Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 346 (2019) 99–110

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Recognizing Graph Search Trees[1,2]

Jesse Beisegel[a], Carolin Denkert[a], Ekkehard Köhler[a], Matjaž Krnc[b,c,3], Nevena Pivač[b,4], Robert Scheffler[a] and Martin Strehler[a]

[a] *Brandenburg University of Technology, Cottbus, Germany*

[b] *University of Primorska, Koper, Slovenia*

[c] *Faculty of Information Studies, Novo mesto, Slovenia*

Abstract

Graph searches and the corresponding search trees can exhibit important structural properties and are used in various graph algorithms. The problem of deciding whether a given spanning tree of a graph is a search tree of a particular search on this graph was introduced by Hagerup and Nowak in 1985, and independently by Korach and Ostfeld in 1989 where the authors showed that this problem is efficiently solvable for DFS trees. A linear time algorithm for BFS trees was obtained by Manber in 1990. In this paper we prove that the search tree problem is also in $\mathcal{P}$ for LDFS, in contrast to LBFS, MCS, and MNS, where we show $\mathcal{NP}$-completeness. We complement our results by providing linear time algorithms for these searches on split graphs.

Keywords: Search tree recognition, LBFS, LDFS, MNS, MCS

1 Introduction

Motivation. Graph searches like *Breadth First Search* (BFS) and *Depth First Search* (DFS) are, in the most general sense, mechanisms for systematically visiting all vertices of a graph. Considered as some of the most basic algorithms in computer science, graph searches are taught in many undergraduate courses around the

[1] A full version of this paper can be found in [3]. The proofs of statements marked with ($\star$) can be found there.

[2] The work of this paper was done in the framework of a bilateral project between Brandenburg University of Technology and University of Primorska, financed by German Academic Exchange Service and the Slovenian Research Agency (BI-DE/17-19-18).

[3] The author gratefully acknowledges the European Commission for funding the InnoRenew CoE project (Grant Agreement #739574) under the Horizon2020 Widespread-Teaming program and the Republic of Slovenia (Investment funding of the Republic of Slovenia and the European Union of the European Regional Development Fund).

[4] Funded in part by the Slovenian Research Agency (research programs P1-0285 and J1-9110 and Young Researchers Grant).

https://doi.org/10.1016/j.entcs.2019.08.010

world and represent an elementary component of several graph algorithms, such as finding connected components, testing for bipartiteness, computing shortest paths with respect to the number of edges, or the Edmonds-Karp algorithm for computing the maximum flow in a network [12]. Similarly, DFS is the basis for algorithms for finding biconnected components in undirected graphs [17], strongly connected components in directed graphs [23], topological orderings of directed acyclic graphs [24], planarity testing [18], or solving mazes [13].

We focus on connected searches, that is, a graph search or graph traversal that starts at a vertex and explores the graph by visiting a vertex in the neighborhood of the already visited vertices. If no further restriction is given, we call such a search a *generic search*. The search paradigms of BFS and DFS can be simply characterized by using a queue or a stack as the data structure for the unvisited vertices in the current neighborhood. However, there are more sophisticated searches like *Lexicographic Breadth First Search* (LBFS) [21] and *Lexicographic Depth First Search* (LDFS) [7]. In this article, we also consider *Maximum Cardinality Search* (MCS) [25] and *Maximum Neighborhood Search* (MNS) [7]. A short overview of these searches can be found in the full version of this paper [3].

Usually, the outcome of a graph search is a *search order*, i.e., a sequence of the vertices in the order they are visited. There are many known results and algorithms that are based on graph search orders. For instance, a perfect elimination order of a chordal graph can be found by reversing an LBFS order on that graph [21]. Apart from a linear recognition algorithm for chordal graphs, LBFS also yields a greedy coloring algorithm for finding a minimum coloring for this graph class [14]. Furthermore, it is possible to generate characterizing vertex orderings for AT-free graphs using BFS [1].

A structure that is closely related to a graph search is the corresponding search tree. Such trees can be of particular interest, as for instance the tree obtained by a BFS contains the shortest paths from the root r to all other vertices in the graph. The trees generated by DFS can be used for fast planarity testing of graphs [18]. Moreover, if a cocomparability graph has a hamiltonian path, then such a path can be found by a combination of various graph searches [5]. First, one can use at most n LBFS runs, where n is the number of vertices, to find a cocomparability ordering [11]. Afterwards, the last visited vertex of an LDFS on this cocomparability ordering is the first vertex of a hamiltonian path. Finally, the search tree of a right most neighbor search on the LDFS ordering is a hamiltonian path.

So far, there is no satisfactory answer as to why graph searching works so well. An interesting example are multi-sweep algorithms, such as finding dominating pairs in connected asteroidal triple-free graphs [8]. One can prove that these algorithms are correct. However, it is not clear why multiple runs of a simple algorithm could give such a strong insight into graph structure. Indeed, there seem to be some hidden structural properties of graph searches, which are waiting for discovery and algorithmic exploitation.

As a step in this direction, we study the problem of whether a given tree can be a search tree of a particular search. For BFS-like searches, one usually connects

each vertex $v \in V$ to its neighbor which appeared first in the BFS order. Contrary, for DFS-like searches, one connects each vertex $v \in V$ to the last neighbor visited before v. However, there is no such obvious definition of a tree for MCS or MNS. Therefore, we define $\mathcal{F}$- and $\mathcal{L}$-trees: Given an ordering, in an $\mathcal{F}$-tree each vertex v is connected to its neighbor which appeared first in the ordering before v, whereas in an $\mathcal{L}$-tree each vertex is connected to its neighbor which appeared last before v.

Related work. Already in 1972, Tarjan [23] gave a complete characterization of DFS trees as so-called palm trees. However, no algorithm that determines if a given spanning tree of a graph G is a DFS tree of G was specified in that work. Using the concept of palm trees, Hopcroft and Tarjan developed a linear time algorithm for testing planarity of a graph [18]. Exploiting properties of DFS and BFS trees, the problem of checking whether a given spanning tree of G can be obtained by a DFS on G was formulated by Hagerup and Novak [16]. A few years later, Korach and Ostfeld gave a linear time algorithm for the proposed problem of recognition of DFS-trees [19]. A similar result for the recognition of BFS-trees was given by Manber in 1990 [20].

A problem that is closely related to the search tree recognition problem is the so-called end-vertex problem, i.e. the problem of determining whether a given vertex v in a graph G can be visited last by some graph search method. As a result of numerous new applications in algorithms, the end-vertex problem has received some attention in recent literature. In particular, the end-vertex of an LBFS on a chordal graph is always simplicial [21]. Furthermore, in a cocomparability graph, the end-vertex of an LBFS is a source/sink in some transitive orientation of its complement [15]. End-vertices are of particular interest for multi-sweep algorithms, as every consecutive search starts at the end vertex of the previous search. For example, one can use five LBFS executions followed by a modified LBFS to recognize interval graphs [9]. Crescenzi et al. [10] have shown that the diameter of huge real world graphs can usually be found with only a few BFS executions.

Surprisingly, the problem of deciding whether a vertex can be an end-vertex of a graph search is hard. In 2010, Corneil, Köhler, and Lanlignel [6] showed that it is $\mathcal{NP}$-hard to decide whether a vertex can be the end vertex of an LBFS. Later, Charbit, Habib, and Mamcarz generalized this result to BFS, DFS, and LDFS [4]. Furthermore, they extended these results to several graph classes. Recently, Beisegel et al. [2] proved $\mathcal{NP}$-hardness results for MCS and MNS, and they also provided linear time algorithms for this problem on split graphs and unit interval graphs.

Our contribution. Although research initially began with the recognition of search trees, the results on the end-vertex problem are currently more extensive. In the light of the new results, we fill in the gaps in the analysis of the complexity of the search tree recognition problem. In this paper, we extend the tree recognition problem to LBFS, LDFS, MCS, and MNS for $\mathcal{F}$- or $\mathcal{L}$-trees, respectively, by showing $\mathcal{NP}$-hardness results for most of these searches on general graphs, a polynomial time recognition algorithm for $\mathcal{L}$-trees of LDFS on general graphs, and linear time algorithms for the $\mathcal{F}$-tree and the $\mathcal{L}$-tree problem on split graphs for various searches.

2 Preliminaries

All graphs considered in this paper are finite, undirected, simple and connected. Given a graph $G = (V, E)$, we denote by n and m the number of vertices and edges in G, respectively. For a vertex $v \in V$, we denote by $N(v)$ the *neighborhood* of v, i.e., the set $N(v) = \{u \in V \mid uv \in E\}$, where an edge between u and v in G is denoted by uv. A *clique* in a graph G is a set of pairwise adjacent vertices and an *independent set* in G is a set of pairwise nonadjacent vertices. If the neighborhood of a vertex v in G is a clique, then v is said to be a *simplicial vertex*. The *complement* of the graph G is the simple graph $\overline{G}$ having the same set of vertices as G where for $x, y \in V$, we have that xy is an edge of $\overline{G}$ if and only if it is not an edge in G. For a graph $G = (V, E)$ and an edge $e = uv$, where u and v are nonadjacent vertices in G, we define $G + e$ to be a graph with vertex set V and edge set $E \cup \{e\}$. Given a subset S of vertices in G, we denote by $G[S]$ the *subgraph of G induced by S*, where $V(G[S]) = S$ and $E(G[S]) = \{xy \in E(G) \mid x \in S, y \in S\}$. By $G - S$ we denote the graph induced by $V(G) \setminus S$. If S contains just one element v, we will simply write $G - v$ to denote the graph induced by $V(G) \setminus \{v\}$.

A graph G that contains no induced cycle of length larger than 3 is called *chordal*. If neither G nor its complement contains an induced cycle of length 5 or more, then G is said to be *weakly chordal*. A *split graph* G is a graph whose vertex set can be divided into sets C and I such that C is a clique in G and I is an independent set in G. It is easy to see, that every split graph is chordal, whereas every chordal graph is also weakly chordal.

An ordering of vertices in G is a bijection $\sigma : V(G) \to \{1, 2, \ldots, n\}$. For an arbitrary ordering σ of vertices in G, we denote by $\sigma(v)$ the position of vertex $v \in V(G)$. Given two vertices u and v in G we say that u is *to the left* (resp. *to the right*) of v if $\sigma(u) < \sigma(v)$ (resp. $\sigma(u) > \sigma(v)$) and we denote this by $u \prec_\sigma v$ (resp. $u \succ_\sigma v$).

A *tree* is an acyclic connected graph. A *spanning tree* of a graph G is an acyclic connected subgraph of G which contains all vertices of G. A tree together with a distinguished *root vertex* r is said to be *rooted*. In such a rooted tree a vertex v is an *ancestor* of vertex w if v is an element of the unique path from w to the root r. In particular, if v is adjacent to w, it is called the *parent* of w. Furthermore, a vertex w is called the *descendant (child)* of v if v is the ancestor (parent) of w. A tree is a caterpillar tree, if and only if it admits a dominating path P, i.e., every vertex is either in P or adjacent to a vertex in P.

The definition of the term *search tree* varies between different paradigms. However, typically, it consists of the vertices of the graph and, given the search order $(v_1, \ldots, v_n)$, for each vertex v_i exactly one edge to a $v_j \in N(v_i)$ with $j < i$. By specifying to which of the previously visited neighbors a new vertex is adjacent in the tree, we can define different types of graph search trees. For example, in a BFS a vertex is typically adjacent to the leftmost neighbor in the search order, while in DFS a vertex v is adjacent to the rightmost neighbor to the left of v. This motivates the following definition.

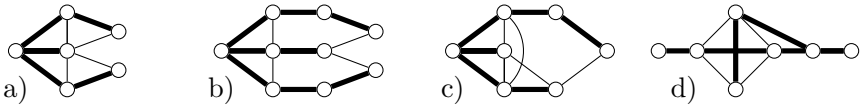

Fig. 1. Four examples of graphs with their search trees denoted by the thick edges. The graph in a) depicts a search tree of BFS that is not an $\mathcal{F}$-tree for LBFS or MNS. The graph in b) depicts an $\mathcal{F}$-tree of MNS and BFS that is not an $\mathcal{F}$-tree for LBFS. The graph in c) shows a search tree that is an $\mathcal{F}$-tree of MNS, BFS and LBFS that is not an $\mathcal{F}$-tree of MCS. Finally, the graph in d) gives an example of a search tree that is an $\mathcal{L}$-tree for DFS, but not for LDFS.

Definition 2.1 Given a search discovery order $\sigma := (v_1, \ldots, v_n)$ of a given search on a connected graph $G = (V, E)$, we define the *first-in tree* (or $\mathcal{F}$-tree) to be the tree consisting of the vertex set V and an edge from each vertex to its leftmost neighbor in σ.

The *last-in tree* (or $\mathcal{L}$-tree) is the tree consisting of the vertex set V and an edge from each vertex v_i to its rightmost neighbor v_j in σ with $j < i$.

As explained above, if σ and T are the output of a classical BFS, then T is an $\mathcal{F}$-tree with respect to σ, while for a classical DFS the tree T is an $\mathcal{L}$-tree with respect to σ. Given this definition, we can state the following decision problem.

$\mathcal{F}$-Tree ($\mathcal{L}$-Tree) Recognition Problem

Instance: A connected graph $G = (V, E)$ and a spanning tree T.

Task: Decide whether there is a graph search of a given type such that T is an $\mathcal{F}$-tree ($\mathcal{L}$-tree) of G.

When comparing the different searches, one can see that graph search trees behave very similarly to the searches themselves, in the sense that, for example, an LBFS tree is also a BFS tree, but not vice versa. Some examples of graph search trees illustrating these relationships can be found in Fig. 1.

3 $\mathcal{NP}$-Completeness for LBFS, MNS and MCS

It was shown in [6] that the LBFS end-vertex problem is $\mathcal{NP}$-complete. In the following we show that the same holds for the tree-recognition problem.

Theorem 3.1 *The $\mathcal{F}$-tree-recognition problem of LBFS is $\mathcal{NP}$-complete on weakly chordal graphs.*

We prove Theorem 3.1 by giving a reduction from 3-SAT. Let $\mathcal{I}$ be an instance of 3-SAT. We construct the corresponding graph $G(\mathcal{I})$ and the spanning tree $T(\mathcal{I})$ as follows (for an example see Fig. 2): Let $X = \{x_1, \ldots, x_k, \overline{x_1}, \ldots, \overline{x_k}\}$ be the set of vertices representing the literals of $\mathcal{I}$. The edge set $E(X)$ forms the complement of the matching in which x_i is matched to $\overline{x_i}$ for every $i \in \{1, \ldots, k\}$. For each clause C_i of $\mathcal{I}$ we have a triangle consisting of vertices a_i, c_i and t_i. For every triangle representing a clause C_i, the vertex c_i is adjacent to each literal of the clause C_i.

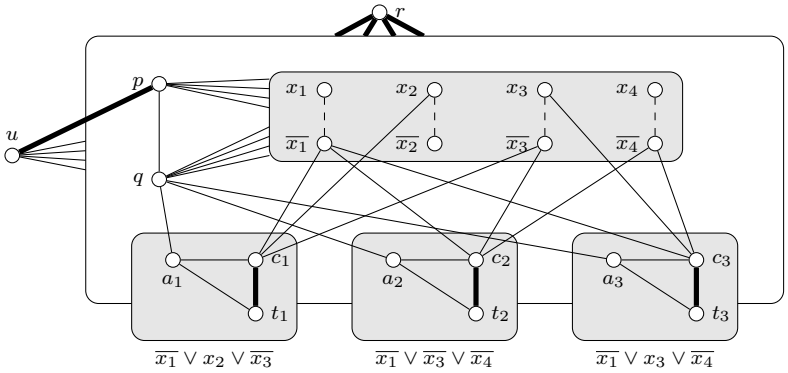

Fig. 2. The $\mathcal{NP}$-completeness construction for the tree-recognition problem of LBFS. The depicted graph is $G(\mathcal{I})$ for $\mathcal{I} = (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_4})$. In the box containing the literal vertices non-edges are displayed by dashed lines. A vertex is connected with a box if it is adjacent to all vertices in this box. Tree edges are depicted by thick edges.

In addition, we have vertices r, p, q and u. Vertex r is adjacent to every vertex apart from the t_i and u, while u is adjacent to all vertices apart from the t_i and r. Vertex p has additional edges to each vertex in X and to q, while q is also adjacent to all vertices in X and each of the a_i. Altogether, $G(\mathcal{I})$ consists of the vertex set $V(G(\mathcal{I})) := X \cup \{r, p, q, u\} \cup C_1 \cup \ldots \cup C_l$, where C_i represents the vertices of the clause-gadget of C_i and the edge set is defined as above.

The corresponding spanning tree $T(\mathcal{I})$ consists of the edges incident to r, an edge between u and p and the edges $c_i t_i$ for all $i \in \{1, \ldots l\}$; they are denoted as thick lines in Fig. 2.

We proceed to prove Theorem 3.1 by showing that $T(\mathcal{I})$ is an $\mathcal{F}$-tree of LBFS of $G(\mathcal{I})$ if and only if $\mathcal{I}$ has a satisfying assignment $\mathcal{A}$.

Lemma 3.2 *If $\mathcal{I}$ admits a satisfying assignment $\mathcal{A}$, then $T(\mathcal{I})$ is a possible $\mathcal{F}$-tree of LBFS on $G(\mathcal{I})$.*

Proof. Let $\mathcal{A}$ be a satisfying assignment of $\mathcal{I}$. The following valid search order produces $T(\mathcal{I})$ as its search tree: We begin in r and then choose p. Next, we can choose vertices from X according to the assignment $\mathcal{A}$ in an arbitrary order, i.e., we choose x_i or $\overline{x_i}$ corresponding to whether the variable x_i is set to 1 or 0 in $\mathcal{A}$. We are then forced to visit the vertex q, as each remaining vertex of X is not adjacent to one of the visited vertices of X. After choosing the remaining vertices of X we proceed to the vertices of the clause gadgets: As a fulfilling assignment sets at least one literal to 1 in each clause, every c_i has a neighbor that appears earlier in the search order than q which is the leftmost neighbor of a_i in the search order. Hence, for each clause gadget C_i we must choose c_i before a_i. Therefore, we can choose all vertices c_i and then all vertices a_i. Finally, we can choose u and then all the t_i.

It is easy to see that all edges incident to r belong to the search tree of the constructed order, as well as pu. On the other hand, $c_i t_i$ must be in the search tree for every $i \in \{1, \ldots, l\}$, as c_i was always chosen before a_i. Therefore, the search tree of the constructed order coincides with $T(\mathcal{I})$. □

We now show the other direction of the proof.

Lemma 3.3 *If $\mathcal{I}$ does not admit a satisfying assignment, then $T(\mathcal{I})$ cannot be an $\mathcal{F}$-tree of LBFS on $G(\mathcal{I})$.*

Proof. We show that for at least one clause gadget C_i the vertex a_i is visited before c_i, thus making $T(\mathcal{I})$ an infeasible search tree.

To prove this, we analyze the order in which the vertices of X are visited in any feasible LBFS search. It is easy to see that any LBFS must begin in r, as r is the only vertex whose incident edges are all tree edges. Next, we are forced to choose p, as otherwise pu cannot be a tree edge. If q is chosen next, then, as a result, a_i must be visited before c_i for every $i \in \{1, \ldots, l\}$ and $T(\mathcal{I})$ cannot be the resulting search tree. Therefore, a subset of the vertices of X must be chosen before the vertex q.

If a vertex x_i is visited, then q receives a larger label than $\overline{x_i}$, as they otherwise share the same set of neighbors among the visited vertices up to that point (and analogously if $\overline{x_i}$ is visited before q). Thus, q must be chosen between any literal vertex and its negation. The largest subset of X that can be visited before q must, therefore, be an assignment of $\mathcal{I}$. As $\mathcal{I}$ is not satisfiable, any such assignment must leave at least one clause unfulfilled. If C_i is such a clause, then at the point at which q is chosen, c_i does not contain any neighbors among the visited literal vertices. As a result, a_i receives a larger label than c_i and is visited earlier.

Consequently, in any LBFS there must be a clause C_i such that a_i is visited before c_i and $c_i t_i$ cannot be in the search tree. This shows that $T(\mathcal{I})$ cannot be a $\mathcal{F}$-tree of an LBFS. □

To conclude the proof of Theorem 3.1 it remains to show that $G(\mathcal{I})$ is weakly chordal for every 3-SAT instance $\mathcal{I}$.

Lemma 3.4 (⋆) *For each instance $\mathcal{I}$ of 3-SAT, the graph $G(\mathcal{I})$ is weakly chordal.*

As we have done for LBFS, we will show that the $\mathcal{F}$-tree problems for MNS and MCS are $\mathcal{NP}$-complete.

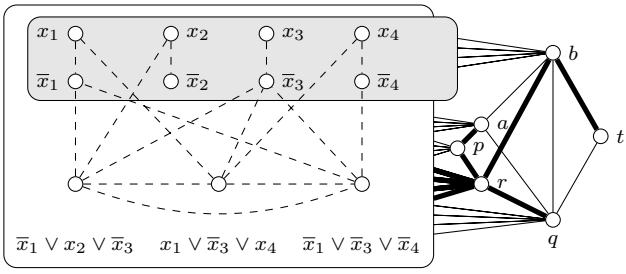

Fig. 3. The $\mathcal{NP}$-completeness construction for the tree-recognition problem of MNS. The depicted graph is $G(\mathcal{I})$ for $\mathcal{I} = (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$. In both boxes non-edges are displayed by dashed lines. A vertex is connected to a box if it is adjacent to all vertices in this box. Tree edges are depicted by thick edges.

Theorem 3.5 (⋆) *The $\mathcal{F}$-tree-recognition problem of MNS and MCS is $\mathcal{NP}$-complete on weakly chordal graphs.*

For the proof we construct a polynomial reduction from 3-SAT. Let $\mathcal{I}$ be an instance of 3-SAT. We construct the corresponding graph $G(\mathcal{I})$ as follows (see Fig. 3

for an example): Let $X = \{x_1, \ldots, x_k, \overline{x}_1, \ldots, \overline{x}_k\}$ be the set of vertices representing the literals of $\mathcal{I}$. The edge-set $E(X)$ forms the complement of the matching in which x_i is matched to $\overline{x}_i$ for every $i \in \{1, \ldots, k\}$. Let $C = \{c_1, \ldots, c_l\}$ be the set of vertices representing the clauses of $\mathcal{I}$. The set C is independent in $G(\mathcal{I})$ and every c_i is adjacent to each vertex of X, except those representing the literals of the clause associated with c_i for every $i \in \{1, \ldots, l\}$. Additionally, we add the vertices r, p, q, a, b and t. The vertices r, p, q and a are adjacent to all literal vertices and all clause vertices and b is adjacent to all literal vertices. Finally, we add the edges ab, ap, aq, bq, br, bt, pr, qr and qt. The spanning tree $T(\mathcal{I})$ of $G(\mathcal{I})$ consists of all edges incident to r and the edges pa and bt.

The idea of the proof is similar to the $\mathcal{NP}$-completeness proof of LBFS. If there is satisfying assignment we can take the vertex b before all clause vertices and before q. If no such assignment exists, we have to take one clause vertex before b and, thus, also q before b, inserting edge qt to the search tree.

4 Polynomial Results

4.1 Lexicographic Depth First Search

As Lexicographic Depth First Search is a special case of DFS, the most natural search tree to be considered here is the $\mathcal{L}$-tree. We give a polynomial-time algorithm (Algorithm 1) which, given a graph G and its spanning tree T, decides whether T is an $\mathcal{L}$-tree of LDFS on G. This is an interesting contrast to the fact that it is $\mathcal{NP}$-complete to decide whether a given vertex is an end-vertex of LDFS, as shown by Charbit et al. [4].

In essence, Algorithm 1 runs an LDFS and at every step checks whether there is still a possible choice of vertex which does not contradict the search tree.

Input: Graph $G = (V, E)$, spanning tree T of G, and a vertex $r \in V$.
Output: T is an $\mathcal{L}$-tree of LDFS on G rooted in r or not.
begin
  $S \leftarrow \{r\}$;
  foreach $v \in V - r$ **do** label$(v) \leftarrow \emptyset$;
  foreach $v \in N(r)$ **do**
    prepend 0 to label(v);
    pred$(v) \leftarrow r$;
  while $S \neq V$ **do**
    choose a node $v \in V - S$ with lexicographic largest label, such that $\{\text{pred}(v), v\} \in E(T)$;
    if *no such v exists* **then return** *T is not an $\mathcal{L}$-tree of LDFS on G rooted in r*;
    $S \leftarrow S \cup \{v\}$;
    foreach $w \in N(v) \setminus S$ **do**
      prepend i to label(w);
      pred$(w) \leftarrow v$;
  return *T is an $\mathcal{L}$-tree of LDFS on G rooted in r.*

Algorithm 1: Algorithm which decides whether T is an $\mathcal{L}$-tree of LDFS on G rooted in r.

To prove that Algorithm 1 works correctly, we first state a few lemmas about $\mathcal{L}$-trees of DFS.

Lemma 4.1 *[23] Let $G = (V, E)$ be a graph and let T be an $\mathcal{L}$-tree of G generated*

by DFS. For each edge $uv \in E$ it holds that either $e \in E(T)$ or u is an ancestor of v in T or v is an ancestor of u in T.

Lemma 4.2 ($\star$) *Let $G = (V, E)$ be a graph with spanning tree T. Let G_i be a connected induced subgraph of G with a spanning tree T_i which is the restriction of T to G_i. If T is an $\mathcal{L}$-tree of LDFS on G, then T_i is an $\mathcal{L}$-tree of LDFS on G_i. In particular, if T is rooted in $r \in V$ and $r \in V(T_i)$, then T_i is also rooted in r.*

Theorem 4.3 *The $\mathcal{L}$-tree recognition problem for LDFS can be solved in polynomial time.*

Proof. Algorithm 1 tests for a fixed $r \in V$ whether T can be an $\mathcal{L}$-tree for LDFS on G that is rooted in r. Therefore, assuming the Algorithm 1 works correctly and in polynomial time, it is enough to apply it to all vertices in G to decide whether T is, in fact, an $\mathcal{L}$-tree of LDFS. As we begin the search in r we from now on assume that T is rooted in a fixed vertex r.

First suppose that the algorithm returns "T is an $\mathcal{L}$-tree of LDFS on G". In this case, the algorithm has successfully executed an LDFS and it remains to show that the resulting search order has T as its $\mathcal{L}$-tree. This, however, is safeguarded by the fact that at every point at which we have added a vertex v to our search order, the predecessor of v, i.e., its parent in the resulting search tree, is also adjacent to v in T.

Now assume that the algorithm returns "T is not an $\mathcal{L}$-tree of LDFS on G". This implies that at some point of Algorithm 1 there is no vertex x of lexicographically largest label, such that the predecessor of x is adjacent to x in T. Let v be such a vertex of lexicographically largest label, whose predecessor is not its parent in T. As v is the first such vertex to appear in the search, the tree R constructed so far by Algorithm 1 is a subtree of T.

Assume that T is, in fact, an $\mathcal{L}$-tree of G generated by LDFS. Let u be the predecessor assigned to v by the algorithm. Thus, due to Lemma 4.1, u must be an ancestor of v in T. Let w be the unique child of u in T that is also an ancestor of v and let P be the unique path from v to r in T; in particular, $u, w \in V(P)$. As a result of Lemma 4.2, P is an $\mathcal{L}$-tree of LDFS on $G[V(P)]$ since T is an $\mathcal{L}$-tree of LDFS on G.

However, Algorithm 1 and Lemma 4.1 imply that P cannot be an $\mathcal{L}$-tree of LDFS on $G[V(P)]$: As we start in r and as P is a path, we must choose all vertices up to u in the order of the path. Due to Lemma 4.1, the vertices have the same labels as they did when Algorithm 1 halted. Therefore, v has a lexicographically larger label than w. As a result, P and, thus, T cannot be a $\mathcal{L}$-trees of LDFS. □

4.2 Linear Time Algorithms for Split Graphs

Surprisingly, for split graphs the set of $\mathcal{F}$-trees is the same for the searches BFS, MNS, MCS, and LDFS, even though this does not hold for the respective search orders. We exploit this special structure to derive a linear time algorithm for split graphs. Note that LDFS is considered together with an $\mathcal{F}$-tree.

Theorem 4.4 *A tree T is an $\mathcal{F}$-tree of BFS on a split graph G if and only if it is an $\mathcal{F}$-tree of MNS (MCS, LBFS, LDFS).*

Proof. Let $G = (V, E)$ be a split graph and let T be an $\mathcal{F}$-tree for BFS on G, generated by the order τ. Let $I = \{i_1, \ldots, i_\ell\}$ be the independent set and $C = \{c_1, \ldots, c_k\}$ be the clique of G. We show that there is an MNS ordering σ that generates a search tree that coincides with T.

Suppose τ starts with a clique vertex, without loss of generality c_1, that is, c_1 is the root of the search tree. Then, all other clique vertices c_2 to c_k are in the first layer of the $\mathcal{F}$-tree, and additionally, all independent set vertices which are adjacent to c_1 are in the first layer as well. Without loss of generality, i_1 to i_q are adjacent to c_1. Then i_{q+1} to i_ℓ are in the second layer of the tree T. Furthermore, suppose c_2 to c_k are indexed in the order of occurrence in the BFS order. Note that BFS may choose i_1 to i_q in arbitrary order before the last clique vertex is chosen.

Now, we construct an MNS order σ, such that the $\mathcal{F}$-tree of σ is T. We simply pick c_1 to c_k in ascending order, that is, we start with the same root c_1, followed by the clique vertices in unchanged order. Since all vertices in the clique have the same neighborhood of visited vertices at every step and none of the i_x has a larger neighborhood, this does not contradict the MNS search paradigm. Finally, we add the independent set vertices to σ. Here, we have to choose the independent vertices with larger neighborhoods first. As the whole neighborhood of each of these vertices is already chosen, this does not change the edges of the tree, i.e., the first visited neighbor. Since the neighbors of the independent set vertices are visited in the same order as in the BFS, the same $\mathcal{F}$-tree T is generated.

Now suppose that τ starts with an independent vertex and, without loss of generality, we label the root of the search tree T by i_1. Then the neighbors of i_1, say c_1 to c_q are in the first layer of the search tree. All other clique vertices and all independent set vertices which are neighbors of c_1 to c_q are in the second layer of the $\mathcal{F}$-tree T. Finally, all remaining independent set vertices are in third layer. Again note that c_1 to c_k are assumed to be indexed in the order of occurrence in the BFS order.

Again, a similar order σ, now starting with i_1, followed by c_1 to c_k in order of the indices, and afterwards followed by i_2 to i_ℓ, respecting neighborhood inclusions, yields the same tree T and it is an MNS order analogous to the above argumentation.

The proof for the other direction can be achieved in the same way. The proofs for MCS, LBFS, and LDFS also follow the same pattern. □

As the $\mathcal{F}$-tree problem can be solved in linear time for BFS [20], this also holds for the other searches.

Corollary 4.5 *The $\mathcal{F}$-tree problem of MNS, MCS, LBFS and LDFS can be solved in linear time on split graphs.*

The following theorem fully characterizes the structure of $\mathcal{L}$-trees on split graphs generated by MNS-type searches.

Theorem 4.6 ($\star$) *A tree T is an $\mathcal{L}$-tree of MNS (MCS, LDFS, LBFS) on a split graph $G = (V, E)$ with clique C and independent set I if and only if:*

(i) *T is a caterpillar tree consisting of a set of leaves L and a dominating path $P = (v_1, \ldots, v_k)$ which contains every vertex of C.*

(ii) *It holds for every leaf $w \in L$ with a neighbor v_i in T that $wv_j \notin E(G)$ for $j > i$.*

(iii) *It holds for every $v_i \in I$ that:*
   (a) *$\{v_1, \ldots, v_{i-1}\} \cap C \subseteq N(v_i)$ with $|\{v_1, \ldots, v_{i-1}\} \cap C| = l$*
   (b) *$v_{i+1}, \ldots, v_{deg(v_i)-l} \subseteq N(v_i)$*

These three conditions can be checked in linear time [3].

Corollary 4.7 *The $\mathcal{L}$-tree problem of MNS, MCS, LBFS and LDFS can be solved in linear time on split graphs.*

5 Conclusion

We have shown that the $\mathcal{F}$-tree problem is $\mathcal{NP}$-complete for LBFS, MCS and MNS. Furthermore, we have given polynomial time algorithms for the $\mathcal{L}$-tree problem of LDFS and for both the $\mathcal{F}$-tree and the $\mathcal{L}$-tree problems of LBFS, LDFS, MCS and MNS on split graphs. To the best of our knowledge, no hardness results for the $\mathcal{L}$-tree problem were known before. Thus, the question arises whether the $\mathcal{L}$-tree recognition problem is easy in general for every graph search.

For the end-vertex problem, there are polynomial algorithms for some chordal graph classes besides split graphs (cf. [2,4,6]). Can these results be transferred to the tree-recognition problem? Up to now, there is no known combination of graph class and search for which the end-vertex problem is easy but the tree-recognition problem is hard.

Moreover, we have considered the search tree recognition problem for labeled, unrooted trees in this paper. As a variant of this problem, one could fix the starting vertex of the search, i.e., the input would be a rooted search tree. As we have already seen in Section 4, if we can solve the problem with fixed start vertex in polynomial time, we can also solve the general problem efficiently by solving it for every vertex as the starting point of the search. Nevertheless, it could be possible that the problem without fixed starting vertex is easier than the problem with fixed start vertex. That is, maybe it is easy to find a search order with arbitrary root, that generates the tree, but it is $\mathcal{NP}$-hard to find one that uses the given root.

As a second variant, one can also consider the unlabeled problem, i.e., no spanning tree is given, but a tree with a matching number of vertices. Thus, we are looking for a search tree which is isomorphic to the given tree. Obviously, this problem is $\mathcal{NP}$-hard for $\mathcal{L}$-trees of DFS, since it includes the hamiltonian path problem. However, it remains open whether there are searches and graph classes where the unlabeled case is easy or even easier than the labeled one.

References

[1] Beisegel, J., *Characterising AT-free graphs with BFS*, in: A. Brandstädt, E. Köhler and K. Meer, editors, *Graph-Theoretic Concepts in Computer Science*, 2018, pp. 15–26.

[2] Beisegel, J., C. Denkert, E. Köhler, M. Krnc, N. Pivač, R. Scheffler and M. Strehler, *On the End-Vertex Problem of Graph Searches* (2018), submitted, preprint on arXiv: https://arxiv.org/abs/1810.12253.

[3] Beisegel, J., C. Denkert, E. Köhler, M. Krnc, N. Pivač, R. Scheffler and M. Strehler, *Recognizing Graph Search Trees* (2018), preprint on arXiv: https://arxiv.org/abs/1811.09249.

[4] Charbit, P., M. Habib and A. Mamcarz, *Influence of the tie-break rule on the end-vertex problem*, Discrete Math. Theor. Comput. Sci. **16** (2014), p. 57.

[5] Corneil, D. G., B. Dalton and M. Habib, *LDFS based certifying algorithm for the Minimum Path Cover problem on cocomparability graphs*, SIAM J. Comput. **42** (2013), pp. 792–807.

[6] Corneil, D. G., E. Köhler and J.-M. Lanlignel, *On end-vertices of lexicographic breadth first searches*, Discret. Appl. Math. **158** (2010), pp. 434–443.

[7] Corneil, D. G. and R. M. Krueger, *A unified view of graph searching*, SIAM J. Discret. Math. **22** (2008), pp. 1259–1276.

[8] Corneil, D. G., S. Olariu and L. Stewart, *Linear time algorithms for dominating pairs in asteroidal triple-free graphs*, SIAM J. Comput. **28** (1999), pp. 1284–1297.

[9] Corneil, D. G., S. Olariu and L. Stewart, *The LBFS structure and recognition of interval graphs*, SIAM J. Discret. Math. **23** (2009), pp. 1905–1953.

[10] Crescenzi, P., R. Grossi, M. Habib, L. Lanzi and A. Marino, *On computing the diameter of real-world undirected graphs*, Theor. Comput. Sci. **514** (2013), pp. 84–95.

[11] Dusart, J. and M. Habib, *A new LBFS-based algorithm for cocomparability graph recognition*, Discret. Appl. Math. **216** (2017), pp. 149–161.

[12] Edmonds, J. and R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. ACM **19** (1972), pp. 248–264.

[13] Even, S., "Graph Algorithms," Cambridge University Press, 2011, 2nd edition pp. 46–48.

[14] Golumbic, M., "Algorithmic Graph Theory and Perfect Graphs," Ann. Discrete Math, Vol. 57, Elsevier, 2004 pp. 98–99.

[15] Habib, M., R. McConnell, C. Paul and L. Viennot, *Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition, and consecutive ones testing*, Theor. Comput. Sci. **234** (2000), pp. 59–84.

[16] Hagerup, T. and M. Nowak, *Recognition of spanning trees defined by graph searches*, Technical Report A 85/08, Universität des Saarlandes (1985).

[17] Hopcroft, J. and R. E. Tarjan, *Algorithm 447: Efficient algorithms for graph manipulation*, Commun. ACM **16** (1973), pp. 372–378.

[18] Hopcroft, J. and R. E. Tarjan, *Efficient planarity testing*, J. ACM **21** (1974), pp. 549–568.

[19] Korach, E. and Z. Ostfeld, *DFS tree construction: Algorithms and characterizations*, in: J. van Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science*, 1989, pp. 87–106.

[20] Manber, U., *Recognizing breadth-first search trees in linear time*, Inform. Process. Lett. **34** (1990), pp. 167–171.

[21] Rose, D. J., G. S. Lueker and R. E. Tarjan, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput. **5** (1976), pp. 266–283.

[22] Spinrad, J. and R. Sritharan, *Algorithms for weakly triangulated graphs*, Discret. Appl. Math. **59** (1995), pp. 181–191.

[23] Tarjan, R. E., *Depth-first search and linear graph algorithms*, SIAM J. Comput. **1** (1972), pp. 146–160.

[24] Tarjan, R. E., *Edge-disjoint spanning trees and depth-first search*, Acta Informatica **6** (1976), pp. 171–185.

[25] Tarjan, R. E. and M. Yannakakis, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, SIAM J. Comput. **13** (1984), pp. 566–579.