PR-trees and graphs with the same set of PT-constraints. Recall that the PT-constraints of a constraint set $S$ are the constraints $S \subseteq U_S$ where $S$ is a path in every path-tree model that satisfies $S$. We will formalize what it means for a graph to be “represented” by a weak PR-tree in section 3.3.2.

Recall that, from any collection of sets $S$ we can construct an intersection graph $I(S)$ where each set is a vertex and two vertices are adjacent iff their corresponding sets have a non-empty intersection. Notice that, for a general collection of sets $S$, the maximal cliques of the intersection graph $I(S)$ do not necessarily correspond to the elements in $U_S$ (e.g., for $S = \{\{u_0, u_1\}, \{u_1\}\}$, $I(S)$ is just a single edge). However, for a weak PR-tree $D$, since the constraint set $S_D$ includes the constraint $S_v = \{u\}$ for each element $u \in U_S$, the graph $I(S_D)$ will have a distinct maximal clique for each $u \in U_S$ (i.e., the maximal clique which contains the vertex $v$). Also, it is not difficult to see that every maximal clique of $I(S_D)$ contains some $v$ with $S_v = \{u\}$ (i.e., $S_D$ is helly)\(^8\).

In fig. 3.4 we present a fairly simple weak PR-tree $D$ and the corresponding intersection graph $I(S_D)$ (note: $D$ trivially has the same constraint set as $I(S_D)$). In fig. 3.5 we give a much simpler graph $G$ where $S_G \subseteq S_D \subseteq PT_G = PT_{S_D}$. Notice that the graph in fig. 3.5 is a false twin odd sun and is an induced subgraph of the graph on the right of fig. 3.4. In particular, these graphs are not path graphs (since false twin odd suns are part of the FISC of path graphs). Thus, many graphs can have the same weak PR-tree and there are graphs that are not path graphs which are “representable” by weak PR-trees.

\(^8\) We do not prove this explicitly nor will we need this in any of our proofs. Also, it follows from the fact that weak PR-tree only represent chordal graphs (see corollary 3.30).
Maximal Cliques:
\[ u_i = \begin{cases} 
\{v_i, q_i, q_{i+1 \text{mod } 3}, e_i\}, & 0 \leq i \leq 2 \\
\{v_i, q_{0}, q_{1}, q_{2}, q_{3}, e_{0}, e_{1}, e_{2}\}, & 3 \leq i \leq 4 
\end{cases} \]

Figure 3.4: A weak PR-tree \( D \) with \( \text{Sources}(D) = \{Q_0, Q_1, Q_2, R, R^*\} \) where \( R_{R'} = \{Q_0, Q_1, Q_2\} \), \( R_T = T_0 \), \( R^*_{R'} = \{Q_3\} \), and \( R^*_T = T_1 \) (top-left), and the graph \( I(S_D) \) where the vertices \( v_i, q_k, e_\ell \) of \( I(S_D) \) correspond to the constraints \( \{u_i\} \), \( L(Q_k) \), and \( L(Q_3) \cup \{u_\ell\} \) respectively for each \( i \in [0, 4] \), \( k \in [0, 3] \), \( \ell \in [0, 2] \) (top-right). The mapping from \( L(D) \) to \( K_{I(S_D)} \) is also given (bottom). Note: in the graph \( I(S_D) \), an edge connecting a vertex to a box indicates that the vertex is adjacent to every vertex inside the box (e.g., the edge connecting \( q_0 \) to the box containing \( e_0, e_1, e_2, q_3, q_4 \) indicates that \( q_0 \) is adjacent to \( e_0, e_1, e_2, q_3, q_4 \).

Maximal Cliques:
\[ u_i = \begin{cases} 
\{v_i, q_i, q_{i+1 \text{mod } 3}, e_i\}, & 0 \leq i \leq 2 \\
\{v_i, q_{0}, q_{1}, q_{2}, q_{3}, e_{0}, e_{1}, e_{2}\}, & 3 \leq i \leq 4 
\end{cases} \]

Figure 3.5: The weak PR-tree \( D \) in fig. 3.4 with \( S_G \subseteq S_D \subseteq PT_G \) for the graph \( G \) (left) with the given mapping from \( L(D) \) to \( K_G \) (right). Note: \( G \) is a false twin odd sun (i.e., the false twin 3-sun) and is an induced subgraph of \( I(S_D) \).

We similarly provide two graphs for the weak PR-tree \( D \) given in fig. 3.3. In particular, fig. 3.6 shows the graph \( I(S_D) \) and fig. 3.7 displays a much simpler graph \( G \) with \( S_G \subseteq S_D \subseteq PT_G \). Notice that both of these graphs contain odd suns (e.g., \( G[v_1, v_4, v_7, q_0, q_1, p_3] \) is a 3-sun) and as such are not directed path graphs (since odd suns are part of the FISC of directed path graphs).

In section 3.3 we will discuss the PT-constraints of a weak PR-tree \( D \). In particular, we will prove that each PT-constraint \( S \) of \( S_D \) (i.e., \( S \in PT_{S_D} \)) is either the leaf set of a P-node (i.e., \( S = L(P) \)) or the leaf set of a path \( N_0, N_1, ..., N_{k-1} \) in the tree of a weak
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

Figure 3.6: The graph $\mathcal{I}(S_D)$ with $S_G = S_D$ where $D$ is the weak PR-tree given in fig. 3.3 and the vertices $v_i, p_j, q_k, e_\ell$ in $G$ correspond to the constraints $\{u_i\}, L(P_j), L(Q_k)$, and $L(P_\ell) \cup \{u_0\}$ respectively for each $i \in [0, 13], j \in [0, 3], k \in [0, 2], \ell \in [0, 2]$, and $e_3$ and $e_4$ correspond to $L(P_1) \cup \{u_{12}\}$ and $\{u_{12}, u_{13}\}$ respectively. The mapping from $L(D)$ to $\mathcal{K}_{\mathcal{I}(S_D)}$ is given on the right.

Figure 3.7: A graph $G$ with $S_G \subseteq S_D \subseteq \mathcal{P}T_S$ where $D$ is the weak PR-tree in fig. 3.3.

R-node (i.e., $S = \bigcup_{i=0}^{k-1} L(N_i)$). We will then use this result to formalize what it means for a constraint set $S$ or graph $G$ to be “representable” by a weak PR-tree (see def. 3.27).

### 3.2 Formalizing PR-trees and Strong PR-trees

In this section we establish the **PR-tree** and **strong PR-tree**. This consists of two parts.

In section 3.2.1 we describe how to, from a given weak PR-tree $D$, extract the trees and directed trees which satisfy $D$’s constraint set. In particular, we establish the **consistent set** and **directed consistent set** of $D$. This discussion of consistent sets provides the definition of the **PR-tree** (def. 3.10) and **strong PR-tree** (def. 3.12). More specifically,
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

the (strong) PR-trees of a weak PR-tree \(D\) map directly to the elements of \(D\)'s (directed) consistent set. In section 3.2.2 we prove that the (directed) consistent set of a weak PR-tree \(D\) is precisely the set of (directed) trees which satisfy \(D\)'s constraint set.

3.2.1 The Consistent Set

In this section we describe how to generate the set of trees and the set of directed trees which satisfy the constraint set from a given weak PR-tree. Namely, we present the consistent set and directed consistent set of a weak PR-tree. This discussion will also provide the definition of the PR-tree and strong PR-tree.

Recall that each P-node and R'-node \(N\) in a drawing of a weak PR-tree \(D\) induces a rooted ordered subtree of \(D\) (i.e., \(N\) is a PQ-tree). Furthermore, we can convert one drawing of a weak PR-tree to another by permuting the children of P-nodes and reversing the children of R'-nodes. With this in mind, one might be inclined to define the “frontier” and “directed frontier” of a fixed drawing \(F\) of a weak PR-tree as follows:

- For each P-node and R'-node \(N\), the “(directed) frontier” of \(N\) is the (directed) path specified by the left-to-right order on \(N\)'s leaves (i.e., as in the case of PQ-trees).
- The “(directed) frontier” of \(F\) is the union of the “(directed) frontiers” of \(D\)'s P-node and R'-node sources.

Unfortunately, this has two problems. The first being that such a “frontier” is not necessarily a tree and the second being that the set of “frontiers” of a weak PR-tree \(D\) does not include all trees satisfying \(S_D\). In particular, neither the “frontier” nor the “directed frontier” (see fig. 3.8) of fig. 3.3 is a tree. Furthermore, no such “frontier” of any drawing of this weak PR-tree will be a tree even though there are many trees which satisfies its constraints (e.g. fig. 3.9).

To handle these issues we add more information to a drawing of a weak PR-tree (this will provide the definition of a PR-tree and strong PR-tree). In particular, we add
orientations to the edges of the trees of the weak R-nodes in order to “guide” the creation of the frontier of a particular drawing. More specifically, the real issue is that $R'$-nodes of a particular weak R-node need to “agree” on the edges they contribute to the frontier. To guarantee this agreement, for each weak R-node $R$, we orient each edge $N, N^*$ of $R_T$ to indicate how the $R'$-nodes that specify this edge should connect $N$ to $N^*$ in their frontiers. In particular, if $N \rightarrow N^*$, then the frontier of $Q$ will connect the right-end of the frontier of $N$ to the left-end of the frontier of $N^*$ regardless of the order in which $N$ and $N^*$ occur as children of $Q$. Now, we say that a weak PR-tree is oriented when the children of its P-nodes and $R'$-nodes are ordered (as in a drawing of a weak PR-tree) and the edges of the trees of its weak R-nodes are oriented (an oriented weak PR-tree is given in fig. 3.10).

Figure 3.10: An oriented weak PR-tree $D = \{Q_0,Q_1,P_3,R\}$, where $R_{R'} = \{Q_0,Q_1,Q_2\}$ and $R_T$ is the tree on the right. Note: the frontier of $D$ is the tree on the left of fig. 3.9.

Notice that, in an oriented weak PR-tree, each P-node and $R'$-node has a clearly
defined left-end-child and right-end-child. In particular, if a P-node or R′-node \( N \) has \( N_0, N_1, ..., N_{k-1} \) as its children (in this order), then \( N_0 \) is \( N \)'s left-end-child and \( N_{k-1} \) is \( N \)'s right-end-child. Similarly, the left-leaf (right-leaf) of a P-node is the left-leaf (right-leaf) of \( N_0 \) (\( N_{k-1} \)) (note: the left-leaf and right-leaf of a leaf \( N \) is \( N \) itself). Furthermore, the left-leaf of an R′-node is either the: left-leaf of \( N_0 \) when \( N_0 \rightarrow N_1 \) is an edge of \((Q_R)T\); or, right-leaf of \( N_0 \) when \( N_0 \leftarrow N_1 \) is an edge of \((Q_R)T\). Similarly the right-leaf of an R′-node is either the: right-leaf of \( N_{k-1} \) when \( N_{k-2} \rightarrow N_{k-1} \) is an edge of \((Q_R)T\); or, left-leaf of \( N_{k-1} \) when \( N_{k-2} \leftarrow N_{k-1} \) is an edge of \((Q_R)T\). For example, in fig. 3.10, the left-leaf of \( Q_0 \) is \( u_1 \) and the right-leaf of \( P_3 \) is \( u_{11} \). Also, the leaf-leaf of \( Q_2 \) is \( u_{13} \) and \( Q_2 \)'s right-leaf is \( u_9 \). With these “lefts” and “rights” we define the frontier and directed frontier of an oriented weak PR-tree constructively (as in the definition of weak PR-trees), as follows.

**Definition 3.6.** The (directed) frontier of an oriented weak PR-tree \( D \) is presented in the following three cases and is denoted \((\text{dir.})\text{Frontier}(D)\).

1. If \( D \) is a leaf, the frontier of \( D \) is the graph with the leaf of \( D \) as its sole vertex.

2. If \( D \) is constructed via a P-node \( P \) (part 2 of def. 3.1) whose children are \( N_0, N_1, ..., N_{k-1} \) (in this order), the (directed) frontier of \( D \) is the following (directed) graph:

\[
\left( \bigcup_{i=0}^{k-1} (\text{dir.})\text{Frontier}(D_i) \right) \cup \{ \text{the edge } u_i(\rightarrow)u_{i+1}^* : i \in [0, k-2] \text{ and } u_i \text{ is the right-leaf of } N_i \text{ and } u_{i+1}^* \text{ is the left-leaf of } N_{i+1} \}.
\]

3. If \( D \) is constructed via a weak R-node \( R \) (part 3 of def. 3.1) whose children are \( N_0, N_1, ..., N_{k-1} \), the (directed) frontier of \( D \) is the following (directed) graph:

\[
\left( \bigcup_{i=0}^{k-1} (\text{dir.})\text{Frontier}(D_i) \right) \cup \{ \text{the edge } u(\rightarrow)u^* : N_i \rightarrow N_i^* \text{ is an edge of } R_T \text{ and } u \text{ is the right-leaf of } N_i \text{ and } u^* \text{ is the left-leaf of } N_i^* \}.
\]

The (directed) frontier of a node \( N \) in \( D \) is the subgraph of the (directed) frontier of \( D \) induced by the leaf set of \( N \); i.e., \((\text{dir.})\text{Frontier}(N) = (\text{dir.})\text{Frontier}(D)[L(N)]\).

Note: For the oriented weak PR-tree \( D \) given in fig. 3.10, \( \text{Frontier}(D) \) is the tree on the left of fig. 3.9 and \( \text{dir.Frontier}(D) \) is given in fig. 3.11.
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

Figure 3.11: The directed frontier of the oriented weak PR-tree in fig. 3.10. Note: this directed tree fails the constraint \( L(Q_0) = \{u_1, u_2, u_3, u_4, u_5, u_6, u_{12}, u_{13}\} \).

Notice that, regardless of the choice of orientation, the (directed) frontier of an oriented weak PR-tree is always a (directed) tree. Additionally, for every P-node and R'-node \( N \) in an oriented weak PR-tree \( D \), \( L(N) \) is connected in the frontier of \( D \). In particular, \((\text{dir.})\text{Frontier}(D)[L(N)]\) is a (directed) tree. Recall that a tree-tree model of a graph \( G \) is a tree \( T \) where \( V(T) = K_G \) and \( T[S_v] \) is a tree for every \( v \in V(G) \). Thus, if there is a weak PR-tree \( D \) where \( S_D \supseteq S_G \) then, for every orientation \( D' \) of \( D \), \( \text{Frontier}(D') \) is a tree-tree model of \( G \). Notice that, this means that \( G \) is a chordal graph since chordal graphs are characterized by having tree-tree models. We formalize this through the following observation and corollary.

**Observation 3.7.** For a weak PR-tree \( D \), the (directed) frontier of each orientation of \( D \) is a (directed) tree. Moreover, for a graph \( G \), if \( S_D \supseteq S_G \), then the frontier of each orientation of \( D \) is a tree-tree model of \( G \).

We now take a slightly closer look at the frontier of certain orientated weak PR-trees. Consider a particular P-node or R'-node \( N \) in a weak PR-tree \( D \). Notice that, we can easily choose an orientation \( D' \) of \( D \) so that \( \text{dir.} \text{Frontier}(D')[L(N)] \) is a directed path (e.g., \( Q_1 \) in fig. 3.10 has the directed path \( u_1, u_2, u_3, u_0, u_7, u_8, u_9 \) as its directed frontier – see fig. 3.11). In particular, for each R'-node \( Q \) that is either \( N \) or a descendant of \( N \), order \( Q \) to match the path specified by \( Q \) and orient the edges of this path from left to right. Also, we order and orient the remainder of \( D \) so that it is a legal orientation (i.e., the remaining R'-nodes according to their paths). This oriented weak PR-tree \( D' \) will clearly have \( \text{dir.} \text{Frontier}(D')[L(N)] \) as a directed path. We formalize this in the following observation.
Observation 3.8. For every P-node and R'-node \( N \) in a weak PR-tree \( D \), there is an orientation \( D' \) of \( D \) such that \( \text{dir.Frontier}(D')[L(N)] \) is a directed path. Moreover, for a graph \( G \), if \( S_D \supseteq S_G \), then \( \text{Frontier}(D') \) is a tree-model of \( G \) where \( L(N) \) is a path.

However, the choice of orientation applied to the tree of a weak R-node is also a significant factor. In particular, to ensure that the frontier of each R'-node \( Q \) is a path, we must avoid having \( N \rightarrow N^* \leftarrow N^{**} \) (or \( N \leftarrow N^* \rightarrow N^{**} \)) in \( (Q_R)_T \) where \( N, N^*, N^{**} \) is a subpath specified by \( Q \) and \( N^* \) has at least two leaves. Such an assignment of orientations would connect the same end of the frontier of \( N^* \) to both the frontier of \( N \) and the frontier of \( N^{**} \) (i.e., the frontier of \( Q \) would not be a path). We refer to such a carefully oriented weak R-node as an \( R \)-node (see def. 3.9). Furthermore, we define a PR-tree as a drawing of a weak PR-tree where every weak R-node is an R-node (see def. 3.10). We similarly define the strong R-node and strong PR-tree in def. 3.11 and 3.12 respectively.

**Definition 3.9.** An \( R \)-node \( R \) is a weak R-node where each R'-node is ordered (i.e., as in a drawing of a weak R-node) and the edges of \( R_T \) have been oriented so that, for each R'-node \( Q \in R_{R'} \), the path \( X = N_0, N_1, ..., N_{k-1} \) in \( R_T \) specified by \( Q \) satisfies the following condition:

For each internal node \( N_i \) of \( X \) (i.e., \( i \in [1, k - 2] \)), if \( N_i \) is not a leaf, then

either \( N_{i-1} \rightarrow N_i \rightarrow N_{i+1} \) in \( R_T \) or \( N_{i-1} \leftarrow N_i \leftarrow N_{i+1} \) in \( R_T \).

Note: The oriented weak R-node \( R \) in fig. 3.10 is an R-node. However, some weak R-nodes cannot be converted into R-nodes (e.g., \( R \) in fig. 3.4).

**Definition 3.10.** A PR-tree \( D \) is a weak PR-tree (see def. 3.1) where every P-node is ordered (i.e., its children are ordered) and every weak R-node is an R-node.

Note: the oriented weak PR-tree in fig. 3.10 is a PR-tree.

**Definition 3.11.** A strong R-node \( R \) is a weak R-node where each R'-node is ordered (i.e., as in a drawing of a weak R-node) and the edges of \( R_T \) has been oriented so that,
for each $R'$-node $Q \in \mathcal{R}_{R'}$ where $N_0, N_1, \ldots, N_{k-1}$ are $Q$’s children and they are in this order, $N_0 \rightarrow N_1 \rightarrow \ldots \rightarrow N_{k-1}$ is a directed path in $\mathcal{R}_T$.

Note: Some weak $R$-nodes can be converted into $R$-nodes but not into strong $R$-nodes (e.g., $\mathcal{R}$ in fig. 3.10 – also see fig. 3.11).

**Definition 3.12.** A strong PR-tree $D$ is a weak PR-tree where every $P$-node is ordered (i.e., its children are ordered) and every weak $R$-node is a strong $R$-node.

Notice that, for a (strong) PR-tree $D$, the (directed) frontier of a $P$-node or $R'$-node $N$ is a (directed) path where the left-leaf of $N$ is “first” on this (directed) path and the right-leaf of $N$ is “last” on this (directed) path. In particular, we denote this “order on the leaves of a $P$-node or $R'$-node $N$” as $\sigma(N)$ and state this observation as follows.

**Observation 3.13.** In a (strong) PR-tree, the (directed) frontier of each $P$-node and $R'$-node $N$ specifies an ordering $\sigma(N) = u_0, u_1, \ldots, u_{z-1}$ (of $L(N)$) where $u_0$ is the left-leaf of $N$ and $u_{z-1}$ is the right-leaf of $N$.

We conclude this section by defining the consistent set and directed consistent set of a weak PR-tree (see def. 3.14 below). To do this we consider equivalent (strong) PR-trees. In particular, we say that two (strong) PR-trees $D$ and $D^*$ are equivalent, denoted $D \equiv D^*$, when they have the same underlying weak PR-tree. In other words, one can be transformed into the other through zero or more of the following equivalence transformations:

1. Permute the children of a $P$-node.
2. Reverse the children of an $R'$-node.
3. Re-orient the edges of the tree of a (strong) $R$-node.

**Definition 3.14.** The (directed) consistent set of a weak PR-tree $D$ is the set of (directed) frontiers of the (strong) PR-trees with $D$ as their underlying weak PR-tree.

Note: We use (dir.)Consistent($D$) to denote this set.
In the next section we prove that the (directed) consistent set does in fact correspond to the (directed) trees which satisfy the constraints $S_D$ of a weak PR-tree $D$.

### 3.2.2 Connecting $T_{S_D}$ and $\text{Consistent}(D)$

In this section we demonstrate the connection between (strong) PR-trees and (directed) path-tree models. In particular, in theorem 3.15, we prove that a tree $T$ satisfies the constraint set of a weak PR-tree $D$ iff it belongs to $D$'s consistent set (i.e., there is a PR-tree of $D$ whose frontier is $T$). Similarly, in theorem 3.16, we prove that a directed tree $T$ satisfies the constraint set of a weak PR-tree iff it belongs to $D$'s directed consistent set (i.e., there is a strong PR-tree of $D$ whose directed frontier is $T$).

**Theorem 3.15.** For a weak PR-tree $D$, a tree $T$ with vertex set $V(T) = L(D)$ belongs to $\text{Consistent}(D)$ iff for every $S \in S_D$, $T[S]$ is a path.

**Proof.**

$\Rightarrow$ This is clear from our definition of $S_D$ and $\text{Consistent}(D)$.

$\Leftarrow$ This is trivially true for the case when $D$ is a leaf. Thus, we can inductively assume this is true for all weak PR-trees with fewer than $w$ P-nodes and weak R-nodes where $w$ is the total number of P-nodes and weak R-nodes in $D$. In particular, we consider the “last” node $N$ added to $D$ (when constructing $D$ as in the definition (3.1) of weak PR-trees) – note: such a node is either a P-node or an R-node and $N$’s children $N_0$, $N_1$, ..., $N_{k-1}$ come from disjoint weak PR-trees $D_0$, $D_1$, ..., $D_{k-1}$. Notice that a tree $T$ satisfying $S_D$ can be partitioned into $k$ disjoint trees $T_0, T_1, ..., T_{k-1}$ such that $T_i$ satisfies $S_{D_i}$ (i.e., $V(T_i) = L(D_i)$) for each $i \in [0, k - 1]$. Furthermore, using our assumption, we order and orient the nodes of each $D_i$ so that it is a PR-tree with $\text{Frontier}(D_i) = T_i$. Also, we use $P_i$ to denote the path $T_i[L(N_i)]$ (note: $P_i = T[L(N_i)]$).

We now consider the two cases for the node $N$:

**Case 1: $N$ is a P-node.** Now, since $N$ is a P-node, we can permute these $P_i$s in
any way we want. Also, for each \( P_i \), we can either reverse it or keep it in its current orientation. Notice that, reversing a \( P_i \) can be thought of as reversing the order of every P-node and \( R' \)-node in \( D_i \) and reversing the orientation on every edge in the tree of each R-node. By thinking of it this way, the reversal of a \( P_i \) does not disrupt the validity of the equivalence transformation which has been applied to \( D \) so far (i.e., the frontier of \( D_i \) is unchanged and its R-nodes remain R-nodes).

Notice that, \( T \) must have \( T[L(N)] \) as a path. Furthermore, for each child \( N_i \) of \( N \), \( T[L(N_i)] \) is a subpath of this path. In particular, these subpaths occur consecutively in \( T[L(N)] \) according to some permutation \( \pi \). Thus, we order the children of \( N \) according to \( \pi \) – w.l.o.g. suppose \( N_0, N_1, ..., N_{k-1} \) is this order. Finally, we reverse each \( P_i \) as needed so that \( P_0, P_1, ..., P_{k-1} \) is the path \( T[L(N)] \).

Therefore, \( D \) is now a PR-tree (since the \( D_i \)s are PR-trees and \( P \) is now ordered) and \( \text{Frontier}(D) = T \) (as needed).

**Case 2: \( N \) is a weak R-node.** Now, since \( N \) is a weak R-node, we can choose orientations of each edge \( N_i, N_i^* \) of \( R_T \) to decide which edges \( N \) will contribute to the frontier of \( D \). Also, as in the P-node case, we have the option of reversing each \( P_i \). We selectively reverse each \( P_i \) and orient the edges of \( R_T \) through the following breadth first traversal of \( R_T \) (this is formalized as algorithm 2 below).

Choose an arbitrary \( N_i \) to start. Now, for each neighbour \( N_i^* \) of \( N_i \) in \( R_T \) we do the following. Notice that \( T \) contains an edge \( u, u^* \) where \( u \in L(N_i) \) and \( u^* \in L(N_i^*) \) (since \( L(N_i) \cup L(N_i^*) \) is a constraint in \( \mathcal{S}_D \)). Furthermore, \( u \) is either a left-leaf or right-leaf of \( N_i \). Similarly, \( u^* \) is either a left-leaf or right leaf of \( N_i^* \). Thus, by choosing between \( N_i \rightarrow N_i^* \) and \( N_i \leftarrow N_i^* \) as the orientation of the edge in \( R_T \) and reversing \( P_i^* \) (as needed), the frontier of \( D \) will have the edge \( u, u^* \) (note: we do not reverse \( P_i \)). For example, if \( u \) is the left-leaf of \( N_i \) and \( u^* \) is the left-leaf of \( N_i^* \), then we use the edge \( N_i \leftarrow N_i^* \) and we reverse \( P_i^* \). We then repeat this process for each such \( N_i^* \) and its unvisited neighbours.
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

Algorithm 2: Processing a weak R-node $R$ in the proof of theorem 3.15.

1. Choose an arbitrary child $N_i$ of $R$ and place it on a queue $Q$.
2. while $Q$ is not empty do
   3. Dequeue the first element $N_i$ from $Q$
   4. for each unvisited neighbour $N_{i*}$ of $N_i$ do
      5. Let $u, u^*$ be the edge of $T$ connecting $u \in V(P_i)$ and $u^* \in V(P_{i*})$
      6. if $u$ is the right-leaf of $N_i$ then
         7. Orient the edge $N_i, N_{i*}$ in $R_T$ as $N_i \rightarrow N_{i*}$
         8. if $u^*$ is the right-leaf of $N_{i*}$ then Reverse $P_{i*}$
      9. else // $u$ is the left-leaf of $N_i$
         10. Orient the edge $N_i, N_{i*}$ in $R_T$ as $N_i \leftarrow N_{i*}$
         11. if $u^*$ is the left-leaf of $N_{i*}$ then Reverse $P_{i*}$
      12. Mark $N_{i*}$ as visited and place $N_{i*}$ on $Q$

Notice that, in this process, once we decide on the orientation of a $P_i$ we never change it (this happens when $N_i$ becomes visited). In particular, after processing an edge $N_i, N_{i*}$ of $R_T$ the edge contributed to the frontier of $D$ by this edge becomes fixed and is the appropriate edge from $T$. Thus, after this process finishes, the frontier of $D$ will be $T$.

Furthermore, $D$ is a PR-tree since each $D_i$ is a PR-tree and $R$ is an R-node. In particular, for each $R'$-node $Q \in R_{R'}$, $Frontier(Q) = T[L(Q)]$ (since $T$ satisfies the constraint $L(Q)$). Thus the orientations applied to the edges of the path specified by $Q$ satisfy the condition given in the definition (3.9) of an R-node.

Theorem 3.16. For a weak PR-tree $D$, a directed tree $T$ with vertex set $V(T) = L(D)$ belongs to $dir.Consistent(D)$ iff for every $S \in \mathcal{S}_D$, $T[S]$ is a directed path.

Proof. This proof follows similarly to the proof of theorem 3.15.

$\implies$ This is clear from our definition of $\mathcal{S}_D$ and $dir.Consistent(D)$.

$\impliedby$ This is trivially true for the case when $D$ is a leaf. Thus, we can inductively assume this is true for all weak PR-trees with fewer than $w$ P-nodes and weak R-nodes where $w$ is the total number of P-nodes and weak R-nodes in $D$. In particular, we consider the “last” node $N$ added to $D$ (when constructing $D$ as in the definition (3.1) of weak PR-
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

trees) – note: such a node is either a P-node or an R-node and N’s children \(N_0, N_1, \ldots, N_{k-1}\) come from disjoint weak PR-trees \(D_0, D_1, \ldots, D_{k-1}\). Notice that a tree \(T\) satisfying \(S_D\) can be partitioned into \(k\) disjoint trees \(T_0, T_1, \ldots, T_{k-1}\) such that \(T_i\) satisfies \(S_{D_i}\) (i.e., \(V(T_i) = L(D_i)\)) for each \(i \in [0, k-1]\). Furthermore, using our assumption, we order and orient the nodes of each \(D_i\) so that it is a strong PR-tree with \(dir.Frontier(D_i) = T_i\). Also, we use \(P_i\) to denote the path \(T_i[L(N_i)]\) (note: \(P_i = T[L(N_i)]\)). Notice that from the definitions of strong PR-tree and directed frontier, each \(P_i\) is a directed path that “starts” with the left-leaf of \(N_i\) and “ends” with the right-leaf of \(N_i\). Thus, we will not be reversing any of the \(P_i\)s.

We now consider the two cases for the node \(N\):

**Case 1: \(N\) is a P-node.** Now, since \(N\) is a P-node, we can permute these \(P_i\)s in any way we want. In particular, we simply permute the children of \(N\) so that they match the order in which \(P_0, P_1, \ldots, P_{k-1}\) occur in \(T[L(N)]\). Thus, since each \(P_i\) is a directed path (from “left” to “right”), the frontier of \(N\) after this transformation is \(T[L(N)]\) and, consequently, the directed frontier of \(D\) is \(T\). Furthermore, \(D\) is a strong PR-tree since \(P\) is ordered and each \(D_i\) is a strong PR-tree.

**Case 2: \(N\) is a weak R-node.** Now, since \(N\) is a weak R-node, we can choose the orientation of each edge \(N_i, N_i^*\) of \(R_T\) to decide which edges \(N\) will contribute to the frontier of \(D\). Also, to ensure that \(N\) becomes a strong R-node, we may need to reverse some of the \(R’\)-nodes in \(R_{R’}\) so that they are consistent with their corresponding directed paths in \(R_T\).

First, we orient the edges of \(R_T\). Notice that, for each edge \(N_i, N_i^*\) in \(R_T\), there are only two ways to connect \(P_i\) and \(P_{i+1}\) so that the result is a directed path (i.e., by connecting the right-leaf of \(N_i\) to the left-leaf of \(N_i^*\) or vice versa). Furthermore, both options are captured by the choice of edge \(N_i \rightarrow N_i^*\) or \(N_i \leftarrow N_i^*\). Thus, the directed edge in \(T\) connecting \(P_i\) to \(P_i^*\) provides the orientation of the edge \(N_i, N_i^*\) in \(R_T\). Notice that applying these orientations necessarily results in each \(R’\)-node \(Q\) in \(R_{R’}\) specifying
a directed path in \( R_T \) (since \( T \) satisfies the constraint \( L(Q) \)). Thus, we simply order \( Q \) to match the direction of this path.

Thus, we have oriented \( R \) so that it is a strong R-node (i.e., \( D \) is a strong PR-tree) with \( T[L(R)] \) as its directed frontier and the directed frontier of \( D \) is \( T \).

\[ \square \]

3.3 Path-Tree Constraints of a Weak PR-tree

We now return to the PT-constraints of a weak PR-tree. This section consists of two parts. In the first part (section 3.3.1) we prove that, when \( T_S \neq \emptyset \) (i.e., \( Consistent(D) \neq \emptyset \)), every PT-constraint of a weak PR-tree is either the leaf set of a P-node or the leaf set of a path in the tree of a weak R-node (see theorem 3.21). We establish this relationship by considering the least common ancestors of subsets of the leaf sets of weak PR-trees similarly to our discussion of PT-constraints in PQ-trees (see section 2.3.2). In the second part (section 3.3.2) we use this relationship to define what it means for a weak PR-tree \( D \) to represent a constraint set \( S \) (i.e., a graph with \( S_G = S \)).

3.3.1 Least Common Ancestors

In this section we discuss the least common ancestors of subsets of the leaf set of a weak PR-tree. The goal of this discussion is to identify certain special leaf sets (i.e., the PT-constraints) with the nodes of a weak PR-tree.

We first note that a set \( L \) of leaves might have more than one node that qualifies as a least common ancestor in the traditional sense (i.e., a node \( N \) where \( L \subseteq L(N) \) and \( N \) has at least two children whose leaf sets contain elements of \( L \)). In particular, when an R’-node \( Q \) is a traditional least common ancestor (TLCA) of a leaf set \( L \), its weak R-node \( Q_R \) is also a TLCA of \( L \) (as in \( L_1 = \{u_0, u_6\} \) in fig. 3.12 below). To handle this issue, for the least common ancestor (LCA) of \( L \), we use the “minimal subtree” of the tree of a weak R-node rather than R’-nodes and weak R-nodes. Notice that, if a P-node
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

$P$ is a TLCA of $L$, then $P$ is the only TLCA of $L$ (this follows from the definition (3.1) of weak PR-trees). We formalize the least common ancestor of a leaf set in def. 3.17.

Furthermore, since a weak PR-tree can have multiple source nodes, some sets of leaves can have no TLCA (e.g., $L_0 = \{u_0, u_3, u_8\}$ in fig. 3.12 below). In such a case, we consider the collection of all maximal subsets $L^*$ of $L$ such that $L^*$ has a least common ancestor. In particular, in the absence of a least common ancestor, we consider the least common set of ancestors (LCSA) as in def. 3.18 below.

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{pr_tree.png}
\caption{A weak PR-tree $D = \{P_1, Q_0, Q_1, R\}$ with $R_T = T$ and $R_{R'} = \{Q_0, Q_1, Q_2\}$. Note: the leaf set $L_0 = \{u_0, u_3, u_8\}$ has no common ancestor, and the leaf set $L_1 = \{u_0, u_6\}$ has multiple TLCAs: $R$, $Q_0$, and $Q_1$. Furthermore, the LCSA of $L_0$ is $R_T[\{P_0, u_7, u_8\}]$, $P_1$ and the LCA of $L_1$ is $R_T[\{P_0, u_6\}]$.}
\end{figure}

**Definition 3.17.** For a weak PR-tree $D$ and $S \subseteq L(D)$, $X$ is the least common ancestor (LCA) of $S$ in $D$, denoted $LCA(D, S)$, when one of the following holds:

1. $X$ is a leaf of $D$ and $S = \{X\}$ (i.e., $X$ is the only element in $L$).

2. $X$ is a P-node, $S \subseteq L(X)$, and $X$ has at least two children whose leaf sets contain elements of $S$.

3. There is a weak R-node $R$ where $S \subseteq L(R)$, $R$ has at least two children whose leaf sets contain elements of $S$, and $X$ is the minimal subtree of $R_T$ such that $S \subseteq L(X)$ (i.e., $L(X) = \bigcup_{N \in V(X)} L(N)$).

Note: when $X$ is a subtree of the tree of a weak R-node $R$ we define $\text{Frontier}(X)$ to be $\text{Frontier}(R)[L(X)]$. 

Definition 3.18. For a weak PR-tree \( D \) and \( L \subseteq L(D) \), the least common set of ancestors (LCSA) of \( L \) in \( D \), denoted \( LCSA(D, L) \), is constructed as follows:

For every \( L^* \subseteq L \), if \( L^* \) is maximal such that \( LCA(D, L^*) \) exists, then \( LCA(D, L^*) \in LCSA(D, L) \).

Notice that, the LCA of a set of leaves \( L \) is well defined when it has a TLCA. In particular, from the definition (3.1) of weak PR-trees, when a P-node is a TLCA of \( L \) it is the only TLCA of \( L \). Furthermore, when a weak R-node \( R \) is a TLCA of \( L \), the only other nodes that could be TLCAs are the \( R' \)-nodes of \( R \). Thus, by choosing the minimal subtree of \( R_T \) rather than these nodes, the LCA is unique. This is formalized in the following observation. This uniqueness immediately generalizes to LCSAs (as in corollary 3.20 below).

Observation 3.19. For a weak PR-tree \( D \) and \( S \subseteq L(D) \), if \( S \) has a common ancestor, then the LCA of \( S \) exists and it is unique.

Corollary 3.20. For a weak PR-tree \( D \) and \( L \subseteq L(D) \), \( LCSA(D, L) \) is unique.

With these tools in place, we are now ready to examine PT-constraints of weak PR-trees. In particular, we consider a set \( S \subseteq L(D) \) where \( S \) is a path in every \( T \in Consistent(D) \). From theorem 3.15, we can see that all such subsets of \( L(D) \) are precisely the PT-constraints of \( S_D \). Thus, to understand the PT-constraints of \( S_D \), we consider the LCSA of a PT-constraint \( S \). In the following theorem we prove that \( S \) has an LCA whose leaf set is precisely \( S \).

Theorem 3.21. For a weak PR-tree \( D \), if \( Consistent(D) \neq \emptyset \) (i.e., \( T_{S_D} \neq \emptyset \)), then, for every PT-constraint \( S \) of \( S_D \), \( X = LCA(D, S) \) exists and one of the following is true:

(1) \( X \) is a leaf or P-node such that \( S = L(X) \).

(2) \( X \) is a path \( N_0, N_1, ..., N_{k-1} \) in the tree of a weak R-node such that \( S = \bigcup_{i=0}^{k-1} L(N_i) \).
Proof. This is trivially true for the case when $D$ is a leaf. Thus, we can inductively assume this is true for all weak PR-trees with fewer than $w$ P-nodes and weak R-nodes where $w$ is the total number of P-nodes and weak R-nodes in $D$. In particular, we consider the “last” node $N$ added to $D$ (when constructing $D$ as in the definition (3.1) of weak PR-trees) – note: such a node is either a P-node or a weak R-node and $N$’s children $N_0, N_1, ..., N_{k-1}$ come from disjoint weak PR-trees $D_0, D_1, ..., D_{k-1}$. Notice that the set $S$ can be partitioned into $k$ disjoint sets $S_0, S_1, ..., S_{k-1}$ such that $S_i \subseteq L(D_i)$ and w.l.o.g., there exists $\ell \in [1, k]$ such that for every $i \in [0, \ell - 1]$, $S_i \neq \emptyset$, and for every $i \in [\ell, k - 1]$, $S_i = \emptyset$. Furthermore, if $\ell = 1$, then $S = S_0 \subseteq L(D_0)$ and $X_0 = LCA(D, S)$ (satisfying the required condition). Thus, we consider the case when $\ell \geq 2$.

Claim: For every $i \in [0, \ell - 1]$, $S_i$ is a PT-constraint of $D_i$.

Suppose $S_i$ is not a PT-constraint of $D_i$, then there is a PR-tree $D_i^*$ of $D_i$ where $\text{Frontier}(D_i^*)[S_i]$ is not a path. Thus, since $\text{Consistent}(D) \neq \emptyset$, we have a PR-tree $D^*$ of $D$ with $D_i$ oriented and ordered as $D_i^*$. Furthermore, the frontier of $D_i^*$ is a subtree of the frontier of $D^*$. Therefore, $\text{Frontier}(D^*)[S]$ is not a path.

Thus, by our assumption, we know that, for each $i \in [0, \ell - 1]$ (i.e., $S_i \neq \emptyset$), $X_i = LCA(D_i, S_i)$ exists and satisfies one of (1)-(2).

Notice that the only way to connect $S_0$ to $S_z$ ($z \in [1, \ell - 1]$) in the frontier of $D$ is through edge(s) introduced by $N$. Therefore, for each $N_i$ ($i \in [0, \ell - 1]$), $L(N_i) \cap S_i \neq \emptyset$ (i.e., $N \in LCSA(D, S)$). Additionally, since $X_i = LCA(D_i, S_i)$ ($i \in [0, \ell - 1]$) one of the following is true: $S_i = L(N_i)$ (i.e., $N_i = X_i$), $S_i \supset L(N_i)$, or $L(N_i) \setminus S_i \neq \emptyset$.

Claim: For every $i \in [0, \ell - 1]$, $L(N_i) \setminus S_i = \emptyset$.

Suppose that, for some $i \in [0, \ell - 1]$, $L(N_i) \setminus S_i \neq \emptyset$ (i.e., $L(N_i)$ is not a subset of $S_i$).

Let $D^*$ be a PR-tree of $D$. Since $S$ is a PT-constraint, $\text{Frontier}(D^*)[S]$ is a path. Now, consider the PR-tree $D^{**}$ where the frontier of $N_i$ is reversed (recall that we can do this by reversing everything in $D_i$ as in the proof of theorem 3.15). Notice
that, since the frontier of \( N_i \) contains elements which are not in \( S \), the frontier of \( N \) in \( D^{**} \) induced on \( S \) is no longer connected (i.e., \( \text{Frontier}(D^{**})[S] \) is not a path).

Thus, for every \( i \in [0, \ell - 1] \), \( L(N_i) \subseteq S_i \); i.e., in every PR-tree \( D^* \) of \( D \), \( \text{Frontier}(N_i) \) is a subpath of \( \text{Frontier}(X_i) \). We now consider the two possible cases for \( N \).

**Case 1:** \( N \) is a P-node. Clearly, \( \ell = k \). Otherwise (\( \ell < k \)), we have a PR-tree \( D^* \) of \( D \) that orders the children of \( N \) such that \( N_\ell \) is between \( N_0 \) and \( N_1 \); i.e., \( \text{Frontier}(D^*)[S] \) is disconnected – contradicting the fact that \( S \) is a PT-constraint of \( D \). We now rule out \( N_i \) being a descendant of \( X_i \) (for every \( i \in [0, k - 1] \)).

Suppose that, for some \( i \in [0, k - 1] \), \( L(N_i) \) is a proper subset of \( S_i \); i.e., in every PR-tree \( D^* \) of \( D \), \( \text{Frontier}(N_i) \) is a proper subpath of \( \text{Frontier}(X_i) \). Since \( N \) is a P-node and P-nodes have at least three children, there is a PR-tree \( D^* \) of \( D \) where \( N_i \) is a middle child of \( N \) with respect to the ordering on \( N \)'s children in \( D^* \). We now consider \( H = \text{Frontier}(D^*)[L(N) \cup L(X_i)] \). Clearly, \( H \) is connected and is not a path (since either the left-leaf or right-leaf of \( N_i \) has degree 3 in \( H \)). Furthermore, \( H \) is a subtree of \( \text{Frontier}(D^*)[S] \) (since \( V(H) = L(N) \cup L(X_i) \subseteq S \)). This contradicts the fact that \( S \) is a PT-constraint.

Therefore, \( N_i = X_i \) for every \( i \in [0, k - 1] \); i.e., \( N = \text{LCA}(D, S) \) and \( S = L(N) \).

**Case 2:** \( N \) is a weak R-node. Notice that, since \( S \) is a PT-constraint, the children \( N_0, N_1, \ldots, N_{\ell - 1} \) must form a path in \( N_T \). Now, as in the P-node, case we just need to exclude each \( N_i \) (\( i \in [0, \ell - 1] \)) from being a descendant of \( X_i \).

Suppose that, for some \( i \in [0, \ell - 1] \), \( L(N_i) \) is a proper subset of \( S_i \) i.e., in every PR-tree \( D^* \) of \( D \), \( \text{Frontier}(N_i) \) is a proper subpath of \( \text{Frontier}(X_i) \). We now consider two cases regarding \( N_i \).

**Case 2a:** \( N_i \) is not a leaf (i.e., \( |L(N_i)| > 1 \)). Let \( D^* \) be a PR-tree of \( D \). Since \( S \) is a PT-constraint, \( \text{Frontier}(D^*)[S] \) is a path. In particular, w.l.o.g., in \( \text{Frontier}(D^*)[S] \), the left-leaf \( u \) of \( N_i \) is adjacent to some leaf \( u_X \in L(X_i) \setminus L(N_i) \) and the right-leaf \( u^* \) of \( N_i \) is adjacent to some leaf \( u_N \in L(N_{i^*}) \) (for some \( i^* \in [0, \ell - 1] \setminus \{i\} \)).
Now, consider the PR-tree $D^{**}$ where the frontier of $N_i$ is reversed (recall that we can do this by reversing everything in $D_i$ as in the proof of theorem 3.15). Notice that this also reverses the frontier of $X_i$ (i.e., $u_X$ is still adjacent to $u$). However, in the frontier of $D^{**}$, we now have both $u_X$ and $u_N$ adjacent to $u$. Thus, since $N_i$ has at least two leaves, $u$ is a degree three node in $\text{Frontier}(D^{**})[S]$ (i.e., contradicting the fact that $S$ is a PT-constraint). Therefore, $N_i$ is $X_i$ when $N_i$ is not a leaf.

**Case 2b:** $N_i$ is a leaf. Notice that, since $N_i$ is a leaf and $L(N_i) \subset L(X_i)$, $N_i$ is a descendant of $X_i$. Furthermore, since $N$ is a weak R-node, no parent of $N_i$ in $D_i$ is a weak R-node (by the definition (3.1) of weak PR-trees). Thus, the parent $N^*$ of $N_i$ in $D_i$ which is either $X_i$ or a descendant of $X_i$, is a P-node. Therefore, there is a PR-tree $D^*$ of $D$ where $N_i$ is a middle child of $N^*$.

We now consider $H = \text{Frontier}(D^*) \left[ L(N^*) \cup \left( \bigcup_{j=0}^{\ell-1} L(N_j) \right) \right]$. Clearly, $H$ is connected and is not a path (since $N_i$ has degree 3 in $H$). Furthermore, $H$ is a subtree of $\text{Frontier}(D^*)[S]$ (since $V(H) \subseteq S$). This contradicts the fact that $S$ is a PT-constraint. Therefore, $N_i$ is $X_i$ when $N_i$ is a leaf.

Therefore the path $N_0, N_1, ..., N_{\ell-1}$ is really the path $X_0, X_1, ..., X_{\ell-1}$ and, as such, it is the LCA of $S$ and $S = \bigcup_{i=0}^{\ell-1} L(X_i)$ as needed. $\square$

We conclude this section with some interesting corollaries of theorem 3.21. In particular, this theorem provides some insights into the implied PT-constraints of a weak PR-tree (see corollary 3.22) and the structure of certain PT-constraints (see corollaries 3.23 and 3.24).

**Corollary 3.22.** For a weak PR-tree $D$, if $\text{Consistent}(D) \neq \emptyset$ (i.e., $T_{S_D} \neq \emptyset$), then, for every implied PT-constraint $S$ (i.e., $S \in \mathbb{PT}_{S_D} \setminus S_D$), there is a unique path $N_0, N_1, ..., N_{k-1}$ in the tree of a weak R-node $R$ such that $S = \bigcup_{i=0}^{k-1} L(N_i)$.

Note: this follows from theorem 3.21 and the definition of $S_D$.

**Corollary 3.23.** For a weak PR-tree $D$ with a P-node $P$, if $T_{S_D} \neq \emptyset$ and $S$ is a PT-
constraint of $D$ that contains leaves from at least two children $N, N^*$ of $P$ (i.e., $S \cap L(N) \neq \emptyset$ and $S \cap L(N^*) \neq \emptyset$), then $S$ contains all of $P$’s leaves (i.e., $S \supseteq L(P)$).

**Proof.** Since $S \in \mathbb{PT}_{SD}$, by theorem 3.21, $X = LCA(D, S)$ exists and $L(X) = S$. Thus, by definition (3.1) of weak PR-trees, $X$ is either $P$ or an ancestor of $P$. □

Corollary 3.23 also applies to the edges in the trees of the weak R-nodes of a weak PR-tree (as in corollary 3.24 below).

**Corollary 3.24.** For a weak PR-tree $D$ and an edge $N, N^*$ of the tree of a weak R-node of $D$, if $T_{SD} \neq \emptyset$ and $S$ is a PT-constraint of $D$ where $S \cap L(N) \neq \emptyset$ and $S \cap L(N^*) \neq \emptyset$, then $S \supseteq L(N) \cup L(N^*)$.

**Proof.** Since $S$ is a PT-constraint of $S_{SD}$, by theorem 3.21, $X = LCA(D, S)$ exists and $L(X) = S$. Thus, by definition (3.1) of weak PR-trees, $X$ is either a path that specifies the edge $N, N^*$ or an ancestor of an R'-node that specifies this edge. □

Some of the implied PT-constraints of a weak PR-tree (i.e., $S \in (\mathbb{PT}_{SD} \setminus S_D)$) are easy to see. In particular, every subpath of a path specified by an R'-node forms a PT-constraint (as in observation 3.25 below). In other words, if $X$ is a proper subpath of the path specified by an R'-node, $X$ is not the path specified by any R'-node, and $X$ consists of more than a single edge, then $L(X)$ is an implied PT-constraint.

**Observation 3.25.** If $D$ is a weak PR-tree with $R'$-node $Q$ and $\text{Children}(Q) = \{N_0, N_1, ..., N_{k-1}\}$ in this order, then, for every subsequence $i, ..., j$ $(i < j)$ of $0, ..., k - 1$, $\bigcup_{\ell=i}^{j} L(N_{\ell})$ is a PT-constraint of $D$.

However, there are also some more subtle implied PT-constraints. More specifically, consider two PT-constraints $S$ and $S^*$ whose LCAs are paths $X = N_0, N_1, ..., N_{k-1}$ and $X^* = N_0^*, N_1^*, ..., N_{k-1}^*$ (respectively) in the tree of a weak R-node $R$ in a weak PR-tree $D$. Now, suppose that $N_{k-1} = N_0^*$, $N_0$ is a leaf of $D$, and $N_{k-2} \neq N_1^*$. In this case, regardless of how the edges of $R_T$ are oriented, we can see that $\text{Frontier}(X) \cup$
"Frontier"($X^*$) is always a path. Additionally, if $N_{k-2} = N_0^*$ and $N_{k-1} = N_1^*$ (with no restriction on these nodes), then $Frontier(X) \cup Frontier(X^*)$ is similarly always a path. Thus, in both of these cases, $L(X) \cup L(X^*)$ is a PT-constraint. With careful examination, one can see that this is simply a special case of part (a) of observation 2.7.

In the next subsection we use these insights to formalize what it means for a weak PR-tree to represent a graph (see def. 3.27).

### 3.3.2 The Weak PR-tree Representing a Constraint Set

Using the tools from the previous subsection (3.3.1) we will now establish what it means for a weak PR-tree to represent a particular constraint set $S$ (or graph $G$ with $S_G = S$) – see theorem 3.26 and def. 3.27.

Notice that, much like constraint sets and the trees that satisfy them, weak PR-trees do not uniquely correspond with their consistent sets. In particular, consider a weak PR-tree $D$ with an implied PT-constraint $S$. By corollary 3.22, the LCA of $S$ is a path (with more than one edge) in the tree of a weak R-node $R$ but there is no $R'$-node that specifies precisely this path. Thus, the weak PR-tree $D^*$ created by adding such an $R'$-node to $D$ has exactly the same PT-constraints as $D$. Similarly, there could be $R'$-nodes in $D$ which are “redundant” (i.e., implied by other $R'$-nodes from the same weak R-node) and can be removed without affecting the PT-constraints of $D$. Thus, for a particular PT-constraint set $\mathbb{PT}$, there can be many weak PR-trees $D$ where $\mathbb{PT}_{S_D} = \mathbb{PT}$. To focus this choice, we present the following theorem (see theorem 3.26) which guarantees that a “representable” constraint set can always be represented by a weak PR-tree where the leaf sets of the P-node sources and $R'$-node sources are constraints in the given constraint set. We then use this theorem to establish what it means for a weak PR-tree to represent a constraint set $S$ or graph $G$ with $S_G = S$.

**Theorem 3.26.** For a weak PR-tree $D$ and a constraint set $S$ where $\mathcal{T}_S \neq \emptyset$, if $\mathbb{PT}_S = \mathbb{PT}_{S_D}$, then there is a weak PR-tree $D^*$ such that: $\mathbb{PT}_S = \mathbb{PT}_{S_{D^*}}$, and for each P-node
source or R'-node source \( N \) of \( D^* \), \( N \)'s leaf set is a constraint in \( S \) (i.e., \( L(N) \in S \)).

Proof. We will construct \( D^* \) from \( D \) in two steps. First, we will add certain R'-nodes (corresponding to elements in \( S \)) to \( D \) to create a weak PR-tree \( D^{**} \). We then remove source nodes (from \( D^{**} \)) which do not correspond to elements of \( S \) to create \( D^* \).

Part 1: Constructing \( D^{**} \) from \( D \).

Consider the LCAs (in \( D \)) of the constraints in \( S \). In particular, for each \( S \in S \), let \( X_S = LCA(D, S) \) (note: by theorem 3.21, \( X_S \) exists and has \( L(X_S) = S \)). If \( X_S \) is a P-node or a leaf of \( D \), then we know there is an explicit node (i.e., \( X_S \)) in \( D \) whose leaf set is \( S \) (i.e., we do not need to add anything to \( D \) for \( S \)). However, if \( X_S \) is a path in the tree of a weak R-node \( R \), then there is not necessarily an R'-node \( Q_S \) that specifies precisely this path. If there is no \( Q_S \), we create \( Q_S \) and add it to \( R \). Notice that, since \( S \) is a PT-constraint of \( D \), adding \( Q_S \) to \( D \) does not change the PT-constraints of \( D \). For convenience, we use \( X_S \) to denote the R'-node \( Q_S \). We refer to the weak PR-tree created by adding these R'-nodes to \( D \) as \( D^{**} \) (note: \( PT_{S,D^{**}} = PT_{S,D} = PT_S \)).

Part 2: Constructing \( D^* \) from \( D^{**} \).

We now demonstrate that the P-node (see 2a) and R'-node (see 2b) sources of \( D^{**} \) whose leaf sets are not in \( S \) can be removed without affecting \( D^{**} \)'s PT-constraints.

Part 2a: Consider a P-node source \( P \) of \( D^{**} \). Notice that, from corollary 3.23, \( L(P) \) is the only PT-constraint that connects (as in def. 2.6) the constraint sets of its children. In particular, if \( L(P) \notin S \), then \( S \) is not connected (i.e., \( PT_S \) is not connected). However, the constraint set (and thus the PT-constraint set) of every weak PR-tree is connected. Thus, for every P-node source \( P, L(P) \in S \) (otherwise, \( PT_{S,D^{**}} \neq PT_S \)).

Part 2b: We now consider an R'-node source \( Q \) of \( D^{**} \) and suppose that \( L(Q) \) is not in \( S \). We claim that each such \( Q \) can be removed from \( D^{**} \) and the result is a new weak PR-tree \( D^* \) with \( PT_{S,D^*} = PT_{S,D^{**}} \).

Notice that \( L(Q) \) is clearly an implied PT-constraint of \( S \) (i.e., \( L(Q) \) is an implied PT-constraint of \( S_{D^{**}} \setminus L(Q) \supset S \)). Thus, if every edge specified by \( Q \) is specified by
some other R'-node, we can remove \( Q \) from \( D^{**} \) to get a new weak PR-tree with the same PT-constraints as \( D^{**} \).

Suppose that some edge \( N, N^* \) is only specified by \( Q \). Notice that \( S^* = L(N) \cup L(N^*) \) is also a PT-constraint of \( D^{**} \) and, consequently, of \( S \). First, suppose that \( S^* \subseteq S \) for some \( S \in S \). Clearly, both \( N \) and \( N^* \) are descendants of \( X_S \). However, \( X_S \) is an explicit node and, since \( Q \) is a source, \( Q \) is the only node in \( D^{**} \) with both \( N \) and \( N^* \) as descendants. Thus, \( X_S = Q \) (contradicting the fact that \( L(Q) \notin S \)). Therefore, \( S^* \) is not a subset of any \( S \in S \). However, from corollary 3.24, we know that every PT-constraint which contains leaves from both \( N \) and \( N^* \) must contain \( S^* \). Thus, no constraint in \( S \) contains leaves from both \( N \) and \( N^* \). In particular, we can satisfy \( S \) without using any edges connecting elements of \( L(N) \) to elements of \( L(N^*) \) (i.e., \( S^* \) is not a PT-constraint of \( S \)). Therefore, every edge specified by \( Q \) is specified by some other R'-node.

Therefore, \( D^* \), obtained by removing the R'-node sources of \( D^{**} \) whose leaf sets are not constraints in \( S \), meets the required criteria.

**Definition 3.27.** A weak PR-tree \( D \) represents a constraint set \( S \) (or graph \( G \) with \( S = S_G \)) when:

(1) \( D \) and \( S \) have the same set of PT-constraints (i.e., \( PT_D = PT_S \)).

(2) For every P-node source or R'-node source \( N \) of \( D \), \( N \)'s leaf set is a constraint in \( S \).

(3) For every \( S \in S \), \( S \) is either the leaf set of a P-node or R'-node in \( D \).

Notice that def. 3.27 provides the following convenient relationship (see observation 3.28) between the vertices of a graph \( G \) and the source nodes of the weak PR-tree that represents \( G \). Furthermore, for a weak PR-tree \( D \) that represents a constraint set \( S \), we insist that the source nodes correspond to elements of \( S \). Thus, every constraint of \( D \) is a subset of some constraint in \( S \) (see observation 3.29). Also, by observation 3.7, when a graph is representable by a weak PR-tree, it is chordal (as in corollary 3.30 below).
Observation 3.28. For a weak PR-tree $D$ representing a graph $G$, every P-node source and $R'$-node source $N$ has a corresponding vertex $v \in V(G)$ such that $S_v = L(N)$.

Observation 3.29. For a constraint set $S$ and a weak PR-tree $D$ representing $S$, every constraint $S$ of $D$ is a subset of some constraint $S' \in S$ (i.e., for each $S \in S_D$, there is some $S' \in S$ such that $S \subseteq S'$).

Corollary 3.30. If a graph $G$ is representable by a weak PR-tree $D$, then $G$ is chordal.

Proof. This follows immediately from observation 3.7, part (3) of def. 3.27, and the fact that a graph is chordal iff it has a tree-tree model.

We conclude this section by demonstrating that, for a constraint set $S$ with $T_S \neq \emptyset$, the weak PR-tree that represents $S$ is unique (when it exists).

Theorem 3.31. For a constraint set $S$ such that $T_S \neq \emptyset$, if there is a weak PR-tree $D$ that represents $S$, then $D$ is unique.

Proof. Let $D$ and $D^*$ be weak PR-trees that represent $S$. First, suppose that $S_D = S_{D^*}$. Notice that, by observation 3.5, $D$ and $D^*$ can only differ by their $R'$-node sources. Thus, since they both represent $S$, $D = D^*$.

We now consider $S_{D^*} \neq S_D$; i.e., w.l.o.g., $S_D \setminus S_{D^*} \neq \emptyset$ and we let $S \in S_D \setminus S_{D^*}$ (note: $S$ is an implied PT-constraint of $S$ by def. 3.27; i.e., $S \in PT_S \setminus \emptyset$). By theorem 3.21, in $D^*$, there is a path $P = N_0^*, N_1^*, ..., N_{\ell^* - 1}$ in the tree of a weak R-node $R^*$ such that $S = \bigcup_{i=0}^{\ell^* - 1} L(N_i^*)$ and there is no $R'$-node in $D^*$ that specifies precisely this path. Since $S \in S_D$, we have the following two cases.

Case 1: There is either a P-node or $R'$-node $N$ in $D$ such that $L(N) = S$. Since $S \notin S$, such an $N$ cannot be a source node of $D$. Thus, $N$ has a parent $X$ in $D$. In particular, we need to be able to represent the corresponding constraint(s) of $X$ in $D^*$. However, since $D^*$ does not have an $R'$-node that specifies precisely the path $N_0^*, N_1^*, ..., N_{\ell^* - 1}$, this is not possible. Thus, in $D$, there is neither a P-node nor an $R'$-node whose leaf set is $S$ (i.e., we now have Case 2).
Case 2: There is an edge $N', N''$ in the tree of a weak R-node $R$ in $D$ such that $L(N') \cup L(N'') = S$ and there is no R'-node in $D$ that specifies precisely this edge. Recall that every edge in the tree of a weak R-node must be specified by some R'-node. Thus, we have an R'-node $Q \in R_{R'}$, such that $N', N''$ is an edge specified by $Q$. Furthermore, $N', N''$ is a strict subpath of the path specified by $Q$ (i.e., $\{N', N''\} \subset Children(Q)$) since no R'-node precisely specifies this edge.

Notice that, since $S \not\in S_{D^*}$, the path $N_0^*, N_1^*, ..., N_{\ell - 1}^*$ cannot be a single edge (i.e., $\ell > 2$). Now, let $S_i = L(N_i^*) \cup L(N_{i+1}^*)$ for each $i \in [0, \ell - 2]$ (note: each $S_i$ is a PT-constraint of $S$). By theorem 3.21, each $S_i^*$ must have an LCA $X_i$ in $D$ such that $L(X_i) = S_i^*$. However, since $\ell > 2$, every $S_i^*$ is a proper subset of $S$. Furthermore, for some $i \in [0, \ell - 2]$, $S_i^*$ necessarily has a non-empty intersection with both $L(N')$ and $L(N'')$. Thus, by corollary 3.24, $S_i^* = L(N') \cup L(N'') = S$; i.e., we have a contradiction.

3.4 The size of a weak PR-tree

In this subsection we consider the size of a weak PR-tree that represents a constraint set $S$ (or graph $G$ with $S = S_G$). In particular, in theorem 3.32 and corollary 3.34, we demonstrate that $D$ can be stored in a linear amount of space relative to the size of $S$ and $G$. We first prove this for constraint sets, then we use the fact that weak PR-trees only represent chordal graphs (as in corollary 3.30) to prove this for graphs.

**Theorem 3.32.** The size of a weak PR-tree representing a constraint set $S$ is $O(\sum_{S \in S} |S|)$.

**Proof.** We separate this into two parts. First, we consider the total space occupied by the P-nodes and R'-nodes in $D$. We then consider the total space of the weak R-nodes.

Notice that, since each P-node has at least three children and each R'-node has at least two children, the number of P-nodes and R'-nodes in the subtree rooted at a any node $N$ is at most the number of leaves of $N$. In particular, for each P-node source and R'-node source $N$, the total number of P-nodes, R'-nodes and leaves in the weak PR-tree
rooted at $N$ is at most $2|L(N)| - 1$. Thus, the total number of $Parent \to Child$ and $Child \to Parent$ pointers between these nodes is at most $4|L(N)|$. Furthermore, since $D$ represents $\mathcal{S}$, each such $N$ corresponds to a distinct $S \in \mathcal{S}$ such that $L(N) = S$. Therefore, we can represent all $P$-nodes, $R'$-nodes, and leaves (including the pointers between them) in less than $(\sum_{S \in \mathcal{S}} 6|S|)$ space (this excludes interactions with weak $R$-nodes).

We now consider the space occupied by weak $R$-nodes. Notice that, each weak $R$-node has a distinct set of $R'$-nodes and a distinct set of children. In particular, the number of weak $R$-node $\to Child$ pointers is bounded by the number of $R'$-node $\to Child$ pointers, and the number of $R'$-node to weak $R$-node connections is bounded by the number of $R'$-nodes. Furthermore, the size of the tree associated with a weak $R$-node is the number of children of that weak $R$-node. With this in mind we can see that each weak $R$-node $\mathcal{R}$ (including the pointers to/from $R'$-nodes in $\mathcal{R}_{R'}$ and the pointers to/from $\mathcal{R}$’s children) can be represented in less space than the $R'$-nodes in $\mathcal{R}_{R'}$ (i.e., all of $D$’s weak $R$-nodes can be represented in less space than all of $D$’s $R'$-nodes). Thus, $D$’s weak $R$-nodes can be represented in less than $(\sum_{S \in \mathcal{S}} 6|S|)$ space.

Therefore, $D$ can be stored in $O(\sum_{S \in \mathcal{S}} |S|)$ space. \hfill \Box

Notice that, the size of the set $\mathcal{S}_G$ is $\sum_{v \in V(G)} |S_v|$. In particular, this is the number of ones in the vertex to maximal clique incidence matrix of $G$. Furthermore, by the following theorem from [15] (see theorem 3.33 below), we can relate the size of a weak PR-tree to the size of any chordal graph it represents.

**Theorem 3.33.** [15] For a chordal graph $G$, the number of ones in the vertex to maximal clique incidence matrix is $O(n + m)$; i.e., $\sum_{S \in \mathcal{S}_G} |S| \in O(n + m)$.

Thus, by corollary 3.30 (i.e., stating that weak PR-trees can only represent chordal graphs) and theorems 3.32 and 3.33, we have the following corollary.

**Corollary 3.34.** The size of a weak PR-tree that represents a graph $G$ is $O(n + m)$. 
3.5 Making (strong) PR-trees from weak PR-trees

In this section we discuss how to convert weak PR-trees into PR-trees and strong PR-trees. Notice that, since P-nodes are handled easily (by simply choosing any permutation on their children), these operations amount to converting weak R-nodes into R-nodes and strong R-nodes. We will do this by using the algorithm from [4] as a black box (note: the results presented in this section other than remark 3.37 are not presented in [4]). In particular, in [4], they use an algorithm, which we denote as \textit{orient}(T,\{q_0, q_1, ..., q_{t-1}\}), to solve the following problem, which we refer to as the \textit{strong-orientation problem}.

Given a tree \(T\) and a collection of paths \(\{q_0, q_1, ..., q_{t-1}\}\) in \(T\), orient the edges of \(T\) so that \(q_i\) is a directed path for every \(i \in [0, t - 1]\).

We first describe how to use any algorithm that solves the strong-orientation problem to convert weak R-nodes into R-nodes and strong R-nodes (see section 3.5.1). We then analyse the timing and implications of using the \textit{orient} algorithm (see section 3.5.2).

3.5.1 Strong Orientation

In this section we prove that the strong-orientation problem is equivalent to both the problem of converting a weak R-node into a strong R-node and the problem of converting a weak R-node into an R-node. We prove this in the following two lemmas.

\textbf{Lemma 3.35.} The strong-orientation problem is equivalent to finding an orientation of a weak R-node which is a strong R-node.

\textit{Proof.} Given a weak R-node \(\mathcal{R}\), we solve the strong-orientation problem on the tree \(\mathcal{R}_T\) and the paths specified by the \(R'\)-nodes in \(\mathcal{R}_{R'}\). If no strong-orientation exists, then there is clearly no strong R-node. Similarly, if this strong-orientation problem has a solution, then by using this solution as the oriented version of \(\mathcal{R}_T\) and reversing the \(R'\)-nodes in \(\mathcal{R}_{R'}\) (as needed), we always get a strong R-node.
Similarly, for a given strong-orientation problem \((T, \{q_0, q_1, ..., q_{t-1}\})\), we form a weak R-node \(\mathcal{R}\) with \(\mathcal{R}_T = T\) and \(\mathcal{R}_{R'}\) as a set of R'-nodes that specify the paths given as input (note: for each edge of \(T\) that is not specified any path \(q_i\), we also add an R'-node that specifies this edge by itself). It is easy to see that \(\mathcal{R}\) has a strong R-node iff the given strong-orientation problem has a solution.

**Lemma 3.36.** The strong-orientation problem is equivalent to finding an orientation of a weak R-node which is an R-node.

**Proof.** For a given weak R-node \(\mathcal{R}\), we create a new weak R-node \(\mathcal{R}^*\) from \(\mathcal{R}\) with \(\mathcal{R}^*_T = \mathcal{R}_T\) such that an orientation of \(\mathcal{R}^*_T\) is a strong R-node iff the same orientation of \(\mathcal{R}_T\) is an R-node. We construct \(\mathcal{R}^*_{R'}\) from \(\mathcal{R}_{R'}\) as follows.

For each \(Q \in \mathcal{R}_{R'}\), let \(N_0, N_1, ..., N_{k-1}\) be the children of \(Q\) (in this order). Now, let \(N_{i_0}, ..., N_{i_{z-1}}\) be the children of \(Q\) that are leaves (i.e., neither P-nodes nor R'-nodes). Also, (w.l.o.g.) suppose \(i_j < i_{j+1}\) for each \(j \in [0, z - 2]\). Notice that, in every R-node of \(\mathcal{R}\), each subpath \((N_0, ..., N_{i_0}), (N_{i_0}, ..., N_{i_1}), ..., (N_{i_{z-2}}, ..., N_{i_{z-1}}), (N_{i_{z-1}}, ..., N_{k-1})\) of \((N_0, N_1, ..., N_{k-1})\) is a directed path (this follows from the definition (3.9) of R-nodes). Furthermore, if we orient \(\mathcal{R}_T\) so that, for every R'-node \(Q\), each such subpath is a directed path, then the result is an R-node. Thus, for each R'-node \(Q \in \mathcal{R}_{R'}\), we add R'-nodes \(Q_0, ..., Q_z\) to \(\mathcal{R}^*_{R'}\) where \(Q_0\) specifies the path \((N_0, ..., N_{i_0})\), \(Q_z\) specifies the path \((N_{i_{z-1}}, ..., N_{k-1})\), and, for each \(j \in [0, z - 2]\), \(Q_{j+1}\) specifies the path \((N_{i_j}, ..., N_{i_{j+1}})\) (note: if \(i_0 = 0\), we exclude \(Q_0\), and if \(i_{z-1} = k - 1\), we exclude \(Q_z\)). Thus, it is easy to see that \(\mathcal{R}^*\) has a strong R-node iff \(\mathcal{R}\) has an R-node. Therefore, by lemma 3.36, the strong-orientation problem consisting of \(\mathcal{R}^*_T\) together with the paths specified by \(\mathcal{R}^*_{R'}\) has a solution iff \(\mathcal{R}\) has an R-node.

Now, for a given strong-orientation problem \((T, \{q_0, q_1, ..., q_{t-1}\})\), we form a weak R-node \(\mathcal{R}\) with \(\mathcal{R}_T = T\) and \(\mathcal{R}_{R'}\) as a set of R'-nodes that specify the paths given as input (note: if there are any edges of \(T\) which are not specified some path \(q_i\), we also add an R'-node that specifies this edge by itself). Additionally, we make each child of \(\mathcal{R}\) into...
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

an R'-node with exactly two children (i.e., in any R-node \( R^* \) of \( R \), this will force each R'-node to specify a directed path in \( R^*_T \)). It is easy to see that \( R \) has an R-node iff the given strong-orientation problem has a solution.

We end this subsection with some complexity considerations of the above proofs.

Consider the instance \( I \) of the strong-orientation problem produced from a weak PR-tree \( R \) as in the proof of lemma 3.35 (i.e., when trying to produce a strong R-node). Clearly, the size of \( I \) is precisely the size of the \( R \) and is easily produced in linear time with respect to this size.

We now discuss \( R^* \) as constructed in the proof of lemma 3.36. Notice that \( R^* \) can be constructed in linear time with respect to the given \( R \). In particular, for each R'-node \( Q \) we are simply partitioning its specified edges into paths according to the children of \( Q \) which are leaves (i.e., this can be done in a single pass over \( Q \)'s children). Furthermore, \( R^* \) is at most twice as large as \( R \). In particular, for each child \( N \) of \( R \), \( N \) has at most twice as many R'-node parents in \( R^* \) as it did in \( R \). Additionally, each edge of \( R_T \) is specified by the same number of R'-nodes in \( R^*_R \) as it is in \( R^*_R \). Thus, the sum of the sizes of the paths specified by the R'-nodes in \( R^* \) is at most twice that of \( R \); i.e., \( R^* \) is at most twice the size of \( R \).

Thus, when trying to solve either weak R-node orientation problem, the corresponding strong-orientation problem can be constructed in linear time and occupies linear space with respect to the size of the given weak R-node.

3.5.2 Using the Algorithm from [4]

In this section we present the timing and implications of using the orient algorithm to convert weak R-nodes into R-nodes and strong R-nodes. In particular, we “lift” the properties of orient (as outlined in remark 3.37) to these weak R-node conversion methods (see theorems 3.38 and 3.39).
Remark 3.37. The characteristics of the orient algorithm [4].

(1) It completes in linear time with respect to the sum of the sizes of the paths given as input (i.e., $O(\sum_{i=0}^{n-1} |q_i|)$).

(2) When it succeeds, it orients $T$ such that each $q_i$ is a directed path.

(3) When it fails, it provides a certificate indicating that no orientation of $T$ satisfies these paths. This certificate (depicted in fig. 3.13) is an ordered odd sized subset $X = \{q_0^*, q_1^*, ..., q_{k-1}^*\}$ (i.e., $k$ is odd) of $\{q_0, q_1, ..., q_{n-1}\}$ such that:

(a) $T[\{u, u_0, ..., u_{k-1}\}]$ is a star (as seen in the centre of fig. 3.13).

(b) The paths in $X$ share a common vertex $u$ of $T$ (i.e., $\cap_{i=0}^{k-1} q_i^* = \{u\}$).

(c) For each $i \in [0, k-1]$, some neighbour $u_i$ of $u$ (in $T$) belongs to the paths $q_{i-1(mod k)}^*$ and $q_i^*$, but no other path in $X$.

Timing and Correctness

From lemmas 3.35 and 3.36 we know that we can use orient (together with appropriate transformations to convert a weak R-node into an input to the strong-orientation problem) to convert weak R-nodes into strong R-nodes or R-nodes as needed. For
convenience, we refer to the algorithm to orient a weak R-node $R$ into an R-node as $\text{fixRnode}(R)$ and the algorithm to orient $R$ into a strong R-node as $\text{strengthenRnode}(R)$.

Recall that, as discussed at the end of section 3.5.1, for a given weak R-node $R$, the related strong-orientation problem can be produced in $O(\sum_{Q \in R} |\text{Children}(Q)|)$ time and space. Thus, since the $\text{orient}$ algorithm runs in $O(\sum_{i=0}^{n-1} |q_i|)$ time where the $q_i$s are the input paths (as in (1) of remark 3.37 above), both $\text{fixRnode}(R)$ and $\text{strengthenRnode}(R)$ run in $O(\sum_{Q \in R} |\text{Children}(Q)|)$ time.

The Failure Certificate

We examine the failure certificate provided by the call to the $\text{orient}$ method in $\text{fixRnode}$ and $\text{strengthenRnode}$. In the following discussion we demonstrate that $G$ contains a false twin odd-sun when $D$ cannot be converted into a PR-tree and $G$ contains an odd-sun when $D$ cannot be converted into a strong PR-tree.

First, we consider a graph $G$ which is representable by a weak PR-tree $D$, but $D$ has no PR-trees. In particular, some weak R-node $R$ in $D$ has no R-nodes. We now examine the failure certificate (i.e., the odd sized ordered collection of paths $\{q_0^*, q_1^*, ..., q_{k-1}^*\}$) provided by the call to $\text{orient}$ on the strong-orientation problem related to $R$. Recall that (by def. 3.27), in $D$, for each vertex $v \in V(G)$, there is either a P-node or R'-node $N$ such that $L(N) = S_v$. In particular, $\text{LCA}(D, S_v)$ exists and has $S_v$ as its leaf set.

Notice that, each $q_i^*$ corresponds to a distinct R'-node $Q_i^*$ of $R^*$ as constructed in the proof of lemma 3.36. Furthermore, since the union of any pair $q_i^*, q_j^*$ ($i \neq j$) is not a path (as seen in fig. 3.13), each $q_i^*$ must correspond to a distinct R'-node $Q_i$ in $R$. Now, let $X_i$ be a source node which is either $Q_i$ or has $Q_i$ as a descendant. Notice that these $X_i$s are distinct by the definition of weak PR-trees. Furthermore, by observation 3.28, each $X_i$ has a corresponding vertex $w_i$ in $G$ such that $S_{w_i} = L(X_i)$ (note: these $w_i$s are also distinct and will form the centre clique of our false twin odd sun).

Also, the vertices $u, u_0, u_1, ..., u_{k-1}$ (of the $q_i^*$s) correspond to children $N, N_0, N_1, ...,$
$N_{k-1}$ of $\mathcal{R}$ such that $\mathcal{R}_T[\{N, N_0, N_1, ..., N_{k-1}\}]$ is a star. Furthermore, $N$ cannot be a leaf of $D$. In particular, we know that $u$ is an internal node of each $q_i^*$ (i.e., $N$ is an internal child of each $Q_i^*$). Thus, from our construction of $\mathcal{R}^*$, $N$ is not a leaf of $D$; i.e., $N$ has at least two leaves $C, C^*$ (recall that leaves of $D$ are maximal cliques of $G$). Notice that these maximal cliques contain $w_0, ..., w_{k-1}$. Clearly, we must also have distinct non-adjacent vertices $c \in C$ and $c^* \in C^*$. Notice that the least common ancestor of $S_c$ cannot be $N$ or have $N$ as a descendant (otherwise, $S_c$ would contain $C^*$). Similarly, $\text{LCA}(D, S_{c^*})$ cannot be $N$ or have $N$ as a descendant (otherwise, $S_{c^*}$ would contain $C$).

Now, for each $N_i$, let $C_i$ be a leaf of $N_i$ and let $c_i$ be a vertex in $C_i$ such that $c_i$ is not in $C$ (i.e., $c_i \in C_i \setminus C$). We claim that $\{c_0, c_1, ..., c_{k-1}\}$ is an independent set. Consider $Y_i = \text{LCA}(D, S_{c_i})$ and $Y_j = \text{LCA}(D, S_{c_j})$ ($i \neq j$). Notice that, the LCA of $\{C_i, C_j\}$ is the path $N_i, N, N_j$ in $\mathcal{R}_T$. In particular, if $c_i = c_j$ for some $i \neq j$, then $Y_i$ (or a descendant of $Y_i$) specifies a path that contains $N_i, N, N_j$ as a subpath (i.e., $C \in S_{c_i}$). Furthermore, the only way that $Y_i$ and $Y_j$ have common descendants is if (w.l.o.g.) $Y_i$ (or a descendant of $Y_i$) specifies the edge $N_i, N$ (since there is no other way to “connect” a leaf of $N_i$ to a leaf of $N_j$). However, we know this is not possible since $C$ is a leaf of $N$ and $C$ is not a leaf of $Y_i$. Thus, $c_i$ and $c_j$ are distinct and non-adjacent for all pairs $i, j$.

Similarly, each $c_i$ is not adjacent to either $c$ or $c^*$. In particular, $Y_i$ (or a descendant of $Y_i$) specifies the edge $N_i, N$ and, thus, falsely indicates that $c_i$ belongs to $C$. Thus, $\{c, c^*, c_0, c_1, ..., c_{k-1}\}$ is an independent set. Furthermore, since $X_i$ and $X_{i-1(\text{mod} \ k)}$ are the only $X$s with $N_i$ as a descendant, the only $w$s that are adjacent to $c_i$ are $w_i$ and $w_{i-1(\text{mod} \ k)}$. Therefore, since $k$ is odd, $G[\{w_0, w_1, ..., w_{k-1}, c_0, ..., c_{k-1}, c, c^*\}]$ is a false twin odd sun. Recall that the false twin odd sun is $F_{11}$ in the FISC of path graphs shown in fig. 2.4 in section 2.4 and thus certifies that $G$ is not a path graph.

Notice that, when $\text{strengthenNode}$ fails, we get nearly the same graph. The only difference is that the node $N$ (corresponding to $u$) could be a leaf of $D$, and, as such we do not necessarily have the vertices $c$ and $c^*$. However, we still have the odd sun
Chapter 3. Weak PR-trees, PR-trees, and Strong PR-trees

Given a weak PR-tree that represents a graph \( G \) we can determine whether or not \( G \) is either a path graph or directed path graph in \( \mathcal{O}(n + m) \) time (i.e., using \texttt{fixRnode} or \texttt{strengthenRnode} respectively). Furthermore, when \( G \) is not a path graph, this process identifies a false twin odd sun in \( G \) to certify that \( G \) is not a path graph. Similarly, when \( G \) is not a directed path graph, this process identifies an odd sun in \( G \) to certify that \( G \) is not a directed path graph. These results are summarized into the following theorems.

**Theorem 3.38.** For a weak PR-tree \( D \) and a graph \( G \) where \( D \) represents \( G \), applying \texttt{fixRnode} to each weak R-node in \( D \) has the following properties.

1. \texttt{fixRnode} completes in \( \mathcal{O}(\sum_{Q \in R}|\text{Children}(Q)|) \) time for each weak R-node \( R \) of \( D \), and thus \( \mathcal{O}(n + m) \) time for all weak R-nodes.
2. If \( D \) has a PR-tree (i.e., \( T_G \neq \emptyset \)), then one is produced.
3. If there are no PR-trees of \( D \), then a false twin odd sun in \( G \) is identified (i.e., certifying that \( T_G = \emptyset \) since false twin odd suns are not path graphs).

**Theorem 3.39.** For a weak PR-tree \( D \) and a graph \( G \) where \( D \) represents \( G \), applying \texttt{strengthenRnode} to each weak R-node in \( D \) has the following properties.

1. \texttt{strengthenRnode} completes in \( \mathcal{O}(\sum_{Q \in R}|\text{Children}(Q)|) \) time for each weak R-node \( R \) of \( D \), and thus \( \mathcal{O}(n + m) \) time for all weak R-nodes.
2. If \( D \) has a strong PR-tree (i.e., \( \overrightarrow{T}_G \neq \emptyset \)), then one is produced.
3. If there are no strong PR-trees of \( D \), then an odd sun in \( G \) is identified (i.e., certifying that \( \overrightarrow{T}_G = \emptyset \) since odd suns are not directed path graphs).
Chapter 4

Weak PR-tree Construction via Split Decomposition

Decompositions play an important role in graph theory.

The standard approach to designing efficient algorithms using decomposition techniques proceeds as follows. We recursively decompose the graph into smaller graphs until these small graphs do not decompose further. Such small graphs are called prime with respect to the given decomposition. We then solve the problem on the prime graphs and combine the solutions recursively until we have a solution for the original graph.

This chapter involves the split decomposition (also called join decomposition) in the context of weak PR-trees, PR-trees, and strong PR-trees. The split decomposition was first introduced by Cunningham and Edmonds [10, 9]. It has been used for circle graph recognition [21, 16, 39], and parity graph recognition [7, 13]. There are a few linear time algorithms for split decomposition. The first is due to Dahlhaus [13], the second due to Montgolfier et al. [6], and the most recent due to Gioan et al. [22]†.

In this chapter we will demonstrate how to construct a weak PR-tree representing

†The timing of this most recent algorithm is $O(n + m)A(n + m)$ where $A(.)$ is the single-parameter inverse Ackermann function.
graph $G$ via split decomposition (note: this will succeed as long as $G$ is representable by a weak PR-tree). In particular, we begin by recalling the definition of split decomposition as stated in [10] (see section 4.1). This is followed by the construction algorithm which is presented in three parts (namely, sections 4.2, 4.3, and 4.4). Section 4.2 consists of how to construct a weak PR-tree of a chordal graph from weak PR-trees of the split subgraphs of $G$ (i.e., the recursive case). In section 4.3 we discuss how to construct a weak PR-tree for the base cases of split decomposition. In section 4.4 we present an implementation (i.e., in $O(nm)$ time) of the weak PR-tree construction algorithm described in the previous two sections. We conclude this chapter with section 4.5 which summarizes the results presented in this chapter and their consequences.

## 4.1 Split Decomposition

This section recalls the original definitions from [10]. We also state some observations regarding splits in chordal graphs.

**Definition 4.1.** A split $(A, A', B', B)$ (depicted in fig. 4.1) of a connected graph $G = (V, E)$ consists of a bipartition $(A, B)$ of $V$, where $|A|, |B| > 1$, such that:

1. $A' \subseteq A$ and $B' \subseteq B$;

2. every vertex in $A'$ is universal to $B'$;

3. no other edges exist between vertices in $A$ and $B$.

![Figure 4.1: A split $(A, A', B', B)$](image)

A graph with no split is called *prime*. Notice that, a graph with 3 vertices or less is prime. A bipartition is *trivial* if one of its parts is the empty set or a singleton. A graph
is a star when it consists of an independent set together with a vertex which is universal to that independent set. Cliques and stars are called degenerate since every non-trivial bipartition of their vertices is a split. Degenerate graphs and prime graphs represent the base cases in the process defining split decomposition:

**Definition 4.2.** Split Decomposition is a recursive process which decomposes a given graph $G$ into a set of disjoint graphs $\{G_0, G_1, \ldots, G_{k-1}\}$, called split components, each of which is either prime or degenerate. There are two cases:

1. If $G$ is prime or degenerate, then return $G$.

2. If $G$ is neither prime nor degenerate, it contains a split $(A, A', B', B)$. The split decomposition of $G$ is then the union of the split decompositions of the graphs $G_A = G[A] + a$ and $G_B = G[B] + b$, where $a$ and $b$ are new vertices, called markers, such that $N_{G_A}(a) = A'$ and $N_{G_B}(b) = B'$.

Notice that during the split decomposition process, the marker vertices can be matched by so called split edges. Then given a split decomposition with the marker vertices and their matchings specified, the input graph $G$ can be reconstructed without ambiguity. The set of split edges defines the split decomposition tree whose nodes are the split components of the split decomposition. Cunningham showed that every graph has a canonical split decomposition tree [10].

We end this section by presenting three observations. The first two observations (see observations 4.3 and 4.4) concern splits in chordal graphs and we expect that these have been seen previously. The third one (see observation 4.6) concerns the graphs whose split components are path graphs. In particular, we observe that path graphs are not closed with respect to “undoing” splits. In other words, it is not sufficient to simply test whether the split components of a chordal graph are path graphs to recognize path graphs. Later in this chapter we will see that directed path graphs are indeed characterized as the chordal graphs whose split components are directed path graphs (see theorem 4.23).
Observation 4.3. If \((A, A', B', B)\) is a split in a chordal graph \(G\), then at least one of 
\(G[A']\) or \(G[B']\) is a clique.

Proof. Suppose neither \(G[A']\) nor \(G[B']\) are cliques. Now, there are \(a_1, a_2 \in A'\), \(b_1, b_2 \in B'\) such that \(a_1a_2 \notin E(G)\) and \(b_1b_2 \notin E(G)\). However, since \((A, A', B', B)\) is a split, we have \(a_1b_1, a_1a_2, a_2b_2, b_2a_1 \in E(G)\). Thus, \(a_1b_1a_2b_2\) is an induced four cycle in \(G\); i.e., \(G\) is not a chordal graph.

From here on we assume that \(G[A']\) is a clique for any split \((A, A', B', B)\) in a chordal graph \(G\). We now consider the structure of \(G[B]\). Notice that while \(G[B]\) can be disconnected, this disconnect implies the disconnection of \(G[B']\). A chordal graph with a split in which \(G[B]\) is disconnected is given in fig. 4.2. Notice that, adding an edge between the connected components of \(G[B]\) while keeping \(G[B']\) disconnected necessarily introduces an induced \(k\)-cycle (for some \(k \geq 4\)). For example adding the edge \(u, v\) to fig. 4.2) connects components 1 and 3 and creates an induced 4-cycle to \(G\).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure4.2}
\caption{A split \((A, A', B', B)\) in a chordal graph where \(G[B']\) is disconnected and \(G[B]\) consists of connected components 1, 2, and 3.}
\end{figure}

Observation 4.4. If \((A, A', B', B)\) is a split in a connected chordal graph \(G\), then \(G[B']\) is connected iff \(G[B]\) is connected.

Proof.

\(\implies\) Suppose \(G[B']\) is connected. Notice that, since \(G\) is connected, for all \(a \in A\) and \(b \in B \setminus B'\), we must have a path from \(a\) to \(b\) in \(G\). Furthermore, this path must use a vertex \(b' \in B'\) since \((A, A', B', B)\) is a split). Thus, \(G[B]\) is connected.
Suppose that \( G[B] \) is connected and \( G[B'] \) is not connected. Then, for some \( u, v \in B' \) where \( u \) and \( v \) are in different connected components of \( G[B'] \), there is a shortest path \( P = u, b_1, ..., b_k, v \in G[B] \) where \( b_1, ..., b_k \in B \setminus B' \). Notice that, for \( a' \in A', a'u, a'v \in E(G) \) and \( a'b_i \not\in E(G) \) for every \( i \in [1, k] \). Thus, \( P \) together with \( a' \) forms an induced cycle of length \( k + 3 \) in \( G \); i.e., we have a contradiction.

We generalize the previous observation in the following corollary.

**Corollary 4.5.** If \((A, A', B', B)\) is a split in a connected chordal graph \( G \), then \( G[B] \) consists of connected components \( G[B_1], ..., G[B_k] \) iff \( G[B'] \) consists of connected components \( G[B'_1], ..., G[B'_k] \) with \( B'_i \subseteq B_i \) and \( B'_i \neq \emptyset \) for every \( i \in [1, k] \) (see fig. 4.3).

![Figure 4.3: A visualization of corollary 4.5.](image)

**Observation 4.6.** If \( G \) is a chordal graph with a split \((A, A', B', B)\) such that \( G_A \) and \( G_B \) are path graphs, then \( G \) is not necessarily a path graph (since it could contain a false twin odd sun).

**Proof.** Recall that the false twin odd sun (i.e., \( F_{11} \) in the FISC of path graphs given in fig. 2.4 on pg. 20). Furthermore, as depicted in fig. 4.4, by setting:

- \( A = \) the odd sun, and \( A' = \) the centre clique of the odd sun; and
- \( B = B' = \) the “false twins”,

we have a split in a chordal graph which is not a path graph (i.e., any false twin odd sun) such that \( G_A \) and \( G_B \) are path graphs.
Figure 4.4: The false twin 3-sun \( G \) (left); \( G \) re-drawn to emphasize the split \((A, A', B', B)\) where \( A = \{v_0, v_1, v_2, q_0, q_1, q_2\} \) (i.e., the \( G[A] \) is the 3-sun), \( A' = \{q_0, q_1, q_2\} \) and \( B = B' = \{v_3, v_4\} \) (centre); and, \( G_A \) with marker vertex \( a \) (top-right) and \( G_B \) with marker vertex \( b \) (bottom-right). Notice that both \( G_A \) and \( G_B \) are path graphs, but \( G \) is not a path graph.

### 4.2 Joining PR-trees Across a Split

In this section, we discuss how to construct a weak PR-tree of a chordal graph \( G \) from weak PR-trees that represent the split subgraphs of \( G \) (i.e., \( G_A \) and \( G_B \) for a split \((A, A', B', B)\) respectively. We refer to this operation as a *join*. It is important to note that we consider an arbitrary split of \( G \) rather than one which arises specifically from \( G \)'s canonical split tree. We additionally consider using PR-trees or strong PR-trees that represent \( G_A \) and \( G_B \) (rather than weak PR-trees).

This section is outlined as follows. We first introduce some notation and the context of the join operation. This is followed by the mechanics of the join operation. We end the section by justifying the correctness of the join operation.

#### 4.2.1 Notation and Context

In this discussion, we use \( S^G_v \) to denote the set of all maximal cliques of \( G \) which contain the vertex \( v \) of \( G \) (i.e., \( G \)'s constraints). Similarly, we use \( \{S^A_v : v \in V(G_A)\} \) and \( \{S^B_v : v \in V(G_B)\} \) for \( G_A \)'s and \( G_B \)'s constraint sets respectively (e.g., for \( v \in V(G_A) \), \( S^A_v \) is the set of maximal cliques of \( G_A \) which contain \( v \)).

For a split \((A, A', B', B)\) in a chordal graph \( G \), we consider weak PR-trees \( D_A \) and \( D_B \) of \( G_A \) and \( G_B \) respectively (recall that \( G_A \) and \( G_B \) contain marker vertices \( a \) and \( b \))
respectively). Furthermore (by observation 4.3), w.l.o.g. suppose that $G[A']$ is a clique. Thus, $C_A = A' \cup \{a\}$ forms a maximal clique of $G_A$; i.e., $D_A$ has a leaf which corresponds to the maximal clique $C_A$. Notice that, since $D_B$ represents $G_B$, there is either a P-node or R'-node $N_B$ in $D_B$ such that $N_B = LCA(D_B, S_B^{b})$ and $L(N_B) = S_B^{b}$.

4.2.2 The Join Operation

We implement the join operation in two separate cases. The first case is when $S_B^{b}$ is a PT-constraint of $G$ and the second is when $S_B^{b}$ is not a PT-constraint of $G$. We end this section with a short comparison of performing the join operation when $D_A$ and $D_B$ are both: weak PR-trees, PR-trees, or strong PR-trees.

Case 1: $S_B^{b}$ is a PT-constraint of $G$

In this case we simply identify $C_A$ with $N_B$ and consider the result of this operation to be $D$ (i.e., we replace $C_A$ with $N_B$ in $D_A$). Notice that, when at most one of $C_A$ or $N_B$ has a weak R-node parent, $D$ is a weak PR-tree since we are essentially “joining” two weak PR-trees by replacing a leaf of one by a P-node or R'-node of the other. However, when both $C_A$ and $N_B$ have weak R-node parents $R_A$ and $R_B$ in their respective weak PR-trees, we need to merge these weak R-nodes. Fortunately, this merger is straightforward. We do this by identifying the vertex of $(R_A)_T$ corresponding to $C_A$ with the vertex of $(R_B)_T$ corresponding to $N_B$. The new weak R-node $R$ in $D$ is easily seen as having $R_{R'} = (R_A)_{R'} \cup (R_B)_{R'}$ and $R_T$ as the merger of $(R_A)_T$ and $(R_B)_T$ according to the identification of $C_A$ and $N_B$. Clearly, the constructed $R$ is a valid weak R-node; i.e., the resulting $D$ is a weak PR-tree.

It is important to note that starting with $D_A$ and $D_B$ as PR-trees (rather than weak PR-trees) does not guarantee that $D$ is a PR-tree. In particular, if $C_A$ has an R-node parent $R$ and $N_B$ is not a leaf of $D_B$, replacing $C_A$ with $N_B$ could result in $R$ losing its status as an R-node and instead being simply an oriented weak R-node (e.g., see...
However, it is clear that this is the only way that $D$ would not be a PR-tree (since $D$ is an oriented weak PR-tree). With this in mind, we can attempt to “fix” $R$ using the technique described in section 3.5. Recall that this method either makes $R$ into an R-node or indicates that no orientation of $R$ is an R-node (i.e., no graph that $D$ represents is a path graph).

![Diagram]

Figure 4.5: For $G$, $G_A$, and $G_B$ from fig. 4.4: A PR-tree $D_A$ of the weak PR-tree of $G_A$ (top-left); A PR-tree $D_B$ of the weak PR-tree of $G_B$ (bottom-left); and, the result $D$ of joining $D_A$ and $D_B$ with respect to the split $(A, A', B', B)$ from fig. 4.4 (right). Notice that $D$ is an oriented weak PR-tree, but is not a PR-tree. The maximal cliques $u_0, ..., u_4$ are defined as in fig. 3.5.

However, starting with $D_A$ and $D_B$ as strong PR-trees does result in $D$ being a strong PR-tree. In particular, the orientation condition required for strong R-nodes does not make any restrictions on the children of the underlying weak R-node $R$ (i.e., it only needs the edges of $R_T$ to be oriented so that $R$’s $R'$-nodes specify directed paths in $R_T$). More specifically, every strong R-node of $D_A$ and $D_B$ continues to be a strong R-node in $D$. This is easy to see since we have not changed the orientation of any edge of the trees of the strong R-nodes, we have not added new R’-nodes, and we have not altered any existing R’-nodes.

**Case 2: $S^B_b$ is not a PT-constraint of $G$**

Notice that, in the absence of $S^B_b$ as a PT-constraint of $G$, the operation from case 1 is clearly the wrong approach. In particular, it includes $N_B$ in $D$ which enforces the constraint $L(N_B) = S^B_b$ on the consistent set of $D$. With this in mind, we need a different approach.
We will see in section 4.2.3 that the absence of $S_b^B$ as a PT-constraint of $G$ corresponds to a very specific structure in both $D_A$ and $D_B$. In particular, when $G$ is a path graph, $S_b^B$ is not a PT-constraint of $G$ iff no vertex $v$ of $G$ has $S_v^G = S_b^B$ and the following conditions hold in $D_A$ and $D_B$.

**Condition 4.7.** In $D_A$, $C_A$ has exactly one parent $N$ and $N$ is either:

(a) a P-node; Or

(b) an R$'$-node with exactly two children and $C_A$ is a leaf of $(N_R)_T$.

**Condition 4.8.** In $D_B$, $N_B$ is a source node and $N_B$ is either:

(a) a P-node; Or

(b) an R$'$-node with exactly two children $N_1, N_2$, and no other R$'$-node of $N_B$’s weak R-node $(N_B)_R$ has both $N_1$ and $N_2$ as its children; i.e., $N_B$ is the only R$'$-node that specifies the edge $N_1, N_2$ of $((N_B)_R)_T$.

When conditions 4.7 and 4.8 are satisfied: we dispose of $N_B$ and $C_A$; then re-allocate $N_B$’s children as children of $N$; and update weak R-nodes as needed (described as follows). Notice that, if $N$ is an R$'$-node (as in condition 4.7(b) above), then we change $N$ into a P-node and update $N$’s weak R-node (i.e., we remove the edge $C_A, N'$ from $(N_R)_T$ and $N$ from $(N_R')_T$). Analogously, if $N_B$ is an R$'$-node (as in condition 4.8(b) above), then we break $(N_B)_R$ into two new weak R-nodes $\mathcal{R}_1$ and $\mathcal{R}_2$ as follows. Let $(\mathcal{R}_1)_T$ and $(\mathcal{R}_2)_T$ be the connected components of $((N_B)_R)_T$ resulting from removing the edge $N_1, N_2$ (where $N_1 \in (\mathcal{R}_1)_T$ and $N_2 \in (\mathcal{R}_2)_T$). Now, since $N_B$ is the only R$'$-node in $((N_B)_R')_T$ that specifies the edge $N_1, N_2$, $((N_B)_R')_T$ can be partitioned into $(\mathcal{R}_1)_R'$ and $(\mathcal{R}_2)_R'$ according to $(\mathcal{R}_1)_T$ and $(\mathcal{R}_2)_T$ respectively. Clearly, such $\mathcal{R}_1$ and $\mathcal{R}_2$ are weak R-nodes. This completes the description of how we construct $D$.

---

$^1$Regardless of whether $G$ is a path graph we will still use these conditions to decide on the operation performed by join (i.e. case 1 vs. case 2).

$^\S$Here we only consider P-node and R$'$-node parents of $C_A$. 
It is easy to see that $D$, as constructed by this operation, is a weak PR-tree. In particular, after disposing of $N_B$ (and breaking $(N_B)_R$ as needed), we will have one weak PR-tree for every child of $N_B$. We then remove $C_A$ from $N$ and attach each such weak PR-tree to $N$ (changing $N$ into a P-node as needed).

Notice that, when $D_A$ and $D_B$ are (strong) PR-trees, this operation preserves all (strong) R-nodes involved (i.e., the (strong) R-nodes in $D_A$ and $D_B$ remain as (strong) R-nodes $D$). In particular, we never replace a child of a weak R-node which is a leaf with a non-leaf. Thus, in this case, having $D_A$ and $D_B$ as (strong) PR-trees results in $D$ being a (strong) PR-tree. This completes Case 2.

**Joining: Weak PR-trees vs. PR-trees vs. Strong PR-trees**

Notice that starting with both $D_A$ and $D_B$ as weak PR-trees guarantees that the result of the join operation is a weak PR-tree. In particular, there is no “failure” condition. Similarly, when $D_A$ and $D_B$ are strong PR-trees, the result is guaranteed to be a strong PR-tree. However, this is not the case for PR-trees. More specifically, a join performed via case 1 does not necessarily result in a PR-tree when both $D_A$ and $D_B$ are PR-trees. We will revisit these properties in section 4.5.

### 4.2.3 The Correctness of the Join Operation

We now claim that the weak PR-tree $D$ (produced by our join operation) represents $G$. First, we relate the maximal cliques of $G$ to the maximal cliques of $G_A$ and $G_B$ (this will allow us to relate the constraints of $G$ with those of $G_A$ and $G_B$). We then discuss the structure of $D_A$ and $D_B$ in the presence and absence of $S^B_b$ as a PT-constraint of $G$. During this discussion we prove that $D$ represents $G$.

**Part 1: Maximal Cliques and Constraints in $G$, $G_A$, $G_B$.**

We begin our discussion of the maximal cliques and constraints with the following remark.
Remark 4.9. Each maximal clique $C$ of $G$ satisfies exactly one of the following:

1. $C \subseteq A$.
2. $C \subseteq A' \cup B'$.
3. $C \subseteq B$.

This follows from the fact that the only edges between $A$ and $B$ connect vertices in $A'$ to vertices in $B'$.

Notice that, via $b$, the maximal cliques of $G$ which contain vertices of $B$ (i.e., those satisfying (2) and (3) above) correspond directly with the maximal cliques of $G_B$. In particular, the maximal cliques of $G$ satisfying (3) are directly present in $G_B$. Furthermore, each maximal clique $C \subseteq A' \cup B'$ of $G$ has a corresponding maximal clique $C' = (C \setminus A') \cup \{b\}$ in $G_B$. Thus, the constraints of $G_B$ are precisely the constraints of $G$ for the vertices of $B$ and the additional constraint $S_b^B$; i.e., $\text{Consistent}(D_B)$ is the set of all path-tree models satisfying the constraints $S_B = \{S_u^G : u \in B\} \cup \{S_b^B\}$.

Additionally, the maximal cliques of $G$ which do not contain vertices of $B$ (i.e., those satisfying (1) above) correspond directly with the maximal cliques of $G_A$ that do not contain the vertex $a$. In particular, each constraint $S_v^A$ corresponds to the constraint $S_v^G$ for every $v \in A \setminus A'$. Similarly, for each $a' \in A'$, the constraints $S_a^A$ and $S_a^G$ are related as follows: $S_a^A = (S_{a'}^G \setminus S_b^B) \cup \{C_A\}$ and $S_a^G = (S_{a'}^A \setminus \{C_A\}) \cup S_b^B$. More generally, since the constraints of $G$ from vertices in $A$ either contain all of $S_b^B$ (i.e., vertices in $A'$) or none of $S_b^B$ (i.e., vertices in $A \setminus A'$), we have the following lemma.

Lemma 4.10. Every PT-constraint $S^A$ of $G_A$ which strictly contains $C_A$ has a corresponding PT-constraint $S^G$ of $G$ such that $S^G = (S^A \setminus \{C_A\}) \cup S_b^B$.

Part 2: The Constraint $S_b^B$.

Notice that, if $S_b^B$ is a PT-constraint of $G$, it is easy to see that $D$ formed by “joining” $D_A$ and $D_B$ as in Case 1 of section 4.2.2 indeed represents $G$.

Thus, it remains to show that our operation as described in Case 2 of section 4.2.2 (namely, when $S_b^B$ is not a PT-constraint of $G$) is correct. To do this we consider the
structure of $D_A$ and $D_B$ in the presence and absence of $S_b^B$ as a constraint of $G$.

Notice that, if $G$ contains a vertex $v$ whose constraint is precisely $S_b^B$, this vertex is either in $G_A$ as a vertex distinct from $a$ with constraint $S_v^A = \{C_A\}$ or in $G_B$ as a vertex distinct from $b$ with $S_v^B = S_b^B$. We will see in section 4.4.2 that such vertices are easily identified. Thus, we assume:

(*) no such vertices are present (i.e., the only way $S_b^B$ is a PT-constraint of $G$ is when it is an implied constraint).

We now show that, when $G$ is a path graph, $D_A$ and $D_B$ satisfy conditions 4.7 and 4.8 (from Case 2 of section 4.2.2) iff $S_b^B$ is not a PT-constraint of $G$. We separate this discussion into two parts (i.e., $D_A$ and $D_B$). We end our discussion of the correctness by justifying the re-allocation of the children of $N_B$ as children of $N$ (the parent of $C_A$).

**Part 2(a): Relating $D_A$ and $S_b^B$.**

We now consider how $S_b^B$ can be implied by constraints of $G$ from vertices in $A$. To do this we show that when $D_A$ fails condition 4.7, $S_b^B$ is a PT-constraint of $G$. At the same time, we show that when $D_A$ satisfies condition 4.7, $S_b^B$ is not implied by constraints of $G$ from vertices in $A$.

Notice that, if $C_A$ has no parents, then $D_A$ is precisely the leaf $C_A$ (i.e., $G_A$ is a clique). Thus, we have a vertex $v$ (distinct from $a$) in $G_A$ with $S_v^A = \{C_A\}$ (contradicting (*)).

Now, suppose that $C_A$ has a P-node parent $P$. We consider two cases.

(i) $C_A$ has a parent $N$ (distinct from $P$) which is either a P-node or R'-node.

By the definition (3.1) of weak PR-trees, $L(N) \cap L(P)$ is precisely $C_A$. Thus, by observation 2.8 and lemma 4.10 (from Part 1 above), $S_b^B$ is in $\mathbb{PT}_{s_G}$.

(ii) Otherwise (i.e., $P$ is the only parent of $C_A$).
By corollary 3.23, the PT-constraints of $D_A$ (i.e., the PT-constraints of $G_A$) which strictly contain \{CA\}, must also contain all of $L(P)$. Thus, by lemma 4.10 and since no vertex $v \in G$ has $S_v^G = S_b^B$ (by (\ast)), $S_b^B$ is not a PT-constraint of $G$ due to vertices from $A$.

We now suppose that $CA$ has a weak R-node parent $R$ and consider three cases.

\begin{enumerate}
  \item \textit{(i)} $CA$ is an internal node of $RT$.

  Consider neighbours $N_1$ and $N_2$ of $CA$ in $RT$. Recall that, the leaf set $L(N) \cup L(N^*)$ of each edge $N, N^*$ of $RT$ is a constraint of $D_A$ (i.e., a PT-constraint of $G_A$). Thus, $L(N_1) \cup \{CA\}$ and $L(N_2) \cup \{CA\}$ are PT-constraints of $G_A$ and, by lemma 4.10, $S_1^* = L(N_1) \cup S_b^B$ and $S_2^* = L(N_2) \cup S_b^B$ are PT-constraints of $G$. Now, by observation 2.8, $S_3^* = S_1^* \cap S_2^*$ is a PT-constraint of $G$. Notice that, $L(N_1) \cap L(N_2) = \emptyset$ since $N_1$ and $N_2$ are both children of $R$. Thus, $S_3^* = S_b^B$ is a PT-constraint of $G$.

  \item \textit{(ii)} $CA$ is a leaf of $RT$ and $CA$ has an R’-node parent $Q$ with at least three children.

  Suppose that (w.l.o.g.) $\text{Children}(Q) = \{CA, N_1, N_2, \ldots\}$ in this order. Now, by observation 3.25 and lemma 4.10, $S_1^* = S_b^B \cup L(N_1) \cup L(N_2)$, $S_2^* = S_b^B \cup L(N_1)$, and $S_3^* = L(N_1) \cup L(N_2)$ are PT-constraints of $G$. Notice that $L(N_1) = S_2^* \cap S_3^*$; i.e., $L(N_1)$ is also a PT-constraint of $G$ by observation 2.8. These constraints indicate that in any path-tree model $T$ of $G$: $T[S_1^*]$ is a path consisting of (w.l.o.g.) a subpath on $L(N_1)$ with a subpath on $S_b^B$ on its “left” and a subpath on $L(N_2)$ on its “right”; i.e., $S_b^B$ is a PT-constraint of $G$.

  \item \textit{(iii)} Otherwise (i.e., $CA$ is a leaf of $RT$, $CA$ has exactly one R’-node parent $Q$, and $Q$ has precisely two children – as in condition 4.8(b)).

  Notice that, from the perspective of PT-constraints, having such an R’-node parent $Q$ is the same as having a sole parent which is a P-node (if we were to allow P-nodes
to have two children). Thus, as in the case when $C_A$ has a sole parent which is a
P-node, $S^B_b$ is not a PT-constraint of $G$ due to vertices from $A$.

Thus, if $S^B_b$ is not a PT-constraint of $G$, $D_A$ satisfies condition 4.7. Furthermore, since
$G$ is a path graph, when $D_A$ satisfies condition 4.7 (and no vertex $v \in A$ has $S^G_v = S^B_b$),
the only way $S^B_b$ can be a PT-constraint of $G$ is when $S^B_b$ is a PT-constraint of $G$ due to
constraints of $B$ in addition to those of $A$.

Part 2(b): Relating $D_B$ and $S^B_b$.

We now suppose that $S^B_b$ is not implied by constraints of $G$ from $A$ and consider how $S^B_b$
can still be implied by constraints of $G$ (i.e., we focus on those from $B$). To do this we
relate the connectedness of $G[B]$ to $D_B$ and $S^B_b$.

We first consider the case when $G[B]$ is connected. Now, the constraint set $\{S^G_v : v \in B\}$ is connected. Consider $U_B = \bigcup_{v \in B} S^G_v$, and $S^G_{a'}$ for $a' \in A'$. Notice that,$S^G_{a'} \cap U_B = S^B_b$. Thus, by observation 2.8, $S^B_b$ is a PT-constraint of $G$. Furthermore,
by observation 4.4, $G[B]$ is connected iff $G[B']$ is connected. Therefore, when $G[B']$ is
connected $S^B_b$ is a PT-constraint of $G$.

We now consider the case when $G[B]$ is not connected. By corollary 4.5, the connected
components of $G[B]$ partition $S^B_b$ into disjoint sets $X_1, ..., X_k$ (one for each connected
component). In particular, the only PT-constraint of $G_B$ which contains elements from
more than one $X_i$, is $S^B_b$. Thus, $N_B$ is a P-node\footnote{When $k = 2$, $N_B$ is an R'-node with exactly two children.} with children $N_1, ..., N_k$ such that
$L(N_i) = X_i$ (i.e., each child of $N_B$ corresponds to a distinct connected component of
$G[B]$). Recall that, by observation 3.28, every source node $N$ corresponds to a vertex
$v$ of $G_B$ with $S^B_b = L(N)$ (i.e., $L(N) = S^G_v$). Thus, $N_B$ has no parents in $D_A$\footnote{Additionally, when $N_B$ is an R'-node, no other R'-node has both $N_1$ and $N_2$ as its children.}. In
particular, a source node $N$ with $L(N) \supset S^B_b$ would necessitate the existence of a vertex
$v$ universal to $B'$ in $G[B']$; i.e., $G[B']$ (and consequently $G[B]$) would be connected.
Thus, when $G[B]$ is not connected, $N_B$ is a source of $D_B$ which is either a P-node (with three or more children) or an R'-node (with exactly two children) which is not contained in any other R'-node. In particular, $S^B_b$ is not a PT-constraint of $G$ due to constraints from vertices in $B$ (since by (⋆) no $v \in B$ has $S^B_v = S^B_b$). Furthermore:

(1) Any PT-constraint of $G$ arising from $A$ is either disjoint from or strictly contains $S^B_b$.

(2) Any PT-constraint $S$ of $G$ arising from $B$ is either disjoint from $S^B_b$ or intersects with $S^B_b$ on exactly one $X_i$ (i.e., for some $i \in [1, k]$, $S \cap X_i \neq \emptyset$ and for every $j \in [1, k]$ with $j \neq i$, $S \cap X_j = \emptyset$).

Therefore, any PT-constraint of $G$ arising from combining constraints of (1) with constraints of (2) will be contained in some $X_i$. In particular, when $G$ is a path graph and $G[B]$ is not connected, $S^B_b$ is not a PT-constraint of $G$.

**Part 2(c): Re-allocating the children of $N_B$ as children of $N$.**

We now consider the case when $S^B_b$ is not a PT-constraint of $G$ (i.e., $G$ is a path graph, no vertex $v$ of $G$ has $S^G_v = S^B_b$, $D_A$ satisfies condition 4.7, and $D_B$ satisfies condition 4.8) and show that re-allocating the children of $N_B$ as children of $N$ is the correct join operation (i.e., the resulting $D$ does indeed represent $G$).

Notice that, as we have seen in part 2(b), the children of $N_B$ correspond to the connected components of $G[B]$. Furthermore, $X_1, ..., X_k$ partition $S^B_b$ according to these connected components. Thus, just as $S^B_b$ is a PT-constraint of $G$ when $G[B]$ is connected, each $X_i$ is a PT-constraint of $G$ when $G[B]$ is not connected. This follows by applying observation 2.8 with respect to $S^G_{a'}$ (for $a' \in A'$) and each connected component of $G[B]$.

Thus, by disposing of $N_B$ and re-allocating its children as children of $N$, we have joined $D_A$ and $D_B$ to create $D$ while eliminating the one superfluous constraint (namely, $S^B_b$) from $D_B$. In particular, $D$ represents $G$ (as needed). Furthermore, if this operation is performed when $G$ is not a path graph, $D$ will still represent $G$ (since the node we
removed did not correspond to a vertex of $G$).

With the above join operation in mind, it only remains to handle the “base cases” of the decomposition. In particular, in the following section we will consider the degenerate and prime path graphs and how to construct their weak PR-trees.

### 4.3 Prime Chordal Graphs and their Weak PR-trees

In this section we examine the weak PR-trees that represent degenerate and prime chordal graphs. We first observe that the weak PR-tree representing a degenerate chordal graph (namely, a clique or star) is either a single leaf or a single P-node. We then demonstrate that the weak PR-tree that represents a prime path graph is a single weak R-node.

The main idea of this section is that a graph that is represented by a weak PR-tree that is neither a single leaf, nor a single P-node, nor a single R-node is not degenerate and contains a split. With this in mind, first note the graphs which correspond to these “simple” weak PR-trees.

**Remark 4.11.** We note three special cases regarding a weak PR-tree $D$ that represents a chordal graph $G$:

1. $G$ is a clique iff $D$ is a leaf.

2. If $G$ is a star with $k \geq 3$ pendant vertices then $D$ is a P-node with $k$ children and these children are the leaves of $D$.

3. $G$ has a unique path-tree model $T$ iff $D$ is a weak R-node $\mathcal{R}$ where:

   - The children of $\mathcal{R}$ are the maximal cliques of $G$ (i.e., $\text{Children}(\mathcal{R}) = V(T)$).

   - For each vertex $v \in V(G)$ whose path $T[S_v]$ has at least two elements (i.e., $|S_v| \geq 2$), we have an $R'$-node $Q$ in $\mathcal{R}_{R'}$ such that $Q$’s children are the maximal cliques $S_v$ in the order specified by $T[S_v]$. 
• \( \mathcal{R}_T = T \).

Note: adding/removing true twins to/from the graph \( G \) does not change its weak PR-tree. In particular, when a \( D \) is a P-node, \( G \) is either a star or a star with true twins added.

We are now ready to prove the main result of this section; i.e., that every prime chordal graph which is representable by a weak PR-tree is a path graph with a unique path-tree model (see theorem 4.12).

**Theorem 4.12.** If \( G \) is a chordal graph which is degenerate or prime with respect to split decomposition and \( D \) is a weak PR-tree that represents \( G \), then \( D \) is precisely one of the following:

- A leaf (i.e., \( G \) is a clique).
- A P-node (i.e., \( G \) is a star).
- A weak R-node (i.e., \( G \) has a unique path-tree model).

Note: in every case, \( G \) is a path graph.

**Proof.** Suppose \( D \) is neither a leaf nor a P-node nor a weak R-node. Notice that, by remark 4.11, \( G \) is not degenerate (since \( D \) is neither a leaf nor a P-node). Thus, we now suppose that \( G \) is prime and demonstrate a contradiction (i.e., a split \((A, A', B', B)\) in \( G \)).

This proof proceeds in two parts. We will first show that the height of \( D \) is at most one (i.e., the children of every non-leaf node in \( D \) are leaves of \( D \)). This is followed by showing that each leaf of \( D \) either has a P-node parent and no other parents, or has no P-node parents.

**Part 1** Suppose \( D \) has a source node \( N \) with a child \( N^* \) which is not a leaf. Note: when \( N \) is an \( R' \)-node we consider \( N \) to be its weak R-node \( N_R \) instead. Recall that, by
observation 3.28, when we have a P-node or R′-node which is a source node, there must be at least one vertex whose constraint is precisely the leaf set of that node. We consider two cases for \( N \) in order to choose \( A' \).

(1) \( N \) is a P-node. Let \( A' \) be the set of vertices whose constraint is \( L(N) \). Notice that, \( A' \) is non-empty since \( N \) is a source node. Furthermore, for every maximal clique \( C \in L(N^*) \), \( A' \) is a subset of \( C \) and, consequently, a strict subset of \( C \) (since \( |L(N^*)| \ge 2 \)).

(2) \( N \) is a weak R-node. Notice that, we may have multiple R′-nodes \( Q_1, \ldots, Q_\ell \) which have \( N^* \) as a child. With this in mind, we choose \( A' = \{ v : S_v \supseteq L(Q_i) \text{ for some } i \in [1, \ell] \} \). Similarly to (1), \( A' \) is not empty since for every \( Q_i \), either \( Q_i \) itself is a source, or \( Q_i \) is descendent from a source node. Furthermore, for every maximal clique \( C \in L(N^*) \), \( A' \) is a strict subset of \( C \).

With \( A' \) chosen, we now let \( B' = \bigcup_{C \in L(N^*)} C \setminus A' \). Notice that our choice of \( A' \) and \( B' \) clearly implies that \( G[A'] \) is a clique and every \( a' \in A' \) is adjacent to every \( b' \in B' \).

We now claim that \( A' \) is an \((x, y)\)-separator in \( G \) for some \( x, y \in V(G) \).

Consider maximal cliques \( C_x \) and \( C_y \)†† where \( C_y \) is a leaf of \( N^* \) and \( C_x \) is a leaf of \( N \) but not a leaf of \( N^* \) (note: \( C_y \) clearly exists, and \( C_x \) exists since \( N \) must have a child distinct from \( N^* \)). Notice that, if a vertex \( v \) belongs to both \( C_x \) and \( C_y \), then \( v \in A' \) by our choice of \( A' \) and def. 3.1. Also, since \( C_y \in L(N^*) \), \( A' \) is a strict subset of \( C_y \). Thus, we let \( y \in C_y \setminus A' \) (note: \( y \notin C_x \) since \( y \notin A' \)). Similarly, \( C_x \setminus A' \neq \emptyset \) since \( A' \subseteq C_y \) and \( C_x \) is a maximal clique. Thus, we let \( x \in C_x \setminus A' \) (note: \( x \notin C_y \) since \( x \notin A' \)).

Now, suppose \( A' \) does not separate \( x \) from \( y \). Then, we have a path from \( x \) to \( y \) in \( G \) which avoids \( A' \). We now consider such a path (e.g., \( x, v_1, \ldots, v_k, y \)) and the sequence of nodes in \( D \) which correspond to these vertices (i.e., the nodes with leaf sets \( S_x, S_{v_1}, \ldots, S_{v_k}, S_y \)). Notice that, the leaf sets of these nodes must intersect in sequence.

††We will later choose vertices \( x \) and \( y \) from \( C_x \) and \( C_y \) respectively.
However, by avoiding the vertices of \( A' \), these nodes essentially belong to \( D \) after removing \( N \) (and \( N \)'s R'-nodes and their ancestors when \( N \) is a weak R-node). Furthermore, without \( N \) (and \( N \)'s R'-nodes and their ancestors when \( N \) is a weak R-node), \( D \) breaks into disjoint weak PR-trees which correspond to \( N \)'s children (by def. 3.1). In particular, \( C_x \) and \( C_y \) belong to disjoint weak PR-trees. Thus, there can be no such sequence of nodes in \( D \); i.e., \( A' \) is an \((x, y)\)-separator in \( G \).

Since \( A' \) is a separator, we set \( B \) to be the vertices of the connected component(s) of \( G \setminus A' \) which contain vertices from \( B' \), and we set \( A = V(G) \setminus B \). With this choice of \((A, A', B', B)\) it only remains to show that \(|A|, |B| > 1\) in order for this to be a split.

Notice that \(|A| > 1\) due to the existence of \( x \) and since \( A' \neq \emptyset \). Now, suppose \(|B'| = 1\), then \( G[B' \cup A'] \) is a clique, but \( G[B' \cup A'] \) is the union of more than one maximal clique (i.e., it must contain at least one pair of non-adjacent vertices). Thus \(|A|, |B| > 1\).

Therefore, \( D \)'s height is at most one.

**Part 2.** We now suppose that \( D \) has a leaf \( C \) which has a P-node parent \( N \), and some other parent \( N^* \). In this case, we will partition the maximal clique \( C \) into two parts to produce the split in \( G \). Notice that, every P-node and R'-node parent \( N' \) of \( C \) is a source of \( D \) (since \( D \) has height of at most one) and, as such, has at least one vertex \( v \) with \( L(N') = S_v \). With this in mind, we set \( A' = \{v : S_v = L(N)\} \) and \( B' = C \setminus A' \).

We once again claim that \( A' \) is an \((x, y)\)-separator in \( G \) for some \( x, y \in V(G) \). In particular, consider vertices \( x \) and \( y \) where \( y \) is in a maximal clique \( C_y \) of \( N^* \) and \( x \) is in a maximal clique \( C_x \) of \( N \) and neither \( C_x \) nor \( C_y \) are \( C \). Now, similarly to the previous argument, \( A' \) is an \((x, y)\)-separator by def. 3.1.

Since \( A' \) is a separator, we set \( B \) to be the vertices of the connected component(s) of \( G \setminus A' \) which contain vertices from \( B' \), and we set \( A = V(G) \setminus B \). Notice that, \( x \in A \setminus A' \) and \( y \in B \setminus B' \). Thus, with this choice of \((A, A', B', B)\) we have a split in \( G \).

Notice that, as a consequence of the above theorem and remark, the weak PR-tree of
a degenerate or prime chordal graph $G$ can be constructed as follows (this is formalized in algorithm 5 in section 4.4.2). When $G$ is degenerate, $G$'s weak PR-tree is either a leaf (when $G$ is a clique) or a P-node (when $G$ is a star). When $G$ is prime, we try to construct a path-tree model $T$ for $G$. If this succeeds, $T$ defines $G$'s weak PR-tree (i.e., $G$'s weak PR-tree is the weak R-node corresponding to $T$); otherwise, $G$ is not a path graph and has no weak PR-tree.

It is useful to note that the best known approach regarding the construction of a path-tree model of a prime chordal graph is to simply use the path graph recognition algorithm of Schäffer [38]. This algorithm runs in $O(nm)$ time (for any graph). However, it is possible that a careful examination of either Schäffer’s algorithm or our PRtreeReduction algorithm (as presented in chapter 5) could provide better time bounds in this restricted case (i.e., when the input is a prime graph).

On the other hand, it is clear that the construction of a weak PR-tree for the degenerate cases can be done in linear time.

Additionally, if we are interested in a PR-tree rather than a weak PR-tree, it is easy to see that any orientation of these simple weak PR-trees will suffice as a PR-tree. In particular, since the weak R-node $R$ produced in the prime case does not have any non-leaf children, every orientation of the edges of $R_T$ results in an R-node. However, if we are interested in a strong PR-tree, then we would need to “strengthen” the weak R-node (as in section 3.5) in the prime case to get a strong PR-tree. Alternatively, if we have a directed path-tree model of $G$, we can use it to form the strong R-node directly (however, the current best known algorithm to produce a directed path-tree model [4] first produces a path-tree model via Schäffer’s algorithm).
4.4 Implementation Details

In this section we present an efficient method to construct a weak PR-tree (when one exists) of a chordal graph from its split decomposition. In particular, we accomplish this using the machinery developed in sections 4.2 and 4.3 together with the notion of modified weak PR-trees (MwPR-trees) (introduced in this section). We first present the MwPR-tree, then we present the algorithm, and we end with a discussion of the runtime of this algorithm and the size of the weak PR-trees it produces.

4.4.1 Modified Weak PR-trees

The modified weak PR-tree is simply a weak PR-tree which has been augmented with additional information relating to a specific graph it represents. This form of augmentation has also been applied to PQ-trees. In particular, Korte & Möhring defined the modified PQ-tree (MPQ-tree) in [28] and, in [29], they used it to provide an incremental linear time PQ-tree construction algorithm.

Recall that, when we are joining weak PR-trees across a split we need a way to access the nodes which correspond to marker vertices (i.e., $C_A$ and $N_B$ as discussed in section 4.2.2). With this in mind, for a weak PR-tree $D$ representing a graph $G$, we augment $D$ to include information regarding the vertices of $G$. We refer to these augmented weak PR-trees as modified weak PR-trees (MwPR-trees). More specifically, if a weak PR-tree $D$ represents a graph $G$, the MwPR-tree $D^*$ of $D$ is simply $D$ together with information regarding the “location” of each vertex of $G$ in $D$. Recall that, by def. 3.27, for every $v \in V(G)$ there is an explicit P-node or $R'$-node $N$ in $D$ so that $L(N) = S_v$ (note: $N$ is also the LCA of $S_v$ in $D$). Thus, the “location” of each vertex of $G$ in $D$ is well-defined, and as such we can include this information in $D$ (as follows).

Definition 4.13. For a weak PR-tree $D$ representing a graph $G = (V, E)$, the modified weak PR-tree $D^*$ of $D$ is $D$ with the following additional information.
• For each vertex \( v \in V \), we store a pointer \( \text{LCA}_v \) to the LCA of \( S_v \) in \( D \).

• For each leaf, \( P \)-node, and \( R' \)-node \( N \) in \( D \), we store the set \( V_N \) of vertices \( v \in V \) with \( S_v = L(N) \). We also refer to \( V_N \) as the label of \( N \).

An example of a graph together with its weak \( PR \)-tree and \( MwPR \)-tree is given in fig. 4.6. Note: for convenience we refer to the vertices in the tree of a weak \( R \)-node \( R \) as the sections of \( R \) (i.e., \( V(R_T) \) is the set of sections of \( R \)). Similarly, for an \( R' \)-node \( Q \in R R' \), the sections of \( Q \) are the sections of \( R \) corresponding to the children of \( Q \).

![Figure 4.6: The false twin 3-sun \( G \) (top-left); The maximal cliques of \( G \) (top-right); A weak \( PR \)-tree \( D \) representing \( G \) (bottom-left); and, the modified version of \( D \) (bottom-right). Notice that we have written the label of each node adjacent to its name (e.g., \( V_{u_0} = \{v_0\} \), \( V_{Q_3} = \emptyset \), and \( V_{Q_2} = \{q_2\} \)).](image)

4.4.2 Constructing MPR-trees using Split Decomposition

In this section we present a complete algorithm to construct a weak \( PR \)-tree (when one exists) representing a graph \( G \) using split decomposition. In particular, we do this using the \( MwPR \)-tree together with the concepts presented in sections 4.2 and 4.3. The main idea of the algorithm is to first generate the split decomposition (i.e., as seen in algorithm...
3 below) using known techniques [13, 6, 22], then use the recursive structure to build the MwPR-tree (when one exists).

**Algorithm 3: SplitDecomp(G).** Producing the split decomposition of a graph.

```
pre : G is a graph.
post: Returns the split decomposition of G.
1 if G is prime or degenerate then return G
2 else // G is neither prime nor degenerate
3    Find a split (A, A', B', B) in G.
4    Let G_A = G[A] + a and G_B = G[B] + b, where a and b are the markers.
5    return (SplitDecomp(G_A), SplitDecomp(G_B), split edge (a, b)).
```

Our algorithm proceeds as follows. We first check whether the given graph $G$ is chordal (when this fails, the graph has no weak PR-tree by corollary 3.30). We then produce the split decomposition of $G$. By following this split decomposition bottom-up, we attempt to construct a weak PR-tree $D$ of $G$ (as in algorithm 4). For the base cases (i.e., the prime and degenerate split components), we directly use the technique as outlined in section 4.3 and formalized in algorithm 5 below. For the recursive case, we use the technique outlined in section 4.2 and formalized in algorithm 6 below. More specifically, for a chordal graph $G$, we return the joined weak PR-tree of the split decomposition of $G$; i.e., we return: $D = PRtreeFromSplit(SplitDecomp(G))$.

No further discussion is needed for algorithm 5 (i.e., all of the details have been previously covered in section 4.3).

In order to implement the join operation (as in section 4.2.2) we need to describe how to test whether or not $S^B_b$ is a PT-constraint of the graph $G$ using the MwPR-trees $D_A$ and $D_B$, and the split edge $(a, b)$ (this test is given as lines 2–6 of algorithm 6). Recall that (when $G$ is a path graph) $S^B_b$ is a not a PT-constraint of $G$ (the “pre-split” graph) iff all of the following are true:
Algorithm 4: \textit{PRtreeFromSplit}(SD). Producing a MwPR-tree representing a graph \(G\) from \(G\)'s split decomposition \(SD\).

\begin{verbatim}
pre : \(SD\) is the Split Decomposition of a graph \(G\).
post: Returns an MwPR-tree \(D\) representing \(G\) when one exists.
1 if \(SD\) is a single graph \(G\) then  // \(G\) is Prime or Degenerate
2     Set \(D = \text{PrimeOrDegeneratePRtree}(G)\). // See algorithm 5.
3 else // \(SD = (SD_A, SD_B, (a, b))\).
4     Set \(D_A = \text{PRtreeFromSplit}(SD_A)\) and \(D_B = \text{PRtreeFromSplit}(SD_B)\).
5     Set \(D = \text{Join}(D_A, a, D_B, b)\). // See algorithm 6.
6 return \(D\).
\end{verbatim}

1. No vertex \(v\) of \(G_A\) (other than \(a\)) has \(S^A_v = \{C_A\}\) (i.e., as in (*) in section 4.2.3 part (2)).

2. \(D_A\) satisfies condition 4.7.

3. No vertex \(v\) of \(G_B\) has \(S^B_v = S^B_b\) (i.e., as in (*) in section 4.2.3 part (2)).

4. \(D_B\) satisfies condition 4.8.

Notice that, the \(LCA_v\) pointers give us easy access to the \(C_A\) and \(N_B\) via \(LCA_a\) and \(LCA_b\) respectively.

With \(C_A\) in hand, it is easy to check (1) and (2). In particular, if \(G_A\) has a vertex \(v\) with \(S^A_v = \{C_A\}\), then \(v\) must belong to the label of \(C_A\) (i.e., \(v \in V_{C_A}\)). Thus, (1) is satisfied iff \(V_{C_A} = \{a\}\). Furthermore, for (2), we just check whether \(C_A\)'s parent(s) satisfy condition 4.7 (i.e, as in lines 2–3).

Also, with \(N_B\) in hand, we check (3) and (4) as follows. First, we check that \(N_B\) has no parents (as in line 4). Notice that, similarly to (1), if \(G_B\) has a vertex \(v\) with \(S^B_v = S^B_b\), then \(v \in V_{N_B}\). Thus, (3) is satisfied iff \(V_{N_B} = \{b\}\). At this point, if \(N_B\) is a P-node then (4) is satisfied, but if \(N_B\) is an R'-node we have a bit more work to do. In particular, \(N_B\) must have exactly two children \(N_1\) and \(N_2\) with respective sections \(X_1, X_2\) in \((N_B)_T\), and no R'-node (other than \(N_B\)) can specify the edge \(X_1, X_2\). Notice that,
Algorithm 5: PrimeOrDegeneratePRtree(G). Producing a MwPR-tree from a chordal graph G which is prime or degenerate with respect to split decomposition.

pre: G is a prime or degenerate chordal graph.
post: Return an MwPR-tree D representing G when one exists.

1 if G is a clique then
2 Set D as the MwPR-tree D consisting of a single leaf C with V_C = V(G).
3 for v ∈ V(G) do Set LCA_v = C.
4 else if G is a star with k ≥ 3 pendant vertices then
5 Let v be the centre of the G and u_1, ..., u_k be the pendant vertices.
6 Set D as the MwPR-tree D with leaves C_1, ..., C_k and a single P-node N
7 with Children(N) = {C_1, ..., C_k}.
8 Set V_N = {v} and LCA_v = N.
9 for i = 1, ..., k do Set V_{C_i} = {u_i} and LCA_{v_i} = C_i.
10 else // G is prime
11 Generate the path-tree model T of G.
12 if No path-tree model exists for G then
13 Output "No weak PR-tree exists for G" and EXIT.
14 Let X_1, ..., X_ℓ be the vertices of T.
15 Set D as the MwPR-tree D with a leaf C_j for each vertex X_j of T and a single weak R-node R with R_T = T (i.e., the sections of R are X_1, ..., X_ℓ), Children(R) = V(T), and R_{R'} constructed as follows.
16 Group V(G) into V_1, ..., V_p by true twins (i.e., if v ∈ V_i, then v’s true twins are also in V_i).
17 for i = 1...p do
18 Let X_{j_1}, ..., X_{j_k} be the path P_v for every v ∈ V_i.
19 if k == 1 then
20 Set V_{C_{j_1}} = V_i.
21 for v ∈ V_i do Set LCA_v = C_{j_1}.
22 else // k > 1
23 Create an R'-node Q ∈ R_{R'} with Children(Q) = {C_{j_1}, ..., C_{j_k}}.
24 Set V_Q = V_i.
25 for v ∈ V_i do Set LCA_v = Q.
26 return D

if an R'-node Q specifies the edge X_1, X_2, then Q ∈ Parents(N_1) ∩ Parents(N_2). Thus, for (4), we check that Parents(N_1) ∩ Parents(N_2) = {N_B} (as in line 5–6).

Using the above approach to testing for the constraint S^R_B together with the join operation (as described in section 4.2.2), we implement Join as in algorithm 6.
Algorithm 6: Join$(D_A, D_B, (a, b))$. Merging two MwPR-trees $D_A$ and $D_B$ representing graphs $G_A$ and $G_B$ (respectively) where $G_A$ and $G_B$ are the result of splitting a chordal graph $G$ according to the split edge $(a, b)$.

**pre** : $D_A$, $D_B$ are MwPR-trees representing $G_A$, $G_B$ (respectively) and $(a, b)$ is the split edge for a split $(A, A', B', B)$ where w.l.o.g. $G_A[A']$ is a clique.

**post**: Return an MwPR-tree $D$ representing $G$.

1. Let $C_A = LCA_a$ and $N_B = LCA_b$.
2. if $V_{C_A} == \{a\}$ and ($C_A$ has exactly one parent $N$) and
   3. ($N$ is a P-node) or ($N$ is an R'-node and $C_A$ is a leaf of $(N_R)_T$) and
   4. $|Parents(N_B)| == 0$ and $V_{N_B} == \{b\}$ and ($N_B$ is a P-node) or
   5. ($N_B$ is an R'-node with exactly two children $N_1, N_2$ and
   6. $Parents(N_1) \cap Parents(N_2) == \{N_B\}$) then
      // $S^B_b$ is not a PT-constraint of $G$.
   7. Let $N$ be the parent of $C_A$ in $D_A$.
   8. if $N$ is an R'-node then
      9. Remove the edge specified by $N$ from the tree of $N$’s weak R-node.
   10. Change $N$ into a P-node.
   11. Set $Children(N) = Children(N) \setminus \{C_A\} \cup Children(N_B)$.
   12. for $N^* \in Children(N_B)$ do Set $Parents(N^*) = Parents(N^*) \setminus \{N_B\} \cup \{N\}$.
   13. if $N_B$ is an R'-node then
      14. Let $X_1, X_2$ be the sections of $N_B$ and $N_1, N_2$ be the children of $N_B$.
      15. Remove the edge $X_1, X_2$ from $((N_B)_R)_T$.
      16. Remove $N_B$ from the parents of both $N_1$ and $N_2$.
      17. Break $(N_B)_R$ into new weak R-nodes according to the connected components of $((N_B)_R)_T$.
   18. else // $S^B_b$ is a constraint of $G$
      19. for $v \in V_{C_A} \setminus \{a\}$ do Set $LCA_v = N_B$
      20. Set $V_{N_B} = (V_{N_B} \cup V_{C_A}) \setminus \{a, b\}$
      21. Set $Parents(N_B) = Parents(N_B) \cup Parents(C_A)$
      22. for $N^* \in Parents(C_A)$ do Set
         $Children(N^*) = (Children(N^*) \setminus \{C_A\}) \cup \{N_B\}$
      23. if $C_A$ has a weak R-node parent $R_A$ and $N_B$ has a weak R-node parent $R_B$ then
         24. Let $T$ be the tree obtained by identifying the section of $(R_A)_T$
            corresponding to $C_A$ with the section of $(R_B)_T$ corresponding to $N_B$
         25. Create a weak R-node $R$ with $R_T = T$ and $R_{R'} = (R_A)_{R'} \cup (R_B)_{R'}$
         26. Create the MwPR-tree $D = D_A \cup D_B$ and the labels $V_N$ and LCA pointers $LCA_v$.
      27. return $D$
4.4.3 Runtime Complexity

In this section, we discuss the runtime complexity of the MwPR-tree construction algorithm as presented in section 4.4.2. We present this discussion in three parts. First, we present a time bound on the base cases (i.e., the creation of the MwPR-trees for the split components). In the second part, we discuss a time bound on the recursive cases (i.e., the Join operations). The final part presents the total complexity. Throughout this section we also consider the size of an MwPR-tree produced using this approach.

Notice that, in linear time, the split decomposition process provides us with the split components and split edges as in algorithm 3. In particular, the graphs embedded in this structure are the split components and the edges \((a, b)\) are the split edges. It is important to note that the “size” of the split decomposition (namely, the sum of the sizes of the split components together with the split edges) produced by the known split decomposition algorithms (e.g., [22]) is in fact linear (i.e., \(O(n + m)\)) with respect to the given graph.

The weak PR-trees of Split Components

We now examine the timing of generating a weak PR-tree (as in algorithm 5) for a prime or degenerate chordal graph and the size of such a weak PR-tree.

Notice that, the bottle-neck step in this operation is the generation of the unique path-tree model. In particular, if we could produce such a path-tree model in linear time, this algorithm would also run in linear time. This is clear due to the fact that for a given path-tree model \(T, \sum_{v \in V(G)} T[S_v] \in O(n + m)\) (by theorem 3.33 since \(G\) is chordal). Unfortunately, the best known algorithm to generate such a path-tree model is to use the general path graph recognition algorithm [38] which takes \(O(nm)\) time. With this in mind, we can see that generating a weak PR-tree (when one exists) for a prime or degenerate graph can be performed in \(O(nm)\) time.

We now consider the size of the weak PR-trees for these graphs. Notice that the size
of a weak PR-tree constructed from a degenerate graph is clearly $O(n)$. Additionally, the size of a weak PR-tree $D$ constructed from a prime graph $G = (V, E)$ is clearly $O(n + m)$ since $D$ is a weak R-node $R$ whose size is the same as $G$’s unique path-tree model. However, we can actually do better than $O(n + m)$. In particular, we can use the implicit approach to representing certain tree-tree models presented in [24] to reduce the space to $O(n)$. More specifically, by arbitrarily rooting $R_T$, each R’-node will either consist of two directed paths with the same starting point or one directed path. Thus, we can represent each R’-node by either two or three sections of $R_T$ (i.e., the starting point together with the end point(s)). Such a representation has size $O(n)$ since chordal graphs have at most $n$ maximal cliques. This implicit representation also allows each R’-node $Q$ to be explicitly determined without needing to consider sections of $R_T$ which do not belong to $Q$. We will revisit the fact that the weak PR-trees of degenerate and prime graphs have an implicit representation in $O(n)$ space when we discuss the total complexity of weak PR-tree construction using split decomposition at the end of this section.

The Runtime Complexity of Join

We now consider the timing of the Join operation as in algorithm 6 and the weak PR-trees it produces. For this discussion we consider $(A, A', B', B)$ to be the split being “joined” through the Join operation.

We start by discussing the timing of this operation. In particular, we claim that we can charge each join against the neighbourhoods of the marker vertices $a$ and $b$.

First, we consider the timing of testing whether $S^B_b$ is a PT-constraint of $G$ (i.e., as in lines 2–6). This only consists of one operation which requires non-constant time (i.e., testing $Parents(N_1) \cap Parents(N_2) = \{N_B\}$ on line 6). To account for this operation we notice that each parent of $N_1$ (respectively $N_2$) has a corresponding source node

\[\text{Note: we are not applying an orientation to } R_T \text{ here, this is strictly part of the underlying data structure.}\]
$N^*$ and each source node has at least one vertex $v$ with $S_v = L(N^*)$. Furthermore, $L(N^*) \supset L(N_1)$ and $L(N_1) \subset S^B_b$. Thus, $S_v \cap S^B_b \neq \emptyset$ (i.e., $v$ is adjacent to $b$). Therefore, the timing of this operation can be charged against the neighbourhood of $b$ (i.e., it is performed in $O(|B'|)$ time).

We now consider the modifications made to $D_A$ and $D_B$ in order to create the new MwPR-tree $D$. This consists of three parts; i.e., updating R-nodes (lines 17 and 25), updating “parent-child” relationships (lines 11–12 and 21–22), and updating labels and LCA pointers (lines 19–20).

Notice that, the only time we would need to access the set $R_{R'}$ of a weak R-node $R$ directly is during the “breaking” and “merging” of weak R-nodes. In particular, this means we do not need to maintain these sets. This is useful since the maintenance of these sets would be costly when we are “breaking” weak R-nodes (as in line 17) and when we are “merging” weak R-nodes (as in line 25). In fact, instead of keeping pointers from $R'$-nodes to their weak R-nodes, we simply connect $R'$-nodes directly to their sections in the tree of the corresponding weak R-node. Thus, the “breaking” and “merging” of weak R-nodes in our implementation are actually accomplished in constant time (through modifications to the trees of weak R-nodes).

We now consider updating the “parent-child” relationships (as in lines 11–12 and 21–22). For lines 11–12, we need to update the parent pointers for each child of $N_B$. This is easily charged against the neighbourhood of $b$ since each child of $b$ corresponds to a different maximal clique to which $b$ belongs. Similarly, for lines 21–22, each parent of $C_A$ corresponds to some source node $N^*$ with $C_A \in L(N^*)$. Thus, we can charge this updating of parent and child pointers against the neighbourhood of $a$ (i.e., this takes $O(|A'|)$ time).

Finally, we consider updating the labels and LCA pointers (as in lines 19–20). Clearly, these operations can be completed in $O(|V_{C_A}|)$ (i.e., $O(|A'|)$) steps.

From the above arguments we can see that $Join$ runs in $O(|A'| + |B'|)$ time. Thus,
since the known techniques for split decomposition produce a canonical split tree, all join
operations performed during \texttt{PRtreeFromSplit} will finish in a total of \(O(n + m)\) time
(disregarding the time used by calls to \texttt{PrimeOrDegeneratePRtree}).

We now consider the MwPR-tree \(D\) resulting from joining \(D_A\) and \(D_B\). Notice that
the \texttt{Join} operation does not create new P-nodes or R'-nodes. In fact, each call to \texttt{Join}
may even remove a P-node or R'-node from the current MwPR-trees (i.e., as in the case
when \(S_B^B\) is not a PT-constraint of \(G\)). Thus, the only place P-nodes and R'-nodes are
created is during the base cases.

\section*{Total Complexity}

To measure the total complexity we need to consider the following: checking for
chordality, producing the split decomposition, constructing MPR-trees for the split com-
ponents, and joining the split components. Notice that, the only one of these which takes
more than a linear amount of time with respect to the input graph is constructing the
MwPR-trees for the split components, which takes \(O(nm)\) time. Thus, this construction
algorithm completes in \(O(nm)\) time. Furthermore, any improvement on the \(O(nm)\) time
bound for constructing a path-tree model of a prime chordal graph translates to an iden-
tical improvement to our construction algorithm. We summarize this in the following
theorem.

\textbf{Theorem 4.14.} For a graph \(G\) with \(N\) vertices and \(m\) edges, a weak PR-tree \(D\) repre-
senting \(G\) (when one exists) can be constructed in \(O(\max\{n + m, f(n, m)\})\) time via split
decomposition (i.e., by setting \(D = \text{PRtreeFromSplit(SplitDecomp(G))}\)), where \(f(n, m)\)
is the amount of time required to construct a path-tree model (when one exists) from a
prime chordal graph \(G\). Currently, \(f(n, m) \in O(nm)\) by [38].

We now consider the size of a weak PR-tree constructed by this construction algo-

...
case. Furthermore, we observed that we can use an implicit representation to store each weak PR-tree for the base cases in terms of the number of vertices in the corresponding split component. Therefore, the size of the weak PR-tree we produce through this construction algorithm is $O(|V(G_1)| + |V(G_2)| + ... + |V(G_k)|)$ where $G_1, ..., G_k$ are the split components of $G$. In particular, this is $O(n + \#(\text{split edges in the split decomposition of } G))$. Furthermore, it is shown in [17] that the number of split edges in any split decomposition of $G$ is bounded by $n$. Thus, the weak PR-tree of $G$ can be represented in $O(n)$ space. This is summarized in the following theorem.

**Theorem 4.15.** For a graph $G$ that is representable by a weak PR-tree $D$, $D$ can be stored in $O(n)$ space.

### 4.5 Consequences of Split Decomposition

In this section we will summarize the results in this chapter and present some interesting consequences of them.

First, in theorem 4.14, for a chordal graph $G$, we have seen that $G$’s split decomposition can be used to construct a weak PR-tree that represents $G$ when $G$ is representable by a weak PR-tree. Additionally, this weak PR-tree can be constructed in $O(\max\{n + m, f(n, m)\})$ time where $f(n, m)$ is the amount of time required to construct a path-tree model (when one exists) from a prime chordal graph $G$. Currently, $f(n, m)$ is known to be $O(nm)$ by [38]. However, by taking a closer look at exactly where this algorithm can fail, we can characterize the graphs which are representable by weak PR-trees via their split decomposition. This is stated as the following theorem.

**Theorem 4.16.** A graph $G$ is representable by a weak PR-tree iff $G$ is chordal and each split component of $G$ is either degenerate or a path graph with a unique path-tree model.

**Proof.** This follows from corollary 3.30, remark 4.11, theorem 4.12, and the discussion of the correctness of the join operation. In particular, once we have weak PR-trees that
represent the split components, using the join operation guarantees us a weak PR-tree representing $G$.

Notice that, we have not yet shown that every path graph is representable by a weak PR-tree. It is easy to see that, by theorem 4.16, one way to do this would be to prove that every prime path graph has a unique path-tree model. For now, we leave this as an open question (and mention it again in the concluding remarks). We do this since we will show that every path graph is indeed representable by a weak PR-tree via the reduction algorithm in chapter 5 (see corollary 5.9). This corollary together with theorem 4.12 provides the following update to theorem 4.16.

**Theorem 4.17.** A graph $G$ is representable by a weak PR-tree iff $G$ is chordal and each split component of $G$ is a path graph.

*Proof.* This follows from corollaries 3.30 and 5.9, remark 4.11, and theorems 4.12 and 4.16.

Recall that, for a graph $G$, having a weak PR-tree $D$ that represents $G$ does not mean that there is necessarily a path-tree model of $G$ (i.e., $G$ is not necessarily a path graph). However, by theorem 3.38 in section 3.5, we can use $D$ to check whether such a path-tree model exists using the *fixRnode* method. In fact, in $O(n + m)$ time, we can either determine a path-tree model of $G$ or provide a certificate indicating that there is no path-tree model of $G$. Theorem 3.39 provides us with a similar result regarding directed path-tree models of $G$. Thus, we have the following theorems and corollaries.

**Theorem 4.18.** For a graph $G$ and $D = \text{PRtreeFromSplit(SplitDecomp}(G))$ (i.e., algorithms 3 and 4), applying *fixRnode* to each weak R-node in $D$ has the following properties:

1. The total time taken to test $G$ for chordality, construct $D$, and execute the calls to *fixRnode* is $O(\max\{n + m, f(n, m)\})$ where $f(n, m)$ is the amount of time required
to construct a path-tree model (when one exists) from a prime chordal graph \( G \).
Currently, \( f(n, m) \in \mathcal{O}(nm) \).

(2) If \( D \) is successfully constructed and \( D \) has a PR-tree (i.e., \( \mathcal{T}_G \neq \emptyset \)), then one is produced.

(3) If \( G \) is not a path graph then either \( D \) fails to be constructed (due to one of \( G \)'s split components not being a path graph) or a false twin odd sun in \( G \) is identified by one of the calls to \texttt{fixRnode} (i.e., certifying that \( \mathcal{T}_G = \emptyset \) since false twin odd suns are not path graphs).

I.e., the PR-tree together with this split decomposition algorithm solves the path-tree problem (see def. 2.2).

Proof. This follows from theorems 4.14, 4.16, and 3.38 and corollary 5.9.

Corollary 4.19. The path graph recognition problem can be solved in \( \mathcal{O}(\max\{n+m, f(n, m)\}) \) time using split decomposition and weak PR-trees where \( f(n, m) \) is the amount of time required to construct a path-tree model (when one exists) from a prime chordal graph \( G \).
Currently, \( f(n, m) \in \mathcal{O}(nm) \).

Proof. This follows from theorem 4.18.

Theorem 4.20. For a chordal graph \( G \) and \( D = \text{PRtreeFromSplit}(\text{SplitDecomp}(G)) \) (i.e., algorithms 3 and 4), applying \texttt{strengthenRnode} to each weak R-node in \( D \) has the following properties:

(1) The total time taken to test \( G \) for chordality, construct \( D \), and execute the calls to \texttt{strengthenRnode} is \( \mathcal{O}(\max\{n + m, f(n, m)\}) \) where \( f(n, m) \) is the amount of time required to construct a path-tree model (when one exists) from a prime chordal graph \( G \). Currently, \( f(n, m) \in \mathcal{O}(nm) \).

(2) If \( D \) is successfully constructed and \( D \) has a strong PR-tree (i.e., \( \mathcal{T}_G \neq \emptyset \)), then one is produced.
(3) If \( G \) is not a directed path graph, then either \( D \) fails to be constructed (due to one of \( G \)'s split components not being a path graph) or \( D \neq \emptyset \) and an odd sun in \( G \) is identified by one of the calls to \texttt{strengthenRnode} (i.e., certifying that \( \overrightarrow{T}_G = \emptyset \) since odd suns are not directed path graphs).

I.e., the strong PR-tree together with this split decomposition algorithm solves the directed path-tree problem (see def. 2.2).

\textbf{Proof.} This follows from theorems 4.14, 4.16, and 3.39, corollary 5.9. \hfill \Box

\textbf{Corollary 4.21.} The directed path graph recognition problem can be solved in \( O(\max\{n+m, f(n,m)\}) \) time using split decomposition and weak PR-trees where \( f(n,m) \) is the amount of time required to construct a path-tree model (when one exists) from a prime chordal graph \( G \). Currently, \( f(n,m) \in O(nm) \).

\textbf{Proof.} This follows from theorem 5.12. \hfill \Box

Notice that, these theorems together with the fact that every path graphs is representable by a weak PR-tree (as we will see later via corollary 5.9), provide the following characterization of path graphs via split decomposition.

\textbf{Theorem 4.22.} \( G \) is a path graph iff \( G \) is a chordal graph, \( G \)'s split components are path graphs, and \( G \) does not contain an induced false twin odd sun (i.e., \( F(4k) \) for any \( k \geq 2 \) as in fig. 4.7).

\textbf{Proof.} This follows from theorems 4.16, 4.18, and corollary 5.9. \hfill \Box

We can actually state a more interesting characterization of directed path graphs. Recall that, when we apply the join operation on two strong PR-trees, the result is always a strong PR-tree. Furthermore, if we have a directed path-tree model \( T \) of a prime chordal graph \( G \), then, as discussed at the end of section 4.3, \( T \) defines \( G \)'s strong PR-tree (which is in fact just a strong R-node). Thus, by implementing the base case
Figure 4.7: The false twin odd suns (i.e., $F_{11}$ from the FISC of path graphs given in fig. 2.4). Note: the vertices in the cycle of bold edges form a clique).

(i.e., $\text{PrimeOrDegeneratePRtree}$) of $\text{PRtreeFromSplit}$ to construct a directed path-tree model (rather than a path-tree model) in the prime case, we end up constructing a strong PR-tree (as long as the split components of the input graph are directed path graphs). In particular, we have the following result.

**Theorem 4.23.** $G$ is a directed path graph iff $G$ is chordal and $G$’s split components are directed path graphs.

**Proof.** This follows from theorem 4.16, corollary 5.9, and the paragraph prior to the statement of this theorem. □
Chapter 5

Weak PR-tree Construction via Reduction

The primary goal of this chapter is to prove that, for every constraint set $S$ (or graph $G$ with $S_G = S$), if $T_S \neq \emptyset$ (i.e., $G$ is a path graph), then there is a weak PR-tree that represents $S$ (i.e., demonstrating that PR-trees and strong PR-trees solve the path-tree and directed path-tree problems). We prove this constructively. In particular, we present an algorithm that uses the reduction approach (as in algorithm 1 from section 2.2.1) to construct the weak PR-tree $D$ that represents a given constraint set $S$ (or graph $G$ with $S_G = S$) when $T_S \neq \emptyset$.

The presentation of this algorithm is separated into four sections (and many technical details are given in the appendix). In section 5.1 we demonstrate that it is sufficient to consider connected graphs and provide a modified version of the general reduction approach to leverage this connectedness. In the next section we provide the weak PR-tree reduction algorithm using the connected reduction approach from the first section. At the end of the second section we will see that this algorithm finishes in $O(A(n+m)nm)\dagger$ time for a given graph. There is one key operation in our algorithm which is quite complicated.

\dagger$A(s)$ is the inverse of Ackermann’s function arising from Union-Find as presented in appendix A.3.1.
and, as such, we present it separately (see section 5.3). Many technical details of this key operation are deferred to the appendix. In the final section we will present some consequences of the results in this chapter.

When we are given a graph $G$, we will need to extract $S_G$. Recall that corollary 3.30 states that weak PR-trees can only represent chordal graphs. In particular, if $G$ is not a chordal graph, then we immediately know that there is no weak PR-tree that represents $G$. Furthermore, since path graphs are a subclass of chordal graphs, if $G$ is not chordal then $G$ is not a path graph (i.e., $T_G = \emptyset$). Thus, this is consistent with our claim that every path graph is representable by a weak PR-tree. Conveniently, testing for chordality can be performed in linear time (i.e., $O(n + m)$) by using lexicographic breadth first search (LexBFS) [37] and, also by using LexBFS, the maximal-clique vs. vertex incidence matrix of a chordal graph (i.e., $S_G$) can be determined in linear time. Thus, if we are given a graph $G$, we can determine $S_G$ in linear time.

In this chapter we will use the following notation regarding a constraint set $S$ or graph $G$ with $S = S_G$. As usual, $U_S$ is the union of the constraints in $S$ (i.e., $U_S = \bigcup_{S \in S} S$), and $T_S$ is the set of trees over $U_S$ that satisfy $S$ (i.e., $T_S = \{T : T$ is a tree, $V(T) = U_S$, and $T[S]$ is a path for every $S \in S\}$). Also, we set $S = S_G = \{S_0, S_1, ..., S_{n-1}\}$ where $S_i = S_{v_i}$ (i.e., $V(G) = \{v_0, v_1, ..., v_{n-1}\}$). Finally, for each $i \in [0, n - 1]$, we use $S_{i+1}$ to denote the first $i + 1$ constraints of $S$ (i.e., $S_{i+1} = \{S_0, ..., S_i\}$) and $S_i^*$ to denote the subset of $S_i$ contained in $U_{S_i}$ (i.e., $S_i^* = S_i \cap U_{S_i}$).

### 5.1 Connected Reduction

In this section we present a variation of the reduction method that follows a connected (as in def. 2.6) order on the constraints. In particular, we first observe that we only need to consider connected constraints sets and then we present this variation.

Notice that the path-tree models of a graph $G$ are characterized by the path-tree
models of the connected components of $G$. In particular, let $H_0, H_1, ..., H_{k-1}$ be connected components. Now, $T \in \mathcal{T}_G$ iff there are $T_0, T_1, ..., T_{k-1}$ where $T_i \in \mathcal{T}_{H_i}$ such that $T$ is obtained by adding edges to the forest $T_0, T_1, ..., T_{k-1}$. Furthermore, when $G$ is not connected, $\mathcal{T}_G$ is fully specified by its connected components (i.e., we only need to consider connected graphs). Equivalently, if we are given a set of constraints, we only need to consider the connected components of the intersection graph $\mathcal{I}(S)$ to capture $\mathcal{T}_S$.

Furthermore, it is easy to see that weak PR-trees can only represent connected graphs. In particular, to represent a disconnected graph, we would simply use one weak PR-tree for each of its connected components.

We now consider a connected graph $G$ (or constraint set $S$) and suppose that we have a connected traversal of $G$ (or $\mathcal{I}(S)$). In particular, the vertices of $G$ are ordered $v_0, v_1, ..., v_{n-1}$ so that for all $i \in [1, n]$ $G[\{v_0, v_1, ..., v_{i-1}\}]$ is connected (i.e., $\forall i \in [1, n], S_i$ is connected). Notice that, by observation 2.8, since $S_i$ and $S_{i+1}$ (i.e., $\{S_0, S_1, ..., S_{i-1}\}$ and $\{S_0, ..., S_i\}$) are both connected, $S^*_i = S_i \cap U_{S_i} \neq \emptyset$ and $S^*_i$ is a PT-constraint of $S_{i+1}$. In particular, every tree $T$ that satisfies $S_{i+1}$ contains a subtree $T^* = T[U_{S_i}]$ which satisfies $S_i$ and has $S^*_i$ as a path. Thus, if we have the set of trees that satisfies $S_i$, then we can reduce it by removing all trees where $S^*_i$ is not a path without losing any subtrees of the trees that satisfy $S_{i+1}$. Furthermore, for any tree $T^*$ satisfying $S_i \cup \{S^*_i\}$, suppose we attach the elements of $S_i \setminus S^*_i$ to the path $T^*[S^*_i]$ such that $S_i$ is a path and call this new tree $T$. Clearly, $T$ satisfies $S_{i+1}$. The approach outlined in this paragraph is formalized in observation 5.1 and as algorithm 7.

**Observation 5.1.** For every constraint set $S$ and constraint $S_i$, if $S$ and $S \cup \{S_i\}$ are connected (i.e., $S^* = U_S \cap S \neq \emptyset$), then: for any tree $T$ with $V(T) = U_{S \cup \{S_i\}}$,

$T \in \mathcal{T}_{S \cup \{S_i\}}$ iff $T[U_S] \in \mathcal{T}_S$, and $T[S^*]$ and $T[S]$ are paths.

**Note:** this justifies the invariant in algorithm 7 below.

‡Any generic graph search (e.g., breadth-first) of $G$ will provide such an ordering.
Algorithm 7: ConnReduction(S) : Solving a connected path-tree problem S via connected reduction.

pre : $S = \{S_0, S_1, ..., S_{n-1}\}$ is an ordered constraint set where $S_i$ is connected $\forall i \in [1, n]$.
post: $T_n = T_S$.

1. Initialize $U_0 = \emptyset$ and $T_0 = \emptyset$ // note: implicitly $S_0 = \emptyset$.
2. for $i = 0$ ... $n - 1$ do // invariant: $T_i = T_{S_i}$
3. | Set $S^*_i = U_i \cap S_i$ and $U_{i+1} = U_i \cup S_i$.
4. | $T^*_{i+1} = T_i \setminus \{T \in T_i : T[S^*_i] \text{ is not a path}\}$.
5. | $T_{i+1} = \{T : T \text{ is a tree, } V(T) = U_{i+1}, T[S_i] \text{ is a path and } T[U_i] \in T^*_{i+1}\}$.
6. return $T_n$.

5.2 Weak PR-tree Reduction

In this section we re-write ConnReduction (i.e., algorithm 7) using weak PR-trees in place of the collections of trees (see algorithm 8 below). In particular, in place of $T_i$ we will have a weak PR-tree $D_i$ that represents $S_i$ and in place of $T^*_i$ we will have a weak PR-tree $D^*_i$ that represents $S^*_i$. This requires operations corresponding to lines 4 and 5 that will produce an appropriate weak PR-tree. We also need a weak PR-tree from which to start our algorithm; i.e., as in line 1 (we present this after discussing lines 4 and 5).

With these weak PR-trees in mind, for line 4, we need an operation that takes $D_i$ and $S^*_i = S_i \cap U_{S_i} = S_i \cap L(D_i)$ (i.e., $S^*_i \subseteq L(D)$) as input, and (when possible) produces a weak PR-tree $D^*_{i+1}$ that represents $S^*_{i+1} = S_i \cup \{S^*_i\}$. Since $D^*_{i+1}$ represents $S^*_{i+1}$, there is either a P-node or R'-node $X_i$ such that $L(X_i) = S^*_i$. This node will be useful for our operation corresponding to line 5. Thus, when there is a weak PR-tree representing $S^*_{i+1}$,

\[
\text{reduce}(D_i, S^*_i) \text{ will return a weak PR-tree } D^*_{i+1} \text{ and } X_i \text{ where } D^*_{i+1} \text{ represents } S^*_{i+1} \text{ and } X_i \text{ is either a P-node or R'-node in } D^*_{i+1} \text{ with } L(X_i) = S^*_i.
\]

Note: when reduce fails to construct such a $D^*_{i+1}$, it will return $(0, 0)$ and guarantee that $T_{S_{i+1}} = \emptyset$. This method is quite complex and its details are discussed in section 5.3 and the appendix. From a high level, reduce will apply a sequence of local adjustments
to the given weak PR-tree to produce its result.

Now, for line 5, we need an operation that constructs a weak PR-tree $D_{i+1}$ that represents $S_{i+1}$. In particular, $D_{i+1}$ will be created from $D_i^*$, $X_i$, and $S_i$ where $D_i^*$ represents $S_i^*$ and $X_i$ is the P-node or R’-node in $D_i^*$ with $L(X_i) = S_i^* = S_i \cap L(D_i^*)$. We denote this operation as join($D_i^*$, $X_i$, $S_i^{**}$) for $S_i^{**} = S_i \setminus S_i^*$, and describe it as follows (note: this operation is depicted in fig. 5.1).

Consider the weak PR-tree $D_{i+1}$ formed by adding a P-node $P$ to $D_i^*$ where $Children(P) = S_i^{**} \cup \{X_i\}$. Notice that, $D_{i+1}$ represents $S_{i+1}$. Unfortunately, when $|S_i^{**}| = 1$ (i.e., $S_i^{**} = \{u\}$), this operation will not create a valid P-node (since P-nodes require at least three children). So, instead, we create an R’-node $Q$, where $Children(Q) = \{u, X_i\}$. Now, if $X_i$ has a weak R-node parent $R$, then we add $Q$ to $R_{R'}$ and the edge $(u, X_i)$ to $R_T$. Otherwise ($X_i$ does not have a weak R-node parent), we create a new weak R-node $R$ where $R_{R'} = \{Q\}$ and $R_T$ is the edge $(u, X_i)$. Thus, we can see that, when reduce produces the appropriate ($D_i^*, X_i$), join($D_i^*$, $X_i$, $S_i^{**}$) always produces a weak PR-tree that represents $S_{i+1}^*$. In particular, by theorem 3.31, $D_{i+1}$ is the unique weak PR-tree that represents $S_{i+1}$.

![Diagram](image)

Figure 5.1: Illustrating $D = \text{join}(D_{i+1}^*, X_i, S_i^{**})$ for a weak PR-tree $D^*$, a P-node or R’-node $X_i$ (where $L(X_i) = \{u_0, u_1, ..., u_{j-1}\}$), and a set $S^{**} = \{u_j, u_{j+1}, ..., u_{k-1}\}$, where $|S^{**}| > 1$ (left) and $|S^{**}| = 1$ (right).

We now return to what to use as our initial weak PR-tree. Consider the following. Pick any element $u$ of $S_0$ and create a weak PR-tree $D_0$ with $u$ as its only node. Notice that $S_0^* = \{u\}$ and reduce($D_0, \{u\}$) returns ($D_1^*, X_0$) = ($D_0, u$). We then apply join($D_1^*, X_0, S_0 \setminus \{u\}$). Clearly, this will result in the appropriate weak PR-tree $D_1$. 


With these operations we can now rewrite algorithm 7 as follows:

Algorithm 8: PRtreeReduction($S$): Constructing a weak PR-tree that represents $S$ via weak PR-tree reduction.

pre: $S = \{S_0, S_1, ..., S_{n-1}\}$ is an ordered constraint set where $S_i$ is connected $\forall i \in [1, n]$.
post: $D_n$ is a weak PR-tree that represents $S$ when one exists.

1. Let $u$ be an element in $S_0$.
2. Initialize $U_0 = \{u\}$, $D_0 = \{u\}$ // note: implicitly $S_0 = \{\{u\}\}$.
3. for $i = 0 ... n - 1$ do // invariant: $D_i$ represents $S_i$

4.   Set $S_i^* = U_i \cap S_i$ and $U_{i+1} = U_i \cup S_i$.
5.   $(D_{i+1}^*, X_i) = \text{reduce}(D_i, S_i^*)$.
6.   if $D_{i+1}^* = 0$ then return $\emptyset$. // $T_{S_{i+1}} = \emptyset$.
7.   $D_{i+1} = \text{join}(D_{i+1}^*, X_i, S_i \setminus S_i^*)$.
8. return $D_n$

We conclude this section by proving that PRtreeReduction is correct and considering the timing of PRtreeReduction assuming the correctness and timing of reduce. In particular, in section 5.3 (and the appendix), we will provide the reduce operation and demonstrate that (note: this is theorem 5.7):

(*) For a weak PR-tree $D$ representing a constraint set $S$ and $S^* \subseteq L(D)$, either:

• $\text{reduce}(D, S^*)$ produces a weak PR-tree representing $S \cup \{S^*\}$; or,
• $\text{reduce}(D, S^*)$ returns $(0, 0)$ and $T_{S \cup \{S^*\}} = \emptyset$.

Using this result, we will now prove that PRtreeReduction is correct. After this we provide a short discussion of the timing of the PRtreeReduction method by similarly “looking ahead” to the timing of the reduce operation.

Theorem 5.2. Assuming (*): For a constraint set $S$, PRtreeReduction($S$) either: produces a weak PR-tree that represents $S$; or, returns $\emptyset$ and $T_S = \emptyset$.

Proof. This proof proceeds inductively based on the size of $S$. Notice that when $|S| = 1$, PRtreeReduction($S$) appropriately produces $D_1$ that represents $S$ (as discussed above).
Thus, we assume that the first \( i < n \) iterations of the main loop produces a weak PR-tree \( D_i \) that represents \( S_i \) and consider the iteration \( i + 1 \). We consider two cases (i.e., by (*), \( \text{reduce}(D_i, S^*_i) \) either produces \( D^*_{i+1} \) representing \( S^*_{i+1} \) or returns \((0, 0)\)). Let \((D^*_{i+1}, X_i) = \text{reduce}(D_i, S^*_i)\).

**Case 1:** \( D^*_{i+1} \) represents \( S^*_{i+1} \). Notice that by using the \textit{join} operation (as discussed earlier in this section), we have \( D_{i+1} \) as needed.

**Case 2:** \( D^*_{i+1} = 0 \). Now, by (*), \( T_{S^*_{i+1}} = \emptyset \). Thus, by observation 5.1, \( T_S = \emptyset \) (since, in every path-tree model \( T \) that satisfies \( S \), \( T[U_{S_{i+1}}] \) would necessarily satisfy \( S_{i+1} \)).

We now consider the timing of \texttt{PRtreeReduction} (i.e., algorithm 8). In corollary A.5 (in section A.3.3 of the appendix), we will see that, during the execution of \texttt{PRtreeReduction}(\( S \)), the total time spent on the calls to the \texttt{reduce} method is \( O(\sum_{i=1}^{n-1} A(\sum_{j=0}^{i-1} |S_j|) \times \sum_{j=0}^{i-1} |S_j|) \) (for \( S = \{S_0, S_1, ..., S_{n-1}\} \)). Thus, it remains to consider the other operations performed during \texttt{PRtreeReduction}.

Notice that prior to and following the main loop of \texttt{PRtreeReduction} a constant amount of work is performed. Furthermore, in each iteration of the loop, four non-constant time operations are performed; i.e., a set intersection \((U_S \cap S_i)\), a set union \((U_S \cup S_i)\), \texttt{reduce}, and \texttt{join}. Clearly, the intersection and union can be performed in \( O(|S_i|) \) time. Furthermore, since \texttt{join} creates at most two new nodes each having at most \( |S_i| \) children, it can be performed in \( O(|S_i|) \) time. Therefore, the time used by the operations other than \texttt{reduce} is: \( O(\sum_{i=0}^{n-1} |S_i|) \). Notice that this is clearly dominated by the time spent during the calls to \texttt{reduce}. In particular, we have the following theorem describing the timing of \texttt{PRtreeReduction}(\( S \)).

**Theorem 5.3.** For a constraint set \( S \), \texttt{PRtreeReduction}(\( S \)) has an execution time of:

\[
O(\sum_{i=1}^{n-1} (A(\sum_{j=0}^{i-1} |S_j|) \times \sum_{j=0}^{i-1} |S_j|)).
\]

Moreover, when \( S = S_G \) for a chordal graph \( G \), by theorem 3.33, this simplifies to:

\[
O(n(A(n + m) \times (n + m))) = O(A(n + m)nm).
\]
Note: $A(s)$ is the inverse Ackermann function described in section A.3.1 of the appendix.

In this section we have seen the straightforward part of the weak PR-tree reduction process. In particular, it remains to present the key step; namely, the `reduce` operation.

## 5.3 The Reduce Operation

In this section we provide the details of the `reduce` operation. Recall that, for a given weak PR-tree $D$ representing a constraint set $S$ and $S^* \subseteq L(D)$, the primary goal of `reduce($D, S^*$)` is to produce a weak PR-tree, $D^*$ representing $S^* = S \cup \{S^*\}$ when $T_{S^*} \neq \emptyset$. Notice that, in such a $D^*$, there will either be a P-node or R'-node $X$ where $L(X) = S^*$. We will refer to $S, S^*, D, S^*, D^*$, and $X$ throughout this section. We call $D$ irreducible when we can guarantee that there is no tree $T \in Consistent(D) = T_S$ such that $T[S^*]$ is a path. Moreover, when $D$ is irreducible, $T_{S^*} = \emptyset$ by observation 5.1.

It is important to note that the `reduce` operation does not require that the given weak PR-tree $D$ has a non-empty consistent set nor does it guarantee that a weak PR-tree $D^*$ it produces will have a non-empty consistent set. In particular, even though `reduce` will return $(0, 0)$ when it can guarantee that $T_{S^*} = \emptyset$, it does not make any claims about $T_{S^*}$ when it produces a weak PR-tree that represents $S^*$.

Our discussion of `reduce($D, S^*$)` follows similarly to the algorithm for constructing a PQ-tree given in [2]. In particular, we implement the `reduce` operation through a bottom-up template matching approach on the nodes of $D$. A template consists of a pattern and a replacement represented in terms of a given constraint $S^*$. We have two types of templates: P-node and weak R-node. The weak R-node templates also make changes to the R'-nodes associated with the weak R-node being processed. Due to the lengthy case based nature of the templates, they are given in the appendix.

The application of a template to a node $N$ only affects $D$ locally (i.e., only $N$, $N$’s children, and the weak R-nodes related to $N$’s children are affected). We consider a node
to be *matched* after a template has been applied to it and we consider a P-node *eligible for matching* after all of its children have been matched. Similarly, we consider a node to be *finished* after all of its P-node parents have been matched, and we consider a weak R-node *eligible for matching* after all of its children are finished. This is how we control the traversal of $D$ in a bottom-up manner. In particular, after a node has been matched, its P-node parents will be matched prior to its weak R-node parent. It is important to note that the templates are formed so that, if an eligible node has no applicable template, $D$ will be irreducible and we will halt the *reduce* operation and return $(0, 0)$.

Notice that we will only need to modify certain nodes in $D$. In particular, a node with no leaves in $S^*$ should not be affected by the template matching process. More specifically, we really only need to consider the nodes which are in the LCSA of $S^*$ in $D$ and their descendants. We will refer to this “subtree” of $D$ as the *pertinent subtree of $D with respect to $S^*$* (and it is discussed in more detail in section 5.3.1). These nodes are the ones to which we will be applying templates.

This discussion of the *reduce* operation is separated into four parts. The first part (section 5.3.1) presents local properties of nodes which imply that $D$ is irreducible (the templates rely heavily on this discussion). The second part (section 5.3.2) consists of an overview of the P-node and weak R-node templates. The third part (section 5.3.3) provides an implementation of *reduce* (note the templates are presented in the appendix). We conclude by proving that *reduce* is correct (see section 5.3.4).

### 5.3.1 Irreducibility and Template Terminology

In this subsection we examine $D$ and its nodes with respect to $S^*$. During this examination we introduce most of the terminology that is used to describe the templates. We will also observe several properties each of which implies that $D$ is irreducible.

A leaf $u$ of $D$ is said to be *full* when $u \in S^*$; otherwise, $u$ is *empty*. A node $N$ is said to be *full* when all of its leaves are full and $N$ is said to be *empty* when all of its leaves are
empty. The \textit{pertinent subtree of }$D$\textit{ with respect to }$S^*$, denoted $\text{Pertinent}(D, S^*)$, is the subtree of minimum height whose frontier contains all of $S^*$ (i.e., the subtree “sourced” from the LCSA of $S^*$, see fig. 5.2). Notice that, (by corollary 3.20) the pertinent subtree is unique. A P-node or R’-node is said to be \textit{pertinent} when it belongs to the pertinent subtree and it has a full leaf (note: since LCSAs do not contain R’-nodes, every pertinent R’-node will have at least one parent in the pertinent subtree). A weak R-node $R$ is \textit{pertinent} when at least one of its R’-nodes is pertinent or when an element of the LCSA of $S^*$ is a subpath of $R_T$.

![Caption](image.jpg)

\text{Figure 5.2: \textit{Pertinent}(D, S^*), for } D \text{ from fig. 3.12 (on pg. 43) and } S^* = \{u_0, u_3, u_7\}. \text{ Note: LCSA}(D, S^*) = \{R_T[\{u_7, P_0\}], P_1\}.

Additionally, for a P-node or R’-node $N$, a parent $N^*$ of $N$ is a \textit{pertinent parent} when it is pertinent and is either a P-node or weak R-node (R’-nodes are never considered as pertinent parents). For example, in fig. 5.2, $P_1$ is the only pertinent parent of $u_3$ and $Q_2$, and $R$ is the only pertinent parent of $P_0$ and $u_7$. Furthermore, $N$ is said to be \textit{bridged} when it has exactly two pertinent parents. Recall that, by def. 3.1, at most one parent of a node can be a weak R-node. Thus, when $N$ is bridged it either has two pertinent P-node parents and we call $N$ $p$-bridged, or it has one pertinent P-node parent and one pertinent weak R-node parent and we call $N$ $r$-bridged. We focus on the case when $N$ has two or fewer pertinent parents since having three or more would indicate that $D$ is not reducible. In particular, each pertinent parent $N^*$ implies the existence of an ancestor $N^{**}$ of $N$ (i.e., $N^*$ or an ancestor of $N^*$) with a full leaf that is not a leaf of $N$ (i.e., $S^* \cap (L(N^{**}) \setminus L(N)) \neq \emptyset$). However, if there is a path-tree model $T$ that satisfies $S^*$, $P = T[L(N) \cap S^*]$ is a path and for some full leaf $u \in S^* \cap (L(N^{**}) \setminus L(N)) \neq \emptyset$, $u$ is
adjacent to one of the ends of $P$. Furthermore, by the definition of weak PR-trees, each pertinent parent corresponds to a distinct such full leaf. Thus, when $N$ has three (or more) pertinent parents there will be three such full leaves, and, consequently, there can be no path-tree model that satisfies $S^*$ (i.e., $D$ is irreducible). With this in mind, we use the term *non-bridged* to indicate having less than two pertinent parents.

A pertinent child $N$ of a weak R-node $R$ is considered *open* when it has no pertinent neighbours in $R_T$; $N$ is considered *one-sided* when it has one pertinent neighbour in $R_T$; $N$ is considered *closed* when it has two pertinent neighbours in $R_T$. We focus on the case when $N$ has two or fewer pertinent neighbours since having three or more would indicate that $D$ is not reducible. In particular, if $D$ has a weak R-node $R$ whose pertinent children do not induce a path in $R_T$ (i.e., the subgraph of $R_T$ induced by $R$’s pertinent children is disconnected or has a node whose degree is three or larger), $D$ is irreducible.

We have similar properties for a full R'-node $Q$ with respect to its “left-most” child $C$ and its “right-most” child $C^*$. In particular, $Q$ is said to be *accessible* when both $C$ and $C^*$ are one-sided. We treat this as the default case for a full R'-node (i.e., unless otherwise stated, a full R'-node is always accessible). Also, $Q$ is said to be *blocked* when (w.l.o.g.) $C$ is one-sided and $C^*$ is closed. Finally, when both $C$ and $C^*$ are closed, $Q$ is called *surrounded*.

Using the above terminology we observe several conditions (similar to those we have already seen) each of which implies that $D$ is irreducible:

1. A pertinent node with $\geq 3$ of any combination of the following: bridged children, blocked-full R'-node children, and pertinent parents.

2. An r-bridged node that is closed.

3. A surrounded R'-node with a pertinent parent.

Recall that the goal of the *reduce* method is to produce a weak PR-tree $D^*$ that represents $S^*$ when $T_{S^*}$ is not empty. Notice that, since such a $D^*$ represents $S^*$, there is
either a P-node or R′-node \( X \) in \( D^* \) such that \( L(X) = S^* \). In particular, the pertinent subtree of \( D^* \) with respect to \( S^* \) has a single “source” (i.e., \( X \)) and every pertinent node is full. Furthermore, there are no bridged nodes in \( D^* \) with respect to \( S^* \). To achieve these properties we prove the following lemma. This lemma is proved in two parts. First with respect to the P-node templates, then with respect to the weak R-node templates (these are proved in the appendix – see lemma A.1 and lemma A.2). Together, these two lemmas imply the Template Lemma.

**Lemma 5.4.** (Template Lemma) The following invariant is maintained throughout the template matching process:

**General Invariant:**

1. If an eligible node has no applicable template, \( D \) is irreducible (i.e., there is no \( T \in \mathcal{T}_{S_D} \) such that \( T[S^*] \) is a path).

2. Once a node \( N \) of \( D \) has been matched (i.e., a template has been applied to \( N \)), it will satisfy \( S^* \) (i.e., \( L(N) \cap S^* \) will have an LCA whose leaf set is precisely \( L(N) \cap S^* \)).

3. The constraints added to \( D \) by matching a node are PT-constraints of \( S^* \) (i.e., PT-constraints of \( S_D \cup \{S^*\} \)).

4. Matching a node \( N \) preserves the property that, for every P-node source or R′-node source \( N^* \) of \( D \), \( N^* \)’s leaf set is a constraint in \( S \), and, for every \( S \in S \), \( S \) is either the leaf set of a P-node or R′-node in \( D \).

5. After the pertinent P-node parents of a node \( N \) are matched, \( N \) will not be bridged.

6. After the pertinent parents of a node \( N \) are matched, \( N \) is pertinent iff it is full.

### 5.3.2 The Templates

Prior to beginning the template matching we traverse the tree bottom-up from the set \( S^* \) to locate its LCSA and to identify the pertinent nodes of \( D \). Additionally, we identify
whether each pertinent node is p-bridged or r-bridged, and open, one-sided, or closed (note: if we encounter a node with three or more pertinent parents or pertinent neighbours, we halt and indicate that $D$ is not reducible). Template matching is only performed on pertinent nodes.

There is only one template for the leaves (i.e., when a leaf is a member of $S^*$). This template does not change $D$, but matching a leaf does mark it as full.

The situation for non-leaf nodes is not so simple. For a non-leaf node $N$, we must ensure that $N$ satisfies $S^*$ (i.e., after $N$ is matched: $S^* \cap L(N) \in \mathbb{P}_S^D$) while providing $N$’s pertinent parents with the opportunity to satisfy $S^*$. Additionally, we must make sure that $N$’s children will not be bridged.

We now provide an overview of the P-node templates followed by an overview of the weak R-node templates (note: the weak R-node templates will affect $R$’s tree and $R$’s set of $R^*$-nodes).

The P-node templates are presented with respect to a pertinent P-node $P$ (see appendix A.1). These templates are presented in five groups. In the first group, $P$ is the sole parent of its children and only has full children. The second group expands on the first by allowing $P$ to have bridged-full children and empty children. In the third group, when $P$ has a pertinent parent, $P$ might be replaced by an $R^*$-node with some full and some empty children. We refer to such an $R^*$-node as partial. The third group further permits $P$ to have blocked-full $R^*$-node children. In the fourth group, $P$ is allowed partial children in addition to the possibilities of the third group. The final group completes the options for $P$’s children by including bridged-partial children. Since $P$ can have at most two pertinent parents, we further subdivide each of the latter four groups into three cases regarding the number of pertinent parents of $P$ (i.e., when $P$ has zero, exactly one, or exactly two pertinent parents).

After presenting these groups, we prove the Template Lemma (i.e., lemma 5.4) in the

\[\text{Notice that } \mathbb{P}_S^{D \cup \{S^*\}} = \mathbb{P}_S^{S \cup \{S^*\}} = \mathbb{P}_S, \text{ and, by observation 2.8, } S^* \cap L(N) \in \mathbb{P}_S^{D \cup \{S^*\}}.\]
context of P-nodes. In particular, we prove the following invariant:

**P-node Invariant:** (see lemma A.1 on pg. 147)

1. If an eligible P-node has no applicable template, $D$ is irreducible.

2. Once a P-node $P$ of $D$ has been matched, it will satisfy $S^*$. 

3. The constraints added to $D$ by matching a P-node are PT-constraints of $S^*$. 

4. Matching a P-node $P$ preserves the property that, for every P-node source or $R'$-node source $N$ of $D$, $N$’s leaf set is a constraint in $S$, and, for every $S \in S$, $S$ is either the leaf set of a P-node or $R'$-node in $D$. 

5. After the pertinent P-node parents of a node $N$ are matched, $N$ will not be bridged. 

6. After the pertinent P-node parents of a node $N$ are matched, $N$ is pertinent iff it is either full or has a pertinent weak R-node parent. 

Notice that, by (5) of the P-node Invariant, if we match all of a node’s pertinent P-node parents prior to matching its pertinent weak R-node parent we do not have to worry about bridged children when matching a weak R-node (i.e., slightly simplifying the templates required for weak R-nodes). This is precisely why our algorithm waits until the children of a weak R-node are finished before processing it. Furthermore, from the definition of weak PR-trees, insisting on a P-node before weak R-node priority when matching the pertinent parents of a bridged node is a valid approach.

We now provide an overview of the weak R-node templates (the templates themselves are presented in appendix A.2). Recall that the pertinent children of a weak R-node $R$ must induce a path in $R_T$ in order for $D$ to be reducible. We refer to this path as $R$’s pertinent path and use $R_{PP}$ to denote it.

The weak R-node templates are separated into five groups. The first group consists of the case when $R$’s pertinent path only consists of full children. In the next group we add the possibility that $R_{PP}$ contains blocked-full $R'$-nodes. In the third group, both ends of
are partial nodes and all of $\mathcal{R}_{PP}$’s internal nodes are full. The forth group considers the case when: one of end of $\mathcal{R}_{PP}$ is a partial node, the other end (when present) is either a full node or a blocked-full $R'$-node, and $\mathcal{R}_{PP}$’s internal nodes are full. The final group describes the case when $\mathcal{R}_{PP}$ has internal nodes which are not full with no restriction on the ends of $\mathcal{R}_{PP}$. Moreover, we separate each of these groups into subcases based on $\mathcal{R}$’s pertinent $R'$-nodes.

After presenting these groups, we prove the Template Lemma (lemma 5.4) in the context of weak R-nodes. In particular, we prove the following invariant:

**Weak R-node Invariant:** (see lemma A.2 on pg. 175)

1. If an eligible weak R-node has no applicable template, $D$ is irreducible.
2. Once a weak R-node $\mathcal{R}$ of $D$ has been matched, it will satisfy $S^*$.
3. The constraints added to $D$ by matching a weak R-node are PT-constraints of $S^*$.
4. Matching a weak R-node $\mathcal{R}$ preserves the property that, for every P-node source or $R'$-node source $N$ of $D$, $N$’s leaf set is a constraint in $S$, and, for every $S \in S$, $S$ is either the leaf set of a P-node or $R'$-node in $D$.
5. When matching a weak R-node $\mathcal{R}$, each pertinent child $N$ of $\mathcal{R}$ (prior to matching) remains pertinent iff it is full.

Notice that the Template Lemma (see lemma 5.4) follows from the combination of the P-node and weak R-node invariants.

### 5.3.3 Finalizing the Reduce Operation

In this subsection we present an implementation of the *reduce* operation. This consists of three parts. First, we discuss the process in which we identify the pertinent subtree (i.e., how to locate the LCSA of $S^*$ in $D$ and mark the pertinent nodes with their relevant properties). We then present the final version of the *reduce* operation. We
finish this subsection by discussing how to locate the LCA of \( S^* \) once the template matching has finished. The timing of these operations is discussed in the appendix (see section A.3.3). In particular, we will see that \( \text{reduce}(D, S^*) \) can be performed in \( O(A(\sum_{j=0}^{i-1}|S_j|) * (\sum_{j=0}^{i-1}|S_j|)^{\frac{1}{2}}) \) time where \( D \) represents \( S_i = \{S_0, S_1, ..., S_{i-1}\} \).

To identify the pertinent subtree we use an operation called \( \text{label}(D, S^*) \). In particular, this operation identifies the LCSA of \( S^* \) in \( D \), the pertinent nodes of \( D \), and whether each pertinent node is:

- Open, one-sided, or closed.
- Accessible, blocked, or surrounded.
- An element of the LCSA(\( D, S^* \)), p-bridged, r-bridged, or otherwise (i.e., has a single pertinent parent).

This method first traverses \( D \) from \( S^* \) (i.e., bottom-up) marking each node it encounters visited (i.e., every node in \( D \) with a leaf in \( S^* \) becomes visited). It also keeps track of the number of visited children of every visited node. Then, it traverses \( D \) from \( D \)'s visited sources (i.e., top-down). On the way down the “highest” P-nodes and weak R-nodes with multiple visited children form the LCSA of \( S^* \) in \( D \) and are the first nodes to be identified as pertinent (and are marked as such). Continuing down \( D \) it records the appropriate properties of each pertinent node. Notice that, this method runs in time proportional to the number of visited nodes (i.e., the number of nodes with leaves in \( S^* \)).

We now present the \( \text{reduce} \) operation (see algorithm 9). Recall that a node \( N \) is: matched once a template has been applied to it, and finished after all of its pertinent P-node parents have been matched (i.e., when \( N \) is not bridged). In this sense we do not enqueue a P-node until all of its pertinent children are matched and we do not enqueue a weak R-node until all of its pertinent children are finished. The correctness of the template matching is discussed briefly in section 5.3.4 and in detail in the appendix.

\( A(s) \) is the inverse Ackermann function (as described in appendix A.3.1).
Algorithm 9: \texttt{reduce}(D, S^*) : Reducing a weak PR-tree by a given constraint.
This method proceeds similarly to \textit{S-reductions} from [2].

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{pre} : $D$ is a weak PR-tree representing a constraint set $S$, and $S^* \subseteq L(D)$.
\State \textbf{post} : $D^*$ is a weak PR-tree representing $S \cup \{S^*\}$ and $X = LCA(D^*, S^*)$.
\State \textbf{label}$(D, S^*)$
\State Initialize $Q$ as a queue containing the elements of $S^*$.
\While{$Q$ is not empty}
\State Dequeue the first element $N$ from $Q$.
\If{a P-node or leaf template applies to $N$}
\State Substitute the replacement for the pattern in $D$.
\State Set $\text{newlyMatched} = \{N\}$.
\State Set $\text{newlyFinished}$ as the children of $N$ that are no longer bridged.
\ElseIf{a weak R-node template applies to $N$}
\State Substitute the replacement for the pattern in $D$.
\State Set $\text{newlyMatched}$ to be the pertinent $R'$-nodes in $N_{R'}$.
\ElseReturn $(0,0)$ ; \\ \text{ // } S_D \text{ does not satisfy } S^*.$
\EndIf
\EndIf
\For{each node $N^*$ in $\text{newlyMatched}$}
\State Mark $N^*$ as matched.
\If{$N^*$ is not bridged} Add $N^*$ to $\text{newlyFinished}$.
\For{each pertinent P-node parent $Y$ of $N^*$}
\If{Y’s pertinent children are all matched} Add $Y$ to $Q$.
\EndFor
\EndFor
\For{each node $N^*$ in $\text{newlyFinished}$}
\State Mark $N^*$ as finished.
\If{$N^*$ has a pertinent weak R-node parent $\mathcal{R}$ and every pertinent child of $\mathcal{R}$ is finished} Add $\mathcal{R}$ to $Q$.
\EndIf
\EndFor
\State $D^* = D$.
\If{$|LCSA(D^*, S^*)| > 1$} \Return $(0,0)$.
\EndIf
\State $X = LCA(D^*, S^*)$
\If{$X$ is a path in a weak R-node $\mathcal{R}$} \Return $(D^*, X)$.
\EndIf
\EndWhile
\Endalgorithm
\end{algorithm}

We conclude this subsection by discussing how to locate the LCA of $S^*$ after the template matching has completed successfully (i.e., how to determine $LCA(D^*, S^*)$). Notice that, through the template matching process some of the pertinent nodes will lose their pertinent status. In particular, it is easy for us to keep track of which nodes remain
pertinent as we apply templates. Thus, we can simply traverse $D^*$ from $S^*$ (i.e., bottom-up) along the pertinent nodes in order to locate $S^*$’s LCSA. Additionally, by the Template Lemma (see lemma 5.4), only full nodes will remain pertinent after the template matching has finished. Furthermore, no node in $D^*$ will be bridged with respect to $S^*$. Thus, for any pair $X, X^*$ of pertinent nodes with no pertinent parents (i.e., $X, X^* \in LCSA(D^*, S^*)$), $X$ and $X^*$ will have no leaves in common (i.e., $L(X) \cap L(X^*) = \emptyset$). In particular, the elements of the LCSA of $S^*$ will partition $S^*$ into disjoint subsets (i.e., each subset will be the leaf set of an element of the LCSA). More specifically, there is no node $N$ in $D^*$ whose leaf set intersects the leaf sets of both $X$ and $X^*$. Therefore, in any tree $T$ where $T[S]$ is connected for each $S \in S_{D^*}$, $T[S^*]$ is disconnected. With this in mind, we will halt unsuccessfully when the LCSA of $S^*$ in $D^*$ has more than one element. Additionally, when the LCSA has exactly one element we will set $X$ to be that element and return it along with $D^*$. This leads to the following two observations which will be useful for our discussion of the correctness of \textit{reduce} in section 5.3.4.

\textbf{Observation 5.5.} For a weak PR-tree $D$ and $S^* \subseteq L(D)$, if the template matching portion of the \textit{reduce} completes successfully (i.e., converting $D$ to $D^*$) but $|LCSA(D^*, S^*)| > 1$, then $T_{S_{D^*}} = \emptyset$ (i.e., $T_{S^*} = \emptyset$ since $PT_{S_{D^*}} \subseteq PT_{S^*}$).

\textbf{Observation 5.6.} For a weak PR-tree $D$ and $S^* \subseteq L(D)$, if $(D^*, X) = \text{reduce}(D, S^*)$ where $D^* \neq 0$, then $X = LCA(D^*, S^*)$, $L(X) = S^*$, and $X$ is either a P-node or $R'$-node.

\section{The Correctness of \textit{reduce}}

In this section we demonstrate the correctness of the \textit{reduce} method (see theorem 5.7 below). Recall that, in theorem 5.2, we demonstrated that \textit{PRtreeReduction} (see algorithm 8) is correct under the assumption that \textit{reduce} is correct. We will see that the correctness of \textit{reduce} follows easily from the Template Lemma (lemma 5.4) and observations 5.5 and 5.6. For convenience, we recall the parts of the general invariant of
the Template Lemma which are used in the proof below.

**General Invariant:**

1. If an eligible node has no applicable template, $D$ is irreducible (i.e., there is no $T \in \mathcal{T}_S$ such that $T[S^*]$ is a path).

2. Once a node $N$ of $D$ has been matched (i.e., a template has been applied to $N$), it will satisfy $S^*$ (i.e., $L(N) \cap S^*$ will have an LCA whose leaf set is precisely $L(N) \cap S^*$).

3. The constraints added to $D$ by matching a node are PT-constraints of $S^*$.

4. Matching a node $N$ preserves the property that, for every P-node source or R$'$-node source $N^*$ of $D$, $N^*$’s leaf set is a constraint in $S$, and, for every $S \in S$, $S$ is either the leaf set of a P-node or R$'$-node in $D$.

**Theorem 5.7.** For a given weak PR-tree $D$ that represents a constraint set $S$ and $S^* \subseteq L(D)$, either $\text{reduce}(D, S^*)$ produces a weak PR-tree representing $S^* = S \cup \{S^*\}$, or: $\text{reduce}(D, S^*)$ returns $(0, 0)$ and $\mathcal{T}_{S \cup \{S^*\}} = \emptyset$.

**Proof.** We consider the two cases regarding $(D^*, X) = \text{reduce}(D, S^*)$.

**Case 1:** $D^* \neq 0$. By property (3) of the general invariant, the constraints added to the input weak PR-tree $D$ by the template matching are PT-constraints of $S^*$ (i.e., $\mathcal{PT}_S = \mathcal{PT}_{S^D} \subseteq \mathcal{PT}_{S^*} \subseteq \mathcal{PT}_{S^*}$). Also, by observation 5.6, $S^* \in S^D$ (since $X = LCA(D^*, S^*)$ is an explicit P-node or R$'$-node in $D^*$ and $L(X) = S^*$). Thus, $\mathcal{PT}_{S^D} = \mathcal{PT}_{S^*}$.

Furthermore, by property (4) and observation 5.6, for every P-node source or R$'$-node source $N$ of $D^*$, $N$’s leaf set is a constraint in $S^*$, and, for every $S \in S^*$, $S$ is either the leaf set of a P-node or R$'$-node in $D^*$. Therefore, $D^*$ represents $S^*$.

**Case 2:** otherwise (i.e., $(D^*, X) = (0, 0)$). Consider the weak PR-tree $D$ at any point during the template matching process. Notice that, by (2), $\mathcal{PT}_S \subseteq S_D \subseteq \mathcal{PT}_{S^*}$ (since $S_D = \mathcal{PT}_S \subseteq \mathcal{PT}_{S^*}$ before any templates have been applied to the nodes of $D$). Thus, if we observe that $D$ is irreducible at any point in the template matching process, then there is no path-tree model that satisfies $S_D$; i.e., $\mathcal{T}_{S^*} = \emptyset$ since $S_D \subseteq \mathcal{PT}_{S^*}$. 
We now consider the two ways that \textit{reduce} can return $(0,0)$; i.e., via lines 13 and 24.

\textbf{Case 2a:} $(0,0)$ is returned via line 13. We now have an eligible P-node or weak R-node $N$ which cannot be matched by a template. By (1) of the general invariant, we know that $D$ is irreducible. Thus, $\mathcal{T}_{S^*} = \emptyset$.

\textbf{Case 2b:} $(0,0)$ is returned via line 24. By observation 5.5, we know that $\mathcal{T}_{S^*} = \emptyset$. Thus, $\mathcal{T}_{S^*} = \emptyset$ since $S_{D^*} \subseteq \mathcal{PT}_{S^*}$.

Recall that we proved the correctness of the \textit{PRtreeReduction} method assuming the correctness of \textit{reduce}. Thus, by theorem 5.2, the \textit{PRtreeReduction} method is correct.

### 5.4 Consequences of Reduction

In this section we will summarize the results in this chapter and present some interesting consequences of them.

First, for a constraint set $S$ or graph $G$ with $S_G = S$, we have seen that the reduction technique can be used to construct a weak PR-tree that represents $S$ when there is a path-tree model that satisfies $S$ (i.e., $\mathcal{T}_S \neq \emptyset$). Additionally, our reduction algorithm finishes in $O(A(n+m)nm)$ time. This result is presented as theorem 5.8. Notice that, this result provides the immediate corollary that every path graph is representable by a weak PR-tree (see corollary 5.9).

**Theorem 5.8.** For a constraint set $S$ or graph $G$ (with $S_G = S$), in $O(A(\sum_{S \in S} |S|) * |S| * \sum_{S \in S} |S|) \text{ or } O(A(n + m)nm)$ time, \textit{PRtreeReduction($S$)} (i.e., algorithm 8) either produces a weak PR-tree that represents $S$, or returns $\emptyset$ to indicate that $\mathcal{T}_S = \emptyset$.

**Proof.** This follows from theorems 5.2, 5.3, and 5.7, corollary A.5, and the fact that $S_G$ can be determined in $O(n + m)$ time since weak PR-trees can only represent chordal graphs (by corollary 3.30).

**Corollary 5.9.**. Every path graph is representable by a weak PR-tree.
Proof. This follows from theorem 5.8.

Recall that, for a graph $G$, having a weak PR-tree $D$ that represents $G$ does not mean that there is necessarily a path-tree model of $G$ (i.e., $G$ is not necessarily a path graph). However, by theorem 3.38 in section 3.5, we can use $D$ to check whether such a path-tree model exists using the $\text{fixRnode}$ method. In fact, in $O(n + m)$ time, we can either determine a path-tree model of $G$ or provide a certificate indicating that there is no path-tree model of $G$. Theorem 3.39 provides us with a similar result regarding directed path-tree models of $G$. Thus, we have the following theorems and corollaries.

**Theorem 5.10.** For a graph $G$ and $D = \text{PRtreeReduction}(S_G)$, applying $\text{fixRnode}$ to each weak R-node in $D$ has the following properties:

1. The total time taken to construct $S_G$, and execute $\text{PRtreeReduction}(S_G)$ and the calls to $\text{fixRnode}$ is $O(A(n + m)nm)$.

2. If $D \neq \emptyset$ and $D$ has a PR-tree (i.e., $T_G \neq \emptyset$), then one is produced.

3. If $T_G = \emptyset$, then either $D = \emptyset$ due to $\text{PRtreeReduction}$ indicating that $T_G = \emptyset$ or $D \neq \emptyset$ and a false twin odd sun in $G$ is identified by one of the calls to $\text{fixRnode}$ (i.e., certifying that $T_G = \emptyset$ since false twin odd suns are not path graphs).

I.e., the PR-tree together with this reduction algorithm solves the path-tree problem (see def. 2.2).

Proof. This follows from theorems 5.8 and 3.38.

**Corollary 5.11.** The path graph recognition problem can be solved in $O(A(n + m)nm)$ time using the reduction technique and weak PR-trees.

Proof. This follows from theorem 5.10.

**Theorem 5.12.** For a graph $G$ and $D = \text{PRtreeReduction}(S_G)$, applying $\text{strengthenRnode}$ to each weak R-node in $D$ has the following properties:
(1) The total time taken to construct $S_G$, and execute $PRtreeReduction(S_G)$ and the calls to $strengthnRnode$ is $O(A(n + m)nm)$.

(2) If $D \neq \emptyset$ and $D$ has a strong PR-tree (i.e., $\mathcal{T}_G \neq \emptyset$), then one is produced.

(3) If $\mathcal{T}_G = \emptyset$, then either $D = \emptyset$ due to $PRtreeReduction$ indicating that $\mathcal{T}_G = \emptyset$ or $D \neq \emptyset$ and an odd sun in $G$ is identified by one of the calls to $strengthnRnode$ (i.e., certifying that $\mathcal{T}_G = \emptyset$ since odd suns are not directed path graphs).

I.e., the strong PR-tree together with this reduction algorithm solves the directed path-tree problem (see def. 2.2).

Proof. This follows from theorems 5.8 and 3.39.

Corollary 5.13. The directed path graph recognition problem can be solved in $O(A(n + m)nm)$ time using the reduction technique and weak PR-trees.

Proof. This follows from theorem 5.12.
Chapter 6

Concluding Remarks

We separate the concluding remarks of this thesis into two sections. The first section summarizes the results we have presented and the second section discusses some open problems and future work relating to these results.

6.1 Summary of Results

The main contributions of this thesis revolve around the PR-tree (see def. 3.10): a new data structure which efficiently captures the path-tree models of graphs. Also, (with minor restrictions to the PR-tree) we have the strong PR-tree which efficiently captures the directed path-tree models of graphs. Additionally, we have the slightly more general (i.e., more general than the PR-tree) weak PR-tree (see def. 3.1) which is the work horse throughout this thesis. For a graph represented by a weak PR-tree / PR-tree / strong PR-tree $D$, $D$ can be explicitly described in $O(n + m)$ space (by corollary 3.34) and implicitly described in $O(n)$ space (by theorem 4.15).

We now recall the characterization results followed by the algorithmic results presented in this thesis.

In the following theorems we summarize the characterizations of graphs which are representable by: weak PR-trees, PR-trees, and strong PR-trees. Recall that theorem
4.17 states that:

A graph $G$ is representable by a weak PR-tree iff $G$ is chordal and the split components of any split decomposition of $G$ are path graphs.

Also, by corollary 5.9 and theorems 3.38 and 4.22 we have the following PR-tree characterization theorem.

**Theorem 6.1.** *(PR-tree Characterization Theorem)* For a graph $G$, the following are equivalent:

1. $G$ is representable by a PR-tree.
2. $G$ is a path graph.
3. $G$ is representable by a weak PR-tree and $G$ contains no induced false twin odd sun.
4. $G$ is chordal, the split components of any split decomposition of $G$ are path graphs, and $G$ contains no induced false twin odd sun.

Notice that these equivalences mean that the consistent sets of PR-trees are precisely the sets of path-tree models of path graphs. Similarly, by corollary 5.9 and theorems 3.39 and 4.23, we have the following strong PR-tree characterization theorem.

**Theorem 6.2.** *(Strong PR-tree Characterization Theorem)* For a graph $G$, the following are equivalent:

1. $G$ is representable by a strong PR-tree.
2. $G$ is a directed path graph.
3. $G$ is representable by a weak PR-tree and $G$ contains no odd sun.
4. $G$ is chordal and the split components of any split decomposition of $G$ are directed path graphs.
Notice that these equivalences mean that the directed consistent sets of weak PR-trees are precisely the sets of directed path-tree models of directed path graphs.

We now summarize the algorithmic results presented in this thesis. In particular, we recall the Split Decomposition (of chapter 4) and reduction (of chapter 5) weak PR-tree construction algorithms.

In section 3.5 we presented two algorithms. For a graph $G$, in $O(n + m)$ time, the first algorithm converts a weak PR-tree representing $G$ into a PR-tree representing $G$ (when possible) and otherwise identifies a false twin odd sun in $G$ (i.e., certifying that $G$ is not representable by a PR-tree). Similarly, for a graph $G$, in $O(n + m)$ time, the second algorithm converts a weak PR-tree representing $G$ into a strong PR-tree representing $G$ (when possible) and otherwise identifies an odd sun in $G$ (i.e., certifying that $G$ is not representable by a strong PR-tree).

In chapter 4 we provided an algorithm to construct a weak PR-tree from a graph $G$ (when possible) in $O(\max\{n + m, f(n, m)\})$ time using split decomposition and weak PR-trees where $f(n, m)$ is the amount of time required to construct a path-tree model (when one exists) from a prime chordal graph $G$. Currently, $f(n, m) \in O(nm)$. In particular, this algorithm directly demonstrates that the graphs representable by weak PR-trees are the chordal graphs whose split components (with respect to split decomposition) are path graphs. Additional consequences of this algorithm are presented in detail in section 4.5. In particular, the recognition of path graphs and directed path graphs can be performed in $O(\max\{n + m, f(n, m)\})$ time via this algorithm (this matches the current best known algorithms to recognize path graphs and directed path graphs).

In chapter 5 we presented an algorithm to construct a weak PR-tree from graph $G$ (when $G$ is a path graph) in $O(A(n + m)nm)$ time using the reduction technique and weak PR-trees. In particular, this algorithm demonstrates that every path graph is representable by a weak PR-tree (note: this does not follow from the results in chapter 4).

†Where $A(s)$ is the inverse Ackermann function presented in appendix A.3.1.
see “A Simpler Road to Path Graph Characterization” in section 6.2). The consequences of this algorithm are presented in detail in section 5.4. Two major consequences of this algorithm are the characterizations of path graphs by PR-trees and directed path graphs by strong PR-trees. Additionally, the recognition of path graphs and directed path graphs can be performed in $O(A(n + m)nm)$ time via this algorithm.

## 6.2 Open Questions and Future Work

There are several avenues of open questions regarding PR-trees and related problems. Some of these are outlined as follows (and discussed further below). First, the recognition problem for path graphs which are prime with respect to split decomposition warrants particular attention. We are also considering other graph classes which can be characterized by PR-trees. Additionally, we believe that the PR-tree can be used to determine graph parameters such as leafage and asteroidal number. Finally, we consider an alternate proof that every path graph is representable by weak PR-trees using our split decomposition algorithm.

**Prime Path Graphs.** We have seen that path graphs which are prime with respect to split decomposition have a unique path-tree model. Also, the speed at which this unique path-tree model can be generated plays a key role regarding constructing weak PR-trees, and recognizing path graphs and directed path graphs. In particular, from our split decomposition approach regarding constructing weak PR-trees, an $O(f(n,m))$ algorithm to generate the path-tree model of a prime chordal graph (and recognize when no such model exists) would provide an $O(f(n,m))$ algorithm for the following problems:

- Generating a weak PR-tree from a graph.
- Recognizing path graphs.
- Recognizing directed path graphs.
Currently, the best approach to this problem is to simply use the path graph recognition algorithm of Schäffer [38]. This algorithm runs in $O(nm)$ time (for any graph). However, it is possible that a careful examination of either Schäffer’s algorithm or our reduction algorithm (as presented in chapter 5) could provide better time bounds in this restricted case (i.e., when the input is a chordal graph which is prime with respect to split decomposition).

Additionally, we are investigating an incremental approach to construct a weak PR-tree (when it exists) from a prime chordal graph via lexicographic breadth-first search (this is similar to the approach that Korte and Möhring have taken to simplify the construction of PQ-trees [29]). We hope that this will lead to a simpler and faster algorithm than either the reduction algorithm presented in chapter 5 or Schäffer’s algorithm [38].

**New Directions in Characterization.** The rooted path graphs form a graph class which we believe is a good candidate to be characterized by a variation of PR-trees. Rooted path graphs are intersection graphs of directed paths in rooted trees and they also have a clique tree theorem [35]. This graph class is contained strictly between the interval and directed path graphs. In particular, we imagine that there could be a variation of PR-trees which captures the set of all “rooted path-tree models” of a graph. Currently there is no known forbidden induced subgraph characterization of this graph class. Also, the best published recognition algorithm for rooted path graphs runs in $O(n^4)$ time (from Gavril [19]). There is also a linear time algorithm to recognize rooted path graphs that has appeared in the PhD thesis of Deitz [14] and this result was never published in a journal. It is important to note that both Deitz and Gavril refer to rooted path graphs as “directed path graphs” in their papers.

**Graph Properties and PR-trees** We expect that some graph properties can be determined from a weak PR-tree that represents a particular graph. Two parameters that we believe are good candidates for this are: leafage and the asteroidal number.
The **leafage** of a chordal graph $G$ is defined to be the minimum number of leaves of a tree-tree model (i.e., a clique tree) of $G$ (i.e., it measures how “close” a chordal graph is to an interval graph). This measure was first introduced by Lin, McKee, and West in [32] where the authors establish several bounds on this parameter for special cases of chordal graphs such as block graphs, split graphs, and k-trees. In [32] it was asked whether a polynomial time algorithm could be devised to compute leafage. This was recently positively answered by Habib and Stacho in [25]. In particular, in [25], an $O(n^3)$ time algorithm is presented which not only determines the leafage of a given graph $G$, but also produces a tree-tree model satisfying it. In a recent paper [5] we have shown that the leafage of a path graph $G$ (i.e., the minimum number of leaves in a tree-tree model $G$) is the same as the minimum number of leaves of a path-tree model of $G$. In particular, since the set of all path-tree models of a path graph $G$ can be generated by applying equivalence transformations to the PR-tree representing $G$, we expect that a path-tree model with the minimum number of leaves can be extracted from a PR-tree of $G$. We are currently actively pursuing this problem.

Another parameter that we conjecture can be calculated from a weak PR-tree is the **asteroidal number** (this parameter was introduced in [42] and is defined as follows). A set of vertices $A$ of a graph $G$ is an **asteroidal set** if for each $a \in A$, the vertices in $A \setminus \{a\}$ belong to a common connected component of $G$ after removing the closed neighbourhood of $a$; the **asteroidal number** of $G$, denoted by $a(G)$, is the size of a largest asteroidal set of $G$. One reason to be interested in the asteroidal number is the fact that we can efficiently solve several problems on chordal graphs that would otherwise be hard when the asteroidal number is bounded. For instance, the maximum independent dominating set can be solved efficiently when the asteroidal number is bounded [3]. Currently, the best known algorithm to calculate the asteroidal number of a path graph (or graph which is representable by a weak PR-tree) is one which will calculate the asteroidal number for HHD-free graphs (a superset of chordal graphs) in $O(n^3 + n^{3/2}m)$ time [27].
A Simpler Road to Path Graph Characterization  Recall that, in theorem 4.12, we proved that, when a graph is representable by a weak PR-tree and prime with respect to split decomposition, it has a unique path-tree model. Notice that, if we had a “simple” proof that every path graph which is prime with respect to split decomposition has a unique path-tree model, then our split decomposition approach to weak PR-tree construction would demonstrate that every path graph is representable by a weak PR-tree (without needing to rely on our reduction algorithm). In particular, the conclusions and consequences arising from our split decomposition algorithm would not rely on our reduction algorithm at all. We are currently working on such a proof.
Bibliography


[18] F. Gavril. The intersection graphs of subtrees of trees are exactly the chordal graphs. 


[21] Emeric Gioan, Christophe Paul, Marc Tedder, and Derek G. Corneil. Circle graph recognition in time \(o(n + m)\alpha(n + m)\). *CoRR*, abs/1104.3284, 2011.


Appendix A

Reduction: Template Matching

In this section we provide the details of the template matching portion of the reduce operation. Recall that, for a given weak PR-tree $D$ representing a constraint set $S$ and $S^* \subseteq L(D)$, the primary goal of $\text{reduce}(D, S^*)$ is to produce a weak PR-tree, $D^*$ representing $S^* = S \cup \{S^*\}$ when $T_{S^*} \neq \emptyset$. Notice that, in such a $D^*$, there will either be a P-node or R'-node $X$ where $L(X) = S^*$. We will refer to $S$, $S^*$, $D$, $S^*$, $D^*$, and $X$ throughout this appendix. Also, it is important to recall the terminology introduced at the beginning of section 5.3 and in section 5.3.1 as it will be used throughout this appendix.

Prior to beginning the template matching we traverse the tree bottom-up from the set $S^*$ to locate its LCSA and to identify the pertinent nodes of $D$. Additionally, we identify whether each pertinent node is p-bridged or r-bridged, and open, one-sided, or closed (note: if we encounter a node with three or more pertinent parents or pertinent neighbours, we halt unsuccessfully indicating that $D$ is irreducible). This is referred to as the label method and is discussed in section 5.3.3.

There is only one template for the leaves (i.e., when a leaf is a member of $S^*$). This template does not change $D$, but matching a leaf does mark it as full.

The situation for non-leaf nodes is not so simple. For a non-leaf node $N$, we must
ensure that $N$ satisfies $S^*$ (i.e., after $N$ is matched: $S^* \cap L(N) \in PT_{S_D}^*$) while providing $N$’s pertinent parents with the opportunity to satisfy $S^*$. Additionally, we must make sure that $N$’s children will not be bridged.

Template matching is only performed on pertinent nodes. In the visualization of the templates we will preserve the left to right order of the subtrees (represented as triangles - see fig. A.1) when going from the pattern to the replacement (i.e., the $i^{th}$-leftmost subtree with respect to the pattern will appear as the $i^{th}$-leftmost subtree of the replacement). Additionally, since we present the pattern of a template with a specific left to right order on subtrees, a match for a template exists when an equivalence transformation can be applied to the node $N$ being matched to re-order these subtrees as in the pattern of the template. In particular, the transformation is applied to $N$ and then the template’s replacement is substituted into $D$.

We will now present the P-node templates with respect to a P-node $P$, followed by the weak R-node templates with respect to a weak R-node $R$ (note: the weak R-node templates will affect $R$’s tree and $R$’s set of R’-nodes).

We first recall the general invariant (as stated in lemma 5.4) to which the templates will adhere and present a symbol legend (see fig. A.1) for the templates. The descriptions of some of these symbols use the terminology that was introduced in section 5.3.1. This figure is placed here for easy reference.

General Invariant:

1. If an eligible node has no applicable template, $D$ is irreducible (i.e., there is no $T \in T_{S_D}$ such that $T[S^*]$ is a path).

2. Once a node $N$ of $D$ has been matched (i.e., a template has been applied to $N$), it will satisfy $S^*$ (i.e., $L(N) \cap L^*$ will have an LCA whose leaf set is precisely $L(N) \cap L^*$).

3. The constraints added to $D$ by matching a node are PT-constraints of $S^*$ (i.e., $PT_{S_D}$

\[ \text{Notice that } PT_{S_D \cup \{S^*\}} = PT_{S_D \cup \{S^*\}} = PT_{S^*} \quad \text{and, by observation 2.8, } S^* \cap L(N) \in PT_{S_D \cup \{S^*\}}. \]
Appendix A. Reduction: Template Matching

133

constraints of $S_D \cup \{S^*\})$.

(4) Matching a node $N$ preserves the property that, for every $P$-node source or $R'$-node source $N^*$ of $D$, $N^*$’s leaf set is a constraint in $S$, and, for every $S \in S$, $S$ is either the leaf set of a $P$-node or $R'$-node in $D$.

(5) After the pertinent $P$-node parents of a node $N$ are matched, $N$ will not be bridged.

(6) After the pertinent parents of a node $N$ are matched, $N$ is pertinent iff it is full.

Figure A.1: The symbol legend for templates.

<table>
<thead>
<tr>
<th>Symbol(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>An empty leaf (i.e., not in $S^*$).</td>
</tr>
<tr>
<td></td>
<td>A full leaf (i.e., in $S^*$).</td>
</tr>
<tr>
<td></td>
<td>An empty or unlabelled $P$-node.</td>
</tr>
<tr>
<td></td>
<td>A full $P$-node.</td>
</tr>
<tr>
<td></td>
<td>An empty or unlabelled $R'$-node.</td>
</tr>
<tr>
<td></td>
<td>A full $R'$-node.</td>
</tr>
<tr>
<td></td>
<td>A bridged-full $R'$-node.</td>
</tr>
<tr>
<td></td>
<td>A partial $R'$-node.</td>
</tr>
<tr>
<td></td>
<td>A bridged-partial $R'$-node.</td>
</tr>
<tr>
<td></td>
<td>A subtree whose root is:</td>
</tr>
<tr>
<td></td>
<td>An empty $P$-node, $R'$-node, or leaf (i.e., an empty subtree).</td>
</tr>
<tr>
<td></td>
<td>A full $P$-node, an accessible full $R'$-node, or a full leaf (i.e., a full subtree).</td>
</tr>
<tr>
<td></td>
<td>A $p$-bridged-full node (i.e., a $p$-bridged-full subtree).</td>
</tr>
<tr>
<td></td>
<td>A open or one-sided $r$-bridged-full node (i.e., an $r$-bridged-full subtree).</td>
</tr>
</tbody>
</table>
Appendix A. Reduction: Template Matching

A full subtree such that: when k=0 its root is open, when k=1 its root is one-sided, and when k=2 its root is closed. Note: we similarly number full P-nodes to indicate their pertinent neighbours.

A.1 P-node Templates

We present the P-node templates with respect to a pertinent P-node $P$. These templates will be presented in five groups. In the first group $P$ will be the sole parent of its children and will only have full children. The second group expands on the first by allowing $P$ to have bridged-full children and empty children. The third group further introduces blocked-full $R'$-node children to $P$. In the fourth group $P$ will be able to have partial children (note: this concept will be defined this later in this subsection) in addition to the possibilities of the third group. The final group $P$ we complete the options for $P$’s children by including bridged-partial children (again, we introduce this concept later on in this section). We further subdivide each of the latter four groups into three cases regarding the number of pertinent parents of $P$ (i.e., when $P$ has zero, exactly one, or exactly two pertinent parents).

A.1.1 Template Group P1: Full Children (an easy case)

The first P-node template (see Template P1 in fig. A.2) concerns the case when $P$ is the sole pertinent parent of its children and all of $P$’s children are full. In this case, the replacement does not change $P$ (i.e. $P$ remains a P-node and its bridged status does not change) and simply marks $P$ as full. This is the only template in which $P$ will become a full P-node.
**Appendix A. Reduction: Template Matching**

**Pattern:**

```
  P
 / \...
  ...  
```

**Replacement:**

```
  P
 / \ الكتاب
  ...  
```

Figure A.2: Template P1: matching a P-node that has only full children.

### A.1.2 Template Group P2: Bridged-Full Children

The second group is the set of templates where \( P \) has a mixture of full, empty, and at most two bridged-full children. We will present this group of templates (and each of the remaining groups of templates) through the following three subcases:

(a) When \( P \) is in the LCSA of \( S^* \) (i.e., \( P \) has no pertinent parents).

(b) When \( P \) has exactly one pertinent parent.

(c) When \( P \) is bridged (i.e., when \( P \) has exactly two pertinent parents).

We will need to collect the bridged-full and full children together in order to ensure that \( P \) will satisfy \( S^* \). Furthermore, the bridged-full children must be forced to occur at opposite ends of the full children so that each bridged-full child’s other pertinent parent will have the opportunity to properly arrange its children. Similarly, when there is only one bridged-full child, we need to insist that it is not allowed to occur in the middle of the full children.

Template \( P2a^* \) (see fig. A.3) performs this operation when \( P \) is a source of the pertinent subtree. In this case \( P \) is left unlabelled since it was part of the LCSA of \( S^* \) (i.e., we will not be adding parents of \( P \) to the queue). Notice that the r-bridged-full child (see \( C \) in the pattern of fig. A.3) of \( P \) gains a pertinent neighbour and is no longer bridged (see \( C \) in the pattern of fig. A.3) and the p-bridged-full child (see \( C^* \) in the pattern of fig. A.3) of \( P \) becomes a one-sided r-bridged-full node (see \( C^* \) in the replacement of fig. A.3).

\[^1\text{Note: the left-to-right order of the subtrees is preserved when going from the pattern to the replacement.}\]
Appendix A. Reduction: Template Matching

Recall that, even though we visualize the pattern of template P2a with a specific ordering on $P$'s children, they could have an alternate ordering and still match this template. In particular, a match for template P2a exists when an equivalence transformation can be applied to $P$ to re-order $P$'s children as in the pattern of template P2a (i.e., the transformation is applied to $P$ and then P2a's replacement is substituted into $D$).

**Pattern:**

![Diagram of Pattern]

**Replacement:**

![Diagram of Replacement]

Figure A.3: Template P2a: matching a P-node in LCSA($D,S^*$) (i.e., with no pertinent parents) that has full, empty, and bridged-full children.

Prior to applying template P2a, the full, empty, and bridged-full children of $P$ are free to rearrange in any order. Afterwards this is not the case. Empty and full children can no longer intermingle and the bridged-full children are forced to be at opposite ends of the full children (this is done via the newly created R'-node). If the “reduced” weak PR-tree (i.e., $D^*$ post-template matching) is to have an orientation in which the elements of $S^*$ occur as a path in its frontier, no node having non-pertinent descendants may occur between two nodes having pertinent descendants. Respecting this inevitability, the replacement shown retains the most flexibility. In particular, empty children are free to rearrange amongst themselves (since their parent is a P-node) and so are the full children (since their parent is the newly created full P-node). Moreover, the group of full children (i.e., the newly created R'-node) is free to occur anywhere within the empty children.

Template P2a has many degenerate forms (see fig. A.4 for a representative subset of them). The need for degenerate cases arises from the fact that a P-node must have at least three children and an R'-node must have at least two children. In particular, these
occur when there are fewer than three (non-bridged) full children (see (i) and (ii) in fig. A.4), fewer than two empty children (see (iii) and (iv) in fig. A.4), or no bridged-full children (see (v) in fig. A.4). Notice that in the replacement of (i), we place the number one below each of the full subtrees of the lower full $R'$-node. This indicates that the root of each of these subtrees will be one-sided (i.e., it has exactly one pertinent neighbour). Similarly, we would place a two or zero below a full subtree to indicate that it is closed or open respectively. To build any degenerate case (other than those in fig. A.4) we simply combine the appropriate elements from the given cases. In particular, from (i) and (ii), we can see that the number of (non-bridged) full children determines the form of the middle child in the newly created $R'$-node in the replacement. Moreover, from (iii) and (iv), we can see that the number of empty children determines whether $P$ is a $P$-node or $R'$-node in the replacement. From (v), when $P$ only has one any bridged-full child, the full $R'$-node in the replacement will only have two children. Finally, from (vi), when $P$ does not have any bridged-full children, we exclude the full $R'$-node from the replacement.

Also, rather than having one p-bridged-full child and one r-bridged-full child, $P$ may have two p-bridged-full children or two r-bridged-full children. The replacement for each of these cases is clear from fig. A.3 (i.e., the p-bridged children become one-sided and r-bridged, and the r-bridged children gain a pertinent neighbour and are no longer bridged). Similarly, $P$ may have an r-bridged-full child rather than a p-bridged-full child (as in (v) from fig. A.4).

The replacements for template P2a (given by figures A.3 and A.4) do not tell the entire story for this case. Recall that every $R'$-node must be associated with exactly one weak $R$-node. In particular, since template P2a will potentially create new $R'$-nodes, we must either create new weak $R$-nodes or modify existing weak $R$-nodes (as necessary) to accommodate these new $R'$-nodes. For example, if both bridged-full children have weak $R$-node parents $R_1$ and $R_2$ (note: these need not be pertinent), the replacement
Figure A.4: A representative subset of the degenerate forms of template P2a.
given in fig. A.3 would also involve replacing $\mathcal{R}_1$ and $\mathcal{R}_2$ with a new weak R-node $\mathcal{R}$ which is the union $\mathcal{R}_1$ and $\mathcal{R}_2$ via the new $R'$-node $Q$ given in the replacement (i.e., $\mathcal{R}_{R'} = (\mathcal{R}_1)_{R'} \cup (\mathcal{R}_2)_{R'} \cup \{Q\}$ and $\mathcal{T}$ is the tree formed by connecting $(\mathcal{R}_1)_{T}$ to $(\mathcal{R}_2)_{T}$ via the path formed by $Q$). This is a common operation that will be used throughout the template matching process and must be considered whenever a new $R'$-node is created. We will refer to this operation as reconciling $P$’s related weak R-nodes.

Most P-nodes do not occur as a source of the pertinent subtree. We now consider the case when $P$ is not bridged and is not a source of the pertinent subtree. As noted earlier, when $P$ has exactly one pertinent parent it can have at most one bridged-full child (otherwise $D$ is irreducible). As in template P2a, we collect the full children together with the bridged-full child. However, since $P$ has a pertinent parent, there is a full leaf which is not a descendent of $P$ and is a descendent of an ancestor of $P$. In the frontier of any orientation $D'$ of $D$ where $S^*$ induces a path, one such leaf will have to become adjacent to a full endpoint of $\text{Frontier}(D')[L(P)]$. This implies that all of the full children of $P$ must be together and not in between empty children of $P$. Similarly, the bridged-full child must occur in between the full and empty children of $P$. Template P2b (see fig. A.5) performs this operation. Notice that this operation converts $P$ into an $R'$-node with some full and some empty children, we refer to such an $R'$-node as partial. Partial nodes are partially shaded, with the shading indicating which children are full (see fig. A.1). We will also refer to a P-node with some full and some empty children as partial, but we do not provide a symbol for it. Additionally, in the replacement of template P2b, we place the number one below the newly created full P-node to indicate that it is one-sided (i.e., it has exactly one pertinent neighbour). Similarly, we would place a two or zero below a full P-node to indicate that it is closed or open respectively.

Similarly to template P2a, there are degenerate forms for template P2b (we have omitted these since they are clear from the degenerate forms of template P2a). Also, since template P2b will introduce a new $R'$-node, we have to be careful to maintain the
Appendix A. Reduction: Template Matching

**Pattern:**

![Diagram of Pattern](image1)

**Replacement:**

![Diagram of Replacement](image2)

Figure A.5: Template P2b: matching a partial P-node that has exactly one pertinent parent (i.e., \( P \notin LCSA(D, S^*) \) and not bridged) and bridged-full children.

appropriate weak R-node relationships.

The final template in the second group concerns the case when \( P \) is bridged and has at least one empty child§. Notice that if \( P \) were to have a bridged child, \( D \) would be irreducible. **Template P2c** (see fig. A.6) performs the appropriate action for this case. There is one degenerate case for template P2c (i.e., when there are exactly two empty children), and its replacement consists of creating an empty \( R' \)-node instead of creating the empty P-node.

**Pattern:**

![Diagram of Pattern](image3)

**Replacement:**

![Diagram of Replacement](image4)

Figure A.6: Template P2c: matching a bridged partial P-node.

Notice that this template has a quite restrictive pattern. In particular, \( P \) must have exactly one full child and that child must be a full leaf (note: we use a shaded diamond to depict a full leaf). Suppose that \( P \) has multiple full leaves (an example of a weak PR-tree with such a \( P \) is given in fig. A.7). Since \( P \) must satisfy \( S^* \), these full leaves must occur consecutively in \( P \)'s frontier for every orientation \( D' \) of \( D \) (after \( P \) has been matched). Furthermore, since \( P \) has at least one empty child, it will have at least one empty leaf. Thus, \( P \)'s frontier will have at most one full endpoint and this full endpoint

---

§Template P1 covers the case when \( P \) is bridged with only full children.
will have a full neighbour in $P$’s frontier. Also, since $P$ has two pertinent parents (i.e., $P$ is bridged), there are two full leaves (which are not leaves of $P$) in the frontier of $D’$ that must each be adjacent to a full endpoint of the frontier of $P$. In particular, these full leaves will be adjacent to the one full endpoint of $P$’s frontier (which already has a full neighbour within $P$’s frontier). Thus, if $P$ is bridged and has two full leaves, $D$ is irreducible. Fig. A.7 shows a weak PR-tree that demonstrates this condition. In this figure we see a weak PR-tree $D$ (left) with a bridged P-node $P$ that has a full child with more than one leaf and pertinent parents $P_1$ and $P_2$. The children of $P$, $P_1$, and $P_2$ are labelled full and empty with respect to the constraint $S^* = \{u_1, u_2, u_3, u_4\}$. The connected subgraphs of the trees from $Consistent(D)$ induced by $S^*$ are given to the right of $D$. Notice that neither of these are paths (i.e., $D$ is irreducible with respect to $S^*$).

![Figure A.7: A weak PR-tree $D$ (left) and the connected subtrees induced by the constraint $\{u_1, u_2, u_3, u_4\}$ from frontiers of orientations of $D$ (right). This demonstrates the necessity of the full child in template P2c being a leaf.](image)

### A.1.3 Template Group P3: Blocked-Full Children

We have now handled the matching process for partial P-nodes with bridged-full children. Recall that we assumed that every full R'-node is, by default, an accessible full R'-node (i.e., in the previous P-node templates every full R'-node child of $P$ was accessible). The next set of templates considers the case when $P$ has at least one blocked-full R'-node child (note: no children of $P$ can be surrounded R'-nodes, since this would make $D$ irreducible). In particular, $P$ will now have either one or two blocked-full R'-node
Appendix A. Reduction: Template Matching

children (note: if $P$ had three of them, $D$ would be irreducible – as previously observed). We now revisit the same three cases for $P$ (i.e., $P$ being bridged, having no pertinent parents, and having exactly one pertinent). As we have already mentioned, when $P$ is bridged, it cannot have a blocked-full $R'$-node child. Thus, we do not match $P$ when it is bridged and has a blocked-full $R'$-node child.

We now consider the case when $P$ has no pertinent parents. As in the previous set of templates, we will group the non-empty children together while ensuring that the blocked-full children are appropriately placed to opposite ends of the full children while allowing this group to appear anywhere within the empty children. However, the blocked-full children require more care than the bridged full children did. In particular, if we simply create a new $R'$-node whose “left-most” and “right-most” children consist of the blocked-full children (similarly to template P2a), we will not sufficiently restrict the consistent set. More specifically, we must force $P$’s full children to occur in between the one-sided children of $P$’s blocked-full children. Template P3a (see fig. A.8) accomplishes this while maintaining maximum flexibility.

![Diagram](image)

**Pattern:**

![Diagram](image)

**Replacement:**

![Diagram](image)

Figure A.8: Template P3a: matching a partial P-node that has no pertinent parents and blocked-full $R'$-node children.
There are alternate forms for template P3a. In particular, rather than having two blocked-full children, $P$ could have one blocked-full child and one of either variety of bridged-full children. These forms are not explicitly presented, since are clear from templates P3a and P2a.

We derive the template for the case when $P$ has exactly one pertinent parent and blocked-full children in an analogous way to how we derived template P2b from template P2a. In particular, we must again ensure that $P$’s full children are available to its pertinent parent. We do this by insisting that they do not appear in between empty children. Also, $P$ can have at most one blocked-full child which will need to be forced to occur between $P$’s empty and full children so that its closed child is adjacent to $P$’s empty children and its one-sided child is adjacent to $P$’s full children. Template P3b (see fig. A.9) accomplishes this.

![Pattern:](image1)

**Pattern:**

![Replacement:](image2)

**Replacement:**

Figure A.9: Template P3b: matching a partial P-node that has exactly one pertinent parent and blocked-full children.

Similarly to the second group of templates, there is a family of degenerate cases that accompany template P3a, the alternate forms of template P3a, and template P3b. Furthermore, as usual, we must be careful to properly merge the relevant weak R-nodes when implementing this group of templates (i.e., we need to reconcile $P$’s related weak R-nodes). Notice that once $P$ has been matched its blocked-full $R'$-node children will no longer have $P$ as a parent. In particular, these blocked-full nodes will no longer be pertinent since they have lost their only pertinent parent. This property will be useful regarding the proof of correctness of our algorithm. Additionally, each blocked-full $R'$-
node child $Q$ of $P$ will be a subpath of the newly created $R'$-node after $P$ has been matched. Thus, if $Q$ has no parents in $D$ after $P$ has been matched, $Q$ will be redundant and we will remove $Q$ from $D$. Notice that no other children of $P$ will become orphaned due to the templates of this group.

### A.1.4 Template Group P4: Partial Children

We have now handled the matching process for P-nodes with any combination of empty children and every type of full children. However, in doing so we have introduced the potential for partial $R'$-nodes to be children of pertinent nodes (i.e., via templates P2b, P2c, and P3b). In the forth (and final) group of P-node templates, $P$ will now have at least one partial $R'$-node child. Notice that $P$ cannot have more than two partial $R'$-node children. In particular, when $P$ has three partial children it is easy to see that regardless of the choice of orientation $D'$ of $D$ (i.e., the order in which these partial children appear) the full leaves of $P$ will induce a disconnected subgraph in the frontier of $D'$ (i.e., $P$ cannot satisfy $S^*$). Thus, $D$ would be irreducible.

Once again, we consider three cases regarding the number of pertinent parents of $P$ (i.e., zero, one, or two). The templates for these cases are $P4a$, $P4b$, and $P4c$ respectively and are given in figures A.10, A.11, and A.12 respectively. These are analogous to templates P3a (fig. A.8), P3b (fig. A.9), and P2c (fig. A.6) respectively. Furthermore, each of these templates has a corresponding a set degenerate forms similar to the templates we have already seen. As with the previous groups, we will need to reconcile $P$'s related weak $R$-nodes during the execution of the templates in this group.

Similarly to template P3a, there are alternate forms for template P4a. In particular, rather than having two partial $R'$-node children, $P$ could have one partial $R'$-node child and a p-bridged-full child, an r-bridged-full child, or a blocked-full $R'$-node child. Since these forms are clear from templates P4a, P3a, and P2a, they are not explicitly presented.

Similarly to template group P3, once $P$ has been matched its partial children will no
Appendix A. Reduction: Template Matching

Pattern:

Replacement:

Figure A.10: Template P4a: matching a partial P-node that has no pertinent parents and partial children.

Pattern:

Replacement:

Figure A.11: Template P4b: matching a partial P-node that has exactly one pertinent parent and partial children.

Pattern:

Replacement:

Figure A.12: Template P4c: matching a bridged partial P-node that has partial children.

longer have \( P \) as a parent. In particular, these partial nodes will no longer be pertinent since they have lost their only pertinent parent. Again, this property will be useful regarding the proof of correctness of our algorithm. Additionally, each partial (or blocked-
full in the alternate forms) R'-node child $Q$ of $P$ will be a subpath of the newly created R'-node after $P$ has been matched. Thus, if $Q$ has no parents in $D$ after $P$ has been replaced, $Q$ is redundant and we remove $Q$ from $D$. Notice that no other children of $P$ become orphaned due to the templates of this group.

A.1.5 Template Group P5: Bridged-Partial Children

The P4 templates handle the case when the partial child is not bridged. However, as in the replacement of template P2c (and now P4c), a partial node can be bridged. A partial R'-node can only be bridged when it has exactly one full child, that full child is a leaf, and it is open (similarly to the replacements of templates P2c and P3c). Any other partial node would imply that $D$ is irreducible (since there would be no way to accommodate both pertinent parents, as demonstrated in fig. A.7). We refer to this form of bridged node as bridged-partial. Unlike the bridged-full nodes we do not distinguish p-bridged from r-bridged in our visualization. In this group we handle the case when $P$ has bridged-partial children. In particular, we provide analogous templates, P5a (see fig. A.13) and P5b (see fig. A.14), for templates P4a and template P4b respectively. As usual, each of these templates has a corresponding set of degenerate templates. There is no corresponding template to template P4c since $P$ being bridged implies it cannot have a bridged child. Furthermore, we again need to reconcile $P$’s related weak R-nodes as part of these templates.

Notice that in templates P5a and P5b the bridged-partial children are not bridged post-replacement. In particular, $P$ is no longer a parent of these nodes. However, since they were bridged to begin with they are not redundant after the replacement and as such remain in $D$. Additionally, for the alternate forms of the templates in this group, each partial (non-bridged) or blocked-full R'-node child $Q$ of $P$ will be a subpath of the newly created R'-node after $P$ has been matched. Thus, if $Q$ has no parents in $D$ after $P$ has been replaced, $Q$ is redundant and we remove $Q$ from $D$. Notice that no other
Appendix A. Reduction: Template Matching

Pattern:

Replacement:

Figure A.13: Template P5a: matching a partial P-node that has no pertinent parents and bridged-partial children.

Pattern:

Replacement:

Figure A.14: Template P5b: when the partial child is bridged.

children of $P$ will become orphaned due to the templates of this group.

A.1.6 The P-node Template Lemma

This completes the presentation of the P-node templates. We now revisit the invariant that we will use to prove the correctness of $\texttt{reduce}$. In particular, we will prove that the P-node templates satisfy the Template Lemma (see lemma 5.4). This was given at the end of subsection 5.3.1 and is repeated here (written with respect to the P-node templates) as the P-node Template Lemma:

Lemma A.1. (P-node Template Lemma) The following invariant holds during the
reduce operation.

P-node Invariant:

1. If an eligible P-node has no applicable template, D is irreducible.
2. Once a P-node P of D has been matched, it will satisfy $S^*$.
3. The constraints added to D by matching a P-node are PT-constraints of $S^*$.
4. Matching a P-node P preserves the property that, for every P-node source or R'-node source N of D, N’s leaf set is a constraint in $S$, and, for every $S \in S$, $S$ is either the leaf set of a P-node or R'-node in D.
5. After the pertinent P-node parents of a node N are matched, N will not be bridged.
6. After the pertinent P-node parents of a node N are matched, N is pertinent iff it is either full or has a pertinent weak R-node parent.

Proof.

1. As we have discussed throughout the presentation of the P-node templates, when such a P-node P is encountered, P will have:
   - Three (or more) of any combination of the following: bridged children, blocked-full R'-node children, partial children, bridged-partial children, and pertinent parents; or
   - A bridged-partial child with multiple full leaves; or
   - A partial child with multiple full leaves when P is bridged; or
   - A closed r-bridged child; or
   - A surrounded R'-node child.

Thus, D is irreducible.

For the remaining cases we let P be an eligible pertinent P-node in the weak PR-tree D which can be matched by some P-node template.
(2) : Notice that each P-node template will collect the full children and (when needed) grandchildren of $P$ together so that they will be consecutive in $P$’s frontier in every orientation of $D$ (after matching $P$). This will ensure that $P$ satisfies $S^*$ once $P$ has been matched. Additionally, no P-nodes or R'-nodes (other than $P$) are modified by the replacement. Furthermore, any weak R-node $R$, which is modified as a result of the replacement, will gain a path of full pertinent nodes which becomes adjacent to either a one-sided or an open node in $R_T$. Therefore, any node which satisfied $S^*$ prior to $P$ being matched will still satisfy $S^*$ after $P$ has been matched.

(3) : To prove this we consider $D^*$ to be the weak PR-tree $D$ after matching $P$. In particular, we will show that, for every tree $T$, $T \in \text{Consistent}(D)$ and $T[S^*]$ is a path iff $T \in \text{Consistent}(D^*)$ and $T[S^*]$.

\[ \iff \] Notice that the only constraints which are removed from $D$ by the application of a template are guaranteed to be implied constraints of $D^*$. This is clear since the only time a constraint $S$ is removed by the application of a P-node template is when: $S$ is the leaf set of R'-node $Q$ that becomes orphaned; and the path in the tree of a weak R-node that $Q$ specifies has become a subpath of the path specified by some newly created R'-node. Thus, $\text{Consistent}(D) \subseteq \text{Consistent}(D^*)$ (i.e., for every $T \in \text{Consistent}(D^*)$, $T \in \text{Consistent}(D))$.

\[ \implies \] Since $T \in \text{Consistent}(D)$, there is a PR-tree of $D$ whose frontier is $T$. Let $D'$ be such a PR-tree. We now let $D''$ be the result of matching $P$ in $D'$. Notice that $D''$ will be an orientation of $D^*$. We further claim that the frontier of $D''$ is the same as the frontier of $D'$. In particular, since $T[S^*]$ is a path, it is easy to see that matching $P$ can be performed without altering the edges that $P$ and $P$’s descendants contribute to $\text{Frontier}(D')$. Thus, since no other nodes are affected by the application of a template, the frontier of $D'$ is the same as the frontier of $D''$. Furthermore, by preserving the frontier, it is clear that the weak R-nodes of $D''$ are actually R-nodes. Thus, $D''$ is not only an orientation of $D^*$, it is also a PR-tree of $D^*$. In particular, $T = \text{Frontier}(D'') \in \text{Consistent}(D^*)$. \[ \square \]
(4) : Notice that, through the application of a P-node template, R'-nodes can become orphaned. Furthermore, when this happens, for such an R'-node Q, either Q specifies a subpath of an R'-node that is newly created (between Q and its P-node parent), or Q’s P-node parent will become an R'-node and Q will specify a subpath of this R'-node. Now, during our discussion of the templates we always discarded such Qs (since they are essentially redundant). However, since we would like to have a weak PR-tree that represents $S^*$, this is not always the correct course of action. In particular, when Q’s leaf set is a constraint in $S$, we need to keep Q. Furthermore, it is easy to track the nodes with leaf sets in $S$ since they are always created during the join operation of PRtreeReduction. More specifically, we can simply flag these nodes as “do not delete” during the join operation and check for this flag prior to discarding any such Q after applying a template. Other than these orphaned R'-nodes, we never delete nodes from $D$ (note: some P-nodes may become R'-nodes, but their leaf sets are unaffected). Thus, we have the desired property.

(5) : Consider any bridged child $N$ of $P$. As we have seen in the templates, $N$ will have one fewer pertinent P-node parent after $P$ has been matched. Furthermore, since $N$ can only have at most one weak R-node parent, $N$ will not be bridged after all of $N$’s pertinent P-node parents are matched.

(6) : Notice that, when $P$ is partial (i.e., pertinent but not full) and $P$ has no pertinent parents (i.e., all of $P$’s pertinent P-node parents have been matched) one of the templates P2a, P3a, P4a, or P5a will apply to $P$. Thus, $P$ will not be pertinent once it has been matched since it only has one child with full leaves and has no pertinent parent.

Furthermore, when $P$ has partial or blocked-full (i.e., pertinent but not full) children, $P$ will no longer be their parent after $P$ has been matched. As we have seen in the templates, these partial or blocked-full children must be R'-nodes and, consequently, will only remain pertinent when they have a pertinent parent (since LCSAs do not contain R'-nodes). Thus, after the pertinent P-node parents of a partial or blocked-full node $N$
have been matched, \( N \) will not have any pertinent P-node parents (i.e., \( N \) will not be pertinent or \( N \) will have a pertinent weak R-node parent).

This completes the proof of the P-node Template Lemma.

In light of property (1) of the P-node Template Lemma, all P-nodes which do not match one of the P-node templates are considered illegal. This lemma concludes our discussion of the P-node templates and will be useful in our proof of correctness of the \texttt{reduce} operation in section 5.3.4.

## A.2 Weak R-node Templates

The weak R-node templates will be presented with respect to a pertinent weak R-node \( \mathcal{R} \). A weak R-node template consists of three parts: one (labelled \((i)\)) to match \( \mathcal{R}_T \), one (labelled \((ii)\)) to match the pertinent \( \mathcal{R}' \)-nodes in \( \mathcal{R}_{R'} \), and one (labelled \((iii)\)) to match the trees of any relevant weak R-node (i.e., any weak R-node that will change through the matching process\(^\dagger\)). We separate these templates into groups based on properties of the children of \( \mathcal{R} \) similarly to the P-node template groups. Before doing this we first discuss several key observations that will aid in this presentation.

For a matched bridged node \( N \), the algorithm will insist on matching its pertinent P-node parent prior to its pertinent weak R-node parent \( \mathcal{R} \). By the P-node templates, this approach implies that \( N \) will no longer be bridged once we are attempting to match \( \mathcal{R} \). In other words, template matching for weak R-nodes will not need to worry about bridged children. Notice that, from the definition (3.1) of weak PR-trees, insisting on a “P-node before weak R-node” priority when matching the pertinent parents of a bridged node is a valid approach. Thus, no weak R-node templates will match \( \mathcal{R} \) when it has bridged children (i.e., no group of weak R-node templates corresponds to P2 or P5).

\(^\dagger\)Recall that each P-node template has an implicit side effect of altering weak R-node(s) related to the children of the P-node being matched.
Recall that the pertinent children of a weak R-node $\mathcal{R}$ must induce a path in $\mathcal{R}_T$ in order for $D$ to be reducible. We will refer to this path as $\mathcal{R}$’s *pertinent path* and use $\mathcal{R}_{PP}$ to denote it and we depict $\mathcal{R}_{PP}$ as an R'-node\(^8\). To match $\mathcal{R}_T$ we will instead match $\mathcal{R}_{PP}$. Additionally, (w.l.o.g.) we use $C$ and $C^*$ to denote the “left-most” and “right-most” nodes on $\mathcal{R}_{PP}$ respectively. When $\mathcal{R}_{PP}$ is a single node (i.e., in the degenerate case), we use $C$ to denote this node. Notice that the only nodes on this path that can be blocked-full R'-nodes are $C$ and $C^*$. In particular, if a closed child of $\mathcal{R}$ is a blocked-full R'-node, $D$ will be irreducible. Similarly, the endpoints of this path can also be partial. However, it is also possible (in a very restricted way) for internal nodes of $\mathcal{R}_{PP}$ to be partial (we discuss this in the final group of weak R-node templates). Thus, we will need weak R-node templates analogous to the templates from P-node groups P3 and P4. We will have three weak R-node groups corresponding to P4 (i.e., when one end of $\mathcal{R}_{PP}$ is partial, when both ends of $\mathcal{R}_{PP}$ are partial, and when $\mathcal{R}_{PP}$ has internal nodes which are partial). Also, when we are matching the trees of relevant weak R-nodes (in the third component of each weak R-node template), we will instead match their pertinent paths.

Since LCSAs do not contain R'-nodes, an R'-node is only pertinent when it is descendent from an element of the LCSA of $S^*$. Thus, every pertinent R'-node will have at least one pertinent parent. In particular, each pertinent R'-node $Q$ implies the existence of a full leaf (of its pertinent parent $N$ or an ancestor of $N$) which is not a leaf of $Q$ that must be adjacent to the frontier of $Q$ in any orientation $D'$ of $D$ where $S^*$ induces a path in the frontier of $D'$. Furthermore, in $D'$, since the frontier of $Q$ is contained in the frontier of $Q_R$, this full leaf is also adjacent to the frontier of $Q_R$. In this light we can see that there can be at most two pertinent R'-nodes in $\mathcal{R}_{R'}$ (otherwise, there would be three full leaves that would have to be adjacent to $\mathcal{R}$’s frontier and, consequently, $D$ would be irreducible). W.l.o.g. we will use $Q$ to denote $\mathcal{R}$’s first pertinent R'-node and

\(^8\)Note: we use an R'-node to depict a pertinent path simply for notational convenience, this does not mean such an R'-node will necessarily be present in the corresponding set of R'-nodes.
$Q^*$ to denote its second. Notice that, if $Q$ exists and is bridged then $Q$ must be the only pertinent $R'$-node in $R_{R'}$ (i.e., there is no $Q^*$).

Based on the above observations, we will separate the weak R-node templates into five groups. The first group consists of the case when $R$’s pertinent path only consists of full children. In the next group we add the possibility that $R_{PP}$ contains blocked-full $R'$-nodes. In the third group, both ends of $R_{PP}$ are partial nodes and all of $R_{PP}$’s internal nodes are full. The forth group considers the case when: one of $R_{PP}$’s ends is a partial node, the other end (when present) is either a full node or a blocked-full $R'$-node, and $R_{PP}$’s internal nodes are full. The final group describes the case when $R_{PP}$ has internal nodes which are not full with all possibilities for the ends of $R_{PP}$. Moreover, we separate each of these groups into subcases based on $R$’s pertinent $R'$-nodes.

A.2.1 Template Group R1: Full Pertinent Path

The first group of weak R-node templates concerns the case when every pertinent child of $R$ is full (i.e., when $R_{PP}$ only contains full nodes). In this group, the replacement does not change $R$ and simply marks $R_{PP}$ and its corresponding parts of $Q$ and $Q^*$ as full. Since we are not modifying any nodes in this case, we do not have the third component for the templates in this section (i.e., label (iii) is not used in the templates). Template $R1a$ (see fig. A.15) depicts the case when $R$ has two pertinent $R'$-nodes (i.e., the first subcase for this group).

Notice that, in template $R1a$ we insist that each pertinent $R'$-node has a distinct one-sided child as one of its endpoints (i.e., $Q$ has $C$ as its left-most endpoint and $Q^*$ has $C^*$ as its right-most endpoint). This is required since we must ensure that the pertinent parent of $Q$ (respectively of $Q^*$) has the opportunity to be matched. In particular, the parent of a pertinent $R'$-node can only be matched when that $R'$-node has at least one one-sided endpoint (i.e., as in P-node template groups P3 and P4, and as we will see in template groups R2 and R3 below).
Appendix A. Reduction: Template Matching

Pattern:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(ii)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Pattern Diagram" /></td>
<td><img src="image2.png" alt="Pattern Diagram" /></td>
</tr>
</tbody>
</table>

Replacement:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(ii)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Replacement Diagram" /></td>
<td><img src="image4.png" alt="Replacement Diagram" /></td>
</tr>
</tbody>
</table>

Figure A.15: Template R1a / R1b / R1c: matching a weak R-node with two / one (non-bridged) / zero pertinent R'-node(s) and whose pertinent path only contains full nodes.

The degenerate form of template R1a (i.e., when $R$ only has one pertinent child) is given in fig. A.16.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(ii)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5.png" alt="Degenerate Pattern Diagram" /></td>
<td><img src="image6.png" alt="Degenerate Pattern Diagram" /></td>
</tr>
</tbody>
</table>

Figure A.16: Template R1a / R1b / R1c with a degenerate pertinent path.

There are two special cases of template R1a which will require extra care (one based on the general case in fig. A.15 and one based on the degenerate case in fig. A.16).

In the first special case (w.l.o.g.) $Q$ is a full R'-node whose endpoints are precisely $C$ and $C^*$ (i.e., $Q$’s path in $R_T$ is precisely $R_{PP}$) and $Q^*$ is as in the general case (note: both $Q$ and $Q^*$ cannot be $R_{PP}$ since each node must be unique). Notice that, once template R1a is applied $Q$ will appear to be accessible-full to its parent and $Q^*$ will either be partial or blocked-full to its parent. However, from templates P3a, P3b, P4a,
and P4b, and in the templates later on in this section we can and will see that $C^*$ will eventually gain a pertinent neighbour due to $Q^*$ being partial (or blocked) and having a pertinent parent. In particular, once $C^*$ gains this pertinent neighbour, $Q$ will become blocked. With this in mind we set $Q$ to be blocked immediately rather than waiting for it to happen naturally “higher-up” in the template matching process. More specifically, from $Q$’s perspective, $C^*$ is treated as closed and $C$ is one-sided (i.e., $C$ is unchanged). This will result in the proper template matching as we proceed up the tree.

The second is a special case of the degenerate form of R1a. In particular, when the pertinent child is a leaf we encounter a similar problem. Notice that both $Q$ and $Q^*$ will appear to be $R'$-nodes with a single full child that is an open leaf. However, since $Q$ has a pertinent parent, $C$ will eventually gain a pertinent neighbour via one of either templates P4a or P4b, or one of the weak R-node templates from the weak R-node groups below (i.e., from $Q^*$’s perspective, $C$ should be considered one-sided). Symmetrically, due to $Q^*$, $C$ will also eventually gain a pertinent neighbour. Under these considerations we immediately set $C$ to be one-sided from the perspective of both $Q$ and $Q^*$. This will ensure that this case is properly handled during the template matching process.

The second subcase of this group concerns the situation when $\mathcal{R}$ has exactly one pertinent $R'$-node $Q$ and $Q$ is not bridged (i.e., when $Q$ is not bridged and $Q^*$ does not exist). Notice that, if we simply exclude $Q^*$ from template R1a and its degenerate form, we have the appropriate template for this case. We will refer to the template for this case as template R1b (note: we do not provide a separate figure for this since it is clear from template R1a - i.e., fig. A.15). Additionally, the absence of $Q^*$ means that we do not have to worry about any special cases similar to those of template R1a.

In the third subcase $\mathcal{R}$ has no pertinent $R'$-nodes. Similarly to template R1b, the template for this case simply consists of template R1a after excluding both $Q$ and $Q^*$. We will refer to the template for this case as template R1c (note: we do not provide a separate figure for this since it is clear from template R1a - i.e., fig. A.15).
In the final subcase, $Q$ is bridged and (as in R1b) $Q^*$ does not exist. To examine this case we must first determine when a pertinent $R'$-node $Q$ can be bridged. There are only two ways in which $Q$ can be bridged. The first is when all of $Q$’s children are full. In this case we must be careful to insist that the “leftmost” child and “rightmost” child of $Q$ are each one-sided (i.e., $Q$ must be accessible). Clearly both of these children are at least one-sided (since all of $Q$’s children are full). However, if one of them was closed, there would be three full leaves (separate from the leaves of $Q$) that would need to be adjacent to the leaves of $Q$ (i.e., $D$ would be irreducible). Notice that, when $Q$ is full it will be precisely the pertinent path of its weak R-node. Template R1d (see fig. A.17) handles the subcase when $Q$ is full and bridged.

![Pattern and Replacement Diagram](image)

Figure A.17: Template R1d: matching a weak R-node $\mathcal{R}$ with exactly one pertinent $R'$-node $Q$, where $Q$ is bridged and $\mathcal{R}_{pp}$ only contains full nodes.

The other way that $Q$ can be bridged is when $Q$ is partial (i.e., $Q$ is bridged-partial). Similarly to template P2c (see fig. A.6), this bridged-partial node must be an $R'$-node with empty children and exactly one full child which is an open leaf. In particular, this second case corresponds to a specific degenerate form of $\mathcal{R}_{pp}$ (i.e., when it is a single full leaf). Fig. A.18 depicts the degenerate form of template R1d to handle this case.

### A.2.2 Template Group R2: Blocked-Full Ends

The second group of weak R-node templates concerns the case when $\mathcal{R}_{pp}$ can have blocked-full $R'$-nodes. Recall that the only valid locations for these blocked-full $R'$-node children are at the endpoints of $\mathcal{R}_{pp}$ (i.e., $C$ and $C^*$). We will have two subcases in this
Figure A.18: Template R1d: degenerate case.

group of templates (i.e. when $C$ is blocked-full and $C^*$ is full, and when both $C$ and $C^*$ are blocked-full). This group of templates will work similarly to template group P3. In particular, we must force the one-sided child of each blocked-full child to be adjacent to the full portion in $R_{PP}$. Notice that, each pertinent $R'$-node must either have an open child or a one-sided child (which is not shared with the other pertinent $R'$-node) as one of its endpoints after replacement. Thus, there will be at most one pertinent $R'$-node in the templates of this group (i.e., the first subcase will have at most one pertinent $R'$-node and the second will not have any). Template R2a demonstrates the general form of the first subcase (i.e., when $R$ has one blocked-full child and $Q$ does not have $C$ as a child). Recall that $C_R$ indicates the weak R-node associated with $C$. With this in mind, we use $(C_R)_{PP}$ to indicate the pertinent path of $C_R$.

At a high level this template operates very similarly to template P3a. In particular, we are moving $R$’s full pertinent children into a new $R'$-node so that they are forced to be adjacent to the one-sided child $Y$ of $R$’s blocked-full child $C$ (just as we did for $P$’s full pertinent children). However, the details of implementing such a simple sounding operation on a weak R-node are quite complicated (see fig. A.20 for a more detailed template describing the changes to $R$ and $C_R$). We will refer to operations similar to the one depicted in fig. A.20 as moving the relevant elements of $R$.

To execute this template, in $R_T$, we will first disconnect $N_0$ from $C$ (the blocked-full $R'$-node child of $R$) and then, in $(C_R)_T$, we connect $N_0$ to $Y$ (the one-sided child of $C$). This will result in “moving” a portion of $R_T$ into $(C_R)_T$. In particular, after “cutting”
the edge $CN_0$ out of $R_T$, the component $H_0$ which contains $N_0$ will be moved from $R_T$ to $(C_R)_{T}$. We will also have to move R'-nodes from $R_{R'}$ into $(C_R)_{R'}$ (i.e., every R'-node $Q^{**}$ that contains a node of $H_0$ as a child will need to be moved). This operation will appropriately move $Q$ (among other R'-nodes) from $R_{R'}$ to $(C_R)_{R'}$. This constitutes the first step of the process of moving the relevant elements of $R$.

Now, consider an R'-node $Q_1$ with $N_0$ as a child. Since $N_0$ belongs to $H_0$, we will have moved $Q_1$ from $R_{R'}$ to $(C_R)_{R'}$. However, if $Q_1$ also has a child $N_1$ which is not in $H_0$ (for example, $N_1$ is a neighbour of $C$ other than $N_0$), then we will have more work to do in order to complete this operation. In particular, we will need to move $N_1$ from $R$ to $C_R$ using a similar approach to how we moved $N_0$. To do this we first disconnect $N_1$ from $C$ (in $R_T$) and then, in $(C_R)_T$ we connect $N_1$ to $Z$ (the closed endpoint of $C$). We note the following two points:

1. Since $N_0$ is connected to $Y$, if we connect $N_1$ to $Y$, $Q_1$ will not be a path in $(C_R)_T$.

2. $N_1$ must be an empty node since it is not present in $R$’s pertinent path. Thus, by making $N_1$ adjacent to $Z$ we are not affecting the potential for $D$ to be reducible.
Appendix A. Reduction: Template Matching

Pattern:

Also, \( Q_1, Q_2, \ldots, Q_k \in \mathcal{R}_R \) where:

Replacement:

Also: \( Q_1, Q_2, \ldots, Q_k \) (among others); are moved from \( \mathcal{R}_R \) to \( (C_R)_R \) and each \( Q_i \) is replaced as follows:

Figure A.20: Demonstrating the movement of relevant elements from \( \mathcal{R} \) to \( C_R \) involved in template R2a. \( \mathcal{R}_{PP} \) is depicted by the bold path from \( C \) to \( C^* \) in the pattern and \( \mathcal{R}_{PP} \) is the single node \( C \) in the replacement. Similarly, \( (C_R)_{PP} \) is depicted by the bold path from \( V \) to \( Y \) in the pattern and \( V \) to \( C^* \) in the replacement. Note: this demonstrates the case when \( k \) is even (i.e., when \( N_k \) becomes attached to \( Y \)).
Once again, we have “cut” part of $\mathcal{R}_T$ and moved it into $(C_R)_T$. More specifically, the component $H_1$ (of $\mathcal{R}_T$ after removing the edge $CN_1$) that contains $N_1$ is moved from $\mathcal{R}_T$ to $(C_R)_T$. We then move the corresponding R'-nodes (i.e., every R'-node $Q^{**}$ that contains a node of $H_1$ as a child will need to be moved). We perform this operation for every such $Q_1$.

However, since more R'-nodes have been moved (due to moving $H_1$), we might end up moving an R'-node $Q_2$ which is a parent of $C$, some $N_1$, and a node $N_2$, where $N_2$ is still in $\mathcal{R}_T$ (i.e., w.l.o.g. $N_2$ is adjacent to $C$ in $\mathcal{R}_T$). In this case we disconnect $N_2$ from $C$ (in $\mathcal{R}_T$) and attach $N_2$ to $Y$ in $(C_R)_T$ (note: we must attach $N_2$ to $Y$, $Q_2$ must be a path and $N_1$ is already attached to $Z$). This will move the component $H_2$ (of $\mathcal{R}_T$ after removing the edge $CN_2$) that contains $N_2$ from $\mathcal{R}_T$ to $(C_R)_T$ and the corresponding R'-nodes from $\mathcal{R}_R$ to $(C_R)_R$. We continue this process until we have reached the point where the children of every R'-node that has been moved to $(C_R)_R$ are contained in the children of $C_R$ (i.e., once we have moved the component $H_k$ as in fig. A.20). At this point $\mathcal{R}_T$ will have been reduced to the component $H$ (as in fig. A.20) and $(C_R)_T$ will have gained the components $H_0$, ..., $H_k$ attached via the appropriate edges.

There is one final issue we must handle regarding this operation. Notice that, when $i$ is even $N_i$ will be connected to $Y$ (the one-sided child of $C$) and when $i$ is odd $N_i$ will be connected to $Z$ (the closed child of $C$). Thus, if one of the R'-nodes $Q^{**}$ that gets moved as a result of this process has some $N_i$ and some $N_j$ as children and $i$ and $j$ have the same parity (i.e., both are connected to the same end of $C$), $Q^{**}$ will not satisfy the definition of R'-nodes (since its children will not form a path in $Q_R$). In particular, this means that $D$ is irreducible since, if $S^*$ induces a path in the frontier of an orientation $D'$ of $D$, then $L(Q^{**})$ (i.e., a PT-constraint of $S_D$) does not induce a path in the frontier of $D'$. Furthermore, when such a problem occurs during the attempted application of a template, we halt indicating that “no template applies” to this weak R-node.

This type of operation (namely, moving the relevant elements of $\mathcal{R}$) will be used in all
of the remaining weak R-node templates (i.e., in R2b, group R3, and group R4). Once
this operation has completed $\mathcal{R}$ is no longer pertinent since it does not have any pertinent
R’-nodes and only has one pertinent child. Furthermore, since $C$’s only pertinent parent
(i.e., $\mathcal{R}$) loses its pertinent status, $C$ will also lose its pertinent status. In fact, we will see
that each blocked-full R’-node child of $\mathcal{R}$ will no longer be pertinent after the replacement.
Recall that the P-node templates regarding blocked-full R’-node children worked similarly
(i.e., after replacement the blocked-full R’-nodes were no longer pertinent). Additionally,
when $\mathcal{R}_R$ is left with only the single node $C$ after $\mathcal{R}$’s relevant elements have been moved,
we will delete $\mathcal{R}$. Furthermore, if $\mathcal{R}$ was $C$’s only remaining parent, then $C$ we also delete
$C$ since it is now a redundant subpath of $Q_0$. This ensures that no nodes become orphaned
during this operation (i.e., no nodes become sources of the working weak PR-tree).

We now consider an alternate form and a degenerate form of template R2a. Notice
that, $Q$ could also have $C$ as a child. This leads to the alternate form of R2a given in
fig. A.21. Additionally, when $C$ is the only node on $\mathcal{R}_{PP}$, we have the degenerate case
form of this alternate case (see fig. A.22).

![Diagram](image)

Figure A.21: Template R2a: the alternate form of $Q$ (i.e., when $Q$ has $C$ as a child).
Note: (i) and (iii) are omitted since they are the same as in fig. A.19.

Notice that, in the degenerate form of R2a, $C$ has no neighbours in the pertinent
path. In particular, there is no neighbour $N_0$ of $C$ (as in the general case) from which
to start the process of moving the relevant elements of $\mathcal{R}$ to $C_R$. However, we note that
$Q$ must be moved from $\mathcal{R}_R$ to $(C_R)_R$ since $Q$ must have a one-sided endpoint in order
to accommodate its pertinent parent (i.e., $Y$ becomes that endpoint in the replacement).
Additionally, $Q$’s “right-most” empty child $W$ will become adjacent to $Z$ (the closed child of $C$). Thus, in the degenerate case, we do not have an $N_0$, but the role of $N_1$ will be played by $W$ (since, like $N_1$, $W$ becomes adjacent to $Z$). From here the process proceeds as before. As in the standard case, $R_T$ may end up with only a single node. In this case we will delete $R$, and when $C$ becomes orphaned we will also delete $C$ since it will be a subpath of $Q$. This will ensure that no nodes become sources during this operation.

The other case in this group concerns the situation when $R$ has two blocked-full $R'$-node children. In this case $R$ cannot have pertinent $R'$-nodes (as noted earlier). Template $R2b$ (see fig. A.23) performs the appropriate action for this case.

Once again, we will need to move the relevant elements of $R$. This operation (depicted by the detailed template in fig. A.24) requires a little more care since there are now three weak $R$-nodes (i.e., $R$, $C_R$, and $C_R^*$) which are involved.

Notice that, after the replacement we have a path connecting the one-sided child $Y$ of $C$ to the one-sided child $Y^*$ of $C^*$ (i.e., as in (iii) in the replacement of fig. A.23). This path means that we must merge $C_R$ and $C_R^*$ to create a new weak $R$-node which we will now refer to as $R^*$. In particular, $R^*$ is the union of $C_R$ and $C_R^*$ via the full
## Appendix A. Reduction: Template Matching

**Pattern:**

<table>
<thead>
<tr>
<th>(i)</th>
<th>(ii)</th>
<th>(iii)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Pattern Diagram" /></td>
<td><img src="image2" alt="Pattern Diagram" /></td>
<td><img src="image3" alt="Pattern Diagram" /></td>
</tr>
</tbody>
</table>

**Replacement:**

<table>
<thead>
<tr>
<th>(i)</th>
<th>(ii)</th>
<th>(iii)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4" alt="Replacement Diagram" /></td>
<td><img src="image5" alt="Replacement Diagram" /></td>
<td><img src="image6" alt="Replacement Diagram" /></td>
</tr>
</tbody>
</table>

Figure A.23: Template R2b: matching a weak R-node with two blocked-full R′-node children (note: we omit (ii) since this pattern requires the absence of pertinent R′-nodes).

Subpath of $R_{PP}$ (i.e., the path from $N_0$ to $N_0^*$). More specifically, in $R_T$, we “cut” the edges $CN_0$ and $C^*N_0^*$, then, in $(C_R)_T$, we connect $N_0$ to Y and, in $(C^*_R)_T$, we connect $N_0^*$ to $Y^*$ (note: in the degenerate case we “cut” the edge between $C$ and $C^*$ out of $R_T$ and simply connect $Y$ to $Y^*$). This process will connect $(C_R)_T$ and $(C^*_R)_T$ via the component $H_0$ (of $R_T$ after removing the edges $CN_0$ and $C^*N_0^*$) which contains both $N_0$ and $N_0^*$. Furthermore, $R^*_T$ will be connected and will contain $(C_R)_T$, $(C^*_R)_T$, and $H_0$ as subtrees together with the edges $YN_0$ and $N_0^*Y^*$) and leave $R_T$ with two components: $(R_T)_1$ (containing $C$) and $(R_T)_2$ (containing $C^*$). Additionally, we initialize $R^*_{R'}$ to be the union of $(C_R)_{R'}$, $(C^*_R)_{R'}$, and every R′-node in $R_{R'}$ with a child in $H_0$ (note: in the degenerate case, we use every R′-node with both $C$ and $C^*$ as children).

From here we proceed as before, first with respect to $C$ and $N_0$ (i.e., looking for $Q_1$’s whose children are spread across $(R_T)_1$ and $R^*$, then $Q_2$’s, etc.), then with respect to $C^*$ and $N_0^*$ (i.e., looking for $Q_1^*$’s whose children are spread across $(R_T)_2$ and $R^*$, then $Q_2^*$’s,
Appendix A. Reduction: Template Matching

**Pattern:**

Also, $Q_1, \ldots, Q_k, Q_1^*, \ldots, Q_k^* \in \mathcal{R}_{R'}$ where:

**Replacement:**

Also: $Q_1, \ldots, Q_k, Q_1^*, \ldots, Q_k^*$ (among others); are moved from $\mathcal{R}_{R'}$ to $(C_R)_{R'}$ and each $Q_i$ and $Q_j^*$ are replaced as follows:

Figure A.24: Demonstrating the splitting of $\mathcal{R}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$, joining of $C_R$ and $C^*_R$, and the movement of relevant elements from $\mathcal{R}$ to $C_R$ and $C^*_R$ involved in template R2b. We again use bold paths to depict the pertinent paths. Note: this demonstrates the case when both $k$ and $k^*$ are even (i.e., when $N_k$ becomes attached to $Y$ and $N_k^*$ becomes attached to $Y^*$).
etc.). During this process we will have removed nodes from the trees \((R_T)_1\) and \((R_T)_2\) as they are moved into \(R^*_T\).

We are now ready to split \(R\) into two new weak R-nodes \(R_1\) (containing \(C\)) and \(R_2\) (containing \(C^*\)) according to the trees \((R_T)_1\) and \((R_T)_2\) (i.e., \(H\) and \(H^*\) respectively in fig. A.24). Notice that, the \(R'\)-nodes that remain in \(R\) will easily partition with respect to these trees since we have moved any \(R'\)-nodes which might cross between them when creating \(R^*\). After this procedure has completed we will have merged \(C_R\) and \(C^*_R\) into the new weak R-node \(R^*_R\) (i.e., \(R^*_R = C_R = C^*_R\) and \(R\) will be split into two weak R-nodes: \(R_1\) (containing \(C\)) and \(R_2\) (containing \(C^*\))). Furthermore, neither \(R_1\) nor \(R_2\) will be pertinent with respect to \(S^*\) since each will only have one pertinent child and no pertinent \(R'\)-nodes. Consequently, neither \(C\) nor \(C^*\) remain pertinent post-replacement.

Additionally, when \((R_1)_T\) (\((R_2)_T\) respectively) is left with only the single node \(C\) (\(C^*\)) after \(R\)'s relevant elements have been moved, we will delete \(R_1\) (\(R_2\)). Furthermore, if \(R_1\) (\(R_2\)) was \(C\)'s (\(C^*\)'s) only remaining parent, then we will also delete \(C\) (\(C^*\)) since it is now a redundant subpath of \(Q_0\) (\(Q^*_0\)). This ensures that no nodes become orphaned during this operation (i.e., no nodes become sources of the working weak PR-tree).

### A.2.3 Template Group R3: Two Partial End-Children

The third group of weak R-node templates concerns the case when \(R_{PP}\) contains exactly two partial nodes and these partial nodes are the endpoints of \(R_{PP}\) (i.e., \(C\) and \(C^*\)). We will need to force the one-sided full child of each partial end-child of \(R\) to be adjacent to the full portion of \(R_{PP}\) just as we did in template R2b. Some additional cases arise in this group since each of \(C\) and \(C^*\) might now have an open child (i.e., when \((C_R)_{PP}\) and/or \((C^*_R)_{PP}\) consists of a single full node) as opposed to being forced to have a one-sided child (as in group R2).

We will present this group in three subcases. These include: when \(R\) has zero, exactly one (non-bridged), and exactly two pertinent \(R'\)-node(s) respectively. Notice that we do
not have a subcase in which $\mathcal{R}$ has a bridged pertinent $R'$-node. In particular, as we have already seen in template R1d, a bridged pertinent $R'$-node would either need to be accessible or have a single full child which is an open leaf after the replacement. Neither of these cases are possible when $\mathcal{R}$ has two partial end-children (this will become clear as we discuss this group further).

*Template R3a* (see fig. A.25) handles the case when $\mathcal{R}$ does not have any pertinent $R'$-nodes. This template has the most general form for the pertinent paths (i.e., $\mathcal{R}_{PP}$, $(C_{R})_{PP}$, and $(C'_{R})_{PP}$) in this group of templates.

This template bears a close resemblance to template R2b. In particular, after the replacement we will have appropriately forced the one-sided full children (i.e., $Y$ and $Y^*$) of each partial node to be consecutive with the full nodes from $\mathcal{R}_{PP}$ (i.e., $Y$ will be adjacent to $N_0$ and $Y^*$ will be adjacent to $N_0^*$). Furthermore, we will need to move...
relevant elements of $\mathcal{R}$ as we did in template R2b. Notice that, we have labelled the empty endpoint of $C$ and the empty endpoint of $C^*$ as $Z$ and $Z^*$ respectively. This labelling corresponds to the labelling of the closed full endpoints of $C$ and $C^*$ as used in template R2b. More specifically, the empty endpoints of $C$ and $C^*$ in template R3a play the role that the closed full endpoints of $C$ and $C^*$ did in template R2b (i.e., the “odd components” will get attached to $Z$ and $Z^*$ just as they did previously). During this process we will once again split $\mathcal{R}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$ and merge $C_R$ and $C^*_R$. Similarly to template R2b, neither $\mathcal{R}_1$ nor $\mathcal{R}_2$ are pertinent and, consequently, $C$ and $C^*$ lose their pertinent status. Through the remaining templates in this section we will see that this is the case for all partial end-children of a weak R-node.

In the second and third subcases of this group we consider the situation when $\mathcal{R}$ has pertinent non-bridged R'-node(s). Recall that, in order to accommodate a pertinent R'-node $Q$ we must be able to accommodate $Q$’s pertinent parent. In particular, after the replacement, $Q$ must have a one-sided full endpoint. The only candidates for such a one-sided full endpoint are $Y$ and $Y^*$ (since these are the only full nodes that are not closed and can become an endpoint of $Q$ after replacement). Furthermore, both $Y$ and $Y^*$ will each gain a pertinent neighbour after the replacement (i.e., $N_0$ and $N^*_0$ respectively). Thus, in the pattern, (w.l.o.g.) $C$ will need to be the “right-most” child of $Q$ and $Y$ will need to be open in order for $Q$ to be a pertinent R'-node of $\mathcal{R}$ (and $D$ to be reducible). Similarly, if $\mathcal{R}$ has two pertinent R'-nodes $Q$ and $Q^*$, then both $Y$ and $Y^*$ will need to be open. Template R3b (see fig. A.26) and template R3c (see fig. A.27) handle the cases when $\mathcal{R}$ has exactly one and exactly two pertinent R'-nodes respectively.

Notice that, in template R3b, $Q$ does not have $N_0$ as a child. This means that $Q$ might not be moved with the other relevant elements of $\mathcal{R}$ (i.e., when “starting” from $N_0$). Recall that, in the degenerate case of template R2a (see fig. A.22) we moved the relevant elements of $\mathcal{R}$ by treating the empty child $W$ of $Q$ as $N_1$ and proceeding as usual from there. In this case we will act similarly (i.e., treating $W$ as just another $N_1$).
Appendix A. Reduction: Template Matching

**Pattern:**

(i) $R_{pp}$

(ii) $Q$

(iii) $(C_n)_{pp}, (C^*_{n})_{pp}$

**Replacement:**

(i) $(R_1)_{pp}$

(ii) $Q$

(iii) $(C_n)_{pp}, (C^*_{n})_{pp}$

Figure A.26: Template R3b: matching a weak R-node with two partial R'-node children and one pertinent R'-node.

Furthermore, we will also need to do this with respect to template R3c (i.e., we will have $W$ and $W^*$ which will be treated as an extra $N_1$ and $N_1^*$ respectively).

Notice that, in every template in this group, $R$ is split into two non-pertinent weak R-nodes $R_1$ and $R_2$ and $R$’s partial end-children $C$ and $C^*$ lose their pertinent status.

### A.2.4 Template Group R4: Exactly One Partial End

The fourth group of weak R-node templates concerns the case when exactly one endpoint of $R_{pp}$ is partial and $R_{pp}$ contains full and blocked-full nodes. Many of the subcases
Figure A.27: Template R3c: matching a weak R-node with two partial R′-node children and two pertinent R′-nodes.

in this group are simply combinations of templates we have already seen and, as such, we will not present explicit templates for them. The templates in this group will operate very similarly to template R2a and to the templates from group R3. In particular, the partial end-child will (w.l.o.g.) be the “left” endpoint of $R_{PP}$ (i.e., $C$) and we will need to force $C$’s one-sided full child $Y$ to be adjacent to the full portion of $R_{PP}$ just as we did in group R3. Since we only insist that one child (i.e., $C$) of $R$ is partial, $R_{PP}$’s other endpoint $C^*$ may be either full or blocked-full.

This group will be discussed in four subcases. These include when $R$ has precisely:

(a) Zero pertinent R′-nodes.
Appendix A. Reduction: Template Matching

(b) One (non-bridged) pertinent $R'$-node.

(c) Two pertinent $R'$-nodes.

(d) One bridged pertinent $R'$-node.

In the first subcase (i.e., when $R$ has no pertinent $R'$-nodes), $C$ will be a partial $R'$-node (as in template R3a - see fig. A.25) and $C^*$ will either be full (as in template R2a - see fig. A.19) or a blocked-full $R'$-node (as in template R2b - see fig. A.23). In particular, template $R4a$ is easily built from templates R3a, and either R2a or R2b depending on the choice of $C^*$.

For the second subcase (i.e., when $R$ has exactly one (non-bridged) pertinent $R'$-nodes), there are two possibilities for $R$. Notice that, similarly to template R2a (see fig. A.19) we gain the ability to accommodate a pertinent $R'$-node simply by having $C^*$ as a full node. Thus, when $C$ is a partial $R'$-node (as in template R3a - see fig. A.25) and $C^*$ is full (as in template R2a - see fig. A.19), $Q$ can take on forms analogous to those in template R2a (i.e., including all three different forms of $Q$ associated with template R2a - see figures A.19, A.21, A.22). Alternatively, when $C$ is a degenerate partial $R'$-node (as in template R3b - see fig. A.26), $Q$’s pattern and replacement will match that of template R3b (regardless of the choice of $C^*$). Thus, template $R4b$ will consist of $C$ being partial and $C^*$ being full or absent with appropriate alternate forms for $Q$.

The general form of third subcase (i.e., when $R$ has exactly two pertinent $R'$-nodes) is just a combination of the general and degenerate forms of template R4b. In particular, in template $R4c$, $C$ will be a degenerate partial $R'$-node (as in template R3b - see fig. A.26) and $C^*$ will be full. Notice that, this will allow $Q$ to have $C$ as its “right-most” child and $Q^*$ to have $C^*$ as its “right-most” child. Notice that, a degenerate form (see fig. A.28) of $R$ associated with this template (i.e., when $R_{PP}$ is the single node $C$ and $(C_R)_{PP}$ is the single node $Y$) will also be able to accommodate two pertinent $R'$-nodes. This template is very similar to template P5b (i.e., when $P$ has a bridged-partial end-child - see fig.
Appendix A. Reduction: Template Matching

A.14). In particular, \( C \) is a partial node which is a child of both \( Q \) and \( Q^* \).

**Pattern:**

\[
\begin{array}{c|c|c}
(i) & (ii) & (iii) \\
\begin{array}{c}
C \\
\end{array} & \begin{array}{c}
Q \\
\end{array} & \begin{array}{c}
Q^* \\
\end{array} \\
\begin{array}{c}
Z \\
\end{array} & \begin{array}{c}
W \\
\end{array} & \begin{array}{c}
W^* \\
\end{array} \\
\begin{array}{c}
Y \\
\end{array} & \begin{array}{c}
\text{open full leaf} \\
\end{array} & \begin{array}{c}
\text{open full leaf} \\
\end{array} \\
\end{array}
\]

**Replacement:**

\[
\begin{array}{c|c|c}
(i) & (ii) & (iii) \\
\begin{array}{c}
C \\
\end{array} & \begin{array}{c}
Q \\
\end{array} & \begin{array}{c}
Q^* \\
\end{array} \\
\begin{array}{c}
Z \\
\end{array} & \begin{array}{c}
W \\
\end{array} & \begin{array}{c}
W^* \\
\end{array} \\
\begin{array}{c}
Y \\
\end{array} & \begin{array}{c}
\text{open full leaf} \\
\end{array} & \begin{array}{c}
\text{open full leaf} \\
\end{array} \\
\end{array}
\]

Figure A.28: Template R4c (degenerate form): matching a weak R-node with one partial R'-node child and two pertinent R'-nodes.

Notice that, similarly to template P5b (see fig. A.14), we will insist that the open full child (i.e., \( Y \)) of the “shared” partial end-child (i.e., \( C \)) is a leaf. Furthermore, after the replacement, \( Q \) and \( Q^* \) will be in the situation described by the special case of the degenerate form of template R1a (i.e., \( Q \) and \( Q^* \) have a common endpoint \( Y^{††} \) which is an open full leaf). Recall that, we need to be careful about how \( Y \) is treated since it is guaranteed to gain one pertinent neighbour “higher up” in the template matching process for each of \( Q \) and \( Q^* \) (since both \( Q \) and \( Q^* \) have pertinent parents). With this in mind, after the replacement we treat \( Y \) as one-sided with respect to each of \( Q \) and \( Q^* \). This will ensure that the template matching proceeds properly as we continue up \( D \).

We now present the final subcase of group R4 (i.e., when \( R \) has exactly one pertinent R'-node \( Q \) and \( Q \) is bridged). This case is very similar to template P4c (i.e., when \( P \) is bridged and has a partial end-child - see fig. A.12) and the degenerate case of template R1d (see fig. A.18). In particular, the partial end-child must have an endpoint which is a single full child and that child is a full leaf. Template \( R4d \) (see fig. A.29) handles this

\[††Y \text{ corresponds to } C \text{ in template R1a (see fig. A.15).}\]
### A.2.5 Template Group R5: Partial Internal-Children

While it may seem surprising at first, there is a very specific instance in which we can successfully proceed when we encounter a partial node which is internal to $\mathcal{R}_{PP}$. In fact, there is no limit on the number of these “special” partial nodes which can occur on $\mathcal{R}_{PP}$. Fortunately, we will see that the templates we have developed in the first four groups can be easily adapted to handle these partial internal nodes. In particular to match a weak R-node with partial internal-children consists of two steps. In the first step we apply template R5 (see fig. A.30) to $\mathcal{R}$ which merges $\mathcal{R}$’s pertinent path with the pertinent paths of $\mathcal{R}$’s partial internal-children (note: this does not match $\mathcal{R}$’s pertinent R’-nodes).

This will result in a new pertinent path with $C$ and $C^*$ as its endpoints and full children between them. We then attempt to apply a template from groups R1-R4 to the weak

---

**Pattern:**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) $\mathcal{R}_{pp}$</td>
<td>(i) $\mathcal{R}_{pp}$</td>
</tr>
<tr>
<td>(ii)</td>
<td>(ii)</td>
</tr>
<tr>
<td>(iii)</td>
<td>(iii)</td>
</tr>
</tbody>
</table>

Figure A.29: Template R4d: matching a weak R-node with one partial R’-node child and one bridged pertinent R’-node.

Notice that, in every template in this group, after the replacement, $\mathcal{R}$ (or $\mathcal{R}_1$ and $\mathcal{R}_2$ when $\mathcal{R}$ is split) is not pertinent and $\mathcal{R}$’s partial end-child $C$ loses its pertinent status. Furthermore, when $\mathcal{R}$ has a blocked-full child $C^*$, it loses its pertinent status.
Appendix A. Reduction: Template Matching

To justify the pattern of template R5 we consider a partial surrounded (i.e., internal) node $Q_i$ on $R_{pp}$; i.e., $Q_i$ is partial and strictly between $C$ and $C^*$ on $R_{pp}$. Additionally, we let $N_i^L$ and $N_i^R$ be the left neighbour and right neighbour of $Q_i$ on $R_{pp}$ respectively; i.e., as in fig. A.30 (note: it is possible that $N_i^L = Q_i - 1$ when $i > 0$ and it is possible that $N_i^R = Q_i + 1$ when $i < k - 1$). Since $Q_i$ is surrounded, its structure must be very specific if $D$ is going to be reducible (i.e., this is similar to P-node template P4c where $P$ is bridged and has a partial child). Note: $Q_i$ is necessarily a R′-node since it is partial, matched, and has a pertinent parent (namely, $R$). Furthermore, since $R$ is eligible, $N$ is finished (i.e., $Q_i$ is not bridged). Finally, we claim that $Q_i$ has precisely one full child, and this full child is: a full leaf, an endpoint of $Q_i$, and open (i.e., has no pertinent neighbours in $N_R$). In particular, if $Q_i$ has at least two full leaves $u$ and $u^*$, then, since $Q_i$ is partial and
surrounded the following is implied. In the frontier of any orientation \( D' \) of \( D \) w.l.o.g. \( u \) would have three full neighbours (i.e., one leaf of each pertinent neighbour of \( Q_i \) in \( \mathcal{R}_{PP} \) and w.l.o.g. \( u^* \)). Thus, \( D \) would be irreducible. Thus, in order for \( D \) to be reducible, each \( Q_i \) must be as depicted in template R5 (i.e., fig. A.30).

However, there is an additional restriction that we require on each \( Q_i \) which is not depicted in template R5. In particular, it is necessary that no \( R' \)-node parent of \( Q_i \) has both \( N_i^L \) and \( N_i^R \) as children. This is easy to see, since such a parent \( Q \) would imply the following. In the frontier of any PR-tree \( D' \) of \( D \) (note: since \( D' \) is a PR-tree \( \text{Frontier}(D')[L(Q)] \) is a path) w.l.o.g. one full leaf of \( N_i^L \) is adjacent to the “full end” of \( Q_i \) and one full leaf of \( N_i^R \) is adjacent to the “empty end” of \( Q_i \) (i.e., \( \text{Frontier}(D')[S^*] \) is not a path).

With this restricted form for each \( Q_i \), the fact that they are surrounded immediately justifies our choice of replacement (note: as usual when applying the replacement we need to move the relevant elements of \( \mathcal{R} \)). In particular, this replacement follows similarly to template R3c.

Notice that, after the replacement associated with template R5, none of \( \mathcal{R}_0, \ldots, \mathcal{R}_{k-1} \) are pertinent (since any pertinent \( R' \)-nodes are necessarily moved to \((Q_0)_R \) by the movement of relevant elements). Furthermore, each of \( \mathcal{R}' \)’s partial internal-children (i.e., \( Q_0, Q_1, \ldots, Q_{k-1} \)) loses its pertinent status.

Finally, we note that once template R5 has been applied we will still try to match \((Q_0)_R \) against one of the template from R1-R4. Alternatively, if one were inclined, one could cross-breed template R5 with all previous weak R-node templates so that each pertinent weak R-node is guaranteed to only be processed by one template (rather than at most two).
A.2.6 The Weak R-node Template Lemma

This completes the presentation of weak R-node templates. We now revisit the Template Lemma (see lemma 5.4) with respect to weak R-nodes. In particular, we prove the Template Lemma for weak R-node templates as the Weak R-node Template Lemma (see lemma A.2).

Lemma A.2. (Weak R-node Template Lemma) The weak R-node invariant holds during the reduce operation.

Weak R-node Invariant:

1. If an eligible weak R-node has no applicable template, $D$ is irreducible.

2. Once a weak R-node $R$ of $D$ has been matched, it will satisfy $S^\ast$.

3. The constraints added to $D$ by matching a weak R-node are PT-constraints of $S^\ast$.

4. Matching a weak R-node $R$ preserves the property that, for every P-node source or $R'$-node source $N$ of $D$, $N$’s leaf set is a constraint in $S$, and, for every $S \in S$, $S$ is either the leaf set of a P-node or $R'$-node in $D$.

5. When matching a weak R-node $R$, each pertinent child $N$ of $R$ (prior to matching) remains pertinent iff it is full.

Proof.

(1) : As we have discussed throughout the presentation of the weak R-node templates, when such a weak R-node $R$ is encountered:

- $R$’s pertinent children do not induce a path in $R_T$; or
- $R$ has a blocked-full child which is not an endpoint of $R_{PP}$; or
- $R_{PP}$ has a partial internal-child $Q_i$: which is not as depicted in fig. A.30, or $Q_i$ is between two pertinent nodes on the path specified by an $R'$-node in $R_R$. 


• $R$ has three (or more) of any combination of the following:

  • blocked-full $R'$-node children, (non-degenerate) partial end-children, and pertinent $R'$-nodes, or
  
  • blocked-full $R'$-node and degenerate partial end-children; or

• $R$ has a surrounded $R'$-node child; or

• Both endpoints of a pertinent $R'$-node of $R$ are a combination of: empty, closed, and one-sided but neither full nor partial; or

• $R$ has two pertinent $R'$-nodes with a common endpoint which is:

  • not open, or
  
  • is neither full nor partial with a single full leaf; or

• $R$ has a bridged pertinent $R'$-node with:

  • An empty child and more than one full leaf, or
  
  • An empty child and a partial child with no endpoint that is full leaf.

Thus, $D$ is irreducible. \qed

For the remaining cases we let $R$ be a pertinent weak $R$-node which can be matched by some template.

(2): Notice that, in template group R1, $R$ will not change and is required to satisfy $S^*$ simply from the pattern (and similarly for $R$’s $R'$-nodes). The other groups will collect the full children and grandchildren of $R$ together so that they will be consecutive in $C_R$’s frontier and in the frontier of $R$’s child $C$ (or $R_1$’s child $C$ and $R_2$’s child $C^*$). Furthermore, $R$’s $R'$-nodes will be appropriately moved to $(C_R)_{R'}$ so that they satisfy $S^*$. This will ensure that $R$ and the $R'$-nodes in $R_{R'}$ satisfy $S^*$ once $R$ has been matched.

Additionally, no P-nodes or $R'$-nodes (other than elements of $R_{R'}$) are modified by the
replacement. Furthermore, any weak R-node \( R^* \), which is modified as a result of the replacement (i.e., \( C_R \) and possibly \( C^*_R \)), will gain a path of full pertinent nodes which becomes adjacent to either a one-sided or an open node in \( R^*_T \). Therefore, any node which satisfied \( S^* \) prior to \( R \) being matched will still satisfy \( S^* \) after \( R \) has been matched. \( \square \)

(3) : To prove this we consider \( D \) immediately prior to matching \( R \) (i.e., \( R \) is eligible in \( D \)) and let \( D^* \) be \( D \) immediately after matching \( R \). We will show that, for every tree \( T, T \in \text{Consistent}(D) \) and \( T[S^*] \) is a path iff \( T \in \text{Consistent}(D^*) \) and \( T[S^*] \) (i.e., the PT-constraints introduced by matching \( R \) are PT-constraints of \( S_D \)).

\[ \Leftarrow \] Notice that the only constraints which are removed from \( D \) by the application of a weak R-node template are guaranteed to be implied constraints of \( D^* \). This is clear since the only time a constraint \( S \) is removed by the application of a weak R-node template is when: \( S \) is the leaf set of \( R^\prime \)-node \( Q \) that becomes orphaned; and the path in the tree of a weak R-node that \( Q \) specifies has become a subpath of the path specified by some newly created \( R^\prime \)-node. Thus, \( \text{Consistent}(D) \subseteq \text{Consistent}(D^*) \) (i.e., for every \( T \in \text{Consistent}(D^*), T \in \text{Consistent}(D) \)).

\[ \Rightarrow \] Since \( T \in \text{Consistent}(D) \), there is a PR-tree of \( D \) whose frontier is \( T \). Let \( D' \) be such a PR-tree. We now let \( D'' \) be the result of matching \( R \) in \( D' \). Notice that \( D'' \) will be an orientation of \( D^* \). We further claim that the frontier of \( D'' \) is the same as the frontier of \( D' \). In particular, since \( T[S^*] \) is a path, it is easy to see that matching \( R \) can be performed without altering the edges that \( R, R \)'s descendants, and \( R \)'s \( R^\prime \)-node descendants R-nodes contribute to \( \text{Frontier}(D') \). Thus, since no other nodes are affected by the application of a template, the frontier of \( D' \) is the same as the frontier of \( D'' \). Furthermore, by preserving the frontier, it is clear that the weak R-nodes of \( D'' \) are actually R-nodes. Thus, \( D'' \) is not only an orientation of \( D^* \), it is also a PR-tree of \( D^* \).

In particular, \( T = \text{Frontier}(D'') \in \text{Consistent}(D^*) \). \( \square \)

(4) : This follows identically to the proof of (4) in the P-node invariant. In particular, it is easy to see that the only time a node is removed by a weak R-node template, is
when an R'-node Q becomes orphaned (as in the P-node case). Thus, using the same argument we can ensure that no node whose leaf set is a constraint in S is removed by the application of a template.

Note: (5) is omitted since it has already proven it in lemma A.1.

(6) : Since the algorithm matches pertinent P-node parents prior to pertinent weak R-node parents we know that (by property (5) of the invariant) no child of R will be bridged. Notice that, if R has exactly one partial or blocked-full (i.e., pertinent but not full) child C, R will no longer be pertinent and neither will C after R has been matched. Furthermore, when R has two such children C and C*, R will be split into non-pertinent weak R-nodes R_1 and R_2 and neither C nor C* will remain pertinent. Additionally, we know that R cannot have more than two such children, otherwise D will not be reducible. Thus proving (6).

This completes the proof of the weak R-node Template Lemma.

In light of property (1) of lemma A.2, all weak R-nodes which do not match one of the weak R-node templates are considered illegal. This lemma concludes our discussion of the weak R-node templates and will be useful in our proof of correctness of the reduce operation in section 5.3.4.

A.3 Complexity Considerations

In this section we will discuss the complexity of executing reduce(D, S*) (see algorithm 9) in the context of our PRtreeReduction algorithm (see algorithm 1). This discussion is presented in three parts. The first two parts concern our use of data structures regarding weak PR-trees. In particular, the Union-Find data structure will play a key role in how we will maintain the weak PR-tree throughout the template matching process. With this in mind, we provide a short discussion of Union-Find (see section A.3.1) followed by our usage of Union-Find regarding weak R-nodes (see section A.3.2). In the final subsection
we will discuss the complexity of reduce which we can achieve via the use of Union-Find.

## A.3.1 The Union-Find Data Structure

The classical Union-Find data structure by Tarjan is used to represent collections of disjoint sets. A detailed discussion of classical Union-Find can be found in most Algorithmic Theory textbooks (for example [40]). It concerns three main operations:

- **makeSet**(*x*): Create the set \{*x*\}.

- **Find**(*x*): Return the unique identifier for the set containing *x*.

- **union**(*x*, *y*): Perform the union of the two sets \(\text{find}(x)\) and \(\text{find}(y)\) (i.e., replace the two sets \(\text{find}(x)\) and \(\text{find}(y)\) with \((\text{find}(x) \cup \text{find}(y))\)).

In an amortized setting the classical Union-Find data structure of Tarjan provides a bound on the runtime of a sequence of \(k\) finds and at most \(\ell\) unions on a universe of \(\ell\) elements. This time bound is expressed as \(O(\ell + k \cdot A(k + \ell, \ell, \log \ell))\) where \(A(m, n, l) = \min\{k | A_k(\lfloor m/n \rfloor) > l\}\) and \(A_i(j)\) is Ackermann’s function as described in [40].

However, for our purposes we would also like to be able to insert and delete elements from the collection of disjoint sets. Thus, we will use an augmented form of Union-Find which includes the following operations (in addition to those above):

- **insert**(*x*, *A*): Insert an item *x* which is not yet in any set into set *A* (since insert already knows which set to modify, insert will have a constant time).

- **delete**(*x*): Delete *x* from the set that contains it (note: delete does not get the set containing *x* as a parameter).

Fortunately, Kaplan et al. [26] have such a version of the Union-Find data structure. They also provide a slight improvement on the classical bound of Tarjan. Specifically, they show that the amortized cost for each union and find operation is \(O(A(k, \ell, \log(s)))\).
Appendix A. Reduction: Template Matching

where \( s \) is the size of the set(s) involved. This new bound is also maintained in the presence of deletions. They also demonstrate that the amortized cost of deleting an element from a set of \( s \) elements is the same as the amortized cost of finding the element (i.e., \( O(A(k, \ell, \log(s))) \)).

In the main body of this paper we use the following simplified (single parameter) version of the inverse Ackermann function.

\[
A(s) = \min\{k | A_k(1) > \log(s)\}; \text{ i.e., } A(s) = A(1, 1, s).
\]

A.3.2 Union-Find and Weak R-nodes

We will now discuss our usage of Union-Find with respect to weak R-nodes. Notice that, in a weak PR-tree, no two weak R-nodes will have a shared child and no two weak R-nodes will have a shared R'-node (i.e., for every pair of weak R-nodes \( R_1 \) and \( R_2 \), \( \text{Children}(R_1) \cap \text{Children}(R_2) = \emptyset \) and \( (R_1)_{R'} \cap (R_2)_{R'} = \emptyset \)). In particular, if \( \{R_0, R_1, ..., R_{k-1}\} \) is the set of all weak R-nodes in a weak PR-tree \( D \), then:

- \( \{(R_0)_{R'}, (R_1)_{R'}, ..., (R_{k-1})_{R'}\} \) is a collection of disjoint sets; and
- \( \{\text{Children}(R_0), \text{Children}(R_1), ..., \text{Children}(R_{k-1})\} \) is a collection of disjoint sets.

We will use one Union-Find instance for each of these collections. Furthermore, we refer to these Union-Find instances as \( F_{R'} \) and \( F_{\text{Children}} \) respectively. In particular, a weak R-node \( R \) will be the representative of the set \( \text{Children}(R) \) in \( F_{\text{Children}} \) and the representative of the set \( R_{R'} \) in \( F_{R'} \). Thus, through \( F_{\text{Children}} \) and \( F_{R'} \) \( R \) will be able to keep track of its children and R'-nodes.

A.3.3 The Complexity of Reduce

We will now analyze the amount of work performed by the sequence of \textit{reduce} operations (see algorithm 9 on page 113) which occur during the \texttt{PRtreeReduction} (see algorithm
In particular, we will provide an upper bound on the amount of work performed by \( \text{reduce}(D_i, S_i^*) \) in the context of an execution of the \textit{Reduction} algorithm.

Recall the following theorem from [15] (see theorem A.3 below, also stated as theorem 3.33 in chapter 3), allows us to relate the size of a PR-tree produced by our algorithm and the running time of our algorithm to the size of the input graph.

**Theorem A.3.** [15] For a chordal graph, the number of ones in the vertex to maximal clique incidence matrix is \( O(n + m) \).

**Theorem A.4.** For a graph \( G \) and \( i \in [1, n-1] \), the execution of \( \text{reduce}(D_i, S_i^*) \) which occurs during \( \text{PRtreeReduction}(S_G) \) (for \( S_G = \{S_0, S_1, ..., S_{n-1}\} \)) can be completed in:

\[
O(A(\sum_{j=0}^{i-1} |S_j|) * \sum_{j=0}^{i-1} |S_j|) \text{ time.}
\]

**Proof.** To provide the desired runtime of this \text{reduce} operation it will suffice to assume that the pertinent subtree (i.e., \text{Pertinent}(D_i, S_i^*)) is the whole weak PR-tree \( D_i \). Recall that, by theorem 3.32, the size of \( D_i \) is \( |D_i| \in O(\sum_{j=0}^{i-1} |S_j|) \).

We now consider the \text{label} operation (i.e., first component of the \text{reduce} operation). This operation will traverse \( D_i \) from \( S_i^* \) until it reaches the source nodes of \( D_i \) and then retrace its steps in order to properly identify the pertinent subtree and the properties of pertinent nodes which are of interest for the template matching process. In particular, at each node \( N \) that is traversed, \text{label} will need to perform a \text{find} operation with respect to \( \mathcal{F}_{\text{Children}} \) when that node has a weak R-node parent. This operation will cost \( O(A(|D_{n-1}|)) \) time. Other than these \text{find} operations, \text{label} will perform a constant amount of work for each time a node is visited. More specifically, the number of times the \text{label} operation will visit each node \( N \) is precisely the number of children of \( N \) that have leaves in \( S^* \). Thus, the total amount of time required by \text{label} \( (D_i) \) is \( O(A(|D_i|) * |D_i|) \).

The use of Union-Find also allows the reconciliation of weak R-nodes associated with the P-node templates to be performed in inverse Ackermann time (i.e., we will use a constant number of \text{union} operations with respect to each of \( \mathcal{F}_{\text{Children}} \) and \( \mathcal{F}_{\text{R}} \) during
the execution of each template). In particular, the reconciliation of related weak R-nodes will take \( O(A(|D_i|)) \) time for each P-node and weak R-node template that is applied during the template matching process (i.e., for each pertinent P-node and weak R-node).

Let the primary operation associated with a template be: the operations performed by the template other than the reconciling of related weak R-nodes (for a P-node template) and the movement of related elements (for a weak R-node template). The primary operation associated with each template can be performed in time proportional to the number of pertinent children of the node being processed (i.e., constant time when amortized over the entire template matching process). This is accomplished by representing each R′-node \( Q \) by its path in \((Q_R)_T\). Therefore, each application of a P-node template will have an amortized time of \( O(A(|D_i|)) \) over a single call to \texttt{reduce}. Moreover, even if every node in the pertinent subtree is a P-node, all P-node template applications will still complete in \( O(A(D_i) \ast |D_i|) \) as needed.

The movement of relevant elements performed by the weak R-node templates (i.e., every weak R-node template which is not in group R1) could require moving empty nodes. In particular, during a single call to \texttt{reduce}, every empty node in \( D \) could be affected by this operation. Fortunately, each node will be affected at most once throughout a call to \texttt{reduce}. This can be seen by examining the weak R-node templates and noticing that we are always moving the relevant elements down to a lower (and previously already matched) weak R-node. Furthermore, as previously mentioned, we will also require a constant number of \texttt{union} and \texttt{find} operations with respect to \( F_{R'} \). However, these will be dominated by the movement of relevant elements when considered over the entire execution of \texttt{reduce}. In particular, in the worst case, we will require \( O(|R_{R'}| + |Children(R)| + A(|(C_R)_{R'}|) + A(|(C^*_{R'})_R|)) \) time where \( R \) is the weak R-node being matched and \( C \) and \( C^* \) are the endpoints of its pertinent path. Moreover, the weak R-node templates have a total cost of:

\[
O(\sum \{|R_{R'}| + |Children(R)| + A(|R_{R'}|) : \forall \text{ weak R-nodes } R \text{ in } D_i\});
\]
equivalently (since $A(|R_R'|)$ is dominated by $|R_R'|$):

$$O(\sum\{|R_R'| + \text{Children}(R)| : \forall \text{ weak R-nodes } R \text{ in } D_i\} ) .$$

Notice that, this quantity is clearly bounded by the size of $D_i$. Therefore, the weak R-node templates will contribute $O(|D_i|)$ to the execution time of each call to $\text{reduce}(D_i, S_i^*)$.

After the template matching has completed we will need to locate the LCSA of $S_i^*$ in the matched weak PR-tree $D^*$. Recall that, to locate the LCSA of $S_i^*$ we only traverse the pertinent nodes in $D^*$. In particular, we will traverse a collection of disjoint subtrees of $D^*$ (i.e., one for each element of the LCSA). Furthermore, since every element of the LCSA is full, every node in every subtree will be full. Therefore, the total number of nodes that we will traverse is proportional to the number of elements in $S_i^*$). Thus, the LCSA of $S^*$ in $D^*$ can be determined in $O(|S_i^*|)$ time.

Therefore, each execution of $\text{reduce}(D_i, S_i^*)$ can be performed in $O(A(|D_i||D_i|)$ time (i.e., $O(A(\sum_{j=0}^{i-1}|S_j|) \times \sum_{j=0}^{i-1}|S_j|)$ time).

The analysis provided in theorem A.4 is not a particularly tight (since we are considering the pertinent subtree of $D_i$ to be all of $D_i$). Moreover, we believe that a more careful analysis of the work involved could improve this bound and, consequently, the overall runtime of the Reduction method. However, from this bound we now know the amount of time that $\text{PRtreeReduction}$ will spend on its calls to the $\text{reduce}$ method.

**Corollary A.5.** The total execution time required by the calls to the $\text{reduce}$ operation throughout the execution of $\text{PRtreeReduction}(S)$ (for $S = \{S_0, S_1, \ldots, S_{n-1}\}$) is:

$$O(\sum_{i=1}^{n-1} A(\sum_{j=0}^{i-1}|S_j|) \times \sum_{j=0}^{i-1}|S_j|).$$

Moreover, when $S = S_G$ for a chordal graph $G$, by theorem A.3, this simplifies to:

$$O(n(A(n + m) \times (n + m))) = O(A(n + m)nm).$$

This completes our discussion of the complexity of the sequence of $\text{reduce}$ operations which occur during an execution of the Reduction method.