UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Master's thesis
(Magistrsko delo)

Graph search algorithms and structure of graph search trees

(Algoritmi iskanja na grafih in struktura iskalnih dreves)

Ime in priimek: Nevena Pivač
Študijski program: Računalništvo in informatika, 2. stopnja
Mentor: doc. dr. Matjaž Krnc

Koper, avgust 2020

Ključna dokumentacijska informacija

Ime in PRIIMEK: Nevena PIVAČ

Naslov magistrskega dela: Algoritmi iskanja na grafih in struktura iskalnih dreves

Kraj: Koper

Leto: 2020

Število listov: 90 Število slik: 28 Število tabel: 3

Število referenc: 43

Mentor: doc. dr. Matjaž Krnc

UDK: 519.17(043.2)

Ključne besede: iskanje na grafih, iskalno drevo, polinomski algoritem, iskanje v širino, iskanje v globino, seznam vozlišč

Math. Subj. Class. (2020): 05C85, 68R10, 05C75

Izvleček:
V magistrskem delu obravnavmo metode iskanja na grafih. Iskanje na grafih predstavlja sistematičen obisk vozlišč grafa, tako da začnemo v enem vozlišču grafa in se sprehajamo po grafu, pri čemer v naslednjem koraku iteracije obiščemo vozlišče, ki že ima obiskanega soseda v tem grafu. V delu opišemo naslednje metode iskanja na grafih: iskanje v širino (BFS), iskanje v globino (DFS), leksikografsko iskanje v širino (LexBFS), leksikografsko iskanje v globino (LexDFS), iskanje po maksimalni kardinalnosti (MCS) in iskanje po maksimalni soseščini (MNS).
V delu podamo pregled znanih rezultatov omenjenih metod iskanja, posebnosti časovno učinkovite implementacije in karakteriziramo sezname vozlišč v grafu, ki so lahko rezultat določenega iskanja na grafu. Potem se osredotočimo na zveze vsebovanosti med določenimi metodami iskanja na grafih in karakteriziramo grafe, za katere drži, da je vsak iskalni seznam vozlišč iskalne metode A tudi rezultat iskalne metode B.
V zadnjem delu naloge se osredotočimo na iskalna drevesa. Študiramo problem prepoznavanja iskalnih dreves in predstavimo NP-težke različice problema. Podamo tudi polinomske algoritme za določene različice problema (prepoznavanje iskalnih dreves za BFS, DFS, in LexDFS). Omenjene polinomske algoritme za prepoznavanje iskalnih dreves grafa implementiramo v programskem jeziku SageMath.

Key document information

Name and SURNAME: Nevena PIVAČ

Title of the thesis: Graph search algorithms and structure of graph search trees

Place: Koper

Year: 2020

Number of pages: 90 Number of figures: 28 Number of tables: 3

Number of references: 43

Mentor: Assist. Prof. Matjaž Krnc, PhD

UDC: 519.17(043.2)

Keywords: graph search, search tree, polynomial-time algorithms, BFS, DFS

Math. Subj. Class. (2020): 05C85, 68R10, 05C75

Abstract: In the thesis we consider graph search methods, the methods of systematic visiting the vertices of a graph so that at every step we visit a neighbor of some already visited vertex. In the thesis we describe the following search methods: Breadth First Search (BFS), Depth First Search (DFS), Lexicographic Breadth First Search (LexBFS), Lexicographic Depth First Search (LexDFS), Maximum Cardinality Search (MCS), and Maximal Neighborhood Search (MNS). There are two possible outcomes of a graph search method: a search ordering and a search tree.

In the first part of the thesis we give an overview of known results of the proposed search methods, details of efficient implementations and structural results characterizing when a given ordering of vertices in G is a search ordering of a given type.

In the second part of the thesis we study the inclusion relations among various search methods, and characterize graphs in which every search ordering of a type A is also a search ordering of type B.

The last part of the thesis is devoted to search trees. We study the problem of search tree recognition and give hardness results for some variants of the problem. Further, we give polynomial-time algorithms that solve some variants of a problem (polynomial-time algorithms for solving the search tree recognition problem for BFS, DFS and LexDFS). Finally, we implement the proposed polynomial-time algorithms in the programming language SageMath.

Acknowledgements

I express my deep gratitude to my advisor Matjaž Krnc for his time, patience and all suggestions during the development of my Master's thesis. Also, I would like to thank Martin Milanič, as well as to Nina Chiarelli, Ekki Köhler, Martin Strehler, Jesse Beisegel and Robert Scheffler for giving me the opportunity to collaborate with them and to study the problems presented in this thesis.

Hvala mojoj porodici.
Posebno hvala Darku za svu podršku, toleranciju, razumijevanje, i ljubav.

List of Contents

List of Tables

List of Figures

List of Abbreviations

i.e.	that is
e.g.	for example
BFS	Breadth First Search
DFS	Depth First Search
LexBFS	Lexicographic Breadtg First Search
LexDFS	Lexicographic Depth First Search
MCS	Maximum Cardinality Search
MNS	Maximal Neighborhood Search
PEO	perfect elimination order
itn.	in tako naprej

1 Introduction

Graph search represents one of the fundamental algorithmic concepts in graph theory and theoretical computer science. In general, graph search is defined as a method of systematically visiting the vertices in a graph starting from a chosen vertex, and visiting all the vertices in a graph so that at every step a vertex selected to be visited next is a neighbor of some already visited vertex. Such a general definition of a selection rule implies the existence of many vertices satisfying the requirement that they have some already visited neighbor, so it leaves much freedom for which vertex will be selected next. There exist various restrictions of the selection rule, and such restrictions define the specific search methods. Among the oldest known search methods we can find the Breadth First Search (BFS) and Depth First Search (DFS) [19]. Breadth First Search is known as a method that at every step of iteration selects a vertex with the earliest visited neighbor, while Depth First Search is known as a method that at every step of iteration selects a vertex with most recently visited neighbor.

The selection rule can be defined in many different ways. For example, one may require that at every step we select a vertex with the largest number of already visited neighbors, or a vertex whose visited neighborhood is not properly contained in the neighborhood of some other unvisited vertex, etc. Various selection rules define the new search methods. Besides BFS and DFS, in this thesis we consider their lexicographic instance, Lexicographic Breadth First Search (LexBFS) and Lexicographic Depth First Search (LexDFS), as well as Maximum Cardinality Search (MCS) and Maximal Neighborhood Search (MNS).

There are two possible outcomes of a graph search: a search ordering and a search tree. A *search order* of a graph G is a sequence of vertices in G in the order they were visited by the search. A *search tree* of a connected graph G is a spanning tree of G satisfying the requirement that every vertex is adjacent in T to exactly one its previously visited neighbor from G. Clearly, the previously visited neighbor of a vertex v is not uniquely determined. In the literature we can find two distinct formulations of a search tree, known as $\mathcal{F}$-tree and $\mathcal{L}$-tree. In an $\mathcal{F}$-tree every vertex is adjacent to its first visited neighbor, while in an $\mathcal{L}$-tree every vertex is adjacent to its most recently visited neighbor.

We present the proposed graph search methods and give an overview of the related

known results in the field, as well as the details of the time-efficient implementation for every search method. Furthermore, for every search method we give a characterization of vertex orderings that can be obtained as a result of that search method.

As already mentioned, every search method is defined using some particular selection rule. If some selection rule is more restrictive than the other one, then the corresponding search method is a restriction (or a special instance) of the search method defined with the more general selection rule. For instance, as we will see later, from the definition of BFS and LexBFS it follows that LexBFS is a special instance of BFS, meaning that the every LexBFS ordering of a graph G is a BFS ordering of a graph G. A natural question that arises is: which conditions must hold for a graph G if every LexBFS ordering of G is a BFS ordering of G? If the sets of vertex orderings produced by two different search methods are equal, we say that the corresponding search methods are *equivalent*. We study the equivalence relations between the following pairs of search methods: BFS and LexBFS, DFS and LexDFS, MNS and MCS, MNS and LexDFS, MNS and LexBFS, and for almost all of them we characterize the graphs G for which the particular equivalence holds.

Another possible outcome of a graph search method is a search tree. In literature we can find the search tree recognition problem: the problem of determining whether a given spanning tree of a graph G is a search tree of a graph G of given type. This problem was first introduced by Hagerup in 1985 [24].

$\mathcal{F}$-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 the given type such that T is its $\mathcal{F}$-tree ($\mathcal{L}$-tree) of G.

We present the hardness results proving that the recognition of search trees is NP-hard for $\mathcal{F}$-trees of LexBFS, LexDFS, MCS and MNS on weakly chordal graphs [4]. Further, we give polynomial-time algorithms that solve some variants of a problem. In particular, we present algorithms that solve the following problems:

1. An algorithm that recognizes whether a tree T is a DFS search tree of a graph G rooted at vertex $r \in V(G)$ [24],

2. An algorithm that recognizes whether a tree T is a LexDFS search tree of a graph G rooted at vertex $r \in V(G)$ [1],

3. An algorithm that recognizes whether a tree T is a BFS search tree of a graph G rooted at vertex $r \in V(G)$ [33].

In the last part of the thesis we present the implementation of the above-mentioned polynomial-time algorithms in the programming software SageMath.

Structure of the thesis. In Chapter 2 we give an overview of notations, definitions, and fundamental theoretical results used in the thesis. The proposed search methods and related known results are presented in Chapter 3. In Chapter 4 we study the equivalence relations between different search methods. Chapter 5 is devoted to the description of search trees, while Chapter 6 contains the details of the implementation of the polynomial-time algorithms for the recognition of search trees in the programming software SageMath.

2 Theoretical background

In this chapter we provide the necessary definitions, notation, and fundamental results used throughout the thesis. In the first section we introduce the basic graph theoretic concepts and graph classes. The second section is devoted to classes of problems, defined with respect to the computational complexity of a problem.

2.1 Graphs and digraphs

All graphs in this thesis are finite, either directed, or undirected. A *simple undirected graph* without loops is denoted by $G = (V, E)$, where V is a set of *vertices* and $E \subset V \times V$ a set of *edges*. In this work we will use the term *graph* for a simple undirected graph. The number of vertices and edges in G is the cardinality of sets V and E, and is denoted by n and m, respectively. An edge of a graph G between vertices u and v is denoted by uv. Two vertices of a graph $G = (V, E)$ are adjacent if $uv \in E$, and in that case we say that u and v are *neighbors* in G. The set of all neighbors of a vertex $v \in V$ is denoted by $N_G(v)$, and the degree of a vertex v is defined as the number of neighbors of v in G and is denoted by $d_G(v)$. The *complement* of a graph $G = (V, E)$ is the graph $\overline{G}$ whose vertex set is V, and edge set is $\{uv \mid u, v \in V, u \neq v, uv \notin E\}$.

A *simple directed graph*, or a *digraph*, is denoted by $D = (V, A)$ where V is a set of vertices and A a set of *arcs* (or *directed edges*). In digraphs all arcs have distinct endpoints, and two arcs forming a cycle of a length two are allowed. A directed edge from u to v is denoted by $u \to v$. Given a vertex $v \in V$, the set of incoming (resp., outcoming) directed edges of a vertex v is a set of all directed edges in D of type $u \to v$ (resp., $v \to u$) in A.

An *orientation* of a graph $G = (V, E)$ is a digraph obtained by assigning each edge of G a direction. The *underlying graph* of a digraph D is the undirected graph created using all of the vertices in D, and replacing all arcs in D with undirected edges, so that multiple edges in the resulting graph are not allowed.

For a graph $G = (V, E)$, a *walk* is defined as a sequence of alternating vertices and edges such as $v_0, e_1, v_1, e_2, ..., e_k, v_k$ where each edge $e_i = v_{i-1}v_i$. A *trail* is a walk with no repeated edges, while a *path* is a trail with no repeated vertices (and consequently $v_0 \neq v_k$). A *cycle* is defined as a trail satisfying $v_0 = v_k$ where no other vertices are

repeated. A *Hamiltonian path* in a graph is a path between two vertices of a graph that visits each vertex of the graph exactly once. With C_n and P_n we denote the cycle and the path graphs on n vertices. The distance between two vertices u and v in G is the length of a shortest path between u and v in G and is denoted by $d_G(u, v)$ (if no path between u and v exists, then it is infinity). The *eccentricity* of a vertex $v \in V(G)$ is denoted by $\epsilon_G(v)$ and defined as $\epsilon_G(v) = \max_{u \in V(G)} d_G(u, v)$. If G is clear from the context, we simply omit G from notation and write $N(v)$, $d(v)$, $d(u, v)$, $\epsilon(v)$.

A *clique* in a G is a set of pairwise adjacent vertices in G, while an *independent set* in G is a set of pairwise non-adjacent vertices in G. A graph $G = (V, E)$ is *complete* if V is a clique, and a complete graph on n vertices is denoted by K_n. A vertex is *simplicial* if its neighborhood is a clique, *universal* if it is adjacent to every other vertex in the graph, and *isolated* if it has degree 0.

Given a set $S \subseteq V$, the *subgraph of G induced by S* is a graph denoted by $G[S]$ and defined with vertex set S and edge set $\{uv \in E \mid u, v \in S\}$. An *induced subgraph* of G is a graph induced by some set $S \subseteq V$. With $G - S$ we denote the subgraph of G induced by $V(G) \setminus S$. Given a graphs G and H, we say that G is H-free if G does not contain H as an induced subgraph.

An *ordering of vertices* in a graph 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 a vertex v, and by $\sigma^{-1}(i)$ the vertex placed on position i, for $i \in \{1, \ldots, n\}$. Two vertices $u, v \in V(G)$ satisfy the relation $u <_\sigma v$ if $\sigma(u) < \sigma(v)$, and in that case we say that *u is to the left of v* and that *v is to the right of u*. If a directed graph D contains no cycle in which all edges are oriented into the same direction, then D is a *directed acyclic graph*. A *topological sort* or *topological ordering* of a directed acyclic graph is a linear ordering of its vertices such that for every directed edge $u \to v$ from vertex u to vertex v, u comes before v in the ordering.

A *connected component* of $G = (V, E)$ is a subgraph of G induced by a maximal set of vertices in G such that each pair of vertices is connected by a path. A vertex $v \in V$ is a *cut-vertex* (or a *separating vertex*) in G if its removal increases the number of connected components in G. A graph G is *connected* if it has exactly one connected component, 2-*connected* if it has at least three vertices and for all $S \subseteq V$ containing at most two vertices the graph $G - S$ is connected, and (*biconnected*) if it is connected and has no cut-vertices. An edge in a graph is called *a bridge* if its removal increases the number of connected components in the graph.

The *n-pan graph*, denoted as C_n^+, is the graph obtained by joining to a cycle C_n an isolated vertex with a bridge. A graph is a *pan* if it is n-pan for some n. A *paw* is the 3-pan graph, a *triangle* is a C_3, and a *diamond* is a C_4 with one additional edge. A *tree* is a connected graph in which every two vertices are connected by exactly one path,

or equivalently, it is an *acyclic connected graph*. A *forest* is a graph each connected component of which is a tree. A *rooted tree* is a tree in which one vertex is determined to be a root of the tree. In a rooted tree, an *ancestor* of a vertex v is any vertex on the path from the root to vertex v, and a *descendant* of v is any vertex with ancestor v. The *parent* of a non-root vertex v is the ancestor of v adjacent to v, while a *child* of a vertex v is any vertex whose parent is v. Given a tree T and a vertex v in T, the *subtree of v in T* is a subtree of T consisting of v and all its descendants. Given a connected graph G, a *spanning tree in G* is any tree T that has the same vertex set as G and satisfies that $E(T) \subseteq E(G)$.

Graph classes

A graph class is *hereditary* if it is closed under deletion of vertices (equivalently, if it closed under induced subgraphs). An *interval graph* is a graph whose vertices can be represented as intervals on the real line, with two vertices being adjacent if and only if the corresponding intervals intersect. If additionally all intervals are of the same length, then we have *a unit interval graph*. A graph is *chordal* if it does not contain any induced cycle of length at least 4. A *perfect elimination order* (or a *perfect elimination scheme*, abbreviated PEO) in $G = (V, E)$ is an ordering of the vertices in G such that for every vertex $v \in V$ the neighbors of v that occur after v in the order form a clique. Equivalently, it means that every vertex v_i is simplicial in the subgraph induced by all vertices that occur after v_i (including v_i) in the order. In the following theorem we give a known characterization of chordal graphs.

Theorem 2.1 (Fulkerson and Gross [20])**.** *A graph G is chordal if and only if G has a perfect vertex elimination order. Moreover, any simplicial vertex can start a perfect elimination order.*

A graph is *weakly chordal* if neither the graph nor its complement contain an induced cycle of a length at least 5. A *k-partite graph* is a graph whose vertices can be partitioned into k pairwise disjoint independent sets. If there is an edge between any two vertices in different independent sets, then the graph is a *complete k-partite graph*. A graph is complete multipartite if it is complete k-partite for some $k \geq 1$. A (complete) 2-partite graph is also called a (complete) bipartite graph. A graph G is *bipartite permutation* if it is bipartite and if there exists some pair P, Q of permutations of $V(G)$ such that there is an edge between vertices x and y if and only if x precedes y in one of P, Q, while y precedes x in the other.

A *transitive orientation* of a graph $G = (V, E)$ is an orientation (V, A) of all edges in G such that for any three vertices $a, b, c \in V$ it holds that $a \to b \in A$ and $b \to c \in A$ imply that $a \to c \in A$. A *comparability graph* is a graph that has a transitive

orientation. A *cocomparability graph* is a graph whose complement is comparability. The ordering $\sigma = (v_1, \ldots, v_n)$ of vertices in a graph G is said to be a *cocomparability ordering* if for any $i, j, k \in \{1, \ldots, n\}$ with $i < j < k$ we have that $v_iv_k \in E$ implies that $v_iv_j \in E$ or $v_jv_k \in E$. Kratch and Stewart showed that a graph $G = (V, E)$ is a cocomparability graph if and only if it has a cocomparability ordering [29].

An *asteroidal triple* in a graph $G = (V, E)$ is an independent triple of vertices such that for any two of them there is a path that does not intersect the closed neighborhood of the third one. An *asteroidal-triple-free graph* (AT-free) graph is a graph that does not contain any asteroidal triple.

In the following we give a result by Olariu, characterizing the paw-free graphs, which will be used in some proofs in this thesis.

Theorem 2.2 (Olariu [36])**.** *A paw-free graph is either triangle-free, or complete multipartite.*

2.2 Complexity theory

One of the most important things in the study of particular decision or optimization problems is the time needed to get a solution of the problem. Clearly, the time necessary to solve a given problem depends on input size. The running time of an algorithm is standardly defined as the function mapping a given positive integer n to the maximum number of arithmetic operations and comparisons that the algorithm performs on an input instance of size n. In complexity theory, problems are divided into classes with respect to the running time of algorithms that solve them. So we say that a problem *is solvable in polynomial time* if there exists an algorithm that solves it in time that is bounded by a polynomial function of input size.

A fundamental complexity class is class P, which consists of problems solvable in polynomial time. Class P is considered to be the set of problems that can be solved efficiently. Inside the class P we can distinguish problems depending on the degree of the polynomial function that bounds the time necessary to solve them. So the easiest among them are problems for which that function does not depend on the size of the input - the problems solvable in *constant time*. This means that there is a constant c such that the running time of the algorithm that solves the problem is at most c, no matter how large the input is. In general, we can think about the function f as $f(n) = c = cn^0$, meaning that f has as a factor the size of input to the power 0.

If we go one step further, we get problems for which the algorithm has to sequentially read its entire input, that is, there is no constant c that bounds the time complexity of the algorithm that solves the problem. The easiest among them are problems for

which the time needed to solve the problem increases at most linearly with the size of the input. More precisely, a function f is a linear function in the size of input - it has a factor of input to the power at most 1. If a problem is bounded by function linear in the size of input, we say that it is solvable in *linear time*. In the particular problem on a graph $G = (V, E)$ can be solved in linear time, then the algorithm that solves the problem sequentially read the vertices and the edges of G, meaning that the execution time of the algorithm is $\mathcal{O}(|V| + |E|)$. Linear time is the best possible time complexity in situations where the time complexity is not bounded by constant, and much research has been invested into discovering algorithms exhibiting linear time or, at least, nearly linear time.

A decision problem Π for which it may not be known whether there exists a polynomial-time algorithm that solves it, but for any input I such that $\Pi(I)$ gives answer yes, there exists a certificate C such that using C, the fact that $\Pi(I)$ gives answer yes can be verified in time polynomial in the size of input I, is said to be solvable in non-deterministic polynomial time. Such problems define the complexity class NP. Clearly, a yes instance of any polynomial-time solvable problem Π can be verified in polynomial time, so it is true that P $\subseteq$ NP. Whether the converse inclusion holds is far from trivial and is a major open question, with a conjecture that P$\neq$ NP. In particular, the conjectured set of problems that are supposed to belong to class NP and not to the class P is of special research interest. A problem Π is said to be NP-hard if the existence of a polynomial-time algorithm that solves Π implies the existence of a polynomial-time algorithm for any problem in the class NP. It means that the existence of a polynomial-time algorithm for at least one NP-hard problem implies the equality between sets P and NP.

Among the NP-hard problems, problems belonging to the class NP are of special interest. We say that a problem Π is NP-complete if it is in NP and if every problem in NP polynomially reduces to Π. Clearly, every NP-complete problem is NP-hard, but there are also NP-hard problems that are not in NP. A conjectured relationships between complexity classes mentioned here are displayed in Fig. 1.

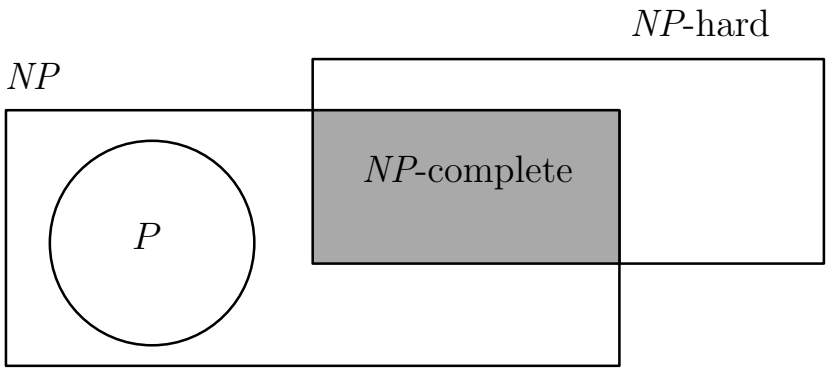


Figure 1: Conjectured relationships between some complexity classes.

Under the assumption that the set of NP-complete problems is non-empty, we can say that NP-complete problems are the hardest problems in NP [21]. The existence of a polynomial-time algorithm for any one of them implies the existence of a polynomial-time algorithm for all of them. One of the fundamental results in this area is the first NP-completeness proof in the literature, guaranteeing that the set of NP-complete problems is non-empty. It is known as Cook's Theorem and proves the NP-completeness of a problem called Satisfiability [10].

Satisfiability	
Instance:	A set U of binary variables $x_1, x_2, \ldots, x_n$, a collection C of clauses representing disjunctions of elements in U or their negations.
Question:	Is there a satisfying truth assignment for C?

The family of known NP-complete problems is growing rapidly, so nowadays there are thousands of problems proved to be NP-complete. Among them we can find a problem known under the name 3-Satisfiability (abbreviated 3-Sat) representing the special case of Satisfiability where each clause has exactly three elements.

3-Satisfiability	
Instance:	A set U of binary variables $x_1, x_2, \ldots, x_n$, a collection C of clauses representing disjunctions of exactly 3 elements, either in U or their negations.
Question:	Is there a satisfying truth assignment for C?

In order to prove that some problem Π belongs to the class of NP-complete problems, it suffices to show that $\Pi \in$ NP and that Π is at least as hard as some other problem in NP, or, in other words, that using a polynomial-time algorithm for problem Π we can construct a polynomial-time algorithm for some problem in NP. Such a correspondence between two problems is called a polynomial reduction.

Definition 2.3. A decision problem Π_1 can be *polynomially reduced* to a decision problem Π_2 if there exists a function f that, given an input I_1 for Π_1, constructs an input $I_2 = f(I_1)$ for Π_2 and has the following properties:

1. $f(I_1)$ can be computed in time that is polynomial in the size of I_1,
2. problem Π_1 has answer yes for input I_1 if and only if problem Π_2 has answer yes for input $f(I_1)$.

Such a reduction is also called *Karp's reduction* [26].

The following theorem by Garey et al. [21] characterizes NP-complete problems using the notion of polynomial reduction.

Theorem 2.4. *A problem* Π *is NP-complete if and only if it is in NP and there exists an NP-complete problem that polynomially reduces to* Π.

Here we list some known NP-complete problems that appear in this work (for details see [21]).

HAMILTONIAN PATH
Instance: A graph $G = (V, E)$.
Task: Decide whether there is a Hamiltonian path in G.

MAXIMUM INDEPENDENT SET
Instance: A graph $G = (V, E)$.
Task: Find an independent set of maximum cardinality in G.

MINIMUM CLIQUE COVER
Instance: A graph $G = (V, E)$.
Task: Find a partition $(V_1, \ldots, V_k)$ of V with minimum k such that every V_i, $i \in \{1, \ldots, k\}$ is a clique.

3 Graph Search Algorithms

Graph search represents a fundamental concept in theoretical computer science. It is known as a general method for traversing the vertices of a graph, and many graph algorithms use some graph search method for traversing the vertices in a given graph. In general, graph search is a method of systematically visiting all vertices in the given graph, starting in some initial vertex and iteratively visiting vertices in a graph such that a new vertex is visited only if it is has a neighbor that is already visited [15]. In a most general instance of a graph search method, at every step of iteration we can have many unvisited vertices satisfying the requirement that each of them have some visited neigbor; in that case one can choose arbitrarily the next visited vertex. Graph search methods are an old and well studied concept in theoretical computer science, and some particular types of graph search methods were used in 19th century for solving the maze problems (see [15]).

A very general definition of a graph search leaves much freedom for a selection rule determining which node is chosen next. By defining some specific rule that restricts this choice, various different graph search methods can be defined. If no such rule is defined, then we have a *generic search method*. Other search methods presented in this work are Breadth First Search, Depth First Search, Lexicographic Breadth First Search, Lexicographic Depth First Search, Maximum Cardinality Search and Maximal Neighborhood Search. The relations between the proposed graph search methods are presented in Fig. 2.

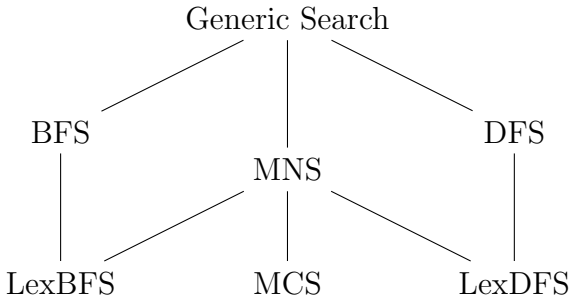


Figure 2: Relations between various graph search methods.

Some important concepts regarding a graph search method are its implementation and its outcome. There are two usual outcomes of a graph search: a search ordering

and a search tree. The algorithms we present here mostly work for connected graphs, while for disconnected graphs, the algorithm should be executed on each connected component. In this section we present some known search methods and give their vertex ordering characterizations, while in Chapter 5 we focus on graph search trees.

3.1 Generic Search

A *generic search* represents a most general search method. The only requirement that the search should satisfy is to visit at each step a neighbor of some of already visited vertices, without any additional requirements. It means that given a connected graph G whose all vertices are unnumbered, we take some vertex $s \in V(G)$ to be a starting vertex, and we choose one of the edges incident with s to traverse. This leads us to a new vertex. Every time we reach some vertex, we assign a number to it, and proceed traversing edges incident with numbered vertices. Every edge is traversed at most once, and if an edge leads us to some already numbered vertex, we traverse another one. Then we repeat the process until all the vertices are numbered. The generic search algorithm is given in Algorithm 1.

Algorithm 1: Generic search.

Input: Connected graph G, and a vertex $s \in V(G)$.
Output: A vertex ordering σ.

```
1 begin
2     S ← {s};
3     for i ← 1 to n do
4         pick and remove an unnumbered vertex v from S;
5         σ(v) ← i;
6         foreach unnumbered vertex w adjacent to v do
7             add w to S;
```

Any vertex ordering produced as a result of some search method on a graph is also generic search ordering, so sometimes the ordering of vertices produced by generic search is called just a *search order*. However, it is not true that any ordering of the vertices of a graph is a (generic) search order.

Example 3.1. Let G be a graph on Fig. 3. It is not difficult to see that, for instance, the orderings $\sigma_1 = (a, b, c, d, e)$, or $\sigma_2 = (b, d, c, e, a)$ are (generic) search orderings, while the ordering $\sigma_3 = (a, b, e, c, d)$ cannot be the result of a generic search on the graph G, since at the moment when the vertex e is added to σ_3, it was not the neighbor of any already visited vertex.

If σ is a generic search ordering of a graph G, then for any vertices $a, b \in V(G)$ such that $a <_\sigma b$ and $ab \notin E(G)$ there should exist some vertex d that appears before b in σ and satisfies that $db \in E(G)$. In order to recognize the orderings of vertices in a graph that can be the result of a generic search on that graph, in the following theorem we give a characterization of a generic search ordering, showed by Corneil and Krueger in [15] (see Fig. 4).

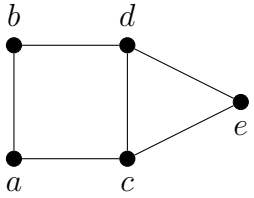


Figure 3: The ordering $\sigma = (a, b, e, c, d)$ is not a generic search ordering of this graph.

Theorem 3.2 (Corneil and Krueger [15])**.** *Let $G = (V, E)$ be a graph. An ordering σ of V is a search ordering of G if and only if the following holds: if $a <_\sigma b <_\sigma c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d such that $d <_\sigma b$ and $db \in E$.*

Proof. Let G be a graph and let σ be a generic search ordering of G. Let $a <_\sigma b <_\sigma c$ be arbitrary triple of vertices in G such that $ac \in E$ and $ab \notin E$. In the moment when b was added to σ, it was an element of a set S from Algorithm 1. From step 6 of algorithm it follows that the only possibility for b to become an element of S is to be a neighbor of some vertex numbered before b. Let d be such a vertex. Then $\sigma(d) < \sigma(b)$ and $db \in E$.

For a proof of the other direction, assume that σ is an ordering of vertices in G such that for every $a <_\sigma b <_\sigma c$ with $ac \in E$, $ab \notin E$, there exists a vertex d, $d <_\sigma b$ and $db \in E$. Assume that σ is not a generic search ordering of G, and let $v_1, v_2, \ldots, v_i$ be the maximal initial segment (prefix) of σ that can be obtained by Algorithm Algorithm 1, with $i \geq 1$. This means that $v_1, v_2, \ldots, v_i, v_{i+1}$ cannot be obtained by Algorithm 1, and the vertex v_{i+1} is the first vertex that cannot be obtained by generic search algorithm. At each step of the algorithm we choose a vertex from the set S, so it follows that in the moment when we wanted to add $(i+1)$-st vertex to σ, v_{i+1} was not an element of S, so v_{i+1} is not adjacent with any vertex in $\{v_1, \ldots, v_i\}$. Let w be a vertex that could be chosen by Algorithm 1 after v_i. It follows that in the $(i+1)$-st step of iteration $w \in S$ and thus w has a neighbor among vertices $\{v_1, \ldots, v_i\}$. Let v_j be a neighbor of w, such that $j \in \{1, \ldots, i\}$. Consider now the vertices v_j, v_{i+1} and w. In ordering σ they satisfy $v_j <_\sigma v_{i+1} <_\sigma w$. From the discussion above we have that $v_j w \in E$, and $v_j v_{i+1} \notin E$. The property of σ then implies that there exists a vertex d, with $d <_\sigma v_{i+1}$ such that $dv_{i+1} \in E$. But vertices satisfying $d <_\sigma v_{i+1}$ in σ are $\{v_1, \ldots, v_i\}$, and none

of them is a neighbor of v_{i+1}; a contradiction. It follows that σ is a search ordering of G. □

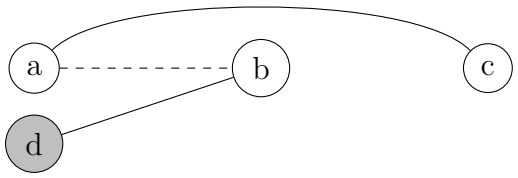


Figure 4: A vertex ordering corresponding to Theorem 3.2. Solid lines represent edges, dashed line represents the non-edge.

3.2 Breadth First Search

Breadth First Search (BFS) represents one of the fundamental algorithms and subroutines in computer science. It is a restriction of a generic search which puts unvisited vertices in a queue and visits a first vertex from the queue in the next iteration. After visiting a particular vertex, all its unvisited neighbors are put at the end of the queue, in arbitrary order (see Algorithm 2).

Algorithm 2: Breadth First Search.

Input: A connected graph G, and a vertex $s \in V(G)$.

Output: A vertex ordering σ.

```
1 begin
2     Q = {s}; i = 1;
3     foreach v ∈ Q do
4         σ(v) ← i; i++;
5         foreach unvisited neighbor w of v with w ∉ Q do
6             append w to Q
7     return σ
```

BFS was first introduced in 1959 by Moore [35] and was used for solving the maze traversal problems. A nice property of BFS is that any BFS starting in a vertex v of a graph G results in a rooted tree (with root v), which contains all the shortest paths from v to any other vertex in G (see [19]). This results in a layered structure of the vertices of given graph, with every layer containing the vertices that are at the same distance from the root. BFS can be implemented to run in time $\mathcal{O}(|V(G)| + |E(G)|)$, so it represents a subroutine in many efficient graph algorithms. In particular, using BFS to find an augmenting path provides a polynomial-time implementation of the Ford-Fulkerson maximum flow algorithm [18].

Theorem 3.3 (Even [19]). *Let G be a graph and let $v \in V(G)$. If σ is an ordering of vertices of G produced by BFS starting on v in G, then for any distinct vertices $u, w \in V(G) \setminus \{v\}$ with $d_G(v,u) > d_G(v,w)$ it holds that $\sigma(u) > \sigma(w)$.*

Although BFS results in an ordering that preserves the distances from the starting vertex, the converse is not true. That is, there exists a graph G, and an ordering σ of vertices in G that satisfies $\sigma(v) = 1$ such that for any two distinct vertices $u, w \in V(G) \setminus \{v\}$ it holds that $d_G(v,u) > d_G(v,w)$ implies that $\sigma(u) > \sigma(w)$, but σ is not a BFS ordering on G.

Example 3.4. Let G be the graph consisting of six vertices, depicted in Fig. 5. Let $\sigma = (a, b, c, f, d, e)$ be an ordering of vertices in G. It is clear that σ satisfies the requirement of non-decreasing distances from vertex a. However, it is not true that σ is a BFS ordering on G, since once we start BFS in the vertex a, no matter how we proceed, it will be true that vertex e either appears before vertex d, or before vertex f in a queue, meaning that e cannot be the last visited vertex in a BFS on G starting in a.

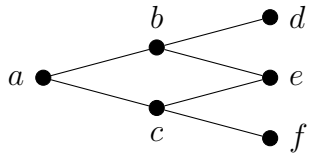


Figure 5: The ordering $\sigma_1 = (a, b, c, f, d, e)$ cannot be the result of any BFS, although it represents the distance-layered structure of the graph. An example of a BFS ordering is $\sigma_2 = (a, b, c, d, e, f)$.

The above example motivates the study of properties of BFS orderings on some graph, so in the following theorem we present the characterization of a BFS ordering (see Fig. 6).

Theorem 3.5 ([Corneil and Krueger [15]). *[30]] An ordering σ of V is a BFS-ordering if and only if the following holds: if $a <_\sigma b <_\sigma c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d such that $d <_\sigma a$ and $db \in E$.*

Proof. Let σ be a BFS ordering of a graph $G = (V, E)$. We want to prove that σ satifies the desired property from theorem. Let a, b, c be arbitrary vertices satisfying $a <_\sigma b <_\sigma c$, $ac \in E$ and $ab \notin E$. We have that $b <_\sigma c$, so b was added to queue Q before c, and by Algorithm 2 it follows that either b and c were added to Q in the same step of iteration, or b was added earlier. In any case, b has a neighbor d that is visited not after a. If $d = a$, then $db = ab \notin E$, so $d \neq a$, and it follows that $d <_\sigma a$, as we wanted to show.

For a proof of the other direction, let $\sigma = (v_1, \ldots, v_n)$ be an ordering of vertices in G, satisfying that for any $a <_\sigma b <_\sigma c$ such that $ac \in E$ and $ab \notin E$ there exists a vertex d such that $d < a$ and $db \in E$ (call this property property (P)). Assume that σ is not a BFS ordering of G, and let $(v_1, v_2, \ldots, v_k)$, $k < n$ be a maximal initial segment of σ that can be obtained by BFS, starting from v_1. Then the vertex v_{k+1} cannot be chosen next, and by Algorithm 2 it follows that after the vertex v_k was chosen, there was a vertex v_i before v_{k+1} in Q, and from definition of σ it follows that $i > k + 1$. Furthermore, the fact that v_i was before v_{k+1} in Q implies that there is a vertex v_j that is the first neighbor of v_i in σ, $j < k + 1$, and v_j is non-adjacent to v_{k+1} in G. Consider now the vertices v_i, v_{k+1}, v_j. We know that $v_j <_\sigma v_{k+1} <_\sigma v_i$, and $v_j v_i \in E$, while $v_j v_{k+1} \notin E$. We assumed at the beginning that σ satisfies property (P), and applied for the mentioned vertices, it implies that there exists a vertex $d < v_j$ such that $dv_{k+1} \in E$. But v_j was the first vertex in σ adjacent to v_i, so $dv_i \notin E$, v_i cannot be visited before v_{k+1} in a BFS that extends $(v_1, \ldots, v_k)$; a contradiction. It follows that σ is a BFS ordering of vertices in G. □

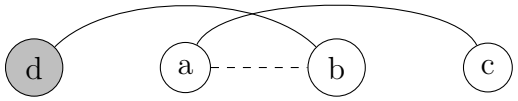


Figure 6: A vertex ordering corresponding to Theorem 3.5. Solid lines represent edges, dashed line represents non-edge.

3.3 Depth First Search

Similarly as Breadth First Search, Depth First Search (DFS) represents one of the fundamental search algorithms in computer science. Although in the literature BFS and DFS are appearing together and represent a well studied concept in computer science, DFS is very different from BFS, in the sense that it traverses the graph as deep as possible, visiting a neighbor of last visited vertex whenever it is possible, and backtracking only when all the neighbors of last visited vertex are already visited. While in BFS the unvisited vertices are put at the end of a queue, in DFS the unvisited vertices are put on top of a stack, so visiting a first vertex in a stack means that we always visit a neighbor of the most recently visited vertex (see Algorithm 3).

Algorithm 3: Depth First Search.

Input: Connected graph G, and a vertex $s \in V(G)$.

Output: A vertex ordering σ.

```
1  begin
2      S = {s}; n = |V(G)|;
3      for i ← 1 to n do
4          pop v from top of S
5          σ(v) ← i
6          foreach unnumbered vertex w adjacent to v do
7              if w is already in S then
8                  remove w from S
9              push w on top of S
10     return σ
```

In the literature we can find the notions of a *discovery DFS ordering* and a *finishing DFS ordering* [15]. The former represents the standard notion of a search ordering, meaning the order in which vertices are visited, while a finishing DFS ordering indicates the order in which each vertex's neighborhood is fully explored with all incident edges traversed and causing a backtracking step. In this thesis we focus on discovery DFS ordering, and simply call it a DFS ordering.

A DFS on a graph G can be done in time $\mathcal{O}(|V(G)| + |V(E)|)$ and such an efficient implementation of DFS influence its use in many graph algorithms. The algorithm has been known since the nineteenth century as a technique for threading mazes; in that time it was known under the name *Trémaux's algorithm* (see [31]). In 1970s, Hopcroft and Tarjan discovered several applications of DFS in graph algorithms and then the method became widely recognized as a method for solving various graph problems. In particular, they developed algorithms for testing whether a graph is biconnected [40], or whether it can be embedded on a plane [25]. Moreover, they considered the DFS on a directed graph and developed the DFS-based algorithm to test whether a directed graph is strongly connected, meaning that for any pair of vertices u and v in a graph there exist a directed paths from u to v and from v to u [40]. One more important application of DFS to directed graphs is computation of a topological ordering for an acyclic digraph. Recall that a topological ordering of an acyclic digraph is a linear ordering of its vertices such that every directed edge $u \to v$ of a digraph satisfies that u is before v in that ordering. Using DFS, a topological ordering of an acyclic digraph can be found in linear time, and is used as a subroutine in many algorithms on digraphs [41].

Many graph algorithms use DFS at some step of their execution, and the proofs of

correctness of such algorithms rely on a structural characterization of a DFS orderings, presented in the following theorem (see Fig. 7).

Theorem 3.6 (Corneil and Krueger [15])**.** *An ordering σ of V is a DFS-ordering if and only if the following holds: if $a <_\sigma b <_\sigma c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d such that $a <_\sigma d <_\sigma b$ and $db \in E$.*

Proof. Let σ be a DFS ordering of a graph $G = (V, E)$. We want to prove that σ satifies the desired property. Let a, b, c be arbitrary vertices satisfying $a <_\sigma b <_\sigma c$, $ac \in E$ and $ab \notin E$. We have that $b <_\sigma c$, so by Algorithm 3 it follows that in the moment when a was added to σ, the neighbors of a were pushed on top of a stack S, and thus c was before b in S. We know that at each step of iteration in DFS, vertices are chosen from the top of the stack S, so there must be a vertex d visited after a and before b that pushed b on top of S (otherwise $c <_\sigma b$). This implies that $db \in E$, as we wanted to show.

For the proof of other direction, let $\sigma = (v_1, \ldots, v_n)$ be an ordering of vertices in G, satisfying that for any $a <_\sigma b <_\sigma c$ such that $ac \in E$ and $ab \notin E$ there exists a vertex d such that $a <_\sigma d <_\sigma b$ and $db \in E$ (call this property property (P)). Assume that σ is not a DFS ordering of G, and let $(v_1, v_2, \ldots, v_k)$, $k < n$ be a maximal initial segment of σ that can be obtained by DFS starting in v_1. Then the vertex v_{k+1} cannot be chosen next, and by Alg. Algorithm 3 it follows that after the vertex v_k was chosen, there was a vertex $v_i \neq v_{k+1}$ on top of a stack S, and from the definition of σ it follows that $i > k + 1$.

Furthermore, the fact that v_i is on the top of S implies that there is a right-most vertex v_j in σ, $j \leq k$, that is a neighbor of v_i and a non-neighbor of v_{k+1} in G. Consider now the vertices v_j, v_{k+1}, v_i. We know that $v_j <_\sigma v_{k+1} <_\sigma v_i$, and $v_j v_i \in E$, while $v_j v_{k+1} \notin E$. We assumed at the beginning that σ satisfies property (P), and applied for the mentioned vertices, this implies that there exists a vertex d, $v_j <_\sigma d <_\sigma v_{k+1}$ such that $dv_{k+1} \in E$. But v_j was the last vertex in $v_i, \ldots, v_k$ adjacent to v_i, so $dv_i \notin E$. Then vertex d pushes its neighbors on the top of a stack S, and it cannot happen that v_i is visited before v_{k+1} in a DFS that extends $(v_1, \ldots, v_k)$; a contradiction. It follows that σ is a DFS ordering of vertices in G. □

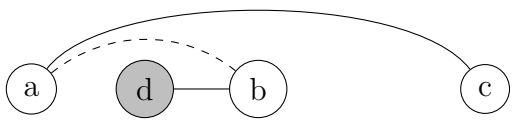


Figure 7: A vertex ordering corresponding to Theorem 3.6. Solid lines represent edges, dashed line represents the non-edge.

3.4 Lexicographic Breadth First Search

Lexicographic Breadth First Search was introduced in the 1970s by Rose, Tarjan and Lueker [37] as part of an algorithm for recognizing chordal graphs in linear time. Since then, it has been used in many graph algorithms mainly for the recognition of various graph classes. Lexicographic Breadth First Search is a restricted version of Breadth First Search, where usual queue of vertices is replaced by a queue of unordered subsets of the vertices which is sometimes refined, but never reordered. The LexBFS can be described as follows: we start a search in some vertex $v \in V(G)$ and we produce an ordering σ of vertices in G, with $\sigma(v) = 1$. For every unvisited neighbor of v we add the value $n+1-\sigma(v)$ to the end of its label, and select a vertex with lexicographic maximal label to be a next visited vertex (see Algorithm 4); we iteratively repeat the process until all vertices are visited. For each vertex, its label consists of a set of numbers listed in decreasing order.

The lexicographic order is the order that appears in dictionaries. More precisely, given a two strings $s_1 = (a_1, a_2, \ldots, a_k)$ and $s_2 = (b_1, b_2, \ldots, b_\ell)$, of symbols over the alphabet $U = \{1, 2, \ldots, n\}$ with a total order $<_U$, it holds that $s_1 <_U s_2$ (that is, s_1 is *lexicographically smaller* than s_2, or s_2 is *lexicographically larger* than s_1), if and only if the following conditions hold:

1. $k < \ell$ and $a_i = b_i$ for all $i \in \{1, \ldots, k\}$, or
2. there is some index $j < \min\{k, \ell\}$ such that $a_i = b_i$ for all $i \in \{1, \ldots, j-1\}$ and $a_j <_U b_j$.

It means that, for example, 34 is lexicographically smaller than 342, and that 5643 is lexicographically smaller than 5735.

If at some step of iteration several vertices have the same label, which is lexicographically maximal, then these vertices are said to be *tied*. A set of all tied vertices is called a *slice*. All vertices from the same slice appear consecutively in the LexBFS ordering σ where at each step of iteration we can choose any among the vertices from the same slice. The concept of slices is used as the main tool in the implementation of LexBFS, and it is known under the name *partition refinement*.

Example 3.7. Let us find a Lexicographic Breadth First Search ordering of graph G on Fig. 8. Assume we start our search in vertex a. The labels of all other vertices in G are equal to empty sets, and we add a number $n = 7$ to the labels of all neighbors of a. Then the vertices with non-empty labels are b, c and d, and all of them have the same label 7, so they form one slice, and we can take an arbitrary vertex among them.

Algorithm 4: Lexicographic Breadth First Search.

Input: Connected graph G, and a vertex $s \in V(G)$.

Output: A vertex ordering σ.

```
1 begin
2     foreach v ∈ V do ℓ(v) = ∅;
3     ℓ(s) = {0}; n = |V(G)|
4     for i ← n to 1 do
5         v ← unnumbered vertex with lexicographically largest label ℓ(v);
6         σ(v) ← n − i;
7         foreach unnumbered neighbor w of v do
8             append i to ℓ(w)
9     return σ
```

Assume we take c to be next visited vertex: $\sigma(c) = 2$. Then the labels of all unvisited neighbors of c are updated, so that $n - 1 = 6$ is appended to the existing labels: now the labels are $\ell(b) = \ell(d) = \{76\}$, $\ell(e) = \ell(g) = \ell(f) = \{6\}$. We can select any among vertices b and d, so assume we take vertex b: $\sigma(b) = 3$. The new labels are $\ell(d) = \{76\}$, $\ell(e) = \ell(g) = \{65\}$, $\ell(f) = \{6\}$. Among them the lexicographically largest label is $\{76\}$, so we take vertex d. Vertex d gets number 4, and its neighbors get label 4, so the labels are $\ell(e) = \{65\}$, $\ell(g) = \{654\}$, $\ell(f) = \{64\}$, and vertex g has the lexicographically largest label. Then $\sigma(g) = 5$, and new labels are $\ell(e) = \{653\}$, $\ell(f) = \{643\}$, so the next visited vertex is e, meaning that f is the last visited vertex. The produced LexBFS ordering is $\sigma = (a, c, b, d, g, e, f)$. The iteration steps are presented in Fig. 9.

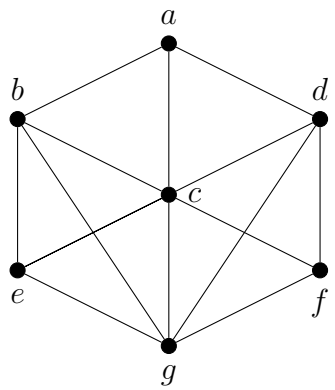


Figure 8: The ordering $\sigma = (a, c, b, d, g, e, f)$ is a LexBFS ordering of G.

a	7	$\emptyset$	7	$\emptyset$	7	$\emptyset$	7	$\emptyset$	7	$\emptyset$	7	$\emptyset$
b		$\{7\}$		$\{76\}$	5	$\{76\}$	5	$\{76\}$	5	$\{76\}$	5	$\{76\}$
c		$\{7\}$	6	$\{7\}$	6	$\{7\}$	6	$\{7\}$	6	$\{7\}$	6	$\{7\}$
d		$\{7\}$		$\{76\}$		$\{76\}$	4	$\{76\}$	4	$\{76\}$	4	$\{76\}$
e		$\emptyset$		$\{6\}$		$\{65\}$		$\{65\}$		$\{653\}$	2	$\{653\}$
f		$\emptyset$		$\{6\}$		$\{6\}$		$\{64\}$		$\{643\}$	1	$\{6432\}$
g		$\emptyset$		$\{6\}$		$\{65\}$		$\{654\}$	3	$\{654\}$	3	$\{654\}$
	Step 1		Step 2		Step 3		Step 4		Step 5		Step 6	

Figure 9: The iteration steps of LexBFS in Example 3.7. In each step the red column represents the numbers and the black one the labels of vertices.

Partition refinement and linear-time implementation

Let G be a graph on n vertices and let $v \in V(G)$ be the initial vertex of LexBFS. We assume that the vertices in G are given in some order with v being the first one. A *partition* $\mathcal{P}$ of a set V is a set of pairwise disjoint subsets of V whose union is exactly V. Every element of a partition $\mathcal{P}$ is *a class* of $\mathcal{P}$. Given partitions $\mathcal{P}$ and $\mathcal{Q}$ of a set V, we say that $\mathcal{P}$ is *a refinement* of $\mathcal{Q}$ if every class of $\mathcal{P}$ is a subset of some class in $\mathcal{Q}$. In the following we describe the partition refinement algorithm for LexBFS, introduced by Habib et al. in 2000 [23].

1. In the first step of execution of the algorithm create one partition class that contains all the vertices of G and put vertex v on the first position.

2. Move the vertex v to its own class of partition, and split the rest of the class into two classes: the set of neighbors of v, $N_G(v)$, and the set of non-neighbors of v, $\overline{N_G(v)}$, where the set $N_G(v)$ appears before. All neighbors of v get label n, while its non-neighbors do not change the label.

3. At each iteration of the LexBFS algorithm, compute the refinement of partition computed in previous step based on the vertex that is chosen to be visited next; this vertex is called *a pivot*. Given a pivot u that is chosen to be visited next, put u in its own class, and for each class S of the existing partition, compute the sets $N_G(u) \cap S$ and $S \setminus N_G(u)$. Moreover, inside each class S the vertices in the set $N_G(u) \cap S$ precede the vertices in $S \setminus N_G(u)$.

4. When the partition is refined, choose a new pivot from the class immediately following the old pivot, and repeat the refinement procedure.

The result of the algorithm is the partition of vertices in G with every class being a singleton. Furthermore, during the iteration, pivots were chosen with respect to

previously visited neighbors, so all vertices in the same class have exactly the same set of already visited neighbors, and thus the same label. Moreover, if two vertices are in distinct classes, then the one that appears earlier has the lexicographically larger label. Altogeter, it follows that the resulting ordering represents a LexBFS ordering.

The refinement can be performed in time $\mathcal{O}(n)$ by using the *doubly linked lists data structure*. We have vertices of a graph in some initial order, and since each partition class contains some consecutive elements from that order, we can think about every class being an interval containing vertices, and it is enough to store the first and the last vertex in every class. For every vertex, we store the class it belongs to. The ordering of the classes is stored using the doubly linked list, where for the each class we have a link to the previous one and to the next one, if it exists. During the refinement at arbitrary step of iteration we have some pivot u and we want to refine the existing partition with respect to $N_G(u)$. It means that for each class C every element in $C \cap N_G(u)$ is removed from C and added to the end of a new class, C_1, where C_1 precedes C. Thus, given a class C, we construct new (possibly empty) classes C_1 and C_2, so that we are checking the elements of E and we add all of them that are adjacent to pivot to the new class C_1, that preceeds the class C_2, while the existing ordering of classes is not changed.

Consider the graph G from Example 3.7. The partition refinement of a graph G that results in the ordering $\sigma = (a, c, b, d, g, e, f)$ is given in Table 1. Every line of a table represents one step of iteration, and vertical lines belonging to a single table line represent the classes of partition. In the last line we have the partition where each class is a singleton, and that partition represents exactly the LexBFS ordering σ of G. Observe that the number of steps in partition refinement should be equal to the number of vertices; in our example this is not the case since we removed the repeated lines.

Table 1: Partition refinement of a search ordering from the Example 3.7.

a	b	c	d	e	f	g
a |	b	c	d	e	f	g
a |	c |	b	d	e	f	g |
a |	c |	b |	d	e	g	f |
a |	c |	b |	d |	g |	e |	f |

Structure of LexBFS ordering and its applications

Since its introduction in 1970s by Rose et al., LexBFS found many applications. The fundamental fact that allows the proofs of correctness of LexBFS-based graph algorithms is a simple characterization of a vertex orderings produced as a result of LexBFS. In the following theorem we present that characterization (see Fig. 10).

Theorem 3.8 (Brändstadt et. al. [7]). *An ordering σ of V is a LexBFS ordering if and only if the following holds: if $a <_\sigma b <_\sigma c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d such that $d <_\sigma a$, $db \in E$, and $dc \notin E$.*

Proof. Let σ be a LexBFS ordering of a graph $G = (V, E)$. We want to prove that σ satifies the desired property. Let a, b, c be arbitrary vertices satisfying $a <_\sigma b <_\sigma c$, $ac \in E$, and $ab \notin E$. We have that $b <_\sigma c$, so by Algorithm 4 it follows that in the moment when a was added to σ, the label of c was updated by adding the number of a vertex a to its end, while the label of b stayed the same. It means that before a was added to σ, a label of b was lexicographically larger than the label of c (otherwise we would have $c <_\sigma b$). Let us denote by $\ell(b) = s_1 s_2 \dots s_{k_1}$ and $\ell(c) = p_1 p_2 \dots p_{k_2}$ the labels of b and c, respectively, just before vertex a was added to σ. Since $\ell(b) > \ell(c)$, we can take the minimal i: $1 \leq i \leq \min\{k_1, k_2\}$ such that $s_i \neq p_i$. Then $s_i > p_i$ and s_i is a number belonging to some vertex d in G, that appears before a in σ. Number s_i belongs to the label of b, so $d \in N(b)$. On the other hand, since labels are decreasing sequences of numbers, it follows that s_i is not contained in the label of c, and thus $d \notin N(c)$. Finally, vertex d satisfies that $d <_\sigma a$, $db \in E$, $dc \notin E$, as we wanted to show. Since vertices a, b, c were arbitrary, the statement follows.

Let us now prove the other direction. Let $\sigma = (v_1, \dots, v_n)$ be an ordering of vertices in G, satisfying that for any $a <_\sigma b <_\sigma c$ such that $ac \in E$ and $ab \notin E$ there exists a vertex d such that $d <_\sigma a$ and $db \in E$, $dc \notin E$ (call this property property (P)). Assume that σ is not a LexBFS ordering of G, and let $(v_1, v_2, \dots, v_k)$, $k < n$ be a maximal initial segment of σ that can be obtained by LexBFS. Then vertex v_{k+1} cannot be chosen next, and by Algorithm 4 it follows that after vertex v_k was chosen, there was a vertex v_i, $i > k+1$ such that v_i has a lexicographically larger label than v_{k+1}. It means that there is a vertex v_j, $j \leq k$, that is a neighbor of u and a non-neighbor of v_{k+1}. Let v_j be the leftmost such vertex in σ.

Consider now the vertices v_j, v_{k+1}, v_i. We know that $v_j <_\sigma v_{k+1} <_\sigma v_i$, and $v_j v_i \in E$, while $v_j v_{k+1} \notin E$. We assumed at the beginning that σ satisfies the property (P), and applied for the mentioned vertices, this implies that there exists a vertex d, $d <_\sigma v_j$, such that $dv_{k+1} \in E$ and $dv_i \notin E$. But v_j was the leftmost vertex that is adjacent to v_i and non-adjacent to v_{k+1}, so every vertex on the left of v_j that is a neighbor of v_i is also a neighbor of v_{k+1}. This means that after adding vertex d to σ, the label

of v_{k+1} enlarges, while the label of v_i stays the same. Then the label of v_i cannot be lexicographically larger than the label of v_{k+1} and thus v_{k+1} has to be chosen before v_i in an LexBFS extension of $v_i, \ldots, v_k$; a contradiction with maximality of k. The statement follows. □

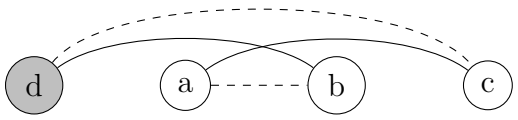


Figure 10: A vertex ordering corresponding to Theorem 3.8. Solid lines represent edges, dashed lines represent non-edges.

The linear-time implementation of LexBFS yields efficient algorithms for a various graph problems. In particular, the best known recognition algorithms for many graph classes are based on LexBFS. Among the results concerning the recognition of graph classes, we should mention the well known linear-time recognition of chordal graph, that follows from the correspondence between LexBFS ordering and a perfect elimination order of chordal graph [37].

Theorem 3.9 (Rose et al. [37])**.** *If G is a chordal graph and σ is a LexBFS ordering of G, then the reverse of σ is a perfect elimination ordering of G.*

Given a graph G, one can find a LexBFS ordering σ of G in linear time. By Theorem 3.9 it follows that checking whether the reverse of σ is a perfect elimination ordering of G will decide whether G is a chordal graph. As showed in [42], this can be done in linear time, so the recognition algorithm for chordal graphs runs in linear time. Tarjan and Yannakakis extended this algorithm, so that it finds in linear time, an induced cycle of length at least 4, when the reverse of σ is not a perfect elimination ordering [42]. There is no known linear-time algorithm that recognizes the chordal graphs, and is not based on LexBFS.

In literature we can find the so-called *multi-sweep LexBFS algorithms*, that are using multiple sweeps of LexBFS in a given graph. At each execution of LexBFS among the vertices in the same slice, we decide which one to visit next, based on the search order produced in the previous LexBFS sweep. Using the information from the previous sweep(s) the ties which vertex to choose next are broken. There are many graph problems that can be solved using multi-sweep LexBFS algorithms [11]. The most notable application of LexBFS is the recognition of various graph classes, and for many of them the best recognition algorithm is based on LexBFS, usually in the form of a multi-sweep algorithm. Among such algorithms there is an algorithm for the recognition of unit

interval graphs, that uses three LexBFS sweeps and checks whether particular conditions are fulfilled in the resulting ordering [12]. There is also a linear-time LexBFS based algorithm for recognition of bipartite permutation graphs, that uses two LexBFS sweeps [16]. Both algorithms, for unit interval and for bipartite permutation graphs run in linear time. One of the first algorithms that was using LexBFS at some point of its execution was a linear-time algorithm given by Corneil et al. [17] that recognizes the interval graphs. Besides mentioned algorithms, there are known LexBFS based algorithms that recognize graphs having some particular structural restriction regarding the existence of P_4. In [9] one can find an algorithm for the recognition of P_4-free graphs (also known as *cographs*). Furthermore, there are algorithms that recognize P_4-reducible graphs and P_4-sparse graphs, given in [8], as well as an algorithm for the recognition of distance-hereditary[1] graphs [8]. All these algorithms run in linear time.

Besides recognition algorithms for various graph classes, LexBFS is used in algorithms for finding a dominating pair[2] in a connected AT-free graph [16], for diameter approximation for various families of graphs [11], and many others. A detailed survey on LexBFS applications can be found in [11].

In some of the above mentioned algorithms at some step of execution there appears a special instance of LexBFS, also known as LexBFS+, where the ties are broken by referencing to a previous LexBFS ordering. It means that we compute a LexBFS ordering σ by Algorithm 4 and then we are able to compute a LexBFS+ ordering σ^+ by the same algorithm, with the additional restriction that at each step of iteration among the vertices with lexicographically largest label we take the one that appears the last in σ. LexBFS+ can be computed in linear time using the implementation of LexBFS [11]. Given a graph G in Example 3.7, we saw that $\sigma = (a, c, b, d, g, e, f)$ is the LexBFS ordering of G. Let us now compute the LexBFS+ ordering σ^+ of G, starting in a. We have that $\sigma^+(a) = 1$ and the labels are $\ell(b) = \ell(c) = \ell(d) = \{7\}$. Now among the vertices with the same lexicographic maximal label we take the the one that appears the last in σ, that is, we take vertex d. Now we have that $\sigma^+(d) = 2$ and new labels are $\ell(b) = \{7\}$, $\ell(c) = \{76\}$, $\ell(f) = \ell(g) = \{6\}$. Now we have to take vertex c. $\sigma^+(c) = 3$. In the next step we get that $\sigma^+(b) = 4$, and labels are $\ell(e) = \{54\}$, $\ell(f) = \{65\}$, and $\ell(g) = \{654\}$. Finally, we take the vertex g, then e, and at the end vertex f. The LexBFS+ ordering σ^+ is equal to (a, d, c, b, g, e, f).

Another nice result considering LexBFS orderings is that given a graph G, one can compute a LexBFS ordering of its complement in time $\mathcal{O}(n + m)$ [11]. This can be

[1] A graph G is *distance-hereditary* if the distance function in every connected induced subgraph of G is the same as in G itself.

[2] A *dominating pair* in a graph G is a pair of vertices in G such that for every path P connecting them in G it holds that every vertex in G is either in P or adjacent to a vertex in P.

done with minor modifications of the implementation of LexBFS, without computing the complement of a graph G.

3.5 Lexicographic Depth First Search

In previous section we introduced the lexicographic instance of BFS, so a natural question that arises is whether there exists a lexicographic restriction of DFS. A *Lexicographic Depth First Search* (LexDFS) was introduced in 2008 by Corneil and Krueger [15] and represents a special variant of Depth First Search. We could see that at each step of iteration of LexBFS a label of every unvisited vertex is updated so that the number of the chosen vertex is appended to the current label of the proposed vertex. In that way we get the labels that contain the numbers in the decreasing order, and at every step of iteration we take a vertex with lexicographically largest label. This means that the next visited vertex is always a vertex that has a neighbor visited as early as possible. In DFS, however, things are different, and we always choose a vertex that is adjacent to as recently numbered vertex as possible. In the LexDFS algorithm, the numbering of vertices starts with 1, and at each step of iteration we *prepend* the current number to the label of all unvisited neighbors of a current vertex. Similarly as in LexBFS, we take the vertex with lexicographically largest label to be visited next (see Algorithm 5).

Algorithm 5: Lexicographic Depth First Search.

Input: Connected graph G, and a vertex $s \in V(G)$.
Output: A vertex ordering σ.

```
1 begin
2     foreach v ∈ V do ℓ(v) = ∅;
3     ℓ(s) = {0}; n = |V(G)|
4     for i ← 1 to n do
5         v ← unnumbered vertex with lexicographically largest label ℓ(v);
6         σ(v) ← i;
7         foreach unnumbered neighbor w of v do
8             prepend i to ℓ(w)
9     return σ
```

Example 3.10. Consider the graph G from Fig. 11. Let us compute a LexDFS ordering σ of G starting in vertex a. Then $\sigma(a) = 1$ and labels are $\ell(b) = \ell(c) = \ell(d) = \{1\}$. We can choose any among vertices b, c, d, so let $\sigma(b) = 2$. Then the labels are $\ell(c) = \{21\}$, $\ell(d) = \{1\}$, and $\ell(e) = \ell(g) = 2$, so we have that $\sigma(c) = 3$. New labels are $\ell(d) = \{31\}$, $\ell(e) = \ell(g) = \{32\}$, $\ell(f) = \{3\}$, and lexicographic the largest is $\{32\}$, so we can

chose among e and g. Let $\sigma(g) = 4$. In the next step we get labels $\ell(d) = \{431\}$, $\ell(e) = \{432\}$, $\ell(f) = \{43\}$, so $\sigma(e) = 5$. After e, we have two unvisited vertices, with labels $\ell(f) = \{43\}$, $\ell(d) = \{431\}$, so we visit d, and finally f. The resulting ordering is $\sigma = (a, b, c, g, e, d, f)$. The iteration steps are presented in Fig. 12.

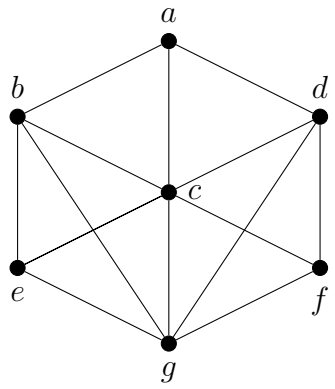


Figure 11: The ordering $\sigma = (a, b, c, g, e, d, f)$ is a LexDFS ordering of a graph G.

	Step 1		Step 2		Step 3		Step 4		Step 5		Step 6	
a	1	$\emptyset$	1	$\emptyset$	1	$\emptyset$	1	$\emptyset$	1	$\emptyset$	1	$\emptyset$
b		$\{1\}$	2	$\{1\}$	2	$\{1\}$	2	$\{1\}$	2	$\{1\}$	2	$\{1\}$
c		$\{1\}$		$\{21\}$	3	$\{21\}$	3	$\{21\}$	3	$\{21\}$	3	$\{21\}$
d		$\{1\}$		$\{1\}$		$\{31\}$		$\{431\}$		$\{431\}$	6	$\{431\}$
e		$\emptyset$		$\{2\}$		$\{32\}$		$\{432\}$	5	$\{432\}$	5	$\{432\}$
f		$\emptyset$		$\emptyset$		$\{3\}$		$\{43\}$		$\{43\}$	7	$\{643\}$
g		$\emptyset$		$\{2\}$		$\{32\}$	4	$\{32\}$	4	$\{32\}$	4	$\{32\}$

Figure 12: The iteration steps of LexDFS in Example 3.10. In each step the red column represents the numbers and the black one the labels.

Similarly as in case of LexBFS, the LexDFS orderings admit a characterization that enables the proofs of correctness of algorithms that use LexDFS at some step of execution. The characterization of LexDFS orderings is given in the following theorem (see Fig. 13).

Theorem 3.11 (Corneil and Krueger [15])**.** *An ordering σ of V is a LexDFS ordering if and only if the following holds: if $a <_\sigma b <_\sigma c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d such that $a <_\sigma d <_\sigma b$, $db \in E$, and $dc \notin E$.*

Proof. Let σ be a LexDFS ordering of a graph $G = (V, E)$. We want to prove that σ satifies the desired property. Let a, b, c be arbitrary vertices satisfying $a <_\sigma b <_\sigma c$, $ac \in E$, and $ab \notin E$. By Algorithm 5 it follows that in the moment when a was added to σ, the label of c was updated by adding the number of vertex a to its beginning,

while the label of b stayed the same. But we have that $b <_\sigma c$, so it has to be that the label of b is lexicographically smaller than the label of c. Labels in LexDFS are decreasing sequences of numbers, so there must be some number larger that $\sigma(a)$ that is added to the beginning of label of b, and not to the label of c. It means that there is a vertex d with $\sigma(d) > \sigma(a)$, such that $db \in E$ and $dc \notin E$. Clearly, it must hold that $\sigma(d) < \sigma(b)$, since otherwise d does not influence the choice of b. The vertices a, b, c were arbitrary, so the statement follows.

For a proof of the other direction, let $\sigma = (v_1, \ldots, v_n)$ be an ordering of vertices in G, satisfying that for any $a <_\sigma b <_\sigma c$ such that $ac \in E$ and $ab \notin E$ there exists a vertex d such that $a <_\sigma d <_\sigma b$ and $db \in E$ and $dc \notin E$ (call this property property (P)). Assume that σ is not a LexDFS ordering of G, and let $(v_1, v_2, \ldots, v_k)$, $k < n$ be a maximal initial segment of σ that can be obtained by LexDFS. Then vertex v_{k+1} cannot be chosen next, and by Algorithm 5 it follows that after vertex v_k was chosen, there is a vertex v_i, $i > k+1$ such that v_i has a lexicographically larger label than v_{k+1}. This implies that there is a vertex v_j, $j \leq k$, that is a neighbor of v_i and non-neighbor of v_{k+1}. Let v_j be the rightmost such vertex in σ.

Consider now the vertices v_j, v_{k+1}, v_i. We know that $v_j <_\sigma v_{k+1} <_\sigma v_i$, and $v_j v_i \in E$, while $v_j v_{k+1} \notin E$. We assumed at the beginning that σ satisfies property (P), and applied for the mentioned vertices, this implies that there exists a vertex d, $v_j <_\sigma d <_\sigma v_{k+1}$ such that $dv_{k+1} \in E$ and $dv_i \notin E$. But v_j was the rightmost vertex that is adjacent to v_i and non-adjacent to v_{k+1}, so every vertex on the right of v_j (that is, vertex between v_j and v_{k+1}) that is a neighbor of v_i is also a neighbor of v_{k+1}. This means that after adding a vertex d to σ, the label of v_{k+1} enlarges, while the label of v_i stays the same. Then the label of v_i cannot be lexicographically larger than the label of v_{k+1} and thus v_{k+1} has to be chosen before v_i in an LexDFS extension of $v_i, \ldots, v_k$; a contradiction with the maximality of k. The statement follows. □

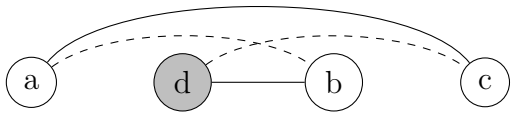


Figure 13: A vertex ordering corresponding to Theorem 3.11. Solid lines represent edges, dashed lines represent non-edges.

Implementation and applications

There is no known linear-time implementation of LexDFS algorithm. So far the best known LexDFS implementation was given by Krueger, and it takes $\mathcal{O}(\min\{n^2, n + m \log\log n\})$ time [30]. In the literature we can find references to the unpublished

work of Spinrad with $\mathcal{O}(m \log\log n)$-implementation of LexDFS [39]. A partition refinement approach in LexDFS would partition the vertices of a graph with respect to the adjacency relations. Starting at some vertex v_1, we would have classes of neighbors of v_1 and non-neighbors of v_1, respectively. In the next step we proceed with some new pivot, say v_2, and we have the following partition classes, in that order: $N_G(v_1) \cap N_G(v_2)$, $N_G(v_2) \cap \overline{N_G(v_1)}$, $N_G(v_1) \cap \overline{N_G(v_2)}$, $\overline{N_G(v_1)} \cap \overline{N_G(v_2)}$. It is clear that we have to reorder partition classes at each step, and up to now the linear-time implementation of that algorithm is not known [27].

In the paper that introduced LexDFS, there was no real algorithmic application of LexDFS proposed. However, just a few years later, LexDFS found many applications in graph algorithms for particular graph classes. The very first such application was given by Corneil et al [13], where LexDFS was used to find a minimum cardinality set of vertex disjoint paths in a graph G that cover the vertices of G (also known as the *minimum path cover problem*) on cocomparability graphs. This problem is a generalization of the Hamiltonian path problem. After that, LexDFS was used to develop a various algorithms on cocomparability graphs, such as computing a maximum cardinality independent set [14], a minimum clique cover [14], and a longest path [34]. An interesting fact is that all these algorithms rely on the same idea: in the first part of algorithm we compute a cocomparability ordering of a graph and preprocess it with LexDFS, while in the second part we extend or slightly modify the existing linear-time algorithms that solve the particular problem on the class of interval graphs. The time complexity of above mentioned algorithms was forced by the non-linearity of LexDFS, until a linear-time implementation of LexDFS for cocomparability graphs was given by Köhler and Mouatadid in [27]. LexDFS (similarly as LexBFS) computes the perfect elimination orders on chordal graphs and thus has some applications also in that class of graphs. In particular, it can be used to find a minimum colorings as well as all minimal separators and all maximal cliques in chordal graphs [43]. A linear-time implementation of LexDFS for chordal graphs was given recently by Beisegel et al. [5].

3.6 Maximum Cardinality Search

Maximum Cardinality Search (MCS) was introduced in 1984 by Tarjan and Yannakakis [42] as a simple search paradigm that has some common features with LexBFS. The main application of LexBFS was the recognition of chordal graphs, and they noticed that in order to use it for the recognition of chordal graphs, it is not necessary to store the order of already visited neighbors of a vertex. The authors managed to define a conceptually and computationally simpler method where the label of a vertex is just a number of already visited neighbors and at each step of iteration we choose a vertex

with maximum label (see Algorithm 6). This means that at each step of iteration we choose a vertex with the largest number of already visited neighbors, that is, a vertex that has the maximum number of already visited neighbors. For the implementation of MCS we have to store just the number of visited neighbors for each vertex, so it is clear that MCS can be implemented in linear time.

Algorithm 6: Maximum Cardinality Search.

Input: Connected graph G, and a vertex $s \in V(G)$.

Output: A vertex ordering σ.

```
1 begin
2     assign label 0 to all vertices;
3     ℓ(s) ← 0; n = |V(G)|;
4     for i ← 1 to n do
5         pick an unnumbered vertex v with maximum label;
6         σ(v) ← i;
7         foreach unnumbered vertex w ∈ N(v) do
8             ℓ(w) = ℓ(w) + 1;
9     return σ
```

Example 3.12. Let G be the graph from Fig. 14 and let us find a MCS ordering of G. Starting in vertex a we have that $\sigma(a) = 1$ and the labels of neighbors of a increase by 1, so the new labels are $\ell(b) = \ell(c) = \ell(d) = 1$. All these vertices have the same label, so we can choose arbitrarily among them; assume we take vertex b. Then $\sigma(b) = 2$ and labels are $\ell(c) = \ell(d) = \ell(e) = 1$. Let $\sigma(c) = 3$. New labels are $\ell(e) = 2$, $\ell(d) = \ell(f) = 1$, and we have to choose vertex e. It follows that $\sigma(e) = 4$ and updated labels are $\ell(g) = \ell(f) = \ell(d) = 1$. We can take any vertex among them, so assume we take vertex d: $\sigma(d) = 5$. Then we have $\ell(f) = 2$ and $\ell(g) = 1$, so the next visited vertex is f, and g is visited at the end. The resulting MCS order is $\sigma = (a, b, c, e, d, f, g)$.

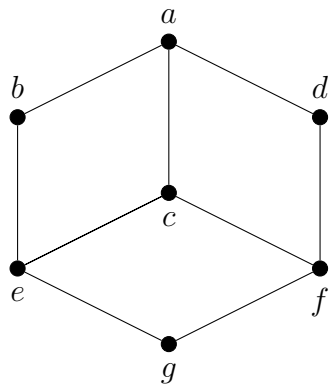


Figure 14: The ordering $\sigma = (a, b, c, e, d, f, g)$ is an MCS ordering of G.

There is a general rule that characterizes the orderings produces by MCS on a graph, and we present it in the following theorem (see Fig. 15).

Theorem 3.13 (Brändstadt et al. [7]). *An ordering σ of V is a MNS ordering if and only if the following holds: if $\{a_1, \ldots, a_k\} <_\sigma b <_\sigma c$, where $a_l, \ldots, a_k$ are pairwise distinct vertices, and $a_ic \in E$ and $a_ib \notin E$, for all $i = 1, \ldots, k$, then there are pairwise distinct vertices, $d_l, \ldots, d_k$ such that $d_i <_\sigma b$, $d_ib \in E$, and $d_ic \notin E$, for all $i = 1, \ldots, k$.*

Proof. Let σ be a MCS ordering, and let $\{a_1, \ldots, a_k\} <_\sigma b <_\sigma c$, where $a_l, \ldots, a_k$ are pairwise distinct vertices and $a_ic \in E$ and $a_ib \notin E$, for all $i = 1, \ldots, k$. We know that b appears before c in σ, so by Algorithm 6 it follows that in the moment just before b was added to σ the label of b was not smaller than the label of c. Since the label in MCS represents the number of already visited neighbors, it follows that the number of neighbors of b that appear before b in σ is at least as large as the number of neighbors of c that appear before b (since otherwise the algorithm would take c instead of b). Vertices $a_1, \ldots, a_k$ are neighbors of c that appear before b in σ, so also the number of neighbors of b that appear before b in σ must be at least k. It follows that there are paiwise distinct vertices $d_1, \ldots, d_k$ that appear before b in σ and are adjacent with b, as we wanted to show.

For a proof of the other direction, let us define a property P to be the following property: if $\{a_1, \ldots, a_k\} <_\sigma b <_\sigma c$, where $a_l, \ldots, a_k$ are pairwise distinct vertices, and $a_ic \in E$ and $a_ib \notin E$, for all $i = 1, \ldots, k$, then there are pairwise distinct vertices, $d_l, \ldots, d_k$ such that $d_i <_\sigma b$, $d_ib \in E$ and $d_ic \notin E$, for all $i = 1, \ldots, k$.

Let σ be an ordering of vertices in G satisfying property P, and let $(v_1, \ldots, v_k)$, $k < n$ be a maximal initial segment of σ that can be constructed using the MCS algorithm. Then vertex v_{k+1} cannot be chosen using the MCS procedure, and let v_i, $i > k+1$, be a vertex that can be chosen after $(v_1, \ldots, v_k)$ by Algorithm 6. Then after the vertices $v_1, \ldots, v_k$ are chosen in σ, the label of v_i is larger that the label of v_{k+1}, meaning that vertex v_i has more neighbors in set $\{v_1, \ldots, v_k\}$ than vertex v_{k+1} does. Let U and W be subsets of $\{v_1, \ldots, v_k\}$ defined as follows:

$$U = \{v_j \mid j \in \{1, \ldots, k\}, v_j \in N_G(v_{k+1}) \setminus N_G(v_i)\},$$

$$W = \{v_j \mid j \in \{1, \ldots, k\}, v_j \in N_G(v_i) \setminus N_G(v_{k+1})\}.$$

From the above discussion it follows that $|W| > |U|$. Let $v_{i_1}, \ldots, v_{i_p}$ be pairwise distinct vertices in W. All vertices in W appear before v_{k+1} in σ, and are nonadjacent with v_{k+1} in G. Moreover, all vertices in W are adjacent with v_i. Applying property P on vertices $v_{i_1}, \ldots, v_{i_p}$, v_{k+1}, v_i, we get that there are pairwise distinct vertices, $d_{i_1}, \ldots, d_{i_p}$ such that $d_j <_\sigma v_{k+1}$, $d_jv_{k+1} \in E$ and $d_jv_i \notin E$, for all $j \in \{i_1, \ldots, i_p\}$. Clearly, all vertices $d_{i_1}, \ldots, d_{i_p}$ are elements of U, implying that $|U| \geq |W|$, contrary to our assumption.

It follows that v_i does not appear before v_{k+1} in an MCS extension of $(v_1, \ldots, v_k)$, and thus σ is a MCS ordering, as we wanted to show. □

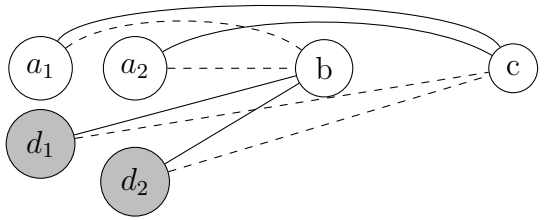


Figure 15: A vertex ordering corresponding to Theorem 3.13. Solid lines represent edges, dashed lines represent non-edges.

A common feature of MCS and LexBFS is that the orderings produced by both of them are the reverse of perfect elimination orderings of a chordal graph [30].

Theorem 3.14 (Tarjan and Yannakakis [42])**.** *A graph G is chordal if and only if every MCS ordering of the vertices in G is the reverse of a perfect elimination scheme of G.*

It is not, however, true that every PEO of a chordal graph can be obtained by MCS or LexBFS, as can be seen in the following example.

Example 3.15. Let G be a graph with vertex set $\{a, b, c, d\}$ and edge set $\{ab, bc, ca, cd\}$. This graph is known as a 3-pan, or a paw, and it is clear that G is a chordal graph. Let $\sigma = (c, a, d, b)$. It holds that the reverse of σ is a perfect elimination ordering of G, while σ is not a LexBFS nor MCS ordering of G. The reverse of σ is (b, d, a, c) and we can see that b is simplicial vertex in the whole graph, d is simplicial in $G - b$, a is simplicial in $G - \{b, d\}$, so by definition it is a perfect elimination ordering of G. Further, observe that after visiting vertices c and a in σ, the labels of vertices b and d are $\ell(b) = 2$ and $\ell(d) = 1$ ($\ell(b) = \{43\}$ and $\ell(d) = \{4\}$) for MNS (LexBFS, resp.). In both cases we have to select vertex b before vertex d, so σ is neither MCS nor LexBFS ordering of G.

Also, there are MCS orderings that cannot be obtained by LexBFS, and, vice versa, there are LexBFS orderings that cannot be obtained by MCS. The ordering $\sigma = (a, b, c, e, d, f, g)$ of the graph from Fig. 14 is an MCS ordering, and not a LexBFS ordering – starting a LexBFS in a cannot happen that $\sigma(e) < \sigma(d)$. Similarly, the ordering $\sigma = (a, c, b, d, e, f, g)$ is a LexBFS ordering of the same graph, while it is not an MNS ordering – after visiting vertices a, b, and c, the label of vertex e is larger than the label of vertex d, and it cannot happen in an MCS ordering that $\sigma(d) < \sigma(e)$.

From Theorem 3.14 it is clear that MCS found its application in recognition of chordal graphs. Besides that, the original paper by Tarjan and Yannakakis presents its applications to testing acyclity of hypergraphs, and how to selectively reduce an acyclic hypergraph to more efficiently compute database queries. MCS can also be used to derive a bounds for a treewidth of a graph. The *visited degree* of an MCS ordering of a graph is the maximum visited degree over all vertices in the graph. The maximum visited degree over all MCS orderings of graph is called its *maximum visited degree*. Lucena showed that the treewidth of a graph is at least its maximum visited degree [32].

3.7 Maximal Neighborhood Search

Maximal Neighborhood Search (MNS) was introduced by Corneil and Krueger in 2008 as a common generalization of LexBFS, LexDFS, and MCS [15]. Unlike the search methods presented in previous sections, which use strings or numbers as labels of vertices, the MNS algorithm uses sets of integers as labels, and at every step of iteration chooses a vertex with maximal label under set inclusion (see Algorithm 7). It follows that in MNS the labels are not totally ordered, so it can happen that few different labels are maximal in the same time. In the already presented search methods, where labels were strings or numbers, this was not possible, since strings and numbers form totally ordered sets.

Algorithm 7: Maximal Neighborhood Search.

Input: Connected graph G, and a vertex $s \in V(G)$.
Output: A vertex ordering σ.

```
1  begin
2      assign label ∅ to all vertices;
3      n = |V(G)|;
4      ℓ(s) ← {1};
5      for i ← 1 to n do
6          pick an unnumbered vertex v with maximal label under set inclusion;
7          σ(v) ← i;
8          foreach unnumbered vertex w ∈ N(v) do
9              add i to ℓ(w);
10     return σ
```

It is not difficult to see that we can obtain LexBFS, LexDFS, or MCS from MNS by precising more specifically the rule on how to make a selection among the vertices with incomparable neigborhoods. In case of LexBFS (resp. LexDFS), we choose the

vertex adjacent to as many earliest (resp. most recently) visited neighbors as possible, while in case of MCS we choose a vertex with maximal label under set cardinality (which is clearly maximal under set inclusion as well). MNS can be seen as some kind of lexicographic instance of a generic search, and is sometimes in literature referred to as the *Lexicographic Generic Search* [30]. The following theorem characterizes the MNS orderings (see Fig. 16).

Theorem 3.16 (Corneil and Krueger [15])**.** *An ordering σ of V is an MNS ordering if and only if the following statement holds: If $a <_\sigma b <_\sigma c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d with $d <_\sigma b$, $db \in E$, and $dc \notin E$.*

Proof. Let σ be an MNS ordering of vertices in a graph G, and let $a <_\sigma b <_\sigma c$ be vertices of G such that $ac \in E$ and $ab \notin E$. We know that b appears before c in σ, so by Algorithm 7 it follows that in the moment just before b was added to σ the label of b was not a proper subset of a label of c. Since the label in MCS represents the set of numbers corresponding to already visited neighbors, it follows that there is a vertex d whose number is an element of the label of b and not of the label of c. Equivalently, there is a vertex d that appears before b in σ such that $db \in E(G)$ and $dc \notin E(G)$, as we wanted to show.

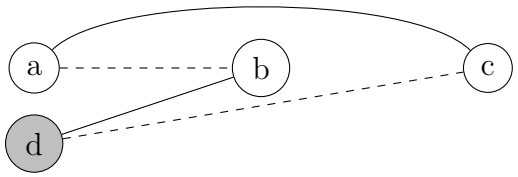


Figure 16: A vertex ordering corresponding to Theorem 3.16. Solid lines represent edges, dashed lines represent non-edges.

Let us now prove the other direction. Let $\sigma = (v_1, \ldots, v_n)$ be an ordering of vertices in G, satisfying that for any $a <_\sigma b <_\sigma c$ such that $ac \in E$ and $ab \notin E$ there exists a vertex d such that $d <_\sigma b$ and $db \in E$, $dc \notin E$ (call this property property (P)). Assume that σ is not an MNS ordering of G, and let $(v_1, v_2, \ldots, v_k)$, $k < n$ be a maximal initial segment of σ that can be obtained by MNS. Then vertex v_{k+1} cannot be chosen next, and by Algorithm 7 it follows that after vertex v_k was chosen, there was a vertex v_i, $i > k+1$, such that v_i has a larger label than v_{k+1} with respect to set inclusion. Equivalently, the label of v_{k+1} is a proper subset of the label of v_i. This means that all among the vertices in $\{v_1, \ldots, v_k\}$ all vertices adjacent with v_{k+1} are also adjacent with v_i. Moreover, there is a vertex v_j, $j \leq k$, that is a neighbor of v_i and a non-neighbor of v_{k+1}. Observe that $v_j <_\sigma v_{k+1} <_\sigma v_i$. Property P applied on vertices v_j, v_{k+1} and v_i implies that there is a vertex d that appears before v_{k+1} in σ and

safisfies $dv_{k+1} \in E(G)$, $dv_i \notin E(G)$. It must be that d is one among vertices $v_1, \ldots, v_k$. We have that $dv_{k+1} \in E(G)$, and $dv_i \in E(G)$, a contradiction. It follows that σ is a MNS ordering of G. □

MNS as a search paradigm. The characterization of MNS ordering from Theorem 3.16 was given in the work by Tarjan and Yannakakis in 1984 [42], where is was referred to as *property* $\mathcal{P}$. They observed that both LexBFS and MCS satisfy property $\mathcal{P}$ and that any ordering with property $\mathcal{P}$ is the reverse of a perfect elimination ordering of a chordal graph. In the mentioned paper no search paradigm was defined using that property. The converse is not true - there exist perfect elimination orderings of chordal graphs that are neither LexBFS nor MNS orderings (see Example 3.15).

The first appearance of some instance of MNS as a search paradigm (without its characterization) can be found in the work by Shier related to generating all the perfect elimination orderings of a chordal graph [38]. There he defined the search paradigms named Maximal Element in Component (MEC) and Maximal Cardinality Component (MCC). These paradigms are of special interest on the class of chordal graphs, since they are in one-to-one correspondence with PEOs of chordal graphs. We describe both of them below for a chordal graph G.

Maximal Element in Component (MEC)

Input: A chordal graph $G = (V, E)$ with $n = |V| > 1$.

Output: An ordering σ.

1. Let $S_n = \{u\}$ consist of an arbitrary vertex $u \in V$; set $\sigma(u) = n$ and $k = n$.
2. Select an element x in some connected component C of $V - S_k$ such that $N_k(x) = N_G(x) \cap S_k$ is maximal among all vertices in C.
3. Define $\sigma(x) = k - 1$. If $k = 2$, then STOP. Otherwise, define $S_{k-1} = S_k \cup \{x\}$, set $k = k - 1$ and go to Step 2.

Maximal Cardinality Component (MCC)

Input: A chordal graph $G = (V, E)$ with $n = |V| > 1$.

Output: An ordering σ.

1. Let $S_n = \{u\}$ consist of an arbitrary vertex $u \in V$; set $\sigma(n) = u$ and $k = n$.
2. Select an element x in some connected component C of $V - S_k$ such that $|N_k(x)| = |N_G(x) \cap S_k|$ is maximum among all vertices in C.
3. Define $\sigma(k - 1) = x$. If $k = 2$, then STOP. Otherwise, define $S_{k-1} = S_k \cup \{x\}$, set $k = k - 1$ and go to Step 2.

Shier observed that any LexBFS and MCS ordering of a chordal graph can be generated by MEC and MCC schemes, so these two represent more general search paradigms. It turns out that using these paradigms one can generate all perfect elimination schemes of a chordal graph, which is not true for MNS.

Theorem 3.17 (Shier [38])**.** *Let σ be an ordering of a chordal graph G. Then the following statements are equivalent:*

1. *σ is an MEC ordering.*
2. *σ is an MCC ordering.*
3. *σ is a PEO.*

4 Relations Among Search Orderings

We already saw that certain search methods can be seen as a restrictions, or special instances of some more general search methods (see Fig. 17). For example, LexBFS is a special instance of BFS and MNS, while LexDFS is a special instance of MNS and DFS. Similarly, it is known that MCS is a special instance of MNS. This means that, for example, every LexBFS ordering is also BFS and MNS ordering at the same time. The converse, however, is not true. There exist orderings that are BFS and MNS, but not LexBFS, or that are DFS and MNS but not LexDFS, or that are MNS and not MCS.

Example 4.1. Let G be a graph consisting of a path on vertices b, c, d, e, and a universal vertex a. Vertex a is universal in G, so any ordering starting in a is a BFS ordering. Let $\sigma = (a, c, d, e, b)$. Observe that σ is a MNS ordering of G. We start in a and after visiting a and c we can choose between b and d. When we proceed with d, the label of e contains d, while that of b does not, so the label of e is not a subset of a label of b, and we can visit e before b. It is not difficult to justify that σ cannot be a LexBFS ordering, since once we visit vertices a and c, in the rest of the graph the neighbors of c must be visited before non-neighbors of c. It follows that e should be visited after b; contrary to the definition of σ.

The above example shows that there is a graph G and a BFS and MNS ordering of G that is not a LexBFS ordering of G. Similarly we can show that there exists a DFS and MNS ordering of some graph G that is not a LexDFS ordering of the same graph.

Example 4.2. Let G be a graph consisting of a 4-cycle a, b, c, d and a vertex e adjacent to vertices b and c (in graph theory this graph is known under the name "house"). Let $\sigma = (a, b, c, d, e)$. Clearly, σ is a DFS ordering of G. It is true that σ is MNS ordering of G, while it is is not a LexDFS ordering of G. After visiting vertices a, b and c, we have to choose among vertex d with label 31 and vertex e with label 32. Clearly, e has the lexicographically larger label, so it must be visited before d, contrary to the definition of σ.

In the following sections we discuss the relations depicted by red lines in Fig. 17.

We know that all the lines in Fig. 17 represent relations of inclusion, so one may wonder under which conditions on a graph G the particular inclusion is not proper. In this section we say that two search methods are *equivalent on a graph* G, if the sets of vertex orderings produced by both of them are the same. Similarly, two graph search methods are *equivalent on a graph class* $\mathcal{G}$ if they are equivalent on every G in $\mathcal{G}$. We give some necessary and/or sufficient conditions on a graph G in order to obtain the equivalence relation on depicted red lines.

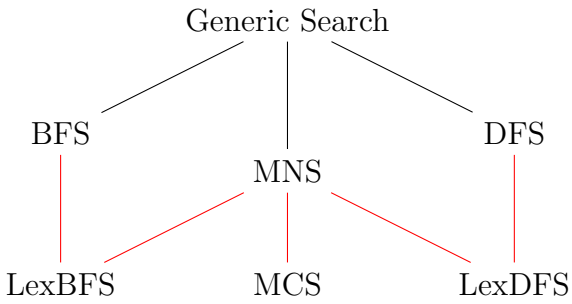


Figure 17: Relations between various graph search methods.

4.1 Breadth First Search vs Lexicographic Breadth First Search

Graph search methods in general do not have the hereditary property. Let G be a graph with a search ordering σ of particular type, and let H be an induced subgraph of G. It is not true that σ^*, the sequence obtained from σ by deleting vertices that are not in H represents a search ordering of the same type of H, as can be seen in the following example.

Example 4.3. Let G be a cycle on 5 vertices, and let us denote its vertices by v_1, v_2, v_3, v_4, v_5 in the cyclic order. It is not difficult to see that $\sigma = (v_1, v_5, v_2, v_4, v_3)$ is a BFS ordering of G. Let H be the subgraph of G obtained by deletion of vertex v_5, and let σ^* be the ordering of vertices in H obtained from σ after deletion of v_5. Then $\sigma^* = (v_1, v_2, v_4, v_3)$ is not a valid BFS ordering of H.

From the above it follows that it could happen that there is an ordering of a graph H that is a BFS and not a LexBFS ordering, while in a graph G containing H as an induced subgraph this is not necessarily true. This means that the equivalence between BFS and LexBFS in G does not imply the same equivalence on every induced subgraph of G. In the following example we can see that a valid LexBFS ordering of G yields an ordering of its subgraph H that is BFS and not LexBFS.

Example 4.4. Let G be the graph from Fig. 18. After removing the vertex u from G we get a 6-pan G'. Observe that in G' we can find a BFS ordering $\sigma^* = (v_1, v_2, v_6, v_3, v_5, v, v_4)$ that is not a valid LexBFS ordering. If σ^* is a part of a valid BFS ordering σ of G, then we must visit u before visiting the non-neighbors of v_1, and after visiting vertices v_2 and v_6. Then it follows that $\sigma = (v_1, v_2, v_6, u, v_3, v_5, v, v_4)$ is a valid LexBFS ordering of G, so is not an example of ordering of G that is BFS and not LexBFS.

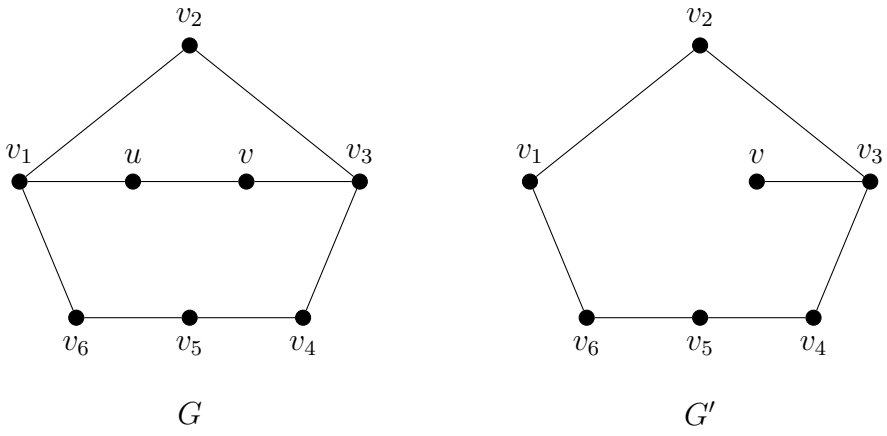


Figure 18: The ordering $\sigma = (1, 2, 6, u, 3, 5, v, 4)$ is a valid LexBFS ordering of G, while the ordering $\sigma^* = (1, 2, 6, 3, 5, v, 4)$ is not a valid LexBFS ordering of G'.

Despite both demotivating examples above, we identify certain graphs such that the equivalence between BFS and LexBFS does not hold in any graph containing them as an induced subgraph.

Example 4.5. Let G be a graph that contains a paw or a diamond as an induced subgraph. In this example we show that there is a BFS ordering of G that is not a LexBFS ordering of G (that is, BFS and LexBFS are not equivalent in G). The claim can be easily justified by giving a prefix of an order σ that is a BFS order and not a LexBFS order of G. Let H be a paw graph contained in G as an induced subgraph. Using the same notation as in Fig. 19 (left) we can define the BFS ordering σ of G starting in c, with first four vertices in σ being c, a, d, b, in that order. Similarly, if H is a diamond contained in G as induced subgraph, we can define a BFS ordering σ of G starting in c and visiting consecutively vertices b, d, a (Fig. 19 right). In both cases σ is a BFS ordering, since it starts with a vertex c and visits its neighbors. Also, σ cannot be a LexBFS ordering, since in both cases vertex a has label $\{n, n-1\}$, while d has a label n, so a should appear before d, no matter how the rest of σ is defined.

$\sigma_1 = (c, a, d, b)$ $\quad$ $\sigma_1 = (a, b, d, c)$
$\sigma_2 = (b, c, d, a)$ $\quad$ $\sigma_2 = (b, c, d, a)$

Figure 19: A paw (left) and a diamond (right) with corresponding search orderings σ_1 (σ_2) that are BFS and not LexBFS orderings (DFS and not LexDFS orderings, resp.).

If every BFS in G is also a LexBFS, then G does not contain a paw, or a diamond as an induced subgraph. If, additionally, G does not contain any pan graph as an induced subgraph, then we can show that every BFS is also a LexBFS, using the following lemmas. Observe that paw is 3-pan, so {paw, diamond, pan}-free graph is exactly the {diamond, pan}-free graph.

Lemma 4.6. *If a connected graph G does not contain a diamond or a pan as an induced subgraph, then G is either acyclic, or a cycle, or a complete graph, or a complete bipartite graph.*

Proof. Let G be a graph that does not contain a diamond or a pan as induced subgraph. From Theorem 2.2 (see Section 2.1) it follows that G is either a complete multipartite graph or a triangle-free graph.

Let first G be a complete multipartite graph, with partition classes $S_1, \ldots, S_k$. If all partition classes of G have one vertex, then G is a complete graph, so we may assume without loss of generality that $|S_1| \geq 2$. Let $x, y \in S_1$, $x \neq y$. If there are exactly two partition classes of G, then G is a complete bipartite graph. Assume that there are at least three partition classes in G, and let $z \in S_2$, $w \in S_3$. Then the vertices $\{x, y, z, w\}$ form a diamond in G; a contradiction.

Let now G be a triangle-free graph. If G does not contain any cycle, then we are done. Assume first that G contains a cycle of length at least five and let C be such a cycle in G. If $G = C$, we are done, so assume that there is a vertex v in $V(G) \setminus V(C)$ having a neighbor in C. If v has exactly one neighbor in C, then $V(C) \cup \{v\}$ induce a cycle with pendant vertex in G, so v has at least two neighbors in C. We know that G is triangle-free, so no two consecutive vertices of C are adjacent to v. Let $v_i, v_j \in C$, $i < j$ be neighbors of v such that $|j - i| = j - i$ is minimal. Then $vv_{i-1} \notin E(G)$ and vertices $v, v_{i-1}, v_i, v_{i+1}, \ldots, v_j$ form a cycle with pendant vertex, unless it holds that $v_{i-1}v_j \in E(G)$, that is, unless the vertices v_{i-1} and v_j are consecutive in C, meaning that the distance between v_i and v_j in C is equal to two and that C is a cycle on four

vertices. Our assumption was that C is a cycle on at least 5 vertices, so we have a contradiction. It follows that vertex v does not exist and $G = C$.

Assume now that any cycle in G contains exactly four vertices, and let C be such a cycle, with vertices v_1, v_2, v_3, v_4 in consecutive order. We know that G has no odd cycles, so G is bipartite graph. Also, we now show that C is a complete bipartite graph. Let F be a subgraph of G that contains C such that F is maximal complete bipartite subgraph of G, and let (A, B) be a bipartition of F. Without loss of generality we may assume that $v_1, v_3 \in A$ and $v_2, v_4 \in B$. If $G = F$, then G is a complete bipartite graph, and we are done. Graph G is connected, so we may assume that there is a vertex $v \in V(G) \setminus V(F)$ such that v has a neighbor in F. Let without loss of generality $u \in A$ be a neighbor of v. We know by definition of F that u is adjacent to all vertices in B, so it cannot be that v has a neighbor in B, since otherwise that neighbor together with vertices u and v would form a triangle in G. It follows that $(A, B \cup \{v\})$ is a bipartition of a bipartite graph, and from the maximality of F it follows that v has a non-neighbor in A. Let $x \in A$ be a non-neighbor of u. (Observe that it can happen that $\{x, u\} \cap \{v_1, v_3\} \neq \emptyset$.) Taking the vertices $\{x, u, v_2, v_4, v\}$ we get the forbidden C_4 with a pendant edge; a contradiction. It follows that $G = F$ and thus G is a complete bipartite graph, as we wanted to show. □

Lemma 4.7. *In the following graph classes every BFS ordering is a LexBFS ordering.*

i) cycles

ii) forests

iii) complete graphs

iv) complete bipartite graphs

Proof. We prove the lemma for each case separately.

i) Assume for a contradiction this is not true, and let G be a cycle with ordering σ that is a BFS ordering and not a LexBFS ordering. By Theorem 3.8 it follows that there are vertices $a <_\sigma b <_\sigma c$ such that $ab \notin E(G)$, $ac \in E(G)$ and for every $d' <_\sigma a$ it holds that either $d'b \notin E(G)$, or $d'c \in E(G)$. Similarly, from Theorem 3.5 it follows that there is a vertex $d <_\sigma$ such that $db \in E(G)$. Then it must be that $dc \in E(G)$, so b and c are both neighbors of d in G. We know that G is a cycle, so every vertex in G is of degree 2, and thus b and c are the only neighbors of d in G. Since σ is a BFS ordering, at every step it visits a neighbor of some already visited vertex, so it must be that $\sigma(d) = 1$. Then the neighbors of d are visited before non-neighbors of d, so vertices b and c must be

visited before a in the BFS ordering σ. This is a contradiction with the definition of a, b, c, so such an ordering σ does not exist, and every BFS ordering of G is also a LexBFS ordering of G.

ii) Let σ be a BFS ordering of a forest graph G, and let $\sigma(v) = 1$. If we do a LexBFS on G starting in v, at every step of iteration all the unvisited vertices have a label consisting just of one number - a number belonging to the parent of the unvisited vertex. Thus, the label of every vertex consists just of a number belonging to the first visited neighbor. It means that putting the vertices in a queue in BFS is exactly the same as ordering vertices with respect to the lexicographic maximal label, so σ is a LexBFS of G.

iii) If v is arbitrary vertex of a complete graph G, once the vertex v is visited, every unvisited vertex in G gets a label from v. It means that at iteration step of LexBFS all the unvisited vertices in G have the same label, so we can choose any among them. Any ordering of vertices of a complete graph is both a BFS and a LexBFS ordering.

iv) Let G be a complete bipartite graph with partition classes A and B, and let σ be a BFS ordering of G. Assume without loss of generality that $v \in A$ is a first vertex in ordering σ. BFS is a layered search on G, so after visiting v we visit all the neighbors of v in B. After that, we visit all the vertices that are on distance 2 from vertex v in G, and so on, until we visit all the vertices in G. This search is also a LexBFS search, since at every step of LexBFS the vertices of G in the same partition of $V(G)$ all have the same labels, and we can choose any among them. Also, all vertices that are on some distance i from v belong either to A or B, so σ is a LexBFS ordering of G.

□

It turns out that BFS and LexBFS are equivalent in a {diamond, pan}-free graph.

Corollary 4.8. *If G is a {diamond, pan}-free graph, then every BFS in G is also LexBFS.*

4.2 Depth First Search vs Lexicographic Depth First Search

In previous section we gave a characterization of graphs that are {diamond, pan}-free and using the structural result written there it is not difficult to show that every DFS

ordering is a LexDFS ordering in a {diamond, pan}-free graphs. In this section we use another approach to prove the equivalence between the proposed search methods, so we use the characterization of (Lex)DFS orderings (so-called "point conditions"). First we give a DFS version of Example 4.5.

Lemma 4.9. *If a graph G contains a paw or a diamond as an induced graph, then there is a DFS ordering of G that is not a LexDFS ordering of G (that is, DFS and LexDFS are not equivalent in G).*

Proof. The claim can be easily justified by giving a prefix of an order σ that is a DFS order and not a LexDFS order of a graph containing a paw or a diamond. Let G be a graph and let H be a paw graph, contained in G as an induced subgraph. Using the same notation as in Fig. 19 (left) we can define the DFS ordering σ of G starting in b, with first four vertices in σ being b, c, d, a, in that order. Similarly, if H is a diamond contained in G as induced subgraph, we can define the DFS ordering σ of G having the same prefix: starting in b and visiting consecutively vertices c, d, a (Fig. 19 right). In both cases σ is a DFS ordering, since it starts with a vertex b and traverses the graph as deep as possible. At the same time, σ cannot be a LexDFS ordering of G, as vertices c, d, a do not satisfy the condition from Theorem 3.11, and the claim follows. □

As we already know, the paw is defined as a 3-pan. In the following lemma we give a result showing that the eqivalence between DFS and LexDFS in G implies that G does not contain any pan as induced subgraph. This result generalizes the part of previous lemma that considered the existence of a paw graph in G.

Lemma 4.10. *If a graph G contains a pan as an induced subgraph, then there is a DFS ordering of G that is not a LexDFS ordering of G (that is, DFS and LexDFS are not equivalent in G).*

Proof. The claim can be easily justified by giving a prefix of an order σ that is a DFS order and not a LexDFS order of a graph containing a pan. Let G be a graph and let H be a pan contained in G as an induced subgraph. Let the vertices of H be denoted by $v_1, \ldots, v_n, v$, where vertices $v_1, v_2, \ldots, v_n$ form a cycle in this order, and v is a vertex of degree 1, adjacent to v_{n-1}. We can define a DFS ordering σ of G starting in v_1, with first n vertices in σ being $v_1, v_2, \ldots, v_{n-2}, v_{n-1}, v$, in that order. It is clear that σ is a DFS order, since it has a prefix that is a path, and continues traversing the graph G using DFS. At the same time we have that σ is not a LexDFS ordering of G. We know that the vertex v_n appears in σ after all other vertices from H. It follows that $v_1 <_\sigma v <_\sigma v_n$ with $v_1v_n \in E(G)$ and $v_1v \notin E(G)$. By Theorem 3.11 it follows that there is a vertex v_i: $v_1 <_\sigma v_i <_\sigma v$, $v_iv \in E(G)$ and $v_iv_n \notin E(G)$. But among the vertices that are visited before v in σ there is just a vertex v_{n-1} that is adjacent to v.

We have that $v_{n-1}v_n \in E(G)$, so the condition of Theorem 3.11 is not fulfilled, and σ is not a LexDFS ordering of G. □

Therefore, equivalence between DFS and LexDFS in a graph G implies that G is a {pan, diamond}-free graph. In the following lemma we show that this is also sufficient.

Lemma 4.11. *If a graph G does not contain a diamond, or a pan as an induced subgraph, then DFS and LexDFS are equivalent in G.*

Proof. Let $\mathcal{G}$ be a class of {diamond, pan}-free graphs. We want to prove that DFS and LexDFS are equivalent in $G \in \mathcal{G}$. Assume for contradiction this is not the case, and let G be a graph and σ an ordering of G that is DFS but not LexDFS ordering.

Since σ is a DFS ordering, it satisfies the characterization given in Theorem 3.6: if $a <_\sigma b <_\sigma c$ and $ac \in E$ and $ab \notin E$, then there exists a vertex d such that $a <_\sigma d <_\sigma b$ and $db \in E$. From Theorem 3.11 it follows that there exist vertices a, b, c in G such that $a <_\sigma b <_\sigma c$, $ab \notin E(G), ac \in E(G)$ and for all vertices d satisfying $a <_\sigma d <_\sigma b$ it holds that either $dc \in E(G)$, or $db \notin E(G)$.

Let $a <_\sigma b <_\sigma c$ be leftmost vertices that do not satisfy the characterization of LexDFS ordering σ given in Theorem 3.11. We know that σ is DFS ordering of G, so there exists a vertex d_1 such that $a <_\sigma d_1 <_\sigma b$ and $d_1b \in E(G)$. Then it follows that $d_1c \in E(G)$. Also, we have that $ad_1 \notin E(G)$ and $bc \notin E(G)$, since otherwise we get a pan or a diamond.

Consider now the vertices a, d_1, c. It holds that $a <_\sigma d_1 <_\sigma c$, with $ac \in E(G)$ and $ad_1 \notin E(G)$. Since $d_1 <_\sigma b$, these vertices satisfy the condition from the LexDFS ordering characterization, so there exists a vertex d_2 such that $a <_\sigma d_2 <_\sigma d_1$, and $d_2d_1 \in E(G)$, $d_2c \notin E(G)$. If $d_2b \in E(G)$, then the vertices d_2, d_1, b, c form a 3-pan. If $ad_2 \in E(G)$, then the vertices a, d_2, d_1, b, c form a 4-pan. Hence, it follows that $ad_2 \notin E(G)$ and $bd_2 \notin E(G)$. Now we can continue this process by considering the vertices a, d_2, c and apply the characterization of the LexDFS ordering. Let $d_1, d_2, \ldots, d_k$ be a sequence of vertices defined in the following way: given a triple $a <_\sigma d_i <_\sigma c$ such that $ad_i \notin E(G)$, $ac \in E(G)$, d_{i+1} is a vertex satisfying the conditions: $a <_\sigma d_{i+1} <_\sigma d_i$, $d_{i+1}d_i \in E(G)$, and $d_{i+1}c \notin E(G)$. Let k be the maximum number of such vertices. We know that the number of vertices between a and c is finite, so k is a finite number. We show the following claims:

i) $d_id_{i+1} \in E(G)$, for all $i \in \{2, \ldots, k\}$ – this is true by definition of vertices d_i,

ii) $d_{i+1}c \notin E(G)$, for all $i \in \{2, \ldots, k\}$ – this is true by definition of vertices d_i,

iii) $d_ia \notin E(G)$, for all $i \in \{1, 2, \ldots, k-1\}$ – this is true by definition of vertices d_i,

iv) $d_k a \in E(G)$ – if this were not true, then we could continue the process, and d_k is not the last vertex in this sequence

v) $d_i d_j \notin E(G)$, for all $i, j \in \{1, \ldots, k\}$, such that $|i - j| \geq 2$. Assume the opposite: let d_i and d_j be adjacent vertices with $|i - j| \geq 2$, such that $|i - j|$ is minimum and among all pairs i, j satisfying this minimality condition, let i, j be the smallest possible (equivalently, the rightmost in the ordering σ). Without loss of generality we may assume that $j < i$. From the minimality of $|i - j|$ it follows that the vertices $d_j, d_{j+1}, \ldots, d_i$ form an induced cycle in G. If $j = 1$, then the vertices $\{d_j, d_{j+1}, \ldots, d_i, c\}$ form a pan in G. Similarly, if $i = k$, then the vertices $\{d_j, d_{j+1}, \ldots, d_i, a\}$ form a pan in G. It follows that $j > 1$ and $i < k$. Consider now the vertex d_{j-1}. By the way we chose i and j it follows that $d_{j-1} d_\ell \notin E(G)$ for all $\ell \in \{d_{j+1}, \ldots, d_{i-1}\}$. If $d_{j-1} d_i \notin E(G)$, then the vertices $\{d_{j-1}, d_j, d_{j+1}, \ldots, d_i\}$ form a pan in G. If $d_{j-1} d_i \in E(G)$, we consider two cases. First, if $i - j = 2$, then the vertices $\{d_i, d_{i-1}, d_{i-2} = d_j, d_{i-3} = d_{j-1}\}$ form a diamond. Second, if $i - j > 2$, then the vertices $\{d_{j-1}, d_j, d_i, d_{i-1}\}$ form a 3-pan. In both cases we get a contradiction with the definition of G, meaning that such an edge $d_i d_j$ cannot exist in G.

vi) $d_i b \notin E(G)$, for all $i \in \{2, \ldots, k\}$. Assume for contradiction that j is a minimal value in $\{2, \ldots, k\}$ such that $d_j b \in E(G)$. Then the vertices $\{d_1, \ldots, d_j, b, c\}$ form a pan in G; a contradiction.

Consider now the vertices $\{d_1, \ldots, d_k, a, c, b\}$. From the above claims it follows that they form a pan, where b is a vertex of degree one. This is a contradiction with the definition of G. It follows that the vertices $d_1, \ldots, d_k$ defined as above cannot exist, so σ is a LexDFS ordering of G, as we wanted to show. □

Corollary 4.12. *Every DFS ordering of a graph G is a LexDFS ordering of G if and only if G is a {diamond, pan }-free graph.*

4.3 Maximal Neighborhood Search vs Maximum Cardinality Search

In this section we give some examples of graphs for which there exists a MNS ordering that is not a MCS ordering. On Fig. 20 there are graphs and corresponding MNS orderings that are not MCS orderings. It is not, however, clear, whether these graphs are forbidden as induced subgraph in a larger graph satisfying the requirement that every MNS ordering is MCS ordering. This means that there could exist a graph G

and its induced subgraph H such that G satisfies the MNS and MCS equivalence, while H does not. Let, for example, H be the top left graph in Fig. 20. It is not difficult to see that the ordering $\sigma = (b, c, d, a, e)$ is a valid MNS ordering, but not a valid MCS ordering of H. This is true since after taking vertices b, c, d, in MCS we must take vertex e, and not a. If G is a graph consisting of H and additional vertex f adjacent with a, b, c, d, then $N_G(e) \subset N_G(f)$ and f must be visited before e in MNS. It follows that after taking vertices b, c, d, we must take f, and then we can take any vertex among a and e. In both cases we get an ordering that is both MNS and MCS. It seems that for any ordering σ of H that is an MNS and not an MCS ordering, we can find some graph G such that H is an induced subgraph of G, and no ordering that is MNS and not MCS ordering of G can have the same prefix as σ. Unfortunately, it is not clear how this construction of a graph G influences the other vertex orderings of G, that have different initial vertices.

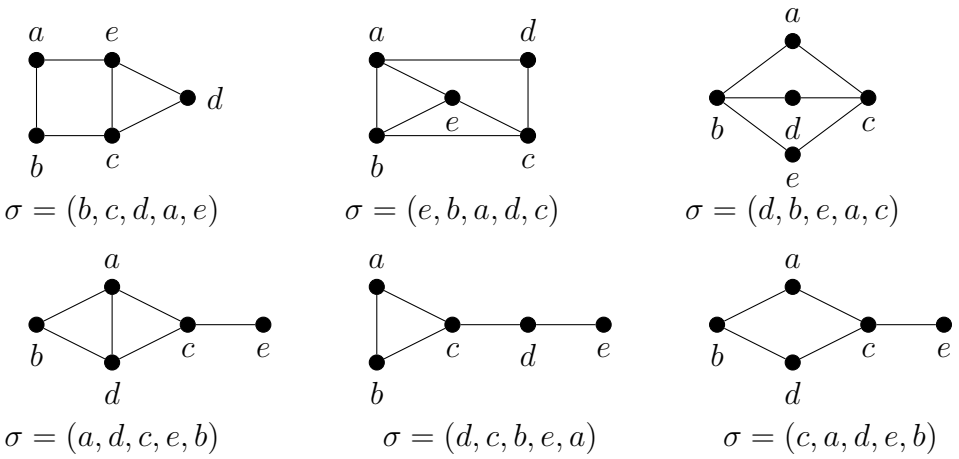


Figure 20: Graphs and corresponding orderings that are MNS and not MCS orderings.

4.4 Maximal Neighborhood Search vs Lexicographic BFS/DFS

In this section we characterize graphs for which it holds that every MNS ordering is also a LexBFS ordering, and graphs for which it holds that every MNS ordering is also a LexDFS ordering. It turns out that in both cases the same graphs are forbidden as induced subgraphs, as the following lemmas show.

Lemma 4.13. *If every MNS ordering of G is also a LexBFS ordering of G, then G is a $\{P_4, C_4\}$-free graph.*

Proof. Let G be a graph in which every MNS ordering is a LexBFS ordering. Assume for contradiction that G is not a $\{P_4, C_4\}$-free graph.

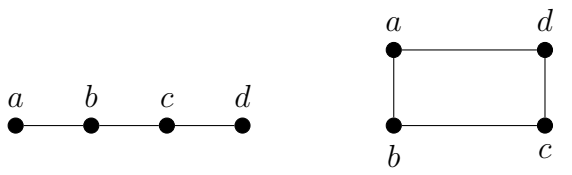
Assume first that G contains an induced P_4, and let a, b, c, d be the vertices of such a P_4 (see Fig. 21 left). Let σ be a MNS ordering of vertices in G with $\sigma(b) = 1$. Then any neighbor of b can be selected next, so let $\sigma(c) = 2$. Then the label of vertex d contains vertex c, while a label of vertex a does not contain it, meaning that the label of vertex d will never be a proper subset of a label of vertex a, and we can select vertex d before vertex a in σ. At the same time once vertices b and c were selected, from the definition of LexBFS it follows that all the neighbors of b must be selected before its non-neighbors, so if σ is a LexBFS ordering of G, it must be that $a <_\sigma d$; a contradiction. If G contains an induced C_4, we construct an ordering σ with initial segment (b, c, d, a) (see Fig. 21 right) and using the similar arguments we get that σ is MNS and not LexBFS ordering of G. It follows that G is a $\{P_4, C_4\}$-free graph. □

Lemma 4.14. *If every MNS ordering of G is also a LexDFS ordering of G, then G is a $\{P_4, C_4\}$-free graph.*

Proof. Let G be a graph in which every MNS ordering is a LexDFS ordering. Assume for contradiction that G is not a $\{P_4, C_4\}$-free graph.

Assume first that G contains an induced P_4, and let a, b, c, d be vertices of P_4 (see Fig. 21 left). Let σ be a MNS ordering of vertices in G with $\sigma(b) = 1$. Then any neighbor of b can be selected next, so let $\sigma(c) = 2$. Then a label of vertex a contains a vertex b, while a label of vertex d does not contain it, meaning that a label of vertex a will never be a proper subset of a label of vertex d, and we can select vertex a before vertex d in σ. At the same time once the vertices b and c were selected, from the definition of LexDFS it follows that (b, c, a, d) cannot be an initial segment of LexDFS ordering of G, since vertices c, a, d do not satisfy the condition from Theorem 3.11.

If G contains an induced C_4, the same reasoning holds: the ordering (b, c, a, d) cannot be the initial segment of LexDFS ordering of G, while it can be the initial segment of MNS ordering of G. It follows that G is a $\{P_4, C_4\}$-free graph. □



$\sigma_1 = (b, c, d, a)$ $\quad$ $\sigma_1 = (b, c, d, a)$

$\sigma_2 = (b, c, a, d)$ $\quad$ $\sigma_2 = (b, c, a, d)$

Figure 21: A path (left) and a cycle (right) on 4 vertices. σ_1 is an MNS ordering that is not a LexBFS ordering. σ_2 is an MNS ordering that is not a LexDFS ordering.

It follows that given a graph G satisfying the property that every MNS ordering is a LexBFS (resp., LexDFS) ordering, it must be true that G is a $\{P_4, C_4\}$-free graph. It turns out that this is also sufficient condition – in a $\{P_4, C_4\}$-free graphs every MNS ordering is also a LexBFS ordering and a LexDFS ordering. We prove these claims in the following two theorems. Observe that $\{P_4, C_4\}$-free graphs are also known as trivially perfect graphs, and can be obtained from the 1-vertex graphs by iteratively applying the operations of disjoint union and addition of universal vertices [22].

Theorem 4.15. *Let G be a $\{P_4, C_4\}$-free graph. Then every MNS ordering of G is also a LexBFS ordering of G.*

Proof. Let G be a $\{P_4, C_4\}$-free graph, and assume for contradiction that there is an ordering σ of vertices in G that is an MNS ordering of G and not a LexBFS ordering of G. From Theorem 3.8 we know that there exist vertices a, b, c in G such that $a <_\sigma b <_\sigma c$ and $ac \in E(G)$, $ab \notin E(G)$, and for every $d <_\sigma a$ it holds that either $db \notin E(G)$ or $dc \in E(G)$. Let a, b, c be a leftmost such triple (that is, for any other triple $a' <_\sigma b' <_\sigma c'$ and $a'c' \in E(G)$, $a'b' \notin E(G)$, with $\sigma(a')+\sigma(b')+\sigma(c') < \sigma(a)+\sigma(b)+\sigma(c)$ the condition from Theorem 3.8 is satisfied).

We know that σ is an MNS ordering, so by Theorem 3.16 it follows that there exists a vertex $d <_\sigma b$ in G such that $db \in E(G)$ and $dc \notin E(G)$. It cannot be that $d <_\sigma a$, so it follows that have that $a <_\sigma d <_\sigma b$. If $ad \in E(G)$, or $bc \in E(G)$, then the vertices $\{a, b, c, d\}$ induce either a P_4, or a C_4 in G; a contradiction. It follows that $ad \notin E(G)$ and $bc \notin E(G)$.

Now the vertices $a <_\sigma d <_\sigma c$ form a triple with $ac \in E(G)$ and $ad \notin E(G)$, so they must satisfy the condition from Theorem 3.8 and there exists a vertex $d_1 <_\sigma a$ such that $d_1 d \in E(G)$ and $d_1 c \notin E(G)$. Moreover, it follows that $d_1 a \notin E(G)$, for otherwise the vertices $\{d_1, a, d, c\}$ form a P_4 in G.

Consider now the vertices $d_1 <_\sigma a <_\sigma d$. They form a triple satisfying $d_1 d \in E(G)$ and $d_1 a \notin E(G)$, so by Theorem 3.8 there exists a vertex $d_2 <_\sigma d_1$ such that $d_2 a \in E(G)$ and $d_2 d \notin E(G)$. If $d_2 d_1 \in E(G)$, then the vertices $\{d_2, d_1, a, d\}$ form an induced P_4, a contradiction. We can continue the same process and apply Theorem 3.8 on vertices d_2, d_1, a in order to obtain a vertex d_3, and then apply the same process on vertices d_i, d_{i-1}, d_{i-2} to obtain vertices d_{i+1}, for $i \geq 3$, as in Theorem 3.8. Since a graph G is finite, in this process we get the vertices $d_1, \ldots, d_k$, for some finite number k. Let k be the maximum length of such a sequence of vertices. It will be true that $d_i <_\sigma d_{i-1}$ for all $i \geq 2$, and

$$d_{i+2} d_i \in E(G) \text{ and } d_{i+3} d_i \notin E(G), \tag{4.1}$$

for all $i \geq 1$.

We prove the following claim inductively.

Claim : $d_i d_{i-1} \notin E(G)$, for $i \in \{2, \ldots, k\}$.

We know that $d_2 d_1 \notin E(G)$, so the inductive basis holds. Assume now that for all $i \leq j$ we have that $d_i d_{i-1} \notin E(G)$. Let $i = j + 1$. If $d_{j+1} d_j \in E(G)$, then the vertices $\{d_{j+1}, d_j, d_{j-1}, d_{j-2}\}$ induce a P_4 in G. This is true since $d_j d_{j-1} \notin E(G)$, $d_{j-1} \notin E(G)$ by inductive hypothesis, while other edges and non-edges follow from (4.1). A contradiction with the definition of G, so the claim follows.

It follows that vertices $d_k <_\sigma d_{k-1} <_\sigma d_{k-2}$ satisfy that $d_k d_{k-2} \in E(G)$ and $d_k d_{k-1} \notin E(G)$, so by Theorem 3.8 there exists a vertex d_{k+1} and k is not maximal; a contradiction.

□

Theorem 4.16. *Let G be a $\{P_4, C_4\}$-free graph. Then every MNS ordering of G is also a LexDFS ordering of G.*

Proof. Let G be a $\{P_4, C_4\}$-free graph, and assume for contradiction that there is an ordering σ of vertices in G that is an MNS ordering of G and not a LexDFS ordering of G. From Theorem 3.11 we know that there exist vertices a, b, c in G such that $a <_\sigma b <_\sigma c$ and $ac \in E(G)$, $ab \notin E(G)$, and for every $a <_\sigma d <_\sigma b$ it holds that either $db \notin E(G)$ or $dc \in E(G)$. Let a, b, c be a leftmost such triple (that is, for any other triple $a' <_\sigma b' <_\sigma c'$ and $a'c' \in E(G)$, $a'b' \notin E(G)$, with $\sigma(a')+\sigma(b')+\sigma(c') < \sigma(a)+\sigma(b)+\sigma(c)$ the condition from Theorem 3.11 is satisfied).

We know that σ is an MNS ordering, so by Theorem 3.16 it follows that there exists a vertex $d <_\sigma b$ in G such that $db \in E(G)$ and $dc \notin E(G)$. It cannot be that $a <_\sigma d <_\sigma b$, so it follows that have that $d <_\sigma a$. If $ad \in E(G)$, or $bc \in E(G)$, then the vertices $\{a, b, c, d\}$ induce either a P_4, or a C_4 in G; a contradiction. It follows that $ad \notin E(G)$ and $bc \notin E(G)$.

Now the vertices $d <_\sigma a <_\sigma b$ form a triple with $db \in E(G)$ and $da \notin E(G)$, so they must satisfy the condition from Theorem 3.11 and there exists a vertex $d <_\sigma d_1 <_\sigma a$ such that $d_1 a \in E(G)$ and $d_1 b \notin E(G)$. Moreover, it follows that $dd_1 \notin E(G)$, for otherwise the vertices $\{d, d_1, a, b\}$ form an induced P_4 in G.

Consider now the vertices $d <_\sigma d_1 <_\sigma b$. They form a triple satisfying $db \in E(G)$ and $dd_1 \notin E(G)$, so by Theorem 3.11 there exists a vertex $d <_\sigma d_2 <_\sigma d_1$ such that $d_2 d_1 \in E(G)$ and $d_2 b \notin E(G)$. If $dd_2 \in E(G)$, then the vertices $\{d, d_2, d_1, b\}$ form a P_4, a contradiction. We can continue the same process and apply Theorem 3.11 on vertices d, d_2, b in order to obtain a vertex d_3, and then apply the same process on vertices d, d_i, b to obtain vertices d_{i+1}, for $i \geq 3$, as in Theorem 3.11. Since a graph G is finite, in this process we get the vertices $d_1, \ldots, d_k$, for some finite number k.

Let k be the maximum length of such a sequence of vertices. It will be true that

$d <_\sigma d_i <_\sigma d_{i-1}$ for all $i \geq 2$, and

$$d_{i+1}d_i \in E(G) \text{ and } d_ib \notin E(G), \tag{4.2}$$

for all $i \geq 1$.

We prove the following claim inductively.

Claim : $dd_i \notin E(G)$, for $i \in \{1, 2, \ldots, k\}$.

We know that $dd_1 \notin E(G)$, so the inductive basis holds. Assume now that for all $i \leq j$ we have that $dd_i \notin E(G)$. Let $i = j + 1$. If $dd_{j+1} \in E(G)$, then the vertices $\{b, d, d_{j+1}, d_j\}$ induce a P_4 in G. This is true since $dd_j \notin E(G)$ by inductive hypothesis, while other edges and non-edges follow from (4.2). A contradiction with the definition of G, so the claim follows.

It follows that vertices $d <_\sigma d_k <_\sigma b$ satisfy that $db \in E(G)$ and $dd_k \notin E(G)$, so by Theorem 3.11 there exists a vertex d_{k+1} such that $d <_\sigma d_{k+1} <_\sigma d_k$ and k is not maximal; a contradiction.

□

From the above theorems we get the following corollary.

Corollary 4.17. *If G is $\{P_4, C_4\}$-free graph, then for every ordering σ of $V(G)$ the following statements are equivalent:*

i) σ is a MNS ordering of G.

ii) σ is a LexDFS ordering of G.

iii) σ is a LexBFS ordering of G.

In other words, it follows that MNS and LexBFS are equivalent in G if and only if G is a $\{P_4, C_4\}$-free graph, and similarly, MNS and LexDFS are equivalent in G if and only if G is a $\{P_4, C_4\}$-free graph.

Corollary 4.18. *MNS and LexBFS are equivalent in G if and only if G is a $\{P_4, C_4\}$-free graph. MNS and LexDFS are equivalent in G if and only if G is a $\{P_4, C_4\}$-free graph.*

5 Graph Search Trees

In Chapter 3 we introduced various graph search methods, and saw that one possible outcome of a graph search is a search order. In this section we focus on the second possible outcome of a graph search: *a search tree*. A search in a connected undirected graph imposes a search direction on each edge of G given by the direction in which the edge is traversed when the search is performed. If we take a subgraph of G defined by the subset of these directed edges such that there is exactly one vertex v with no incoming edges and every other vertex in G has exactly one incoming edge, we get a directed spanning tree of G. The underlying graph T of this directed spanning tree is called a *search tree* of a graph G, and vertex v is the root of T. More precisely, given a graph G and a search order $\sigma = (v_1, \ldots, v_n)$ of vertices in G, *a search tree* T is defined as a spanning tree of G with every vertex $v_i \in V(T)$ being adjacent in T to exactly one vertex $v_j \in N_G(v_i)$, with $j <_\sigma i$.

Given a search order σ of a graph G and some vertex $v_i \in V(G)$, $i \neq 1$, one may observe that the neighbor v_j of v_i, $j < i$ in T is not uniquely determined. If σ is a BFS order of G, one usually connects every vertex of G to its neighbor that appeared first in σ. Contrary, if σ is a DFS order of G, then every vertex v of G is connected to its most recently visited neighbor (last neighbor of v visited before v). In literature we can find particular definitions of $\mathcal{F}$-trees (first-in trees) and $\mathcal{L}$-trees (last-in trees), depending whether we connect a vertex v to its earliest, or most recently visited neighbor [1].

Definition 5.1. Given a search 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 $j < i$.

The search tree recognition problem asks whether a given spanning tree of a given graph is a search tree of a given search method. Using the definitions of first-in and last-in trees we formally define the problem as follows.

$\mathcal{F}$-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 the given type such that T is its $\mathcal{F}$-tree ($\mathcal{L}$-tree) of G.

Observe that the $\mathcal{F}$-Tree ($\mathcal{L}$-Tree) Recognition Problem is defined with no starting vertex of a graph search. If, additionally, the starting vertex of a graph search is given, the problem is referred to as the *rooted $\mathcal{F}$-Tree ($\mathcal{L}$-Tree) Recognition Problem* [1]. If the rooted version of a problem can be solved in polynomial time, then solving the problem n times (once for each vertex) gives a result for the non-rooted version. The other direction is not necessarily true, and theoretically it can happen that the non-rooted version can be solved efficiently, while the rooted version is hard [1].

Example 5.2. Consider the graph G given in Fig. 22 a). One possible outcome of a DFS on G is a vertex order $\sigma = (a, b, e, c, f, d, g)$. A search tree T corresponding to σ is constructed so that we start in a and traverse σ so that for each vertex we connect it with its most recently visited neighbor. For example, vertex c is connected to e and not to a, since $a <_\sigma e$. We thus have that the tree T depicted on Fig. 22 b) with thick edges is a DFS $\mathcal{L}$-tree on G.

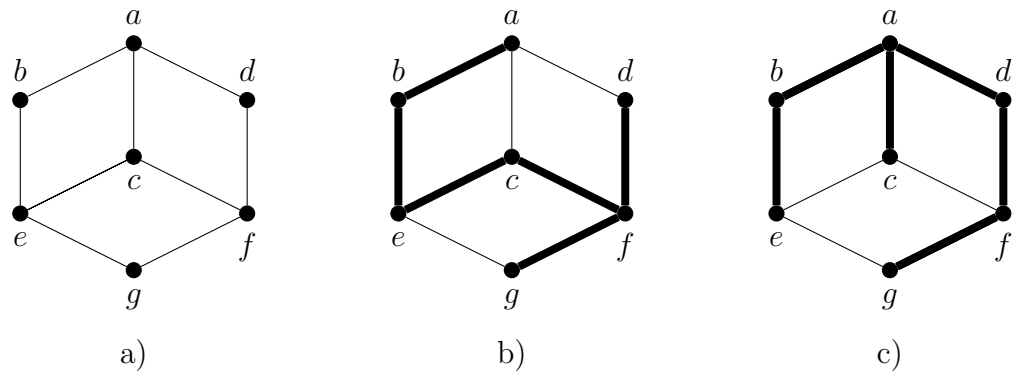


Figure 22: A graph G (a)) and its spanning trees (b) and c)). The spanning tree on b) is a DFS $\mathcal{L}$-tree of G, while the spanning tree on c) is not a DFS $\mathcal{L}$-tree of G.

Consider now the spanning tree T of a graph G depicted with thick edges on the same figure c). Let us determine whether T is a DFS $\mathcal{L}$-tree of G. If T would be a DFS $\mathcal{L}$-tree of G, then there would exist a DFS ordering σ of G corresponding to T. We consider few cases depending on the first vertex of σ. Since σ is supposed to be a DFS ordering of a biconnected graph G, from definition of DFS it follows that $\sigma^{-1}(1) \in \{e, c, g\}$. If $\sigma(e) = 1$, then from the construction of T it follows that $\sigma = (e, b, a, d, f, g, c)$, so the edge cf should belong to $E(T)$, which is not the case; a contradiction. If $\sigma(c) = 1$, we can see that the thick paths don't go as deep in G as possible, so T cannot be a DFS search tree. Finally, if $\sigma(g) = 1$, then from the construction of T it follows that $\sigma = (g, f, d, a, b, e, c)$, and the edge ec should belong to $E(T)$, which is not the case; a contradiction. It follows that there is no DFS ordering σ of G such that T is a search tree corresponding to σ, and so T is not a DFS $\mathcal{L}$-tree of G.

From the definition of search trees it follows that every search tree corresponds to some search order. In Fig. 2 (Chapter 3) we saw the relations between graph orders, and the same relations hold for a corresponding search trees, for both $\mathcal{F}$- and $\mathcal{L}$-trees, respectively [2]. Although in general it can happen that distinct search orders produce the same $\mathcal{F}$- or $\mathcal{L}$-tree, the relations represented in Fig. 2 represent proper inclusion relations, so there are search trees that are BFS, and not LexBFS trees, or that are DFS, and not LexDFS trees, etc. For examples of a corresponding search trees, see Fig. 23 [4].

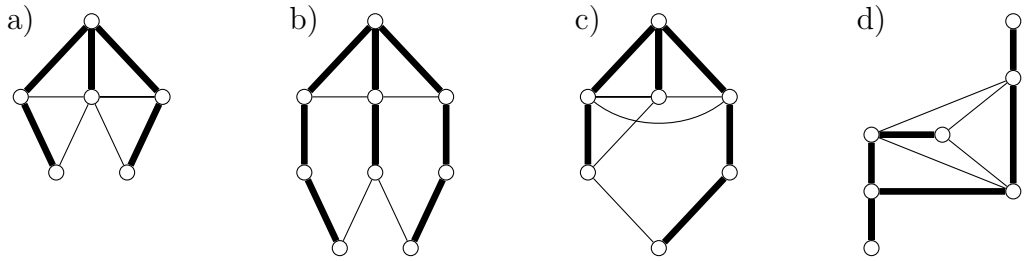


Figure 23: Four examples of graphs with their search trees denoted by the thick edges. The graph in a) depicts an $\mathcal{F}$-tree of BFS that is not an $\mathcal{F}$-tree for LexBFS or MNS. The graph in b) depicts an $\mathcal{F}$-tree of MNS and BFS that is not an $\mathcal{F}$-tree for LexBFS. The graph in c) shows a search tree that is an $\mathcal{F}$-tree of MNS, BFS and LexBFS and 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 LexDFS.

In the following sections we consider the search tree recognition problems for $\mathcal{L}$- and $\mathcal{F}$-trees, respectively. Historically, the very first results regarding the recognition of search trees considered DFS, and since all DFS-like searches produce $\mathcal{L}$-trees, we start with $\mathcal{L}$-trees. In the second section we proceed to $\mathcal{F}$-trees.

5.1 Last-in Trees

A search tree of some graph is called a last-in tree if there is a corresponding ordering of vertices σ such that every vertex in a search tree is adjacent to its most recently visited neighbor. The main representative of $\mathcal{L}$-trees is a DFS $\mathcal{L}$-tree, and in literature it is often simply referred to as a *DFS tree*. The first known paper that considers the proposed search trees was written in 1972 by Tarjan [40], where the author characterized the DFS trees, without mentioning the recognition problem and algorithm that solves it. The DFS-tree recognition problem was formulated for the first time by Hagerup and Nowak in 1985 [24], where the authors gave a linear-time recognition algorithm. The same result was given independently by Korach and Ostfeld in [28]. After that, $\mathcal{L}$-trees were not studied for a while. Recently, Beisegel et. al. [4] gave new results in

this direction. Using similar ideas as given by Korach and Ostfeld for DFS trees, they showed that the recognition problem of LexDFS $\mathcal{L}$-trees is solvable in polynomial time. Regarding the other search methods, Beisegel gave efficient algorithms that solve the problem for particular search methods on chordal graphs, with the complexity of the problem remaining open in general case [2]. An overview of known results regarding $\mathcal{L}$-trees is given in Table 2. In the rest of this section we present proposed results in more detail.

Table 2: Complexity of the $\mathcal{L}$-tree recognition problem. Letter P denotes the existence of a polynomial-time algorithm, while L denotes that the problem is solvable in linear time.

Tree results	LexBFS	DFS	LexDFS	MCS	MNS
All Graphs	?	L [24, 28]	P [4]	?	?
Chordal Graphs	L [2]	L [2]	L [2]	L [2]	L [2]

5.1.1 DFS trees

Trees obtained as a result of DFS on a graph were studied by Tarjan in 1972 [40]. He gave a complete characterization of DFS trees using the so-called *palm trees*. A *palm tree* is a directed graph $D = (V, A)$, consisting of two disjoint sets of arcs (directed edges) X and Y, such that the subgraph containing the arcs in X is a directed spanning tree of the underlying graph P of D and for every directed edge $v \to w \in Y$ there exists a directed path wv completely contained in X. The arcs in X are called *tree arcs*, and the arcs in Y are called *the fronds*. Intuitively, it means that the arcs in X define a directed spanning tree T of P, with root v, and all the arcs in T are oriented away from the root, while all other arcs in P are oriented towards the root. Tarjan proved that a directed graph D generated by a DFS of a connected graph G is a palm tree, and conversely, any palm tree D can be generated by some depth-first search of the undirected version of D [40]. Additionally, he used DFS for the construction of two efficient graph algorithms: an algorithm for finding the biconnected components of an undirected graph and an improved version of an algorithm for finding the strongly connected components of a directed graph. Both algorithms run in linear time, and represent important applications of DFS method. Using the concept of palm trees, Hopcroft and Tarjan developed a linear-time algorithm for testing planarity of a graph [25]. The following lemma appeared in a bit different form in the first DFS-based paper by Tarjan and it turned out to be useful for development of the algorithm for

the DFS tree recognition [40].

Lemma 5.3 (Tarjan [40])**.** *Let $G = (V, E)$ be a graph and let T be a rooted DFS $\mathcal{L}$-tree of G. For each edge $uv \in E$ it holds that either $uv \in E(T)$, or u is an ancestor of v in T, or v is an ancestor of u in T.*

A simple linear-time algorithm that recognizes whether a given spanning tree T of a given graph G can be obtained as a result of DFS on G was developed using the above characterization of DFS trees. Hagerup and Nowak observed it in 1985 and therefore developed a linear-time algorithm that answers whether a spanning tree T of a graph G is a DFS tree [24]. Their algorithm checks for every edge of a graph G whether it satisfies the conditions of the above lemma, and if this is not the case for at least one of them, the algorithm returns the negative answer, while otherwise it returns a positive answer. If the answer is positive, then the algorithm outputs also the vertex r of G for which it is true that T is a DFS tree of G rooted at r (see Algorithm 8). For a linear-time implementation of the algorithm, the authors used the recursive relations developed on relations father–son in a rooted tree T, so they showed that the set of all non-tree edges uv for which it is not true that u is either ancestor or a descendant of v, for each root $r \in V(G)$ can be computed in linear time [24]. This means that in the algorithm we do not have to check each non-tree edge in every vertex iteration, and that is why the algorithm works in linear time.

Algorithm 8: Algorithm which decides whether T is a DFS tree of G.

Input: Graph $G = (V, E)$, spanning tree T of G.

Output: T is a DFS tree of G or not.

```
1  begin
2      isTree← true;
3      foreach r ∈ V(G) do
4          foreach uv ∈ E(G) \ E(T) do
5              if u not ancestor of v and v not ancestor of u then
6                  isTree ← false
7          if isTree then
8              return T is a DFS tree of G rooted at r
9          isTree← true;
10     return T is not a DFS tree of G
```

A few years later, Korach in Ostfeld [28] independently developed a new linear-time algorithm that recognizes the DFS trees, and in case of negative answer returns the proof for that fact. Their algorithm consists of four phases, and is based on a hereditary property of the DFS search trees, as the following lemma states.

Lemma 5.4 (Korach and Ostfeld, [28]). *Let $G = (V, E)$ be a graph with spanning tree T. Let G_i be a subgraph of G with a spanning tree T_i which is the restriction of T to G_i. If T is an DFS tree on G, then T_i is a DFS tree on G_i.*

Korach and Ostfeld observed that the recognition of DFS trees in 2-connected graphs can be solved efficiently. More precisely, they showed that given a 2-connected graph $G = (V, E)$ and its spanning tree T, one of the following holds:

i) T is a path and there are two DFS orientations of T (where the roots of the oriented trees are the two ends of the path).

ii) T is not a path and there is at most one DFS orientation of T. Equivalently, there is at most one leaf s in T where T rooted in s satisfies the property that for every non-tree edge uv of G it holds that u is an ancestor or descendant of v in T.

The algorithm developed by Korach and Ostfeld answers whether T is a DFS tree of a graph G and here we briefly explain the steps of the algorithm.

1. Decompose the graph G into its 2-connected components. Denote the 2-connected components of G by $C_1, C_2, \ldots, C_k$, and the separating vertices of G by $s_1, \ldots, s_p$. Compute a graph G' with $V(G') = \{C_1, C_2, \ldots, C_k\} \cup \{s_1, \ldots, s_p\}$ and $E(G') = \{C_i s_j \mid s_j \text{ is a vertex of } C_i \text{ in } G\}$ (see [19]).

2. For each 2-connected component $G_i = (V_i, T_i)$ of G check whether T_i (the induced subtree of T in G_i) is a DFS tree in G_i.

3. Compute the orientation $G'_A = (V(G'), F)$ of a graph G' in the following way: add the directed edge $C_i \to a_j$ to F whenever $a_j \in V(C_i)$. If T_i rooted at a_m is a DFS tree in G_i, add $C_i \to a_m$ to F.

4. Find all the roots of the graph G'_A. If C_i is a root of G'_A, then all the (one or two) roots of a DFS run that gives T_i in G_i are output (i.e., they are the roots of a DFS run in G that gives T).

For all the details of the linear-time implementation of the above algorithm, see [28].

5.1.2 LexDFS trees

As we have already seen, the LexDFS represents a special instance of a DFS, so it is natural to expect that LexDFS trees have some in a sense common properties with DFS trees. The following lemma represents a LexDFS analogue of Lemma 5.4, and is a main ingredient of a polynomial-time algorithm for the recognition of LexDFS trees [4].

Lemma 5.5 (Beisegel et al. [4]). *Let $G = (V, E)$ be a graph with spanning tree T. Let G_i be an induced subgraph of G with a spanning tree T_i which is the restriction of T to G_i. If T is an LexDFS $\mathcal{L}$-tree on G, then T_i is an LexDFS $\mathcal{L}$-tree.*

The algorithm that solves the rooted LexDFS $\mathcal{L}$-tree recognition problem is presented in Algorithm 9. Observe that the input of the algorithm consists of a graph G, a spanning tree T, and a vertex $r \in V$, and it determines whether T is an LexDFS $\mathcal{L}$-tree of G, with the corresponding LexDFS search starting in the vertex r. For a non-rooted version of a problem, we run the algorithm at most n times, that is once for each vertex of a graph.

Algorithm 9: Algorithm which decides whether T is an $\mathcal{L}$-tree of LexDFS on G rooted in r.

Input: Graph $G = (V, E)$, spanning tree T of G, and a vertex $r \in V$.
Output: T is an $\mathcal{L}$-tree of LexDFS on G or not.

```
1  begin
2      S ← {r};
3      foreach vertex v ∈ V − r do label(v) ← ∅;
4      foreach vertex v ∈ N(r) do
5          prepend 0 to label(v);
6          pred(v) ← r;
7      while S ≠ V do
8          choose a node v ∈ V − S with lexicographically largest label, such that
             {pred(v), v} ∈ E(T);
9          if no such v exists then
10             return T is not an L-tree of LexDFS on G
11         S ← S ∪ {v};
12         for w ∈ N(v) \ S do
13             prepend i to label(w);
14             pred(w) ← v;
15     return T is an L-tree of LexDFS on G.
```

Let us see how the above algorithm works on an example.

Example 5.6. Let G be the graph from Fig. 24 and let a spanning tree T of G be depicted with thick edges. Let us first check whether T is a LexDFS $\mathcal{L}$-tree of G rooted at c. We see that c is a universal vertex in G, so after setting $\sigma(c) = 1$, all other vertices have the same label, and by algorithm we can choose the vertex b after c since $bc \in E(T)$. Then the labels are $\ell(a) = \ell(e) = \ell(g) = 21$, $\ell(d) = \ell(f) = 1$, so among vertices a, e, g we choose a since it is adjacent in T with vertex b (step 8 in the

algorithm). After taking a, the labels are $\ell(d) = 31$, $\ell(e) = \ell(g) = 21$, $\ell(f) = 1$, so we have to choose vertex d, but the edge ad is not an edge of a tree T, so the algorithm returns *"T is not an $\mathcal{L}$-tree of LexDFS on G"*.

Let us now check whether T is a LexDFS $\mathcal{L}$-tree of G rooted at a. If we start in a, in the next step we have to choose among vertices b, c, d and we choose b since $ab \in E(T)$. Then the labels are $\ell(c) = 21$, $\ell(e) = \ell(g) = 2$, $\ell(d) = 1$, and in the next step we choose vertex c. After choosing c, every unvisited vertex gets number 3 at the beginning of its label, and by algorithm we choose the vertex g. Now the labels are $\ell(e) = 432$, $\ell(d) = 431$, $\ell(f) = 43$, and we take vertex e. Then step 11 of the algorithm says that we have to do something for each unvisited neighbor of e. This is trivial, since all neighbors of e are already visited, so lines $12, 13, 14$ of the algorithm are not executed, and we still have to choose v with lexicographic largest label such that $gv \in E(T)$ (since $\text{pred}(v) = g$). Among vertices d and e we choose vertex d, and finally e at the end. The resulting ordering is $\sigma = (a, b, c, g, e, d, f)$ and it corresponds to tree T, rooted at a. The algorithm returns *"T is an $\mathcal{L}$-tree of LexDFS on G"*.

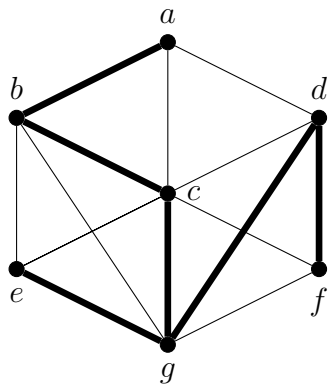


Figure 24: A graph G with a spanning tree T. Tree T is an $\mathcal{L}$-tree of LexDFS on G rooted at a. T is not an $\mathcal{L}$-tree of LexDFS on G rooted at c.

Using Algorithm 9, we conclude that the recognition of LexDFS trees can be done in polynomial time, for both rooted and unrooted versions of a problem. An interesting fact is that, given a graph G and a vertex $v \in V(G)$, it is NP-hard to decide whether there exists a LexDFS ordering σ of vertices in G such that v is the last vertex in σ [3]. This problem is known in the literature as the END-VERTEX PROBLEM (see e.g. [2]).

Theorem 5.7 (Beisegel et al. [4])**.** *The (rooted) $\mathcal{L}$-tree recognition problem for LexDFS can be solved in polynomial time.*

5.2 First-in Trees

A search tree of some graph is called a first-in tree if there is a corresponding ordering σ of vertices such that every vertex in a search tree is adjacent to its first visited neighbor. The main representative of $\mathcal{F}$-trees is a BFS $\mathcal{F}$-tree, and it is often referred to as a *BFS tree*. Very first results concerning $\mathcal{F}$-trees were published in 1990, and were dealing with the structure and the recognition of BFS trees [33]. In that work Manber characterized the edges of a graph that do not appear in a BFS tree, and developed an algorithm that recognizes the BFS trees in polynomial time. Later, Beisegel et al. [4] were studying the complexity of $\mathcal{F}$-tree recognition for various graph search methods, and obtained proofs of NP-completeness for lexicographic instances of BFS and DFS, and for MNS and MCS in the class of weakly chordal graphs. In contrast with the proofs of NP-completeness, in [2] one can find polynomial-time algorithms that solve the tree recognition problem for the mentioned search methods on chordal graphs. An overview of known results is presented in Table 3. In the first subsection we present the results concerning BFS trees, while in the second subsection we focus on hardness results for NP-hard instances of a problem.

Table 3: Complexity of the $\mathcal{F}$-tree recognition problem. NPC denotes the NP-complete instances of a problem. The letter P denotes the polynomial-time algorithm, while L denotes the linear-time algorithm.

Tree results	BFS	LexBFS	LexDFS	MCS	MNS
All Graphs	L [24, 33]	NPC	NPC	NPC	NPC
Weakly Chordal	L	NPC [4]	NPC [4]	NPC [4]	NPC [4]
Chordal	L [2]	P [2]	P [2]	P [2]	P [2]

5.2.1 BFS trees

Trees that can be obtained as a result of BFS on a graph were studied by Manber and independently by Hagerup and Nowak. However, the only published work concerning the BFS trees was done by Manber, while the work by Hagerup and Nowak remained unpublished (see [24], [33]). In his work Manber gave a characterization of BFS trees of a graph using the structure of the edges of a graph that do not appear in the proposed spanning tree. The proposed characterization leads to a polynomial-time algorithm that recognizes the BFS search trees of a graph G, and gives a corresponding BFS ordering of G. We should mention that the algorithm solves the rooted instance of the problem, so for the unrooted case it should be executed n times, with a root

being any vertex in a graph. For better understanding of the algorithm, we introduce some terminology. Let G be a graph and T a rooted spanning tree of G, with root r. Then for any vertex $v \in V(G)$ the *level of v with respect to r in T* is denoted by $v.level$ and represents the distance $d_T(v, r)$. An edge $uv \in E(G)$ is k-level edge if $|v.level - u.level| = k$. For each level i, $i \in \{1, \ldots, \epsilon_G(r)\}$ we define a *level digraph* G_i to be a directed graph consisting of vertices at distance i from r in T, and having an empty set of arcs. It is clear that any edge from T connects two vertices of consecutive levels, so any edge in T is a 1-level edge. Also, if T is a BFS tree of G, it is not possible that some edge in G is of level k with respect to T, for $k \geq 2$. For, if uv is an edge of G of level k, $k \geq 2$, then there is some $i \geq 0$, such that $d_T(u, r) = i$ and $d_T(v, r) = i + k \geq i + 2$, so the distance between u and v in T is at least two. This is not possible, since BFS visits all neighbors of already visited vertex consecutively, and cannot skip some vertices. This condition is a part of the following characterization theorem of BFS trees.

Theorem 5.8 (Manber [33]). *Let $G = (V, E)$ be a connected graph, let T be a spanning tree of G, and let $r \in V$ be a root of T. Then, T is a BFS tree of G rooted at r if and only if the following conditions hold:*

1. *All edges in G are either 0-level edges or 1-level edges with respect to T, and*
2. *It is possible to order the vertices in each level such that*
   a) *if x and y are vertices of the same level and x appears before y in the order of that level, then all of x's children appear before all of y's children in the order of the next level, and*
   b) *if (v, w) is a nontree edge with $v.level = i$ and $w.level = i + 1$ and if u is the parent of w in T, then v appears after u in the order of level i.*

Using the above theorem, Manber developed the algorithms for the recognition of rooted and unrooted BFS trees. The algorithm that determines whether a given rooted spanning tree T of a graph G, with root r, is a BFS tree of G works in two phases, where in the first phases it determines whether T is a BFS tree of G, and in the second phases it gives a corresponding BFS order, if the result of the first phases was positive. In the first phase, the algorithm checks whether both conditions of Theorem 5.8 are fulfilled. The first condition can be checked easily, so that we consider each edge in the graph and compute whether it is 0- or 1-level edge. In the second condition, the existence of a corresponding nontree edge uv gives particular ordering conditions at each level where u and v both have ancestors. This may require more than a constant number of steps per one non-tree edge, so here we describe the idea to avoid it, presented by

Manber [33]. Given a nontree edge uv with $u.level = i$ and $v.level = i + 1$, let w be the lowest common ancestor of u and v in T, and let w_u (resp., w_v) be the child of w, belonging to the same subtree of w as u (resp., v) in T. If v is a child of w, then $w_v = w$. By condition 2b) of Theorem 5.8 we have that w_u appears after w_v in the order of that level. Additionally, from condition 2a) it follows that every descendant of w_v appears before descendants of w_u, if both of them are in the same level of T. This means that we consider each nontree edge uv with $u.level = i$ and $v.level = i + 1$, and for the corresponding vertices w_u and w_v we add a directed edge $w_v \to w_u$ to the level digraph G_i. If some level digraph G_i has a directed cycle, then T is not a BFS tree of G. Otherwise, we find a topological order of vertices in each level digraph G_i and we do a BFS on G, starting in r, and respecting the topological order in each level, with the restriction that the condition 2a) of Theorem 5.8 must be satisfied. The algorithm is presented in Algorithm 10.

The above algorithm has a linear-time implementation [33], using the recursive $*$-tree data structure of Berkman and Vishkin (see [6]).

Example 5.9. Let G be the graph depicted in Fig. 25, and let T be a spanning tree of G represented by thick edges. We will use Algorithm 10 to determine whether T is a BFS tree of G.

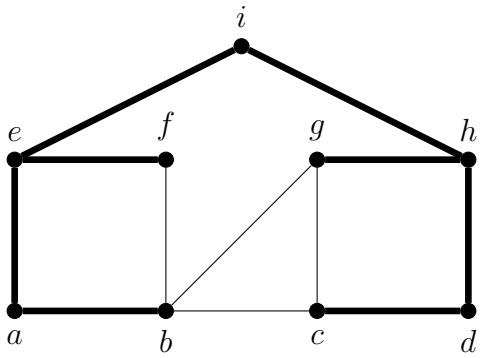


Figure 25: A graph G and its spanning tree T. Tree T is a BFS tree of G rooted in i, and it is not a BFS tree of G rooted in e.

Let us for instance first check whether T is a BFS tree of G rooted in e. If T has a root e, then the tree T and the corresponding vertex labels look as in Fig. 26 a). We know that $bc \in E(G) \setminus E(T)$ and since $b.level = 2$ and $c.level = 4$, it follows that the edge bc is of level 2, so T is not a BFS tree of G rooted at e. On the other side, if T is a BFS tree of G rooted at e, then T contains all the shortest paths from e to any other vertex in G. This, however, is not true, since $d_G(e, c) = 3$ and $d_T(e, c) = 4$.

Let us now check whether T can be a BFS tree of G with root i. If T is rooted in i, then T and the corresponding vertex levels look as in Fig. 26 b). A set of edges

Algorithm 10: Algorithm for deciding whether T is a BFS tree of G rooted in r.

Input: Connected graph G, spanning tree T of G and a vertex $r \in V(G)$.

Output: T is an $\mathcal{F}$-tree of BFS on G rooted at r or not. If answer is positive, the corresponding ordering σ is returned

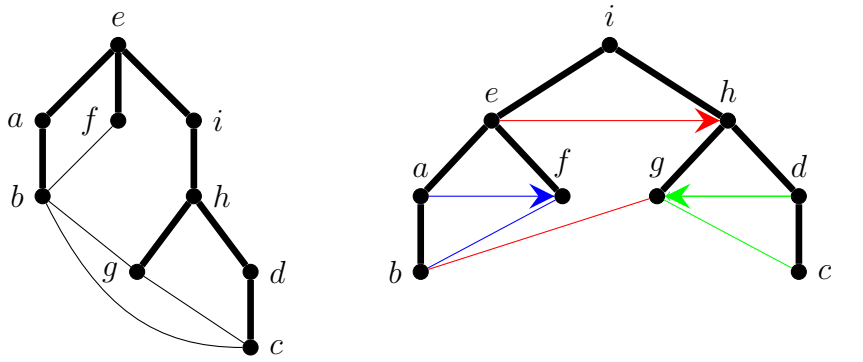
```
1  begin
2      foreach v ∈ V \ {r} do
3          v.level ← d_T(r, v)
4      foreach uv ∈ E(G) do
5          if |u.level − v.level| > 1 then
6              return T is not a BFS tree of G
7      foreach uv ∈ E(G) \ E(T) do
8          if u.level − v.level = 1 then
9              swap u and v
10         if v.level − u.level = 1 then
11             w ← the lowest common ancestor of u and v;
12             w_v ← child of w in the same subtree of w as v;
13             w_u ← child of w in the same subtree of w as u;
14             i ← w.level + 1;
15             E(G_i) = E(G_i) ∪ {w_v → w_u}
16     for i ← 1 to ε_G(r) do
17         foreach component H of G_i do
18             if H has a directed cycle then
19                 return T is not a BFS tree of G
20             find a topological order σ^i_H of H;
21     do a BFS of G starting in r and adding vertices from each level i to a
       queue with respect to topological order of component σ^i_H of a level graph
       G_i and return BFS ordering σ
```

that are in G but not in T consists of the edges $\{bf, bg, gc, bc\}$ and it is not difficult to see that all of them have levels 0 or 1. Let us now check the execution of lines 6-16 of Algorithm 10. First we have $u = b$, $v = f$ and $u.level = 3$, $v.level = 2$. It follows that $u.level - v.level = 1$, so we swap u and v, and get $v = b$, $u = f$. In lines 9-14 we get $w = e$, $w_u = f$, $w_v = a$, and $i = 3$, so $E(G_3) = \{a \to f\}$. Edge bg in the same steps gives the following: $v = b$, $u = g$, $w = i$, $w_v = e$, $w_u = h$, and $E(G_2) = \{e \to h\}$. Similarly, edge gc gives the following: $v = c$, $u = g$, $w = h$, $w_v = d$, $w_u = g$, and $E(G_3) = \{d \to g\}$. Edge bc is of level 0, so in the loop it does not satisfy the *if* condition, and nothing executes.

We know that $\epsilon_G(i) = 3$ so we have 3 level digraphs, and for each of them we find a corresponding topological ordering. For each $j = 1, 2, 3$, both digraph and its topological order are displayed on Fig. 26 c). As we can see, all digraphs are acyclic, so it remains to execute line 22 of the algorithm. We start BFS in i, and visit vertices of G_1 with respect to topological order, meaning that e is visited before h. Then in G_2 we visit the component corresponding to subtree of e before the component corresponding the subtree of h, and inside one component we visit vertices in the topological order obtained before. This means that we visit the vertices of level 2 in the order: a, f, g, d. Finally, the resulting BFS order is $\sigma = (i, e, h, a, f, d, g, b, c)$, and we can easily check that T is a BFS tree corresponding to σ.

It is clear that the unrooted version of a problem can be solved using Algorithm 10, by considering every vertex of a graph G as a potential root. This results in the larger complexity of the algorithm, since we have to make at most $|V(G)|$ iterations, and the resulting algorithm will not work in linear time. There exists, however, a linear-time algorithm that solves the unrooted version of a problem, also developed by Manber [33]. In order to present it here, we need some definitions.

Given a connected graph G with a spanning tree T, let $xy \in E(G) \setminus E(T)$. Let $u_0, \ldots, u_k$ be the vertices of the unique x, y-path in T. A *middle vertex associated with xy edge* is the vertex $u_{(k+1)/2}$ if k is odd, while in case when k is even there are two such vertices and are defined as: $u_{k/2}$ and $u_{(k+2)/2}$. *Cut-edges associated with xy edge* are defined as the directed edges $u_{(k+1)/2} \to u_{(k-1)/2}$ and $u_{(k+1)/2} \to u_{(k+3)/2}$ for k odd, and $u_{k/2} \to u_{k/2-1}$ and $u_{k/2+1} \to u_{k/2+2}$ for k even. From the first condition of Theorem 5.8 it follows that the only vertices of G that have to be considered as a potential root of BFS are all the vertices in the subtree of T containing the middle vertices obtained after removing the cut-edges from T. For, otherwise, if r is a root of a BFS tree T not satisfying that condition, than $|d_T(r, x) - d_T(r, y)| \geq 2$, a contradiction. The proposed algorithm has two phases. In the first one we take an arbitrary root r and compute the corresponding levels for all the vertices $v \in V(G)$: $v.level = d_T(r, v)$. After that, for each non-tree edge xy we find a middle vertex and cut edges, and once

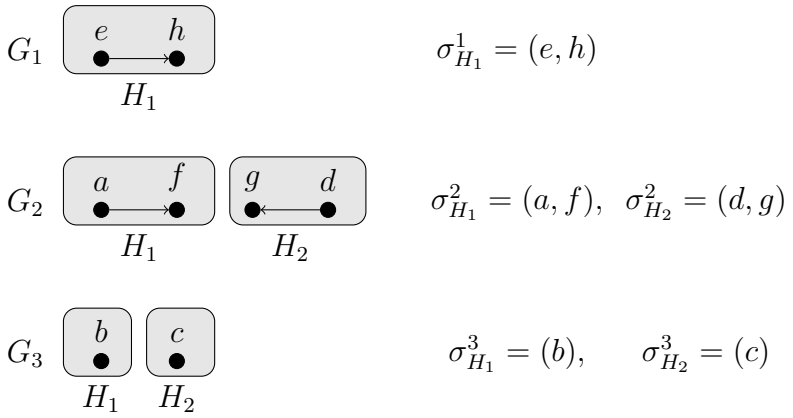


a	b	c	d	e	f	g	h	i
1	2	4	3	0	1	3	2	1

a)

a	b	c	d	e	f	g	h	i
2	3	3	2	1	2	2	1	0

b)



c)

Figure 26: A graph G and its spanning tree T from Example 5.9. Part a) shows the tree T rooted at e (thick edges), and the levels of vertices in T with respect to vertex e. Part b) shows the tree T rooted at i (thick edges), and the levels of vertices in T with respect to vertex i. The edges depicted with blue (red, green) are the non-tree edges of G that imply the arcs of the same color in the level graph of tree T rooted at i. In c) the level digraphs of the tree T rooted at i, and the corresponding topological orders of connected components of level digraphs are depicted.

the set of all the cut edges is formed, we eliminate vertices that are not potential BFS roots, as described above. We get a set $R \subseteq V(G)$ of all the potential BFS roots of T. The interesting result from [33] is that all the vertices in R induce a subtree of T. A tree T_C induced by the set R is a subtree of T and is called a *core subtree*. Given an arbitrary vertex $x \in R$, let $c_1, \ldots, c_k$ be neighbors of x that are not contained in T_C. Then the rooted subtree T_x of T containing vertex x as a root, $c_1, \ldots, c_k$, and all their descendants away from x is called the *side subtree of* x. Using all these concepts, Manber showed the following theorem that characterizes the valid BFS trees.

Theorem 5.10 (Manber [33])**.** *Given a graph G and a spanning tree T in G, it holds that T is a valid BFS tree of G if and only if every side subtree T_v is a valid BFS tree of $G[V(T_v)]$.*

Algorithm 11: Algorithm which decides whether T is a BFS tree of G.

Input: Connected graph G and spanning tree T of G.

Output: T is an $\mathcal{F}$-tree of BFS on G or not.

```
1  begin
2      choose any root r ∈ V(G);
3      E_C = ∅;
4      R = V(T);
5      foreach v ∈ V \ {r} do
6          v.level ← d_T(r, v)
7      foreach uv ∈ E(G) \ E(T) do
8          a ← the lowest common ancestor of u and v;
9          find middle vertices and cut-edges corresponding to edge uv;
10         add cut edges to the set E_C
11     foreach (u → v) ∈ E_C do
12         X = {vertices that are not potential BFS roots with respect to
               (u → v)};
13         R = R \ X
14     T_C ← T[R];
15     foreach v ∈ R do
16         T_v ← side subtree of v;
17         use Algorithm 10 for graph G[V(T_v)], tree T_v and root v;
18         if T_v is not a BFS tree of G[V(T_v)] rooted at v then
19             return T is not a BFS tree of G
20     return T is a BFS tree of G rooted at any vertex v ∈ R
```

Once the core subtree and the corresponding side subtrees are generated, Algorithm 10 is used to check for each side subtree whether it is a valid BFS tree. If it turns out that T is a valid BFS tree of G for some vertex in R, then this is true for any vertex in R. This means that the algorithm can have two possible outputs: either T is not a valid BFS tree, or it is a valid BFS tree with a root being any vertex in R. Proofs omitted here can be found in original paper by Manber. All the steps of the algorithm are described in Algorithm 11. The algorithm can be implemented in linear time, using the *-tree data structure [33].

5.2.2 NP-hardness

In this section we present the known polynomial-time reductions that show NP-hardness of the search tree recognition problem for LexBFS, LexDFS, MCS and MNS. All the reductions were developed by Beisegel et al. [4], where all the details can be found.

LexBFS

In order to prove that the $\mathcal{F}$-tree recognition problem is NP-hard, we use the 3-SAT PROBLEM. Let $\mathcal{I}$ be an instance of a 3-SAT PROBLEM with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$. In the following describe the construction of a graph G and a spanning tree T in G.

- The vertex set of G is equal to $V(G) = X \cup C_1 \cup C_2 \cup \cdots \cup C_m \cup \{p, q, r, u\}$. X is a set of literal vertices $\{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$ representing literals of $\mathcal{I}$ and $G[X]$ induces the complement of a matching where each literal x_i is matched to its negation $\overline{x_i}$, for $i \in \{1, \ldots, n\}$. For all $i \in \{1, \ldots, m\}$, set C_i consists of vertices $\{a_i, c_i, t_i\}$ that form a triangle, and every vertex c_i is adjacent to literal vertices representing literals belonging to the clause C_i. Additionally, u and r are non-adjacent vertices, adjacent to every other vertex apart from the t_i, $i \in \{1, \ldots, m\}$. Vertex p is adjacent to all vertices in X, while q is adjacent to all vertices in X and also to each vertex a_i, $i \in \{1, \ldots, m\}$. For an example of such a graph, see Fig. 27.

- T is a spanning tree of G, so $V(T) = V(G)$. The set of edges of T consists of all edges incident with vertex r, of all edges of form $c_i t_i$, $i \in \{1, \ldots, m\}$, and of edge up. These edges are depicted as thick lines in Fig. 27.

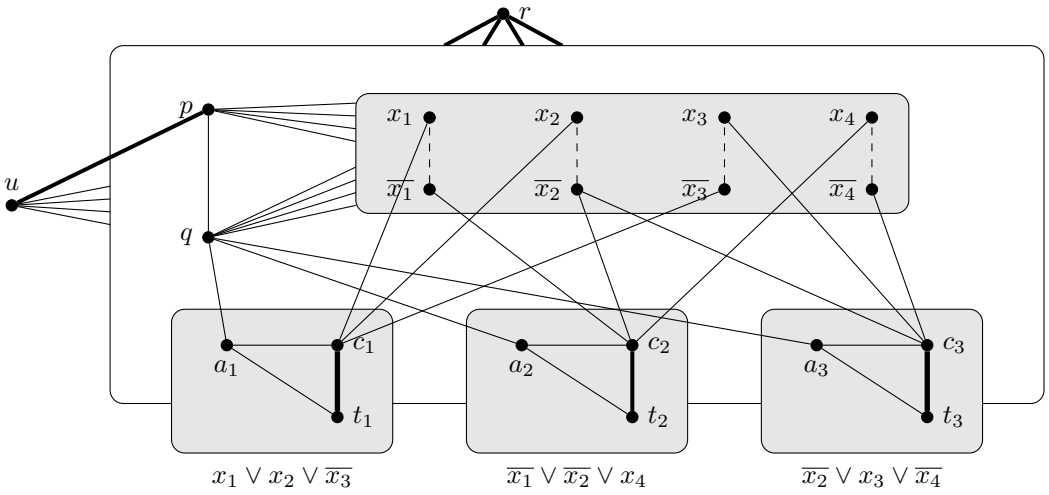


Figure 27: The NP-completeness construction for the tree recognition problem of LexBFS.

In [4] the authors showed that that the 3-SAT instance $\mathcal{I}$ has a satisfying assignment $\mathcal{A}$ if and only if T is an $\mathcal{F}$-tree of LexBFS on G. Clearly, the graph G and the tree T can be constructed in polynomial time, so it implies that the recognition of $\mathcal{F}$-tree of LexBFS is NP-hard. It can be shown that the graph obtained by the construction described above is weakly chordal, so the hardness result holds for the class of weakly chordal graphs.

Theorem 5.11 (Beisegel et al. [4])**.** *The $\mathcal{F}$-tree-recognition problem of LexBFS is NP-complete on weakly chordal graphs.*

LexDFS, MCS, and MNS

As already seen in previous chapters, every MCS ordering of some graph is also a MNS ordering of that graph. Similarly, given a graph G, every $\mathcal{F}$-tree of MCS on G is also an $\mathcal{F}$-tree of MNS on G. Beisegel et al. [4] gave a construction that reduces 3-SAT to the $\mathcal{F}$-tree recognition problem in polynomial time. Surprisingly, the construction shows the hardness result for MNS, MCS, and LexDFS. More precisely, they showed that given an arbitrary instance $\mathcal{I}$ of 3-SAT one can construct in polynomial time a graph G and a spanning tree T of G, such that the following conditions are equivalent:

i) $\mathcal{I}$ has a satisfying assignment.

ii) T is an $\mathcal{F}$-tree of MNS on G.

iii) T is an $\mathcal{F}$-tree of MCS on G.

iii) T is an $\mathcal{F}$-tree of LexDFS on G.

This equivalence does not mean that the MNS, MCS, and LexDFS trees are equivalent on the resulting graph G, but that the particular tree T corresponding to the satisfying assignment of $\mathcal{I}$ is either obtained as a result of all MCS, MNS, LexDFS, or none of them. Let us now describe the proposed polynomial reduction given by Beisegel et al [4]. Let $\mathcal{I}$ be an instance of a 3-SAT PROBLEM with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$.

- The vertex set of G is equal to $V(G) = X \cup C \cup \{a, b, p, q, r, t\}$. X is a set of literal vertices $\{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$ representing literals of $\mathcal{I}$ and $G[X]$ induces the complement of a matching where each literal x_i is matched to its negation $\overline{x_i}$, for $i \in \{1, \ldots, n\}$. C is a set of clause vertices $\{c_1, \ldots, c_m\}$. Vertices in C are pairwise non-adjacent, and every vertex $c_i \in C$ is adjacent to all vertices in X, except those representing literals included in clause C_i. Vertices a, p, r, q are adjacent to all vertices in X and C, and vertex b is adjacent to all vertices in X. Finally, we add the edges $ab, ap, aq, pr, rq, bq, bt, qt$.
- T is a spanning tree of G, so $V(T) = V(G)$. The set of edges of T consists of all edges incident with vertex r, and of the edges ap and bt. These edges are depicted as thick lines in Fig. 28.

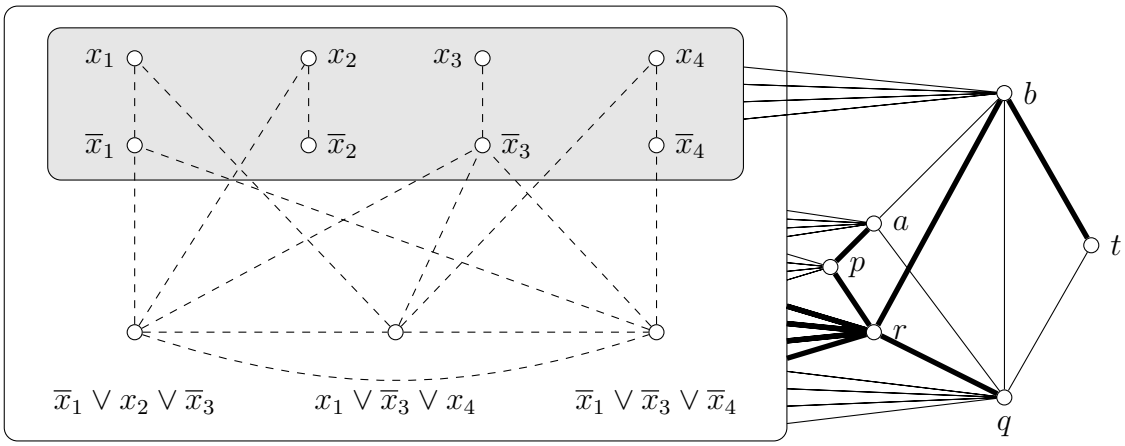


Figure 28: The $\mathcal{NP}$-completeness construction for the tree-recognition problem of MNS and MCS.

Any MNS or LexDFS resulting in the search tree T must start in r, since every other vertex is incident to an edge in G which is not an element of T. Analysing the order of visited vertices in G shows that T is a MNS tree of G if and only if it is a MCS tree of G, and also that T is a MNS tree of G if and only if it is a LexDFS tree of G. Every LexDFS, MNS, or MCS ordering of vertices in G that results in a tree T is in the one-to-one correspondence with a satisfying assignment of $\mathcal{I}$. Moreover, it holds that

the graph G is weakly chordal, so the polynomial reduction proves the NP-hardnes of the problem in the class of weakly chordal graphs.

Theorem 5.12 (Beisegel et al. [4])**.** *The $\mathcal{F}$-tree-recognition problem of LexDFS, MNS and MCS is NP-complete on weakly chordal graphs.*

6 Implementation

In this chapter we focus on polynomial-time algorithms presented in previous chapters, and present their implementation in the mathematics software SageMath. SageMath (see `www.sagemath.org`) is a free open-source mathematics software system with features covering many aspects of mathematics, including algebra, combinatorics, graph theory, numerical analysis, number theory, calculus and statistics. It builds on top of many existing open-source packages: NumPy, SciPy, matplotlib, Sympy, Maxima, GAP, FLINT, R and many more, through a common, Python-based language or directly via interfaces or wrappers. SageMath was released in 2005 by William Stein.

Functions implemented in SageMath cover various fields of mathematics, including graph theory, so there we can create graphs, and make various operations on them. The nature of these operations ranges from finding a neighborhood of some vertex, finding some particular structure in a graph, to solving more difficult problems on graphs like finding a maximum independent set in a graph, the dominanting number of a graph, etc. Many existing graph algorithms were implemented in Sage, and can be easily used. Among them there are the algorithms for graph traversing presented in this thesis, as for example the Lexicographic Breadth First Search of a graph G can be performed using the command `lex_BFS(G)`.

We present a SageMath implementation of the following polynomial-time algorithms presented in Chapter 5:

1. The algorithm that recognizes whether a tree T is a DFS search tree of a graph G rooted at vertex $r \in V(G)$ (Algorithm 8),

2. The algorithm that recognizes whether a tree T is a LexDFS search tree of a graph G rooted at vertex $r \in V(G)$ (Algorithm 9),

3. The algorithm that recognizes whether a tree T is a BFS search tree of a graph G rooted at vertex $r \in V(G)$ (Algorithm 11).

DFS-tree-recognition

Here we present the function `DFS_tree_recognition` that takes as input a graph G, a tree T and a vertex $r \in V(G)$, and checks whether T is a DFS tree of G, rooted at r,

using Algorithm 8. If the answer is positive, the function returns also the corresponding DFS ordering of G.

```
def DFS_tree_recognition(G,T,r):
    from sage.graphs.views import EdgesView
    rootedT=DiGraph(list(T.breadth_first_search(r, edges=True)))
    EG = set(EdgesView(G, labels=False, sort=True))
    ET = set(EdgesView(T, labels=False, sort=True))
    for e in EG.difference(ET):
        path1=rootedT.all_simple_paths(starting_vertices=[e[0]],
            ending_vertices=[e[1]], use_multiedges=False)
        path2=rootedT.all_simple_paths(starting_vertices=[e[1]],
            ending_vertices=[e[0]], use_multiedges=False)
        if (path1==[] and path2==[]):
            return ["T is not a DFS tree of G.", []]
    ordering = list(G.depth_first_search(r))
    return ["T is a DFS tree of G", ordering]
```

LexDFS-tree-recognition

The implementation of Algorithm 9 is stored in function `LexDFS_tree_recognition`, that takes as input a graph G, a tree T and a vertex $r \in V(G)$, and checks whether T is a spanning tree of G rooted at r. If the answer is positive, the function returns a corresponding LexDFS ordering of G.

```
def LexDFS_tree_recognition(G, T, r):
    import collections
    from collections import deque
    from sage.ext.memory_allocator import MemoryAllocator
    from sage.graphs.views import EdgesView
    S=set([r])
    V=set(range(G.order()))
    ET = EdgesView(T, labels=False, sort=True)
    nV = G.order()
    pred = [-1]* nV
    def l_func(x):
        return code[x]
    code = [collections.deque([]) for i in range(nV)]
    for v in G.neighbors(r):
        code[v].appendleft(1)
        pred[v]=r
    ordering= [r]
    label=2
    while not S==V:
        maxL=[]
        maxLabelVertex=max(V.difference(S), key=l_func)
```

```
        for u in V.difference(S):
            if (code[u]==code[maxLabelVertex] and [pred[u],u] in ET):
                maxL.append(u)
        if(maxL==[]):
            return ["T is not a LexDFS tree of G", []]
        v=maxL[0]
        S.add(v)
        ordering.append(v)
        for w in set(G.neighbors(v)).difference(S):
            code[w].appendleft(label)
            pred[w]= v
        label = label + 1
    return ["T is a LexDFS tree of G", ordering]
```

BFS-tree-recognition

In order to be able to implement Algorithm 10, we first implemented method `lowest_common_ancestor(G,r,u,v)` that takes as input a tree G and its vertices r, u, and v. The function finds a directed tree of G (denoted by `rootedG`), with r being a vertex of incoming degree 0, and returns a list containing 3 elements: a vertex w that is the lowest common ancestor of u and v in G, a child of w in the subtree of w containing u, and a child of w in the subtree of w containing v. If u is an ancestor, or descendant of v, the method outputs the empty list.

```
def lowest_common_ancestor(G,r,u,v):
    a=list(G.breadth_first_search(r, edges=True))
    rootedG=DiGraph(a)
    pathUtoV=rootedG.all_simple_paths(starting_vertices=[u],
            ending_vertices=[v], use_multiedges=False)
    pathVtoU=rootedG.all_simple_paths(starting_vertices=[u],
            ending_vertices=[v], use_multiedges=False)
    if (pathUtoV!= [] or pathVtoU!= []):
        return []
    pathV=rootedG.all_simple_paths(starting_vertices=[r],
            ending_vertices=[v], use_multiedges=False)
    pathU=rootedG.all_simple_paths(starting_vertices=[r],
            ending_vertices=[u], use_multiedges=False)
    c=min(len(pathV[0]), len(pathU[0]))
    for i in range(0,c):
        if pathV[0][i] != pathU[0][i]:
            return pathV[0][i-1], pathU[0][i], pathV[0][i]
```

Once the function `lowest_common_ancestor(G,r,u,v)` is defined, we can define the function `BFS_tree_recognition` that takes as input a graph G, a tree T and a vertex $r \in V(G)$ and decides whether T is a BFS tree of G rooted at r. In case the

answer is positive, the function produces the topological sorting of level graphs, as in Algorithm 10 and then starts the modified BFS search of G in r, so that the new vertices are added to a queue Q with respect to topological ordering of components of level graphs (where the work *component* refers to the subgraph of a level graph induced by a set of vertices having the same father). The function returns a corresponding BFS ordering of G, in case of a positive instance of the problem.

```
def BFS_tree_recognition(G,T,r):
    from sage.graphs.views import EdgesView
    from sage.graphs.distances_all_pairs import eccentricity
    from sage.graphs.connectivity import
   connected_components_subgraphs
    a=list(T.breadth_first_search(r, edges=True))
    rootedT=DiGraph(a)
    TopSort=[ [] for _ in range(T.eccentricity(r)+1) ]
    E=[ [] for _ in range(T.eccentricity(r)+1) ]
    V=[ [] for _ in range(T.eccentricity(r)+1) ]
    EG = EdgesView(G, labels=False, sort=True)
    ET = EdgesView(T, labels=False, sort=True)
    for e in EG:
        if abs(T.distance(r,e[0])-T.distance(r,e[1]))>1:
            return "T is not a BFS tree of G."
    for v in G.vertices():
        V[T.distance(r,v)].append(v)
    for e in EG:
        if not e in ET:
            u=e[0]
            v=e[1]
            if T.distance(r,e[0])-T.distance(r,e[1])==1:
                u=e[1]
                v=e[0]
            if T.distance(r,v)-T.distance(r,u)==1:
                a=lowest_common_ancestor(T,r,u,v)
                if a==[]:
                    return "Error"
                i=T.distance(r,a[0])+1
                E[i].append((a[2], a[1]))
    for i in range(T.eccentricity(r)+1):
        G = DiGraph()
        G.add_vertices(V[i])
        G.add_edges(E[i])
        if(not G.is_directed_acyclic()):
            return ["T is not a BFS tree of G.", []]
        for H in connected_components_subgraphs(G):
            TopSort[i].append(H.topological_sort())
```

```
38      Q=[r]
39      ordering = []
40      for v in Q:
41          i=T.distance(r,v)+1
42          ordering.append(v)
43          for u in set(T.neighbors(v)).difference(set(Q)):
44              for j in range(len(TopSort[i])):
45                  if (u in set(TopSort[i][j]).difference(set(Q))):
46                      for k in range(len(TopSort[i][j])):
47                          Q.append(TopSort[i][j][k])
48      return ["T is a BFS tree of G.", ordering]
```

7 Conclusion

The main topic of the thesis are graph search methods, the methods of systematic visiting the vertices of a graph so that at every step we visit a neighbor of some already visited vertex. In the thesis we described the following search methods: Breadth First Search (BFS), Depth First Search (DFS), Lexicographic Breadth First Search (LexBFS), Lexicographic Depth First Search (LexDFS), Maximum Cardinality Search (MCS), and Maximal Neighborhood Search (MNS). In the first part of the thesis we gave an overview of known results of the proposed search methods, details of efficient implementations, and structural results characterizing when a given ordering of vertices in G is a search ordering of given type in G. In the second part of the thesis we were studying the inclusion relations among various search methods, and characterized graphs in which every search ordering of a type A is also a search ordering of type B. The last part of the thesis was devoted to search trees. We were studying the problem of the search tree recognition and gave hardness results for some variants of the problem. Further, we presented polynomial-time algorithms that solve some variants of a problem (polynomial-time algorithms for solving the search tree recognition problem for BFS [33], DFS [24] and LexDFS [4]). Finally, we implemented the proposed polynomial-time algorithms in the programming language SageMath.

Results presented in Chapter 4 are developed during the writing of this thesis, and are supposed to be submitted for publication. Results presented in Chapter 5 concerning the hardness results and the polynomial-time algorithm for LexDFS are published (see [4]) and the author of the thesis is among the authors of the mentioned paper.

The first possible direction for further research could be the characterization of graphs where some of the following equivalences is true: MNS vs MCS, Generic search vs MNS (DFS, BFS, resp.). Another set of open problems consists of cases where the recognition of particular search trees is still open. In particular, the recognition of $\mathcal{L}$-trees for LexBFS, MCS, and MNS.

8 Povzetek naloge v slovenskem jeziku

Algoritmi iskanja na grafih predstavljajo enega najbolj splošnih konceptov za sistematičen pregled vozlišč grafa. Veliko računalniških algoritmov temelji na pregledu točk grafa in tako v določenih korakih svojega izvajanja uporabljajo iskanja na grafih in strukture, ki jih ta iskanja porodijo. V splošnem, iskanje na grafih predstavlja sistematičen obisk vozlišč grafa, tako da začnemo v enem vozlišču grafa in se sprehajamo po grafu, pri čemer v naslednjem koraku iteracije obiščemo vozlišče, ki ima že obiskanega soseda v tem grafu. Če ni dodatnih omejitev o izboru naslednjega obiskanega vozlišča, potem imamo t.i. *generično iskanje* na grafih, ki je obenem najbolj splošna metoda iskanja na grafih. Najbolj znani metodi iskanja na grafih sta iskanje v globino (ang. Depth First Search – DFS) in iskanje v širino (ang. Breadth First Search – BFS). Metodi sta uporabljeni pri algoritmih za iskanje poti in ciklov v grafih, za iskanje povezanih komponent itn.

Poleg omenjenih metod iskanja v globino in širino obstajajo tudi nekoliko bolj specifične, leksikografske različice metod iskanja na grafih, npr. leksikografsko iskanje v globino (LexDFS) in leksikografsko iskanje v širino (LexBFS). V splošnem je določena metoda iskanja na grafih definirana s pravilom, ki v množici potencialnih vozlišč izbere tisto, ki ga bomo obiskali v naslednjem koraku. Tako lahko izberemo vozlišče, katerega obiskana soseščina je maksimalna glede na relacijo vsebovanosti, ali glede na leksikografsko urejenost, ali glede na kardinalnost, itn. Ta pravila definirajo različne metode iskanja, kot so iskanje po maksimalni soseščini ali po maksimalni kardinalnosti, ter leksikografske oblike iskanja v globino ali širino. Zaradi svoje specifične strukture, omenjeni algoritmi iskanja na grafih omogočajo reševanje določenih problemov v linearnem času (npr. prepoznavanje tetivnosti v grafih).

Kot rezultat iskanja na grafih dobimo lahko dva koncepta: iskalni seznam vozlišč in iskalno drevo. Iskalni seznam vozlišč predstavlja razvrstitev vozlišč grafa v vrstnem redu, kot so obiskana. Iskalni seznami imajo lepe strukturne lastnosti, in so uporabljeni v različnih algoritmih. Vdak LexBFS seznam vozlišč tetivnega grafa predstavlja popolno eliminacijsko shemo tega grafa, zadnje vozlišče v LexDFS seznamu vozlišč pri določenih grafih je prvo vozlišče hamiltonske poti v grafu itn.

Iskalno drevo je vpeto drevo grafa, ki ga dobimo tako, da vsako točko grafa povežemo z enim od že obiskanih sosedov. Očitno je, da je takih vozlišč lahko več in da ni enolično določeno, s katerim predhodno obiskanim vozliščem bomo povezali trenutno vozlišče. V literaturi tako najdemo definiciji $\mathcal{F}$-dreves in $\mathcal{L}$-dreves, pri čemer je $\mathcal{F}$-drevo definirano tako, da vsako vozlišče grafa G povežemo z njegovim prvim že obiskanim sosedom v G, in $\mathcal{L}$-drevo tako, da vsako vozlišče grafa povežemo z zadnjim že obiskanim sosedom.

V magistrski nalogi smo naredili sistematično študijo algoritmov iskanja na grafih. Obravnavali smo generično iskanje, (leksikografsko) iskanje v širino in globino, iskanje po maksimalni soseščini (MNS) in iskanje po maksimalni kardinalnosti (MCS), ter karakterizirali sezname vozlišč v grafu, ki so lahko rezultat določenega iskanja na grafu. Podali smo prednosti in slabosti uporabe določenih metod grafovskega iskanja, ter posebnosti časovno učinkovite implementacije teh metod.

Osredotočili smo se na zveze med določenimi algoritmi iskanja na grafih in predstavili znane relacije vsebovanosti iskalnih seznamov vozlišč. Tako smo opazili, da je vsak LexBFS seznam vozlišč hkrati tudi BFS in MNS seznam vozlišč, in podobno je vsak LexDFS seznam vozlišč tudi DFS in MNS seznam vozlišč, ter tudi vsak MCS seznam vozlišč hkrati MNS seznam vozlišč. Študirali smo omenjene relacije vsebovanosti in karakterizirali grafe pri katerih je ta relacija vsebovanosti dejansko stroga vsebovanost. Z drugimi besedami, podali smo popolno ali delno karakterizacijo grafov, v katerih je izpolnjena ena izmed naslednjih lastnosti: vsak BFS seznam vozlišč je tudi LexBFS seznam vozlišč, ali vsak DFS seznam vozlišč je tudi LexDFS seznam vozlišč, ali vsak MNS seznam vozlišč je tudi LexBFS (LexDFS) seznam vozlišč. Podali smo primere grafov na majhnem številu točk, za katere drži, da obstja MNS seznam vozlišč, ki ni MCS seznam vozlišč.

V drugi polovici magistrske naloge smo se osredotočili na karakterizacijo sorodnega rezultata iskanja na grafih – iskalno drevo grava. Predstavili smo in opisali problem prepoznavanja iskalnih dreves grafa. Problem prepoznavanja iskalnih dreves grafa kot vhodni podatek vsebuje graf G, vpeto drevo T grafa G, in izbrano obliko iskanja na grafu, in sprašuje, ali je drevo T lahko dobljeno kot rezultat izbrane oblike iskanja na grafu G. V magistrskem delu smo naredili sistematičen pregled znanih rezultatov. Tako smo predstavili NP-težke različice problema, kot so prepoznavanje $\mathcal{F}$-dreves za metode LexBFS, LexDFS, MNS in MCS za splošne grafe. Posebno pozornost smo namenili določenim različicam problema, za katere rešitev lahko dobimo v polinomskem času. Tako smo predstavili algoritme, ki rešijo problem prepoznavanja iskalnih dreves za BFS [33], DFS [24], in LexDFS [4]. Omenjene algoritme smo implementirali v programskem okolju SageMath, pri čemer implementirani del nameravamo prispevati v javni SageMath repozitorij, da bo s tem na voljo vsem uporabnikom SageMath sistema.

9 References

[1] Jesse Beisegel. Characterising AT-free graphs with BFS. In Andreas Brandstädt, Ekkehard Köhler, and Klaus Meer, editors, *Graph-Theoretic Concepts in Computer Science*, pages 15–26, 2018. *(Cited on pages 2, 51, and 52.)*

[2] Jesse Beisegel. *Convexity in graphs: vertex order characterisations and graph searching.* PhD thesis. Brandenburgischen Technischen Universität., 2020. *(Cited on pages 53, 54, 58, and 59.)*

[3] Jesse Beisegel, Carolin Denkert, Ekkehard Köhler, Matjaž Krnc, Nevena Pivač, Robert Scheffler, and Martin Strehler. On the end-vertex problem of graph searches. *Discrete Mathematics & Theoretical Computer Science*, 21, 2019. *(Cited on page 58.)*

[4] Jesse Beisegel, Carolin Denkert, Ekkehard Köhler, Matjaž Krnc, Nevena Pivač, Robert Scheffler, and Martin Strehler. Recognizing graph search trees. *Electronic notes in theoretical computer science*, 346:99–110, 2019. *(Cited on pages 2, 53, 54, 56, 57, 58, 59, 66, 67, 68, 69, 75, and 77.)*

[5] Jesse Beisegel, Ekkehard Köhler, Robert Scheffler, and Martin Strehler. Linear time lexDFS on chordal graphs. *arXiv preprint arXiv:2005.03523*, 2020. *(Cited on page 29.)*

[6] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993. *(Cited on page 61.)*

[7] Andreas Brandstädt, Feodor F. Dragan, and Falk Nicolai. LexBFS-orderings and powers of chordal graphs. *Discrete Math.*, 171(1-3):27–42, 1997. *(Cited on pages 23 and 31.)*

[8] Anna Bretscher. *LexBFS based recognition algorithms for cographs and related families.* PhD Thesis, University of Toronto, 2005. *(Cited on page 25.)*

[9] Anna Bretscher, Derek Corneil, Michel Habib, and Christophe Paul. A simple linear time lexBFS cograph recognition algorithm. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 119–130. Springer, 2003. *(Cited on page 25.)*

[10] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971. *(Cited on page 9.)*

[11] Derek G. Corneil. Lexicographic breadth first search – a survey. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–19. Springer, 2004. *(Cited on pages 24 and 25.)*

[12] Derek G. Corneil. A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs. *Discrete Applied Mathematics*, 138(3):371–379, 2004. *(Cited on page 25.)*

[13] Derek G. Corneil, Barnaby Dalton, and Michel Habib. LDFS based certifying algorithm for the Minimum Path Cover problem on cocomparability graphs. *SIAM Journal on Computing*, 42(3):792–807, 2013. *(Cited on page 29.)*

[14] Derek G. Corneil, Jérémie Dusart, Michel Habib, and Ekkehard Kohler. On the power of graph searching for cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 30(1):569–591, 2016. *(Cited on page 29.)*

[15] Derek G. Corneil and Richard M. Krueger. A unified view of graph searching. *SIAM Journal on Discrete Mathematics*, 22(4):1259–1276, 2008. *(Cited on pages 11, 13, 15, 17, 18, 26, 27, 33, and 34.)*

[16] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. Linear time algorithms for dominating pairs in asteroidal triple-free graphs. *SIAM Journal on Computing*, 28(4):1284–1297, 1999. *(Cited on page 25.)*

[17] Derek G. Corneil, Stephan Olariu, and Lorna Stewart. The LBFS structure and recognition of interval graphs. *SIAM Journal on Discrete Mathematics*, 23(4):1905–1953, 2009. *(Cited on page 25.)*

[18] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972. *(Cited on page 14.)*

[19] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011. *(Cited on pages 1, 14, 15, and 56.)*

[20] Delbert Fulkerson and Oliver Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965. *(Cited on page 6.)*

[21] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman, 29th edition, 2002. *(Cited on pages 9 and 10.)*

[22] Martin Charles Golumbic. Trivially perfect graphs. *Discrete Mathematics*, 24(1):105–107, 1978. *(Cited on page 48.)*

[23] Michel Habib, Ross McConnell, Christophe Paul, and Laurent Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition, and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000. *(Cited on page 21.)*

[24] Torben Hagerup and Manfred Nowak. Recognition of spanning trees defined by graph searches. Technical Report A 85/08, Universität des Saarlandes, 1985. *(Cited on pages 2, 53, 54, 55, 59, 75, and 77.)*

[25] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the Association for Computing Machinery*, 21:549–568, 1974. *(Cited on pages 17 and 54.)*

[26] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972. *(Cited on page 10.)*

[27] Ekkehard Köhler and Lalla Mouatadid. Linear time lexDFS on cocomparability graphs. In *Scandinavian Workshop on Algorithm Theory*, pages 319–330. Springer, 2014. *(Cited on page 29.)*

[28] Ephraim Korach and Zvi Ostfeld. DFS tree construction: Algorithms and characterizations. In Jan van Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science*, pages 87–106, Berlin, Heidelberg, 1989. *(Cited on pages 53, 54, 55, and 56.)*

[29] Dieter Kratsch and Lorna Stewart. Domination on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 6(3):400–417, 1993. *(Cited on page 7.)*

[30] Richard Melvin Krueger. *Graph searching*. PhD thesis. University of Toronto, 2005. *(Cited on pages 15, 28, 32, and 34.)*

[31] Édouard Lucas. *Récréations mathématiques: Les traversees. Les ponts. Les labyrinthes. Les reines. Le solitaire. La numération. Le baguenaudier. Le taquin*, volume 1. Gauthier-Villars et fils, 1882. *(Cited on page 17.)*

[32] Brian Lucena. A new lower bound for tree-width using maximum cardinality search. *SIAM Journal on Discrete Mathematics*, 16(3):345–353, 2003. *(Cited on page 33.)*

[33] Udi Manber. Recognizing breadth-first search trees in linear time. *Information Processing Letters*, 34(4):167–171, 1990. *(Cited on pages 2, 59, 60, 61, 63, 65, 66, 75, and 77.)*

[34] George B. Mertzios and Derek G. Corneil. A simple polynomial algorithm for the longest path problem on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 26(3):940–963, 2012. *(Cited on page 29.)*

[35] Edward F. Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959. *(Cited on page 14.)*

[36] Stephan Olariu. Paw-free graphs. *Information Processing Letters*, 28(1):53–54, 1988. *(Cited on page 7.)*

[37] Donald J. Rose, George S. Lueker, and Robert E. Tarjan. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976. *(Cited on pages 19 and 24.)*

[38] Douglas R. Shier. Some aspects of perfect elimination orderings in chordal graphs. *Discrete Applied Mathematics*, 7(3):325–331, 1984. *(Cited on pages 35 and 36.)*

[39] Jeremy Spinrad. Efficient implementation of lexicographic depth first search. *Submitted for publication*, 2012. *(Cited on page 29.)*

[40] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972. *(Cited on pages 17, 53, 54, and 55.)*

[41] Robert E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, Jun 1976. *(Cited on page 17.)*

[42] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, 13(3):566–579, 1984. *(Cited on pages 24, 29, 32, and 35.)*

[43] Shou-Jun Xu, Xianyue Li, and Ronghua Liang. Moplex orderings generated by the lexDFS algorithm. *Discrete Applied Mathematics*, 161(13-14):2189–2195, 2013. *(Cited on page 29.)*