Institut für theoretische Informatik

Leibniz Universität Hannover

Masterarbeit

Completeness Results for Graph Isomorphism on Restricted Graph Classes

von Maurice Chandoo

Oktober 2014

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst habe und keine anderen Hilfsmittel und Quellen als angegeben verwendet habe.

_____

Maurice Chandoo

Contents

Preface **2**

1 Preliminaries **4**

2 Trees **7**
2.1 Tree Representations 7
2.2 Linear order on tree isomorphism classes 10
2.3 Undirected trees, forests, colored trees 16
2.4 Hardness results 18

3 k-Trees **22**
3.1 Canonically labeling k-Trees 23
3.2 Logspace implementation 28
3.3 Hardness results 29

4 Helly Circular-Arc graphs **32**
4.1 Turning HCA graphs into interval matrices 34
4.2 Calculating the Δ Tree of an interval matrix 41
4.3 Canonically choosing an interval orientation 49
4.4 CA graph isomorphism 53

List of Figures, Tables and Algorithms . . . . . **i**

Bibliography . . . . . **iii**

Preface

The graph isomorphism problem GI is to decide whether two graphs G, H are isomorphic. Formally, this means to determine if there exists a bijection π from the vertex set of G to the vertex set of H such that there is an edge between u and v in G if and only if there is one between $\pi(u)$ and $\pi(v)$ in H as well. Such a bijection π is called an isomorphism between G and H. From the perspective of computational complexity it is clear that GI is in NP since a witness of polynomially bounded length is given by an isomorphism. However, despite much research effort since one of the first mentions of this problem by [Kar72] neither a proof for NP-hardness nor a polynomial-time algorithm for GI have been found. One of the reasons for the enticing nature of this problem can be attributed to the fact that it is not only easy to state and understand but also appears as very approachable and deceivingly promising to resolve at first glance. So much in fact that it has been labeled as the graph isomorphism disease because of its contagious nature in [RC77] which evidently has infected me as well.

The algorithm with the best known asymptotic runtime for solving GI is mentioned in [BL83] and runs in $2^{c\sqrt{n \log n}}$ for some constant c. The best known hardness result states that GI is hard for the class DET which is the class of problems NC^1 reducible to the determinant [Tor04]. Here is an overview of the complexity classes mentioned in this thesis and related ones; partially taken from [Köb06]:

$$\mathsf{AC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{DET} \subseteq \mathsf{TC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC} \subseteq \mathsf{P}$$

Another remarkable result reveals that the polynomial-time hierarchy collapses to its second level if GI happens to be NP-hard [Sch88]. If one believes such a collapse to be unlikely then one must consequently reckon that $\mathsf{P} \neq \mathsf{NP}$ and GI is in the class of NP intermediate problems, also known as NPI. This class consists of the problems which are in NP but neither in P nor NP-hard. In the context of the possibility that $\text{GI} \in \mathsf{NPI}$ it seems reasonable to define a complexity class GI that contains every problem which is polynomial-time Turing-reducible to GI. Additionally, a problem is GI-complete if it is in GI and there is a polynomial-time Turing reduction from GI to that problem. That means a problem A is GI-complete if and only if $\mathsf{P}^A = \mathsf{P}^{\text{GI}}$ in oracle notation. It has been shown that the isomorphism problem for various other structures is GI-complete, for instance hypergraphs, finite automata and context-free grammars [ZKT85]. Analogously, it has been shown that there are restricted classes of graphs for which the isomorphism problem remains GI-complete. An example for which this can be easily seen is the class of connected graphs.

Fortunately, for sufficiently restricted graph classes it can be shown that the isomorphism problem is tractable. In this thesis we shall consider three graph classes for which

the isomorphism problem is L-complete. These three classes are trees, k-trees and the somewhat exotic sounding class of Helly circular-arc (HCA) graphs. We will see that we are not only able to test if two members of one of these classes are isomorphic but that we can compute a canonical form for them, which is also useful from a practical point of view. Imagine you have a database table where one of the columns contains a tree and you want to query if any two rows have isomorphic trees. By precomputing the canonical form and storing it in the database the expensive operation of testing two trees for isomorphism can be replaced by simply comparing the strings of their canonical forms.

The fact that trees can be canonically labeled space-efficiently is a fundamental insight, which has proven to be a very useful tool in this research branch. For instance, both of the two subsequent results about k-trees and HCA graphs make use of it. Moreover, the algorithm which computes the canonical form of a tree employs the resources available in logspace in a novel and witty way to execute a recursion of linear depth which would normally be considered to be a prohibitive operation in a logspace setting. Thus, this idea might be relevant in itself when searching for new logspace algorithms.

The class of k-trees can be seen as generalization of trees such that for $k = 1$ both classes coincide. Usually, they are described as graphs with tree width k such that adding any new edge leads to a graph with tree width $k+1$. For unbounded k the isomorphism problem for k-trees is GI-complete [ADKK12] which makes this class interesting from a fixed-parameter tractable point of view. Indeed, [ADKK12] have also supplied an implementation of their algorithm which runs in $\mathcal{O}((k+1)!\,n)$ time. Furthermore, a partial k-tree is a subgraph of a k-tree and the set of all partial k-trees is the set of graphs with tree width at most k. An extension of this result to partial k-trees would therefore imply that GI is fixed-parameter tractable with respect to tree width. It should be mentioned that a quite recent result has possibly shown that GI is in fact fixed-parameter tractable w.r.t. tree width [LPPS14]. Another result in this regard is that the class of partial 2-trees – which happens to coincide with the class of series-parallel graphs – can be canonized in logspace [ADK08].

HCA graphs belong to the broader class of geometric intersection graphs where vertices are associated with certain geometrical objects – in this case circular arcs arranged on a circle – and there is an edge between two vertices if their respective geometrical objects intersect. To arrive at the conclusion that HCA graphs can be canonically labeled in logspace various interesting concepts such as transitive orientations and modular decompositions had to be adequately modified and combined. The two mentioned concepts have been previously applied in the context of permutation and comparability graphs in [MS99]. For the more general class of circular-arc graphs tractability of the isomorphism problem still remains an intriguing open problem which we will address at the end of the last chapter.

1 Preliminaries

A directed graph D is a tuple (V, E) with vertex set V and edge set $E \subseteq V \times V$ without loops, i.e. $(v, v) \notin E$. We also write $V(D), E(D)$ for V, E. An undirected graph G is a directed graph with symmetric edge relation. The vertex set of any graph can be assumed to be linearly ordered. The set of all undirected graphs is denoted by $\mathcal{G}$. A square matrix $A = (A_{u,v})_{u,v \in V}$ with entries 0 and 1 on a vertex set V is the adjacency matrix of a digraph $D = (V, E)$ if for all vertices $u, v \in V$ it holds that $A_{u,v} = 1$ iff $(u, v) \in E$. The vertex set of a square matrix A is also denoted by $V(A)$. As only square matrices are considered the qualifier square will be omitted henceforth.

Hypergraphs are a generalization of graphs such that for a hypergraph $H = (V, E)$ the set E is a family of sets over V, i.e. a subset of the power set of V excluding the empty set.

Two matrices A, B are isomorphic, in symbols $A \cong B$, if there exists a permutation $\pi : V(A) \to V(B)$ such that $A_{u,v} = B_{\pi(u),\pi(v)}$ for all $u, v \in V(A)$. The permutation π is called an isomorphism. Two graphs G, H are **isomorphic**, in symbols $G \cong H$, iff their adjacency matrices are isomorphic. If π is an isomorphism from G to itself then π is called an **automorphism** of G. The identity permutation is called the trivial automorphism. A graph G is called rigid if it has only the trivial automorphism. For a permutation π on the vertex set of G we write $\pi(G)$ to denote the graph obtained after relabeling G according to π.

A graph property $f : \mathcal{G} \to \mathbb{N}$ is called an **invariant** if $f(G) = f(H)$ whenever $G \cong H$ for all graphs G, H. If the reverse directions holds as well, i.e. $G \cong H$ whenever $f(G) = f(H)$, then f is called a **complete invariant**. Additionally, if the image of a complete invariant f can be interpreted as graph $f(G) = G'$ and $G' \cong G$ then G' is a **canonical form** of G and f is a canonization function. A **canonical labeling** is an isomorphism between G and its canonical form G'.

An example of a canonization function is a function f which given a graph G returns the lexicographically smallest graph G' in the isomorphism class of G with respect to a certain graph encoding.

The **graph isomorphism problem** GI is to decide for two given graphs G, H whether they are isomorph. The graph automorphism problem GA is to decide whether a given graph G has a non-trivial automorphism. The complete invariant problem CINV is to compute a complete invariant. Analogously, the **canonization problem** CANON is to compute a canonization function. Obviously, GI is reducible to CINV and CINV to CANON using constant-depth reductions. It is also not hard to show that GA is logspace Turing-reducible to GI, in fact even polynomial-time many-to-one reducible [KST93].

GI and GA can also be defined for restricted graph classes. Let $\mathcal{C} \subset \mathcal{G}$ be a graph class, then for instance a pair of graphs G, H is in GI($\mathcal{C}$) iff G and H are in $\mathcal{C}$ and both

are isomorphic. The **recognition problem** for a graph class $\mathcal{C}$ is to decide whether a given graph G belongs to $\mathcal{C}$. In the following two chapters we investigate the complexity of the restricted versions of the graph isomorphism problem for trees, k-trees and HCA graphs.

An extension of the graph isomorphism problem is the isomorphism problem for vertex-colored graphs. A **colored graph** is a tuple (G, c) with graph G and coloring $c : V(G) \to \{0,1\}^*$. Two colored graphs $(G, c), (H, d)$ are isomorphic iff there exists an isomorphism $\pi : V(G) \to V(H)$ between G and H such that $c(v) = d(\pi(v))$ for all $v \in V(G)$. We say π respects the coloring. The same definition of isomorphism can be applied to vertex-colored matrices.

We say two sets A, B **intersect** if they have a non-empty intersection. Two sets A, B **overlap**, in symbols $A \between B$, if $A \cap B$, $A \setminus B$, $B \setminus A$ are all non-empty. The set $\{1, \ldots, n\}$ for any natural number n is abbreviated with $[n]$.

Given a graph G and a subset $V' \subseteq V(G)$ we say that G' is the vertex-induced subgraph of G for the vertex set V' if $G' = (V', E(G) \cap (V' \times V'))$. A connected component $C \subseteq V(G)$ of a graph G is an inclusion-maximal set of vertices such that there is a path between every pair of vertices in C. The **neighborhood** of a vertex v in a graph G is denoted by $\mathrm{N}(v)$ and consists of all vertices that are adjacent to v. The **closed neighborhood** of a vertex v in G is denoted by $\mathrm{N}[v]$ and defined as $\mathrm{N}(v) \cup \{v\}$. The **common neighborhood** of two vertices u, v denoted by $\mathrm{N}[u, v]$ is $\mathrm{N}[u] \cap \mathrm{N}[v]$. Two vertices u, v of a graph G are said to be **twins** if $\mathrm{N}[u] = \mathrm{N}[v]$. The inclusion-maximal set of all twins is called a **twin class**, i.e. a set of all vertices which have the same closed neighborhood. The set of all twin classes form a partition of the vertex set of a graph. A graph G is **twin-free** if every twin class consists of exactly one vertex. A **universal** vertex v of a graph G is a vertex such that $\mathrm{N}[v] = V(G)$ and an **independent** vertex v of G is a vertex such that $\mathrm{N}(v) = \emptyset$. Given two vertices u, v of a graph G the **distance** $d_G(u, v)$ between u and v is the the length of the shortest path between u and v. The **eccentricity** of a vertex $u \in V(G)$ is the longest distance to another vertex v, which is $ecc_G(u) = \max\{d_G(u, v) \mid v \in V(G)\}$. The **center** of a graph G consists of all vertices with minimal eccentricity.

A **logspace transducer** is a Turing machine with a read-only input tape, a work tape and a write-only output tape such that the work tape requires only logarithmic space with respect to the input size. Notice that such a transducer can have a polynomially sized output. A function f is said to be **logspace computable** if there exists a logspace transducer which outputs $f(x)$ on input x for all inputs x. The complexity class L contains all languages whose characteristic function is logspace computable. In abuse of notation we say f is in L to denote that f is logspace computable.

Let f and g be two logspace computable functions, then their composition $f \circ g$ is logspace computable as well. This will be used to present logspace algorithms in a modular way. The idea to compute $f(g(x))$ is to combine the logspace transducers M_f, M_g for f and g such that M_f is executed until a bit of $g(x)$ is needed. In that case halt the execution of M_f and execute M_g until the desired output bit is computed and then resume M_f. This construction can be iterated a constant number of times without violating the logspace restriction. Whenever we present a logspace algorithm that seems

to require to store more than is possible, e.g. the algorithm states to compute $f(x)$ which is of polynomial size and then to do further operations on $f(x)$, it is meant that there exists a logspace transducer which can compute $f(x)$ and if certain information of $f(x)$ is required it can be computed on the fly.

It is presumed that addition, subtraction and multiplication are known to be logspace computable. Additionally, we utilize the fact that [Rei05] has shown that given an undirected graph G and two vertices $u, v \in V(G)$ it can be decided in L whether there is a path between u and v. It follows that the connected components of an undirected graph can be calculated in logspace as well.

As we will talk about completeness results for L we need to specify with respect to what type of reduction the lower bounds hold. Unless stated otherwise the hardness results will hold under **constant-depth reductions** also known as L-uniform AC^0 reductions, i.e. decision problem A reduces to decision problem B if there exists a family of polynomially-sized circuits $(C_n)_{n\in\mathbb{N}}$ with constant depth which may use oracle gatters for B such that for an input x of length n the output of C_n on x is 1 iff $x \in A$. The uniformity condition means that there exists a logspace transducer M which on input 1^n outputs C_n.

2 Trees

In this chapter we show how to canonize directed trees in logspace following the ideas of [Lin92]. This result can be easily extended to undirected trees, forests and colored trees. This is shown in the third section. At the end the logspace hardness for tree isomorphism and tree automorphism is shown.

For a directed tree T we write $|T|$ to denote its size $|V(T)|$. For a node v of T we write T_v to denote the subtree induced by v. We use $\#v$ to denote the number of children of v.

2.1 Tree Representations

Graphs in general can be represented either by their adjacency matrix or their edge relation as list along with the number of nodes (to represent isolated vertices). We call the second kind pointer list representation. It is not hard to see that adjacency matrix and pointer list representation are logspace equivalent. That means that given either one we compute the other representation in L.

For trees there is yet another representation which is called string (of balanced parentheses) representation. It's inductive definition will help us to compute a canonical form of trees later on.

Definition 2.1. *Given a directed tree T with root node t and children $t_1, \ldots, t_k$ ordered according to their labels its string representation is inductively defined as*

$$str(T) = \begin{cases} \langle\rangle & \text{, if } T \text{ is the single node tree} \\ \left\langle str(T_{t_1}) str(T_{t_2}) \ldots str(T_{t_k}) \right\rangle & \text{, otherwise} \end{cases}$$

We remark that the complexity of deciding whether two trees given in their string representation are isomorphic is NC^1-complete. Membership was shown in [Bus97] and the hardness was proved in [JKMT03].

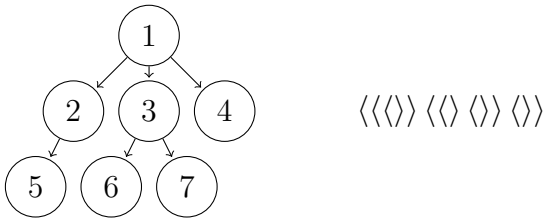

Figure 2.1: Exemplary directed tree T and its string representation $str(T)$

It is not quite obvious that the pointer list and string representation are logspace equivalent, which is in fact true. First, let us show how to compute the pointer list representation of trees that are given in string representation. As input we assume a string S of length $|S|$ with $S[i]$ being the i-th character of S.

Algorithm 2.1 Convert tree from string to pointer representation

```
 1: function stringToPointer(S)
 2:     n ← nodeLabel(|S|, S)
 3:     print "n;"
 4:     for i = 1 ... n do
 5:         for pos = 1 ... |S| do
 6:             if nodeLabel(pos, S) = i then break
 7:         if S[pos]S[pos + 1] = "⟨⟩" then continue
 8:         opened ← 0
 9:         for j = pos + 1 ... |S| do
10:             if S[j] = "⟨" then
11:                 if opened = 0 then print "(i, nodeLabel(j))"
12:                 opened ← opened + 1
13:             else if S[j] = "⟩" then
14:                 opened ← opened − 1
15: function nodeLabel(k, S)
16:     counter ← 0
17:     for i = 1 ... k do
18:         if S[i] = "⟨" then counter ← counter + 1
19:     return counter
```

The function nodeLabel in Algorithm 2.1 counts the number of occurrences of "⟨" in S up to the k-th character. For $k = |S|$ this number coincides with the the number of nodes that the tree represented by S has. The i-th node of S can be associated with the i-th opening bracket in S for $1 \leq i \leq n$.

In line 4 we iterate over all nodes i and look for directed edges from i to some other node x. Note that in the string representation such a node x can only occur after the opening bracket of the i-th node. Therefore we calculate the position *pos* of the opening bracket associated with the i-th node in line 5 to 6. In line 7 we check whether the i-th node is a leave in which case we can continue with the next node. In line 9 – 14 we iterate over the part of S which comes after the opening bracket of the i-th node and keep a counter *opened* which keeps track of how many brackets have been opened but not yet closed. If this counter is 0 and we encounter an opening bracket the edge from i to the node associated with the opening bracket at position j is printed in line 11.

It should be remarked that the labels of the nodes of S are ordered by a depth-first traversal. To show that the pointer list of a tree T can be converted to its string representation in L we employ a depth-first traversal.

In order to navigate through T the following functions are needed:

– PARENT(v) returns the parent of v and $\varnothing$ if v is the root node. To implement this iterate through all vertices of T and return the unique node u which has a directed edge to v

– FIRSTCHILD($v, <$) returns the first child of v according to the order specified by $<$ and $\varnothing$ if v has no children. To implement this find the smallest node u which has v as its parent

– NEXTSIBLING($v, <$) returns the next sibling of v w.r.t. $<$ and $\varnothing$ if v was the last sibling. To implement this search for the smallest node u which is larger than v and a child of v's parent

Algorithm 2.2 Convert tree from pointer to string representation

```
1:  function pointerToString(T, <)
2:      cur ← root(T) ; state ← "down"
3:      while cur ≠ ∅ do
4:          if cur has no children then print "⟨⟩"
5:          else if state ∈ {"down","over"} then print "⟨"
6:          else if state = "up" then print "⟩"
7:          if state = up or cur has no children then
8:              if NextSibling(cur, <) ≠ ∅ then
9:                  cur ← NextSibling(cur, <) ; state ← "over"
10:             else
11:                 cur ← Parent(cur) ; state ← "up"
12:         else
13:             if FirstChild(cur, <) ≠ ∅ then
14:                 cur ← FirstChild(cur, <) ; state ← "down"
15:             else if NextSibling(cur, <) ≠ ∅ then
16:                 cur ← NextSibling(cur, <) ; state ← "over"
17:             else
18:                 cur ← Parent(cur) ; state ← "up"
```

The algorithm performs a depth-first traversal in logspace by remembering whether the last move was "*up*" in order to prevent retracing an already visited path. For $<$ we use the order induced by the labeling of T.

Table 2.1: Applying Algorithm 2.2 to the directed tree in Figure 2.1

cur	1	2	5	2	3	6	7	3	4	1	$\varnothing$
state	down	down	down	up	over	down	over	up	over	up	up
Print	$\langle$	$\langle$	$\langle\rangle$	$\rangle$	$\langle$	$\langle\rangle$	$\langle\rangle$	$\rangle$	$\langle\rangle$	$\rangle$	

2.2 Linear order on tree isomorphism classes

To canonize trees we define a linear order $<_T$ on tree isomorphism classes. This means the order relation is only defined for non-isomorphic pairs of trees. Otherwise, i.e. for two isomorphic trees S, T, neither $S <_T T$ nor $T <_T S$ holds and we will write $S =_T T$. Similarly, $S \leq_T T$ is used if either $S <_T T$ or $S =_T T$.

Definition 2.2. *Let S, T be two directed trees with root nodes s, t. The relation $S <_T T$ holds if*

(i) $|S| < |T|$, *or*

(ii) $|S| = |T|$ *and* $\#s < \#t$, *or*

(iii) $|S| = |T|$, $\#s = \#t = k$ *and* $(S_1, \ldots, S_k) <_T (T_1, \ldots, T_k)$
where $S_1, T_1, \ldots, S_k, T_k$ denote the subtrees induced by the children of s, t respectively and $<_T$ in this context is meant as lexicographic order using $<_T$ to compare two subtrees. Furthermore, the sequences of subtrees are ordered according to $\leq_T$, e.g. $S_i \leq_T S_j$ whenever $i < j$

Notice, the ordering of the sequences of subtrees is not unique whenever there are at least two subtrees which are isomorphic but the third condition is well-defined nonetheless if we can show that

Lemma 2.3. *The relation $<_T$ is a linear order on tree isomorphism classes*

Proof. This statement is equivalent to $<_T$ being transitive and $S =_T T$ iff $S \cong T$. We will prove this statement by induction over tree depth. Our inductive hypothesis for trees S, T, R of depth at most d consists of two parts and states

(1) $S =_T T \iff S \cong T$

(2) $S <_T T$ and $T <_T R \implies S <_T R$

For the base case we look at trees of depth at most 1. For this narrow class of trees the number of children of the root node is a complete invariant and thus totally characterizes all trees in this class. Therefore the third condition of Definition 2.2 never applies. It is immediate that both parts of the inductive hypothesis hold in this case.

Now, let us consider trees S, T, R of depth at most $d+1$ with $d > 0$. To show the first part of our inductive hypothesis we distinguish between two cases. The first case is that S has depth $d+1$ and T has depth at most d. As S and T have different depth they cannot be isomorphic. Thus we have to show that either $S <_T T$ or $T <_T S$. Assume the opposite $S =_T T$ and therefore $(S_1, \ldots, S_k) =_T (T_1, \ldots, T_k)$. However, the sequences of subtrees can never match as there exists an S_i in the sequence which isn't isomorphic to any T_j for $1 \leq j \leq k$ and by inductive hypothesis it follows $S_i \neq_T T_j$. This S_i is a subtree with depth d as any subtree T_j can have depth at most $d-1$.

For the second case S and T both have depth $d+1$. Here we need to show both directions of the equivalence. First, let us show that if $S \cong T$ then $S =_T T$ must hold.

This boils down to arguing why $(S_1, \ldots, S_k) =_T (T_1, \ldots, T_k)$. As S and T are isomorphic there exists an isomorphism π s.t. $S_i \cong T_{\pi(i)}$ for all $1 \leq i \leq k$. By inductive hypothesis it must hold that $S_i =_T T_{\pi(i)}$ and that $(T_{\pi(1)}, \ldots, T_{\pi(k)})$ is a sequence ordered according to $\leq_T$. Therefore we can conclude that $S =_T T$. For the other direction $S =_T T$ and $(S_1, \ldots, S_k) =_T (T_1, \ldots, T_k)$. Again, by inductive hypothesis $S_i \cong T_i$ for all $1 \leq i \leq k$. Let s_i, t_i be the root nodes which correspond to S_i, T_i. With that we can construct a partial isomorphism between S and T by mapping s_i to t_i. Successively applying this method enables us to construct a complete isomorphism between S and T.

Finally, we show that transitivity holds in the inductive step. Let $S <_T T$ and $T <_T R$. Whenever the first or second condition of Definition 2.2 holds for any of the two comparisons it is clear that $S <_T R$ must hold as well. For the other case this means that S, T, R have the same number of nodes and their roots have the same number k of children, i.e.

$$(S_1, \ldots, S_k) <_T (T_1, \ldots, T_k) \text{ and } (T_1, \ldots, T_k) <_T (R_1, \ldots, R_k)$$

There must be minimal indices $1 \leq i, j \leq k$ s.t. $S_i <_T T_i$ and $T_j <_T R_j$. Notice that for any $l < i$ it must hold that $S_l =_T T_l$ and the same applies to j and T, R. It remains to verify for the three cases $i < j$, $i = j$,$i > j$ that transitivity holds. □

The next crucial step is to show that given two directed trees S, T we can compute this order relation in logspace. Given a node v of a tree T we need to count $\#v$, the number of children of v, and the number of nodes in the subtree T_v induced by v in L. To compute $\#v$ we iterate through all children of v by calling FirstChild(v) and then repeatedly calling NextSibling. To count $|T_v|$ iterate through all nodes of T and check whether the current node is a descendant of v, i.e. if repeatedly calling Parent returns v. Also add 1 to the counter for v itself.

The algorithm to compute $<_T$ requires us to partition the children of a given node v in T into so called equicardinality blocks.

Definition 2.4. *For a directed tree T and a node v of T we call a non-empty subset $B = \{v_1, \ldots, v_l\}$ of the children of v an equicardinality block if the size of the induced subtree of every element in B has the same size k, i.e.*

$$|T_{v_1}| = \cdots = |T_{v_l}| = k$$

and B is inclusion-maximal. We say B has block size k and cardinality l. As k uniquely determines B we also write $B(k)$ to refer to B

We need to be able to determine the block size and cardinality of every equicardinality block for a given node. Furthermore, we need to iterate through all equicardinality blocks in ascending order of their block size and through all elements of a block in arbitrary order. For example, let T be a tree with a node t which has children $t_1, \ldots, t_6$ with

$$(|T_{t_1}|, \ldots, |T_{t_6}|) = (2, 5, 2, 4, 4, 2)$$

then t has equicardinality blocks $B(2), B(4), B(5)$ with respective cardinalities $3, 2, 1$. For a given tree T the following functions enable us to do all these operations on the equicardinality blocks:

– MINSUBTREESIZE$(v, k') := \min\{|T_{v_i}| \mid |T_{v_i}| > k'$ and v_i is a child of $v\}$ and $\varnothing$ if such a minimum does not exist. It returns the smallest block size k larger than k' w.r.t. node v.

– KSIZEDSUBTREES$(v, k) := |\{v_i$ is a child of $v \mid |T_{v_i}| = k\}|$. Returns the cardinality of $B(k)$

– KBLOCKMEMBER(v, k, u) returns the smallest child which comes after u in $B(k)$ w.r.t. the order induced by the labeling and $\varnothing$ if there is no such successor. If $u = \varnothing$ then the first child is returned.

These three functions can be easily implemented in logspace by iterating over the sets specified in their definition using the previously defined functions.

The main function to compute our linear oder for two trees S, T and $s \in V(S), t \in V(T)$ is defined as

$$\text{COMPARE}(s,t) := \begin{cases} -1 & , \text{ if } S_s <_T T_t \\ 0 & , \text{ if } S_s =_T T_t \\ 1 & , \text{ if } T_t <_T S_s \end{cases}$$

Let us consider the implementation of COMPARE in Algorithm 2.3. In line 2 we check whether the induced subtrees S_s and T_t are both the single node tree in which case they are isomorphic. In line 3 the first two conditions of Definition 2.2 are verified. If the sizes don't match the according result is returned, i.e. if $|S_s| < |T_t|$ or $\#s < \#t$ return -1 and vice versa. In line 5 we iterate over all block sizes k in ascending order. In line 8 we check that the block size and cardinality of the current blocks under consideration match. If this isn't the case then we know that S_s, T_t can't be isomorphic. If $k < k_t$ then return -1 as the (next) smallest subtree of the children of s is smaller than that for t. The case $l \neq l_t$ is special as it needs further recursive calls to decide the result which will be elaborated later.

If the current blocks match, i.e. we pass line 8, then there are two cases. For $l = 1$ this means both blocks consists of only one child s', t'. If their subtrees don't match then the same result applies to S_s and T_t. Otherwise, we continue with the next block. If $l > 1$ then we have l children $s_1, t_1, \ldots, s_l, t_l$. One might think that we need to order both sequences of children and then compare them pairwise to check whether the third condition of Definition 2.2 is satisfied. However, there exists a smarter way to accomplish this. For a child s_i let $(s_{i,gt}, s_{i,eq})$ be its order profile. Then $s_{i,gt}$ is defined as the number of children t_j for which it holds that $t_j <_T s_i$ with $1 \leq j \leq l$. Analogously, $s_{i,eq}$ is the number of children for which $t_j =_T s_i$ holds and similarly the order profile for a child t_i is defined. Then we search for a child s_i with $s_{i,gt} = 0$. If such a child doesn't exist we know that there must be a child t_j such that its induced subtree is smaller than ($<_T$) any induced subtree of the children of s and therefore $T_t <_T S_s$. If such an s_i exists then let us call it s_{min}. In the same fashion we look for a t_{min}. If it exists as well then by minimality it must hold that $S_{s_{min}}$ and $T_{t_{min}}$ are isomorphic and thus $S_{s_{min}} =_T T_{t_{min}}$. Next, we need to check whether $s_{min,eq} = t_{min,eq}$. If this isn't the case, e.g. $s_{min,eq} < t_{min,eq}$ then $T_t <_T S_s$. If they happen to be equal we look for the

Algorithm 2.3 Given trees S,T compute $<_t$-relation for the subtrees induced by s,t

```
1: function COMPARE(s,t)
2:     if #s = #t = 0 then return 0
3:     if |S_s| ≠ |T_t| or #s ≠ #t then return ...
4:     k, k_t ← 0
5:     while k ≠ ∅ do
6:         k ← MINSUBTREESIZE(s, k) ; k_t ← MINSUBTREESIZE(t, k_t)
7:         l ← KSIZEDSUBTREES(s, k) ; l_t ← KSIZEDSUBTREES(t, k_t)
8:         if k ≠ k_t or l ≠ l_t then return ...
9:         if l = 1 then
10:            s' ← KBLOCKMEMBER(s, k, ∅)
11:            t' ← KBLOCKMEMBER(t, k, ∅)
12:            if COMPARE(s', t') ≠ 0 then return COMPARE(s', t')
13:        else if l > 1 then
14:            h ← 0
15:            repeat
16:                s_min, t_min ← ∅
17:                for (p, q) ∈ {(s, t), (t, s)} do
18:                    p_i ← ∅
19:                    while (p_i ← KBLOCKMEMBER(p, k, p_i)) ≠ ∅ do
20:                        p_eq, p_gt ← 0
21:                        q_i ← ∅
22:                        while (q_i ← KBLOCKMEMBER(q, k, q_i)) ≠ ∅ do
23:                            res ← COMPARE(p_i, q_i)
24:                            if res = 1 then p_gt ← p_gt + 1
25:                            else if res = 0 then q_eq ← q_eq + 1
26:                        if p_gt = h then p_min ← s_i ; break
27:                    if p_min = ∅ then return ...
28:                if s_eq ≠ t_eq then return ...
29:                h ← h + s_eq
30:            until h = l
31:    return 0
```

next minimal pair of children. For that purpose we keep a counter h which starts with 0 and every iteration is incremented with $s_{min,eq}$ (or $t_{min,eq}$). If $h = l$ we know that the children of s can be bijectively mapped to the children of t such that their induced subtrees are isomorph. In line 17 we use the tuple (p, q) in the sense of a preprocessor macro since the procedure to find s_{min} is the same as the one to find t_{min}. For instance, if $(p, q) = (s, t)$ then p_i means s_i and q_i means t_i.

For the special case $l \neq l_t$ mentioned above we basically apply the same method as for $l = l_t$. We compute the order profiles and compare them. W.l.o.g. assume that $l < l_t$. We need to figure out if the children $s_1, \ldots, s_l$ can be injectively mapped to $t_1, \ldots, t_{l_t}$ such that their induced subtrees are isomorph. If that is the case$^{(*)}$ then we know that $T_t <_T S_s$ since we are forced to compare two children t_i, s_j with $k = |T_{t_i}| < |S_{s_j}|$ where k refers to the block size last assigned in line 6. This is not necessarily the case. Therefore we need to verify whether the above mentioned mapping exists. To do this we apply the same routine as in line 15 – 30 with an exception in the case that our current s_{min} is the last s_{min}, i.e. $s_{min,eq} + h = l$. If no matching t_{min} exists we return -1 ($S_s <_T T_t$) as usual. Otherwise we return 1, which is clear if $t_{min,eq} < s_{min,eq}$. If $t_{min,eq} \geq s_{min,eq}$ then the above argumentation$^{(*)}$ applies.

Theorem 2.5. *Given two directed trees S, T with two nodes $s \in V(S), t \in V(T)$ calling the recursive procedure* COMPARE(s, t) *described in Algorithm 2.3 works in* L

Proof. To prove that this algorithm works in L w.r.t. S, T as input let us first explain how a usual implementation of this recursion would work and where it fails. What happens if a call COMPARE(s, t) induces a recursive call COMPARE(s', t')? Let us name the first call the parent call and the second one the child call. Then all variables which have been assigned in the parent call along with a pointer to the line at which the child call has been made are pushed onto a global stack $\mathcal{S}$. In this case the pointer would either point to line 12 or 23. After that the child call is executed, returns its result and the environment of the parent call is restored by popping the necessary information from $\mathcal{S}$. So, for each recursive call some information needs to be pushed onto $\mathcal{S}$. As the recursive depth can be linear w.r.t. the tree size it is prohibitive to store information for every recursive call. Therefore we need a more subtle way to restore the environment of the parent call. Let us begin with the simple observation that the parents of s' and t' are s and t hence we can recompute s and t without storing anything. Next, let us consider the lines 12 and 23 at which the recursive calls are made. We know that $k = k_t$ and $l = l_t$ in the parent call as we must have passed line 8 before the child call. Since $k = |S_{s'}|$ and $l =$ KSIZEDSUBTREES(s, k) we can recompute this information without additional storage as well. By knowing l we can deduce whether the child call was made in line 12 or 23. If $l = 1$ we can check the condition in line 12 and continue with the parent call. Notice, how no information at all needs to be pushed onto $\mathcal{S}$. If $l > 1$ then we need to push $s_{min}, s_{eq}, s_{gt}, s_i, t_{min}, t_{eq}, t_{gt}, t_i, h$ and (p, q) onto $\mathcal{S}$ before the child call. We only need 1 bit to restore (p, q) and $\log l$ bits for each of the other variables as they range between 0 and l.

Therefore if storing $\mathcal{O}(\log l)$ bits on $\mathcal{S}$ for each parent call doesn't exceed the logarithmic space bound w.r.t. $n = |S| = |T|$ this algorithm works in L. The worst case is if we

travel from the root nodes all the way down to some leaves as we execute the recursion since the most information needs to be stored in $\mathcal{S}$. Let $(l_1, \ldots, l_r)$ be the sequence of l values which we encounter for each recursive call. The first call COMPARE(s, t) deals with the whole trees S and T and therefore with n nodes. For the second call we consider subtrees of size at most n/l_1. For example, assume $l_1 = 2$ then the root nodes of S and T have two children whose induced subtrees have the same size. It follows that these subtrees can be of size $n/2$ at most. For the third call only subtrees of size at most $n/(l_1 l_2)$ are considered and so forth. For the last call we reach the leaves and thus consider single node trees:

$$\frac{n}{l_1 l_2 \cdots l_r} = 1 \iff n = l_1 l_2 \cdots l_r$$

For each call i at most $10 \log l_i$ bits need to be stored on the stack (the 10 variables in the case that $l_i > 1$). Therefore we need to show that

$$10 \sum_{i=1}^{r} \log l_i \leq c \log n$$

This follows from applying the logarithm to the previous equation

$$\log n = \log(l_1 l_2 \cdots l_r) = \sum_{i=1}^{r} \log l_i$$

□

It is worth noting that to perform this recursion in logspace as described above one has to adequately replace the recursive calls in Algorithm 2.3 with multiple Gotos. Unfortunately, the readability of the resulting code decreases dramatically.

It remains to verify the correctness of Algorithm 2.3 which can be done by induction over tree depth.

Now, to compute a canonical form of a tree we extend the order $<_T$ to nodes.

Definition 2.6. *Given a tree T and two nodes $u, v \in V(T)$ we define a total linear order $u <_t v$ which holds if*

1. *$T_u <_T T_v$, or*
2. *$T_u =_T T_v$ and $u < v$*

Corollary 2.7. *A canonical form of directed trees can be computed in* L

Proof. Given a directed tree T compute its string representation T' using Algorithm 2.2 and $<_t$ as order. It follows that T' is a canonical form in string representation (if we need to distinguish two nodes u, v using their labels we know that their induced subtrees are isomorphic and therefore their order is irrelevant). Using Algorithm 2.1 we can convert T' back into its pointer list representation. □

Corollary 2.8. *A canonical labeling for directed trees can be computed in* L

Proof. Recall that a canonical labeling is an isomorphism between a given tree T and its canonical form T'. Concurrently perform a depth-first traversal on T and T' as employed in Algorithm 2.2 and use $<_t$ as order. As $T \cong T'$ the nodes which are visited at the same time can be mapped to each other to obtain the wanted isomorphism. □

2.3 Undirected trees, forests, colored trees

The result of canonizing directed trees in logspace can be extended to undirected trees, colored trees, forests and combinations thereof using some simple ideas.

To show that an undirected tree can be canonized in L we need to introduce the concept of rooting a tree. Given an undirected tree T and a vertex t of T we say that we root T at t if we orient an edge $\{u, v\} \in T$ as (u, v) iff u is closer to t than v, i.e. $d_T(u, t) < d_T(v, t)$. It is clear that the resulting tree is a directed version of T with root node t.

Lemma 2.9. *Given an undirected tree T and a vertex v of T the tree T can be rooted at v in* L

Proof. It suffices to show that the distance between any two nodes s, t in an undirected tree T can be computed in L.

Algorithm 2.4 Compute distance between two nodes s, t in an undirected tree T

```
1: function DISTANCE(T,s,t)
2:     cur ← s ; d ← 0
3:     while cur ≠ t do
4:         for each neighbor v of cur do
5:             if there is a path from v to t in T − {cur, v} or v = t then
6:                 cur ← v ; d ← d + 1
7:                 break
8:     return d
```

The correctness of this algorithm is due to the fact that there is exactly one path between any two nodes s, t of a tree. It follows that there can be only one neighbor v of s which is still connected to t when removing the edge $\{s, v\}$ in T. □

We remark that there is an alternative way to root undirected trees which does not rely on the complex result of [Rei05]. This method can be found in [ADKK12, Fact 4.2].

Corollary 2.10. *Given an undirected tree T it can be canonized in* L

Proof. Any node of T can be considered to be the root. Therefore find the minimal vertex v of T w.r.t. $<_t$, root T at v and proceed to apply the canonization algorithm for directed trees. □

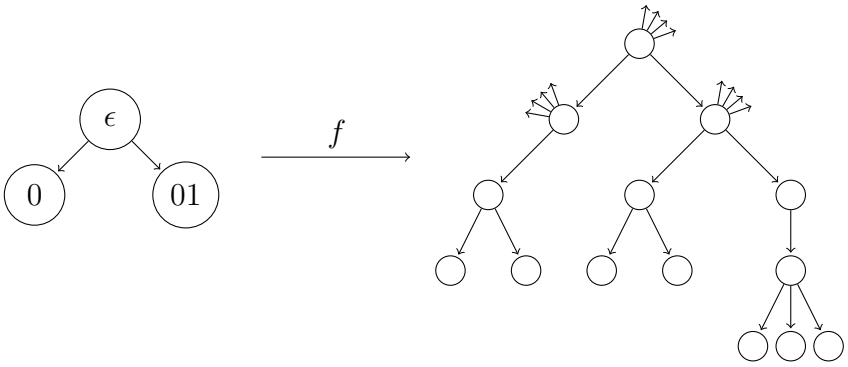

Figure 2.2: Mapping f described in the proof of Corollary 2.12

Corollary 2.11. *Given a directed forest T it can be canonized in* L

Proof. Iterate over all root nodes of T in ascending order w.r.t. $<_t$ and canonize them. To do this remember the last minimal root node and find the next larger one. □

Corollary 2.12. *Given a colored directed tree (T, c) with $c : V(T) \to \{0,1\}^*$ it can be canonized in* L

Proof. We provide an injective mapping f from colored directed trees to directed trees with the property that two colored directed trees $(T, c), (S, d)$ are isomoprhic iff $f(T, c)$ and $f(S, d)$ are isomorphic. Then to canonize a colored tree (T, c) we compute $T' = f(T, c)$, apply the tree canonization algorithm to obtain the canonical form T'' of T' and finally return $f^{-1}(T'')$. It is not hard to verfiy that this yields a canonical form for colored directed trees as we will argue later on.

We constructively define $f(T, c) = T'$. To obtain T' take T and for each node u of T add 4 new nodes which are only connected to u. Let $c(u) = c_1 c_2 \dots c_k$ be the color associated with u. For every $1 \leq i \leq k$ attach a new path of the same depth to u, i.e. T' has a path $(u, x_1, \dots, x_i)$. Attach two new vertices to x_i if $c_i = 0$ otherwise if $c_i = 1$ attach three new vertices to x_i. If $c(u)$ is the empty word ϵ no paths are attached to u, see Figure 2.2.

It is easy to recognize that f is an invariant since the labeling is ignored. To see that f is a complete invariant we show how to compute $f^{-1}(T')$. Additionally, it holds that whenever $f(T, c) \cong f(S, d)$ via π the same isomorphism π proves that (T, c) and (S, d) are isomorphic, of course ignoring the newly added vertices for the uncolored trees.

To compute $f^{-1}(T')$ notice that the maximal out-degree of the newly added vertices in T' is 3. As we have attached four new nodes to every old node we can easily distinguish between these two kinds. To reconstruct the color associated with an old node u we decode the paths in an obvious way.

As f and f^{-1} can be computed in L this concludes our proof. □

A simpler way of showing that colored directed trees can be canonized in logspace is to incorporate the coloring as additional condition in Definition 2.2, i.e. check whether

the color of the root node of S is lexicographically smaller than that of the root node of T before the first condition. From that it also conveniently follows that a canonical labeling for colored directed trees can be computed in L.

So, why have we bothered showing the previous reduction? The reason is that the general proof concept employed in this example is essentially the same one used in the following two chapters and can be easily explained using this example. Let us outline the argument in a general setting where $\mathcal{C}$ is the class of graphs we want to canonize and $\mathcal{T}$ is a class of graphs which we can already canonize via a function $canon(\cdot)$. We need a complete invariant $f : \mathcal{C} \to \mathcal{T}$, i.e. for all $G, H \in \mathcal{C}$ it holds that $G \cong H$ iff $f(G) \cong f(H)$ and the image of f is a graph of the class $\mathcal{T}$. Let $class(\cdot)$ denote the set of isomorphism classes of a graph class. Notice how f induces an injective mapping $\gamma : class(\mathcal{C}) \to class(\mathcal{T})$. In our example this means that every isomorphism class of colored directed trees is uniquely associated with an isomorphism class of directed trees but not all isomorphism classes of directed trees have a preimage in γ, for example consider the trees where every node has degree at most 3.

Furthermore, we need a function $g : \mathcal{T} \to \mathcal{C}$ which given an isomorphic copy $\hat{T}$ of $T_G = f(G)$ returns a graph $g(\hat{T})$ isomorphic to G. In terms of Figure 2.3 this means that g must map its input to the same isomorphism class that the preimages of T_G and T_H had. In the context of our example $g = f^{-1}$ and therefore this requirement immediately follows. The canonical form of $G \in \mathcal{C}$ is given by

$$g(canon(f(G)))$$

2.4 Hardness results

To show that the isomorphism and automorphism problem for directed and undirected trees is logspace-hard we introduce the following L-complete problem

Problem ORD

Input	Directed path P, vertices $s, t \in V(P)$
Question	Is there a path from s to t in P?

A directed path can be seen as directed tree with exactly one leave. Likewise, an undirected path can be seen as an undirected tree with at most two leaves. Obviously,

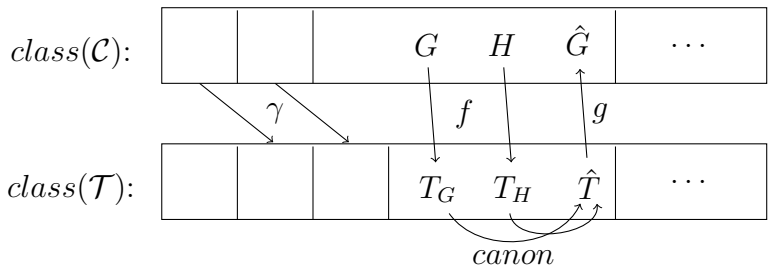

Figure 2.3: Canonization reduction argument

ORD can be decided in L. The hardness for this problem was shown in [Ete97] via quantifier-free projections which are an even weaker type of reductions than constant-depth reductions.

We show that the following problem is L-complete as well using ORD.

Problem PATHCENTER

Input	Undirected path P, vertex $v \in V(P)$
Question	Is $\lvert V(P)\rvert$ odd and v the unique center of P?

The logspace upper bound for this problem is clear since the eccentricity of vertices in paths can be easily computed in logspace.

Lemma 2.13. PATHCENTER *is* L*-complete*

Proof. For the hardness we use the reduction given in [ADKK12]. Let the mapping $(P, s, t) \mapsto (P', n)$ be the reduction from ORD to PATHCENTER where n denotes the last vertex of the directed path P with no outgoing edges

$$
\begin{aligned}
V(P') &= V(P) \cup \{i' \mid i \in V(P)\} \cup \{s''\} \\
E(P') &= E_1 \cup E_2 \cup E_3 \cup E_4 \\
E_1 &= \{\{i, j\} \mid (i, j) \in E(P) \text{ and } j \neq t\} \\
E_2 &= \{\{i', j'\} \mid (i, j) \in E(P) \text{ and } j \notin \{s, t\}\} \\
E_3 &= \{\{i', s''\} \mid (i, s) \in E(P)\} \cup \{\{s'', s'\}\} \\
E_4 &= \{\{i, t'\} \mid (i, t) \in E(P)\} \cup \{\{i, t'\} \mid (i, t) \in E(P)\} \cup \{\{n, n'\}\}
\end{aligned}
$$

We basically take the path P and copy it twice and undirect its edges (E_1, E_2). Additionally, we insert a new vertex s'' in the second copy of P in-between s' and its predecessor (E_3). Let x and x' be the vertices that come right before t and t'. Then we swap the edges, i.e. instead of $\{x, t\}$ and $\{x', t'\}$ we choose the edges $\{x, t'\}$ and $\{x', t\}$. Lastly, we connect n and n'.

To prove the correctness we show that if s comes before t, i.e. we have an instance of ORD, then n is the center of P' and if s doesn't come before t this isn't the case. Because $|V(P')| = 2|V(P)| + 1$ the undirected path P' always has an odd number of

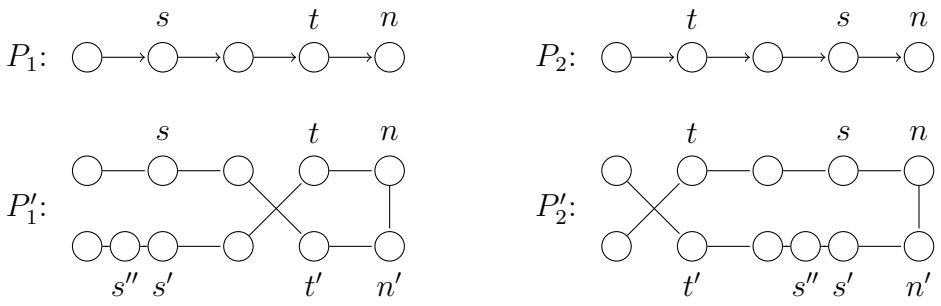

Figure 2.4: Reduction from ORD to PATHCENTER

vertices. In an undirected path a node v is the center node iff the distance from both ends to v is equal. Let d_t and $d_{t'}$ be the distances of the path between n and the end node which goes over t and t' respectively. Assume s comes before t in P and a is the first node in P. Then these distances are given by

$$d_t = d_P(n,t) + d_P(t,s) + d_P(s,a) + 1$$
$$d_{t'} = 1 + d_P(n,t) + d_P(t,s) + d_P(s,a)$$

Obviously, $d_t = d_{t'}$ in this case. In d_t the plus one comes from the additional edge caused by s'' whereas in $d_{t'}$ the plus one comes from the edge from n to n'. Now, assume that t comes before s. Then we have

$$d_t = d_P(a,n)$$
$$d_{t'} = 2 + d_P(a,n)$$

For $d_{t'}$ the edge between n and n' as well as the edge from s' to s'' occurs thus the plus two. □

Finally, we can prove that the automorphism and isomorphism problem for directed and undirected trees is L-complete. It should be mentioned that to solve the tree automorphism problem for a tree T in logspace it suffices to iterate over all pairs of nodes $u \neq v$ of T and check if T_u and T_v are isomorph.

Theorem 2.14. *The graph isomorphism and automorphism problem for directed and undirected trees is* L*-complete*

Proof. We show that PathCenter can be reduced to all these problems. First, let us consider the case of automorphism for undirected trees. The reduction is given by the mapping $(P,c) \mapsto (T,c)$

$$V(T) = V(P) \cup \{c'\} \ , \ E(T) = E(P) \cup \{\{c,c'\}\}$$

To understand the correctness the key fact is that every isomorphism and thus every automorphism must respect all invariants. In our case this means that c must be mapped to itself in every automorphism as it is the only vertex with degree three. As the distance between two vertices is an invariant as well and the ends of P can be only mapped to their kind as they have degree one (with an exception being that c is one of the end nodes) it follows that one end can be mapped to the other in an automorphism only if they have the same distance to c. This means that c is the center node of P iff T has a non-trivial automorphism.

For the isomorphism of undirected trees the following mapping $(P,c) \mapsto (T_1,T_2)$ works. Let e_1, e_2 be the end nodes of P.

$$V(T_1) = V(P) \cup \{c', e_1', e_1''\} \ , \ E(T_1) = E(P) \cup \{\{c,c'\}, \{e_1,e_1'\}, \{e_1,e_1''\}\}$$
$$V(T_2) = V(P) \cup \{c', e_2', e_2''\} \ , \ E(T_1) = E(P) \cup \{\{c,c'\}, \{e_2,e_2'\}, \{e_2,e_2''\}\}$$

In essence, for the correctness of this reduction the same arguments as before apply.

For the case of directed trees simply choose c to be the root node in the reductions. □

Note, that the set of graphs constructed in the reductions of Theorem 2.14 can be exactly characterized as the set of undirected trees which become a path when all leaves are removed. This class of graphs is also known as caterpillars. It follows that every graph class which is a superset of caterpillars must be logspace-hard due to the same reductions. The graph classes (CA, HCA, interval) considered in the last chapter fulfill this property and thus are logspace-hard.

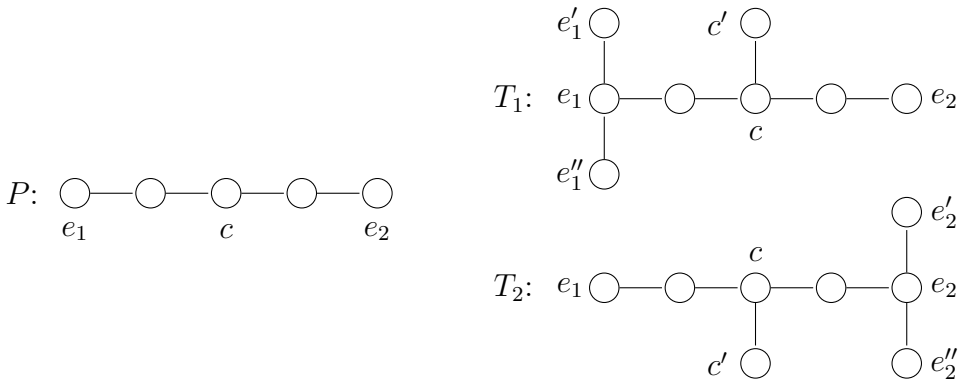

Figure 2.5: Reduction from PathCenter to isomorphism of undirected trees

3 k-Trees

In an informal sense the tree width is a measure of how much a graph resembles a tree. k-Trees can be described as the set of maximal graphs with tree width k such that adding another edge leads to a tree width of $k+1$. In this chapter we show how to canonically label k-trees in L and why deciding isomorphism for this class is logspace-hard following [ADKK12].

Definition 3.1 ([Die12, cf. p. 337])**.** *Given a graph G and a tree $T = (\mathcal{V}, E)$ such that $\mathcal{V}$ is a family of vertex sets, i.e. $V_t \subseteq V(G)$ for all $V_t \in \mathcal{V}$. The tuple $(T, \mathcal{V})$ is called a tree decomposition of G if the following holds*

1. *$V(G) = \bigcup_{V_t \in \mathcal{V}} V_t$*
2. *for all $\{u, v\} \in E(G)$ there exists a $V_t \in \mathcal{V}$ s.t. $\{u, v\} \subseteq V_t$*
3. *for all $V_1, V_2, V_3 \in \mathcal{V}$ it holds that whenever V_2 is a node on the unique path from V_1 to V_3 in T then $V_1 \cap V_3 \subseteq V_2$*

A vertex set $V_t \in \mathcal{V}$ is called a bag. The width of $(T, \mathcal{V})$ is defined as the size of the largest bag minus one, i.e.

$$\max_{V_t \in \mathcal{V}} \{|V_t|\} - 1$$

The third condition can be reformulated as the requirement that whenever two bags V_t, V_s have non-empty intersection then for all nodes V_r on the unique path between V_t and V_s in T it must hold that V_r contains all nodes of this intersection $V_t \cap V_s$.

The tree width of a graph G is defined as the minimum width among all possible tree decompositions of G. For a tree S its tree width is always one (with exception of the single node tree) and the obvious tree decomposition $(T, \mathcal{V})$ has $\mathcal{V} = E(S)$ as its bags and there is an edge between two nodes V_s, V_t of T if they have non-empty intersection. The purpose of the minus one in the definition of the width of tree decompositions is to associate trees with a tree width of one instead of two as this is arguably more natural.

As already mentioned a k-tree is a graph G with tree width k such that adding any new edge to G leads to an increased tree width $k+1$. For example, trees are exactly the class of 1-trees. A forest F, which is a graph whose connected components are trees, has tree width one as well. However, connecting two disjoint trees of F doesn't increase the tree width therefore F isn't a 1-tree. In [AP89] the following inductive definition has been given for k-trees.

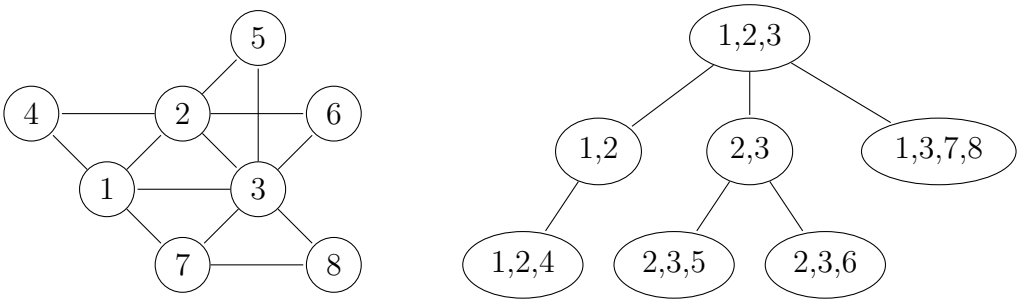

Figure 3.1: Example of a 2-tree and its tree decomposition

Definition 3.2. *A k-clique is a k-tree. Let G be a k-tree and M is a k-clique in G. If a new vertex v is added to G such that v is connected to all vertices of M then the resulting graph remains a k-tree. M is called support of v. The initial k-clique is called the base of G*

Any k-clique in G can be chosen as base from which point on the support of every newly added vertex is unambiguous.

In Figure 3.1 a 2-tree G along with a possible tree decomposition is shown. Notice that the given tree decomposition of G has width 3 and therefore isn't minimal due to the bag $\{1, 3, 7, 8\}$.

3.1 Canonically labeling k-Trees

To show how to canonically label k-trees in L we will first describe the algorithm and show its correctness in this section. In the next section a logspace implementation is described.

Definition 3.3. *For a graph G we define $T(G)$ by*

- $V(T(G)) = \left\{ M \subseteq V(G) \;\middle|\; \begin{array}{l} M \text{ is a } (k+1)\text{-clique in } G \text{ or } M \text{ is a } k\text{-clique in } G \\ \text{that is not contained in exactly one } (k+1)\text{-clique} \end{array} \right\}$
- $\{M_1, M_2\} \in E(T(G)) \iff M_1 \subsetneq M_2$

We show that $T(G)$ is a tree decomposition of G. But since $T(G)$ doesn't contain enough information such that its canonical form can be used as complete invariant the kernel K of G is computed in Algorithm 3.5. The invariant kernel K allows us to root $T(G)$ and encode additional information into the so called colored rooted tree $T(G, \hat{\pi})$. The canonized form of this tree $\hat{T}(G, \hat{\pi})$ then satisfies the requirements of a complete invariant.

Lemma 3.4. *G is a k-tree iff $T(G)$ is a tree decomposition of G*

Algorithm 3.5 Canonically label k-trees (cf. [ADK12, Algorithm 3.1])

1. Compute the graph $T(G)$ (Definition 3.3) and check that it is a tree decomposition of G. If this is the case then let us call $T(G)$ the tree representation of G. Otherwise, reject since G is no k-tree (Lemma 3.4)
2. Compute the kernel K of G, which is a k- or $(k+1)$-clique (Definition 3.6)
3. For each labeling π of the kernel K, i.e. a bijection $\pi : K \to \{1, \ldots, |K|\}$, do the following steps
   a) Compute the colored rooted version $T(G, \pi)$ of $T(G)$ (Definition 3.7)
   b) Among the labelings π choose the one which leads to the lexicographically smallest canonical form of the colored tree $T(G, \pi)$. Let us denote this labeling with $\hat{\pi}$ and let $\hat{T}(G, \hat{\pi})$ be this smallest canonical form
4. Compute a canonical labeling φ from $T(G, \hat{\pi})$ to its canonical form $\hat{T}(G, \hat{\pi})$
5. Let $\hat{G}$ be the canonical form of G which is derived from $\hat{T}(G, \hat{\pi})$ according to Theorem 3.9 . Compute an isomorphism φ' between G and $\hat{G}$ using φ, again due to Theorem 3.9. Then φ' is the wanted canonical labeling

---

Proof. First, we show that if G is a k-tree then $T(G)$ is a tree decomposition of G. Let G be a k-tree with base clique $B = \{v_1, \ldots, v_k\}$ and let $(v_{k+1}, \ldots, v_n)$ be the rest of the vertices added to G in that order. Let us write G_i for the vertex-induced subgraph $G[v_1, \ldots, v_i]$. For G_k it is clear that it is a k-tree and the single node tree $T(G_k)$ is a tree decomposition thereof. Now, for $i > k+1$ let the k-clique S_i be the support of v_i. We can inductively assume that $T(G_{i-1})$ is a tree decomposition of G_{i-1}. If the support S_i is a node in $T(G_{i-1})$ then it occurs in more than one $(k+1)$-cliques. For $T(G_i)$ we add the node $M_i = S_i \cup \{v_i\}$ and connect it with S_i. Otherwise, we need to add not only M_i as node to $T(G_i)$ but also S_i. In this case S_i must be connected to M_i and the unique $(k+1)$-clique in which it was previously contained. It is simple to check for both cases that the conditions of Definition 3.1 still hold for $T(G_{i+1})$.

For the other direction we show that if $T(G)$ is a tree decomposition of G then G is a k-tree by constructing G from $T(G)$ as described in Definition 3.2. If $T(G)$ consists of a single node of size k or $k+1$ then these vertices must form a clique in G and thus G is a k-tree. Otherwise, let M be a k-clique node in $T(G)$. We can use M as base to construct G iteratively by adding all vertices $u \in V(G) \setminus M$. Let M_u be the $(k+1)$-clique node that contains u and is closest to M. To add u we choose the first k-clique S on the path from M_u to M in $T(G)$ as support of u. Since $T(G)$ is a tree there is only one path from M_u to M. □

Observe that the number of vertices of $T(G)$ is polynomially bounded w.r.t. $n = |V(G)|$ as there can be at most $n^{k+1} + n^k$ $(k+1)$- and k-cliques in G. This is a necessary condition to be able to compute $T(G)$ in logspace.

Next, we explain how the kernel of a k-tree is defined using the unique center node of $T(G)$.

Lemma 3.5. *If G is a k-tree then the center of $T(G)$ is a single node*

Proof. Assume this is not the case. Then the center must consist of two adjacent nodes. One must be a k-clique and the other a $(k+1)$-clique because k- and $(k+1)$-cliques must alternate in $T(G)$. Additionally, a k-clique must be always connected to at least 2 other nodes in $T(G)$. Therefore no leave of $T(G)$ can be a k-clique. It follows that the eccentricity of a k-clique must be odd whereas for a $(k+1)$-clique it must be even. This contradicts that both center nodes have the same eccentricity. □

Definition 3.6. *The kernel of a k-tree G is defined as the set of vertices in the center node of $T(G)$*

Depending on the structure of G its kernel K can be either a k- or a $(k+1)$-clique. It is not hard to see that the kernel of a k-tree must be an invariant. We choose K as root node to convert $T(G)$ into a directed tree. This enables us to uniquely associate every $(k+1)$-clique M in $V(T(G)) \setminus K$ with the vertex $v \in V(G) \setminus K$ which is contained in M but not in any ancestor of M, i.e. a node on the path from K to M. Consider Figure 3.3 for an example. Given a vertex $v \in V(G) \setminus K$ we write M_v to denote its corresponding $(k+1)$-clique and $v(M)$ is defined as v such that $M = M_v$ for any $(k+1)$-clique M but K.

Before we define the colored rooted tree $T(G, \pi)$ let us consider why a canonical form of $T(G)$ fails to be a complete invariant. In Figure 3.2 we see two non-isomorphic 2-trees which have isomorphic tree representations, i.e. both have a path with 7 vertices as tree representation. The reason for this is that the labels of $T(G)$ contain necessary information to reconstruct G which is lost when one simply canonizes $T(G)$. Given a $(k+1)$-clique node in $T(G)$ one has $k+1$ possible choices to select a support for a new $(k+1)$-clique in G. This is encoded in the labels of the k-cliques of $T(G)$. Using the following method to color $T(G, \pi)$ this information can be preserved.

Besides, one could ask what is the purpose of considering every possible labeling $\pi : K \to \{1, \ldots, |K|\}$ for the kernel K of G. The reason is that the vertices in K act as an anchor in the canonization process. More specifically, given a k-tree G the vertices of its kernel can be canonically labeled via $\hat{\pi}$. The labeling $\hat{\pi}$ is then used to extend this canonical labeling to all vertices of G.

Figure 3.2: Two non-isomorphic 2-trees that have isomorphic tree representations

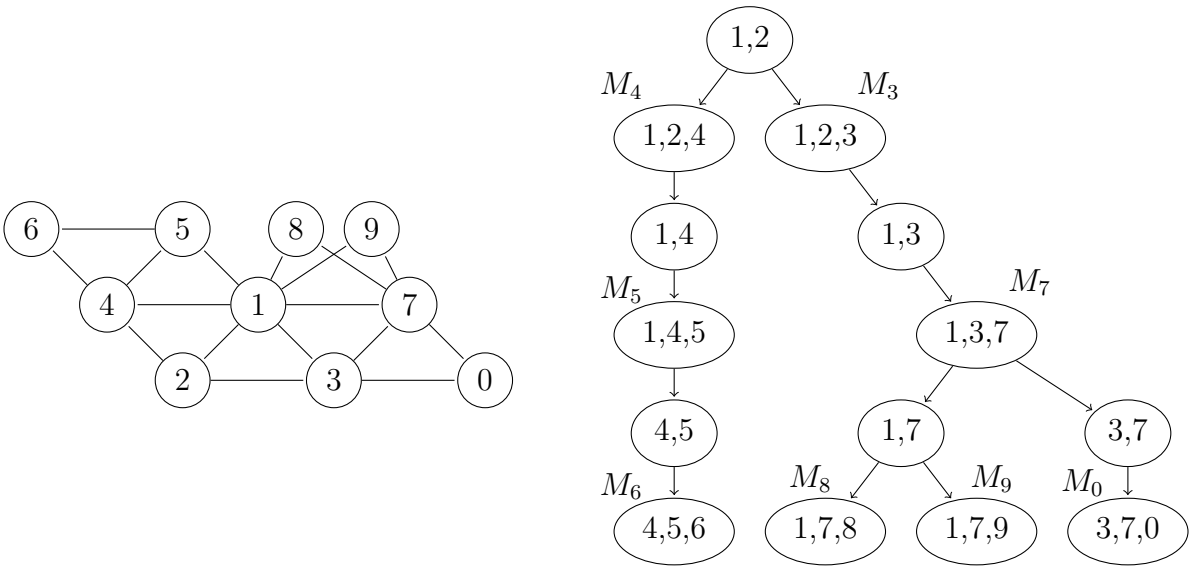

Figure 3.3: A 2-tree G and its tree representation $T(G)$ rooted at its center

Definition 3.7. *Given a k-tree G with kernel K and a bijection $\pi : K \to \{1, \ldots, |K|\}$ we define the colored rooted tree representation $T(G, \pi)$ as follows. $T(G, \pi)$ is the same tree as $T(G)$ with the exception that it is directed with K as its root node.*

Given a vertex $v \in V(G) \setminus K$ let the level of v in G be

$$l_G(v) = \lceil d_{T(G)}(K, M_v)/2 \rceil$$

The color of a node M in $T(G, \pi)$ is given by the set $c(M) = \{c(v) \mid v \in M\}$ with

$$c(v) = \begin{cases} \pi(v) & , \text{ if } v \in K \\ l_G(v) + |K| & , \text{ else} \end{cases}$$

Let us consider what the color mapping c for the vertices of the 2-tree G in Figure 3.3 looks like given a labeling $\pi : \{1, 2\} \to \{1, 2\}$ of its kernel. The mapping for $c(v)$ is then given by

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 0 \\ \pi(1) & \pi(2) & 3 & 3 & 4 & 5 & 4 & 5 & 5 & 5 \end{pmatrix}$$

We continue with proving that the colored rooted tree is a complete invariant.

Lemma 3.8. *Let G, H be two isomorphic k-trees with an isomorphism $\varphi : V(G) \to V(H)$. Then $T(G, \pi_G)$ and $T(H, \pi_H)$ are isomorphic via $\varphi'(M) = \{\varphi(v) \mid v \in M\}$ for M in $V(T(G))$ provided that π_G agrees with π_H through φ, i.e.*

$$\pi_G(u) = \pi_H(\varphi(u)) \text{ for all } u \text{ in the kernel of } G$$

Proof. It is not hard to see that an isomorphism between G and H leads to an isomorphism of $T(G)$ and $T(H)$ as well since the tree representations are invariant. It

remains to show that the colors are respected, i.e. for each M in $V(T(G, \pi_G))$ it holds that $c(M) = c(\varphi'(M))$ To show this we prove that the colors of the vertices of G and H coincide, which means $c(v) = c(\varphi(v))$. If v is a vertex in the kernel of G then

$$c(v) = \pi_G(v) = \pi_H(\varphi(v)) = c(\varphi(v))$$

holds by assumption. Let k' be the size of the kernels of G and H. If v is not part of the kernel of G then

$$c(v) = l_G(v) + k' = l_G(\varphi(v)) + k' = c(\varphi(v))$$

since the level of a vertex is an invariant and hence must be preserved by an isomorphism. □

The more demanding part of our claim is to prove that this invariant is complete. As usual, this is shown constructively.

Theorem 3.9. *Given a k-tree G, its colored rooted tree $T(G, \pi)$ and an isomorphic copy T' of $T(G, \pi)$. A k-tree G' can be constructed from only T' such that $G \cong G'$. Additionally, given an isomorphism between $T(G, \pi)$ and T an isomorphism between G and G' can be constructed*

Proof. Let K be the kernel of G and M_r the root node of T'. The size of the kernel $|K|$ is given by $k' = |c(M_r)|$. Then the number of vertices n in G' is given by k' plus the number of $(k+1)$-clique nodes in T' excluding the root node where M is called an l-clique node in T if $|c(M)| = l$. The correctness follows from the fact that there is a one-to-one correspondence between every vertex $v \in V(G) \setminus K$ and the $(k+1)$-clique nodes via the mapping M_v. Let $\{v_1, \ldots, v_n\}$ be the vertex set of G' and let $\{v_1, \ldots, v_{k'}\}$ be a clique in G'. Furthermore, let $(M_{k'+1}, \ldots, M_n)$ be the ordered set of $(k+1)$-cliques in T' excluding the root node using the ordering induced by the labeling of T'. Then we can associate the $(k+1)$-clique node M_i with the vertex v_i in G for all $k'+1 \leq i \leq n$. Now, for every vertex v_i with $k'+1 \leq i \leq n$ we add the following edges. For each color c_i in $c(M_i)$ with $c_i < k'$ add an edge between v_{c_i} and v_i. Since every vertex in the kernel can be distinguished from all other vertices by its color (consider the offset $k' = |K|$ in Definition 3.7 of $c(v)$) it follows that all edges between a vertex in the kernel and one outside of the kernel can be reconstructed by this part.

For each color $c_i \in c(M_i)$ with $k' < c_i < c_{max} = \max\{c_j \mid c_j \in c(M_i)\}$ let M_j be the $(c_i - k')$-th node on the path from the root node to M_i counting only $(k+1)$-clique nodes. Notice how for c_{max} this $(c_i - k')$-th node is M_i itself. It can be shown that the color of a vertex $v \in V(G) \setminus K$ along with one of the descendants of M_v unambiguously references M_v itself.

A witnessing isomorphism $\varphi : G \to G'$ can be derived from an isomorphism $\phi : T(G, \pi) \to T'$ as

$$\varphi(v) = \begin{cases} \pi(v) & , \text{ if } v \in K \\ v(\phi(M_v)) & , \text{ else} \end{cases}$$

By an inductive argument over the level of a vertex v it can be shown that this is indeed a valid isomorphism.

□

This directly implies that our invariant is complete.

Theorem 3.10. *Algorithm 3.5 works correctly, i.e. given two isomorphic k-trees G, H they receive canonical labelings ψ_G, ψ_H such that $\psi_G(G) = \psi_H(H)$*

Proof. Let $\hat{\pi}_G, \hat{\pi}_H$ be the two labelings of the kernels of G and H respectively which give rise to the lexicographically smallest canonical rooted trees. The colored rooted tree representations $T(G, \hat{\pi}_G)$ and $T(H, \hat{\pi}_H)$ are isomorphic due to Lemma 3.8. It follows that they have identical canonical forms $\hat{T} = \hat{T}(G, \hat{\pi}_G) = \hat{T}(H, \hat{\pi}_H)$. Let φ_G, φ_H be the canonical labelings for $T(G, \hat{\pi}_G)$ and $T(H, \hat{\pi}_H)$ to their canonized colored rooted tree representation $\hat{T}$. By Theorem 3.9 we can compute a G' from $\hat{T}$ which is isomorphic to both G and H. The canonical labelings ψ_G, ψ_H are constructed with the help of φ_G and φ_H. It follows that

$$\psi_G(G) = G' = \psi_H(H)$$

□

3.2 Logspace implementation

It remains to show that each step of the presented algorithm can be performed in logspace. First, we show that this is the case for computing the tree representation $T(G)$.

Lemma 3.11. *Given a graph G the tree representation $T(G)$ can be computed in logspace. Additionally, it can be checked that $T(G)$ is a tree decomposition of G in* L

Proof. Consider Algorithm 3.6. For brevity the part where the number of vertices of $T(G)$ is counted has been omitted but can be easily added. The function isKNode counts in how many $(k+1)$-cliques the given k-clique occurs and concludes if the k vertices form a k-clique node in $T(G)$. Since we know that every edge in $T(G)$ is given by a $(k+1)$-clique node and a k-clique node we can iterate over all $(k+1)$-clique nodes and check which k-clique nodes in $T(G)$ are a proper subset thereof. The condition that $v_i < v_{i+1}$ in the loop ensures that the same edge isn't printed multiple times.

To verify that $T(G)$ is a tree decomposition we have to check that $T(G)$ constitutes a tree and the conditions of Definition 3.1 hold, which is not difficult. To see that $T(G)$ is a tree consider the characterization that an undirected tree is a graph for which there is exactly one path between any two vertices. This condition can be restated as follows. Given an undirected graph G for any two vertices $s \neq t \in V(G)$ it must hold that t is either a neighbor of s or there exists exactly one neighbor s' of s such that there is a path from s' to t in the graph G without the edge $\{s, s'\}$. Clearly, this can be checked in logspace. Obviously, when G is a tree then this condition holds for the same reasons that Algorithm 2.4 – computing the distance between two nodes in an undirected tree –

is correct. For the other direction we first argue that G must be connected. If this isn't the case then there are two vertices s, t for which there is no path and thus no suitable s'. If G is connected but no tree then there must be a cycle $s - s' - \cdots - t - \cdots - s'' - s$ in G. The vertices s', s'' are both neighbors of s for which there is a path to t when removing the edge between s and s' or s'' and therefore the condition fails again. For a triangle in G notice that t must be either a neighbor of s or connected to s via a path of length at least two. □

Algorithm 3.6 Compute tree representation $T(G)$ on input G for fixed k

```
1: function T(G)
2:     for v_1 < v_2 < ··· < v_{k+1} ∈ V(G) do
3:         if {v_1, ..., v_{k+1}} is clique in G then
4:             if isKNode(G, v_1, ..., v_k) then
5:                 print "{{v_1, ..., v_k}, {v_1, ..., v_k, v_{k+1}}}"
6: function isKNode(G, v_1, ..., v_k)
7:     i ← 0
8:     for v ∈ V(G) \ {v_1, ..., v_k} do
9:         if {v, v_1, ..., v_k} is clique in G then i ← i + 1
10:    return true iff i ≠ 1
```

Lemma 3.12. *The colored rooted tree $T(G, \pi)$ for a k-tree G and a labeling π of its kernel can be computed in* L

Proof. First, observe that the center of an undirected tree can be computed in logspace as this basically only involves computing the eccentricity of each node which in turn means to compute the distance between all pairs of nodes. This is covered by Algorithm 2.4. Therefore we can also find the kernel of a k-tree. For the same reason we can also compute the colors of each vertex in the nodes of $T(G)$ and by Lemma 2.9 we can root $T(G, \pi)$ at its center node. □

It is not difficult to see that the rest of Algorithm 3.5 can be computed in L as well. For step 4 we use Corollary 2.8 and the computation of step 5 is clear.

Corollary 3.13. *k-Trees can be canonically labeled in* L *for a fixed k using Algorithm 3.5*

3.3 Hardness results

We show that the automorphism and isomorphism problem for k-paths is L-hard. A k-path is a k-tree for which there exists a tree decomposition that is a path. As k-paths are a subset of k-trees it follows that the hardness applies to k-trees as well.

Lemma 3.14. *The automorphism problem for k-trees is* L*-hard for a fixed k*

Proof. We show this by reducing from the problem PathCenter, which was introduced in the last chapter. The reduction mapping $(P, c) \mapsto P'$ where the neighbors of c are n_1 and n_2 is defined by

$$\begin{aligned} V(P') &= V(P) \cup \{c_1, c_2\} \\ E(P') &= \{\{u, v\} \mid 1 \leq d_P(u, v) \leq k \text{ for } u, v \in V(P)\} \\ &\quad \cup \{\{u, c_i\} \mid 0 \leq d_{P-n_i}(u, c) < k \text{ for } u \in V(P), i \in \{1, 2\}\} \end{aligned}$$

where $P - n_i$ refers to the vertex-induced subgraph after removing the vertex n_i. In Figure 3.4 an example is shown. The path P is copied and two new vertices c_1, c_2 are added. The first part of the edge relation forces every $(k+1)$ consecutive nodes in P to be a $(k+1)$-clique. The second part of E connects c_1 and c_2 with c and the $(k-1)$ vertices to left and to the right of c respectively.

To see that P' is a k-tree verify that $T(G)$ defines a tree decomposition of P'. To see that P' is a k-path we show how to construct a tree decomposition T of P' which is a path. Let $\mathcal{M}$ be the set of all $(k+1)$-cliques in the vertex-induced subgraph of P' after removing c_1 and c_2. Let M_{c_i} be the unique $(k+1)$-clique which contains c_i for $i \in \{1, 2\}$. Let T be a tree which has $\mathcal{M} \cup \{M_{c_1}, M_{c_2}\}$ as its nodes. For two nodes $M_a, M_b \in \mathcal{M}$ connect them if they share k common vertices, i.e. $|M_a \cap M_b| = k$. Now, let $M_a, M_b \in \mathcal{M}$ be the two connected nodes which both share k vertices with M_{c_i} and insert M_{c_i} between M_a and M_b, i.e. $M_a - M_b$ becomes $M_a - M_{c_i} - M_b$ for $i \in \{1, 2\}$. In the case that $k = 1$ it holds that for M_{c_1} and M_{c_2} the same pair of vertices $M_a, M_b \in \mathcal{M}$ is selected. In this case replace $M_a - M_b$ with $M_a - M_{c_1} - M_{c_2} - M_b$.

We assume that the distance between c to one of the both end nodes of P is larger than k. Otherwise, we can trivially check the instance as k is a constant and map it appropriately. For a path P on n vertices the only automorphism besides the trivial one is the mapping $\binom{1\ \ 2\ \ \ldots\ n-1\ \ n}{n\ n-1\ \ldots\ \ 2\ \ 1}$. If P consists of an even number of vertices this mapping has no fix point, i.e. a vertex which is mapped onto itself. If P has an odd number of vertices then this single fix point is always the center node. The node c has a distinguished degree from all other vertices in P' and therefore must be mapped onto itself for a valid automorphism. It follows that P' can only have a non-trivial automorphism iff c is the center of P.

□

Lemma 3.15. *The isomorphism problem for k-trees is* L*-hard for a fixed k*

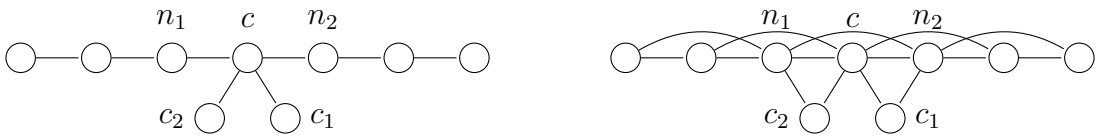

Figure 3.4: Reduction from PathCenter to automorphism of k-trees
Resulting k-paths for path on 7 vertices for $k = 1$ (left) and $k = 2$ (right)

Proof. We slightly modify the previous reduction as follows. Given a path P and a node $c \in V(P)$ let n_1, n_2 again refer to the neighbors of c and let e_1, e_2 be the both end nodes of P. The reduction mapping $(P, c) \mapsto (P_1, P_2)$ is given by

$$\begin{aligned} V(P_j) &= V(P) \cup \{c_1, c_2, e\} \\ E(P_j) &= \{\{u, v\} \mid 1 \leq d_{P-n_i}(u, v) \leq k \text{ for } u, v \in V(P)\} \\ &\quad \cup \{\{u, c_i\} \mid 0 \leq d_{P-n_i}(u, c) < k \text{ for } u \in V(P), i \in \{1, 2\}\} \\ &\quad \cup \{\{u, e\} \mid 0 \leq d_P(u, e_j) < k \text{ for } u \in V(P)\} \end{aligned}$$

for $j \in \{1, 2\}$.

This is basically the same reduction as before with an additional vertex. Given the input (P, c) compute the k-path P' as before. Then create two copies P_1, P_2 of P' with an additional vertex e. For P_i this vertex e is connected to the end of P' given by the node e_i such that a $(k+1)$-clique is formed and $i \in \{1, 2\}$. This forces any isomorphism between P_1 and P_2 to swap the end nodes of P' and thus represent a non-trivial automorphism for P'. This is similar to the construction in Theorem 2.14 shown in Figure 2.5. □

Corollary 3.16. *The isomorphism and automorphism problem for k-trees is* L*-complete for a fixed k*

4 Helly Circular-Arc graphs

A graph is called an interval graph if each of its vertices can be assigned to an interval on a line such that two vertices are adjacent if and only if their intervals have non-empty intersection. An extension of this concept is to assign the vertices to intervals on a circle instead. The corresponding graph is called a circular-arc (CA) graph. Additionally, if such a set of circular arcs abides by a certain property the graph is called Helly circular-arc (HCA) graph. The class of interval graphs is a subclass of HCA graphs.

Fields of application for these graph classes are, among others, scheduling problems and computational tasks related to DNA due to the fact that "the set of intersections of a large number of fragments of genetic material in a virus [is] an interval graph" and "genetic information [is] arranged inside a linear structure[...]" [McC03].

The recent result of [KKV13] – whom we will closely follow in this chapter – has revealed how to canonize HCA graphs in L. As byproduct this algorithm allows recognition of HCA graphs in L as well. The matching lower bound for the isomorphism problem for HCA and interval graphs follows from the remark made after Theorem 2.14.

Let us start by formally introducing HCA graphs and their related concepts.

Definition 4.1. *A set system $\mathcal{S}$ consists of a set of subsets w.r.t. some set A, i.e. for all $X \in \mathcal{S}$ it holds that $X \subseteq A$*

Definition 4.2. *A set system $\mathcal{S}$ is Helly if for every subset $S \subseteq \mathcal{S}$ it holds that $\bigcap_{X \in S} X \neq \emptyset$ whenever $X \cap Y \neq \emptyset$ for all $X, Y \in S$*

Definition 4.3. *The intersection graph $\mathbb{I}(\mathcal{S})$ of a set system $\mathcal{S}$ is the graph $(\mathcal{S}, E)$ with the edge relation being defined by $\{X, Y\} \in E$ iff $X \cap Y \neq \emptyset$*

A circular-arc (CA) system $\mathcal{A}$ is a set system of arcs on a circle. Additionally, if $\mathcal{A}$ is Helly then it is an HCA system. Likewise, an interval system $\mathcal{I}$ is a set of intervals on a line. Observe that every interval system is Helly and can be seen as a special case of a CA system which doesn't cover the whole circle.

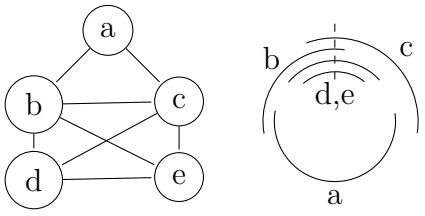

Figure 4.1: HCA graph and a non-Helly CA representation

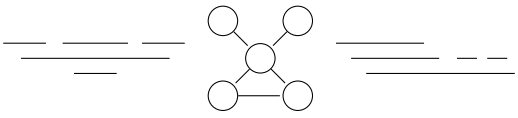

Figure 4.2: Two different interval models of the same graph

Definition 4.4. *A graph G is a CA graph (HCA graph, interval graph) if there exists a CA system (resp. HCA system, interval system) $\mathcal{S}$ such that $G \cong \mathbb{I}(\mathcal{S})$*

By the above remarks it follows that every interval graph is an HCA graph which in turn must be a CA graph. As a side note, given a CA system we will sometimes talk about it in graph theoretical terms because its intersection graph is unambiguous. For instance, a subset of arcs is naturally a clique if the respective vertices in its intersection graph form a clique.

Definition 4.5 ([KKV13, p. 3])**.** *A CA representation of a graph G is a function $\rho : V(G) \to \mathcal{A}$ which maps each vertex of G to an arc such that two arcs intersect iff their vertices are adjacent. The CA system $\rho(G) = \{\rho(v) \mid v \in V(G)\}$ is said to be a CA model of G. If $\rho(G)$ is Helly (an interval system) then ρ is an HCA representation (resp. interval representation)*

To verify whether the CA representation of a CA graph is Helly one can check if for every maxclique in the graph the corresponding arcs have overall non-empty intersection. Look at Figure 4.1 and observe how this condition fails for the maxclique $\{a, b, c\}$.

Imposing certain restrictions on the considered CA systems helps us in reducing the complexity of later arguments. One such restriction is that we don't consider arcs which are empty or cover the whole circle. The justification for that is that the corresponding vertices in the intersection graph must be independent or universal. These types of vertices can be counted and removed before applying the canonization. Therefore, each arc can be defined by its two different endpoints. So a CA system $\mathcal{A}$ with n arcs can be described by its $2n$ endpoints on the circle. For two given endpoints a, b the arc $A = [a, b]$ is obtained by going in clockwise direction starting from a until point b is reached. Hence, $[a, b]$ and $[b, a]$ cover exactly opposing parts of the circle. We also say that $\bar{A} = [b, a]$ is obtained by flipping the arc $A = [a, b]$.

It is not hard to see that restricting the list of endpoints to be all pairwise different does not leave any CA graph without a CA model thus being a legitimate limitation. This especially means that each arc in a CA system is unique.

Obviously, only the relative position of the endpoints to each other is relevant when considering the structure of the CA system. Assuming the reference point 12 o'clock on the circle we can now describe any positioning of endpoints as list by going in clockwise direction, starting from our reference point, and writing down each endpoint as it occurs. For an arc $A = [a, b]$ let us say a is the left endpoint $l(A)$ of A and b is the right endpoint $r(A)$. For Figure 4.1 this position list(sequence) is $r(b), r(e), r(d), l(a), r(c), l(b), \ldots$ assuming that e refers to the innermost arc. This leads to the following definition

Definition 4.6. *A CA system $\mathcal{A}$ with pairwise different arcs $(A_1, \ldots, A_n)$ of which non is empty or covers the whole circle is determined by a position list M, i.e. $l(A_i)$ and $r(A_i)$ occur exactly once in M for all $1 \leq i \leq n$*

Using this definition a representation ρ of a graph G with n vertices can be described as tuple (f, M) with $f : V(G) \to [n]$ bijectively mapping the vertices to the arcs $A_1, \ldots, A_n$ and position list M being the underlying model $\rho(G)$.

To canonize an HCA graph a canonical HCA representation of it is computed. A representation, or more accurately a function assigning graphs to representations, is canonical if two isomorphic graphs G and H always receive representations ρ_G and ρ_H such that their underlying models $\rho_G(G)$ and $\rho_H(H)$ are identical. Taking the intersection graph of the underlying model then enables us to canonize the graphs.

In order to demonstrate how to obtain a canonical HCA representation in $\mathtt{L}$ we proceed in three steps. First, we show that finding a canonical HCA representation of a graph G can be reduced to finding a canonical interval representation of a certain matrix $\lambda_G^{(M)}$. Next, we calculate a Δ tree which contains all possible interval representations for a given interval graph. And in the final step we show how to canonically choose one of these interval representations.

4.1 Turning HCA graphs into interval matrices

In Figure 4.2 it becomes apparent that a CA graph can have structurally different models. This concept will be formalized as intersection matrix of a CA system. An analogous idea, namely neighborhood matrices, for graphs is introduced. Linking these both concepts enables us to unambiguously associate certain CA systems with CA graphs.

In [Hsu95] it was observed that the relation between two circular arcs A, B can be described as one of the following: disjoint($\mathtt{di}$), A contains B($\mathtt{cs}$), A is contained by B($\mathtt{cd}$), both overlap($\mathtt{ov}$) or both jointly cover the circle($\mathtt{cc}$).

Definition 4.7. *An intersection matrix is a matrix μ with entries $\mu_{u,v} \in \{\mathtt{di}, \mathtt{cd}, \mathtt{cs}, \mathtt{ov}, \mathtt{cc}\}$ such that if $\mu_{u,v} = \mathtt{cs}$ then $\mu_{v,u} = \mathtt{cd}$ and else $\mu_{u,v} = \mu_{v,u}$ for all $u \neq v \in V(\mu)$*

Definition 4.8 ([KKV13, Def. 2.1])**.** *Let $\mathcal{A}$ be a CA system. The intersection matrix induced by $\mathcal{A}$ is denoted by $\mu_{\mathcal{A}} = (\mu_{A,B})_{A \neq B \in \mathcal{A}}$ and defined by the entries*

$$\mu_{A,B} = \begin{cases} \mathtt{di} & , \text{if } A \cap B = \emptyset \\ \mathtt{cd} & , \text{if } A \subsetneq B \\ \mathtt{cs} & , \text{if } B \subsetneq A \\ \mathtt{cc} & , \text{if } A \between B \text{ and } A \cup B \text{ covers the whole circle} \\ \mathtt{ov} & , \text{if} A \between B \text{ and } A \cup B \text{ does not cover the whole circle} \end{cases}$$

The intersection matrix induced by an interval system is also covered by this definition, however the case $\mathtt{cc}$ can't occur.

Definition 4.9. *An intersection matrix μ is called a CA matrix (HCA matrix, interval matrix) if there exists a CA system (resp. HCA system, interval system) $\mathcal{S}$ such that $\mu \cong \mu_{\mathcal{S}}$*

Definition 4.10. *A CA representation (HCA representation, interval representation) ρ with underlying model $\mathcal{A}$ is a representation of a CA matrix (resp. HCA matrix, interval matrix) μ if $\mu \cong \mu_{\mathcal{A}}$ via ρ*

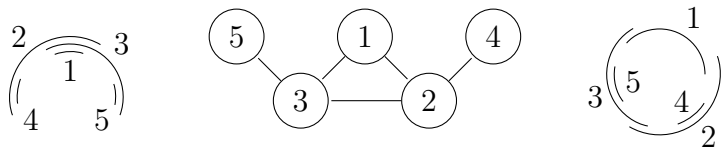

Figure 4.3: Normalized and non-normalized representation of an HCA graph

The next step is to transfer the concept of intersection matrices to graphs. Given a graph we define its intersection matrix that is called the neighborhood matrix of the graph.

Definition 4.11. *The neighborhood matrix $\lambda_G = (\lambda_{u,v})_{u \neq v \in V(G)}$ of a graph G is defined by the entries*

$$\lambda_{u,v} = \begin{cases} \texttt{di} & , \textit{if } \{u,v\} \notin E(G) \\ \texttt{cd} & , \textit{if } N[u] \subsetneq N[v] \\ \texttt{cs} & , \textit{if } N[v] \subsetneq N[u] \\ \texttt{cc} & , \textit{if } N[u] \not\subset N[v] \textit{ and } N[u] \cup N[v] = V(G) \\ & \quad \textit{and } \forall w \in N[u] \setminus N[v] : N[w] \subset N[u] \\ & \quad \textit{and } \forall w \in N[v] \setminus N[u] : N[w] \subset N[v] \\ \texttt{ov} & , \textit{otherwise} \end{cases}$$

The neighborhood matrix can be seen as augmented adjacency matrix where 0 entries correspond to `di` and 1 entries to `cd`, `cs`, `cc` or `ov`. If the intersection matrix is clear from the context we also write u `cs` v for $\lambda_{u,v} =$ `cs`, u `cd` v for $\lambda_{u,v} =$ `cd` and so on.

Linking these both ideas gives rise to normalized representations.

Definition 4.12. *A CA representation ρ of a CA graph G is called normalized if ρ is an isomorphism between the neighborhood matrix λ_G and the intersection matrix $\mu_{\rho(G)}$ induced by the underlying model of ρ*

In Figure 4.3 the intersection matrix of the left representation equals the neighborhood matrix of the depicted graph. Therefore this is a normalized representation. For the right representation there is a mismatch between the neighborhood matrix and the intersection matrix. For example, the vertices $1, 3$ in the neighborhood matrix are associated with the entry `cd`, i.e. 1 is contained by 3, whereas the intersection matrix of the representation contains an `ov` entry meaning the arcs 1 and 3 overlap. Thus it isn't a normalized representation.

Hsu investigated under what circumstances such a normalized representation can exist and provided a constructive proof of the fact that

Lemma 4.13 ([KKV13, Lem. 2.3])**.** *Any twin-free CA graph G without universal vertices has a normalized CA representation*

Because we are particularly interested in HCA graphs it is of importance that:

Lemma 4.14 ([KKV13, Lem. 2.4])**.** *Any normalized CA representation of a twin-free HCA graph G without universal vertices is also an HCA representation of G*

Proofs for these Lemmas can be found in [Hsu95] and [JLM+11, Thm. 4.1].

As a consequence of Lemma 4.14 we just need to argue that the calculated CA representation of the input HCA graph is normalized which implies that it must be Helly. Using this fact we will reduce the canonical HCA representation problem for HCA graphs to the canonical CA representation problem for vertex-colored HCA matrices.

Problem Canon-HCA-Repr(HCA Graph)

Input	HCA graph G
Output	Canonical HCA representation of G

Problem Canon-CA-Repr(Col. HCA Matrix)

Input	HCA matrix μ, coloring $c : V(\mu) \to \mathbb{N}$
Output	Canonical CA representation (ρ, c_ρ) of (μ, c)

Recall Definition 4.10, a CA representation ρ with underlying model $\mathcal{A}$ is a representation of a CA matrix μ if $\mu \cong \mu_{\mathcal{A}}$ via ρ. This means that the arcs of $\mathcal{A}$ have the pairwise relations specified by the intersection matrix μ. For a colored CA matrix (μ, c) the tuple (ρ, c_ρ) is a CA representation if the previous holds and additionally $c(v) = c_\rho(\rho(v))$ for all $v \in V(\mu)$.

Lemma 4.15. *The* Canon-HCA-Repr(HCA Graph) *problem can be solved in* L *using a* Canon-CA-Repr(Col. HCA Matrix) *oracle*

Proof. Let G be the input HCA graph without universal vertices. For every twin class in G choose the smallest vertex as representative of this class and remove the other ones from G. Color the representative with the number of vertices that used to be in its twin class. Let us call this modified graph G' and the accompanying coloring c. Calculate a canonical CA representation (ρ, c_ρ) for the neighborhood matrix $\lambda_{G'}$ of G' using the oracle. Now, given a vertex v of G let $\hat{v}$ denote its twin representative we have chosen before. With that we define the final canonical CA representation ρ' as

$$\rho'(v) = \rho(\hat{v}) \text{ for all } v \in V(G)$$

To understand why this algorithm works as intended the following questions need to be accounted for

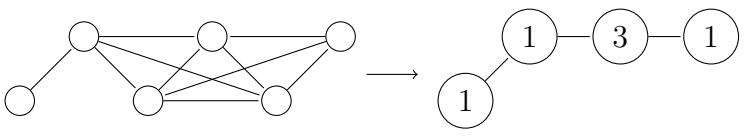

Figure 4.4: Reducing twins as described in Lemma 4.15
The labels of the right graph correspond to the coloring

1. Is $\lambda_{G'}$ a correct input to the oracle?
2. Is ρ' an HCA representation of G?
3. Is ρ' canonical?
4. Why does this work in L?

The neighborhood matrix $\lambda_{G'}$ must be an HCA matrix in order to be a correct input to the oracle. That means there exists an HCA system $\mathcal{A}$ such that $\lambda_{G'}$ is isomorphic to its intersection matrix $\mu_{\mathcal{A}}$. Since G' has no universal vertices and twins there exists a normalized CA representation ρ with underlying CA model $\mathcal{A}$ due to Lemma 4.13. Lemma 4.14 states that $\mathcal{A}$ is also an HCA model. Per definition $\lambda_{G'}$ is isomorphic to $\mu_{\mathcal{A}}$ via ρ and therefore must be an HCA matrix thus answering the first question.

Due to the same arguments ρ must be an HCA representation of G'. ρ' is computed by duplicating certain arcs according to the size of the twin classes encoded in the coloring. It is not hard to see that the Helly property is preserved and that ρ' must be a correct representation of G therefore concluding the second question.

To see that ρ' is canonical assume that two isomorphic HCA graphs G and H are given. By construction the twin-free colored graphs $(G', c_G), (H', c_H)$ of G and H respectively are isomorphic iff $G \cong H$. Therefore G and H receive representations with identical underlying models.

Lastly, we argue why this algorithm can be implemented in logspace. Computing the twin-free graph G' and its coloring can be accomplished using the following subroutines: determine whether two vertices are twins, determine the twin class and its size for a given vertex, choose the smallest twin of a twin class as representative. All of these routines are logspace computable. For the second one can simply derive ρ' from ρ.

As stated in the beginning of this chapter universal vertices can be excluded w.l.o.g. since they can be treated before applying this algorithm □

In the next step we show how to convert an HCA system into an interval system by flipping certain arcs. More formally, the problem of canonical CA representation for vertex colored HCA matrices is reduced to the canonical representation problem for vertex colored interval matrices.

Problem Canon-Intv-Repr(Col. Interval Matrix)

Input	Interval matrix μ, coloring $c : V(\mu) \to \mathbb{N}$
Output	Canonical interval representation (ρ, c_ρ) of (μ, c)

Consider a CA system $\mathcal{A}$ and an arbitrary point x on the circle. Let $X = \{A \in \mathcal{A} \mid x \in A\}$ denote the set of arcs in $\mathcal{A}$ which contain this point x. Let $\mathcal{A}^{(X)}$ be the CA system which is obtained by flipping all arcs which contain the point x

$$\mathcal{A}^{(X)} = (\mathcal{A} \setminus X) \cup \left\{ \bar{A} \,\middle|\, A \in X \right\}$$

It is not hard to see that $\mathcal{A}^{(X)}$ must be an interval system as no arc covers the point x. However, we want to transform an HCA matrix μ into an interval matrix. So, a

Table 4.1: Effects of flipping arcs in the intersection matrix

$\mu_{A,B}$	di	cd	cs	cc	ov
$\mu_{\bar{A},B}$	cs	cc	di	cd	ov
$\mu_{A,\bar{B}}$	cd	di	cc	cs	ov
$\mu_{\bar{A},\bar{B}}$	cc	cs	cd	di	ov

representation of μ is not at our disposal. Therefore we have to determine how an intersection matrix changes if an arc is flipped and how a set X of arcs can be found that share a common point x using only the information of the HCA matrix.

The first difficulty can be solved in a straightforward fashion. Given an intersection matrix μ of a CA system $\mathcal{A}$ and a set of arcs $X \subseteq \mathcal{A}$ to be flipped then Table 4.1 describes a mapping from μ to $\mu^{(X)}$ which is the intersection matrix of $\mathcal{A}^{(X)}$.

The second difficulty is to obtain an appropriate subset X of arcs that when flipped lead to an interval system. This can be accomplished by finding an inclusion-maximal subset of arcs which share a common point on the circle. Maximality here means that no other arc can be added to X such that the previously common point remains.

Lemma 4.16. *For any maxclique M of an HCA system $\mathcal{A}$ there exists a point which all the arcs of M have in common*

Proof. Since all arcs of M have pairwise non-empty intersection and $\mathcal{A}$ is Helly it follows that they have overall non-empty intersection which yields a common point. □

Further, it is not hard to see that any such maxclique is maximal in the above sense. Unfortunately, finding a maxclique for a graph is well-known to be a difficult task in general. Luckily, for the case of HCA graphs at least one of the maxcliques can be characterized as the common neighborhood of two vertices.

Theorem 4.17. *For an HCA graph G there exist two vertices $u, v \in V(G)$ (possibly $u = v$) such that $N[u, v]$ is a maxclique*

Proof. For an HCA graph G with vertex set V let λ be the neighborhood matrix of G and ρ be a normalized representation of G.

To find the wanted maxclique we look for a vertex v such that there is no vertex x with v cs x. This implies that there cannot be a vertex x' with v cc x' as this would require a vertex which is contained by v thus contradicting the non-existence of x stated previously. If there isn't a vertex u such that v ov u then $\mathrm{N}[v]$ is a maxclique. By method of elimination it follows that for all vertices $x \in \mathrm{N}[v]$ it must hold that v cd x which means $\mathrm{N}[v] \subset \mathrm{N}[x]$. Therefore for $x, x' \in \mathrm{N}[v]$ we get $x' \in \mathrm{N}[v] \subset \mathrm{N}[x]$ showing that $\mathrm{N}[v]$ indeed constitutes a clique for which maximality can be easily seen.

Otherwise, there is a vertex u with u ov v. Choose such a u that $\mathrm{N}[u, v]$ is inclusion-minimal, which means that for any other u' that overlaps with v it holds that $\mathrm{N}[u, v] \subset \mathrm{N}[u', v]$. We claim that $\mathrm{N}[u, v]$ is a maxclique. To show maximality assume there is a vertex $x \notin \mathrm{N}[u, v]$ such that $\mathrm{N}[u, v] \cup \{x\}$ is still a clique. The clique property implies that x must be in the neighborhood of u and v which is a contradiction.

To show that $\mathrm{N}[u, v]$ is a clique as claimed we derive a contradiction from the statement that there exist two vertices $x, x' \in \mathrm{N}[u, v]$ such that $x' \notin \mathrm{N}[x]$. Recall that for any $y \in \mathrm{N}[v]$ it holds that $\lambda_{v,y} \in \{\texttt{cd}, \texttt{ov}\}$. Therefore we have three cases on our hands

1. x `cs` v, x' `cs` v
2. x `ov` v, x' `ov` v
3. x `cs` v, x' `ov` v

A fourth case obtained by swapping x and x' in the third case is redundant as $x \notin \mathrm{N}[x']$ iff $x' \notin \mathrm{N}[x]$. In the first and third case it is apparent that x and x' must be adjacent for the same reasons. If $\rho(x')$ intersects with $\rho(v)$ it must intersect with $\rho(x)$ as well since $\rho(v) \subset \rho(x)$. This leaves us with the second case. There are three ways for an arc $\rho(y)$ to overlap with $\rho(v)$ and intersect with $\rho(u)$ at the same time. One possibility is that exactly all three arcs $\rho(y), \rho(u), \rho(v)$ and no less jointly cover the circle. This contradicts the Helly property and thus cannot be the case. The second possibility is that $\rho(y)$ and $\rho(u)$ jointly cover the circle and the third one is that $\rho(y)$ overlaps with $\rho(v)$ from the same side as $\rho(u)$ does; consider Figure 4.5 for these three cases in the same order. The only non-trivial case to be accounted for is if $\rho(x)$ covers the circle with $\rho(u)$ and $\rho(x')$ simply overlaps with $\rho(v)$ from the same side as $\rho(u)$ does. In that case $\mathrm{N}[x', v] \subset \mathrm{N}[u, v]$ thus contradicting our choice of u which is depicted in Figure 4.5 as well. □

Equipped with the last Theorem 4.17 we can formulate the proof for the final reduction of this section.

Theorem 4.18. *The* Canon-CA-Repr(Col. HCA Matrix) *can be solved in* L *using a* Canon-Intv-Repr(Col. Interval Matrix) *oracle*

Proof. Given a colored HCA matrix (μ, c) with vertex set V Algorithm 4.7 solves the problem. Correctness follows from answering the following questions

1. Why is $\rho_{\mu,M}$ a CA representation of μ?
2. Why is the outputted representation canonical?
3. Why does this work in L?

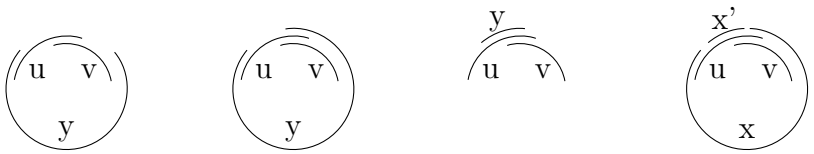

Figure 4.5: Cases occurring in the last step of the proof for Theorem 4.17
$\rho(\cdot)$ has been omitted

Algorithm 4.7 Find canonical CA representation for a given colored HCA matrix (μ, c)

1. Find all pairs $u, v \in V(\mu)$ s.t. $\mathrm{N}[u, v]$ is a maxclique in the graph given by μ when considering all non-disjoint entries to represent an edge. Let $\mathcal{M}$ denote the set of all the respective maxcliques

2. For each $M \in \mathcal{M}$ calculate $\mu^{(M)}$ and mark the vertices in M by modifying the coloring, i.e. for $v \in M$ let the new color be $c'(v) = 2c(v)$ and for $v \notin M$ let it be $c'(v) = 2c(v) + 1$

   2.1. Compute a canonical interval representation of $\mu^{(M)}$ and its modified coloring c' using the oracle and flip back all marked arcs in the underlying model of the canonical representation. This yields a CA representation $\rho_{\mu,M}$ of μ. To reconstruct the coloring c from c' simply halve even colors and for odd colors subtract one and halve them

3. Return the CA representation with the lexicographically smallest model among $\{\rho_{\mu,M} \mid M \in \mathcal{M}\}$ along with its coloring

---

Let μ be an HCA matrix with a maxclique M and ρ is an interval representation of $\mu^{(M)}$. Let ρ' be the same representation as ρ but for every $v \in M$ $\rho'(v)$ is exactly the flipped arc of $\rho(v)$. Consider the relation between $\mu, \mu^{(M)}, \rho$ and ρ'.

$$(\mu^{(M)})_{u,v} = \begin{cases} \mu_{\overline{u},v} & \text{, if } u \in M, v \notin M \\ \mu_{u,\overline{v}} & \text{, if } v \in M, u \notin M \\ \mu_{\overline{u},\overline{v}} & \text{, if } u, v \in M \\ \mu_{u,v} & \text{, else} \end{cases}$$

The overlines above the indices in the three cases refer to the mapping induced by Table 4.1. Let $\mu_\rho, \mu_{\rho'}$ be the intersection matrices induced by the underlying models of ρ and ρ' respectively. Then $\mu^{(M)}$ and μ_ρ are isomorphic via ρ. The matrices μ_ρ and $\mu_{\rho'}$ have the same relation as μ and $\mu^{(M)}$:

$$(\mu_{\rho'})_{u,v} = \begin{cases} (\mu_\rho)_{\overline{u},v} & \text{, if } u \in M, v \notin M \\ (\mu_\rho)_{u,\overline{v}} & \text{, if } v \in M, u \notin M \\ (\mu_\rho)_{\overline{u},\overline{v}} & \text{, if } u, v \in M \\ (\mu_\rho)_{u,v} & \text{, else} \end{cases}$$

Since flipping the same arc twice doesn't change the arc it holds that $(\mu^{(M)})^{(M)} = \mu$ and thus

$$\mu = (\mu^{(M)})^{(M)} \cong (\mu_\rho)^{(M)} = \mu_{\rho'}$$

which shows that ρ' is a CA representation of μ.

Given two isomorphic HCA matrices G, H they lead to the same set of maxcliques $\mathcal{M}$. It follows that G and H receive identical models in their representations which respect the modified colorings.

Obviously, the first and second step can be accomplished in logspace. To find the lexicographically smallest model consider the ordered version of $\mathcal{M}$ as induced by the graph labeling. We introduce two pointers with initial values 1 and 2 which refer to the maxcliques in $\mathcal{M}$ by their position. Whenever the model induced by the maxclique referred to by pointer 1 is larger than the other then advance pointer 1 and vice versa. Repeat this until one pointer leaves the valid range. The other pointer yields the maxclique leading to the smallest model.

□

4.2 Calculating the $\triangle$ Tree of an interval matrix

Due to the last reduction it remains to show that a canonical interval representation for colored interval matrices can be computed in L. We start by describing interval orientations and how they correspond to interval representations. For an intersection matrix λ let $G_{\texttt{ov,di}}$ be the graph on the vertex set $V(\lambda)$ with an edge between u and v whenever $\lambda_{u,v} \in \{\texttt{di}, \texttt{ov}\}$.

Definition 4.19. *A transitive orientation of an undirected graph $G = (V, E)$ is a directed graph $D = (V, E')$ which is obtained by assigning each edge of G a direction such that D is transitive. This means E' must obey*

$$\{u, v\} \in E \iff \text{either } (u, v) \in E' \text{ or } (v, u) \in E'$$

and

$$(u, v), (v, w) \in E' \implies (u, w) \in E'$$

Definition 4.20. *An interval orientation of an interval matrix λ is a transitive orientation $D_{\texttt{ov,di}}$ of $G_{\texttt{ov,di}}$ that remains transitive when all edges (u, v) in $G_{\texttt{ov,di}}$ with $\lambda_{u,v} = \texttt{ov}$ are removed and furthermore it holds that*

$$\lambda_{u,v} = \texttt{di} \wedge \lambda_{u,w} = \lambda_{v,w} = \texttt{ov} \implies \text{either } (u, w) \in E(D_{\texttt{ov,di}}) \text{ or } (v, w) \in E(D_{\texttt{ov,di}})$$

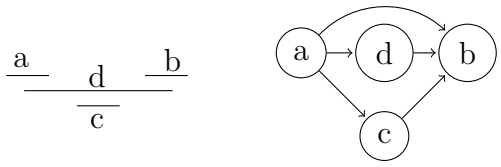

Figure 4.6: Interval orientation induced by interval representation

It is not hard to see that any interval representation ρ of an interval matrix λ induces an interval orientation $D_{\texttt{ov,di}}$. An edge $\{u, v\} \in G_{\texttt{ov,di}}$ is oriented as (u, v) in $D_{\texttt{ov,di}}$ iff $\rho(u)$ is left of $\rho(v)$. An example is given in Figure 4.6. Every interval orientation is a partial order on the set of vertices due to the transitivity requirement. The last condition of Definition 4.20 states that whenever there is an interval (resp. vertex) w which is placed in between two disjoint intervals u, v then in the order defined by the interval orientation w must lie between u and v as well. Considering Figure 4.6 this means that there cannot be an interval orientation such that there is an edge from d to a and from d to b for the intersection matrix specified by the interval model.

Not only does every interval representation induce an interval orientation but the converse holds as well.

Lemma 4.21. *For an interval matrix λ with interval orientation $D_{\texttt{ov,di}}$ there exists an interval representation ρ which induces $D_{\texttt{ov,di}}$*

Proof. Given an interval matrix λ with vertex set V and an interval orientation $D_{\texttt{ov,di}}$ of it. Let $D_{\texttt{ov,di,cs}}$ be the directed graph $D_{\texttt{ov,di}}$ with additional edges from u to v for $u, v \in V$ whenever u $\texttt{cs}$ v. Analogously define $D_{\texttt{ov,di,cd}}$ with an edge from u to v whenever u $\texttt{cd}$ v. We claim that $D_{\texttt{ov,di,cs}}$ and $D_{\texttt{ov,di,cd}}$ define total orders on V. As stated before $D_{\texttt{ov,di}}$ defines a partial order. Any pair of vertices which is incomparable in this partial order must be in a containment relation. Adding edges as described by $D_{\texttt{ov,di,cs}}$ (or $D_{\texttt{ov,di,cd}}$) leads to a total order. It remains to check case-by-case that the transitivity property is maintained.

Now, the relative order of the left endpoints of the intervals in representation ρ is given by the total order defined by $D_{\texttt{ov,di,cs}}$ and similarly for the right endpoints the relative order is determined by $D_{\texttt{ov,di,cd}}$. Interleaving both orders such that the entries of λ are obeyed yields ρ.

For correctness the following easy claim has to be verified. Given an interval representation ρ which induces $D_{\texttt{ov,di}}$ it holds for any two intervals u and v that $l(u)$ is left of $l(v)$ iff u comes before v in the order given by $D_{\texttt{ov,di,cs}}$. An analogous claim for the right endpoints and $D_{\texttt{ov,di,cd}}$ has to be made. It follows that there is an interleaving of both order sequences such that the entries of λ are obeyed and thus the constructed ρ must be correct. □

Let us apply this construction to the interval orientation $D_{\texttt{ov,di}}$ displayed in Figure 4.6 to reconstruct its interval representation ρ. Let λ be the intersection matrix of the underlying model of ρ. The directed graph $D_{\texttt{ov,di,cs}}$ is the same as $D_{\texttt{ov,di}}$ with an additional edge from d to c since d contains c. The order of $D_{\texttt{ov,di,cs}}$ is (a, d, c, b) thus the relative order of the left endpoints is $l(a) < l(d) < l(c) < l(b)$. The order of $D_{\texttt{ov,di,cd}}$ is (a, c, d, b) which implies $r(a) < r(c) < r(d) < r(b)$. To obtain the representation we start with the sequence of left endpoints and insert the right endpoints successively in a suitable position. Obviously, for any interval the left endpoint comes before the right endpoint. This means $r(a)$ must be placed after $l(a)$. As a overlaps with d and is disjoint with c it follows that $r(a)$ must be placed between $l(d)$ and $l(c)$. Completing this construction by placing each right endpoint at the first possible place that is conform

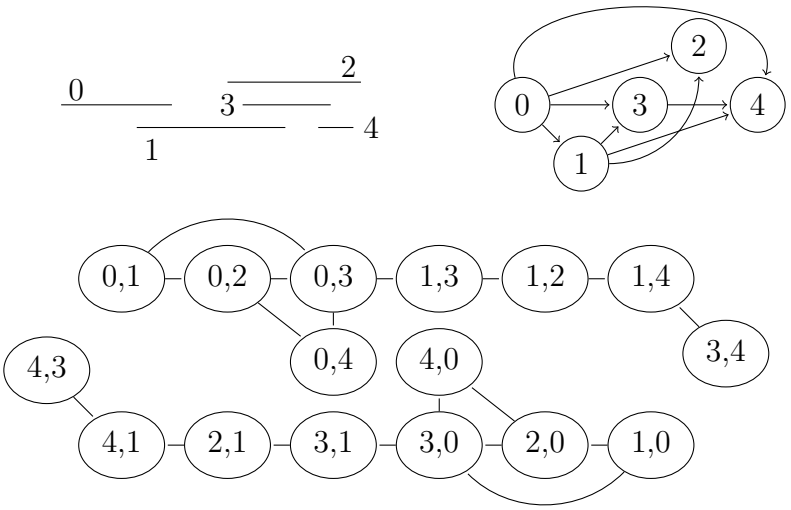

Figure 4.7: Interval system with its Δ implication classes and an interval orientation

with the entries of the intersection matrix λ leads to the same interval representation as shown in Figure 4.6. This construction enables us to calculate an interval representation in `L` given an interval matrix and an interval orientation as input. Therefore the next step is to obtain such an interval orientation for a given interval matrix.

Definition 4.22 ([KKV13, Def. 4.2])**.** *Let λ be an intersection matrix and let $\{u, v\}$ and $\{u, w\}$ be edges in $G_{\texttt{ov,di}}$. The binary relation Δ contains the entries $(u, v)\Delta(u, w)$ and $(v, u)\Delta(w, u)$ iff one of the following holds*

(1) u `di` v, u `di` w and $\lambda_{v,w} \neq$ `di`

(2) $\lambda_{v,w} \in \{\texttt{cs}, \texttt{cd}\}$

(3) u `di` v, u `ov` w, w `ov` v

The idea behind this definition is that for an entry $(u, v)\Delta(u, w)$ w.r.t. some interval matrix λ it must hold that in any interval representation ρ of λ $\rho(v)$ and $\rho(w)$ must be on the same side of $\rho(u)$. In the first case u is disjoint from v and w which both have non-empty intersection. In the second case v is contained by w (or vice versa) and as $\{u, v\}, \{u, w\}$ are edges in $G_{\texttt{ov,di}}$ none of them contains or is contained by v. In the third case w is placed in between u and v. It easily follows that for every case $\rho(v)$ and $\rho(w)$ must be on the same side of $\rho(u)$ indeed.

When trying to construct an interval representation of an interval matrix λ one has to decide for two adjacent vertices u, v in $G_{\texttt{ov,di}}$ which one is put on the left side of the other. This decision has implications on the positioning of the other intervals which are captured by the following definition.

Definition 4.23. *The Δ implication classes are the equivalence classes of the symmetric transitive closure of Δ*

Definition 4.24. *A Δ color class is the union of a Δ implication class and its transpose*

In Figure 4.7 let λ be the intersection matrix of the shown interval system. The bottom graph G has $E(G_{\texttt{ov,di}})$ as its vertex set and the symmetric closure of Δ as its edge relation. The connected components of G are exactly the Δ implication classes. Let us denote the connected components containing the vertex (x, y) with $C_{x,y}$. Then $C_{4,3}$ is the transpose of $C_{3,4}$ and the union of both form a Δ color class.

Lemma 4.25 ([McC03, Thm. 6.4])**.** *Each interval orientation of an interval matrix λ contains exactly one Δ implication class from each Δ color class*

However, not every selection of one Δ implication class from each Δ color class leads to an interval orientation. In the case of Figure 4.7 this difficulty doesn't occur as the interval model only consists of one Δ color class. Therefore the two possible interval orientations are given by $C_{4,3}$ and $C_{3,4}$ with the nodes of the connected components corresponding to the edge relation of the interval orientation. The interval orientation shown in Figure 4.7 is obtained by picking $C_{3,4}$.

The set of all valid selections of Δ implication classes is supplied by an object called Δ tree. In the following, a number of hardly digestible definitions describing the Δ tree along with its relation to the Δ color classes are given. A concrete example of such a Δ tree is shown at the end.

Definition 4.26. *A module of a matrix λ is a set $U \subseteq V(\lambda)$ such that $\lambda_{u,v} = \lambda_{u',v}$ and $\lambda_{v,u} = \lambda_{v,u'}$ for all $u, u' \in U$ and $v \in V(\lambda) \setminus U$*

In the case of an adjacency matrix of an undirected graph G this means that for any pair of vertices u, u' in a module U of G there is an edge between u and v iff there is one between u' and v for all v not in the module U. One could say that the vertices of the module U are not distinguishable from the outside of the graph, i.e. $V \setminus U$.

Definition 4.27. *A module U of an intersection matrix λ is a Δ module if it is a clique in the corresponding intersection graph or if there is no $v \in V(\lambda) \setminus U$ such that $\lambda_{v,u} = \texttt{ov}$ for all $u \in U$*

The intuition is that a Δ module can be seen as the result of two kinds of substitution operation on interval systems as remarked in [McC03, p. 107]. Consider two interval systems $\mathcal{I}, \mathcal{I}'$. Let $x \in \mathcal{I}$ be an interval such that its two endpoints are consecutive. That means no interval overlaps with x and x contains no interval. Now, replace the single interval x with the set of intervals in $\mathcal{I}'$, i.e. replace the both endpoints of x with

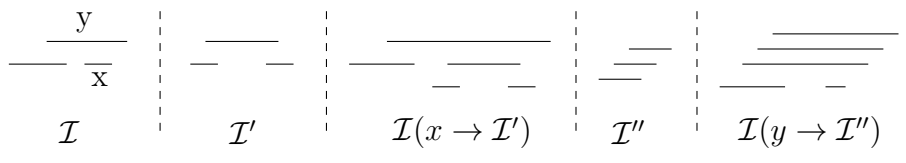

Figure 4.8: Two kinds of substitution operations leading to Δ modules

the position list of $\mathcal{I}'$. The result is denoted by $\mathcal{I}(x \to \mathcal{I}')$. Next, consider an interval system $\mathcal{I}''$ such that the left endpoints and right endpoints of its intervals are consecutive ($\mathcal{I}''$ satisfies this condition iff all intervals have overall non-empty intersection and hence form a Helly clique). For an arbitrary $y \in \mathcal{I}$ we can now replace y's left endpoint with the position list of left endpoints in $\mathcal{I}$ and do the same for the right endpoints. An example of both operations can be seen in Figure 4.8 with respective resulting interval systems $\mathcal{I}(x \to \mathcal{I}')$ and $\mathcal{I}(y \to \mathcal{I}'')$. The second kind of operation leads to a module which must be a clique and the first one to a module such that there is no interval outside of the module which overlaps with one of its intervals as described in the definition. Additionally note that if $\mathcal{I}_r$ is the result of any such substitution operation involving interval systems I_1, I_2 then two interval orientations for I_1 and I_2 induce an interval orientation for I_r. Stated differently, we can calculate independent interval orientations for I_1 and I_2 to obtain one for I_r.

The set of Δ modules are a tree-decomposable family which leads to the Δ tree.

Definition 4.28 ([McC03, cf. Def. 6.7])**.** *A set system $\mathcal{S}$ w.r.t. some set V is called a tree-decomposable family if it satisfies the following properties*

1. *V and its singleton subsets, i.e. $\{x\}$ for all $x \in V$, are in $\mathcal{S}$*
2. *Overlap closure: If $X, Y \in \mathcal{S}$ overlap, i.e. $X \between Y$, then $X \cup Y, X \cap Y, X \setminus Y, Y \setminus X, (X \setminus Y) \cup (Y \setminus X)$ are members of $\mathcal{S}$ as well*

A member X of such a tree-decomposable family $\mathcal{S}$ is called strong if it does not overlap with any other member.

Lemma 4.29 ([McC03, cf. Thm. 6.8])**.** *If $\mathcal{S}$ is a tree-decomposable family then the transitive reduction of the containment relation on strong members of $\mathcal{S}$ is a tree*

Lemma 4.30 ([KKV13, p. 9])**.** *The Δ modules of an interval matrix λ form a tree-decomposable family*

Altogether, a Δ tree T has the set of strong Δ modules of an intersection matrix λ as its vertex set. The transitive reduction of the containment relation among these strong Δ modules constitutes the edge relation. This means there is an edge $(A, A') \in E(T)$ iff A is the smallest superset of A'. Such a smallest superset is unique as otherwise the supersets in question must overlap and therefore wouldn't be strong. Every Δ tree of an intersection matrix λ has $V(\lambda)$ as its root node and the singleton subsets of $V(\lambda)$ as its leaves. These are also called the trivial nodes of T.

We will refer to the inner nodes, which are all but the leaves, as either prime or degenerate nodes according to

Definition 4.31 ([MS99, cf. Thm. 2.2])**.** *Let U be an inner node with children $C = \{W_1, \ldots, W_k\}$ of a Δ tree T w.r.t an intersection matrix λ. U is called prime iff there doesn't exist a strict subset C' of C consisting of at least two elements such that the union of its members $\bigcup_{U' \in C'} U'$ is a Δ module. Otherwise, U is called degenerate*

A degenerate node can be further classified as either overlap, disjoint or containment node.

Definition 4.32. *If U is an inner node with children $W_1, \ldots, W_k$ in the Δ tree of λ then the quotient of λ at U is the submatrix $\lambda[U] = (\lambda_{u,v})_{u \neq v \in W}$ with $W = \{w_1, \ldots, w_k\}$ and $w_i \in W_i$ for all $1 \leq i \leq k$*

Notice, since $W_1, \ldots, W_k$ are strong modules and therefore don't overlap pairwise. The choice of w_i's in W_i doesn't affect the quotient matrix. It can be seen as operation where the vertex set of each child module is shrunk into a single vertex.

Lemma 4.33 ([McC03, p. 110])**.** *A degenerate node U of a Δ tree of an interval matrix λ can be classified as one of the following according to its quotient $\lambda[U]$:*

- *Overlap node: $\lambda[U]$ consists of only* `ov` *entries*
- *Disjoint node: $\lambda[U]$ consists of only* `di` *entries*
- *Containment node: $\lambda[U]$ consists of only* `cs` *and* `cd` *entries*

McConnell has proven the following results which show that Δ trees enable us to compactly represent all interval representations.

Lemma 4.34 ([McC03, Lem. 6.14])**.** *The set of vertices spanned by a Δ color class in an interval matrix λ is a Δ module of λ*

Lemma 4.35 ([McC03, Thm. 6.15])**.** *A set of edges of $G_{ov,di}$ is a Δ color class iff it is the set of edges connecting all children of a prime node or a pair of children of a degenerate node in the Δ tree*

Lemma 4.36 ([McC03, Thm. 6.19])**.** *Any acyclic union of Δ implications classes from each Δ color class gives an interval orientation of λ*

Let us take a look at Figure 4.9 to visualize the new definitions and lemmas. The intersection matrix of the interval system is called λ with $V(\lambda) = \{1, \ldots, 13\}$. The graph underlying λ consists of three connected components to which the three children of the root node in the Δ tree correspond. It is not hard to see that for any connected component the interval orientation can be chosen independently. It remains to select a linear order of the connected components as whole. Every connected component is a Δ module because it is disjoint from the rest and therefore doesn't overlap with any other interval. It remains to be checked that these are strong Δ modules. The root node V is degenerate because the union of any of its children is a Δ module as well. Furthermore, V is a disjoint node because its quotient $\lambda[V]$ consists of three pairwise disjoint intervals. The rest of the inner nodes happen to be prime in this case. To see that 6-9 and 8-9 are Δ modules recall that Δ modules can be seen as result of an substitution operation as shown in Figure 4.8. We show the reverse operations. Take the intervals 8 and 9 and shrink them into the interval $8 \cup 9$. The substitution of $8 \cup 9$ with the interval system $\mathcal{I} = \{8, 9\}$ is of the second type, i.e. the left and right endpoints of $\mathcal{I}$ are arranged

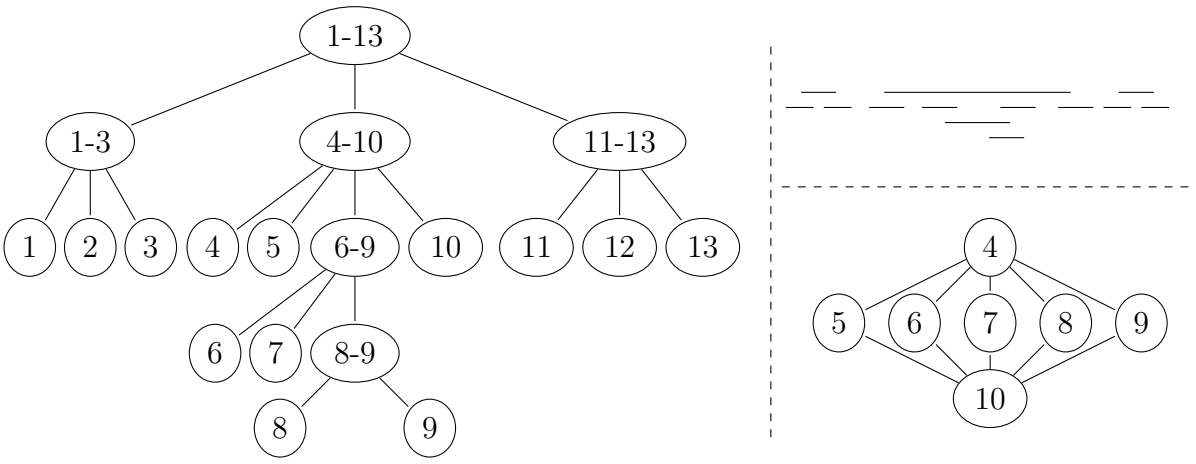

Figure 4.9: An interval system, its Δ tree and a Δ color class
The intervals are labeled according to the order of their left endpoints from 1 to 13

consecutively. Now, shrink $6, 7$ and $8 \cup 9$ into the single interval $6 \dots 9$. The substitution here is of the first type which means the endpoints of $6 \dots 9$ are consecutive.

For Lemma 4.35 let us consider the node 4-10. Since 4-10 is a prime node a Δ color class is given by the edges in the graph $G_{\mathtt{ov,di}}$ of the quotient $\lambda[4-10]$. The graph shown in the bottom right of Figure 4.9 can be obtained by constructing $G_{\mathtt{ov,di}}$ of the quotient λ[4-10] and then replacing each non-trivial child with its members such that they inherit the edge relation. By doing the reverse, that is merging the vertices $6, 7, 8, 9$ into a single one, the graph $G_{\mathtt{ov,di}}$ of the quotient λ[4-10] is obtained.

An example of Lemma 4.34 can be seen in the previous Figure 4.7 for which the Δ tree consists of only trivial nodes and the vertex set spanned by the sole Δ color class is the whole vertex set and therefore the root node.

Now, we can show how to extract the different interval orientations of λ from a Δ tree.

Lemma 4.37. *For an interval matrix λ with $U_1, \dots, U_k$ as the inner nodes of its Δ tree a set of interval orientations $D_1, \dots, D_k$ for the quotient matrices $\lambda[U_1], \dots, \lambda[U_k]$ induces an interval orientation D for λ. The induced D can be computed in* L

Proof. We start off by specifying the induced D for λ. For an inner node U_i with children $W_1, \dots, W_k$ and interval orientation D_i for $\lambda[U_i]$ the interval orientation D receives the following edges. Whenever $(W_i, W_j) \in E(D_i)$ then $(u, v) \in E(D)$ for all $u \in W_i$ and $v \in W_j$. This can be calculated in L.

D contains a Δ implication class from every Δ color class because of Lemma 4.35. It remains to argue that D is acyclic. This implies that D is an interval orientation of λ due to Lemma 4.36.

We show that for two edges $(u, v), (v, w)$ in D it follows that there is an edge (u, w). Therefore, a directed cycle C in D would imply that there is a directed cycle between any pair of vertices participating in C. This would contradict that $D_1, \dots, D_k$ are interval orientations and therefore only contain one Δ implication class of each Δ color class.

Let $U_{u,v}, U_{v,w}$ be the inner nodes that contain u, v resp. v, w in different children. There must be a path from $U_{u,v}$ to $U_{v,w}$ or vice versa in the directed Δ tree. Let us assume that there is a path from $U_{v,w}$ to $U_{u,v}$, that means $U_{v,w}$ is on a higher level. Because there is an edge from v to w in D there is an edge from the child containing v to the child containing w in the interval orientation associated with the quotient matrix $\lambda[U_{v,w}]$. Since u and v occur in the same child module w.r.t. $U_{v,w}$ it follows that $(u, w) \in E(D)$ by construction. The same argument can be made in the case that $U_{u,v}$ is on a higher level than $U_{v,w}$. □

Given the Δ tree of an interval matrix λ interval orientations for the quotient matrices of the inner nodes can be easily calculated. If U is a prime node then choose one of the two possible Δ implication classes from the Δ color class connecting U's children. For a degenerate U of type disjoint or overlap any linear ordering of its children is a valid interval orientation. If U is of type containment no edges have to be oriented.

Finally, we prove that the Δ tree can be computed in L.

Theorem 4.38. *The Δ implications classes, the Δ color classes and the Δ tree can be computed in L*

Proof. The Δ implication and Δ color classes of an interval matrix λ are given by the connected components of the graph $G = (E(G_{\mathtt{ov},\mathtt{di}}), E)$ with E being the symmetric closure of the binary relation Δ. This is the same construction as shown in Figure 4.7.

Consider the overlap graph O which has the vertex sets spanned by the Δ color classes as its node set and the edges are given by the overlap relation on these vertex sets. By Lemma 4.34 the nodes of O are Δ modules. The connected components of O are also Δ modules due to the fact that the union of two overlapping Δ modules $U_1 \between U_2$ is a Δ module as well, which follows from the overlap closure of tree-decomposable families, see Definition 4.28. The Δ modules given by the connected components of O are strong since they don't overlap with any other Δ module and therefore are nodes of the Δ tree.

The connected components of O correspond to the nodes of the type prime, disjoint and overlap in the Δ tree because they don't overlap with any other Δ module and thus must be strong Δ modules.

It remains to calculate the nodes of type containment for the Δ tree. First, we show that every containment node is the union of some non-containment nodes which have been already calculated. Let U be an inner node of type containment. The structure of its quotient matrix $\lambda[U]$ induces a linear order on its children. Let its children $U_1, \ldots, U_k$ be ordered accordingly, i.e. for all $u_a \in U_a$ and $u_b \in U_b$ with $1 \leq a < b \leq k$ it holds that $u_a \ \mathtt{cd}\ u_b$. Note that no child U_a of U can be a containment node. Otherwise we could construct an overlapping Δ module by taking the union of the children of U_a and one of the siblings of U_a right next to it in the linear order. This contradicts that U_a must be a strong Δ module.

$$\forall u_1 \in U_1, u_2 \in U_2, u_3 \notin U_1 \cup U_2 : u_1 \ \mathtt{cd}\ u_2 \wedge \lambda_{u_1,u_3} = \lambda_{u_2,u_3}$$

There is an edge from U_a to U_b in C iff they both occur in the same containment node and U_a comes right before U_b in the order mentioned above. Therefore the connected components of C correspond to the containment nodes.

At last, add the leaf nodes, which are the singleton subsets of $V(\lambda)$, and calculate the edge relation of the Δ tree as transitive reduction of the containment relation among the given nodes. More precisely, there is and edge between U_a and U_b iff U_b is the smallest superset of U_a as argued in the paragraph after Lemma 4.30.

□

Corollary 4.39. *For a given interval matrix λ an interval representation can be calculated in* L

This is accomplished by computing the Δ tree of an interval matrix λ using Theorem 4.38 and arbitrary interval orientations for the quotient matrices of the inner nodes. Then the interval representation is induced by the interval orientation according to Lemma 4.21.

4.3 Canonically choosing an interval orientation

To obtain a canonical interval representation we will canonize interval matrices. The idea is to enrich the Δ tree with enough information such that its interval matrix can be reconstructed. This enriched Δ tree will be called the colored Δ tree $\mathbb{T}(\lambda)$ of an interval matrix λ. Using the remark after Corollary 2.12 the colored tree $\mathbb{T}(\lambda)$ can be canonically labeled, which is denoted by $\hat{\mathbb{T}}(\lambda)$. Constructing the intersection matrix λ' of $\hat{\mathbb{T}}(\lambda)$ yields the canonical form of λ. The use of Corollary 4.39 leads to a canonical interval representation.

First, let us consider for each type of inner node U of a Δ tree of an interval matrix λ what kind of different interval models M_U for the quotient matrix $\lambda[U]$ are possible and how the children of U have to be mapped to the intervals in M_U.

If U is a prime node then $\lambda[U]$ consists of exactly one color class due to Lemma 4.35. This means there can be at most two interval models (and interval representations) for U given by the two Δ implications classes. If these two interval models happen to be equal then U is called symmetric and its interval model admits two mappings of the children to the intervals. Otherwise, each of the two models for U allows only one mapping of the children to its intervals.

Consider the Δ tree of the intersection matrix λ of the interval system in Figure 4.7. The root node is clearly a prime node whose children are all leaves. In an interval system it is clear that the left endpoint comes always before the right endpoint, therefore we can omit $l(\cdot), r(\cdot)$ in the position list. For an interval model let us fix the labeling of the intervals by the order of appearance from left to right starting with 1. For the interval system of Figure 4.7 its interval model would be

$$M_1 = 1, 2, 1, 3, 4, 2, 5, 4, 5, 3$$

If one takes the other Δ implication class this yields

$$M_2 = 1, 2, 3, 2, 4, 3, 1, 5, 4, 5$$

as model. Let the tuples $(f_1, M_1), (f_2, M_2)$ be interval representations of λ with bijections $f_1, f_2 : \{0, \ldots, 4\} \to \{1, \ldots, 5\}$. In this case the prime root node isn't symmetric as $M_1 \neq M_2$. Additionally, f_1 and f_2 are unique. This means that both models allow only one mapping of the children to the intervals.

For an example of a symmetric prime node consider node $U = 11\text{-}13$ in Figure 4.9. It is not hard to verify that U has exactly one model $M = 1, 2, 1, 3, 2, 3$. The two possible mappings of U's children to M allow $11, 13$ to be mapped either to $1, 3$ or $3, 1$ respectively.

Notice, for every prime node U the interval model(s) of its quotient matrix $\lambda[U]$ and one (resp. both) mappings of the children to the intervals can be computed in L.

For any degenerate node U with n children its model and mapping is uniquely determined by its type and n. If U is of type disjoint then its model M is given by n pairwise disjoint intervals, i.e. $1, 1, 2, 2, \ldots, n, n$. If U is of type overlap the M is given by n pairwise overlapping intervals, i.e. $1, 2, \ldots, n, 1, 2, \ldots, n$. In both cases the mapping of U's children to the intervals of the model can be arbitrary. If U is of type containment then $M = 1, 2, \ldots, n, n, \ldots, 2, 1$ and the mapping of the children to the intervals is fixed by the containment order.

A subsumption of the previous observations

1. symmetric prime U: one model M_U, two mappings from U's children to M_U
2. asymmetric prime U: two models $M_{U,1}, M_{U,2}$, one mapping for each model
3. overlap or disjoint U: one model, arbitrary mapping
4. containment U: one model, one mapping

Definition 4.40. *Given an interval matrix λ its colored Δ tree $\mathbb{T}(\lambda)$ is defined as follows. $\mathbb{T}(\lambda)$ has all of the nodes of the Δ tree of λ. Additionally, for each symmetric prime node U there are three nodes lo_U, mi_U, hi_U between U and its children. Each inner node U receives a tuple (p_U, M_U) as color, where M_U is the interval model of the quotient $\lambda[U]$ (if there are two different models take the smaller one) and p_U is the position of U among the children of its parent: If U is the root or if the parent of U admits an arbitrary mapping of its children to the quotient intervals then $p_U = 0$. If the parent of U has a fixed assignment of its children to the intervals then let p_U be the position of the interval corresponding to U in the quotient of its parent. If the parent U' of U allows two assignments of its children, let $p_{U,1}, p_{U,2}$ be the positions of U under the two assignments, respectively. If $p_{U,1} < p_{U,2}$ make U a child of $lo_{U'}$ and define $p_U = (p_{U,1}, p_{U,2})$, if $p_{U,1} = p_{U,2}$ make U a child of $mi_{U'}$ and $p_U = (p_{U,1}, p_{U,2})$, otherwise let U be a child of $hi_{U'}$ and $p_U = (p_{U,2}, p_{U,1})$. Finally, color all lo_U, hi_U nodes with -1 and all mi_U nodes with -2*

With the help of Theorem 4.38 we can calculate the colored Δ tree for a given interval matrix in L.

Lemma 4.41. *If λ and λ' are isomorphic interval matrices, then $\mathbb{T}(\lambda) \cong \mathbb{T}(\lambda')$*

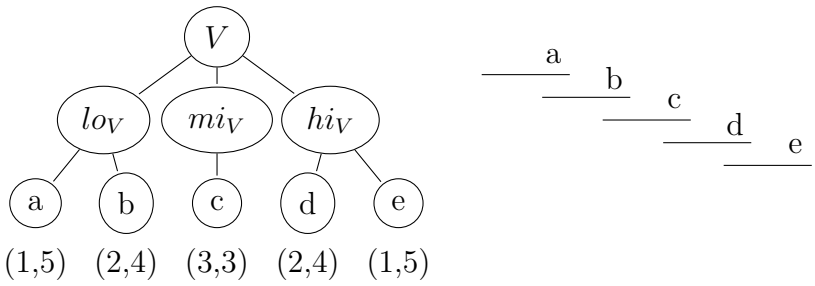

Figure 4.10: Colored Δ tree with symmetric node and its children's position tuples

Proof. For two isomorphic interval matrices λ, λ' let $\pi : V(\lambda) \to V(\lambda')$ be an isomorphism between them. We construct an isomorphism φ between $\mathbb{T}(\lambda)$ and $\mathbb{T}(\lambda')$.

Observe that π induces a bijection between the Δ modules of λ and λ', i.e. $U = \{v_1, \ldots, v_k\}$ is a Δ module of λ iff $\varphi(U) = U' = \{\pi(v_1), \ldots, \pi(v_k)\}$ is a Δ module of λ'. It follows that the Δ trees of λ and λ' are isomorphic as well as the quotient matrices $\lambda[U]$ and $\lambda'[U']$. Therefore the models $M_U, M_{U'}$ chosen for the colored Δ trees $\mathbb{T}(\lambda), \mathbb{T}(\lambda')$ are identical. It can be easily seen that the positions for U and U' are equal if U is a degenerate or asymmetric prime node. If U is a symmetric prime node then φ has to be extended with either

$$\begin{pmatrix} lo_U & mi_U & hi_U \\ lo_{U'} & mi_{U'} & hi_{U'} \end{pmatrix} \text{ or } \begin{pmatrix} lo_U & mi_U & hi_U \\ hi_{U'} & mi_{U'} & lo_{U'} \end{pmatrix}$$

For a symmetric prime node U the two possible mappings to its model M_U are the reverse of each other. More precisely, if U has k children then for a child U_x which has position p_x in the first mapping to model M_U it must have position $k - p_x + 1$ in the other mapping to M_U. Thus, in the relabeling of λ given by π it is possible that the children of lo_U and hi_U have been swapped. Otherwise, the children of the lo_U and hi_U node must be the same in $\mathbb{T}(\lambda')$. □

Consider the intersection matrix λ with $V = \{a, \ldots, e\}$ of the interval system shown in Figure 4.10. Let λ' be an isomorphic copy of λ via $\pi = \left(\begin{smallmatrix} a & b & c & d & e \\ e & d & c & b & a \end{smallmatrix}\right)$. In the colored Δ tree of λ' the nodes a, b are connected to the *hi* node and d, e to the *lo* node. Thus, when trying to construct an isomorphism φ between λ and λ' the nodes lo_V, hi_V have to be swapped in φ.

Lemma 4.42. *Let λ be an interval matrix. Given an isomorphic copy T of $\mathbb{T}(\lambda)$ with an isomorphism $\varphi : \mathbb{T}(\lambda) \to T$, an isomorphic copy λ' of λ along with an isomorphism $\pi : \lambda \to \lambda'$ can be computed in* L *depending only on T*

Proof. The first step in reconstructing λ' from the colored Δ tree T is to define the vertex set of λ' as the leaves of T. Since the leaves of T correspond to the vertices of λ via φ this already yields π. More specifically, let $\pi(v) = v'$ iff $\varphi(\{v\}) = \{v'\}$. This means for all $u, v \in V(\lambda)$ λ' needs to satisfy

$$\lambda_{u,v} = \lambda'_{\pi(u),\pi(v)}$$

To construct λ' using only T we proceed as follows. Let U' be an inner node of T that isn't colored -1 or -2, i.e. isn't a lo_U, mi_U, hi_U node, such that $\varphi(U) = U'$. This means that U is the corresponding inner node in $\mathbb{T}(\lambda)$. Let $\lambda'[U']$ denote the intersection matrix of the model M_U associated with U' via the coloring. Since U and U' share the same model M_U it holds that $\lambda[U]$ is isomorphic to $\lambda'[U']$. Let a_U be the mapping which assigns the children of U to the intervals in M_U. Observe that if we can compute this mapping then we obtain an isomorphism between $\lambda[U]$ and $\lambda'[U']$

$$\lambda[U]_{U_1,U_2} = \lambda'[U']_{a_U(\varphi(U_1)),a_U(\varphi(U_2))}$$

This enables us to construct all entries of λ'. For any two vertices u', v' of λ' let $U(u', v')$ denote the inner node in T which has u' and v' in different children. The entry $\lambda'_{u',v'}$ is given by $\lambda[U(u', v')]_{U',V'}$ where U', V' are the children which contain u', v' respectively.

To compute the the mapping a_U for an inner node U' one needs to consider the positions of the children of U':

- If the children of U' all have position 0 (U' is an overlap or disjoint node), then the mapping a_U can be arbitrary. Define a_U such that it preserves the order of $U's$ children, that is if child U_i comes before U_j then $a_U(U_i) < a_U(U_j)$.
- If the chidren of U' all have pairwise different positions (U' is an asymmetric prime or containment node), then a_U is directly given by the positions of the children
- If U' has three children of which two have color -1 and one has color -2 (U' is a symmetric prime node) then let $mi_{U'}$ be the node with color -2 and $lo_{U'}, hi_{U'}$ be the nodes with color -1 such that $lo_{U'}$ comes before $hi_{U'}$ in the order of $V(T)$. For the children of $lo_{U'}$ and $mi_{U'}$ use the first entry of the position tuple as mapping in a'_U. For the children of $hi_{U'}$ use the second entry of the position tuple. This leads to one of two possible assignments.

□

Algorithm 4.8 Compute canonical representation for interval matrix λ

1. Compute the colored Δ tree $\mathbb{T}(\lambda)$ (Theorem 4.38 and Definition 4.40)
2. Compute the canonized version $\hat{\mathbb{T}}(\lambda)$ of $\mathbb{T}(\lambda)$ along with an isomorphism $\varphi : V(\mathbb{T}(\lambda)) \to V(\hat{\mathbb{T}}(\lambda))$ (Corollary 2.8)
3. Obtain a canonical form λ' of λ along with an isomorphism $\pi : \lambda \to \lambda'$ using $\hat{\mathbb{T}}(\lambda)$ and φ (Lemma 4.42)
4. Compute an interval representation $\rho' = (f', M)$ of λ' (Corollary 4.39). Then (f, M) with $f(v) = f'(\pi(v))$ for all $v \in V(\lambda)$ is a canonical interval representation of λ.

Theorem 4.43. *The canonical representation problem for interval matrices can be solved in* L

Proof. Algorithm 4.8 computes such a canonical interval representation in L. □

Corollary 4.44. *The canonical representation problem for colored interval matrices can be solved in* L

Proof. Let (λ, c) be the colored interval matrix and $\mathbb{T}(\lambda)$ is the colored Δ tree of λ. There is an obvious one-to-one correspondence between the vertex set of λ and the leave nodes of $\mathbb{T}(\lambda)$. For a vertex $v \in V(\lambda)$ let U_v denote the corresponding leave node in $\mathbb{T}(\lambda)$. To incorporate the coloring c into the Δ tree append the color of a vertex $v \in V(\lambda)$ to the color of U_v, i.e. if U_v already has the color x then make it $(x, c(v))$.

□

Corollary 4.45. *The canonical HCA representation problem for HCA graphs can be solved in* L

Corollary 4.46. *The isomorphism and automorphism problem for HCA and interval graphs are* L*-complete*

Corollary 4.47. *The recognition of HCA graphs can be solved in* L

If a graph G isn't an HCA graph the presented algorithm to calculate an HCA representation will either fail at one of the steps, for example no suitable maxclique M can be found, or the returned CA representation ρ is no representation of G, which can be easily checked, or it isn't Helly. It is shown in [JLM+11, Thm. 3.1] how to check whether a given CA system is Helly.

4.4 CA graph isomorphism

As mentioned in the preface it is still unknown whether isomorphism for CA graphs can be decided in polynomial time. It was shown in [McC03] that CA graphs can be recognized in linear time and for a given CA graph a CA representation can be computed in linear time. However, the representation isn't canonical as this would imply a canonization algorithm.

Considering that we have just proved that HCA graphs can be canonized one might wonder wherein lies the difficulty in adapting this method to the general class of CA graphs? Recall that the canonical representation problem for HCA matrices (and therefore HCA graphs) is reduced to the canonical representation problem for interval matrices in Section 4.1 by finding a suitable set of vertices which can be flipped to obtain an interval matrix. The crux here is that every maxclique in an HCA graph must share a common point in the normalized representation of that graph due to the Helly property, see Lemma 4.16. Furthermore, at least one maxclique of an HCA graph can be found by checking the common neighborhood of all pairs of vertices as stated in Theorem 4.17. For CA graphs both statements fail, see Figure 4.11 for a counterexample to the

first statement (add additional arcs to force this to be a normalized representation) and [KKV13, Fig. 2(c)] for the latter.

An alternative approach is to study what makes a CA graph non-HCA. Luckily, this has been already answered by [JLM+11].

Definition 4.48. *A CA system $\mathcal{A}$ is called non-Helly triangle if it consists of three arcs such that all three arcs and no less jointly cover the circle*

Theorem 4.49 ([JLM+11, cf. Thm. 3.1])**.** *Let $\mathcal{A}$ be a CA system and M_ℓ is the position list of $\mathcal{A}$ after removing the right endpoints. Then $\mathcal{A}$ is minimally non-Helly iff either it is a non-Helly triangle or $\mathcal{A}$ consists of at least four arcs and for every pair of arcs in $\mathcal{A}$ it must hold that they cover the circle precisely when their (left) endpoints are not consecutive in M_ℓ*

Minimality here means that no arc in a minimally non-Helly CA system can be removed without making it Helly. Consider Figure 4.11 again for an example of two such minimally non-Helly culprits. For a CA graph G this means it isn't an HCA graph if and only if for all CA representations ρ of G there exists a subset $V' \subseteq V(G)$ such that the arcs in ρ corresponding to V' form a minimally non-Helly CA system.

In the following we show how to recognize non-Helly triangles in graphs and why this might be of interest when trying to solve the isomorphism problem for CA graphs.

Definition 4.50. *For a CA graph G that is no interval graph we call a triangle $\{u, v, w\} \subseteq V(G)$ non-Helly if the following holds*

1. $N[u] \cup N[v] \cup N[w] = V(G)$
2. $N[x] \setminus (N[y] \cup N[z]) \neq \emptyset$ *for all* $x \neq y \neq z \in \{u, v, w\}$

Lemma 4.51. *Let G be a CA graph which is no interval graph. A triangle $\{u, v, w\} \subseteq V(G)$ is non-Helly in G iff in every CA representation ρ of G it holds that $\{\rho(u), \rho(v), \rho(w)\}$ is a non-Helly triangle.*

Proof. Let $T = \{u, v, w\}$ be a triangle in G. As G is no interval graph it holds that in every CA representation ρ of G the arcs of $\rho(G)$ cover the whole circle. Let us write $\rho(T)$ to refer to the arcs $\rho(u), \rho(v), \rho(w)$.

We start by showing the direction "⇒". Assume that there exists a CA representation ρ of G such that the arcs $\rho(T)$ don't cover the whole circle and hence are three pairwise

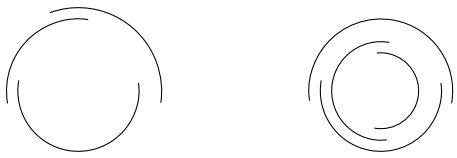

Figure 4.11: Two minimally non-Helly CA systems

overlapping intervals. This implies that $\rho(x) \subseteq \rho(y) \cup \rho(z)$ for some $x \neq y \neq z \in T$ and therefore $\mathrm{N}[x] \subseteq \mathrm{N}[y] \cup \mathrm{N}[z]$ which contradicts the second condition in Definition 4.50.

Next, assume there exists a CA representation ρ of G in which less than all three arcs $\rho(u), \rho(v), \rho(w)$ jointly cover the circle. This implies that there is a x such that $\rho(x) \subseteq \rho(y) \cup \rho(z)$ for $x \neq y \neq z \in T$. Again, this contradicts the second condition of Definition 4.50.

For the other direction "$\Leftarrow$" assume that the first condition of Definition 4.50 fails. This means there exists a vertex $x \in V(G)$ such that it isn't connected to any vertex in T. Because the arcs $\rho(T)$ must cover the whole circle in every CA representation this is obviously false. Next, assume that the second condition fails, i.e. there is a vertex x such that $\mathrm{N}[x] \subseteq \mathrm{N}[y] \cup \mathrm{N}[z]$ for $x \neq y \neq z \in T$. Let ρ be a CA representation of G. By assumption the arcs $\rho(T)$ must be a non-Helly triangle in this representation. With ρ we can construct a CA representation ρ' of G by extending the arcs $\rho(y), \rho(z)$ such that they form a circle cover. As x has no exclusive neighbor it holds that ρ' is a CA representation of G as well. Clearly, this contradicts the initial assumption that T must be represented as non-Helly triangle in every representation. □

Corollary 4.52. *If a CA graph G contains no circle covers and non-Helly triangles it must be an HCA graph*

Two vertices u, v of a graph G form a circle cover if $\lambda_{u,v} = \mathtt{cc}$ in the neighborhood matrix λ of G as formulated in Definition 4.11.

It is imaginable that the same concept as used in the proof of Corollary 2.12 can be employed for CA graphs. That means converting a CA graph G into an HCA graph $f(G)$ by adequately replacing all non-Helly triangles and circle covers. Then we could use the canonization algorithm for HCA graphs to solve the problem. Consider Figure 4.12 for an idea on how to replace these structures by assuming the normalized representation for CA graphs. A problem for circle covers is that their intersection consists of two non-consecutive parts. This means we need to introduce two new vertices and adequately connect the common neighbors of the circle cover to these two new vertices. It is not clear how to decide to which of the both new vertices a neighbor should be connected.

For a non-Helly triangle this ambiguity doesn't exist. Let T_1 and T_2 be two non-Helly triangles in a CA graph G. If T_1 and T_2 don't share a common vertex then the two substitutions can be performed in arbitrary order. However, if they share a common vertex it isn't clear what to do. This motivates the following definitions.

Definition 4.53. *For a CA graph G the set Δ_G is defined as the set of non-Helly*

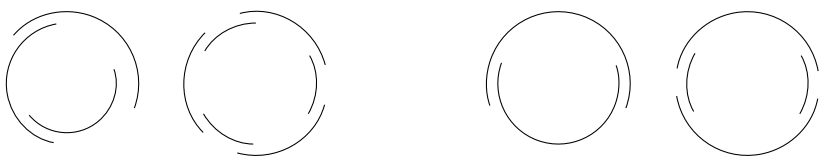

Figure 4.12: Helly substitutes for non-Helly triangles and circle covers

triangles in G. Two non-Helly triangles $T_1, T_2 \in \Delta_G$ are said to be related if $T_1 \cap T_2$ is not empty

Definition 4.54. *Let R be a binary relation on the set of non-Helly triangles Δ_G of a CA graph G. Then xRy holds iff the non-Helly triangles x and y are related. Let R^* be the transitive closure of R. We call an equivalence class of R^* a family of non-Helly triangles*

A family of non-Helly triangles can also be understood as certain subset of Δ_G. The families of non-Helly triangles partition Δ_G.

Let us write $V(\Delta_G)$ to denote the set of vertices spanned by Δ_G, i.e. $v \in V(\Delta_G)$ if there exists a $T \in \Delta_G$ with $v \in T$. Then we can define isomorphism for non-Helly triangles as

Definition 4.55. *Let G, H be CA graphs. We say Δ_G and Δ_H are isomorphic if there exists a bijection $\pi : V(\Delta_G) \to V(\Delta_H)$ such that*

$$\{u, v, w\} \in \Delta_G \iff \{\pi(u), \pi(v), \pi(w)\} \in \Delta_H$$

Can we efficiently decide the isomorphism for non-Helly triangles? This boils down to deciding the isomorphism for two families of non-Helly triangles for the same reason GI reduces to isomorphism of connected graphs by matching the connected components of both graphs. Notice that the intersection structure among the vertices spanned by the set of non-Helly triangles is ignored by this definition. This additional requirement can be incorporated as follows.

Definition 4.56. *Let G, H be CA graphs with neighborhood matrices λ_G, λ_H. We say Δ_G and Δ_H are strongly isomorphic if they are isomorphic via π and for all $u \neq v \in V(\Delta_G)$ the following holds*

$$(\lambda_G)_{u,v} = (\lambda_H)_{\pi(u),\pi(v)}$$

The previous definitions can be slightly modified to adjust them to circle covers and the same question about isomorphism for circle covers can be asked.

List of Figures

2.1 Exemplary directed tree T and its string representation $str(T)$ 7
2.2 Mapping f described in the proof of Corollary 2.12 17
2.3 Canonization reduction argument 18
2.4 Reduction from ORD to PATHCENTER 19
2.5 Reduction from PATHCENTER to isomorphism of undirected trees 21

3.1 Example of a 2-tree and its tree decomposition 23
3.2 Two non-isomorphic 2-trees that have isomorphic tree representations . . 25
3.3 A 2-tree G and its tree representation $T(G)$ rooted at its center 26
3.4 Reduction from PATHCENTER to automorphism of k-trees 30

4.1 HCA graph and a non-Helly CA representation 32
4.2 Two different interval models of the same graph 32
4.3 Normalized and non-normalized representation of an HCA graph 35
4.4 Reducing twins as described in Lemma 4.15 36
4.5 Cases occurring in the last step of the proof for Theorem 4.17 39
4.6 Interval orientation induced by interval representation 41
4.7 Interval system with its Δ implication classes and an interval orientation 43
4.8 Two kinds of substitution operations leading to Δ modules 44
4.9 An interval system, its Δ tree and a Δ color class 47
4.10 Colored Δ tree with symmetric node and its children's position tuples . . 51
4.11 Two minimally non-Helly CA systems 54
4.12 Helly substitutes for non-Helly triangles and circle covers 55

List of Tables

2.1 Applying Algorithm 2.2 to the directed tree in Figure 2.1 9

4.1 Effects of flipping arcs in the intersection matrix 38

List of Algorithms

2.1 Convert tree from string to pointer representation 8
2.2 Convert tree from pointer to string representation 9
2.3 Given trees S,T compute $<_t$-relation for the subtrees induced by s,t . . . 13
2.4 Compute distance between two nodes s, t in an undirected tree T 16

3.5 Canonically label k-trees (cf. [ADK12, Algorithm 3.1]) 24
3.6 Compute tree representation $T(G)$ on input G for fixed k 29

4.7 Find canonical CA representation for a given colored HCA matrix (μ, c) 40
4.8 Compute canonical representation for interval matrix λ 52

Bibliography

[ADK08] Arvind, Vikraman ; Das, Bireswar ; Köbler, Johannes: A Logspace Algorithm for Partial 2-tree Canonization. In: *Proceedings of the 3rd International Conference on Computer Science: Theory and Applications*. Berlin, Heidelberg : Springer-Verlag, 2008 (CSR'08). – ISBN 3–540–79708–4, 978–3–540–79708–1, 40–51

[ADKK12] Arvind, V. ; Das, Bireswar ; Köbler, Johannes ; Kuhnert, Sebastian: The Isomorphism Problem for K-trees is Complete for Logspace. In: *Inf. Comput.* 217 (2012), August, S. 1–11

[AP89] Arnborg, Stefan ; Proskurowski, Andrzej: Linear time algorithms for NP-hard problems restricted to partial k-trees. In: *Discrete Applied Mathematics* 23 (1989), Nr. 1, S. 11 – 24. – ISSN 0166–218X

[BL83] Babai, László ; Luks, Eugene M.: Canonical Labeling of Graphs. In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. New York, NY, USA : ACM, 1983 (STOC '83). – ISBN 0–89791–099–0, S. 171–183

[Bus97] Buss, Samuel R.: Alogtime algorithms for tree isomorphism, comparison, and canonization. In: Gottlob, Georg (Hrsg.) ; Leitsch, Alexander (Hrsg.) ; Mundici, Daniele (Hrsg.): *Computational Logic and Proof Theory* Bd. 1289. Springer Berlin Heidelberg, 1997, S. 18–33

[Die12] Diestel, Reinhard: *Graduate texts in mathematics*. Bd. 173: *Graph Theory, 4th Edition*. Springer, 2012. – ISBN 978–3–642–14278–9

[Ete97] Etessami, Kousha: Counting Quantifiers, Successor Relations, and Logarithmic Space. In: *Journal of Computer and System Sciences* 54 (1997), Nr. 3, S. 400 – 411. – ISSN 0022–0000

[Hsu95] Hsu, Wen-Lian: $O(M.N)$ Algorithms for the Recognition and Isomorphism Problems on Circular-Arc Graphs. In: *SIAM J. Comput.* 24 (1995), Juni, Nr. 3, S. 411–439. – ISSN 0097–5397

[JKMT03] Jenner, Birgit ; Köbler, Johannes ; McKenzie, Pierre ; Torán, Jacobo: Completeness Results for Graph Isomorphism. In: *J. Comput. Syst. Sci.* 66 (2003), Mai, Nr. 3, S. 549–566

[JLM+11] Joeris, BensonL. ; Lin, MinChih ; McConnell, RossM. ; Spinrad, JeremyP. ; Szwarcfiter, JaymeL.: Linear-Time Recognition of Helly Circular-Arc Models and Graphs. In: *Algorithmica* 59 (2011), Nr. 2, S. 215–239. – ISSN 0178–4617

[Kar72] Karp, R.: Reducibility among combinatorial problems. In: Miller, R. (Hrsg.) ; Thatcher, J. (Hrsg.): *Complexity of Computer Computations*. Plenum Press, 1972, S. 85–103

[KKV13] Köbler, Johannes ; Kuhnert, Sebastian ; Verbitsky, Oleg: Helly Circular-Arc Graph Isomorphism Is in Logspace. In: Chatterjee, Krishnendu (Hrsg.) ; Sgall, Jirí (Hrsg.): *Mathematical Foundations of Computer Science 2013* Bd. 8087. Springer Berlin Heidelberg, 2013, S. 631–642

[Köb06] Köbler, Johannes: *Complexity of Graph Isomorphism for Restricted Graph Classes (Presentation)*. 2006

[KST93] Köbler, Johannes ; Schöning, Uwe ; Torán, Jacobo: *The Graph Isomorphism Problem: Its Structural Complexity*. Basel, Switzerland, Switzerland : Birkhauser Verlag, 1993

[Lin92] Lindell, Steven: A Logspace Algorithm for Tree Canonization (Extended Abstract). In: *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*. New York, NY, USA : ACM, 1992 (STOC '92), S. 400–404

[LPPS14] Lokshtanov, Daniel ; Pilipczuk, Marcin ; Pilipczuk, Michal ; Saurabh, Saket: Fixed-parameter tractable canonization and isomorphism test for graphs of bounded treewidth. In: *CoRR* abs/1404.0818 (2014)

[McC03] McConnell, Ross M.: Linear-Time Recognition of Circular-Arc Graphs. In: *Algorithmica* 37 (2003), Nr. 2, S. 93–147

[MS99] McConnell, Ross M. ; Spinrad, Jeremy P.: Modular decomposition and transitive orientation. In: *Discrete Mathematics* 201 (1999), Nr. 13, S. 189 – 241

[RC77] Read, Ronald C. ; Corneil, Derek G.: The graph isomorphism disease. In: *Journal of Graph Theory* 1 (1977), Nr. 4, S. 339–363

[Rei05] Reingold, Omer: Undirected ST-connectivity in Log-space. In: *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*. New York, NY, USA : ACM, 2005 (STOC '05), S. 376–385

[Sch88] Schöning, Uwe: Graph isomorphism is in the low hierarchy. In: *Journal of Computer and System Sciences* 37 (1988), Nr. 3, S. 312 – 323. – ISSN 0022–0000

[Tor04] Torán, Jacobo: On the Hardness of Graph Isomorphism. In: *SIAM J. Comput.* 33 (2004), Mai, Nr. 5, S. 1093–1108. – ISSN 0097–5397

[ZKT85] Zemlyachenko, V.N. ; Korneenko, N.M. ; Tyshkevich, R.I.: Graph isomorphism problem. In: *Journal of Soviet Mathematics* 29 (1985), Nr. 4, S. 1426–1481. – ISSN 0090–4104