Fast recognition of H-free graphs

Anna Lindeberg

January 16, 2023

1 | Introduction

The classification of graphs in terms of different properties is an unavoidable direction of study within the field of graph theory. As different classes of graphs appear in theory and practice, it is a natural problem within algorithmic graph theory to decide wether or not a given graph lies in a particular graph class. Early and well-known examples of this type of decision problem, as seen in virtually any introductory course in graph theory, includes bipartite graphs, which can be recognized with a modification of breadth or deapth first search, and Eulerian graphs, which can be recognized by studying the parity of the input graph's vertex degrees. We call an algorithm like this, whose purpose is the decide (non-)membership in a graph class, a *recognition algorithm.*

Aside from the purely curious point of view, we study recognition algorithms since there are many examples where computational problems that are typically hard to solve can be answered remarkably efficiently as long as the input graph is known to belong to some particular graph class. For example, Köning's theorem (see e.g. [15, Thm. 2.1.1]) states that in a bipartite graph the size of any maximum matching equals the size of any minimum vertex cover. This can be used to construct a minimum vertex cover of bipartite graphs, although this problem is, in general, $\mathcal{NP}$–hard (for discussion on the algorithmic aspect of this, see e.g. [8, Sec. 26.3]). As another simple example, many well-known $\mathcal{NP}$–complete graph problems become tractable when the input graphs are restricted to trees. To mentioned a more sophisticated result, both optimal vertex colorings, maximum cliques and minimum vertex covers can be computed for perfect graphs in polynomial time (see [22, Ch. 9], and Section 5 for definition of perfect graphs). There are often good sources for more information on a particular graph class, e.g. [12] proves that several $\mathcal{NP}$–complete problems can be answered in linear time for cographs. An excellent overview of graph classes and computational problems restricted to families of graphs can be found in [2], and a good online encyclopaedia of graph classes in [33].

Succinctly put, graph classes are sets of graphs unified by some property. A common way to characterize or even define a property is by some forbidden substructure or collection of forbidden substructures. Perhaps the most well-known example is that of planar graphs, via Kuratowski's theorem: a graph is planar if and only if it doesn't have K_5 or $K_{3,3}$ as a minor [27] (c.f. [36], for several shorter proofs in English).

We will focus on graph classes that are characterized by one or several forbidden *induced subgraphs*, that is, subgraphs where each possible edge from the underlying graph appears. These graph classes are what we call *H–free* graph classes. Consider, for example, the two graphs in Figure 1.1, and whether or not they are C_k–free for different k, where C_k denotes the cycle graph on k vertices. On the left the wheel graph W_6 is given where four vertices and four edges are marked in red, together forming a cycle of length 4. However, these four vertices does not induce a C_4, since $\{a, c\}$ is an edge in G. With some careful consideration, one can see that W_6 is C_4–free and, similarily, C_5–free. However, a C_6

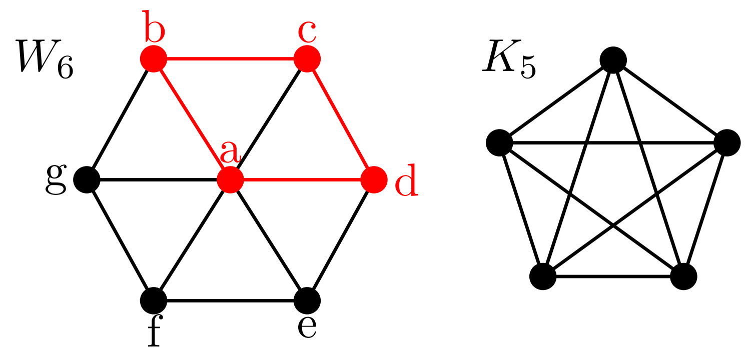

Figure 1.1: The four highlighted vertices belong to a subgraph of W_6 (left), but these vertices and the red edges does not form an induced subgraph, since $\{a, c\}$ is an edge of W_6. The graph W_6 is, for example, C_4–free and C_5–free but neither C_3–free nor C_6-free. The graph K_5 (right) is not C_3–free, but is C_k–free for all $k \geq 4$.

is induced by the six vertices of degree 3. On the right, the complete graph K_5 is given. Here any subset of the vertices of cardinality $l \leq 5$ induce a complete graph K_l. In particular, this means that K_5 is C_k–free for all $k \geq 4$.

The choice of focusing on H-free graph classes is, for one thing, a rather natural restriction of the vast number of graph classes that exists. Moreover, a characterization in terms of one or several forbidden induced subgraphs gives us more structural information about the graphs, which can aid us when we either try to construct algorithms or when we prove their correctness. This information, that is, the H–freeness, can also be thought of as a ”local” structure that have useful global properties: an interesting situation in itself.

It is also worth mentioning that if the list of forbidden subgraphs is finite, then one can always construct a naive polynomial time recognition algorithm by inspecting every element of the search space, i.e. iterating over each possible vertex subset of appropriate size and checking for isomorphism with the forbidden induced subgraphs. Since the best time complexity achievable in the context of graphs is a running time proportional to the number of vertices and edges, that is, an asymptotic running time of $O(n + m)$ for input graphs with n vertices and m edges, the naive approach performs very poorly except in the most trivial cases. With that said, it can be a comforting thought to know that a polynomial time algorithm exists before devoting time constructing an efficient recognition algorithm. In contrast, recognition of perfect graphs (which are characterized by an infinite set of forbidden subgraphs) was suspected to be a $\mathcal{NP}$–complete problem until Chudnovsky et al. constructed a polynomial time algorithm of time complexity $O(n^9)$ in 2005 [5].

The purpose of this project is to investigate, understand and describe the current state of research on recognizing H-free graphs in linear time. Which H-free graphs can we recognize efficiently, and how? Are there notable similarities or differences? Which problems in the area remain open? To this end, we will examine a handful of such graph classes and their linear recognition algorithms closely, and summarize similar results for others.

As we will see, the recognition algorithms we cover can be put into two categories: they are either degree-based or based on a particular type of graph traversal called lexicographic breadth first search (LBFS). With degree-based methods, we mean that either the collection of degrees or the ordering of the vertices with respect to non-increasing degrees aid in efficiently recognizing the particular graph class. Here we will see that split graphs and threshold graphs are excellent examples of this particular paradigm. LBFS-based recognition algorithms will follow a very specific pattern of two steps. Firstly, the vertices of the input graph is ordered via one or multiple special searches prioritizing vertices with multiple neighbors already ordered. When a satisfactory order of the vertices is achieved, some property characterizing the graph class in question is verified or refuted, thus correctly concluding membership or non-membership.

We begin with a section introducing necessary graph theoretic definitions, then consider degree-based recognition algorithms in Section 3 and LBFS-based recognition algorithms in Section 4. We finish with some closing remarks and open problems in Section 5.

2 | Preliminaries

In this section we focus on summarizing and establishing notation for the graph theoretic concepts we need. For more information, the reader is referred to any introductory text in graph theory, e.g. [15].

2.1. Graphs. A *graph* $G = (V, E)$ consists of a non-empty finite vertex set V and an edge set E where

$$E \subseteq \{\{x, y\} \,|\, x \neq y,\ x, y \in V\}.$$

We say that $G = (V, E)$ has *order* $n = |V|$ and *size* $m = |E|$. Occasionally, we let $V(G)$ and $E(G)$ denote the vertex and edge set, respectively, of G. Two vertices u and v in V are said to be *adjacent* or *neighbors* if $\{u, v\} \in E$. An edge $e = \{u, v\}$ has *endpoints* u and v, or is said to be *incident* to the vertices u and v. Two edges are *incident* if they share an endpoint.

For a graph $G = (V, E)$ we say that the collection $\mathcal{P} = \{V_1, V_2, \ldots, V_k\}$ is a *partition* of V if each V_i is a subset of V, the V_is are pairwisely disjoint, and $V = \cup_{i=1}^{k} V_i$.

The *complement* of a graph $G = (V, E)$ is the graph $\overline{G} = (V, \overline{E}$ where $\overline{E} = \{\{x, y\} \,|\, x, y \in V,\ x \neq y,\ \{x, y\} \notin E\}$. That is, the edge set of $\overline{G}$ consists precisely of the non-edges of G, and vice versa.

The *neighborhood* of a vertex v is the set

$$\mathrm{N}_G(v) = \{u \in V \,|\, \{v, u\} \in E\},$$

and the *closed neighborhood* of a vertex v is defined as $\mathrm{N}_G[v] = \mathrm{N}_G(v) \cup \{v\}$. The *degree* of v equals the cardinality of its neighborhood, and it is denoted by $\deg_G(v)$. If the graph G is understood from context we drop the subscript G.

Two graphs $G = (V, E)$ and $H = (V', E')$ are said to be *isomorphic* if there exists a bijective map $\varphi : V \to V'$ such that $\{u, v\} \in E$ if and only if $\{\varphi(u), \varphi(v)\} \in E'$. We denote this as $G \approx H$.

A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$ where the endpoints of all edges in E' lies in V'. For any set $S \subseteq V$, the *induced subgraph* $G[S]$ is the subgraph of G which has S as vertex set and the set $\{\{u, v\} \in E \,|\, u, v \in S\}$ as edge set. If $S = \{v_1, v_2, \ldots, v_k\}$ we simplify the notation as $G[S] = G[v_1, \ldots, v_k]$. If H is a graph, we say that S *induces* H *in* G if $G[S]$ and H are isomorphic.

For any $S \subseteq V$, we let $G - S$ denote the induced subgraph $G[V \setminus S]$. If $S = \{v\}$, we simplify the notation as $G - v$.

For any graphs G and H, we say that G is *H–free* if there exists no set $S \subseteq V(G)$ such that S induces H in G. If $\mathcal{H} = \{H_1, H_2, \ldots, H_k\}$ is a set, possibly infinite, of graphs, we say that G is *$\mathcal{H}$–free* if it is H_i–free for $i = 1, 2, \ldots, k$.

A *hereditary property* of a graph is a property P that exhibits the following implication:

$$\text{the graph } G \text{ satisfies } P \implies \text{every induced subgraph } H \text{ of } G \text{ satisfies } P.$$

A *path* P_k is the graph with vertex set $\{v_1, v_2, \ldots, v_k\}$ and edge set $\{\{v_i, v_{i+1}\} \mid 1 \leq i \leq k-1\}$. P_k is said to have *length* $k-1$. We sometimes identify a path P_k only by the sequence $v_1v_2 \ldots v_k$, and call v_1 and v_k *endpoints* of the path, and the remaining vertices *midpoints* of the path. For any graph G, a *path in* G is a subgraph (not necessarily induced) of G isomorphic to P_k for some k.

If there exists a path $xv_1v_2 \ldots v_ky$ between any two distinct vertices x and y in a graph G, we say that G is *connected*, otherwise it is *disconnected*. A maximal connected subgraph of a graph is called a *connected component*, or simply a *component* of the graph. Clearly, if a disconnected graph is H-free for some connected graph H, then so is each of its components, and vice versa.

A *cycle* C_k, where $k \geq 3$, is the graph with vertex set $\{v_1, v_2, \ldots, v_k\}$ and edge set $E(P_k) \cup \{\{v_1, v_k\}\}$. The *length* of a cycle C_k is the same as its size, i.e. k. As for paths, a *cycle in* G (an *induced cycle in* G) is a subgraph (an induced subgraph) of G isomorphic to C_k for some k.

A *complete graph* K_n is the graph with vertex set $\{v_1, v_2, \ldots, v_n\}$ and edge set $\{\{v_i, v_j\} \mid 1 \leq i, j \leq n, i \neq j\}$. If $S \subseteq V(G)$ for some graph G induces a K_i in G (i.e $G[S] \approx K_i$) we call S a *clique in* G.

If $E(G[S]) = \emptyset$ for some graph G and $S \subseteq V(G)$, then we say that S is an *independent set of* G.

A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two sets X and Y such that there exists no edge between two vertices of the same set. That is, both X and Y are independent sets of G. The partition $\{X, Y\}$ is called a *bipartition* of G. A well known characterization of bipartite graphs is that they are the graphs that contain no cycles of odd length. Since every odd cycle contains an induced odd cycle, the bipartite graphs are precisely the $\{C_3, C_5, C_7, \ldots\}$–free graphs. We let $K_{n,m}$ denote the *complete bipartite graph*, i.e. the graph with bipartition $\{X, Y\}$ such that $|X| = n$ and $|Y| = m$, where

$$E(K_{n,m}) = \{(x, y) \mid x \in X,\ y \in Y\}.$$

A graph G is a *tree* if it is connected and has no cycles. If it has no cycles, but is disconnected, it is called a *forest*. By definition, the forests and trees together form the $\{C_3, C_4, \ldots\}$–free graphs.

2.2. Graph classes. We will now introduce the four *graph classes*, that is, families of graphs with some common property, whose recognition algorithms compounds the bulk of this text. Other graph classes will be introduced as they appear. A thorough overview of graph classes can be found in e.g. [2, 33].

Chordal graphs. A cycle $v_1v_2 \ldots v_kv_1$ in a graph G, where $k \geq 4$, has a *chord* if there exists some $1 \leq i < j \leq k$ such that $\{v_i, v_j\}$ is an edge in G. The graph G is said to be *chordal* if every cycle in G of length at least four has a chord. Clearly, this means that a graph is chordal if and only if it is $\{C_4, C_5, \ldots\}$–free. Sometimes, chordal graphs are called *triangulated graphs*, e.g. in [21]. Examples of chordal graphs include all trees, forests and complete graphs. See also Figure 2.1.

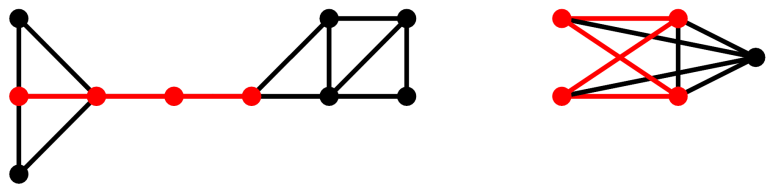

Figure 2.1: A chordal graph (left) with induced P_4 in red and the cograph $(K_1 \oplus K_1) \otimes (K_1 \otimes K_1 \otimes K_1)$ (right) with induced C_4 in red. That is, chordal graphs need not be cographs, and cographs need not be chordal.

Cographs. Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be graphs with disjoint vertex sets. We define the *disjoint union* of G and H as the graph $G \oplus H = (V_G \cup V_H, E_G \cup E_H)$. Moreover, the *join* of G and H is the graph $G \otimes H$ with vertex set $V(G \otimes H) = V_G \cup V_H$ and edge set

$$E(G \otimes H) = E_G \cup E_H \cup \{\{g, h\} \mid g \in V_G,\, h \in V_H\}.$$

With this, we can formulate the recursive definition of a *cograph*:

- K_1 is a cograph
- if G and H are cographs, then $G \oplus H$ is a cograph, and
- if G and H are cographs, then $G \otimes H$ is a cograph.

See Figure 2.1 for an example. Cographs were introduced under the name "complement reducible graphs" by Corneil et al in [12], and can be shown to be precisely the class of P_4–free graphs [12, Thm. 2]. Corneil et al. also notes that every cograph can be represented by a cotree, which is a special type of labelled trees capturing the structure of the cographs. For us, it suffices to know that cotrees can act as certificates of membership in the class of cographs, and is useful in the construction of efficient algorithms on cographs.

Split graphs. A graph $G = (V, E)$ is a *split graph* if its vertex set can be partitioned into a clique K and an independent set I. We call such a partition (K, I) a *split partition*. Split graphs was characterized as $\{C_4, C_5, 2K_2\}$–free graphs in [17] (see [21, Thm. 6.3] for a reproduction), where $2K_2$ is the graph with two disjoint edges (i.e. $2K_2 = \overline{C_4}$). See Figure 2.2 for an example.

Threshold graphs. Lastly, we introduce threshold graphs. They have appeared in literature under several equivalent definitions (c.f. [32, 21]), but is given its name from the following one: a graph $G = (V, E)$ is a *threshold graph* if there exists an integer t, called *threshold value*, and a labelling $\nu : V \to \mathbb{N}$ of the vertices such that

$$\sum_{v \in S} \nu(v) \leq t \quad \iff \quad S \text{ is an independent set of } G.$$

In terms of forbidden subgraphs, threshold graphs are the $\{C_4, 2K_2, P_4\}$–free graphs (due to [7], reproduced in [21, Thm. 10.7]). An example is given in Figure 2.3.

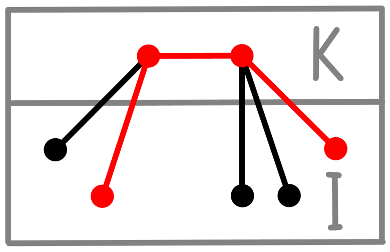

Figure 2.2: An example of a split graph with an induced P_4, so it is not a threshold graph. The split partition (K, I) of the graph is in this case unique; compare with Figure 3.1.

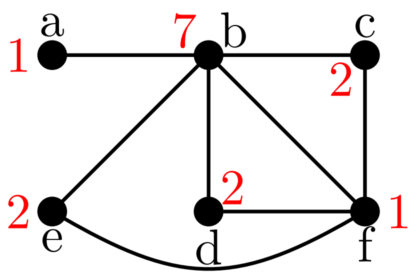

Figure 2.3: A threshold graph with threshold value $t = 7$. The threshold labeling of each vertex is marked in red, eg. $\nu(b) = 7$. Note that every independent set of the graph is either a subset of $\{a, c, d, e\}$, of $\{a, f\}$ or of $\{b\}$. In particular $\sum_{x \in S} \nu(x) \leq 7$ for each $S \subseteq \{a, c, d, e\}$, while $\sum_{x \in S} \nu(x) > 7$ if $b \in S$ and $|S| > 1$.

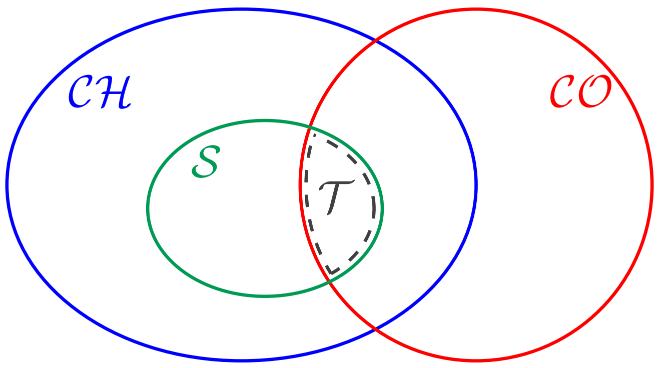

Figure 2.4: A schematic figure of containments of the classes of chordal graphs ($\mathcal{CH}$), cographs ($\mathcal{CO}$), split graphs ($\mathcal{S}$ and threshold graphs ($\mathcal{T}$). We have $\mathcal{CH} \cap \mathcal{CO} \neq \emptyset$, $\mathcal{S} \subset \mathcal{CH}$ and $\mathcal{T} = \mathcal{S} \cap \mathcal{CO}$.

These graph classes exemplify that we may deal with classes that are H–free for one forbidden subgraph H, or $\mathcal{H}$–free for either a finite or infinite set of subgraphs $\mathcal{H}$. Moreover, they act as an excellent example of how different graph classes can "overlap", and that these inclusions and intersections can easily be derived from the sets of forbidden subgraphs. In general, the following holds

Lemma 2.1 ([2, p.229])**.** *Let $\mathcal{A}$ be the class of $\mathcal{H}_1$–free graphs and $\mathcal{B}$ the class of $\mathcal{H}_2$–free graphs. Then the following holds:*

(i) If $H \in \mathcal{H}_1$ and H is an induced subgraph of the graph G, then $\mathcal{A}$ is G–free.
(ii) The class $\mathcal{A} \cap \mathcal{B}$ is the class of $(\mathcal{H}_1 \cup \mathcal{H}_2)$–free graphs.
(iii) If, for every graph G in $\mathcal{H}_2$, there is some graph in $\mathcal{H}_1$ that is an induced subgraph of G, then $\mathcal{A} \subseteq \mathcal{B}$.

For example, principle (iii) of Lemma 2.1 implies that the class of split graphs is properly contained in the class of chordal graphs, since each cycle C_k either lies in $\{C_4, C_5, 2K_2\}$ itself, or has two "opposite edges" inducing a $2K_2$.

Another slightly more complicated application of Lemma 2.1 is the following. The intersection of the class of split graphs and the class of cographs must, by property (ii) be precisely the class of $\{C_4, C_5, P_4, 2K_2\}$–free graphs. Since P_4 is an induced subgraph of C_5, every $\{C_4, P_4, 2K_2\}$–free graph is C_5–free as well, by property (i). In other words, the threshold graphs are precisely the graphs that are both split graphs and cographs.

Also recall that a chordal graphs need not be a cograph and vice versa (see Figure 2.1). Since there are certainly graphs contained in both classes (e.g. complete graphs), their intersections is non-empty as well. These containments are visualized in Figure 2.4.

2.3. Vertex orderings. As we will see, it is often helpful to order the vertices of a graph when dealing with them algorithmically. Here we formalize the notion of different vertex orderings.

Assume a graph $G = (V, E)$ of order n is given. An *ordering* of the graph's vertex set is a bijective map $\sigma : \{1, 2, \ldots, n\} \rightarrow V$. We will often use the notation $\sigma = (v_1, v_2, \ldots, v_n)$ to denote an ordering σ where $\sigma(i) = v_i$. For any ordering σ, we let σ^{-1} denote its unique inverse. Moreover, $v <_\sigma u$ denotes that v precedes u w.r.t the ordering, i.e. $\sigma^{-1}(v) < \sigma^{-1}(u)$.

We say that a vertex v of the graph $G = (V, E)$ is a *simplicial vertex*, if $\mathrm{N}(v)$ induces a clique in G. An ordering $\sigma = (v_1, v_2, \ldots, v_n)$ of V is a *perfect elimination ordering* (PEO) if, for all $1 \leq i \leq n$, v_i is simplicial in the graph $G[v_1, v_2, \ldots, v_i]$. We warn the reader that PEOs sometimes are defined as some σ where, instead, v_i is simplicial in the graph $G[v_i, v_{i+1}, \ldots, v_n]$ for each $1 \leq i \leq n$ (see e.g. in [21]). By considering the reverse $(v_n, v_{n-1}, \ldots, v_1)$ of σ the definition we use is obtained. For examples of PEOs, see Section 4.2.

A *non-increasing degree ordering* of a graph G is an ordering $\sigma = (v_1, v_2, \ldots, v_n)$ such that $\deg(v_1) \geq \deg(v_2) \geq \ldots \geq \deg(v_n)$. The *(non-increasing) degree sequence* of a graph G is an ordered sequence of integers $d = (d_1, d_2, \ldots, d_n)$ where $d_i = \deg(v_i)$ for *every* non-increasing degree ordering $\sigma = (v_1, v_2, \ldots, v_n)$ of G. In particular this means that the degree sequence is uniquely defined for each graph G, while there may exists multiple different (non-increasing) degree orderings of G.

3 | Degree orderings and sequences

We will begin by considering a handful of recognition algorithms that uses the degrees of the input graph's vertices in some way. The most rudimentary example is that of recognizing trees: as noted in any introductory text on graph theory, a graph of order n is a tree if and only if its degrees are all nonzero and sum to $2n-2$. Since all degrees can be calculated in linear time, this directly yields a fast recognition algorithm for trees. By generalizing the formula somewhat, it can be extended to recognize forests as well. Here we will begin by studying how split graphs can be recognized in linear time, followed by a section describing how threshold graphs can be recognized efficiently.

3.1. Recognizing split graphs. Recall that a graph is a split graph if it has a split partition or, equivalently, if it is $\{C_4, C_5, 2K_2\}$-free. The split partition of a split graph is not necessarily unique, as seen in Figure 3.1. There exists another characterization of split graphs in terms of its degree sequence by Hammer and Simeone in [25] which, as remarked in [2, p.203], can be used as a recognition algorithm of split graphs. The characterization is the following.

Theorem 3.1. *A graph G with non-increasing degree sequence $d = (d_1, d_2, \ldots, d_n)$ is a split graph if and only if*

$$\sum_{i=1}^{m} d_i = m(m-1) + \sum_{i=m+1}^{n} d_i \tag{3.1.1}$$

where $m = \max\{i \mid 1 \leq i \leq n,\, d_i \geq i-1\}$.

Hammer and Simeone proves this in the context of the *splittance* of a graph — a graph invariant which measures how many inserted and deleted edges are needed for a graph to become a split graph (e.g. the splittance of a split graph is zero). We instead suggest a direct proof. Before so, we say that the split partition (K, I) is a *clique-maximal split partition* if there exists no vertex $v \in I$ adjacent to every vertex in K and introduce the following lemma.

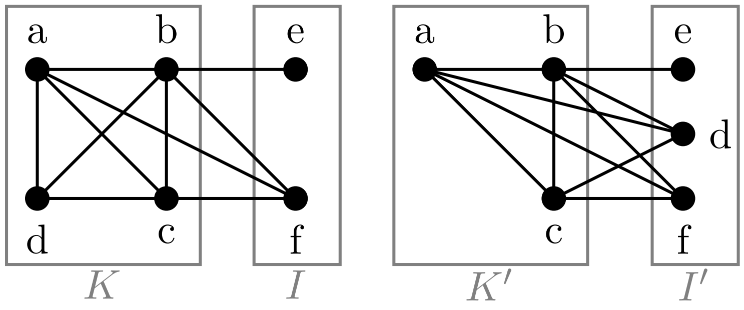

Figure 3.1: A split graph G and two different split partitions of G. Note that the partition (K, I) is clique-maximal, since neither e nor f neighbors all of K. The split partition (K', I') is not clique-maximal, since e.g. $N(d) = K'$.

Lemma 3.1. *Given a non-increasing degree ordering $\sigma = (v_1, v_2, \ldots, v_n)$ of the split graph G, there exists a clique-maximal split partition (K, I) of G such that $K = \{v_1, v_2, \ldots, v_{|K|}\}$ and $I = \{v_{|K|+1}, v_{|K|+2}, \ldots, v_n\}$.*

Proof. Let G be a split graph and let $\sigma = (v_1, v_2, \ldots, v_n)$ be a non-increasing degree ordering of G. For any clique-maximal split partition (K, I) of G the following two implications hold for all $v \in V$:

(i) $\deg(v) < |K| - 1 \implies v \in I$, and

(ii) $\deg(v) > |K| - 1 \implies v \in K$.

The implication in (i) holds since if a vertex has fewer than $|K| - 1$ neighbors, then it cannot lie in a clique of size $|K|$. Hence, it lies in I. To verify (ii), note that if $\deg(v) = |K|$, then the clique-maximality of (K, I) ensures $v \in K$. If, on the other hand, $\deg(v) > |K|$, then v has at least some neighbor in I and must therefore lie in K.

Either all, some or none of the vertices of degree $|K| - 1$ are elements of K. To deal with this, first suppose there exists vertices $u \in I$ and $v \in K$ such that $\deg(v) = \deg(u) = |K| - 1$ and $u <_\sigma v$. Since $\deg(v) = |K| - 1$ and $v \in K$ the vertex v is not adjacent to any vertex in I, in particular not to u. But then u, a vertex which has no neighbors in I, must be adjacent to all vertices of K but v. Hence $K' := (K \setminus \{v\}) \cup \{u\}$ is a clique, and $I' := (I \setminus \{u\}) \cup \{v\}$ an independent set. Moreover, this exchange of vertices preserves the clique-maximality, i.e. (K', I') is a clique-maximal split partition. Since we can repeat the exchange for any two vertices of degree $|K| - 1$ satisfying the critera above, we can always construct a clique-maximal split partition satisfying the assertion. □

The key argument of this proof is the exchange of vertices between the clique and the independent set. For a concrete example, consider Figure 3.1. Suppose we are given the non-increasing degree ordering $\sigma = (b, a, c, f, d, e)$. We have seen that the given split partition (K, I) is clique-maximal, but $K \neq \{b, a, c, f\}$. Since $N(d) = N(f) = \{a, b, c\}$, we easily establish that $(\{a, b, c, f\}, \{d, e\})$ indeed is another clique-maximal split partition, satisfying the assertion of Lemma 3.1.

With this established, we continue with

Proof of Theorem 3.1. First, let $G = (V, E)$ be a split graph with non-increasing degree ordering $\sigma = (v_1, v_2, \ldots, v_n)$. Lemma 3.1 ensures the existence of a clique-maximal split partition (K, I) of G such that $K = \{v_1, \ldots, v_{|K|}\}$ and $I = \{v_{|K|+1}, \ldots, v_n\}$. Note that $d = (d_1, \ldots, d_n)$ where $d_i = \deg(v_i)$ for each i is the non-increasing degree sequence of G. Let $E(X, Y)$ denote the set of edges in G with one endpoint in X and one endpoint in Y. If we sum the degrees of vertices in K we count each edge in the clique K twice, and each edge with an incident vertex in I only once. That is, we get

$$\sum_{i=1}^{|K|} d_i = 2|E(K, K)| + |E(K, I)| = |K|(|K| - 1) + \sum_{i=|K|+1}^{n} d_i.$$

To show (3.1.1) it thus suffices to show that $|K| = m$. Clearly $\deg(v_{|K|}) \geq |K|-1$, since $v_{|K|} \in K$. Hence $|K| \in \{i \mid 1 \leq i \leq n,\, d_i \geq i-1\}$. Thus, if $m \neq |K|$, then $m > |K|$. If so, then the vertex $v_m \in I$ satisfies $\deg(v_m) \geq m-1 > |K|-1$. This is a contraction — since (K, I) is clique-maximal and $v_m \in I$ this vertex has at most $|K|-1$ neighbors. Hence $m = |K|$, and (3.1.1) is satisfied.

Conversely, suppose $G = (V, E)$ has a non-increasing degree sequence $d = (d_1, \dots, d_n)$ satisfying (3.1.1). Let $\sigma = (v_1, v_2, \dots, v_n)$ be any fixed non-increasing degree ordering of G. Furthermore, put $K = \{v_1, \dots, v_m\}$ and $I = \{v_{m+1}, \dots, v_n\}$. For contradiction, assume that (K, I) is not a split partition of G. If K is not a clique, then the number of edges in $G[K]$ is strictly bounded by the number of edges in a clique of size $|K| = m$. That is,

$$\frac{1}{2}m(m-1) - |E(K,K)| > 0$$

On the other hand, if I is not an independent set, then $G[I]$ has at least some edge, i.e. $|E(I,I)| > 0$. Since either K is not a clique or I is not an independent set we must therefore have that

$$\frac{1}{2}m(m-1) - |E(K,K)| + |E(I,I)| > 0. \tag{3.1.2}$$

Now, note that

$$\sum_{i=1}^{m} d_i = 2|E(K,K)| + |E(K,I)| \iff |E(K,K)| = \frac{1}{2}\left(\sum_{i=1}^{m} d_i - |E(K,I)|\right)$$

and

$$\sum_{i=m+1}^{n} d_i = 2|E(I,I)| + |E(K,I)| \iff |E(I,I)| = \frac{1}{2}\left(\sum_{i=m+1}^{n} d_i - |E(K,I)|\right)$$

by similar argument as above. Inserting these values of $|E(K,K)|$ and $|E(I,I)|$ in (3.1.2) gets us

$$\frac{1}{2}\left(m(m-1) - \sum_{i=1}^{m} d_i + \sum_{i=m+1}^{n} d_i\right) > 0.$$

This contradicts our assumption in (3.1.1), so (K, I) is a split partion of G. □

With this characterization, we present the rather obvious recognition algorithm for split graphs in Algorithm 1.

Theorem 3.2. *Split graphs of order n and size m can be recognized in $O(n+m)$ time.*

Proof. Consider Algorithm 1. Its correctness follows directly from Theorem 3.1. For the linear running time, consider any input graph of order n and size m. An array of the vertices' degrees can be constructed in $O(n+m)$ time by iterating

Algorithm 1: Recognizing split graphs using a characterization of its degree sequence.

```
Input: A graph $G = (V, E)$
Output: True if $G$ is a split graph, False otherwise.
1 begin
2     $d \leftarrow (d_1, d_2, \ldots, d_n)$, a non-increasing degree sequence of $G$
3     $k \leftarrow \max\{i \mid d_i \geq i - 1\}$
4     $S_1 \leftarrow \sum_{i=1}^{k} d_i$
5     $S_2 \leftarrow k(k-1) + \sum_{i=k+1}^{n} d_i$
6     if $S_1 = S_2$ then
7         return True
8     else
9         return False
```

over the neighborhood of each vertex. Since the degrees are bounded above by n one can use e.g. Counting sort [8, Sec. 8.2] to sort the array in $O(n)$ time, so that it represents the non-increasing degree sequence of the input graph. Since the index k and the two sums can clearly be calculated in $O(n)$ time simply by iterating over the relevant values we conclude a final time complexity of $O(n + m)$. □

Although Theorem 3.1 precedes the article [26] of Heggernes and Kratsch by some 25 years, the latter two authors claim to have constructed the first recognition algorithm of split graphs.* Even though their algorithm was not the first of its kind it has one particular strength: it outputs both certificates of membership (i.e. a split partition) and non-membership (a set of vertices inducing either a C_4, a C_5 or a $2K_2$). Unsurprisingly, the calculation of the certificate in question requires some extra steps and produces a somewhat involved algorithm. We refer to [26] for details.

3.2. Recognizing threshold graphs. Threshold graphs are, as noted in the preliminaries, precisely the graphs that are both cographs and split graphs. We will see that cographs can be recognized in linear time in Section 4.3 and have just now seen that the same is true for split graphs. Obviously, this already means that threshold graphs can be recognized just as efficiently, with no new algorithms to consider! However, threshold graphs also have a characterization in terms of its degrees, originally credited to Chvátal and Hammer in [7]. To state this result, let $0 < \delta_1 < \delta_2 < \ldots < \delta_m < |V|$ be the degrees of non-isolated vertices for a given graph $G = (V, E)$. Moreover, put $\delta_0 = 0$ and $\delta_{m+1} = |V| - 1$.

*To be fair though, Hammer and Simeone never mention that the characterization in Theorem 3.1 can be used in an algorithmic context, and it does not seem to be a very wide-spread fact.

By defining

$$D_i = \{v \in V \mid \deg(v) = \delta_i\}$$

for $i = 0, 1, \ldots, m$ we obtain a partition of V into $m+1$ sets, where only D_0 can possibly be empty. We call this partition a *degree partition* of G.

Theorem 3.3 ([21, Thm. 10.4]). *Let G be a graph and let $\{D_0, \ldots, D_m\}$ a degree partition of G. Then G is a threshold graph if and only if*

$$\delta_{i+1} = \delta_i + |D_{m-i}|$$

for $i = 0, 1, \ldots, \lfloor m/2 \rfloor - 1$.

Much like in the case of recognizing split graphs, this result can immediately be repurposed as a linear-time recognition algorithm, since the δ_is and the cardinalities of the D_is can easily be extracted from a degree sequence (c.f. [21, Cor. 10.5]). The proof of Theorem 3.3 uses induction for the *only if*-direction, and constructs a threshold labeling for the *if*-direction, and require no particularly difficult steps.

It is worth mentioning that Heggernes and Kratsch also has provided a recognition algorithm of threshold graphs which outputs both a certificate of membership and of non-membership [26], but it involves no specific techniques related to threshold graphs. Perhaps more important to emphasize here, is that even though the study of how different graph classes are related in terms of inclusions surely yields edvances in the field of recognition algorithms, better algorithms can sometimes be constructed when we focus on a single graph class. Here, ”better algorithms” are not necessarily asymptotically faster, but have other good properties in terms of simplicity and implementability. For us, threshold graphs acts as a very good example of this.

3.3. Other degree-related recognition algorithms. Inspired by the $\{C_4, C_5, 2K_2\}$–free characterization of split graphs, Maffray and Preissman studied $\{C_4, 2K_2\}$–free graphs in [31]. They decided to call graphs in this class *pseudo-split* graphs, in light of the following result.

Theorem 3.4 ([31, Thm. 3]). *Let $G = (V, E)$ be a graph of order $n \geq 5$. Let $d = (d_1, d_2, \ldots, d_n)$ be the non-increasing degree sequence of G. The following statements are equivalent*

1. *G is $\{C_4, 2K_2\}$–free, but is not C_5–free.*
2. *V can be partitioned into sets X, Y and Z such that X is a clique, Y is an independent set and $G[Z]$ is a C_5. Moreover, every vertex in Z is adjacent to every vertex in X and adjacent to no vertex in Y.*
3. *With $m = \max(\{i \mid d_i \geq i+4\} \cup \{0\})$ we have*

$$\sum_{i=1}^{m} d_i = m(m+4) + \sum_{i=m+6}^{n} d_i$$

and $d_i = m+2$ for $i = m+1, m+2, \ldots, m+5$.

Since graphs of order strictly less than five are vacously C_5–free, this means that every pseudo-split graph is either a split graph or satisfies the conditions of Theorem 3.4. Algorithm 1 can thus be extended to recognize pseudo-split graphs as well as split graphs in linear time.

Another related recognition algorithm is that of trivially perfect graphs presented in [37]. This is a graph class named by Golumbic in [20], where a graph G is said to be *trivially perfect* if the number of maximum cliques in G equals the number of vertices in a maximum independent set of G. Yan et al. presents several additional characterizations of trivially perfect graphs, and to understand the one used in the recognition algorithm we need the following definitions. A *rooted tree* is a tree with a distinguished vertex r called the root, and a *rooted forest* is the disjoint union of multiple rooted trees (and thus have several roots). An *induced graph of a rooted forest* $F = (V, E)$ is the graph $G(F) = (V, E')$, where $\{u, v\} \in E'$ if and only if $u \neq v$ and there exists a path avoiding all roots of F with u and v as endpoints. The abovementioned characterization is the following.

Theorem 3.5 ([37, Thm. 3] and [20, Thm. 2]). *Let G be a graph. The following statements are equivalent.*

1. *G is a trivially perfect graph.*
2. *G is $\{P_4, C_4\}$–free.*
3. *G is a cograph and a chordal graph.*
4. *For any path $v_1 v_2 \ldots v_k$ in G such that $\deg(v_1) \geq \deg(v_2) \geq \ldots \geq \deg(v_{n-1})$, the set $\{v_1, v_2, \ldots, v_n\}$ is a clique in G.*
5. *G is induced by some rooted forest F.*

Much like in the case of threshold graphs, in Section 4 we will see that both chordal graphs and cographs are recognizable in linear time. Hence the same holds for trivially perfect graphs. However, this complicates the matter more than necessary, as Yan et al. shows in their algorithm, which is based on the last two statements of Theorem 3.5. Since the algorithm requires some arguments specific to rooted trees, we refer to [37] for details. See also Section 4.4.

4 | Lexicographic breadth first search

In this section we will see how lexicographic breadth first search can be used to recognize several H-free graph classes. These recognition algorithms follow a general paradigm of two steps: first the vertex set of the input graph is ordered, then a verification of some characterizing property of the graph class in question is performed. We begin with a general discussion of lexicographical breadth first search, followed by a section each on how it can be used to recognize chordal graphs respectively cographs. The last section review the more involved multi-sweep variants that can be used to recognize a couple of other graph classes, e.g. interval graphs.

4.1. BFS and LBFS. The idea of starting at a certain vertex and then traversing (or searching) the vertices of a graph along its edges is a necessity in algorithmic graph theory. Formally, such an approach should, at the very least, yield an ordering of the graph's vertex set.

One of the simplest and best-known ways to traverse a graph and determine an ordering σ starting at a fixed starting vertex is by a breadth-first search (BFS). In Algorithm 2 we present one way to understand BFS: we use a scheme of integer labels of vertices, where a vertex v is visited before a vertex u only if $\text{label}(v) \geq \text{label}(u)$. Moreover, if $\text{label}(v) > \text{label}(u)$, then v will precede u in σ. Positive labels are assigned to vertices as soon as they appear as a neighbor of a vertex that is currectly being dealt with (i.e. visited), and once a vertex has a positive label it remains fixed.

Algorithm 2: Using labels to find `BFS` ordering

```
Input: A graph $G = (V, E)$ of order $n$, and a vertex $v \in V$
Output: An ordering $\sigma$ of $V$
1  begin
2      $\text{label}(v) \leftarrow n$
3      $\text{label}(u) \leftarrow 0$ for all $u \in V \setminus \{v\}$
4      for $i = 1, 2, \dots, n$ do
5          Pick unvisited $u \in V$ with largest label, breaking ties arbitrarily
6          Mark $u$ as visited
7          $\sigma(i) \leftarrow u$
8          $U \leftarrow \{w \in V \mid w \in \mathrm{N}(u), \text{label}(w) = 0\}$
9          $\text{label}(w) \leftarrow n - i$ for all $w \in U$
10     return $\sigma$
```

Before continuing, it is worth mentioning that perhaps one of the most well-known applications of BFS is that of a linear time recognition algorithm of bipartite graphs. Since the bipartite graphs are C_i–free for each odd integer $i \geq 3$, and BFS can be tweaked into finding odd cycles, a linear time algorithm for bipartite graphs is readily obtained.

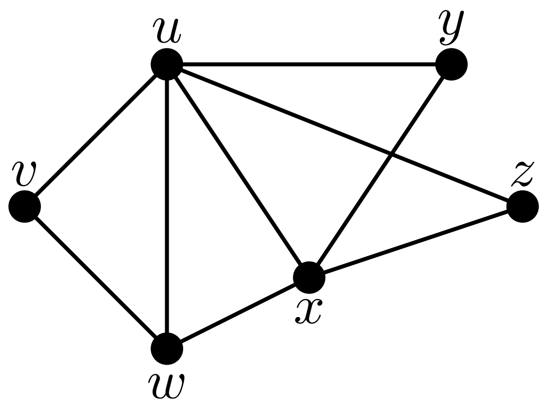

Figure 4.1: Since $d(v, u) = d(v, w) = 1$ and $d(v, x) = d(v, y) = d(v, z) = 2$ in the given graph G any run of $\texttt{BFS}(G, v)$ will order u and w after v, but before x, y, and z. The order between u and w depend on how we break ties, and the same holds for x, y and z, as seen in Figure 4.2. Instead considering $\texttt{LBFS}(G, v)$, we note that x will always be placed after u and w, but never after y nor z —— compare with Figure 4.3.

For those who have seen BFS before, this labeling scheme might seem unusual. Typically, no such labels are used in practice, since they introduce the need to make many comparisons in step 5 of Algorithm 2. Typically, a queue is used instead, since this is easily implemented to yield a linear time complexity of $O(n + m)$ for an input graph $G = (V, E)$ of order n and size m. Note, however, that we expect both variations of the algorithm to output the same order on the vertices, at least if the tie-breaking in step 5 of Algorithm 2 is made in the same way as the neighbors are added to the queue. For more information on the ”standard” BFS algorithm, see virtually any standard text on graph theory or algorithms, for example, [8, sec. 22.2].

The main idea of *lexicographic breadth-first search* (LBFS) is to break ties slightly less arbitrarily. Broadly speaking, it may seem more urgent to visit vertices that are neighbors to more than one already visited vertex, than those which are neighbors to solely one already visited vertex. Consider, for example, the graph G in Figure 4.1. In a BFS starting at vertex v the vertices u and w will be visited directly after vertex v (in arbitrary order), but depending on how we break ties we might visit either one or both of vertices y and z before vertex x, although x is a neighbor of both u and w while y and z are neighbors of u but not of w. As we will see, any run of LBFS starting at v will visit x before both y and z.

Now, with $x = (x_1, x_2, \ldots, x_s)$ and $y = (y_1, y_2, \ldots, y_t)$ as ordered tuples in $\mathbb{Z}^*$, we say that x is *lexicographically larger* than y, in symbols $x >_{\text{lex}} y$, if one of the two following conditions is satisfied:

- there is some index j, $1 \le j \le \max(s, t)$, such that $x_i = y_i$ for all $1 \le i < j$ and such that $x_j > y_j$, or
- $t < s$ and $x_i = y_i$ for all $1 \le i \le t$.

In particular, any tuple with at least one entry is larger than the empty tuple (). For example, $(6,4,7) >_{\text{lex}} (5,8,7) >_{\text{lex}} (5,8) >_{\text{lex}} ()$.

Algorithm 3 describes a small modification to the labeling scheme of Algorithm 2 which motivates the use of the word "lexicographic" in LBFS. This algorithm was introduced by Rose, Tarjan and Leuker in [35], and discussed in detail both in the survey article of Corneil [10] and in [21, Ch. 4]. The only difference between Algorithm 3 and Algorithm 2 is that we modify the labels by appending integer values, and with it keep track of which unvisited vertices that are neighbors to multiple already visited vertices. Compare Figure 4.2 and Figure 4.3 for an example of the difference between BFS and LBFS on the graph in Figure 4.1.

Clearly, Algorithm 3 will always return an ordering of the vertex set of the input graph, and is in that sense correct. We will call any ordering that is returned by a run of LBFS an *LBFS ordering* (in fact, LBFS orderings can be neatly characterized, see [11, Thm. 2.4]).

Algorithm 3: Using labels to find `LBFS` ordering

```
   Input: A graph $G = (V, E)$ of order $n$, and a vertex $v \in V$
   Output: An ordering $\sigma$ of $V$
1  begin
2      label($v$) ← ($n$)
3      label($u$) ← () for all $u \in V \setminus \{v\}$
4      for $i = 1, 2, \dots, n$ do
5          Pick unvisited $u \in V$ with lexicographically largest label,
             breaking ties arbitrarily
6          Mark $u$ as visited
7          $\sigma(i) \leftarrow u$
8          $U \leftarrow \{w \in V \mid w \in \mathrm{N}(u), w$ is not visited$\}$
9          Append $n - i$ to label($w$) for each $w \in U$
10     return $\sigma$
```

Much like BFS, we need some tricks to implement LBFS efficiently. Translating the label scheme of Algorithm 2 into its queued version is simply a question of continously arranging the unvisited vertices in the same order as their labels would have been ordered. We will do the same for LBFS, although the nature of these labels require us to use a more sophisticated queue. Consider Algorithm 4, where an ordered partition of the vertex set V is used as a sort of queue of sets. We call the sets of the partition *cells*. The idea of the ordered partition is that two vertices should belong to the same cell if and only if they, at that point of the run of the algorithm, would have been assigned the same label. Moreover, the cells of the partition should be ordered such that their corresponding (and hypothetical!) labels appear in decreasing order. To achieve these two properties a *pivot element* u, that is, the left-most vertex not yet assigned to σ, is chosen at each step of the algorithm, put in its own cell, and assigned the next number in σ

node	label	no.
v	(6)	–
u	()	–
w	()	–
x	()	–
y	()	–
z	()	–

$\rightarrow$

$i = 1$

node	label	no.
v	(6)	1
u	(5)	–
w	(5)	–
x	()	–
y	()	–
z	()	–

$\dashrightarrow$

$i = 2$

node	label	no.
v	(6)	1
u	(5)	2
w	(5)	–
x	(4)	–
y	(4)	–
z	(4)	–

$\rightarrow$

$i = 3$

node	label	no.
v	(6)	1
u	(5)	2
w	(5)	3
x	(4)	–
y	(4)	–
z	(4)	–

$\dashrightarrow$

$i = 4$

node	label	no.
v	(6)	1
u	(5)	2
w	(5)	3
x	(4)	–
y	(4)	4
z	(4)	–

$\dashrightarrow \cdots$

$\dashrightarrow$

$i = 2$

node	label	no.
v	(6)	1
u	(5)	–
w	(5)	2
x	(4)	–
y	()	–
z	()	–

$\rightarrow$

$i = 3$

node	label	no.
v	(6)	1
u	(5)	3
w	(5)	2
x	(4)	–
y	(3)	–
z	(3)	–

$\rightarrow$

$i = 4$

node	label	no.
v	(6)	1
u	(5)	3
w	(5)	2
x	(4)	4
y	(3)	–
z	(3)	–

$\dashrightarrow \cdots$

Figure 4.2: Description of the first four iterations of `BFS`(G, v) for the graph G in Figure 4.1. Dotted arrows indicate a tie between labels, and the algorithm may choose arbitrarily which vertex to continue with, while plain arrows indicate enforced choice of next vertex. Note that vertex x may or may not appear before y and z in the ordering from the output.

$i = 2$

node	label	no.
v	(6)	1
u	(5)	2
w	$(5,4)$	–
x	(4)	–
y	(4)	–
z	(4)	–

$\rightarrow$

$i = 3$

node	label	no.
v	(6)	1
u	(5)	2
w	$(5,4)$	3
x	$(4,3)$	–
y	(4)	–
z	(4)	–

$\rightarrow$

$i = 4$

node	label	no.
v	(6)	1
u	(5)	2
w	$(5,4)$	3
x	$(4,3)$	4
y	$(4,2)$	–
z	$(4,2)$	–

$\dashrightarrow \cdots$

node	label	no.
v	(6)	–
u	$()$	–
w	$()$	–
x	$()$	–
y	$()$	–
z	$()$	–

$\rightarrow$

$i = 1$

node	label	no.
v	(6)	1
u	(5)	–
w	(5)	–
x	$()$	–
y	$()$	–
z	$()$	–

$i = 2$

node	label	no.
v	(6)	1
u	$(5,4)$	–
w	(5)	2
x	(4)	–
y	$()$	–
z	$()$	–

$\rightarrow$

$i = 3$

node	label	no.
v	(6)	1
u	$(5,4)$	3
w	(5)	2
x	$(4,3)$	–
y	(3)	–
z	(3)	–

$\rightarrow$

$i = 4$

node	label	no.
v	(6)	1
u	$(5,4)$	3
w	(5)	2
x	$(4,3)$	4
y	$(3,2)$	–
z	$(3,2)$	–

$\dashrightarrow \cdots$

Figure 4.3: Description of the first four iterations of `LBFS`(G, v) for the graph G in Figure 4.1. Dotted arrows indicate a tie between labels, and the algorithm may choose arbitrarily which vertex to continue with, while plain arrows indicate enforced choice of next vertex. Note that vertex x always appear before y and z in the ordering from the output. In fact, $\sigma(4) = x$ in every possible output.

(counting upwards). The algorithm then uses `Refine`, described in Algorithm 5, to split each remaining cell into two: one containing the vertices adjacent to the pivot element, and one containting the non-adjacent vertices. An example of the algorithm is given in Figure 4.4.

Algorithm 4: Using a set-queue to find `LBFS` ordering

```
   Input: A graph $G = (V, E)$ of order $n$, and a vertex $v \in V$
   Output: An ordering $\sigma$ of $V$
1  begin
2      Initialize ordered partition of $V$ as $Q \leftarrow (\{v\}, V \setminus \{v\})$
3      for $i = 1, 2, \ldots, n$ do
4          $C \leftarrow$ next cell of $Q$
5          Select any vertex $u$ in $C$ as pivot element
6          $\sigma(i) \leftarrow u$
7          Place $u$ in its own cell
8          $U \leftarrow \{w \mid w \in \mathrm{N}(u),\ \sigma^{-1}(w)$ undefined$\}$
9          Refine($Q$, $U$)
10     return $\sigma$
```

Algorithm 5: The algorithm `Refine` for partition refinement of a queue of sets.

```
   Input: An ordered partition $(V_1, V_2, \ldots, V_k)$ of a set $V$, and a subset
          $S \subseteq V$.
   Output: A refined, ordered partition $(V_1', V_2', \ldots, V_l')$ of $V$.
1  begin
2      for $i = 1, 2, \ldots, k$ do
3          $U \leftarrow V_i \cap S$
4          if $U \neq \emptyset$ and $U \neq X_i$ then
5              Add $U$ before $X_i$ and replace $X_i$ by $X_i \setminus U$
```

To implement Algorithm 4 in $O(n+m)$ time for a graph $G = (V, E)$ of order n and size m, we need to ensure that `Refine`$(\mathcal{P}, S)$ runs in $O(|S|)$ time for any ordered partition $\mathcal{P}$ of V and any set $S \subseteq V$. As explained in [24], this can be done by representing the ordered partition of V as a doubly linked list of its cells. Additionally, the vertices of V are kept in another doubly linked list, arranged such that each cell may keep a start- and end-pointer, representing which interval in the vertex list is included in the cell in question. That is, the cell includes all vertices from the vertex at its start-pointer up until (and including) the vertex at its end-pointer. Together with a constant-time function that maps each vertex to its current cell, `Refine`$(\mathcal{P}, S)$ can be implemented by iterating over the elements of S, rather than over each cell in $\mathcal{P}$. Every vertex v in S can be moved to appear before the first vertex in its cell in constant time.

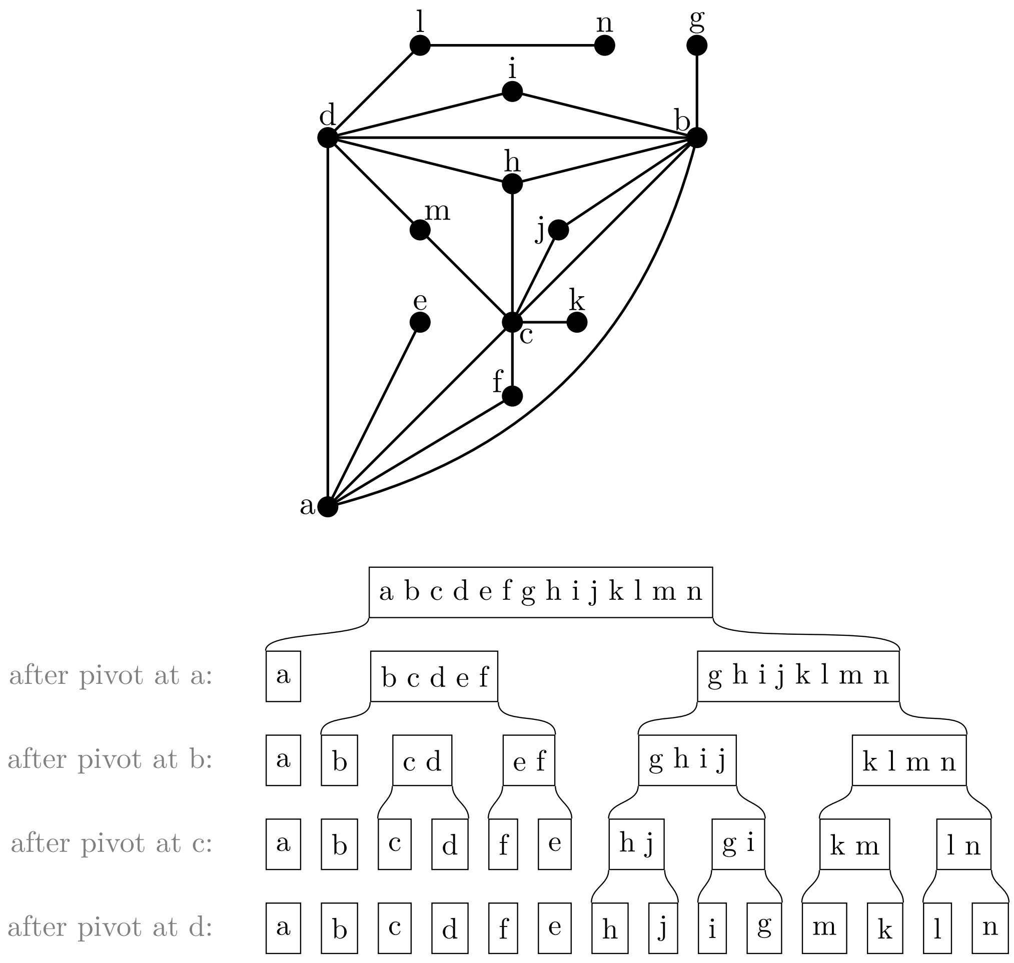
l
n
g
i
d
b
h
m
j
e
k
c
f
a
a b c d e f g h i j k l m n
after pivot at a:
a | b c d e f | g h i j k l m n
after pivot at b:
a | b | c d | e f | g h i j | k l m n
after pivot at c:
a | b | c | d | f | e | h j | g i | k m | l n
after pivot at d:
a | b | c | d | f | e | h | j | i | g | m | k | l | n

Figure 4.4: Exemple of how pivoting in Algorithm 4 can be used to run LBFS on the given graph G, starting at vertex a. Comparing to the labeled version in Algorithm 3, vertices c and d would satisfy label(c) = label(d) = $(14, 13)$ after two iterations, so vertices c and d are placed in the same cell after the second pivoting i.e. after pivoting at b. At the same time, label(e) = label(f) = $(14) <_{\text{lex}} (14, 13)$, hence their common cell is placed after the cell containing c and d.

Moreover, it is straightforward (but perhaps somewhat tedious) to verify that insertion of new cells and re-linking of start- and end-pointers from cells can similarily be made in constant time (we refer to the original algorithm in [35], if the details are of interest), resulting in a time complexity of $O(|\operatorname{N}(u)|)$ for the call to `Refine` in step 9 of Algorithm 4. This sketches the proof of the following lemma.

Lemma 4.1. *Lexicographic breadth first search can be implemented in $O(n+m)$ time for input graphs of order n and size m.*

Note that despite that we cannot implement the "labeled version" of LBFS without some more complicated data structures than solely the labels, it might be conceptually easier to think about LBFS in the way it is presented in Algorithm 3. We will shift between the two ways to think about LBFS, as to suit the context.

4.2. Recognizing chordal graphs. Recall that chordal graphs are graphs with no induced cycle of length at least 4, that is, a graph is chordal if and only if it is $\{C_4, C_5, \ldots\}$-free. LBFS can be used to recognize chordal graphs using the following two theorems.

Theorem 4.1 ([35, Thm. A])**.** *A graph is chordal if and only if its vertex set has a perfect elimination ordering.*

Theorem 4.2 ([21, Thm. 4.3])**.** *If $G = (V, E)$ is a chordal graph, and σ is an LBFS ordering of V, then σ is a perfect elimination ordering.*

This already gives us a clear picture of how we can recognize chordal graphs: we find an LBFS ordering σ, and verify whether or not it is a PEO. It is worth mentioning that in the original version of [35] the LBFS algorithm took no starting vertex as input (it picked an arbitary vertex first) and constructed an LBFS order "backwards", so that the obtained ordering need to be reversed before testing if it is a PEO (or, as in [21], the definition of PEO needs to be "reversed"), which frankly, seems unnecessary.

Before we continue with proving the two theorems, let us consider two examples. Firstly, recall that one possible LBFS ordering on the graph G in Figure 4.1 was

$$\sigma = (v, u, w, x, y, z).$$

This ordering is a PEO, as verified in Figure 4.5. It is easy to verify that G indeed is a chordal graph. A non-example of a PEO is the LBFS ordering

$$\pi = (a, b, c, d, f, e, h, j, i, g, m, k, l, n)$$

from Figure 4.4 since, for example, the induced subgraph on $\operatorname{N}(h)$ in $G[a, b, c, d, f, e, h]$ is no clique — see Figure 4.6. The underlying graph is not chordal, since e.g. the vertices a, c, d and h induces a C_4.

Now, we begin with a lemma which originally is credited to Dirac in [16], but here stated as in [21].

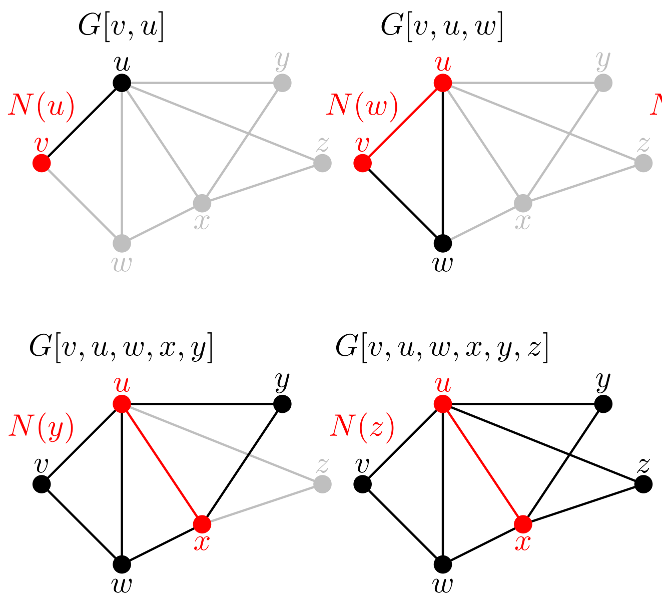

Figure 4.5: Verify that the ordering $\sigma = (v, u, w, x, y, z)$ is a perfect elimination ordering. The induced graph on the respective neighborhoods always form a clique. The shaded vertices and edges are elements of G, but not of the respective induced subgraph.

Lemma 4.2 ([21, Lem. 4.2]). *Every chordal graph has a simplicial vertex.*

Although its proof is not too complicated, only slightly technical, we omit it. The statement is only needed for the only if-statement of Theorem 4.1, and the most important result needed in showing that chordal graphs are recognizable with LBFS and PEOs is the if-statement of Theorem 4.1 together with Theorem 4.2.

Proof of Theorem 4.1. First assume $G = (V, E)$ is a graph with a perfect elimination ordering $\sigma = (v_1, v_2, \dots, v_n)$. If G has no cycle of length at least four, it is clearly chordal. Hence assume G has some cycle C with $k \geq 4$ vertices. Let v be the vertex on C such that $v >_\sigma u$ for all other vertices u on C, i.e. v is maximal on C w.r.t σ. Since σ is a PEO, v is simplicial in the graph $G[v_1, v_2, \dots, v]$. Let u and w be the two neighbors of v on C. Note that the length of the cycle ensures that u and w are not adjacent on C. However, they must both belong to the clique induced by $\mathrm{N}(v)$ in $G[v_1, v_2, \dots, v]$, since they precede v in σ. Hence the edge $\{u, w\}$ exists in G and forms a chord of C. Since C was an arbitrary cycle of length $k \geq 4$, this concludes that G is chordal.

Conversely, suppose $G = (V, E)$ is a chordal graph. If $|V| = n = 1$ the unique, trivial ordering of V vacously satisfies the property of being a perfect elimination ordering. We then proceed by induction: assume that for some fixed $k \geq 1$, every chordal graph of order k has a perfect elimination ordering, and suppose G has $k + 1$ vertices. By Lemma 4.2, G has a simplicial vertex v. Since

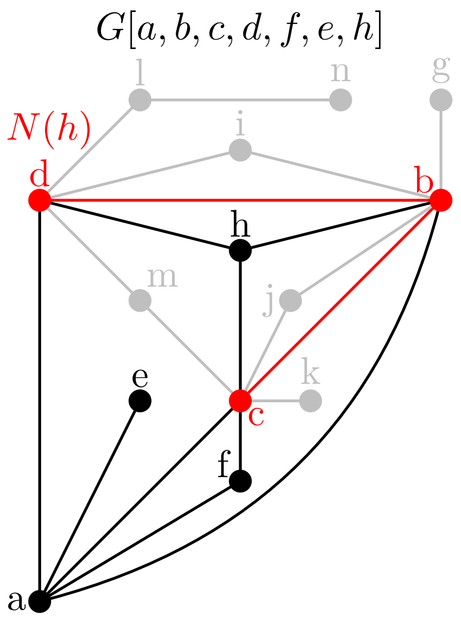

Figure 4.6: The ordering $\pi = (a, b, c, d, f, e, h, j, i, g, m, k, l, n)$ is no perfect elimination ordering, since N(h) does not form a clique in the induced subgraph $G[a, b, c, d, f, e, h]$. Note that the "missing" edge $\{c, d\}$ correspond to a "missing" chord in the cycle a, c, h, d, a. The shaded vertices and edges are elements of G, but not of the induced subgraph.

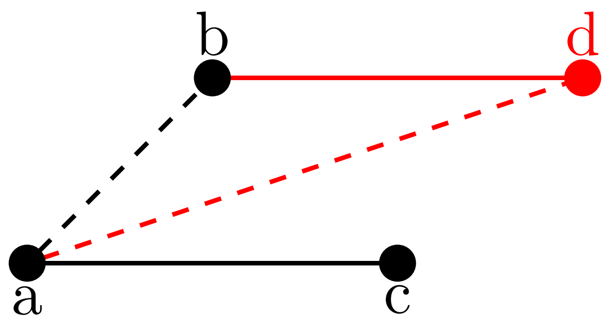

Figure 4.7: If $a >_\sigma b >_\sigma c$ for some LBFS ordering σ and c is adjacent to a but not to b, then there exists a vertex d adjacent to b but not a, such that $d <_\sigma c$.

chordality is a hereditary property, the graph $G - v$ is a chordal graph with k vertices, and thus has a PEO σ starting at some vertex other than v. We may then extend σ to the ordering $\sigma' : \{1, 2, \ldots, k+1\} \to V$ of the vertices of G by defining

$$\sigma'(i); = \begin{cases} \sigma(i) & \text{if } 1 \le i \le k \\ v & \text{if } i = k+1. \end{cases}$$

Since σ is a PEO of $G - v$ and v is simplicial in G, σ' must be a PEO of G. □

For our second important proof we need the following lemma, which is visualized in Figure 4.7.

Lemma 4.3. *Let a, b, and c be vertices of some graph $G = (V, E)$, and let σ be an LBFS ordering of V. Suppose c precedes b, and b precedes a in σ, that is, suppose $a >_\sigma b >_\sigma c$. If $\{a, c\} \in E$ but $\{b, c\} \notin E$, then there exists a vertex d such that $c >_\sigma d$, $\{b, d\} \in E$ and $\{a, d\} \notin E$.*

Proof. Let $L_i(v)$ denote the label of vertex v after the i:th iteration of Algorithm 3. Notice that labels of vertices can only increase during the algorithm; that is, if $i < j$, then $L_i(v) \le L_j(v)$ for all vertices v. Moreover, if $L_i(v) < L_i(u)$ for some vertices v and u and a fixed index i, then $L_j(v) < L_j(u)$ for all indices $j > i$. These two remarks together imply that if a, b, and c are vertices in some graph with an LBFS ordering σ that satifies $a >_\sigma b >_\sigma c$, then $L_i(a) \le L_i(b) \le L_i(c)$ for all $i = 1, 2, \ldots, n$.

Assuming that c is a neighbor of a but not of b implies that $L_{\sigma^{-1}(c)-1}(a) < L_{\sigma^{-1}(c)}(a)$ while $L_{\sigma^{-1}(c)-1}(b) = L_{\sigma^{-1}(c)}(b)$. Since $L_i(a) \le L_i(b)$ for all i, this means that the label of b must have been increased at some earlier step, when the label of a was not increased. That is, there exists some vertex d with $d <_\sigma c$ such that $L_{\sigma^{-1}(d)-1}(b) < L_{\sigma^{-1}(d)}(b)$ and $L_{\sigma^{-1}(d)-1}(a) = L_{\sigma^{-1}(d)}(a)$. For this to happen, we must have that $\{b, d\} \in E$ but $\{a, d\} \notin E$. □

This lemma, although rather easy to prove, really captures the most important structural property of an LBFS ordering: the existence of preceding vertices that act as tie-breakers.

Proof of Theorem 4.2. The proof is done by induction on the number of vertices. The statement is trivial for graphs of order $n = 1$. Then let $G = (V, E)$ be a chordal graph of order n, and assume that the assertion holds for all chordal graphs of order $n-1$. Let σ be an LBFS ordering of V and suppose $x = \sigma(n)$. Since chordality is hereditary, the graph $G - x$ is chordal. Moreover, the restriction of σ to the domain $\{1, 2, \ldots, n-1\}$ is an LBFS ordering of $G - x$, and by the hypothesis it thus suffices to show that the vertex x is simplicial in G.

Assume the contrary, namely that x, the largest vertex in σ, is not simplicial. We will show that this assumption makes it possible to construct an infinite sequence of unique vertices in the finite graph G. To this end, assume vertices x_1, x_2, ..., x_m are given, and consider the following four properties

(i) $\{x, x_i\} \in E \iff i \in \{1, 2\}$
(ii) $\{x_i, x_j\} \in E \iff |i - j| = 2$
(iii) $x_1 >_\sigma x_2 >_\sigma \ldots >_\sigma x_m$
(iv) For each j with $2 \leq j \leq m$, the vertex x_j is the smallest vertex w.r.t σ such that $\{x_{j-2}, x_j\} \in E$ but $\{x_{j-3}, x_j\} \notin E$.

Initially, we start with the case when $m = 2$, and pick x_1 and x_2 to be nonadjacent neighbors of x (they surely exist, otherwise $\mathrm{N}(x)$ would be a clique, but x is not simplicial) such that $\sigma^{-1}(x_2)$ is as small as possible. The properties (i), (ii) and (iii) are then surely satisfied. Property (iv) is satisfied if we, for notational reasons, put $x_0 = x$ and $x_{-1} = x_1$.

Now consider any sequences of vertices x_1, x_2, ..., x_m that satisfies properties (i)–(iv). Properties (ii) and (iii) ensure that the vertices x_{m-2}, x_{m-1} and x_m satisfy the hypothesis of Lemma 4.3 as a, b and c, respectively, hence we may choose a vertex x_{m+1} that is adjacent to x_{m-1} but not to x_{m-2} and such that $x_{m+1} <_\sigma x_m$. If there are multiple such vertices, we pick the one that is smallest with respect to σ. See Figure 4.8 for a visualization of the situation.

Property (i) is satisfied for this new sequence, since x_{m+1} is not adjacent to x — otherwise, it would have been chosen as x_2, since $x_{m+1} <_\sigma x_2$. By construction, the sequence x_1, x_2, ..., x_m, x_{m+1} also satisfies property (iii) and property (iv).

It remains to show property (ii), namely, to show that x_{m+1} is adjacent to x_{m-1} but no other vertices of the sequence. Firstly, if x_{m+1} were adjacent to x_{m-3}, then we could apply Lemma 4.3 to the vertices x_{m-3}, x_{m-2} and x_{m+1} as a, b and c respectively. This would imply the existence of a vertex y preceding x_{m+1}, hence preceding x_m, adjacent to x_{m-2} but not to x_{m-3}, which contradicts that property (iv) is satisfied for the sequence x_1, x_2, ..., x_m. Hence $\{x_{m+1}, x_{m-3}\} \notin E$. Secondly, x_{m+1} is not adjacent to x, x_1, x_2, ..., x_{m-4} nor to x_m, since if so, then properties (i) and (ii) would imply the existence of a chordless cycle of length at least four in G. Again, see Figure 4.8 for a visual reminder of the situation.

The extended sequence thus satisfies the four properties, and this procedure can be repeated indefinitely. But this implies G is infinite, which it is not. By

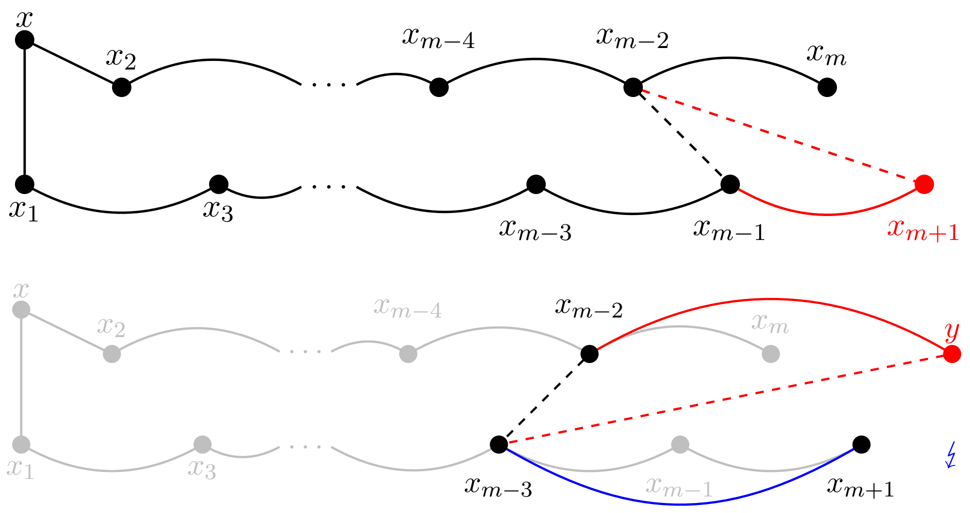

Figure 4.8: Visualization of the induction step in the proof of Theorem 4.2.

contradiction, this means that x is simplicial in G. Thus σ is a PEO of V, and the induction principle implies the assertion of the theorem. □

As mentioned earlier, the established results can be combined to recognize chordal graphs as in Algorithm 7. Before we leave the topic of recognizing chordal graphs, we need to make sure that we can verify wether or not a given ordering is a PEO. To do this in linear time we follow the approach of [21], as described in Algorithm 6 (c.f.[35] for a completely different approach). The main idea is to iterate through the LBFS ordering backwards, and continously keep track of which neighboring vertices that need to be adjacent. We will see that this yields a linear time complexity, as opposed to checking that (the restriction to preceding vertices of) each neighborhood induces a clique.

To be more precise, suppose $G = (V, E)$ is a graph and $\sigma = (v_1, v_2, \dots, v_n)$ an ordering of V. In Algorithm 6, we store a set $A(v)$, initially empty, for each vertex v. If this set contains vertices not neighboring v, then some vertex larger than v in σ is not simplicial, implying σ is no PEO. To populate these sets appropriately, we construct the set X of preceding neighbors of v. If this set contains at least two vertices, we pick its largest (with respect to the ordering) vertex u and add the remaining vertices of X to the set $A(u)$. In this way, when $i = \sigma^{-1}(u)$ either v is correctly identified to not be simplicial in $G[v_1, \dots, v_{\sigma^{-1}(v)}]$, or the neighbors of u are added to $A(w)$ for some vertex w, continuing the check of simpliciality. By this argument and observing that the cardinalities of the sets $A(u)$ are bounded by the cardinalities of $\mathrm{N}(v)$, Golumbic proves the following Lemma.

Lemma 4.4 ([21, Thm. 4.5])**.** *Let $G = (V, E)$ be a graph of order n and size m, and suppose σ is an ordering of V. Then it can be decided whether or not σ is a perfect elimination ordering in $O(n + m)$ time.*

Algorithm 6: The algorithm `Perfect` recognizes perfect elimination orderings.

```
   Input: A graph $G = (V, E)$ of order $n$, and an ordering $\sigma$ of $V$
   Output: True if $\sigma$ is a PEO, False otherwise.
1  begin
2      $A(v) \leftarrow \emptyset$ for each $v \in V$
3      for $i = n, n-1, \ldots, 2$ do
4          $v \leftarrow \sigma(i)$
5          if $A(u) \setminus \mathrm{N}(u) \neq \emptyset$ then
6              return False
7          $X \leftarrow \{x \in \mathrm{N}(v) \mid x <_\sigma v\}$
8          if $|X| > 1$ then
9              $u \leftarrow \sigma(\max\{\sigma^{-1}(x) \mid x \in X\})$
10             $A(u) \leftarrow A(u) \cup (X \setminus \{u\})$
11     return True
```

Algorithm 7: The full recognition algorithm of chordal graphs, using LBFS and Algorithm 6

```
   Input: A graph $G = (V, E)$
   Output: True if $G$ is chordal, False otherwise.
1  begin
2      Let $v$ be any vertex of $V$
3      $\sigma \leftarrow$ LBFS$(G, v)$
4      return Perfect$(G, \sigma)$
```

This means our hard work has paid off: chordal graphs are recognizable in linear time!

Corollary 4.2.1. *Chordal graphs of order n and size m can be recognized in $O(n+m)$ time.*

Proof. For correctness, note that Lemma 4.4 ensures that the LBFS ordering calculated in Algorithm 7 will correctly be identified to be, or not be, a PEO. If the ordering is a PEO, then Theorem 4.1 implies that the input graph is chordal. If the LBFS ordering is not a PEO, then the contrapositive of Theorem 4.2 ensures the input graph isn't chordal.

The linear runtime of Algorithm 7 follows immediately from Lemma 4.1 and Lemma 4.4. □

4.3. Recognizing cographs. As mentioned earlier, the chordal graphs are not the only graphs we can recognize with LBFS. In this section and next we will examine how multiple, and slightly modified, runs of LBFS on the input graph can be used to recognize these graph classes. We will begin with the cographs; recall that these graphs are characterized as the P_4-free graphs.

Let us start by example: what happens in a run of LBFS on a cograph? Consider the cograph in Figure 4.9 and the LBFS ordering $\sigma = (a, g, f, b, c, d, e)$. In contrast to, for example, the LBFS run depicted in Figure 4.4 cells are, so to speak, always left intact until its first vertex is used as a pivot. To explain this behavior, which turns out to be a characterizing trait for cographs, we need to develop some terminology on sets of vertices that are tied with respect to the LBFS labels.

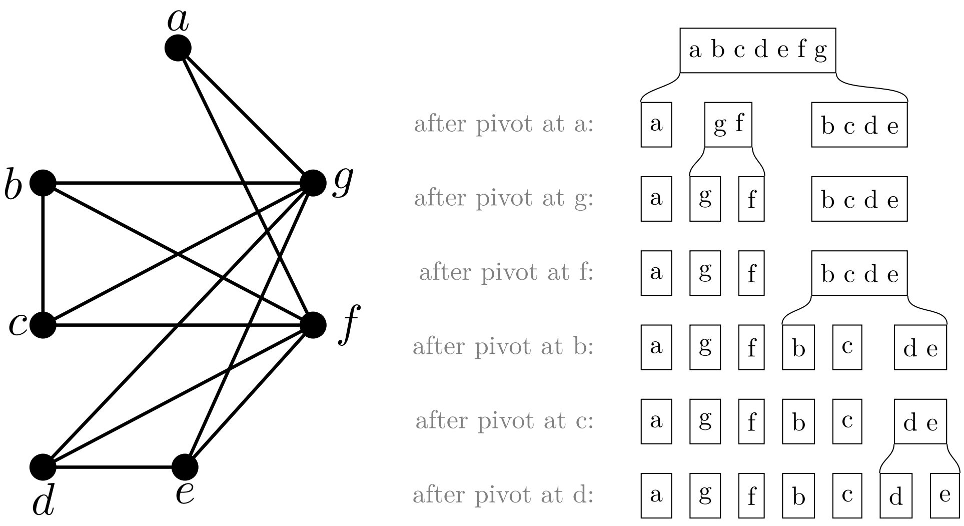

Figure 4.9: The cograph $G = (a \oplus (b \otimes c) \oplus (d \otimes e)) \otimes (g \oplus f)$ and a visualization of the cells during one possible run of LBFS(G, a). Roughly speaking, succeeding cells are left intact until one of its elements has been used as pivot, something which is not necessarily satisfied for arbitrary graphs.

Bretscher et al. defined the following concepts in [3]. Let $G = (V, E)$ be any graph, and σ any ordering of V. For any vertex $x \in V$, we let $N_i(x)$ denote the subset of $N(x)$ containing vertices strictly smaller than i w.r.t. σ. That is,

$$N_i(x) := \{u \in V \mid u \in N(x),\, \sigma(u) < i\}.$$

For each vertex $x \in V$, we let $S(x)$ denote the set of vertices tied with x when x was chosen as pivot, that is, the set of tied vertices in step 5 of Algorithm 4 or, equivalently, the cell C in step 4 of Algorithm 3. We call $S(x)$ the *slice of σ starting at x* or simply a *slice*. For notational reasons, we let the slice starting at the first vertex of σ contain all of V. Note that during an LBFS run any cell containing solely vertices not yet assigned a value in σ is either a slice or the disjoint union of several slices. Moreover, it is simply observed that the restriction of an LBFS ordering to a slice $S(x)$ is an LBFS ordering of the induced subgraph $G[S(x)]$.

Examples of slices in Figure 4.9 include $S(a) = \{a, b, c, d, e, f, g\}$, $S(b) = \{b, c, d, e\}$ and $S(c) = \{c\}$. Some slices of Figure 4.4 are $S(b) = \{b, c, d, e, f\}$ and $S(d) = \{d\}$. Note that the cell containing vertices e and f after pivoting at vertex b in Figure 4.4 is the union of the slices $S(e)$ and $S(f)$. However, every cell in Figure 4.9 containing unnumbered vertices is always a slice starting at one of its vertices, e.g. the cell $\{b, c, d, e\}$ is precisely the slice $S(b)$. As we will see, the same behavior can be observed in *any* LBFS run on *any* cograph.

Although not apperant in these smaller examples, one technical difficulty here is that for larger graphs slices can be deeply nested, so to speak. Let us first investigate the "local" level closer. Given a slice $S(x)$ we define $S^A(x) := S(x) \cap N(x)$ or, equivalently, $S^A(x) := S(x')$ where x' is the vertex immediately succeeding x in σ (formally, $x' = \sigma^{-1}(\sigma(x) + 1)$). During the LBFS run, the vertices in $S^A(x)$ will be chosen as pivots, in some order, directly after the vertex x acts as pivot — no other pivots are used between. Hence, after the last pivoting of a vertex in $S^A(x)$ the set $S(x) \cap \overline{N}(x)$ consists of the disjoint union of cells $S_1(x)$, $S_2(x)$, ..., $S_k(x)$ such that each $S_i(x)$ consists of vertices with identical neighborhoods in $S^A(x)$. We call the $S_i(x)$:s the *x-cells of $S(x)$*, and furthermore let $S^N(x)$ denote the ordered collection $(S_1(x), \dots, S_k(x))$ of all x-cells. Note that any of these sets might be empty for a particular vertex x. These definitions are visualized in Figure 4.10.

Some examples are called for at this point. In Figure 4.4 we have $S(b) = \{b, c, d, e, f\}$, $S^A(b) = \{c, d\}$ and $S^N(b) = (S_1(b), S_2(b)) = (\{f\}, \{e\})$. Examples involving empty sets are $S^A(c) = \emptyset$ with $S^N(c) = (S_1(c)) = (\{d\})$ while $S^A(d) = \emptyset$. In general, both $S^A(x)$ and $S_1(x)$ will be a slice of the underlying ordering, as long as they are non-empty. However, for $i > 1$, the non-empty x-cell $S_i(x)$ might be a union of several slices since there might be edges between the different x-cells — although not in the case of cographs. To state this formally, we say that an LBFS ordering of a graph satisfy the *x-cell condition* if there exists no edge between a vertex in $S_i(v)$ and a vertex in $S_j(v)$ for $i \neq j$ and every vertex v.

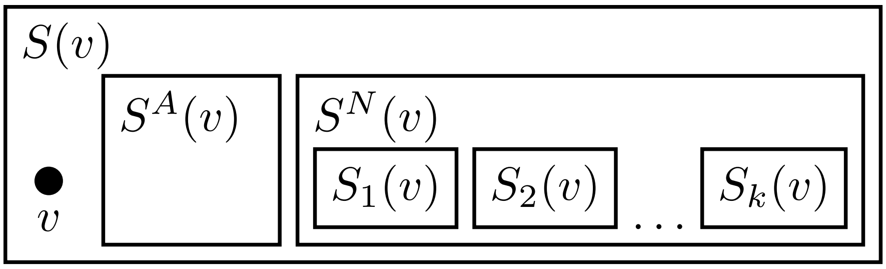

Figure 4.10: Visualization of the definitions of $S(v)$, $S^A(v)$ and $S^N(v)$. In particular $S(v) = \{v\} \cup S^A(v) \cup S_1(v) \cup \ldots \cup S_k(v)$, where one or several of the sets in the union might be empty.

Lemma 4.5 ([3, Lem. 3.6])**.** *Let $G = (V, E)$ be a cograph and suppose σ is an LBFS ordering of V. Then σ satisfies the x-cell condition.*

The proof is a simple use of contradiction: if the conclusion is not satisfied, then we can find an induced P_4. Moreover, the contrapositive statement will be used to identify graphs that are not cographs. Before we dive deeper into this identification, note that it is easy enough to verify the following observation:

Observation 4.1. *Let σ be an LBFS ordering of a P_4 starting at an endpoint and let σ' be an LBFS ordering of a P_4 starting at a midpoint. Then σ' satisfies the x-cell condition, while σ does not.*

One simple property of P_4s is that it is self-complimentary. That is, the complement of a P_4 is itself a P_4. Moreover, the midpoints of the graph $P \approx P_4$ are the endpoints of its complement $\overline{P}$, and vice versa. In light of Observation 4.1, it thus seems as if we might get away with testing for the x-cell condition on a graph G and its complement $\overline{G}$. This is, at least, a good intuitive idea of how we might recognize cographs.

The next observation we make is the following: we can, given a graph G, perform a run of LBFS on the complement $\overline{G}$ *without calculating the edge set of* $\overline{G}$. This is easiest realized by considering the labeled version of LBFS in Algorithm 3. Clearly, given any graph $G = (V, E)$, the neighbors of a vertex v in $\overline{G} = (V, \overline{E})$ are precisely the non-neighbors of v in G, and vice versa. When a vertex v is chosen as pivot, we split each current cell C into three, namely, into the single-vertex cell containing v, and the cells $C \cap N_G(v)$ and $C \cap N_{\overline{G}}(v)$. To find an LBFS ordering of $\overline{G}$ we thus only need to interchange the order of how these new cells are inserted to the ordered partition of V (the "set queue"). In other words, we simply replace step 5 of Algorithm 5 by

5' | Add U *before* X_i and replace X_i by $X_i \setminus U$.

to obtain the modified subroutine $\overline{\texttt{Refine}}$.

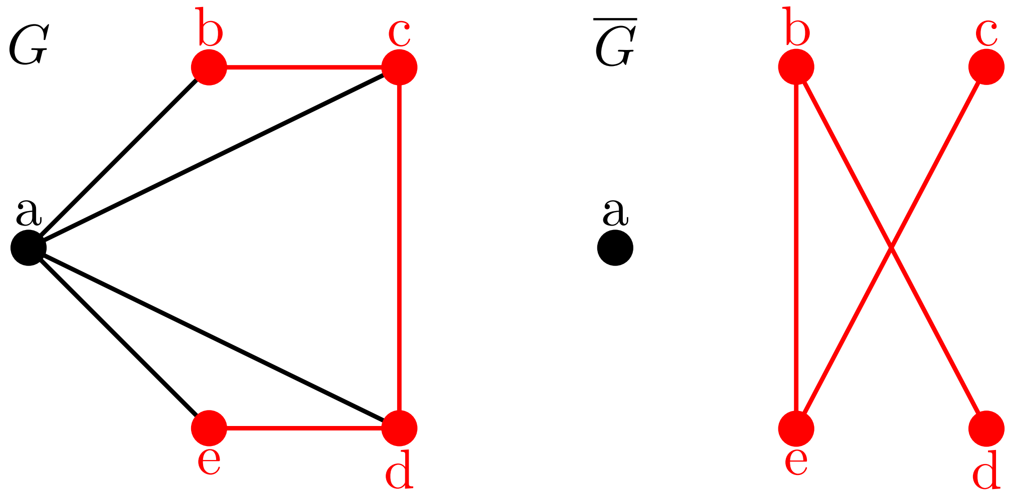

Figure 4.11: G is a graph with precicely one induced P_4, namely, *bcde*. Therefore, neither G nor $\overline{G}$ is a cograph. One possible LBFS ordering of G is $\sigma = (a, c, b, d, e)$, and one possible LBFS ordering of $\overline{G}$ is $\overline{\sigma} = (a, b, e, d, c)$. Moreover, the run of $\texttt{LBFS}^{-}(G, a)$ would yield the ordering $\overline{\sigma}^{-} = (a, c, e, b, d)$. By Observation 4.1 both σ and $\overline{\sigma}$ satisfies the x-cell condition, while $\overline{\sigma}^{-}$ does not. Hence only $\overline{\sigma}^{-}$ verifies that G is indeed *not* a cograph, by Lemma 4.5.

Unfortunately, given a graph $G = (V, E)$ where $P \subset V$ induces a P_4, performing an LBFS run on G and $\overline{G}$ is not sufficient in itself to ensure that we always manages to pick an endpoint of either the P or of $\overline{P}$. By way of example, Figure 4.11 highlights the problem. It turns out that the second modification of LBFS needed to recognize cographs has to do with how we pick the pivoting element from slices. Bretscher et al. shows that if we use an LBFS ordering σ of G when breaking ties in a run of LBFS on $\overline{G}$, then we will always manage to choose an endpoint of either P or $\overline{P}$ as the first vertex, thereby ensuring that any induced P_4 in the input graph is recognized as such. More truthfully, the modified $\overline{\texttt{LBFS}}^{-}$ in Algorithm 8 ensures that any input graph that is not P_4–free will fail the x-cell condition.

We are now ready to tie everything together in the recognition algorithm for cographs presented in Algorithm 9. Although the x-cell condition captures the situation rather well, it is a condition that is complicated to check efficiently, due to its ”local” nature, roughly put. For completeness, we now formulate the condition Bretscher et al. actually uses in Algorithm 9.

Definition 4.3. Let $G = (V, E)$ be a graph and σ an LBFS ordering of G. Fix some vertex x. We define the *local neighborhood of an x-cell $S_i(x)$*, denoted $N^l(S_i(x))$, as the set of vertices in $S(x)$ adjacent to some vertex in $S_i(v)$ which preceeds all vertices of $S_i(v)$. That is, if x_i is the smallest vertex of $S_i(x)$, then

$$N^l(S_i(x)) := \{x \in S(v) \mid x <_\sigma x_i,\ N(x) \cap S_i(v) \neq \emptyset\}.$$

Algorithm 8: The $\overline{\texttt{LBFS}}^{-}$ variant of LBFS.

```
   Input: A graph $G = (V, E)$ of order $n$, a vertex $v \in V$, and an LBFS
          ordering $\sigma$ of $V$
   Output: An ordering $\overline{\sigma}^{-}$ of the vertices of $\overline{G}$
 1 begin
 2     Initialize ordered partition of $V$ as $Q \leftarrow (\{v\}, V \setminus \{v\})$
 3     for $i = 1, 2, \ldots, n$ do
 4         $C \leftarrow$ next cell of $Q$
 5         Select vertex $u \in C$, minimal w.r.t $\sigma$, as pivot element
 6         $\overline{\sigma}^{-}(i) \leftarrow u$
 7         Place $u$ in its own cell
 8         $U \leftarrow \{w \mid w \in \mathrm{N}(u),\ w$ unnumbered$\}$
 9         $\overline{\texttt{Refine}}(Q, U)$
10     return $\overline{\sigma}^{-}$
```

Algorithm 9: The full recognition algorithm of cographs.

```
   Input: A graph $G = (V, E)$
   Output: True if $G$ is a cograph, False otherwise.
 1 begin
 2     Let $v$ be any vertex of $V$
 3     $\sigma \leftarrow \texttt{LBFS}(G, v)$              // LBFS ordering of $G$
 4     $\overline{\sigma}^{-} \leftarrow \texttt{LBFS}^{-}(G, v, \sigma)$          // LBFS$^{-}$ ordering of $\overline{G}$
 5     if $\sigma$ and $\overline{\sigma}^{-}$ satisfies the NSP on $G$ resp. $\overline{G}$ then
 6         return True
 7     else
 8         return False
```

We say that σ satisfies the *neighborhood subset property* if

$$\forall v \in V, \forall i < j \text{ s.t. } S_j(v) \neq \emptyset : \; N^l(S_i(v)) \supseteq N^l(S_j(v)).$$

Intuitively, we may think of the neighborhood subset property (NSP) as a "global" version of the x-cell condition on σ. Bretscher et al. [3] both shows that the NSP can be verified in linear time, and proves the following characterization of cographs.

Theorem 4.4 ([3, Thm. 4.3])**.** *Let $G = (V, E)$ be a cograph and suppose σ is an LBFS ordering of V. Let $\overline{\sigma}^{-}$ be the output of the run $\overline{LBFS}^{-}(G, \sigma)$. Then G is a cograph if and only if σ satisfies the NSP in G and $\overline{\sigma}^{-}$ satisfies the NSP in $\overline{G}$.*

That is, Algorithm 9 is correct. Since both LBFS, the modified $\overline{\texttt{LBFS}}^{-}$ and the check of the NSP are known to be of linear time-complexity, so is the recognition algorithm for cographs — as promised.

Lastly, two remarks: Bretcher et al. also includes specifics on how a cotree can be constructed, and how a set inducing a P_4 can be found, both in linear time. These acts as certificates of membership respectively non-membership and, more importantly, the cotree is highly useful in applied contexts (see e.g. [12]). Secondly, there exists other linear time recognition algorithms for cographs [14, 19, 23] using several different techniques, but the algorithm presented here seem to be widely recognized as the simplest one.

4.4. Other LBFS based recognition algorithms. In this section, we summarize the collection of linear time recognition algorithms which are largely based on multiple runs of LBFS or modified versions of LBFS (e.g. $\overline{\text{LBFS}}^{-}$).

Firstly, we have mentioned an efficient recognition algorithm for trivially perfect graphs earlier, and have already noted that the respective recognition algorithms for chordal graphs and cographs can be combined as well. Still, there is a perhaps even simpler LBFS-based algorithm for this class, given in [4]. It uses only a single run of LBFS, but makes sure that the vertices of the input graph initially are ordered as a non-increasing degree ordering. Moreover, it performs a check on the cells during the run of the LBFS, rather than afterwards, thus deviating slightly from the general pattern of "order and check" exhibited by these LBFS-based algorithms.

To continue, we first introduce a handful of new graph classes. A graph $G = (V, E)$ is an *interval graph* if there exists a family of intervals $\mathcal{F}$ of any linearly ordered set $\mathcal{R}$ (e.g. the real numbers) and a surjective map $\varphi : V \to \mathcal{F}$ such that $\{u, v\} \in E$ if and only if $\varphi(u) \cap \varphi(v) \neq \emptyset$. If all intervals in $\mathcal{F}$ have unit length, then G is said to be a *unit interval graph*, and if no interval of $\mathcal{F}$ is properly contained in another interval of $\mathcal{F}$, then G is said to be a *proper interval graph*. It turns out that the classes of unit interval graphs and proper interval graphs actually coincide: they are the same class, despite there different definitions (this was originally established in [34], a more recent proof can be found in [18, Thm. 1]).

Strictly speaking, interval graphs can be characterized as $\mathcal{H}$–free, but the infinite set $\mathcal{H}$ is not very easy to describe. We refer to [28, Thm. 4] for details. Moreover, unit interval graphs (and hence proper interval graphs) are precisely the $K_{1,3}$–free interval graphs [34], so unit interval graphs can also be seen as a H–free graph class.

Linear time recognition algorithms of interval graphs and unit interval graphs have been studied extensively, not only in connection to LBFS. Much like in the case of cographs, the LBFS based methods seem to be widely accepted as the simplest recognition algorithms in terms of, for example, required data structures. Li and Wu [29] present the most recent LBFS-based recognition algorithm, which simultanously recognizes both interval graphs and unit interval graphs with four sweeps of (modified) LBFS, improving on the first LBFS solution for interval graphs in [13] (which needed six sweeps of different types of modified LBFS). Another mention-worthy recognition algorithm of specifically unit interval graphs, using only three modified LBFS sweeps, can be found in [9].

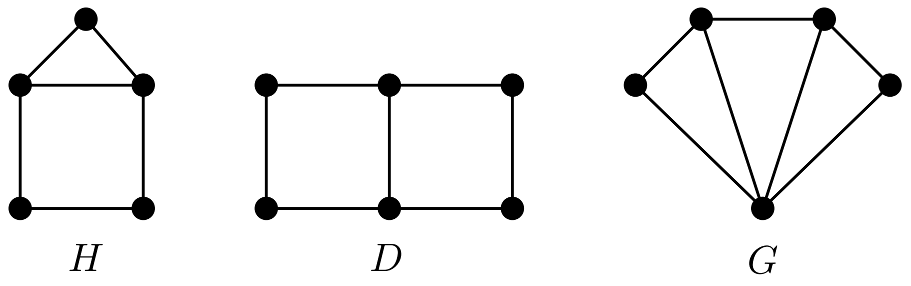

Figure 4.12: The house graph H, the domino graph D and the gem graph G appearing in the forbidden subgraph characterization of distance-hereditary graphs.

The respective introductions of [9] and [29] both include good historical notes on the progression towards simpler recognition algorithms for interval and unit interval graphs. If interested, see also the earlier mentioned survey article [10].

Another class of graphs where the use of LBFS has proven to be a successful strategy is that of *distance-hereditary graphs*. A graph lies in this class if $d_H(x, y) = d_G(x, y)$ for any vertices x and y in any connected, induced subgraph H of G. Distance-hereditary graphs are precisely the $\{H, D, G, C_5, C_6, \ldots\}$–free graphs, where the graphs the *house graph* H, the *domino graph* D and the *gem graph* G are given in Figure 4.12.

5 | Concluding remarks

We have now seen examples of linear time recognition algorithms for quite many H–free graph classes, and there are yet still others we have omitted or possibly even overlooked completely. As a final tricky endeavour, we shall try to summarize a couple of open problems within the subject, even though we cannot hope to cover everything. Again, both [2] (in particular appendix A) and [33] are great resources for navigating in the vast subject of linear time recognition algorithms and, more generally, graph classes.

Perhaps surprisingly, it is still not known whether or not members of the class of *triangle–free* graphs, i.e. C_3–free graphs, can be recognized in linear time or not, despite large combined scientific efforts. The most efficient algorithms for recognizing triangle-free graphs rely on studying the adjacency matrix-representation in various ways, and is thus dependend on the time complexity of multiplying two square matrices — a large topic in itself. See [1] for algorithms with time complexity $O(n^{\omega})$ respectively $O(m^{2\omega/(\omega+1)})$ for a graph of order n and size m, where $O(n^{\omega})$ is the time complexity of multiplying two $n \times n$ matrices. The same article give similar bounds for recognizing C_k–free graphs for $k \geq 4$.

Another interesting graph class in this context is that of *perfect graphs.* This is a large class in the sense that it, amongst other, properly contains more or less every graph class we have discussed: split graphs, threshold graphs, chordal graphs, interval graphs and cographs [21]. One definition is as follows: a graph G is *perfect* if, for every induced subgraph H of G, the size of a maximum clique of H equals the chromatic number of H. Several other definitions have occured, and were proved to be equivalent in what has become to be known as the Perfect graph theorem [30]. The characterization of perfect graphs as the $\{C_5, \overline{C_5}, C_6, \overline{C_6}, \ldots\}$–free graphs was conjectured some 60 years ago by Berge, but the proof remained elusive until 2006 [6]. At approximately the same time, some of the authors proved that perfect graphs can be recognized in polynomial time (although with a time bound of $O(n^9)$) [5]. It is ofcourse no easy task to improve this time bound, but since so many of the subclasses of the perfect graph class can be recognized efficiently the optimistic scientist can remain hopeful.

As we have seen, mostly in Section 3, different containments and relationships between graph classes can help us to linear time recognition. In its most crude form, we know that if $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are graph classes such that $\mathcal{C} = \mathcal{A} \cap \mathcal{B}$ and linear time recognition algorithms for members of $\mathcal{A}$ and $\mathcal{B}$ are known, then $\mathcal{C}$ can be recognized efficiently as well. Threshold graphs and trivially perfect graphs were example of this.

Another example of a subtle use of different containments is that of pseudo-split graphs, where any recognition algorithm of split graphs is applied as a first step, since the class of split graphs is contained in the class of pseudo-split graphs. Then the degree-characterization of Theorem 3.4 can be used to check for pseudo-split graphs that are not split.

We can thus conclude that the study of containments often help in finding linear time recognition algorithms. However, a recognition algorithm consisting of putting different recognition algorithms of other graph classes together may get rather unreasonably complicated even though it has a linear running time. If a recognition algorithm specifically constructed for the class in question can be found, implementability increases drastically.

Another important note here is that the property of being H–free has several advantages: it aids us in finding abovementioned relationships between graph classes, it is helpful in the proofs of correctness (e.g. the contradictory use of induced P_4s for the recognition algorithm of cographs) and may, if needed, act as certicates of non-membership. Hence, even though the H–freeness in itself rarely (possibly never!) yield fast recognition algorithms directly, they remain important to study.

We have also seen a general paradigm of two steps in the presented recognition algorithms: order the vertices, then check some characterizing property. This is perhaps most noticeable in the LBFS-based algorithms, but one can argue that the algorithms of Section 3 are included in this category as well, since the degree sequence and degree orderings that are used is one way of ordering the vertices. In the future, we thus might expect that further study of different search-methods will turn out to be a successful approach to finding new linear time recognition algorithms. Similarly, characterizations in terms of degree sequences remain an interesting direction of study, since they often immediately imply the existence of a very simple, linear time recognition algorithm.

References

[1] N. Alon, R. Yuster, and U. Zwick. “Finding and counting given length cycles”. In: *Algorithmica* 17.3 (1997), pp. 209–223.

[2] A. Brandstädt, V. B. Le, and J. Spinrad. *Graph Classes: A Survey.* Society for Industrial and Applied Mathematics, 1999.

[3] A. Bretscher et al. “A Simple Linear Time LexBFS Cograph Recognition Algorithm”. In: *SIAM Journal on Discrete Mathematics* 22.4 (2008), pp. 1277–1296.

[4] F.P.M. Chu. “A simple linear time certifying LBFS-based algorithm for recognizing trivially perfect graphs and their complements”. In: *Information Processing Letters* 107.1 (2008), pp. 7–12.

[5] M. Chudnovsky et al. “Recognizing Berge Graphs”. In: *Combinatorica* 25.2 (2005), pp. 143–186.

[6] M. Chudnovsky et al. “The strong perfect graph theorem”. In: *Annals of Mathematics* 164 (1 2006), pp. 51–229.

[7] V. Chvátal and P. Hammer. “Set-packing and threshold graphs”. In: *Univ. Waterloo Res. Report* CORR 73-21 (1973).

[8] T. Cormen et al. *Introduction to Algorithms.* 3rd ed. The MIT Press, 2009.

[9] D. Corneil. “A simple 3-sweep LBFS algorithm for the recognition of unit interval graphs”. In: *Discrete Applied Mathematics* 138.3 (2004), pp. 371–379.

[10] D. Corneil. “Lexicographic Breadth First Search – A Survey”. In: *Graph-Theoretic Concepts in Computer Science.* Ed. by J. Hromkovič, M. Nagl, and B. Westfechtel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–19. ISBN: 978-3-540-30559-0.

[11] D. Corneil and R. Krueger. “A Unified View of Graph Searching”. In: *SIAM Journal of Discrete Math.* 22.4 (2008), pp. 1259–1276.

[12] D. Corneil, H. Lerchs, and L. Stewart Burlingham. “Complement reducible graphs”. In: *Discrete Applied Mathematics* 3.3 (1981), pp. 163–174.

[13] D. Corneil, S. Olariu, and L. Stewart. “The LBFS Structure and Recognition of Interval Graphs”. In: *SIAM Journal on Discrete Mathematics* 23.4 (2010), pp. 1905–1953.

[14] D. Corneil, Y. Perl, and L. Stewart. “A Linear Recognition Algorithm for Cographs”. In: *SIAM Journal on Computing* 14.4 (1985), pp. 926–934.

[15] R. Diestel. *Graph Theory.* 5th ed. Graduate Texts in Mathematics. Springer, 2017.

[16] G. Dirac. “On rigid circuit graphs”. In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 25.1 (1961), pp. 71–76.

[17] S. Földes and P. Hammer. “Split graphs”. In: *Proc. 8th Southeastern Conf. on Combinatorics, Graph Theory and Computing*. Ed. by F. Hoffman and et al. Louisiana State Univ., 1977, pp. 311–315.

[18] F. Gardi. “The Roberts characterization of proper and unit interval graphs”. In: *Discrete Mathematics* 307.22 (2007), pp. 2906–2908.

[19] E. Gioan and C. Paul. “Split decomposition and graph-labelled trees: Characterizations and fully dynamic algorithms for totally decomposable graphs”. In: *Discrete Applied Mathematics* 160.6 (2012), pp. 708–733.

[20] M. Golumbic. “Trivially perfect graphs”. In: *Discrete Mathematics* 24.1 (1978), pp. 105–107.

[21] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. 2nd ed. Vol. 57. Annals of Discrete Mathematics. Elsevier, 2004.

[22] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. 2nd ed. Algorithms and Combinatorics. Springer, 1993.

[23] M. Habib and C. Paul. “A simple linear time algorithm for cograph recognition”. In: *Discrete Applied Mathematics* 145.2 (2005), pp. 183–197.

[24] M. Habib et al. “Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing”. In: *Theoretical Computer Science* 234.1 (2000), pp. 59–84. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/S0304-3975(97)00241-7`.

[25] P. Hammer and B. Simeone. “The splittance of a graph”. In: *Combinatorica* 1.3 (1981), pp. 275–284.

[26] P. Heggernes and D. Kratsch. “Linear-time certifying recognition algorithms and forbidden induced subgraphs”. In: *Nordic Journal of Computing* 14 (2007), pp. 87–108.

[27] K. Kuratowski. “Sur le problème des courbes gauches en topologie”. In: *Fund. Math.* 15 (1930), pp. 271–283.

[28] C. Lekkeikerker and J. Boland. “Representation of a finite graph by a set of intervals on the real line”. In: *Fundamenta Mathematicae* 51 (1962), pp. 45–64.

[29] P. Li and Y. Wu. “A four-sweep LBFS recognition algorithm for interval graphs”. In: *Discrete Mathematics & Theoretical Computer Science* Vol. 16 no. 3 (2014).

[30] L. Lovász. “Normal hypergraphs and the perfect graph conjecture”. In: *Discrete Mathematics* 2.3 (1972), pp. 253–267.

[31] F. Maffray and M. Preissmann. “Linear recognition of pseudo-split graphs”. In: *Discrete Applied Mathematics* 52.3 (1994), pp. 307–312.

[32] N. Mahadev and U. Peled. *Threshold Graphs and Related Topics*. Vol. 56. Annals of Discrete Mathematics. Elsevier, 1995.

[33] H. N. de Ridder et al. *Information System on Graph Classes and their Inclusions (ISGCI).* `https://www.graphclasses.org`. Feb. 22, 2022.

[34] F. Roberts. “Indifference graphs”. In: *Proof techniques in graph theory* (1969), pp. 139–146.

[35] D. Rose, R. Tarjan, and G. Lueker. “Algorithmic Aspects of Vertex Elimination on Graphs”. In: *SIAM Journal on Computing* 5.2 (1976), pp. 266–283.

[36] C. Thomassen. “Kuratowski’s theorem”. In: *Journal of Graph Theory* 5.3 (1981), pp. 225–241.

[37] J.-H. Yan, J.-J. Chen, and G. Chang. “Quasi-threshold graphs”. In: *Discrete Applied Mathematics* 69.3 (1996), pp. 247–255.