# A Linear-Time Implementation of PC-trees

Bachelor Thesis of

Matthias Pfretzschner

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science


Reviewer: Prof. Dr. Ignaz Rutter
Advisor: Simon Fink, M.Sc.

Time Period: 13th April 2020 – 13th July 2020

**Statement of Authorship**

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Passau, June 29, 2020

**Abstract**

A PC-tree is a data structure that represents certain circular permutations of elements of a set. It allows restrictions on these permutations, i.e., constraints where specific elements have to be consecutive. In this thesis, we explain the structure and functionality of PC-trees. We describe and implement an algorithm in C++ that allows applying restrictions to a PC-tree in linear time. This algorithm turns out to be significantly faster than the PQ-tree, a similar data structure that represents non-circular permutations with restrictions. Additionally, we adapt an existing linear-time algorithm for the intersection of two PQ-trees and implement it for our PC-tree.

**Deutsche Zusammenfassung**

Ein PC-Baum ist eine Datenstruktur, die bestimmte zirkuläre Ordnungen von Elementen einer Menge repräsentiert. PC-Bäume ermöglichen Restriktionen auf diesen Ordnungen, d.h., Beschränkungen, die erzwingen, dass bestimmte Elemente benachbart sein müssen. In dieser Arbeit beschreiben wir den Aufbau und die Funktionsweise von PC-Bäumen. Wir implementieren einen Algorithmus in C++, der es ermöglicht, Restriktionen in Linearzeit auf einen PC-Baum anzuwenden. Dieser Algorithmus erweist sich als deutlich schneller als der PQ-Baum, eine ähnliche Datenstruktur, die gewöhnliche Permutationen mit Restriktionen darstellt. Außerdem passen wir einen existierenden Linearzeit-Algorithmus für den Schnitt von zwei PQ-Bäumen an und implementieren ihn für unseren PC-Baum.

# Contents

# 1. Introduction

A restriction on the permutations of a set is a constraint that requires specific elements to be consecutive. For example, given set $V = \{1, 2, 3, 4\}$, there exist $4! = 24$ different permutations of the elements in $V$ without any restrictions. With the restrictions $\{2, 3\}$ and $\{3, 1\}$, the only remaining valid permutations are $(1, 3, 2, 4)$, $(2, 3, 1, 4)$, $(4, 1, 3, 2)$, and $(4, 2, 3, 1)$.
To solve problems related to permutations with restrictions, Booth and Lueker [BL76] introduced the PQ-tree data structure, a rooted tree that represents all possible permutations of a set $V$ with certain restrictions. Every leaf of the tree corresponds to an element in $V$, the inner nodes represent the restrictions. Booth and Lueker gave an algorithm that applies restrictions to a PQ-tree in linear time by updating the tree accordingly. The algorithm applies nine different templates to multiple nodes of the tree, depending on their position in the PQ-tree. These templates make the algorithm very bulky and complicated, implementing PQ-trees and algorithms depending on them can be very difficult.
In order to eliminate the flaws of PQ-trees, Shih and Hsu [SH99] introduced the PC-tree, a generalization of PQ-trees. PC-trees represent circular permutations with restrictions, i.e., distinct arrangements of elements around a circle. Hsu and McConnell [HM03, HM04] gave a linear-time algorithm for applying restrictions to PC-trees and proved that computing the PQ-tree reduces in linear time to computing the PC-tree. The PC-tree is unrooted and completely omits the complex template matching, making the algorithm much simpler.
Although PC-trees eliminate most of the disadvantages of the PQ-tree, there is currently no linear-time implementation of PC-trees publicly available. Hsu and McConnell's description of the algorithm is rather short and superficial. The main goal of this thesis is to develop the details of the algorithm and to implement the PC-tree data structure in C++.

Historically, Booth and Lueker's [BL76] initial application of PQ-trees was to determine the consecutive-ones property of a $(0, 1)$-matrix in linear time, i.e., detecting whether the columns of the matrix can be rearranged so that, in every row, the ones are consecutive. Figure 1.1 gives an example of a matrix with the consecutive-ones property and shows how its columns can be rearranged to make all ones consecutive. Every row of the matrix defines a restriction on the permutations of its columns. Booth and Lueker's algorithm starts with a PQ-tree that represents all permutations of the columns. For every row, the algorithm applies the corresponding restriction to the PQ-tree. If all restrictions are compatible, the matrix has the consecutive-ones property.

$$\begin{array}{ccccc} a & b & c & d & e \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \longrightarrow \begin{array}{ccccc} b & e & d & a & c \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

Restrictions:
$\{b, d, e\}$
$\{d, e\}$
$\{a, c, d, e\}$
$\{a, d\}$
$\{a, c\}$

Figure 1.1.: A matrix with the consecutive-ones property.

This problem is also closely related to interval graph recognition. For a set $I$ of intervals, the interval graph is a graph with a node for every interval in $I$. If two intervals in $I$ intersect, their corresponding nodes are adjacent in the interval graph, as illustrated in Figure 1.2. Booth and Lueker [BL76] gave a linear-time algorithm that recognizes whether a graph is an interval graph using their PQ-tree algorithm.

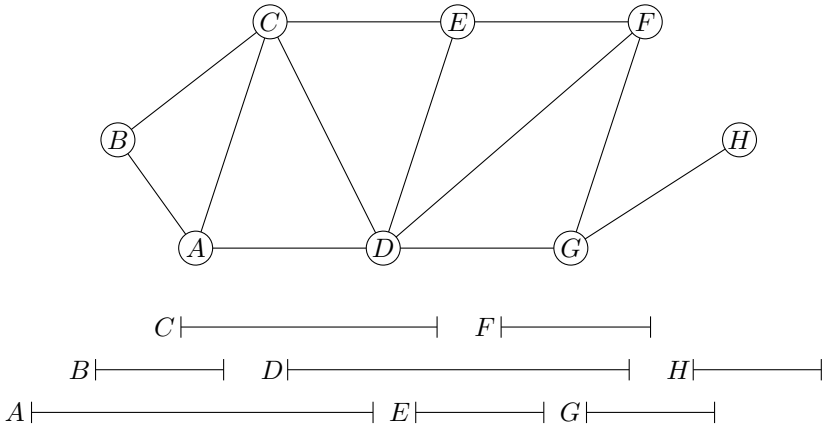


Figure 1.2.: A set of intervals and their corresponding interval graph.

Another application of PQ-trees are planarity tests. A graph is planar if it can be drawn on the plane without any edges intersecting, except for their endpoints. Such a representation of the graph is called a planar embedding. Figure 1.3 shows an example of a planar graph and one of its planar embeddings. Booth and Lueker [BL76] introduced a linear-time planarity test using PQ-trees as a modification of the planarity test by Lempel, Even and Cederbaum [LEC67].

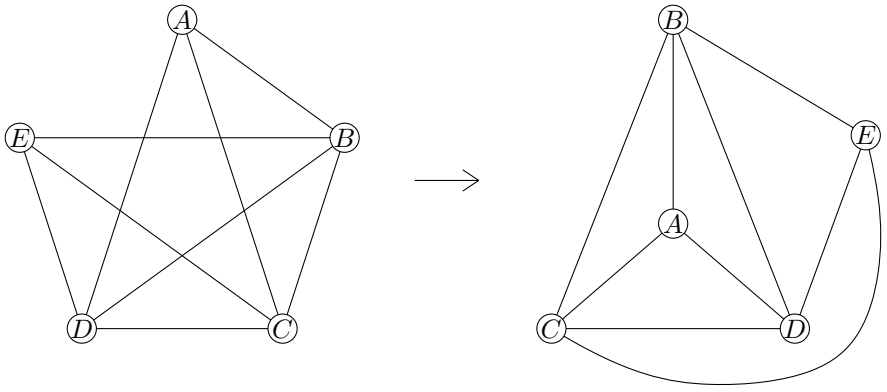


Figure 1.3.: A planar embedding of a graph.

The linear-time level planarity test by Jünger et al. [JLM98] also uses PQ-trees as the underlying data structure. The linear-time radial level planarity test by Bachmaier et al. [BBF05] uses PQR-trees, a modification of PQ-trees. The Simultaneous PQ-Ordering

problem introduced by Bläsius and Rutter [BR13] takes a set of PQ-trees as its input.
Shih and Hsu [SH99] introduced a planarity test specifically designed for PC-trees. In contrast to the planarity test by Booth and Lueker, the structure of the PC-tree corresponds to the structure of the original graph. This makes finding a planar embedding of the graph much easier.

Hsu and McConnell [HM03, HM04] gave a brief description of how the PC-tree could be implemented and introduced an algorithm that applies restrictions on a PC-tree in linear time. We build on their algorithm and implement the PC-tree in C++, giving an extensive description of the implementation details. In particular, we expand their algorithm for finding the path of nodes that need to be altered when updating the tree and show that our approach handles all possible cases. Our algorithm is also able to detect that a restriction is impossible before any modifications are made to the structure of the tree.
In order to evaluate the performance of our data structure, we implement the planarity test by Booth and Lueker [BL76] twice, once using PC-trees and once using PQ-trees. Our benchmarks show that our implementation of the PC-tree is significantly faster at applying restrictions than the implementation of the PQ-tree in the Open Graph Drawing Framework (OGDF) [CGJ+14]. Since computing the PQ-tree reduces in linear time to computing the PC-tree, some of the algorithms using PQ-trees mentioned above could therefore benefit from the performance advantage of PC-trees.
Additionally, we implement the first known linear-time intersection algorithm for PC-trees. Booth briefly described in his PhD-thesis [Boo75] how such an algorithm could be implemented for PQ-trees. We adapt his algorithm for PC-trees, describe its functionality in detail and prove its linear time bound.

Chapter 2 introduces the PC-tree and provides the foundation necessary for the implementation. Chapter 3 describes the details of the data structure and its implementation, starting with its general structure and the initialization. Afterwards, we show how to implement the different steps of the linear-time algorithm for applying restrictions. We also describe and implement an algorithm that allows the intersection of two PC-trees in linear time, based on the algorithm for PQ-trees by Booth [Boo75]. In Chapter 4 we evaluate the performance of our implementation. We compare the running time of our restriction algorithm to PQ-trees in Chapter 4.1 and we compare the performance of our intersection algorithm to a similar, more naive intersection algorithm in Chapter 4.2. Chapter 5 summarizes the results of this thesis.

# 2. The PC-tree data structure

In this chapter, we introduce PC-trees using circular permutations and illustrate the structure and functionality of PC-trees. We also explain the fundamentals of our algorithm that are necessary for the implementation.

## 2.1. Circular permutations and restrictions

Given a ground set $V$ with $n$ elements, there are $(n-1)!$ distinct circular permutations of $V$. Figure 2.1 shows all six circular permutations of $V = \{1, 2, 3, 4\}$ as an example.

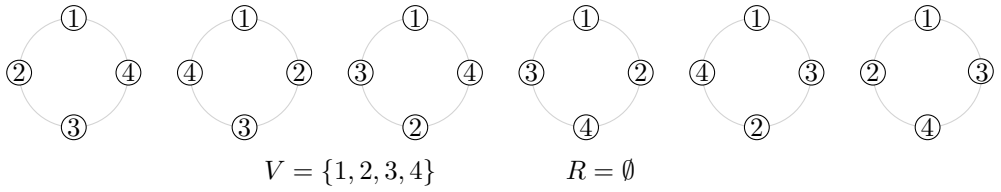


Figure 2.1.: Distinct circular permutations with no restrictions.

Let $R \subseteq \mathcal{P}(V)$ be a family of subsets of $V$. Every $R_i \in R$ represents a *restriction* on $V$, i.e., the elements of $R_i$ have to be consecutive in the resulting permutations. Set $R$ is the set of restrictions on $V$. Figure 2.2 shows the previous example, but extended with restrictions $R_1 = \{1, 2\}$ and $R_2 = \{2, 4\}$. In this case, only two valid permutations satisfying all restrictions in $R$ remain.

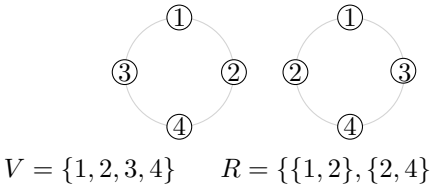


Figure 2.2.: Distinct circular permutations with two restrictions.

Adding restriction $R_3 = \{1, 4\}$ in this example would result in zero circular permutations, as no permutation could satisfy all four restrictions. Therefore, $R_3$ is an *impossible restriction* on $V$ with restrictions $R$.
Adding restriction $R_3 = \{1, 2, 4\}$ would result in the same valid circular permutations,

because $R_3$ is implied by $R_1$ and $R_2$. Therefore, $R_3$ is a *trivial restriction* on $V$ with restrictions $R$. Any restriction $R_i$ with $|R_i| \in \{0, 1, n-1, n\}$ is also trivial.

## 2.2. The PC-tree

Shih and Hsu [SH99] introduced the *PC-tree*, a data structure that represents all circular permutations of $V$ with restrictions $R$. PC-trees are similar to the PQ-trees introduced by Booth and Lueker [BL76], which represent all non-circular permutations. In contrast to PQ-trees, PC-trees are unrooted, which makes modifications on PC-trees much simpler, because there is no need to track the position of the root node.
A PC-tree consists of *P-nodes*, *C-nodes* and *leaves*. Edges incident to C-nodes have a specific circular order, which can be reversed. Therefore, a C-node represents exactly two circular permutations of its neighbors. Edges incident to P-nodes can be rearranged arbitrarily, a P-node therefore represents all circular permutations of its neighbors. Every leaf of the tree represents one element of $V$. Embedding the leaves in a circle while taking into account the constraints of the P-nodes and C-nodes yields one possible permutation of the elements of $V$.
Figure 2.3 illustrates the structure of a PC-tree and the role of C-nodes. The C-nodes are represented by big double circles and the P-nodes are represented by small circles. Figure 2.3 ($b$) shows the same PC-tree as ($a$), but the order of edges incident to a C-node is reversed. Both trees represent a valid permutation of $V$.

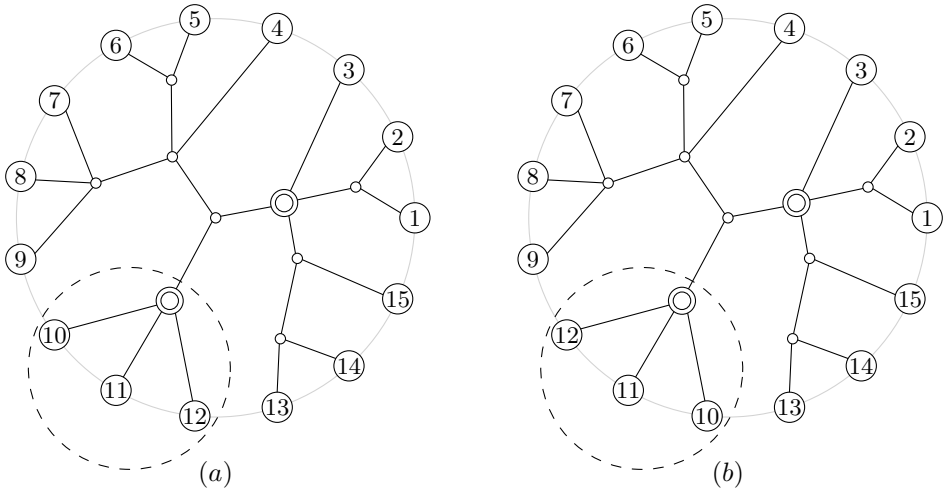


Figure 2.3.: Reversing the order of edges at C-nodes results in a valid permutation.

Similar to the previous example, Figure 2.4 illustrates the role of P-nodes in the PC-tree. Figure 2.4 ($b$) shows the same PC-tree as ($a$), but incident edges are rearranged at multiple P-nodes. Both trees represent a valid permutation of $V$.
The structure of the PC-tree for the permutations in the example in Figure 2.1 and Figure 2.2 is simple, as shown in Figure 2.5. Initially, the PC-tree contains a P-node that represents all permutations of $V$. For $R_1 = \{1, 2\}$, a new P-node is created in order to ensure that 1 and 2 remain adjacent. With $R_2 = \{2, 4\}$, the order of the elements in $V$ is unambiguous and can only be reversed. Therefore, the resulting tree contains a C-node adjacent to all leaves.

## 2.3. Updating the PC-tree

Because a PC-tree $T$ represents all permutations of a set $V$ with restrictions $R = \{R_1, ..., R_i\}$, a new restriction $R_{i+1}$ is possible if and only if edges incident to P-nodes can be rearranged

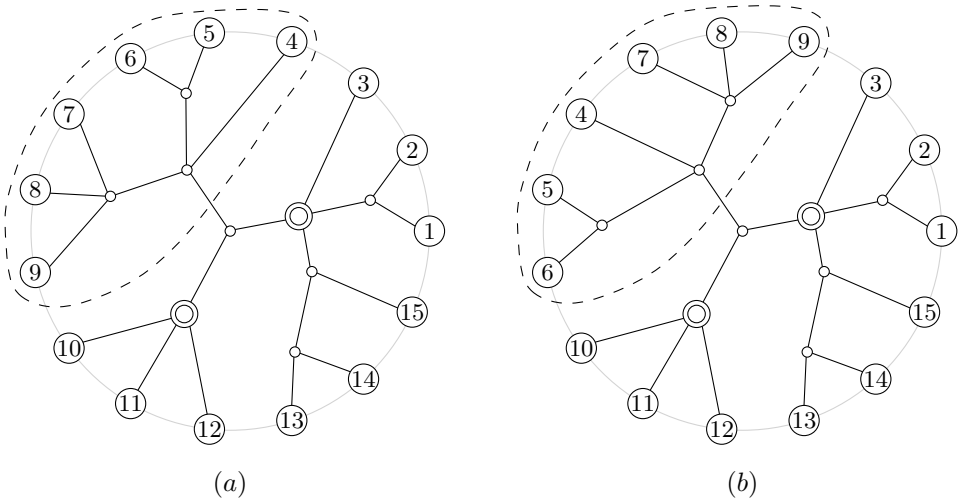


Figure 2.4.: Arbitrarily rearranging edges incident to P-nodes results in a valid permutation.

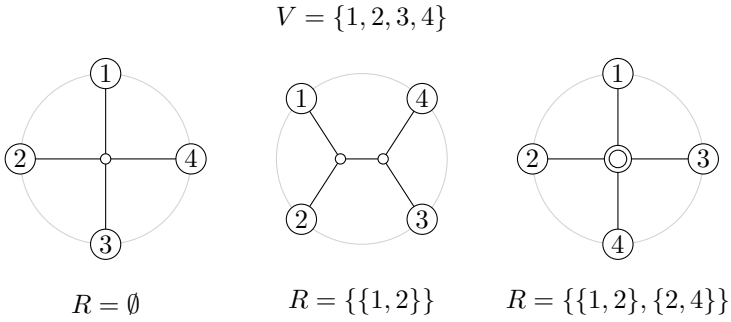


Figure 2.5.: Resulting PC-trees after applying restrictions on $V = \{1, 2, 3, 4\}$.

and orders of edges incident to C-nodes can be reversed in a way so that all leaves of $T$ that represent the elements in $R_{i+1}$ are consecutive.
Let a leaf $x$ be *full* if $x \in R_{i+1}$ and *empty* otherwise. Let an inner node be *partial* if at least one of its neighbors is full and *full* if all its neighbors except one are full. Hsu and McConnell [HM03] showed that, if $R_{i+1}$ is possible, $T$ has a path that contains all nodes and edges that must be modified when applying $R_{i+1}$ to $T$. This path is called the *terminal path*, the nodes at the ends of the terminal path are the *terminal nodes*. Figure 2.6 illustrates the terminal path. The white nodes represent empty nodes, the black nodes represent full nodes and the gray nodes represent partial nodes. The thick edges represent the terminal path with terminal nodes $t_1$ and $t_2$.
Every edge $e$ on the terminal path connects two subtrees with both full and empty leaves. After rearranging $T$ so that all leaves in $R_{i+1}$ are consecutive, reversing the order of edges at one of the two nodes incident to $e$ always means that the leaves in the restriction are no longer consecutive. Since such an operation is no longer valid after applying the restriction, all edges on the terminal path become defunct, i.e., they allow invalid modifications on the tree and must therefore be deleted when updating the tree.
The terminal path contains all partial nodes in $T$ and both terminal nodes are partial. Note that the terminal path may contain empty nodes, but cannot contain full nodes, because no full node can be on a path between partial nodes.
If the defunct edges do not form a path, because one node is incident to more than two defunct edges, the full leaves cannot be made consecutive, since there will always be a subtree with empty leaves between them. In this case, the restriction is impossible.

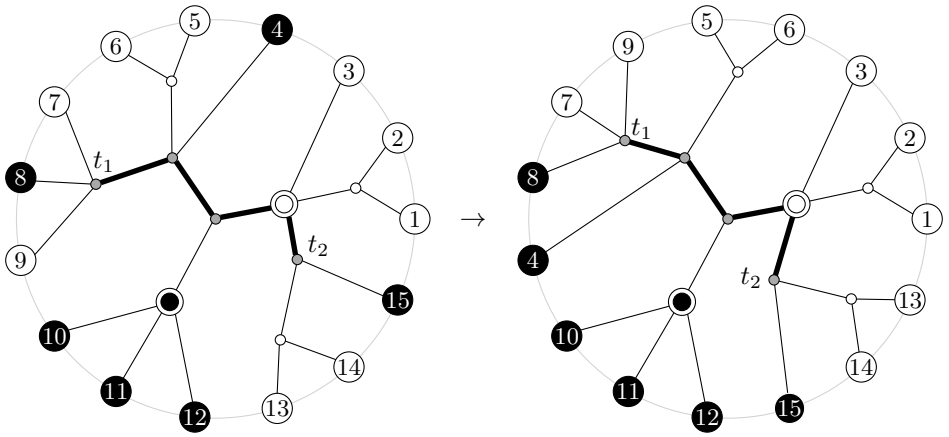


Figure 2.6.: Given a possible restriction, all full leaves can be made consecutive and all nodes that must be modified lie on a path.

When updating $T$ in order to apply the restriction, all defunct edges on the terminal path are deleted as the first step. After that, every node on the terminal path is split into two nodes, one of which holds all edges to full neighbors of the original node, the other holds all edges to empty neighbors. A new central C-node $c$ is created that is adjacent to all the split nodes. This C-node will maintain the order of nodes on the terminal path, but does no longer allow the invalid modifications at defunct edges mentioned above. Contracting all edges to the split C-nodes incident to $c$ and contracting all nodes with degree two results in the updated tree that represents the new restriction.
Figure 2.7 shows the updated tree for the restriction illustrated in Figure 2.6.

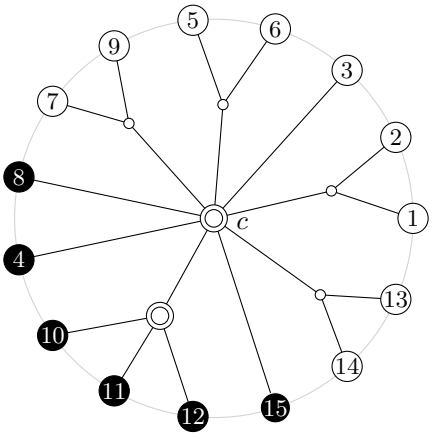


Figure 2.7.: Updated PC-tree with new central C-node $c$.

**Theorem 2.1.** *Let $T$ be a PC-tree over ground set $V$. Let $R = \{R_1, ..., R_i\}$ be a set of restrictions and $m = \sum_{j=1}^{i} |R_j|$. Let $p_j$ be the length of the terminal path for restriction $R_j$. If every restriction $R_j$ takes $O(p_j + |R_j|)$ time, applying all restrictions in $R$ takes $\Theta(m)$ time.*

*Proof.* See the proof by Hsu and McConnell [HM03]. □

Therefore, if the implementation in the following chapter ensures that every restriction $R_j$ takes $O(p_j + |R_j|)$ time, Theorem 2.1 shows that applying multiple restrictions takes linear time in the combined size of all restrictions.

# 3. Implementation

In this chapter we implement a data structure that represents PC-trees, as well as linear-time algorithms for applying restrictions and for intersections of PC-trees. The implementation is realized in C++. Additionally, we use the Open Graph Drawing Framework (OGDF) [CGJ+14] for visualizing the resulting graphs and for general graph-related utilities.

## 3.1. The data structure for representing the PC-tree

For the implementation, we declare an arbitrary node of the tree as the root node. This node has no special function, its sole purpose is to establish a parent function among the nodes, which will help us traversing the tree during the algorithm.
Each undirected edge $xy$ between nodes $x$ and $y$ is represented by two oppositely directed *twin arcs* $(x, y)$ and $(y, x)$. Every arc stores a pointer to its twin arc and to its two neighboring arcs oriented in the same direction. An arc $(x, y)$ also contains a flag specifying whether $y$ is the parent of $x$ and, if $y$ is a P-node, a pointer to $y$. Some arcs also require a pointer to the arc at the other end of their block, which will be explained in Section 3.3.1 as part of the labeling algorithm.
Every node stores its own label, which is either *empty*, *partial*, or *full*, a pointer to its parent arc and pointers to up to two predecessors on the terminal path.
Additionally, a P-node stores its own degree and a list of pointers to all of its incident arcs to full neighbors.
In contrast to P-nodes, we do not permanently store C-node objects, as maintaining the pointers to the C-node in the neighboring arcs over several iterations of the algorithm would be too time-consuming. However, we may keep temporary C-node objects within one iteration of the algorithm in order to simplify the implementation.
Therefore, the C-node is only specified by the order of the edges in the doubly linked circular list given by the edges to its neighbors. Whenever a traversal of this list is necessary, we need to keep track of both the previous and the current arc in every step, picking the neighbor that is different from the previous as the next arc (Algorithm 3.1). This is necessary, because we cannot guarantee a specific order in the list after updating the tree in the last step of the algorithm.

We keep a global *timestamp*, a counter indicating the number of iterations the algorithm has already passed. Whenever we need to store information in a node or an arc that is only valid for one iteration (e.g., labels), we store this information using tuples. The first index is the actual value, the second index is the current timestamp. This way, we can

**Algorithm 3.1:** FINDING THE NEXT ARC

**Input:** Previous arc $p$, current arc $c$
**Output:** Next arc in the list of neighbors

```
1 if p = c.neighbor1 then
2 |   return c.neighbor2
3 else
4 |   return c.neighbor1
```

immediately determine whether a given variable is updated by comparing the timestamp in the tuple with the global timestamp and we do not have to manually clean the tree from obsolete records.

Whenever we delete an edge in the update step of the algorithm, we delete both twin arcs that represent it. For a node $x$, all arcs that store a pointer to $x$ share ownership of $x$. Therefore, if we delete all arcs with a pointer to $x$ and we no longer need the information stored in $x$, we can delete $x$.

## 3.2. Initializing the PC-tree

Given the ground set $V$, we initialize the tree as follows. First, we create a P-node $r$ and declare it as the root of the tree. For every element in $V$, we construct a leaf $x$ with an edge represented by two twin arcs and establish $r$ as the parent of $x$ by setting the pointers and parent labels in the respective arcs. Leaf $x$ stores the element of $V$ it represents. Additionally, the children of $r$ are ordered circularly around $r$ by setting the neighbor pointers in every edge.

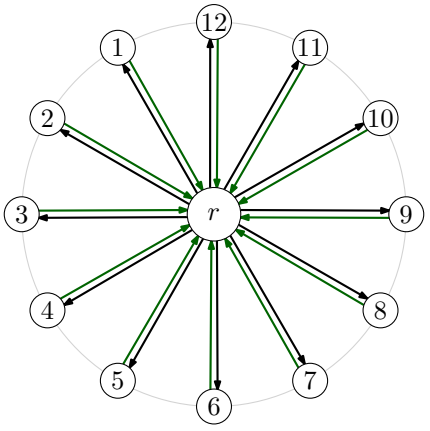


Figure 3.1.: Initial PC-tree for $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$.

Figure 3.1 shows the structure of the initial PC-tree. Note that the green arcs form a circular list, while every other arc forms a new list on its own.
This initial tree now represents all permutations of $V$, as no restrictions are currently represented.

## 3.3. Applying restrictions

Using the data structure defined in Section 3.1, we now give a detailed description of the implementation of a linear-time algorithm for applying restrictions on PC-trees. The implementation is based on the algorithm by Hsu and McConnell [HM03, HM04].

### 3.3.1. Assigning labels

When a new restriction is given as a set $R_i$, the nodes of the tree need to be labeled in order to find the terminal path. Initially, we start by labeling every leaf that is present in $R_i$ as *full*. We declare a node as *partial*, when at least one of its neighbors is full. We declare a node as *full*, when all of its neighbors except one are full. Every node that is not labeled full or partial is considered *empty*.
Whenever a node $x$ becomes full, we inform its non-full neighbor $y$ and $x$ keeps a pointer to arc $(x, y)$ to simplify finding the informed neighbor $y$, when we traverse the tree a second time. The following algorithm ensures that, if $x$ is a C-node, it is represented by a temporary object where we can store this information.
If $y$ is a P-node, we add $(x, y)$ to $y$'s list of arcs to full neighbors. Once the size of this list is greater than 0, $y$ is labeled partial. Once it becomes one less than the degree of $y$, $y$ is labeled full and we inform $y$'s non-full neighbor $z$. We find $z$ by traversing the list of arcs incident to $y$.
If $y$ is a C-node, there is no explicit counter for its full neighbors. Instead, we combine consecutive edges of full neighbors into *blocks*. The first and the last edge in every block stores a pointer to each other, thus creating another doubly linked circular list within the list of all neighbors. Additionally, each block holds a pointer to a temporary C-node object, stored in both ends of the block. Once a block is adjacent to the parent arc, its C-node keeps a pointer to the parent arc.
When an adjacent node $x$ of $y$ becomes full, there are three different cases that need to be considered, depending on whether the two neighbors $u$ and $v$ of $x$ are full or not.

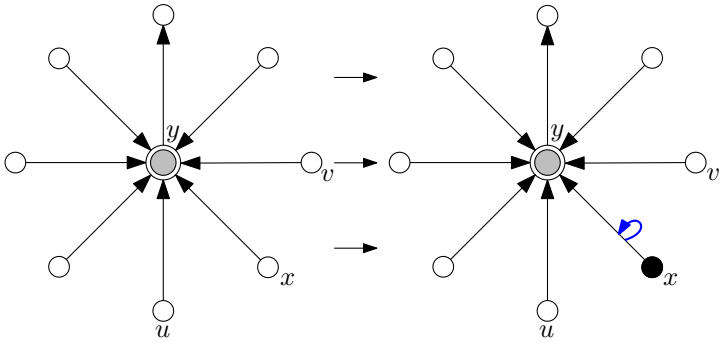


Figure 3.2.: Creating a new block.

Case 1: If the two adjacent arcs $(u, y)$ and $(v, y)$ both contain no block pointer, $u$ and $v$ are both not full and therefore, they are not part of a block. In this case, we create a new block consisting only of arc $(x, y)$ and it stores a newly created C-node object, as well as a block pointer to itself, as shown in Figure 3.2. If arc $(x, y)$ is adjacent to the parent edge, the C-node object stores a pointer to the parent edge.
Case 2: If only one neighboring arc $(v, y)$ has a full child, we append $(x, y)$ to its block, as shown in Figure 3.3. We do this by following the block pointer of $(v, y)$ to the arc $(w, y)$ at the other end of the block, removing $(v, y)$'s block pointer and updating $(w, y)$'s block pointer to $(x, y)$. After that, we give $(x, y)$ a block pointer to $(w, y)$ and the pointer to the C-node object previously stored in $(v, y)$. If the other neighbor $u$ is the parent of $y$, the C-node object stores a pointer to the parent arc $(y, u)$. In case $(w, y)$ and $(x, y)$ are both adjacent to the same, non-full neighbor $(u, y)$, we can label $y$ full and inform $u$.
Case 3: If both $(u, y)$ and $(v, y)$ have full children, we combine both blocks to one, as shown in Figure 3.4. We follow the two block pointers of $(u, y)$ and $(v, y)$ to the ends of both blocks $(m, y)$ and $(n, y)$, remove the block pointers to them and then update the block pointers in $(m, y)$ and $(n, y)$ to point to each other. If the C-node object $C(a)$ of a

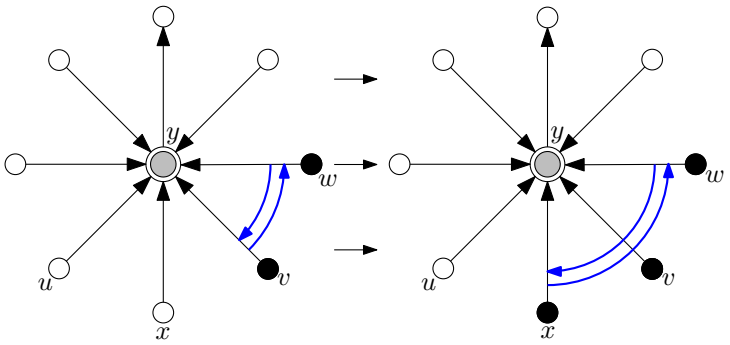


Figure 3.3.: Extending a block.

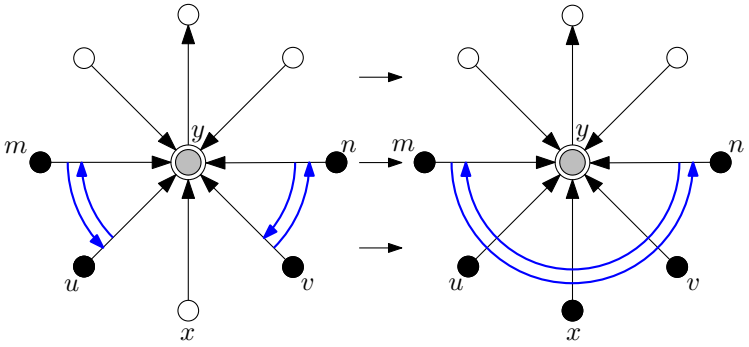


Figure 3.4.: Combining two blocks.

block $a$ already knows its parent arc, the end of the other block $b$ adopts $a$'s pointer to $C(a)$. Otherwise, we choose the block that adopts the C-node object arbitrarily. We also need to check whether $(m, y)$ and $(n, y)$ are both adjacent to the same, non-full neighbor $(z, y)$. In this case, we label $y$ full and inform $z$.
Assuming the current restriction is possible, all full neighbors of a C-node $y$ have to form a single, consecutive block after the labeling algorithm finishes. This one block might not be adjacent to the parent edge, as shown in Figure 3.5. If this is the case, $y$ has to be the apex, because flipping only the full nodes to one side of the terminal path would be impossible otherwise.

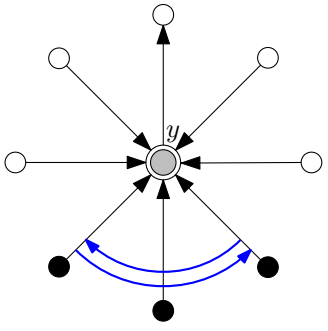


Figure 3.5.: Resulting C-node after the labeling algorithm, where the block of full nodes is not adjacent to the parent edge.

A pointer to the temporary C-node object is only stored in both ends of the block. Spreading this pointer to other arcs is unnecessary, because we will only pass edges adjacent to

the full block while searching the terminal path.
We traverse the tree an additional time, starting at the full leaves. We mark every node $x$ we process to ensure we will not pass the same path multiple times. If $x$ is full, $x$ has a pointer to arc $(x, y)$ it informed when $x$ became full. If $(x, y)$ has a pointer to $y$ and $y$ is not marked, we advance to $y$, which is either full or partial. We do this until we find the first node that is not labeled full and add it to a list that keeps track of all partial nodes.

A node with degree $n$ is only labeled full if $n - 1$ of its neighbors are labeled full. Since every node has a degree of at least 3, the total number of full nodes is in $O(k)$, where $k$ is the number of full leaves. Every partial node has at least one full neighbor, thus there are at most as many partial nodes as full nodes. For this reason, the total number of processed nodes during the labeling algorithm is in $O(k)$. Since we only process edges incident to full nodes and we only process every edge a constant number of times, both traversals of the tree take $O(k)$ time during the labeling algorithm.
Figure 3.6 gives an example of a fully labeled PC-tree. The black nodes represent full nodes and the gray nodes represent partial nodes.

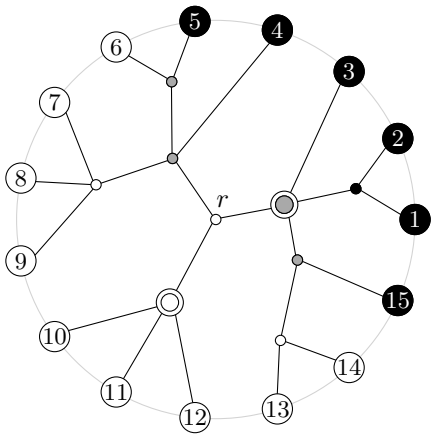


Figure 3.6.: A fully labeled PC-tree with root node $r$.

### 3.3.2. Finding the terminal path

Let the *apex* be the highest node on the terminal path, i.e., the node that is an ancestor of all other nodes on the terminal path. After the labeling algorithm finishes, there are two possible structures for the terminal path, assuming the restriction is possible. The structure depends on the position of the apex, which in turn depends on the position of the root node.
Case 1: If the apex lies on one of the ends of the terminal path and is therefore a terminal node at the same time, the terminal path extends in a single path towards the root node, as shown in Figure 3.7. In this case, every node on the terminal path has exactly one child on the terminal path, except for the terminal node $t_1$.

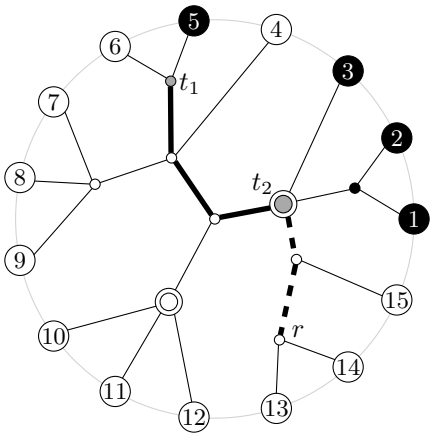


Figure 3.7.: A terminal path where $t_2$ is both apex and terminal node.

Case 2: If the apex does not lie on one of the ends of the terminal path, two paths join in the apex, as shown in Figure 3.8. In this case, the apex $a$ has two children on the terminal path, while the terminal nodes $t_1$ and $t_2$ have none.

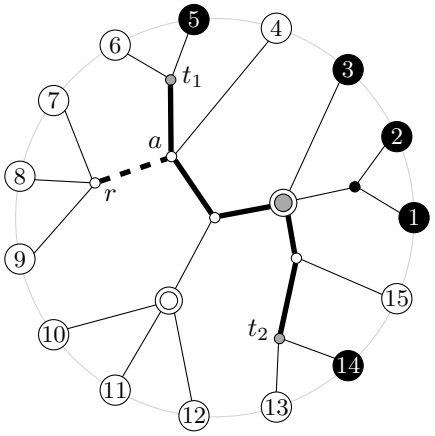


Figure 3.8.: A terminal path where two paths join in the apex $a$.

In order to find the terminal path efficiently, we conduct parallel searches, starting at every partial node and extending paths through their ancestors at the same rate. For this purpose, we use a queue $Q$, where we store the nodes we still need to process. Initially, $Q$ contains all partial nodes found by the labeling algorithm. We track the number of paths that are currently being processed, which is the size of $Q$ initially. Every node keeps pointers to its direct predecessors on the terminal path and a flag indicating whether it has already been processed. Whenever the current node $x$ has already been processed, we stop extending that path and decrease the path counter. Otherwise, we add $x$ to the predecessors of its parent $y$ and enqueue $y$. In case $y$ has two predecessors, we know the terminal path has the structure shown in Figure 3.8 and $y$ is the apex node. We keep a pointer to $y$ as the apex candidate. If a node has more than two predecessors or we find multiple apex candidates, the restriction is impossible.
Finding the parent $y$ of the current node $x$ during the search is always an $O(1)$ operation, as shown in the following cases. Given a pointer to the parent arc $(x, y)$, there are four cases for finding $y$, depending on the type and label of $y$.
Case 1: If $y$ is full, it cannot be on the terminal path, because no full node can be on a path between two partial nodes. Therefore, we stop extending that path.
Case 2: If $y$ is a P-node, it is represented by a permanent record and $(x, y)$ stores a pointer

to $y$.
Case 3: If $y$ is a partial C-node, $(x, y)$ does not know $y$, since $x$ cannot be full. However, $(x, y)$ has to be adjacent to the full block for the restriction to be correct, because otherwise, flipping all non-full children to one side of the terminal path would be impossible. We obtain a pointer to $y$ from the full block and also store it in $(x, y)$. Figure 3.9 gives an example for this case. The green arcs represent the terminal path.

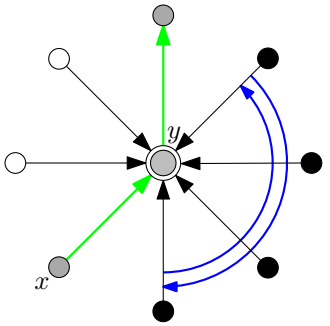


Figure 3.9.: Terminal path at a partial C-node with non-full child $x$.

Case 4: If $y$ is a C-node and it is not full or partial, we did not create a temporary object for it during the labeling step. As shown in Figure 3.10, arc $(x, y)$ has to be adjacent to the other arc on the terminal path, because otherwise, flipping all non-full children to one side of the terminal path would be impossible. If neither of the two edges on the terminal path is the parent edge, $y$ has to be the apex. To detect this, we examine both arcs adjacent to $(x, y)$. If neither of them contains a pointer to a C-node object, we create a new one. Otherwise, we obtain the pointer from the adjacent arc and store it in $(x, y)$. If $(x, y)$ is adjacent to the parent arc, the C-node object stores a pointer to it, otherwise we stop extending this path. Therefore, if $y$ is the apex, the first path entering $y$ will create a new C-node object for $y$, the second path will obtain the object from its neighbor and detect $y$ as the apex when setting the second predecessor of $y$.

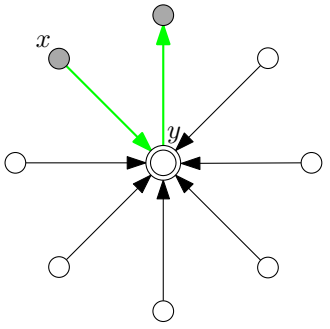


Figure 3.10.: Terminal path at an empty C-node with non-full child $x$.

Case 4 ensures that every node $x$ encountered during the parallel searches is represented by an object. Therefore, finding the parent arc $(x, y)$ of $x$ is trivial, because $x$ stores a pointer to it. If $x$ does not store a pointer to $(x, y)$, we stop extending that path, thus we do not need to find the parent edge. Additionally, if $x$ is a partial C-node without a pointer to its parent arc, we store $x$ as the apex candidate.
Whenever we stop extending a path in case 1 or because the current node $x$ has no pointer to its parent arc, we keep a pointer to $x$ as he highest node in the subtree, but we do

not decrease the path counter. This way, if the path counter ever hits 1, we know we are currently extending above the apex and stop, storing the current node as the highest node. After the parallel searches finish when the path counter hits 1 or the queue is empty, we found an apex candidate if a node has two predecessors or a partial C-node does not know its parent arc. If this is not the case, the terminal path must have the structure shown in Figure 3.7 and we have to manually search for the apex node. In this case, we stored a pointer to the highest node of the subtree, which is located somewhere between the apex and the root node. Starting from this node, we iterate top-down through its predecessors until we find the first partial node, which has to be the apex.
We follow the path of predecessors starting from the apex, adding every node on the way to a list. We do this until we reach a terminal node, which has no predecessor. If the apex has two predecessors, we do the same for the path starting at its second predecessor. The resulting list contains all nodes on the terminal path.

The algorithm above is capable of finding the terminal path if the terminal path has one of the two structures shown in Figure 3.7 and Figure 3.8. However, if the restriction is not possible, there are four additional possibilities for the resulting path and we need to provide extra measures to detect that the restriction is not possible.

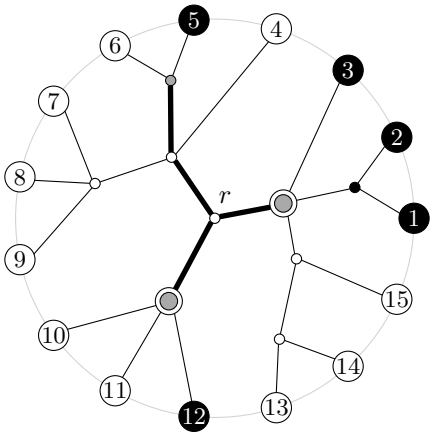


Figure 3.11.: An impossible restriction where three paths join in root r.

Case 1: If more than two paths join in a node, the result cannot be a single path, as shown in Figure 3.11. Therefore, the restriction is impossible. We can easily detect this when we add a third predecessor to a node and stop the parallel searches.

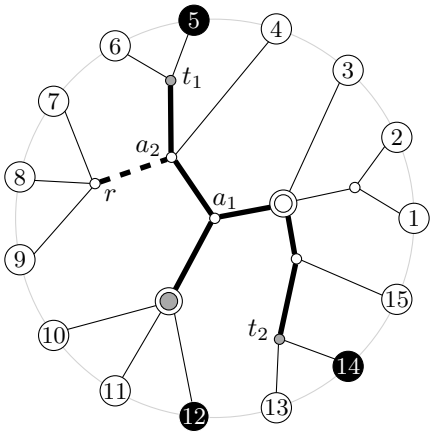


Figure 3.12.: An impossible restriction with multiple apex candidates $a_1$ and $a_2$.

Case 2: Similar to the previous case, the algorithm may detect multiple apex candidates, as shown in Figure 3.12. In this example, $a_1$ and $a_2$ both have two predecessors and are therefore both considered apex candidates. In this case, we can terminate the search when we find a second apex candidate and consider the restriction impossible.

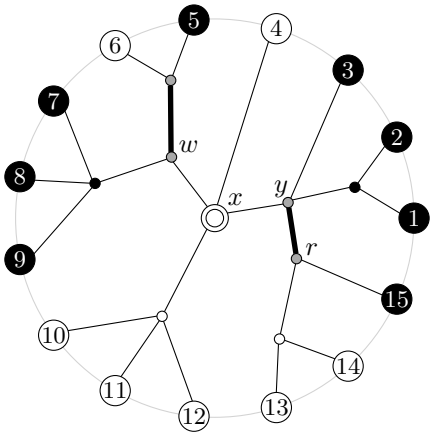


Figure 3.13.: An impossible restriction where the terminal path is incoherent.

Case 3: In case 4 of finding the parent node, we may detect that an empty C-node $x$ is not on the terminal path, because its parent edge is not adjacent to the child edge. This could lead to two separate paths that are not connected after the parallel searches finish, as shown in Figure 3.13. In this case, our algorithm stores $x$ as the highest point of the subtree and will later find $w$ as the apex. As a result, the list that represents the terminal path will not contain the nodes $y$ and $r$. In order to detect this, we simply count the number of partial nodes visited, when we iterate top-down from the apex. If the number of visited partial nodes is less than the total number of partial nodes, we know the terminal path is incoherent.

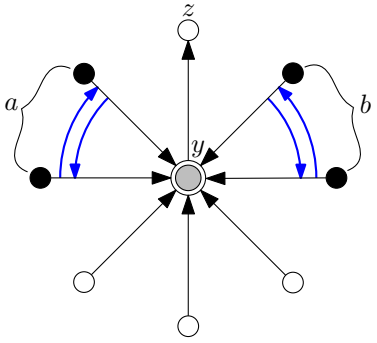


Figure 3.14.: Two incoherent blocks of full neighbors adjacent to the parent edge.

Case 4: The last cause for impossible restrictions is incoherent blocks of full neighbors at C-nodes, as shown in Figure 3.14. In this example, the blocks $a$ and $b$ are not adjacent, thus they contain different C-node objects $C(a)$ and $C(b)$. Since both blocks $a$ and $b$ are adjacent to the parent edge, $C(a)$ and $C(b)$ both have a pointer to the parent arc $(y, z)$. The terminal path algorithm will set both $C(a)$ and $C(b)$ as the predecessors of $z$ and declare $z$ as the apex. To prevent this, we compare the parent arcs of both predecessors whenever we add a second predecessor to a node. If both predecessors have the same parent arc, we terminate the search.
Any other combination of incoherent blocks of full neighbors at a C-node will result in multiple apex candidates or incoherent paths, because the algorithm will not detect that

the C-node objects created for every block represent the same C-node. Therefore, no further measures are required to detect that the restriction is impossible.

For pseudo-code illustrating the steps of the terminal path search, see Appendix A.

Since we provided mechanisms to prevent the parallel searches from extending indefinitely above the apex, the number of processed edges in this step is in $O(p)$, where $p$ is the length of the terminal path. Since finding the parent node is always an $O(1)$ operation, the parallel searches take $O(p)$ time. Combined with the labeling algorithm, finding the terminal path takes $O(p + k)$ time.

### 3.3.3. Updating the terminal path

After finding the terminal path, we use the approach shown in Figure 3.15 to update all nodes on the terminal path. First, we delete all edges between nodes on the terminal path. After that, we split every node $x$ on the terminal path into two nodes, one of which holds all edges to full neighbors of $x$, the other holds all edges to empty neighbors. We create a central C-node $c$ that is adjacent to all newly created nodes in order to maintain the order of nodes on the terminal path. In the last step, we contract all nodes with degree 2 and contract all edges between $c$ and the split C-nodes.

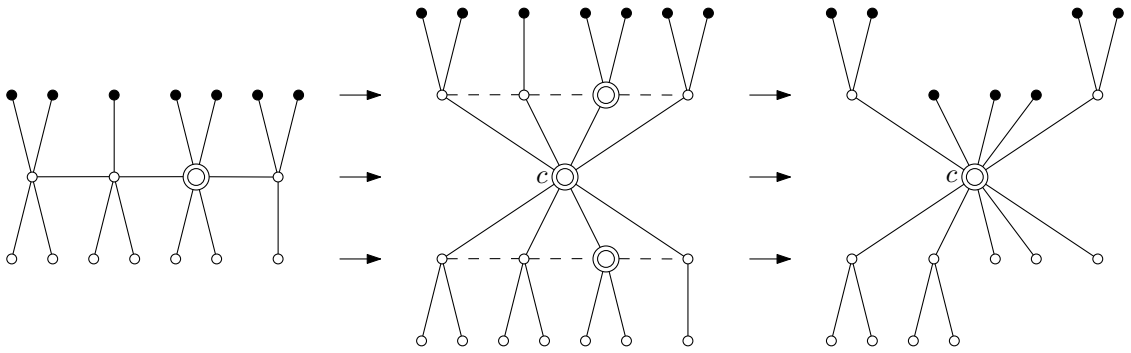


Figure 3.15.: Performing the update step on the terminal path.

In order to be able to split a node $y$ on the terminal path, we need to identify all of its incident edges to full neighbors.
If $y$ is a P-node, $y$ stores a list of pointers to all of its incident arcs to full neighbors. Therefore, no further action is required.
If $y$ is a C-node, all full neighbors form a single, consecutive block on one side of the terminal path, all empty neighbors form a block on the opposite side. Therefore, we only have to obtain pointers to the first and last arc in each block. We do this before we delete the edges on the terminal path. Given arc $(x, y)$ that enters $y$ on the terminal path, we have to consider three different cases, depending on the position of $y$ on the terminal path and the neighbors $(s, y)$ and $(t, y)$ of arc $(x, y)$.
Case 1: If $y$ is a terminal node, $(x, y)$ has a full neighbor $(s, y)$ and an empty neighbor $(t, y)$, as shown in Figure 3.16. We keep pointers to $(s, y)$ as one end of the full block and to $(t, y)$ as one end of the empty block. We follow the block pointer of $(s, y)$ to $(v, y)$, which is the other end of the full block. The empty neighbor $(u, y)$ of $(v, y)$ is the other end of the empty block.

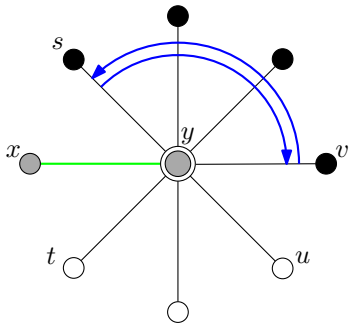


Figure 3.16.: Identifying blocks on a terminal C-node.

Case 2: If $y$ is not a terminal node and one neighboring arc $(t, y)$ is also a terminal path arc, there is either a single full block or a single empty block, as shown in Figure 3.17. We set $(s, y)$ as one end of the block and the neighbor $(u, y)$ of $(t, y)$ as the other end. Note that we have to distinguish whether the block is full or empty, in order to be able to insert it at the correct side of the central C-node later on.

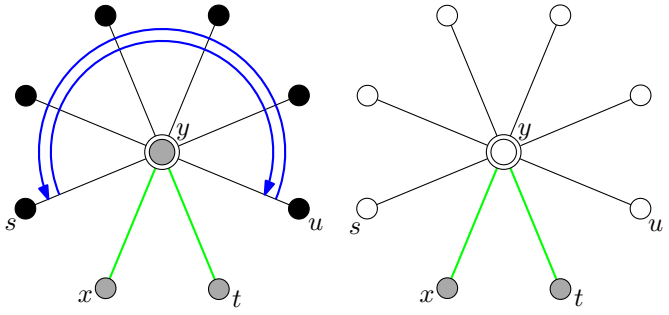


Figure 3.17.: C-nodes with only one block.

Case 3: If $y$ is not a terminal node and none of both neighboring arcs is a terminal path arc, $(x, y)$ has a full neighbor $(s, y)$ and an empty neighbor $(t, y)$, as shown in Figure 3.18. We keep pointers to $(s, y)$ as one end of the full block and to $(t, y)$ as one end of the empty block. We follow the block pointer of $(s, y)$ to $(w, y)$, which is the other end of the full block. Arc $(w, y)$ has to be adjacent to the other terminal path arc $(v, y)$, which in turn has an empty neighbor $(u, y)$. We set $(u, y)$ as the other end of the empty block.

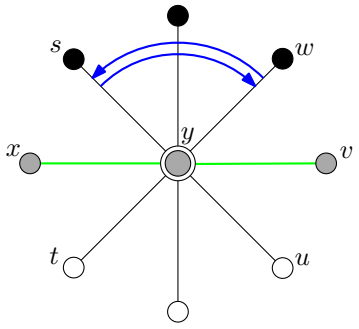


Figure 3.18.: A C-node with a full and an empty block.

After obtaining pointers to the ends of the blocks at C-nodes, we delete all edges between nodes on the terminal path. We create a new C-node $c$ that will preserve the order of all

edges incident to nodes on the terminal path. C-node $c$ keeps pointers to its last inserted full and empty arc to allow inserting new arcs at the correct position. We iterate through all nodes on the terminal path again, splitting every node $x$ in two separate nodes. The new node $x_f$ holds all edges to full neighbors, $x_e$ holds all edges to empty neighbors. We insert edges to $x_f$ and $x_e$ at the end of the respective blocks at $c$.
If $x$ is a C-node, explicitly creating $x_f$ and $x_e$ is unnecessary, because we will contract the edges $cx_f$ and $cx_e$ anyway. Instead, we use the pointers to the arcs at the ends of the full and the empty block of $x$ obtained in the earlier step and directly add them to the full and the empty block of $c$. Note that the direction of the inserted block is important, i.e., the last arc of the block of $c$ has to be adjacent to the first arc of the block of $x$.

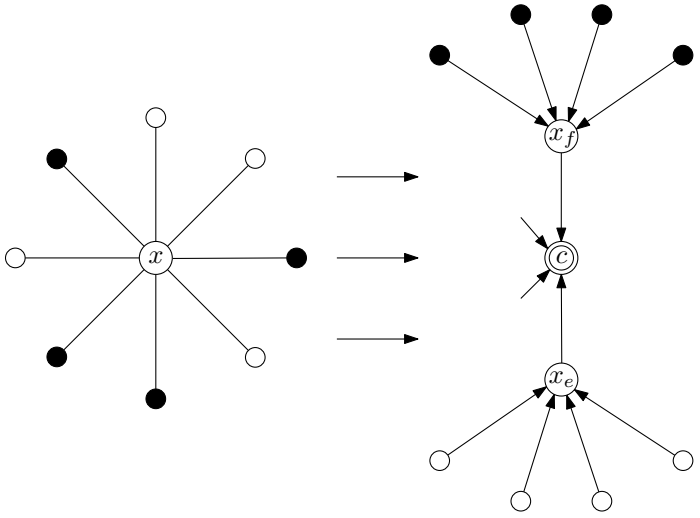


Figure 3.19.: Splitting a P-node.

If $x$ is a P-node, we iterate through the list of arcs entering $x$ from its full neighbors. We extract every full arc $(y, x)$ at $x$ and insert it at $x_f$. Node $x$ remains with only empty neighbors, therefore we rename $x$ to $x_e$. Following this, we create new edges $cx_e$ and $cx_f$, as shown in Figure 3.19 and we update the degrees of $x_e$ and $x_f$. If $x$ was not the apex, we set $c$ as the parent of $x_e$ and $x_f$, because we have deleted the parent edge of $x$. If $x$ was the apex, let $x_p$ be the node of $x_f$ and $x_e$ that retains the parent edge and let $x_o$ be the other node. We set $c$ as the parent of $x_o$ and $x_p$ as the parent of $c$. This guarantees a correct parent function after the update step.
In case $x_f$ or $x_e$ has degree 2, it is redundant. Therefore, we delete that node and add node $y$ adjacent to it directly to $c$, as shown in Figure 3.20.
If $c$ only has degree 2 after splitting all nodes, we delete it and make its two neighbors adjacent. This is only the case if the terminal path consists of a single P-node.

Splitting a C-node takes O(1) time. Splitting a P-node takes time proportional to the number of its full neighbors. Since the number of full neighbors adjacent to the terminal path is in $O(k)$, splitting all nodes takes $O(p + k)$ time. Creating all new edges to the new C-node $c$ takes $O(p)$ time. Therefore, updating the terminal path takes $O(p+k)$ time total.

See Appendix B for pseudo-code illustrating the update step.

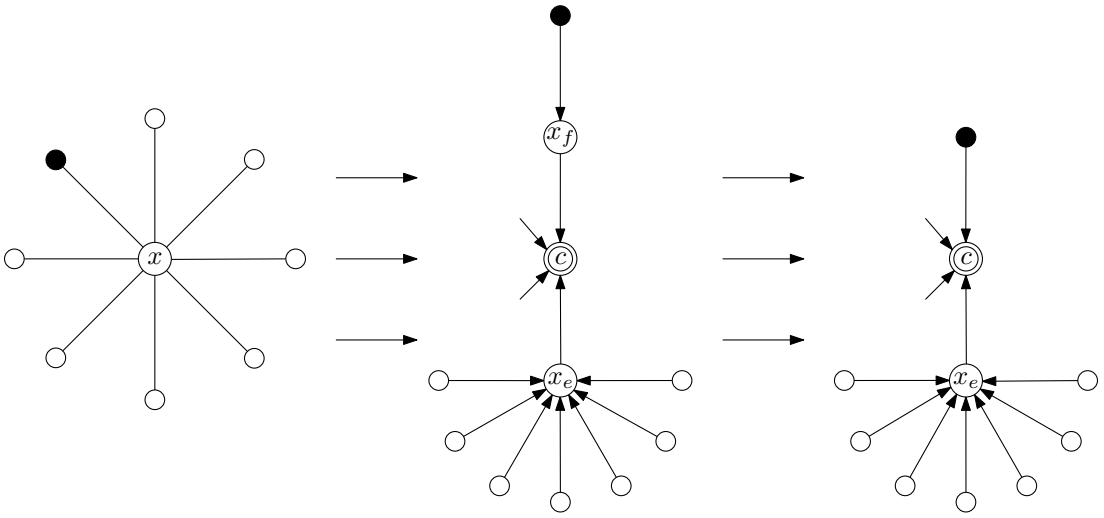


Figure 3.20.: Splitting a P-node with only one full neighbor.

## 3.4. Linear-time intersection of PC-trees

Let $T_1$ be a PC-tree over ground set $V$ with restrictions $R_1$ and let $T_2$ be a PC-tree over ground set $V$ with restrictions $R_2$. The *intersection* of $T_1$ and $T_2$ is a PC-tree $T_1 \cap T_2$ over the same ground set $V$ with restrictions $R_1 \cup R_2$. Figure 3.21 gives an example of such an intersection. In this example, $T_2$ contains most of the restrictions represented by $T_1$. The only missing restrictions are $\{6, 7\}$ and $\{8, 9\}$. Therefore, set $R_2 \cup \{\{6, 7\}, \{8, 9\}\}$ contains all restrictions for the intersection $T_1 \cap T_2$.

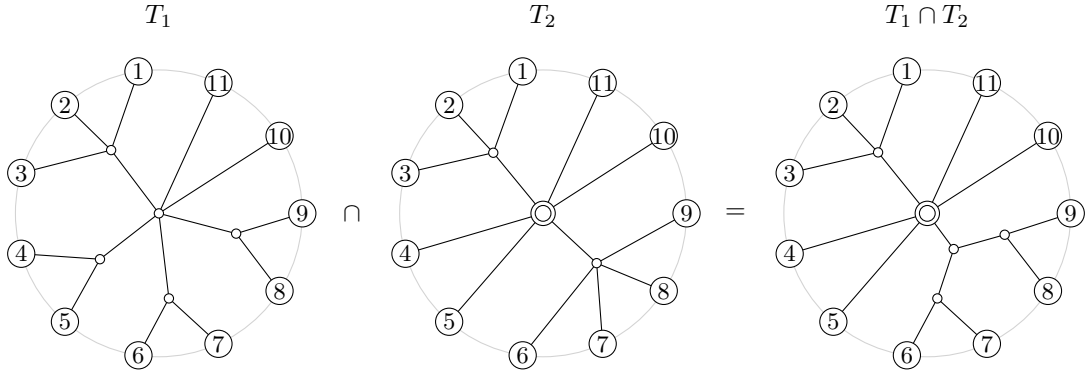


Figure 3.21.: Intersection of two PC-trees $T_1$ and $T_2$.

In order to compute the intersection of $T_1$ and $T_2$, we examine $T_2$ to find all restrictions $R_2$ it represents and apply every restriction in $R_2$ to $T_1$. Note that making all leaves of multiple subtrees in $T_1$ consecutive is too expensive, because every restriction may contain $\Theta(|V|)$ leaves and there can be $\Theta(|V|)$ restrictions in $R_2$. According to Theorem 2.1, applying all restrictions in $R_2$ to $T_1$ would therefore take $\Theta(|V|^2)$ time.
To avoid this, we use the linear-time intersection algorithm for PQ-trees by Booth [Boo75] and adapt it for PC-trees. After applying a restriction from $T_2$ to $T_1$, we merge the leaves we just made consecutive to a single leaf $a_i$ in both $T_1$ and $T_2$. Leaf $a_i$ represents the whole consecutive subtree. Further restrictions will only contain $a_i$ instead of the whole subtree, which helps us keep the running time within the linear time bound. After applying all restrictions to $T_1$, we can simply replace all merged leaves with the subtree they represent. The result is the intersection $T_1 \cap T_2$.

In every step of the algorithm, when we search for the next restriction in $T_2$, we need to find an inner node in $T_2$ that has at most one non-leaf neighbor. There are two options to achieve this.
The first is to implement the algorithm recursively, starting at an arbitrary node and merging all non-leaf neighbors recursively. Thus, the recursion finds the inner nodes with at most one non-leaf neighbor implicitly.
The second option is using iteration and determining the processing order of nodes in advance. This can be done using a labeling algorithm similar to the one in Section 3.3.1. We start by labeling every leaf in $T_2$ full and inform non-full neighbors using a breadth-first search. Whenever a node becomes full, we push its non-full neighbor into a queue and inform it once it is the first node in the queue. This way, all inner nodes become full in the exact order we want to process them for the intersection algorithm.
Both options produce the same result in the same asymptotic execution time. While the recursive option is much simpler to implement, it may lead to an overflow in the call stack for big inputs, because a recursive call occurs for every inner node in $T_2$.

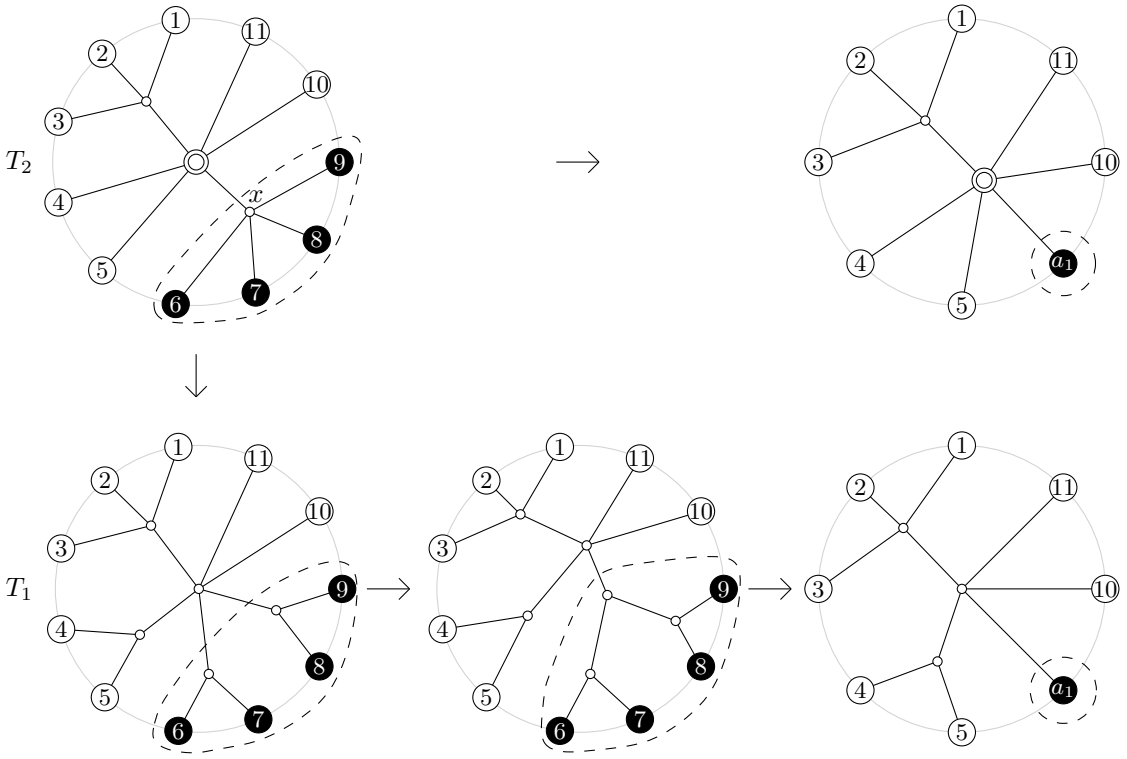


Figure 3.22.: Applying the restriction represented by a P-node in $T_2$ to $T_1$ and merging the leaves.

Given an inner node $x$ of $T_2$ with at most one non-leaf neighbor, $x$ represents different restrictions, depending on the type of $x$.
If $x$ is a P-node, it simply represents that all its adjacent leaves have to be consecutive. Therefore, we make all leaves adjacent to $x$ consecutive in $T_1$. After that, we merge these leaves in both $T_1$ and $T_2$, as shown in Figure 3.22. Since we will restore the merged subtrees in $T_1$ later, we store the entry point of the subtree in the merged leaf in $T_1$. This entry point is either a single edge incident to a P-node or it constitutes a consecutive block at the central C-node created in the update step when applying the restriction. In both cases, we can easily obtain pointers to the entry point during the update step.
If $x$ is a C-node, it represents a specific order of its incident leaves. Therefore, we cycle through the edges incident to $x$ and make the successive leaves pairwise consecutive in $T_1$, as shown in Figure 3.23. After this, we merge all leaves adjacent to $x$ in both $T_1$ and $T_2$,

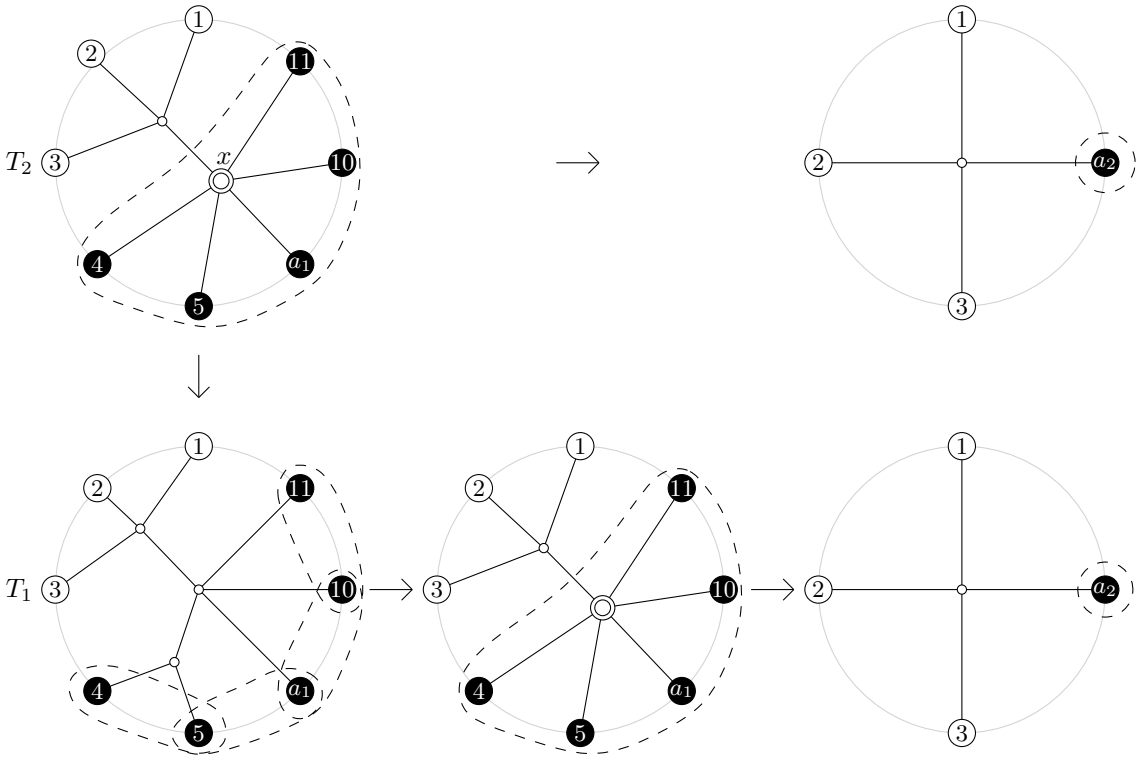


Figure 3.23.: Applying the restriction represented by a C-node in $T_2$ to $T_1$ and merging the leaves.

like in the previous case.

We stop merging leaves, once $T_2$ only contains a single inner node with all leaves adjacent to it. If this last node is a C-node, we make the successive leaves pairwise consecutive in $T_1$. If applying a restriction fails in any step of the algorithm, the intersection is not possible.
After applying all restrictions to $T_1$, we need to restore the merged subtrees. We traverse $T_1$ and whenever we find a merged leaf, we replace it with the subtree it represents, as shown in Figure 3.24. Note that this subtree may also contain merged leaves, i.e., we also need to traverse all restored subtrees. The resulting tree is $T_1 \cap T_2$.

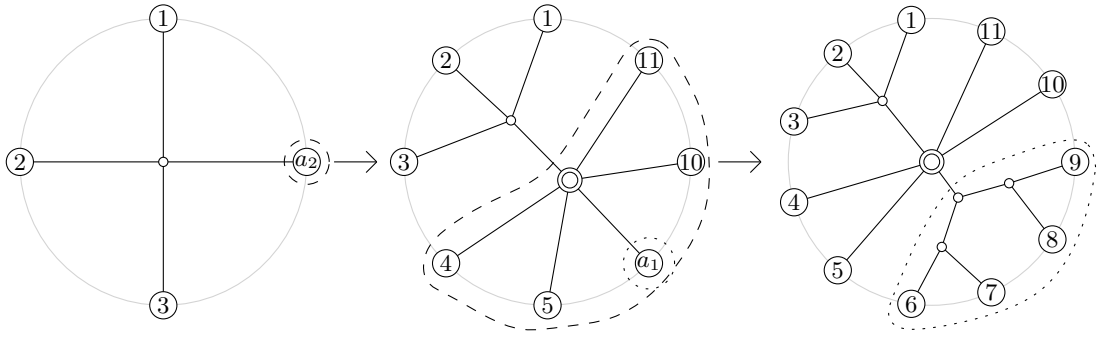


Figure 3.24.: Restoring the merged subtrees in $T_1$ yields the intersection $T_1 \cap T_2$.

**Theorem 3.1.** *Computing the intersection of two PC-trees $T_1$ and $T_2$ takes $O(|V|)$ time, where $V$ is the ground set of both trees.*

*Proof.* When applying the restrictions of $T_2$ to $T_1$, every leaf in $V$ is part of one restriction, if it is adjacent to a P-node or two restrictions, if it is adjacent to a C-node. Since we merge the leaves after applying the restriction, they cannot be part of any other restriction. If $T_2$ has $n$ inner nodes, we create exactly $n-1$ new leaves when merging subtrees. Each one of these new leaves is also part of only up to two restrictions. Due to the structure of a PC-tree, $n$ has to be in $O(|V|)$. Thus, the combined number of leaves in all applied restrictions is also in $O(|V|)$. According to Theorem 2.1, applying all restrictions to $T_1$ therefore takes $O(|V|)$ time.
Merging leaves after applying a restriction takes $O(1)$ time. Replacing a merged leave with its respective subtree also takes $O(1)$ time. Restoring the whole tree therefore takes time proportional to the size of the whole tree, which is also in $O(|V|)$. This means computing the intersection of $T_1$ and $T_2$ takes $O(|V|)$ time. □

# 4. Evaluation

To ensure the linear time bound of our implementation of the algorithms for applying restrictions and for intersections, we utilize two similar planarity tests that use our PC-tree implementation.
For the evaluation, we use the following platform to run the planarity tests:

- Cluster *Chimaira* at the University of Passau
- Linux Kernel Version 4.19
- CPU: Intel Xeon E5-2690v2 @ 3.00 GHz, 10 Cores, 20 Hyperthreads
- 64 GiB RAM
- OGDF Version 2020.02 (Catalpa)

## 4.1. Applying restrictions

In order to evaluate the performance of the algorithm for applying restrictions on PC-trees, we use the planarity test by Booth and Lueker [BL76]. An implementation of this algorithm using PQ-trees as the underlying data structure is available in OGDF. We implement the same test using our PC-tree as the underlying data structure and measure the time it takes to apply restrictions on our PC-trees compared to PQ-trees.

### 4.1.1. The planarity test

For simplicity, the test assumes that the graph $G$ to be tested for planarity is biconnected, i.e., removing an arbitrary node from the graph does not disconnect the graph. As the first step, we use OGDF to compute an st-numbering for $G$. If $G$ has $n$ nodes, the source node $s$ is assigned number 1 and the sink node $t$ is assigned number $n$. Every other node in $G$ has both neighbors with a smaller number and neighbors with a higher number. The source and sink of the st-numbering can be chosen arbitrarily. For every node $x$, we call edges to nodes with a lower number than $x$ *incoming edges* and edges to nodes with a higher number *outgoing edges*. Figure 4.1 shows an example of a biconnected graph with an st-numbering. The labels of the nodes represent their position in the st-numbering.

The planarity test works as follows. We iterate through all nodes in $G$ in the order defined by the st-numbering. For the first node $x_1$, we create a new PC-tree $T$ with a single P-node

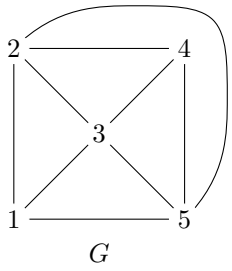


Figure 4.1.: A biconnected planar graph $G$ with five nodes.

and create a leaf for every edge incident to $x_1$. For every other node $x$, we inspect all its incident edges and separate the incoming edges from the outgoing edges. For every incoming edge, there already exists a leaf in $T$ that represents it. We make the leaves in $T$ that represent the incoming edges consecutive. After that, we merge these consecutive leaves to a single leaf, i.e., we extract the whole subtree of the consecutive leaves from the tree and replace it with a single leaf. We replace this merged leaf with a new P-node and add a leaf to it for every outgoing edge of $x$.
If at any point of the algorithm a set of leaves cannot be made consecutive, graph $G$ is not planar, as shown by Booth and Lueker [BL76]. If applying the restriction to $T$ succeeds for the penultimate node in $G$, we know that $G$ is planar.

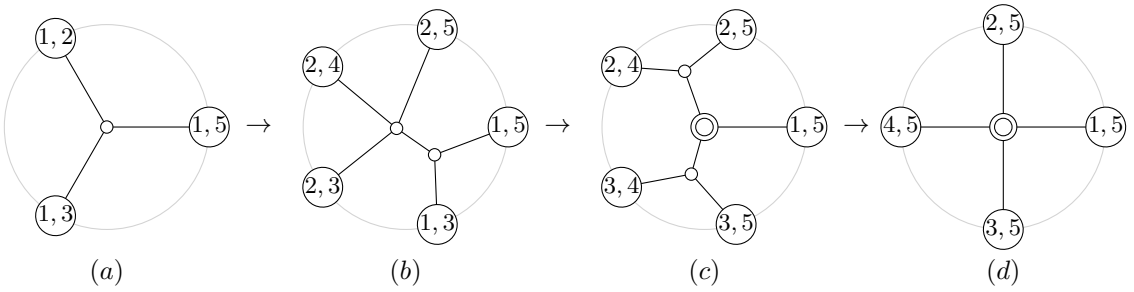


Figure 4.2.: The state of the PC-tree after every step of the planarity test.

Figure 4.2 shows the state of $T$ after every step of the algorithm for the graph in Figure 4.1, the labels of the leaves refer to their respective edge in $G$. In step $(a)$, we create $T$ with a single P-node and a leaf for every of the three outgoing edges of node 1 in $G$. In step $(b)$, making the leaves that represent the incoming edges of node 2 consecutive is trivial, since it is only edge $(1, 2)$. We replace the leaf with a new P-node and add a leaf for every outgoing edge of node 2. In step $(c)$, we make the leaves that represent edges $(2, 3)$ and $(1, 3)$ consecutive, we merge them and replace them with a P-node with leaves for edges $(3, 4)$ and $(3, 5)$. In step $(d)$, we make the leaves $(2, 4)$ and $(3, 4)$ consecutive. Since node 4 only has one outgoing edge, we replace the merged leave directly with leaf $(4, 5)$. Since a restriction that contains all leaves of $T$ is trivial, there is no need to process node 5. By the results of the test, graph $G$ is planar.

### 4.1.2. Evaluation results

We run the planarity test on different graphs and measure the time it takes to apply the restrictions on the PC-tree compared to the PQ-tree.
As the first test, we use OGDF to generate multiple random planar graphs with different node and edge counts and measure the combined time of all restrictions of the planarity test. We create graphs with node count $n \in \{100, 500, 1000, 5000, 10000, 50000, 100000,$

200000, 250000, 300000, 400000, 500000, 600000, 700000, 750000, 800000, 900000, 1000000} and edge count $m \in \{n, 2n, 3n-6\}$. For every graph size, we generate ten different graphs in order to minimize variations while measuring. Note that it may be necessary to manually increase the stack size of the program for large graphs, because the st-numbering in OGDF is implemented recursively.
Figure 4.3 shows the results for edge count $m = 3n - 6$, the maximum edge count for a planar graph. As the plot shows, the PC-tree is over 20% faster at applying all restrictions for big graphs compared to the PQ-tree.

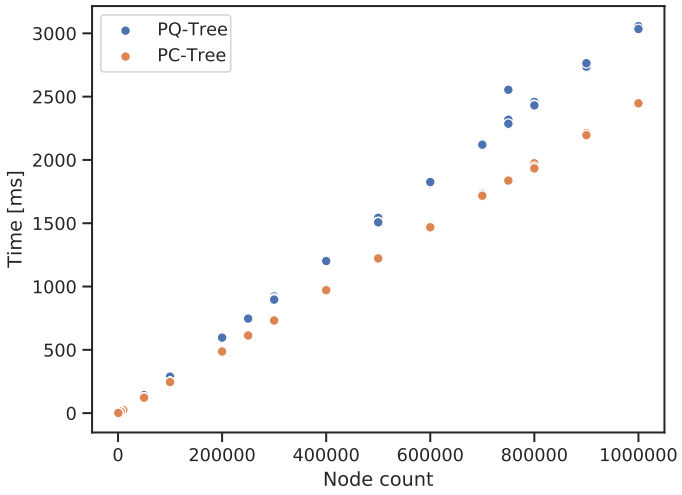


Figure 4.3.: Combined time of all restrictions for $m = 3n - 6$.

Figure 4.4 shows the results for $m = 2n$. In this case, the PC-tree is over 30% faster for graphs with a high node count.

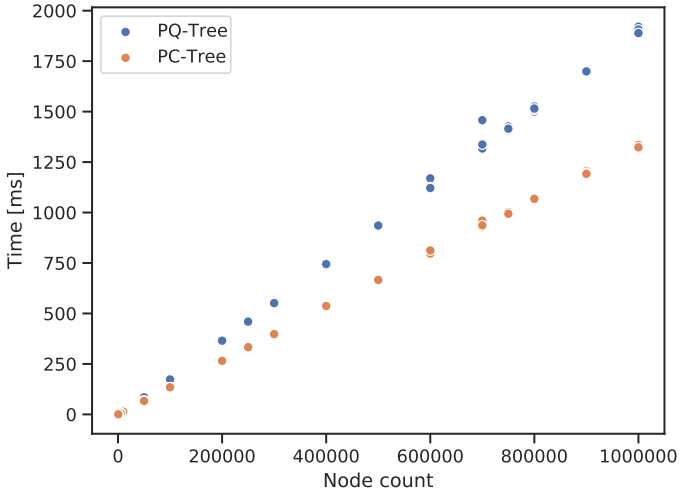


Figure 4.4.: Combined time of all restrictions for $m = 2n$.

Figure 4.5 shows the results for $m = n$. The PC-tree is about 95% faster at applying all restrictions compared to the PQ-tree. Note that for $m = n$ the planarity test only applies restrictions of size 1 in every step. This means that Figure 4.5 only indicates that the PC-tree is significantly faster at applying trivial restrictions of size 1. Therefore, the

previous tests are more suitable for comparing the overall performance, since they also apply restrictions of much bigger size.

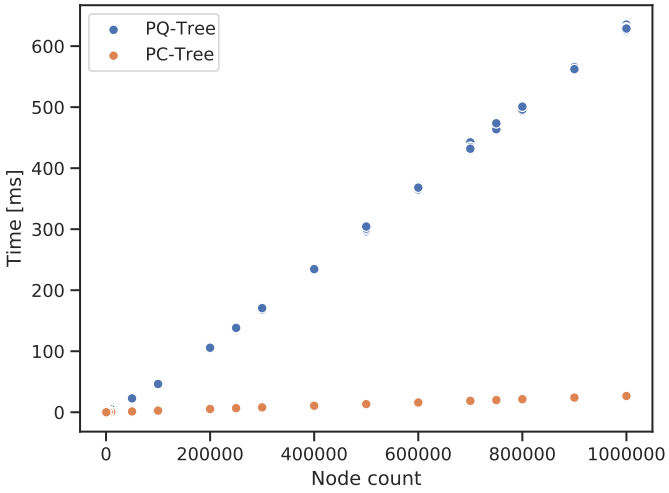


Figure 4.5.: Combined time of all restrictions for $m = n$.

As the second test, we consider random non-planar graphs of different sizes. This time, we create graphs with node count $n \in \{1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000\}$ and edge count $m \in \{10n, 20n, 30n\}$. We generate ten different graphs for every graph size. Figure 4.6 shows the results for all graphs. Since the planarity test stops once it detects that the graph is not planar, we only consider the number of processed nodes in the plot. Although the results are not as consistent as in the previous test, the plot clearly indicates that the PC-tree is significantly faster at applying all restrictions. On average for all processed non-planar graphs, the PC-tree is 70% faster than the PQ-tree.

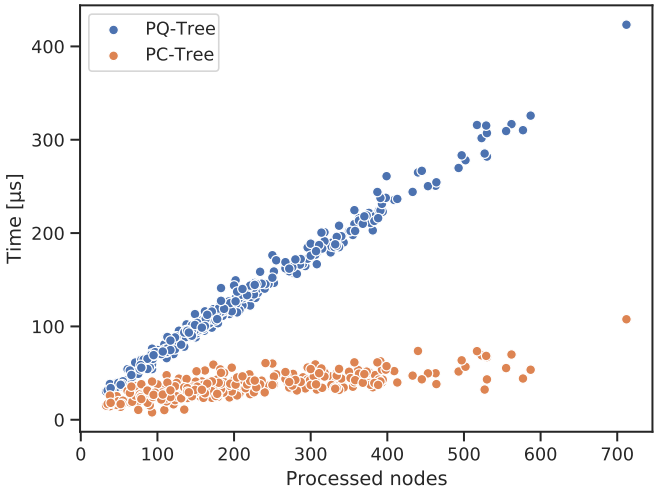


Figure 4.6.: Combined time of all restrictions for non-planar graphs.

As the last test, we measure the time for every individual restriction. Again, we use the random planar graphs from the first test with $m = 3n - 6$, because they create a wide range of restriction sizes. Figure 4.7 shows the required time for every restriction with a size over 25 and below 3000, which results in a total of 985992 samples. Figure 4.8 shows

the speedup of these PC-tree restrictions compared to their related PQ-tree restrictions. The restriction size is grouped into intervals of size 100 and the plot shows the distribution for every interval, with outliers being ignored. For small restriction sizes, the performance advantage of the PC-tree is rather low, but increases with rising restriction sizes until it hits a local peak of almost 50% at around 400. Restrictions with size around 1000 result in about 40% performance increase. Restriction sizes higher than that approach a performance advantage of almost 50%. Restrictions of size smaller than 25 follow the trend visible in the graph, the minimum performance increase is about 15% for restrictions of size 2.

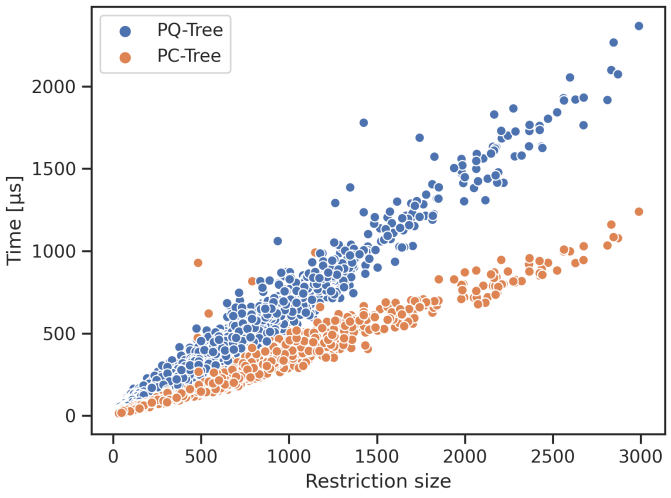


Figure 4.7.: Individual time for restrictions with size between 25 and 3000.

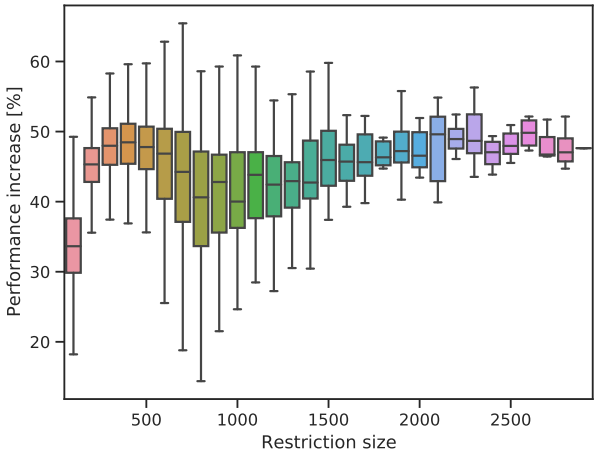


Figure 4.8.: Performance increase of restrictions on the PC-tree compared to the restrictions on the PQ-tree by restriction size.

Figure 4.9 uses the same data as Figure 4.8, but the restrictions are ordered by their terminal path length instead of their size. The restrictions are grouped by their terminal path length in intervals of size 5. The PC-tree is about 35% faster for small terminal path lengths, but the performance advantage increases to over 50% with rising terminal path lengths. A possible reason for this could be the template matching in the PQ-tree

implementation. For every node on the terminal path, the PQ-tree checks up to nine different templates for the position of the node in the tree before updating the node. Since our implementation simply splits every node on the terminal path without much preparation, this could explain the increasing performance difference for rising terminal path lengths.

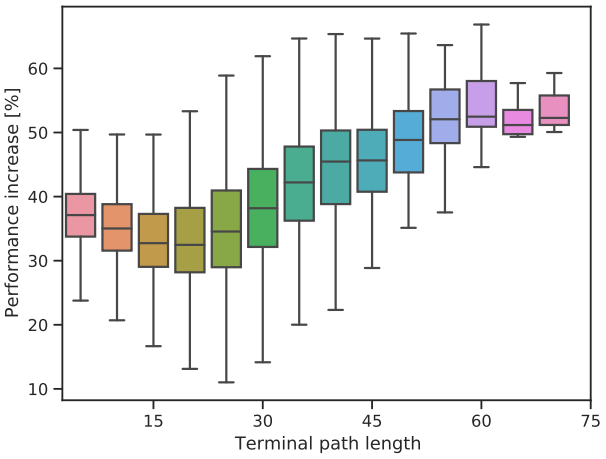


Figure 4.9.: Performance increase of restrictions on the PC-tree compared to the restrictions on the PQ-tree by terminal path length.

## 4.2. Intersection

In order to evaluate the running time of our implementation of the intersection algorithm from Section 3.4, we additionally implement the naive intersection algorithm that does not merge leaves after applying a restriction. We use a planarity test similar to the one from Section 4.1 that makes use of the intersection and compare the running time of our intersection algorithm to the naive intersection algorithm.

### 4.2.1. The planarity test

Given a biconnected graph $G$, we use OGDF to compute an st-numbering of $G$ and cut $G$ along the middle of the resulting ordering. For both of the resulting sets of nodes, we conduct the planarity test from Section 4.1, starting at the source and sink of the st-numbering. If one of the two tests fails, the graph is not planar. Otherwise, the two resulting PC-trees $T_1$ and $T_2$ both have the same number of leaves representing the edges of $G$ that cross the cut. If the intersection of $T_1$ and $T_2$ is possible, graph $G$ is planar.

### 4.2.2. Evaluation results

As the input for the planarity test, we use planar graphs with node count $n \in \{100, 500, 1000, 5000, 10000, 50000, 100000, 200000, 250000, 300000, 400000, 500000, 600000, 700000, 750000, 800000, 900000, 1000000\}$ and edge count $m = 3n - 6$. Again, we generate ten different graphs for every graph size.
Figure 4.10 shows the required time for the merge intersection and the naive intersection, depending on the number of leaves of the intersected trees. The running time of the merge intersection is clearly linear in the number of leaves. Figure 4.11 shows the performance increase of the merge intersection compared to the naive intersection. The samples are

grouped in intervals of size 1000. For small trees, the speedup is negligible, but it rapidly increases with rising tree sizes, because the naive intersection is asymptotically slower than the merge intersection. As pointed out in Section 3.4, the naive intersection takes quadratic time in the number of leaves.

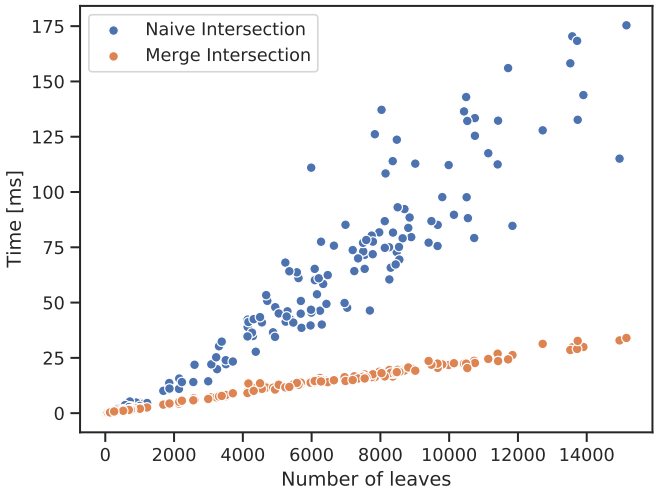


Figure 4.10.: Required time for intersections by number of leaves.

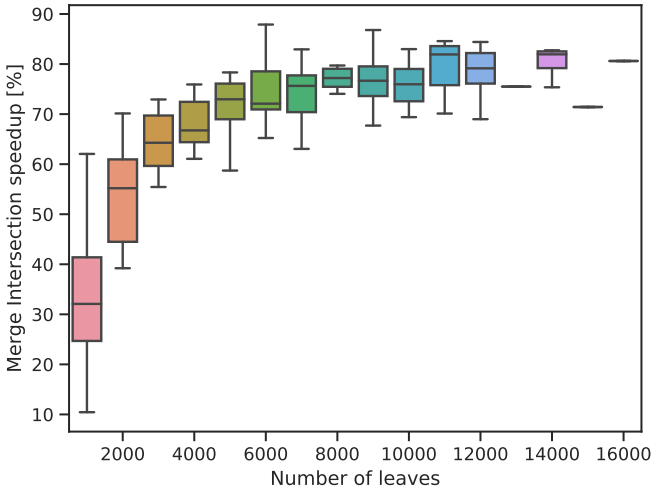


Figure 4.11.: Performance increase of the merge intersection compared to the naive implementation.

# 5. Conclusion

In this thesis we introduced PC-trees and characterized their functionality. We gave a detailed description of the implementation of an algorithm for applying restrictions on PC-trees in linear time, based on the algorithm by Hsu and McConnell [HM03, HM04]. This implementation is much shorter and simpler, compared to the implementation of the PQ-tree by Booth and Lueker [BL76] in OGDF, because the lack of a distinguished root node in our implementation renders the numerous templates in the PQ-tree implementation unnecessary. We implemented the planarity test by Booth and Lueker using PQ-trees and PC-trees in order to compare their performance and to test the correctness of our implementation. Our benchmarks showed that our implementation is significantly faster at applying restrictions compared to the PQ-tree, while still producing the same results.
We also implemented the first known linear-time algorithm for the intersection of two PC-trees, where we ensure the linear-time bound by merging the applied restrictions to a single leaf. Replacing these leaves with the original subtree gives us the intersection. This approach is based on the intersection algorithm for PQ-trees by Booth [Boo75]. We also validated the correctness and the linear time bound of our intersection implementation by comparing it to the naive implementation, which does not merge leaves and takes quadratic time.
Overall, our resulting PC-tree implementation in C++ allows applying restrictions and the intersection of two PC-trees in linear time. The data structure is sufficiently generic to enable assembling custom PC-trees and to store information at its nodes. The implementation also includes a linear-time planarity test.

Hsu and McConnell [HM03] showed that computing the PQ-tree reduces in linear time to computing the PC-tree, i.e., the PC-tree could replace the PQ-tree in all of its applications. Therefore, it may be interesting in the future to reconsider some of the many algorithms using PQ-trees and examine whether using PC-trees could improve the algorithm's simplicity and performance. For example, the level planarity test by Jünger et al. [JLM98] and the radial level planarity test by Bachmaier et al. [BBF05] could possibly benefit from the performance advantage of PC-trees. Reconsidering embedding problems related to the Simultaneous PQ-Ordering problem [BR13] may also be interesting.

# Bibliography


[BBF05] Christian Bachmaier, Franz J. Brandenburg, and Michael Forster. Radial Level Planarity Testing and Embedding in Linear Time. *Journal of Graph Algorithms and Applications*, 9:53–97, 2005.

[BL76] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13:255–379, 1976.

[Boo75] Kellogg S. Booth. *PQ-tree Algorithms*. PhD thesis, University of California, Berkeley, 1975.

[BR13] Thomas Bläsius and Ignaz Rutter. Simultaneous PQ-Ordering with Applications to Constrained Embedding Problems. In *Proceedings of the 24th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'13)*. SIAM, 2013.

[CGJ+14] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, chapter 17. CRC Press, 2014.

[HM03] Wen-Lian Hsu and Ross McConnell. PC trees and circular-ones arrangements. *Theoretical Computer Science*, 296:99–116, 2003.

[HM04] Wen-Lian Hsu and Ross McConnell. PQ Trees, PC Trees, and Planar Graphs. In Dinesh P. Mehta and Sartaj K. Sahni, editors, *Handbook of Data Structures and Applications*. CRC Press, 2004.

[JLM98] Michael Jünger, Sebastian Leipert, and Petra Mutzel. Level planarity testing in linear time. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *Lecture Notes in Computer Science*, pages 224–237. Springer, 1998.

[LEC67] Abraham Lempel, Shimon Even, and Israel Cederbaum. An Algorithm for Planarity Testing of Graphs. In *Proceedings of the International Symposium on the Theory of Graphs*, pages 215–232. Gordon and Breach, 1967.

[SH99] Wei-Kuan Shih and Wen-Lian Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.

# Appendix

## A. Pseudo-code terminal path

The following pseudo-code illustrates the steps of the terminal path algorithm.

**Algorithm 5.1:** FINDING THE TERMINAL PATH

**Input:** Queue $Q$ containing all partial nodes
**Output:** List $TP$ containing all nodes on the terminal path

```
// Initialization
originalPartialNodesCount ← Q.size()
pathCounter ← Q.size()
apexCandidate ← null
highestNode ← null
invalidRestriction ← false
// Main Loop
while !invalidRestriction and Q is not empty do
    currentNode ← Q.pop()
    if currentNode.marked then
        pathCounter ← pathCounter − 1
        continue
    currentNode.marked ← true
    if pathCount = 1 then
        highestNode ← currentNode
        break
    parent ← findParent(currentNode)
    if parent = null then
        highestNode ← currentNode
        continue
    setPredecessor(parent, currentNode)
    Q.push(parent)
if invalidRestriction then
    return
if apexCandidate = null then
    apexCandidate ← highestNode
    while apexCandidate is not partial do
        apexCandidate ← apexCandidate.predecessor1
partialNodesCount ← addAllPredecessors(apexCandidate, TP)
if originalPartialNodesCount ≠ partialNodesCount then
    invalidRestriction ← true
```

**Algorithm 5.2:** FINDING THE PARENT OF THE CURRENT NODE

**Input:** Node $n$ on the terminal path
**Output:** The parent of $n$, or `null`, if the parent of $n$ cannot be on the terminal path

```
1  if n.parentArc = null then
2      if n is partial C-node then
3          setApexCandidate(n)
4      return null
5  parent ← n.parentArc.yNode
6  if parent ≠ null then
7      if parent is full then
8          return null
9      else
10         return parent
11 parent ← null
12 nodeObjectNeighbor1 ← n.parentArc.neighbor1.yNode
13 nodeObjectNeighbor2 ← n.parentArc.neighbor2.yNode
14 if nodeObjectNeighbor1 ≠ null and nodeObjectNeighbor2 ≠ null then
15     if nodeObjectNeighbor1 ≠ nodeObjectNeighbor2 then
16         invalidRestriction ← true
17     else
18         parent ← nodeObjectNeighbor1
19 else if nodeObjectNeighbor1 ≠ null then
20     if nodeObjectNeighbor1 is partial and n.parentArc.neighbor1 is not an end
        of a full block then
21         invalidRestriction ← true
22     else
23         parent ← nodeObjectNeighbor1
24 else if nodeObjectNeighbor2 ≠ null then
25     if nodeObjectNeighbor2 is partial and n.parentArc.neighbor2 is not an end
        of a full block then
26         invalidRestriction ← true
27     else
28         parent ← nodeObjectNeighbor2
29 else
30     parent ← new C-node
31     n.parentArc.yNode ← parent
32     if n.parentArc.neighbor1 is not a parent arc then
33         parent.parentArc ← n.parentArc.neighbor1.twin
34     else if n.parentArc.neighbor2 is not a parent arc then
35         parent.parentArc ← n.parentArc.neighbor2.twin
36 return parent
```

**Algorithm 5.3:** Setting a predecessor

**Input:** Node *parent*, Node *predecessor*

```
1 if parent.predecessor2 ≠ null then
2 |   invalidRestriction ← true
3 else if parent.predecessor1 ≠ null then
4 |   if parent.predecessor1.parentArc = predecessor.parentArc then
5 |   |   invalidRestriction ← true
6 |   parent.predecessor2 ← predecessor
7 |   setApexCandidate(parent)
8 else
9 |   parent.predecessor1 ← predecessor
```

# B. Pseudo-code update step

The following pseudo-code illustrates the update step of the algorithm.

**Algorithm 5.4:** FINDING THE ENDS OF THE BLOCKS OF A C-NODE

**Input:** C-node $n$ on the terminal path, Arc *entryArc* entering $n$ on the terminal path, List $TP$ of nodes on the terminal path
**Output:** Pointers to the arcs at the ends of the full and empty blocks of $n$

```
   // Initialization
 1 fullNeighbor ← null
 2 otherNeighbor ← null
 3 if entryArc.neighbor1 is full then
 4 |   fullNeighbor ← entryArc.neighbor1
 5 |   otherNeighbor ← entryArc.neighbor2
 6 else if entryArc.neighbor2 is full then
 7 |   fullNeighbor ← entryArc.neighbor2
 8 |   otherNeighbor ← entryArc.neighbor1
   // Finding the blocks
 9 if n is first node in TP then
10 |   fullBlockLast ← fullNeighbor
11 |   fullBlockFirst ← fullNeighbor.blockPointer
12 |   emptyBlockLast ← otherNeighbor
13 |   emptyBlockFirst ← emptyNeighbor(fullBlockFirst)
14 else if n is last node in TP then
15 |   fullBlockFirst ← fullNeighbor
16 |   fullBlockLast ← fullNeighbor.blockPointer
17 |   emptyBlockFirst ← otherNeighbor
18 |   emptyBlockLast ← emptyNeighbor(fullBlockFirst)
19 else
20 |   if fullNeighbor ≠ null then
21 |   |   fullBlockFirst ← fullNeighbor
22 |   |   fullBlockLast ← fullNeighbor.blockPointer
23 |   |   if otherNeighbor is not terminal path arc then
24 |   |   |   emptyBlockFirst ← otherNeighbor
25 |   |   |   emptyBlockLast ←
   |   |   |     getNextArc(fullBlockLast, emptyNeighbor(fullBlockLast))
26 |   else if entryArc.neighbor1 is terminal path arc then
27 |   |   emptyBlockFirst ← entryArc.neighbor2
28 |   |   emptyBlockLast ← getNextArc(entryArc, entryArc.neighbor1)
29 |   else
30 |   |   emptyBlockFirst ← entryArc.neighbor1
31 |   |   emptyBlockLast ← getNextArc(entryArc, entryArc.neighbor2)
```

**Algorithm 5.5:** Splitting a P-node

**Input:** P-node $n$ on the terminal path, Central C-Node $c$

```
1  fullParentArc ← false
2  fullPNode ← new P-node
3  forall Arc fullArc ∈ n.fullNeighbors do
4  |  n.extractArc(fullArc)
5  |  fullPNode.addArc(fullArc)
6  |  if fullArc is part of parent edge of n then
7  |  |  fullParentArc ← true
8  fullPNode.degree ← n.fullNeighborsCount + 1
9  n.degree ← n.degree − n.fullNeighborsCount + 1
10 createNewEdge(c, fullPNode)
11 if fullParentArc and n is apex then
12 |  c.setParent(fullPNode)
13 else
14 |  fullPNode.setParent(c)
15 if fullPNode.degree = 2 then
16 |  condenseNode(fullPNode)
17 createNewEdge(c, n)
18 if not fullParentArc and n is apex then
19 |  c.setParent(n)
20 else
21 |  n.setParent(c)
22 if n.degree = 2 then
23 |  condenseNode(n)
```