The Isomorphism Problem for k-Trees is Complete for Logspace

V. Arvind[a], Bireswar Das[b], Johannes Köbler[c], Sebastian Kuhnert[c,1,*]

[a] *The Institute of Mathematical Sciences, Chennai 600 113, India*
[b] *Indian Institute of Technology Gandhinagar, Ahmedabad 382 424, India*
[c] *Humboldt-Universität, Institut für Informatik, 10099 Berlin, Germany*

Abstract

We show that, for k constant, k-tree isomorphism can be decided in logarithmic space by giving an $\mathcal{O}(k \log n)$ space canonical labeling algorithm. The algorithm computes a unique tree decomposition, uses colors to fully encode the structure of the original graph in the decomposition tree and invokes Lindell's tree canonization algorithm. As a consequence, the isomorphism, the automorphism, as well as the canonization problem for k-trees are all complete for deterministic logspace. Completeness for logspace holds even for simple structural properties of k-trees. We also show that a variant of our canonical labeling algorithm runs in time $\mathcal{O}((k+1)!\, n)$, where n is the number of vertices, yielding the fastest known FPT algorithm for k-tree isomorphism.

Keywords: graph isomorphism, graph canonization, k-trees, space complexity, logspace completeness.

1. Introduction

Two graphs G and H are called *isomorphic* if there is a bijective mapping ϕ between the vertices of G and the vertices of H that preserves the adjacency relation, i.e., ϕ relates edges to edges and non-edges to non-edges. *Graph Isomorphism* (GI) is the problem of deciding whether two given graphs are isomorphic. The problem has received considerable attention since it is one of the few natural problems in NP that are neither known to be NP-complete nor known to be solvable in polynomial time.

It is known that GI is contained in coAM [17, 31] and in SPP [7], providing strong evidence that GI is not NP-complete. On the other hand, the strongest known hardness result due to Torán [33] says that GI is hard for the class DET (cf. [10]). DET is a subclass of NC^2 (even of TC^1) and contains NL as well as all logspace counting classes [3, 9].

For some restricted graph classes, the known upper and lower complexity bounds for the isomorphism problem match. For example, a linear time algorithm for tree isomorphism was already known in 1974 to Aho, Hopcroft and Ullman [1]. In 1991, an NC algorithm was developed by Miller and Reif [27], and one year later, Lindell [26] obtained an L upper bound. On the other hand, in [20] it is shown that tree isomorphism is L-hard (provided that the trees are given in pointer notation). In [6], Lindell's logspace upper bound has been extended to the class of partial 2-trees, a class of planar graphs also known as generalized series-parallel graphs. Recently, it has been shown that even the isomorphism problem for all planar graphs is in logspace [12] (in fact, excluding one of K_5 or $K_{3,3}$ as minor is sufficient [13]). Much of the recent progress on logspace algorithms for graphs has only become possible through Reingold's result that connectivity in

*Corresponding author
Email addresses: arvind@imsc.res.in (V. Arvind), bireswar@iitgn.ac.in (Bireswar Das), koebler@informatik.hu-berlin.de (Johannes Köbler), kuhnert@informatik.hu-berlin.de (Sebastian Kuhnert)

[1] Partially supported by DFG grant KO 1053/7–1.

undirected graphs can be decided in deterministic logspace [29]. Our result does not depend on this, yielding a comparatively simple algorithm.

In this article we show that the isomorphism problem for k-trees is in logspace for each fixed $k \in \mathbb{N}^+$, matching the lower bound. This combines two conference papers that improved the previously known upper bound of TC^1 [18] first to StUL [5] and then to L [23]. In fact, we prove the formally stronger result that a canonical labeling for a given k-tree is computable in logspace. Recall that the *canonization problem* for graphs is to produce a *canonical form* $canon(G)$ for a given graph G such that $canon(G)$ is isomorphic to G and $canon(G_1) = canon(G_2)$ for any pair of isomorphic graphs G_1 and G_2. Clearly, graph isomorphism reduces to graph canonization. A *canonical labeling* for G is any isomorphism between G and $canon(G)$. It is not hard to see that even the search version of GI (i.e., computing an isomorphism between two given graphs in case it exists) as well as the automorphism group problem (i.e., computing a generating set of the automorphism group of a given graph) are both logspace reducible to the canonical labeling problem.

The parallel complexity of k-tree isomorphism has been previously investigated by Del Greco, Sekharan, and Sridhar [14] who introduced the concept of the *kernel* of a k-tree in order to restrict the search for an isomorphism between two given k-trees. We show that the kernel of a k-tree can be computed in logspace and exploit this fact to restrict the search for a canonical labeling of a given k-tree G. To be more precise, we first transform G into an undirected tree $T(G)$ whose nodes are formed by all $(k+1)$-cliques and some k-cliques of G. Then we compute the center node of $T(G)$ which coincides with the kernel of G and try all labelings of the kernel vertices. In order to extend such a labeling to the other vertices of G in a canonical way, we color the nodes of the tree $T(G)$ to encode additional structural information about G. (Note that this differs from *valid colorings* using few colors.) Finally, we apply a variant of Lindell's algorithm [26] to compute a canonical labeling for the colored $T(G)$ and derive a canonical labeling for the k-tree G.

Fig. 1 shows the known inclusions between the mentioned complexity classes, where GI is the class of all problems that can be reduced to graph isomorphism in polynomial time.

As the k-tree isomorphism problem, for unbounded k, is graph isomorphism complete [21], it makes sense to study the fixed parameter tractability of this problem. A problem is called *fixed parameter tractable (FPT)* with respect to some parameter k, if it is solved by an algorithm in time $f(k)\, n^{\mathcal{O}(1)}$, where $f(k)$ can be an arbitrary function not depending on the input size n. We show that our k-tree canonization algorithm can also be implemented in $\mathcal{O}((k+1)!\, n)$ time, yielding the fastest known FPT isomorphism algorithm for this class. Previously, Klawe et al. gave an FPT isomorphism algorithm for general chordal graphs with maximum clique size $k+1$, which runs in $\mathcal{O}((k+1)!^2 n^3)$ time [21]. Nagoya improved this to $\mathcal{O}((k+1)! n^3)$ [28]. Toda gave an isomorphism algorithm that is FPT in the maximum size s of simplicial components, using $\mathcal{O}((s!\, n)^{\mathcal{O}(1)})$ time [32]. While the exact running time of this algorithm is hard to analyze, the parameter s can be smaller than $k+1$ by a factor of up to $\Theta(n)$ (and is never larger). We remark that the kernel of a k-tree is always a simplicial component, implying that $k \leq s \leq k+1$ for all k-trees. Relatedly, Yamazaki et al. showed how to check isomorphism for graphs of rooted tree distance width k in time $\mathcal{O}(k!^2 k^2 n^2)$ [36].

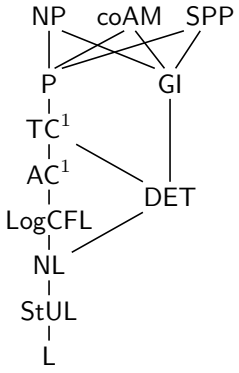

Figure 1: Known inclusions between the mentioned complexity classes

This graph class contains non-chordal graphs and is incomparable to k-trees, but like k-trees it is a subclass of treewidth k graphs.

In Section 3, we describe our algorithm and prove its correctness; in Sections 4 and 5 we give implementations in logspace and FPT, respectively. In Section 6, we show that several simple structural properties of k-trees that can be computed using our tree representation are also hard for logspace.

2. Preliminaries

As usual, L is the class of all languages decidable by Turing machines with read-only input tape and an $\mathcal{O}(\log N)$ bound on the space used on the working tapes, where N is the input size. FL is the class of all functions computable by Turing machines that additionally have a write-only output tape. Note that FL is closed under composition: To compute $f(g(x))$ for $f, g \in \mathsf{FL}$, simulate the Turing machine for f and keep track of the position of its input head. Every time this simulation needs a character from f's input tape, simulate the Turing machine for g on input x until it outputs the required character. Note also that g can first output a copy of its input x, so we can assume that f has access to both x and $g(x)$. This construction can be iterated a constant number of times, still preserving the logarithmic space bound. We will utilize this closure property by employing pre- and post-processing steps to describe our logspace algorithms.

Given a graph G, we use $V(G)$ and $E(G)$ to denote its vertex and edge sets, respectively. We define the following notations for subgraphs of G. For $M \subseteq V(G)$, $G[M]$ denotes the subgraph of G induced by M and we use $G - M$ as a shorthand for $G[V(G) \setminus M]$.

Given a graph G and two vertices $u, v \in V(G)$, the *distance* $d_G(u, v)$ is the length of the shortest path from u to v. The *eccentricity* of a vertex $u \in V(G)$ is the longest distance to another vertex, i.e., $ecc_G(u) = \max\{d_G(u, v) \mid v \in V(G)\}$. The *center* of G consists of all vertices with minimal eccentricity.

Given two graphs G and H, an *isomorphism* from G to H is a bijection $\phi\colon V(G) \to V(H)$ with $\{u, v\} \in E(G) \Leftrightarrow \{\phi(u), \phi(v)\} \in E(H)$. On colored graphs, an isomorphism must additionally preserve colors. G and H are called *isomorphic*, in symbols $G \cong H$, if there is an isomorphism from G to H. Given a graph class $\mathcal{G}$, a function f defined on $\mathcal{G}$ computes an *invariant* for $\mathcal{G}$ if

$$\forall G, H \in \mathcal{G} : G \cong H \Rightarrow f(G) = f(H)\,.$$

If the reverse implication also holds, f is a *complete invariant* for $\mathcal{G}$. If additionally $f(G) \cong G$ for all $G \in \mathcal{G}$, f computes *canonical forms* for $\mathcal{G}$. Given a function f that computes canonical forms, an isomorphism ψ_G from G to its canonical form $f(G)$ is called a *canonical labeling*.

The isomorphisms from a graph G to itself are called *automorphisms*; they form a group which we denote by $\mathrm{Aut}(G)$. An automorphism is called *non-trivial* if it is not the identity. The *graph automorphism problem (GA)* is to decide if a graph has a non-trivial automorphism. A graph without non-trivial automorphisms is called *rigid*.

Fix any $k \in \mathbb{N}^+$. The class of *k-trees* was introduced in [4] and is inductively defined as follows. Any k-clique is a k-tree. Further, given a k-tree G and a k-clique M in G, one can construct another k-tree by adding a new vertex v and connecting v to every vertex in M. The initial k-clique is called *base* of G, and the k-clique M the new vertex v is connected to is called *support* of v. Note that each k-clique of a k-tree G can be used as base for constructing G – but once the base is fixed, the support of each vertex is uniquely determined. Fig. 2 shows a 2-tree.

An interesting special case of k-trees are *k-paths*, where the support M_i of any new vertex v_i (except the first vertex added to G) must either contain the vertex v_{i-1} added in the previous step or be equal to the support M_{i-1} of v_{i-1}. Removing the vertex 8 from the graph G in Fig. 2 results in a 2-path.

Let G be a graph. A tree T is a *tree decomposition* of G, if each node $M \in V(T)$ is a subset $M \subseteq V(G)$ such that (a) for every edge $\{u, v\} \in E(G)$, there is a node $M \in V(T)$ with $\{u, v\} \subseteq M$, and (b) for every vertex $v \in V(G)$, the nodes containing v induce a non-empty subtree of T. The *width* of T is one less than the cardinality of the largest node in $V(T)$. A graph has *treewidth* k if it admits a width k tree decomposition, but none of width $k - 1$. It is well known that a graph has treewidth at most k if and only if it is a partial k-tree, i.e., a subgraph of a k-tree (see e.g. [22]).

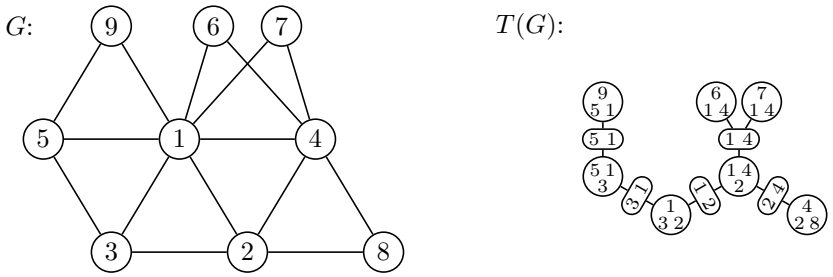

Figure 2: A 2-tree G and its tree representation $T(G)$

3. Canonizing k-trees

In this section, we describe our algorithm for k-tree canonization, starting with an overview.

Algorithm 3.1. Given a graph G, perform the following steps.

1. Compute an auxiliary graph $T(G)$ (see Definition 3.2) and check that G is indeed a k-tree. In this case, $T(G)$ will be called the *tree representation* of G.
2. Compute the *kernel* K of G, which is a set of k or $(k+1)$ vertices that induce a clique (see Definition 3.5).
3. For each labeling of the kernel, i.e., for each bijection $\pi\colon K \to \{1, \ldots, \|K\|\}$, perform the following steps:
   (a) Compute $T(G, \pi)$, which is a rooted and colored version of the tree representation $T(G)$ (see Definition 3.6).
   (b) Compute a canonical labeling $\psi_{T(G,\pi)}$ of $T(G, \pi)$.
4. Choose a labeling π_1 of the kernel such that $\psi_{T(G,\pi_1)}(T(G, \pi_1))$ becomes minimal.
5. Derive from $\psi_{T(G,\pi_1)}$ a canonical labeling ψ_G of G (see Eq. (1) in the proof of Lemma 3.8).

The remainder of this section describes this algorithm in more detail and proves that the result is indeed a canonical representation. Section 4 shows how to implement it in logspace, while Section 5 gives an FPT implementation.

We now define the *tree representation* $T(G)$ for k-trees G. Following the definition, we prove that $T(G)$ is a tree decomposition of G. We choose this particular tree decomposition because it can be computed efficiently and $G \cong H$ implies $T(G) \cong T(H)$.

Definition 3.2. For a graph G, define $T(G)$ by

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

We first give a characterization of k-trees in terms of $T(G)$.

Lemma 3.3. *G is a k-tree if and only if $T(G)$ is a tree decomposition of G.*

Proof. To prove the "only if" part, let G be a k-tree, let $\{v_1, \ldots, v_k\}$ be the base k-clique, and let $v_{k+1}, \ldots, v_n$ be the sequence in which the remaining vertices of G were added. We write G_i for $G[\{v_1, \ldots, v_i\}]$. $T(G_k)$ consists of a single k-clique node and $T(G_{k+1})$ consists of a single $(k+1)$-clique node, so they are tree decompositions of G_k and G_{k+1}, respectively. For $i > k+1$, let P_i be the support of v_i, which is a k-clique in G_{i-1}. By inductive hypothesis, $T(G_{i-1})$ is a tree representation of G_{i-1}. If P_i is not a node in $T(G_{i-1})$, it is contained in a unique $(k+1)$-clique node of that tree, and can be added as its neighbor. After that, the $(k+1)$-clique node $M_i = P_i \cup \{v_i\}$ can be added as neighbor of P_i. This results in $T(G_i)$ and can easily be seen to be a tree decomposition of G_i.

For the "if" part, let G be a graph such that $T(G)$ is a tree decomposition of G. If $T(G)$ consists of a single $(k+1)$-clique node, then G consists only of this clique and thus is a k-tree. Otherwise let M be any

k-clique node in $T(G)$ and use it as base of G. All vertices $u \in V(G) \setminus M$ can be added iteratively: Let M_u be the $(k+1)$-clique that contains u and is closest to M. Then the first k-clique on the path from M_u to M in $T(G)$ can be used as support for u. □

We now observe some structural properties of $T(G)$. Note that $T(G)$ has $\mathcal{O}(n)$ vertices, so computations involving $T(G)$ can be implemented efficiently.

Lemma 3.4. *For any k-tree G, the center of $T(G)$ is a single node.*

Proof. Suppose not. Then the center consists of two adjacent nodes, one a k-clique and one a $(k+1)$-clique. This leads to a contradiction because k-clique nodes have odd eccentricity while that of $(k+1)$-clique nodes is even: k-cliques and $(k+1)$-cliques alternate on every path, and all leaves are $(k+1)$-clique nodes (unless $T(G)$ consists of a single k-clique, which is the unique center in that case). □

Definition 3.5. The clique corresponding to the center node of $T(G)$ is called *kernel* of G and denoted $\ker(G)$.

Note that $\ker(G)$ can be either a k-clique or a $(k+1)$-clique, depending on the structure of G. The concept of the kernel of a k-tree was introduced in [14]. The definition there is slightly different but the equivalence can be easily verified.

In what follows, we consider the kernel K of G to be the root of $T(G)$. This allows us to identify each $(k+1)$-clique $M \in V(T(G)) \setminus \{K\}$ with the unique vertex $v \in M$ that is not present in the k-clique M' that lies next to M on the path from K to M in $T(G)$. For later use, we denote this vertex by $v(M)$ and for each $v \in V(G) \setminus K$, we use M_v to denote the unique $(k+1)$-clique $M \in V(T(G)) \setminus \{K\}$ with $v(M) = v$. For $v \in K$, we set $M_v = K$.

It is clear that the tree representation $T(G)$ might not provide complete structural information about G since it is possible that for an added vertex u, only one of the k edges between u and its support in G can be recovered from $T(G)$. Fig. 3 shows a 2-tree G' that is not isomorphic to G from Fig. 2, but has a tree representation $T(G')$ isomorphic to $T(G)$. To add the missing information, we give individual colors to the kernel vertices and color the nodes of $T(G)$ as well.

Definition 3.6. Let G be a k-tree with kernel K. For each vertex v of G, we denote by

$$l_G(v) = \left\lfloor d_{T(G)}(K, M_v)/2 \right\rfloor$$

the *level* of v in G (dividing by 2 has the effect of ignoring k-clique nodes for the distance). A bijection $\pi\colon K \to \{1, \ldots, k'\}$ is called *labeling* of the kernel K. For each such labeling, let $T(G, \pi)$ denote the colored directed tree obtained from $T(G)$ by choosing K as the root and coloring each node $M \in V(T(G))$ by the set $c(M) = \{c(v) \mid v \in M\}$, where

$$c(v) = \begin{cases} \pi(v) & \text{if } v \in \ker(G), \\ l_G(v) + k' & \text{otherwise.} \end{cases}$$

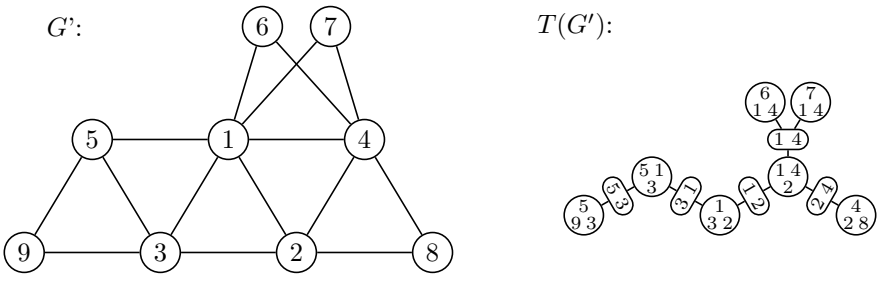

Figure 3: A 2-tree G' and its tree representation $T(G')$. G' is not isomorphic to G from Fig. 2 as it has no node of degree 7, yet $T(G') \cong T(G)$.

The definition of the colored tree $T(G, \pi)$ is similar to that of $T(G, B, \theta)$ in [5]. The main advantage of our construction lies in the fact that $T(G, \pi)$ can be directly constructed from G in logspace, whereas the tree representation used in [5] (which in turn is related to the decomposition defined in [21]) is defined as the reachable subgraph of a graph (known as mangrove) derived from G. This allows us to decide reachability in the tree $T(G, \pi)$ in logspace, an essential step to achieve our logspace upper bound in Section 4.

We now show that the colored tree representations of isomorphic k-trees are also isomorphic, provided that the kernels are labeled accordingly.

Lemma 3.7. *Let ϕ be an isomorphism between two k-trees G and H, and let K be the kernel of G. Then ϕ, viewed as a mapping from $V(T(G))$ to $V(T(H))$, is an isomorphism between $T(G, \pi_1)$ and $T(H, \pi_2)$, provided that $\pi_1(u) = \pi_2(\phi(u))$ for all $u \in K$.*

Proof. It can be easily checked that any isomorphism between G and H is also an isomorphism between $T(G)$ and $T(H)$ that maps the kernel K of G to the kernel of H. In order to show that the color of a node $M \in V(T(G, \pi_1))$ coincides with the color of $\phi(M) \in V(T(H, \pi_2))$, we prove the stronger claim that $c(v) = c(\phi(v))$ for all $v \in V(G)$. For $v \in K$, we have $c(\phi(v)) = \pi_2(\phi(v)) = \pi_1(v) = c(v)$ by assumption. Further, since ϕ must preserve the level of the vertices, it follows for $v \in V(G) \setminus K$ that

$$c(\phi(v)) = l_H(\phi(v)) + \|K\| = l_G(v) + \|K\| = c(v) \,.$$

This completes the proof of the lemma. □

Conversely, the next lemma shows that the isomorphism type of $T(G, \pi)$ contains complete information about G, proving that the algorithm outlined at the beginning of this section indeed computes a canonical labeling for G.

Lemma 3.8. *Let G be a k-tree with kernel K and let π be a labeling of the kernel K. Then from any colored directed tree T that is isomorphic to $T(G, \pi)$, an isomorphic copy G' of G can be reconstructed. Further, an isomorphism between G and G' can be constructed from any given isomorphism between $T(G, \pi)$ and T.*

Proof. For a given colored directed tree T, construct G' as follows. Let $V(G') = \{1, \ldots, n\}$, where n is k plus the number of $(k+1)$-clique nodes in T (we call $t \in V(T)$ an *l-clique node*, if $l = \|c(t)\|$). Note that G has indeed n vertices due to the one-to-one correspondence between the vertices $v \in V(G) \setminus K$ and the $(k+1)$-clique nodes $M_v \in T(G, \pi) \setminus \{K\}$. Let r be the root node of T, and let $k' = \|c(r)\|$. Next, make $\{1, \ldots, k'\}$ a clique in G'. Let v be the bijection between the non-root $(k+1)$-clique nodes of T and the remaining vertices $\{k'+1, \ldots, n\}$ of G' with $v(t) < v(t') \Leftrightarrow t < t'$, where the latter is the natural order given by the node names. For each such $(k+1)$-clique node t with color $c(t) = \{c_1, \ldots, c_{k+1}\}$, add the following edges to $E(G')$: For each $c_i \leq k'$ add an edge $\{c_i, v(t)\}$ and for each $c_i > k'$ with $c_i < c_{\max} = \max\{c_i \mid c_i \in c(t)\}$ add an edge $\{v(t), v(t')\}$, where t' is the $(c_i - k')$th node on the path from r to t. This completes the construction of G'.

Now let ϕ be an isomorphism from $T(G, \pi)$ to T. Construct an isomorphism ϕ' from G to G' as follows:

$$\phi'(v) = \begin{cases} \pi(v) & v \in K \\ v(\phi(M_v)) & v \notin K \end{cases} \tag{1}$$

By induction on the level of v in G, it can be proved that this is indeed an isomorphism. □

It remains to show that the canonical labelings of any two isomorphic k-trees G and H map these graphs to the same canon $\psi_G(G) = \psi_H(H)$.

Lemma 3.9. *Suppose G and H are isomorphic k-trees. Let ψ_G and ψ_H be the labelings computed by the algorithm outlined at the beginning of this section. Then $\psi_G(G) = \psi_H(H)$.*

Proof. Let π_1, π_2 be two kernel labelings that give rise to the lexicographically smallest trees $\psi_{T(G,\pi_1)}(T(G,\pi_1))$ and $\psi_{T(H,\pi_2)}(T(H,\pi_2))$, respectively. Since by Lemma 3.7 for any tree $T(G,\pi_1)$ that can be derived from G via some labeling π_1 there is an isomorphic tree $T(H,\pi_2)$ that can be derived from H via some labeling π_2 (and vice versa), it follows that $\psi_{T(G,\pi_1)}(T(G,\pi_1))$ and $\psi_{T(H,\pi_2)}(T(H,\pi_2))$ are equal, implying that $canon(G) = canon(H)$. □

We note that our algorithm can be extended to colored k-trees as follows. Let $\zeta\colon V(G) \to C$ be a vertex coloring of G. Modify the coloring of $T(G,\pi)$ (cf. Definition 3.6) by replacing $c(v)$ with the pair $c'(v) = (c(v), \zeta(v))$.

4. Logspace implementation

In this section, we prove that Algorithm 3.1 can be implemented in logspace. We start with the first step.

Lemma 4.1. *Given a graph G, the auxiliary graph $T(G)$ can be computed in $\mathcal{O}(k \log n)$ space. Also, it can be checked if G is a k-tree within the same space bound.*

Proof. To compute $T(G)$, first iterate over all subsets M of $V(G)$ of size $k+1$ and output M as a node if M is a $(k+1)$-clique in G. Likewise, find all k-cliques M, and count the vertices $v \in V(G) \setminus M$ for which $M \cup \{v\}$ is a $(k+1)$-clique. If there is not exactly one such v, then output M as a node, with edges to each such $M \cup \{v\}$. These steps require $(k+2)\log n$ space.

To check if G is a k-tree, it suffices by Lemma 3.3 to check if $T(G)$ is a tree decomposition of G. So we test if $T(G)$ is a tree. We also test if, for each vertex $v \in V(G)$, the subgraph of $T(G)$ induced by $\{M \in V(T(G)) \mid v \in M\}$ is a tree. In Fact 4.2 we will show how trees can be recognized in logspace. Finally, we check that all edges $\{u,v\} \in E(G)$ are contained in some $M \in V(T(G))$, which is clearly possible in logspace. □

Fact 4.2. *Given a graph $G = (V,E)$ and a node $r \in V$, in $\mathcal{O}(\log n)$ space it can be checked if G is a tree and if so, all edges of G can be directed away from r.*

Proof. Let G' be a directed copy of G where each edge $\{u,v\}$ is replaced by the two arcs (u,v) and (v,u). In a pre-processing step, try the following to compute an Euler tour of G' (cf. [2, p. 123]): Define a circular order on the neighborhood of each node using the natural order on nodes (given by their names). Start the Euler tour with the lexicographically least arc leaving $w_0 = r$. When reaching some node w_i from w_{i-1}, choose w_{i+1} as the successor of w_{i-1} in the circular order on the neighborhood of w_i. This way, only the two previous nodes have to be remembered. Stop when the arc (w_0, w_1) would be traversed again. If G is a tree, then this results in an Euler tour of G' with the following property: For all nodes v and consecutive occurrences w_i and w_j of v in this tour, it holds that $w_{i+1} = w_{j-1}$. If G contains cycles, this condition will be violated, and if G is not connected, the computed tour will not reach all nodes. This can be checked in $\mathcal{O}(\log n)$ space.

To give all edges of G an orientation directed away from r, replace each edge $\{u,v\}$ with the arc (u,v) if the latter precedes (v,u) in the above Euler tour of G'. □

To show that step 2 of the algorithm can be implemented in logspace, we recall some basic facts concerning undirected trees.

Fact 4.3. *Given an undirected tree T and two nodes $u, v \in V(T)$, the distance $d_T(u,v)$ can be computed in $\mathcal{O}(\log n)$ space.*

Proof. Compute an oriented copy of T with root u using the algorithm of Fact 4.2. Then the unique path from v to u can be found by always choosing the unique incoming edge as next step. Only the current node and the number of steps have to be remembered. Upon reaching u, output the number of steps taken. □

Fact 4.4. *The center of an undirected tree T can be computed in $\mathcal{O}(\log n)$ space.*

Proof. We first show that the eccentricity $ecc_T(u)$ of each node $u \in V(T)$ is computable in logspace. This can be done by iterating over all $v \in V(T)$, each time calculating $d_T(u, v)$ (this can be done in logspace by Fact 4.3). Only the maximum distance has to be remembered, the result being $ecc_T(u)$.

Observe now that the maximum eccentricity $ecc_{\max}$ of all nodes $u \in V(T)$ is computable in logspace by iterating over all $u \in V(T)$. Then compute again the eccentricity of all nodes u, this time outputting u if $ecc_T(u) = ecc_{\max}$. □

We now turn to step 3 of the algorithm.

Lemma 4.5. *For a k-tree G and a labeling π of the kernel K of G, $T(G, \pi)$ can be computed in $\mathcal{O}(k \log n)$ space.*

Proof. $T(G)$ can be computed within this space bound by Lemma 4.1, the kernel of G by Fact 4.4, and a copy of $T(G)$ rooted at the kernel by Fact 4.2. It remains to compute the color $c(M)$ of each node $M \in V(T(G))$.

For each $v \in M$ calculate $c(v)$ by examining the unique path from M to K in $T(G)$ (the path can be found by following the unique incoming edge at each node). Store the length ℓ of the path and the position p_v where v was last found (this can be done in parallel for all $v \in M$). If $p_v = \ell$ (i.e., $v \in K$), then add the number $c(v) = \pi(v)$ to the color $c(M)$ of M. If $p_v < \ell$, add the number $c(v) = \ell - p_v + \|K\|$ to $c(M)$. The latter is correct, because by Lemma 3.3 the nodes containing v form a subtree of $T(G)$ and thus the node that is closest to K and contains v is on the path from K to M. □

Note also that the color sets can be sorted in logspace to give them a unique string representation.

To compute a canonical labeling of $T(G, \pi)$, we observe the following generalization of Lindell's logspace tree canonization algorithm [26], which allows vertex colors and computes not only a canonical form, but a canonical labeling.

Lemma 4.6. *Given a colored directed tree T, a canonical labeling of T can be computed in $\mathcal{O}(\log n)$ space.*

Proof. We first sketch Lindell's algorithm and describe the necessary modifications afterwards. Given a rooted tree and an order on the children of each node, this tree can be traversed in logspace: When visiting a node v for the first time, go to its first child, if it has one. If v has no children or if v is visited for the second time, proceed with its next sibling, if it has one. Otherwise, return to its parent. Note that it is only necessary to store the current node and whether we arrived there from its last child.

Lindell's algorithm canonizes a rooted tree by traversing it using a *tree isomorphism order*: Two siblings s and t are ordered $s < t$ if the subtree S rooted at s has fewer nodes than the subtree T rooted at t, or (for equally large subtrees) if s has fewer children than t, or (for the same number of children) if S has a smaller canon than T. Lindell shows how to implement the recursion in the latter case without a stack to achieve the overall logspace bound; see [26] for details.

Colors can be handled by refining the tree isomorphism order with the additional condition $color(s) < color(t)$ (for efficiency, give it the highest priority). The canonical labeling can be computed by using a counter i initialized to 0; whenever a node v is visited for the first time in the traversal using the tree isomorphism order, increment i and print "$v \mapsto i$". □

Now we are ready to prove our main result.

Theorem 4.7. *Given a k-tree G, a canonical labeling ψ_G of G can be computed in $\mathcal{O}(k \log n)$ space.*

Proof. The correctness of Algorithm 3.1 was shown in Lemma 3.9. Earlier in this section we showed logspace algorithms to compute $T(G)$ (Lemma 4.1), $\ker(G)$ (Fact 4.4), and $T(G, \pi)$ (Lemma 4.5). To enumerate all labelings π of the kernel, only $k' \log k'$ bits are required. A canonical labeling of $T(G, \pi)$ can be computed in logspace by Lemma 4.6. It remains to observe that the canonical labeling ψ_G for G can be derived as in Eq. (1), which is easily possible in logspace. □

Theorem 4.7 immediately yields the following corollaries.

Corollary 4.8. *For any fixed k, k-tree (and k-path) isomorphism is* L*-complete.*

We delay the hardness part to Proposition 6.3.

Note that fixing k is essential, as the isomorphism problem for the class of all k-trees, k unbounded, is isomorphism complete [21] and hence unlikely to be decidable in polynomial time.

Furthermore, there is a standard Turing reduction of the automorphism group problem (i.e., computing a generating set of the automorphism group of a given graph) to the search version of GI for colored graphs; a similar reduction exists for counting the number of automorphisms (cf. [19, 24]). It is not hard to see that these reductions can be performed in logspace.

Corollary 4.9. *For any fixed k, a generating set of the automorphism group of a given k-tree can be computed in logspace, and hence also a canonical labeling coset for a given k-tree.*

Corollary 4.10. *For any fixed k, the number of automorphisms of a given k-tree can be computed in logspace.*

Corollary 4.11. *For any fixed k, the k-tree (and k-path) automorphism problem (i.e., deciding whether a given k-tree has a non-trivial automorphism) is* L*-complete.*

We postpone the proof of the hardness to Proposition 6.2.

5. Fixed parameter tractability

In this section, we give a time efficient implementation of the algorithm presented in Section 3.

Theorem 5.1. *Given a k-tree G, a canonical labeling ψ_G for G can be computed in $\mathcal{O}((k+1)!\, n)$ time.*

Proof. First note that k can easily be obtained from the input graph, as all inclusion-maximal cliques have size $k+1$. The tree representation $T(G)$ of the input graph G can be computed in linear time by iteratively removing simplicial vertices [30]. In this process it can also be checked that G is indeed a k-tree. Additionally, for each of the $k'! \leq (k+1)!$ labelings π of the kernel, the colored tree $T(G,\pi)$ can be constructed in linear time by first finding the center of $T(G)$, computing all distances from the center to the other tree nodes using breadth first search and finally outputting the color sets. Finally, a canonical labeling for the colored tree $T(G,\pi)$ can be computed in linear time [1, p. 84]. □

6. Complete problems for logspace

In this section we prove some completeness results for logspace that are related to our algorithm. The hardness is under $\mathsf{DLOGTIME}$-uniform AC^0-reductions. We first recall that Ord is L-complete, where Ord is the problem of deciding for a directed path P and two vertices $s,t \in V(P)$ if there is a path from s to t [16].

In Fact 4.4 we have seen that the center of an undirected tree can be computed in logspace. We now show that the decision variant is hard for L even when restricted to paths.

Theorem 6.1. *Given an undirected path P and a vertex $c \in V(P)$, it is* L*-complete to decide if c belongs to the center of P. This remains true if P is restricted to be a path of odd length.*

We call this problem PathCenter. This theorem implies that the following problem is L-hard: Given a k-tree (or k-path) G and a vertex $c \in V(G)$, decide whether c belongs to the kernel of G. The reduction for this is $(P,c) \mapsto (E_k(P),c)$, where E_k transforms a path P (resp. tree T) into a k-path $E_k(P)$ (resp. a k-tree $E_k(T)$) by adding a $(k-1)$-clique M and connecting M to all nodes in $V(P)$ (resp. $V(T)$) (cf. Fig. 4).

Proof. We reduce from Ord using $(P,s,t) \mapsto (P',n)$ as reduction, where

$$
\begin{aligned}
V(P') &= V(P) \cup \big\{ i' \;\big|\; i \in V(P) \big\} \cup \{s''\} \\
E(P') &= \big\{ \{i,j\} \;\big|\; (i,j) \in E(P) \wedge j \neq t \big\} \cup \big\{ \{n,n'\} \big\} \\
&\quad \cup \big\{ \{i',j'\} \;\big|\; (i,j) \in E(P) \wedge j \notin \{s,t\} \big\} \\
&\quad \cup \big\{ \{i',s''\} \;\big|\; (i,s) \in E(P) \big\} \cup \big\{ \{s'',s'\} \big\} \\
&\quad \cup \big\{ \{i,t'\} \;\big|\; (i,t) \in E(P) \big\} \cup \big\{ \{i',t\} \;\big|\; (i,t) \in E(P) \big\}
\end{aligned}
$$

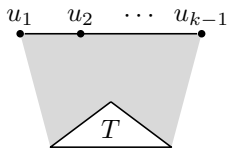

Figure 4: The transformation $E_k(T)$

and n is the vertex without successor in P. P' is the undirected path that consists of two copies of P that are twisted before t, connected at their ends and have the second copy of s duplicated (cf. Fig. 5). If s precedes t in P (left side), then n is the center of P', but if t precedes s then n' is the center of P' (right side). □

Using the hardness of PathCenter, we now prove the hardness of the automorphism and isomorphism problems for k-paths.

Proposition 6.2. *For any fixed k, the automorphism problem for k-paths (and thereby k-trees) is* L*-hard.*

Proof. We reduce from PathCenter using the function $(P, c) \mapsto P'$ where the neighbors of c are n_1 and n_2, and where

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

We assume that c has distance more than k to both ends, as otherwise the problem is trivial. It is easy to see that P' is a k-path: The first k vertices on the path are the base. As the following vertices are added, the support is always a set of k consecutive nodes on the path, with the additional vertices c_i being added after c enters the support and before c leaves the support, respectively. It is obvious that $P' - \{c_1, c_2\}$ has the non-trivial automorphism that maps the ith vertex on the path to the $(n - i)$th one, but is otherwise rigid. This automorphism can be extended to all of P' by exchanging c_1 and c_2 if and only if c is the center of P. □

Proposition 6.3. *For any fixed k, the isomorphism problem for k-paths (and thereby k-trees) is* L*-hard.*

Proof. Again we reduce from PathCenter. Modifying the previous construction, we reduce $(P, c) \mapsto (P_1, P_2)$ where the neighbors of c are n_1 and n_2, the ends of P are v_1 and v_2, and where

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

Figure 5: The reduction of Ord to PathCenter

We assume that c has distance more than $k+1$ to both ends. If c is the center of P, the function that maps the ith vertex of P to the $(n-i)$th, exchanges c_1 and c_2 and maps e_1, e_2 to themselves is an isomorphism from P_1 to P_2. Conversely, if there is such an isomorphism then the e_i force the original path vertices to be mirrored and the c_i ensure that c is the center. □

Finally, we examine two problems related to the structure of k-trees. Let G be a graph. A vertex $v \in V(G)$ is called *simplicial* in G, if its neighborhood induces a clique. A bijective mapping $\sigma\colon \{1, \ldots, \|V(G)\|\} \to V(G)$ of the vertices of G is called *perfect elimination order (PEO)*, if for all i, $\sigma(i)$ is simplicial in $G - \{\sigma(1), \ldots, \sigma(i-1)\}$. Note that a graph can have several perfect elimination orders. It is well known that a graph has a PEO if and only if it is chordal. As k-trees are a subclass of chordal graphs, each k-tree has a PEO.

A related problem is the *fast reordering problem (FRP)* which is defined in [14] as a preprocessing step for parallel algorithms. It consists of finding a partition $R_0, \ldots, R_\ell$ of $V(G)$, such that R_ℓ is a clique and each R_i, $i < \ell$, is a maximal independent set of simplicial vertices of $G - \bigcup_{0 \le j < i} R_j$. For general chordal graphs there can be several such sequences, but for k-trees this sequence is unique and the remaining clique R_ℓ is the kernel. In [14] it was shown that if the input graphs are restricted to k-trees, the FRP can be solved in NC. We improve this and show logspace completeness for both problems:

Theorem 6.4. *For k-trees (k fixed), it is logspace complete to find a perfect elimination order and to solve the fast reordering problem.*

Proof. We first solve the fast reordering problem in logspace. Let G be a k-tree. We compute the level $l_G(v)$ for each $v \in V(G)$ (cf. Definition 3.6). By Lemma 4.5, this can be done in logspace. Let $l_{\max} = \max\{l_G(v) \mid v \in V(G)\}$. Output $R_i := \{v \in V(G) \mid l_G(v) = l_{\max} - i\}$ for $i = 0, \ldots, l_{\max}$. It follows from the structure of $T(G)$ that $R_0, \ldots, R_{l_{\max}}$ is a solution for FRP.

Next, we note that a perfect elimination order can be efficiently computed when a solution $R_0, \ldots, R_\ell$ to the FRP is known (i.e., finding a PEO reduces to solving the FRP): Take the members of the R_i in ascending order, first those from R_0, then those from R_1 and so on up to R_ℓ. By the definition of FRP, it follows that the result is a PEO, no matter which order is chosen within each R_i.

Finally we show that finding a perfect elimination order is hard for logspace even for paths. The result for k-trees (and k-paths) can be obtained using the construction of E_k given above. We solve an ORD instance (P, s, t) in DLOGTIME-uniform AC^0 with a single oracle gate for computing a PEO of the path P' given by

$$\begin{aligned} V(P') &= V(P) \cup \left\{ i' \;\middle|\; i \in V(P) \setminus \{n\} \right\} \\ E(P') &= \left\{ \{i, j\} \;\middle|\; (i,j) \in E(P) \right\} \\ &\quad \cup \left\{ \{i', j'\} \;\middle|\; (i,j) \in E(P), j \neq n \right\} \cup \left\{ \{i', n\} \;\middle|\; (i,n) \in E(P) \right\} \end{aligned}$$

where n is the vertex in P without successor. We claim that for any PEO σ of P' (where p_i is a shorthand for the position $\sigma^{-1}(i)$ of a vertex in σ):

$$(P, s, t) \in \text{ORD} \Leftrightarrow p_s \le p_t \le p_n \vee p_{s'} \le p_{t'} \le p_n$$

Figure 6: The reduction of ORD to finding a PEO

If $(P, s, t) \notin \textsc{Ord}$, then s is between t and n, and s' is between t' and n in P' (right side in Fig. 6). Thus σ cannot satisfy both $p_s \leq p_t$ and $p_{s'} \leq p_{t'}$. If $(P, s, t) \in \textsc{Ord}$ (left side of Fig. 6), n does not become simplicial until at least one copy of P is completely removed. Similarly, if the first copy is completely removed before n, t does not become simplicial before s is removed; and if the second copy is completely removed before n, t' does not become simplicial before s' is removed. □

7. Conclusions

We proved that the isomorphism and automorphism problems for k-trees are complete for logspace, and that canonical labelings of k-trees can be computed in logspace. Further, we described an $\mathcal{O}((k+1)!\,n)$ time implementation of our canonical labeling algorithm, yielding the fastest known fixed parameter tractable isomorphism algorithm for k-trees.

Our motivation for studying the isomorphism of k-trees stems from the fact that graphs of treewidth at most k coincide with partial k-trees, i. e. subgraphs of k-trees. Finding space efficient and parallel algorithms for this class is an active research area. We hope that our result can be generalized from k-trees to partial k-trees in the future. Das, Torán and Wagner already used some of our techniques to put isomorphism of tree distance width k graphs in L [11], a subclass of treewidth k graphs that is incomparable to k-trees. They also reduce isomorphism of decomposed bounded treewidth graphs to isomorphism of bounded tree distance width graphs, obtaining an LogCFL algorithm for isomorphism of bounded treewidth graphs [11] (previously, a TC^1 algorithm for this problem was given in [18]). The remaining obstacle to put this problem in L is thus the computation of compatible tree decompositions in logspace, compatible meaning that if two partial k-trees are isomorphic then there should be an isomorphism that maps one decomposition to the other. Elberfeld, Jakoby and Tantau showed how to compute a tree decomposition in logspace [15] (previously, LogCFL was known [35]), using and improving the mangrove technique we employed in our intermediate StUL result for k-tree isomorphism [5]. However, the constructed tree decomposition strongly depends on the names of the vertices, and computing compatible tree decompositions in logspace seems to require new ideas. Building upon techniques of [15], Wagner proved that partial k-trees can be canonized in AC^1 [34] (previously, a TC^2 algorithm was known [25]). It remains an intriguing open problem if this can be improved to logspace.

Our logspace algorithm for k-tree isomorphism takes $\mathcal{O}(k \log n)$ space. Can this be improved to $\mathcal{O}(f(k) + \log n)$? It can be argued that this is the right notion for "fixed parameter logspace", as it directly implies an $\mathcal{O}(f(k)n^{\mathcal{O}(1)})$ time bound.

Bodlaender shows that isomorphism of partial k-trees can be decided in $\mathcal{O}(n^{k+4.5})$ time [8]. Is this problem fixed parameter tractable?

Acknowledgment

We thank the anonymous referees for helpful comments and detailed suggestions that helped to improve this article.

References

[1] A. Aho, J. Hopcroft, J. Ullman, The design and analysis of computer algorithms, Addison-Wesley, 1974.

[2] E. Allender, M. Mahajan, The complexity of planarity testing, Information and Computation 139 (2004) 117–134.

[3] C. Àlvarez, B. Jenner, A very hard log-space counting class, Theoretical Computer Science 107 (1993) 3–30.

[4] S. Arnborg, A. Proskurowski, Linear time algorithms for NP-hard problems restricted to partial k-trees, Discrete Applied Mathematics 23 (1998) 11–24.

[5] V. Arvind, B. Das, J. Köbler, The space complexity of k-tree isomorphism, in: Algorithms and Computation. Proceedings of 18th ISAAC, number 4853 in LNCS, Springer, 2007, pp. 822–833.

[6] V. Arvind, B. Das, J. Köbler, A logspace algorithm for partial 2-tree canonization, in: Proceedings of the 3rd International Computer Science Symposium in Russia (CSR), number 5010 in LNCS, Springer, 2008, pp. 40–51.

[7] V. Arvind, P.P. Kurur, Graph isomorphism is in SPP, Information and Computation 204 (2006) 835–852.

[8] H.L. Bodlaender, Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees, Journal of Algorithms 11 (1990) 631–643.

[9] G. Buntrock, C. Damm, U. Hertrampf, C. Meinel, Structure and importance of logspace-MOD classes, Mathematical Systems Theory 25 (1992) 223–237.
[10] S.A. Cook, A taxonomy of problems with fast parallel algorithms, Information and Control 64 (1985) 2–22.
[11] B. Das, J. Torán, F. Wagner, Restricted space algorithms for isomorphism on bounded treewidth graphs, in: J.Y. Marion, T. Schwentick (Eds.), Proceedings of 27rd Annual Symposium on Theoretical Aspects of Computer Science (STACS), number 5 in LIPIcs, Leibniz-Zentrum für Informatik, Dagstuhl, 2010, pp. 227–238.
[12] S. Datta, N. Limaye, P. Nimbhorkar, T. Thierauf, F. Wagner, Planar graph isomorphism is in log-space, in: Proceedings of 24th Annual IEEE Conference on Computational Complexity (CCC), IEEE Computer Society, 2009, pp. 203–214.
[13] S. Datta, P. Nimbhorkar, T. Thierauf, F. Wagner, Graph isomorphism for $K_{3,3}$-free and K_5-free graphs is in log-space, in: R. Kannan, K.N. Kumar (Eds.), Proceedings of Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), number 4 in LIPIcs, Leibniz-Zentrum für Informatik, Dagstuhl, 2009, pp. 145–156.
[14] J.G. Del Greco, C.N. Sekharan, R. Sridhar, Fast parallel reordering and isomorphism testing of k-trees, Algorithmica 32 (2002) 61–72.
[15] M. Elberfeld, A. Jakoby, T. Tantau, Logspace versions of the theorems of Bodlaender and Courcelle, in: Proceedings of 51st IEEE Symposium on Foundations of Computer Science (FOCS 2010), pp. 143–152.
[16] K. Etessami, Counting quantifiers, successor relations, and logarithmic space, Journal of Computer and System Sciences 54 (1997) 400–411.
[17] S. Goldwasser, M. Sipser, Private coins versus public coins in interactive proof systems, in: Randomness and Computation, volume 5 of *Advances in Computing Research*, JAI Press, 1989, pp. 73–90.
[18] M. Grohe, O. Verbitsky, Testing graph isomorphism in parallel by playing a game, in: Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Proceedings, Part I, number 4051 in LNCS, Springer, 2006, pp. 3–14.
[19] C. Hoffmann, Group-Theoretic Algorithms and Graph Isomorphism, number 136 in LNCS, Springer, 1982.
[20] B. Jenner, J. Köbler, P. McKenzie, J. Torán, Completeness results for graph isomorphism, Journal of Computer and System Sciences 66 (2003) 549–566.
[21] M.M. Klawe, D.G. Corneil, A. Proskurowski, Isomorphism testing in hookup classes, SIAM Journal on Algebraic and Discrete Methods 3 (1982) 260–274.
[22] T. Kloks, Treewidth, number 842 in LNCS, Springer, 1994.
[23] J. Köbler, S. Kuhnert, The isomorphism problem for k-trees is complete for logspace, in: Proceedings of 34th International Symposium Mathematical Foundations of Computer Science (MFCS), number 5734 in LNCS, Springer, 2009, pp. 537–548.
[24] J. Köbler, U. Schöning, J. Torán, The Graph Isomorphism Problem: Its Structural Complexity, Progress in Theoretical Computer Science, Birkhäuser, Boston and others, 1993.
[25] J. Köbler, O. Verbitsky, From invariants to canonization in parallel, in: Proceedings of the 3rd International Computer Science Symposium in Russia (CSR), number 5010 in LNCS, Springer, 2008, pp. 216–227.
[26] S. Lindell, A logspace algorithm for tree canonization. extended abstract, in: Proceedings of the 24th Annual ACM Symposium on Theory of Computing, ACM, New York, 1992, pp. 400–404.
[27] G. Miller, J. Reif, Parallel tree contraction part 2: further applications, SIAM Journal on Computing 20 (1991) 1128–1147.
[28] T. Nagoya, Counting graph isomorphisms among chordal graphs with restricted clique number, in: Proceedings of 12th International Symposium on Algorithms and Computation (ISAAC), number 2223 in LNCS, Springer, 2001, pp. 136–147.
[29] O. Reingold, Undirected connectivity in log-space, Journal of the ACM 55 (2008) 17:1–17:24.
[30] D.J. Rose, R.E. Tarjan, G.S. Lueker, Algorithmic aspects of vertex elimination on graphs, SIAM J. Comput. 5 (1976) 266–283.
[31] U. Schöning, Graph isomorphism is in the low hierarchy, Journal of Computer and System Sciences 37 (1988) 312–323.
[32] S. Toda, Computing automorphism groups of chordal graphs whose simplicial components are of small size, IEICE Transactions on Information and Systems E89-D (2006) 2388–2401.
[33] J. Torán, On the hardness of graph isomorphism, SIAM J. Comput. 33 (2004) 1093–1108.
[34] F. Wagner, Graphs of bounded treewidth can be canonized in AC^1, in: A. Kulikov, N. Vereshchagin (Eds.), Proceedings of 6th International Computer Science Symposium in Russia (CSR), number 6651 in LNCS, Springer, 2011, pp. 209–222.
[35] E. Wanke, Bounded tree-width and LOGCFL, Journal of Algorithms 16 (1994) 470–491.
[36] K. Yamazaki, H.L. Bodlaender, B. de Fluiter, D.M. Thilikos, Isomorphism for graphs of bounded distance width, Algorithmica 24 (1999) 105–127.