Jagiellonian University
Department of Theoretical Computer Science

Artemy Oueiski

Implementation of algorithms on cographs

Bachelor Thesis

Supervisor: dr hab. Krzysztof Turowski

June 2024

Contents

Chapter 1

Definitions

1.1 Introduction

The first definition of a cograph was introduced by Seinsche in 1974 [14]. Since then, the class of cographs has been actively studied. Many algorithms for recognizing and calculating their properties have been invented.

The most common definition of cographs is graphs do not contain P_4 as an induced subgraph. This article implements a cograph recognition algorithm using factorizing permutations. The idea of the algorithm was conceived by Michel Habib and Christophe Paul [8]. The algorithm consists of two parts. In the first part, a permutation of the graph's vertices that meets certain conditions is constructed. For this, vertex partitioning and part refinement are used. The second part of the algorithm is testing the obtained permutation of the graph. During testing, some vertices are removed from the permutation. If at the end of the testing there is exactly one vertex left in the permutation, then the graph is a cograph. Both parts of the algorithm work in linear time.

It is a well-known fact that for a cograph, there exists a canonical tree called a cotree [7]. For constructing a cotree we used the algorithm described by Michel Habib and Christophe Paul. It is based on the vertex removal order obtained in the second part of the recognition algorithm. This cotree construction algorithm works in linear time.

Using a cotree corresponding to a cograph, we can solve in linear time problems that are NP-complete in the general case. These solutions are often achieved using dynamic programming. This paper presents an algorithm for finding the pathwidth and treewidth of a cograph. The algorithm was introduced by Hans Bodlaender and Rolf Möhring [3]. Algorithms for finding the maximum clique and the maximum independent set were invented by Derek Corneil, Helmut Lerchs, and Lauren Stewart [7]. The general scheme for calculating properties of cograph is demonstrated on the example of a cograph coloring algorithm.

There are many problems with polynomial solutions in the class of cographs, in addition to those mentioned. For example: algorithm for simple max-cut [2] or hamiltonicity [6]. But not every NP-complete problem in the general case has a polynomial solution for the class of cographs. For example, problems such as determining whether a given cograph G is a subgraph of a given cograph H [12], computing the achromatic number [1], or list coloring [10] for cographs remain NP-hard.

Many problems about whether a graph can be transformed into a cograph by deleting some edges or vertices are FPT-complete. For example:

- k-edge-deletion to cographs in $O(2.562^k \cdot (m+n))$ time [9].
- k-vertex-deletion algorithm in $O(3.303^k \cdot (m+n))$ time [9].
- k-edge-edited to cographs in $O(4.612^k \cdot (m+n))$ time [11].
- a graph can be made into a cograph by deleting at most i vertices, at most j edges is fixed parameter tractable with respect to i and j [5].

Chapter 1 presents the main definitions and the structure of cograph and cotree. Chapter 2 is dedicated to the cograph recognition algorithm. In Chapter 3 there are present algorithms for finding the maximum clique, the maximum independent set, and the graph coloring. Chapter 3 also presents algorithms for finding pathwidth and treewidth. The description of the code and testing can be found in Chapter 4.

1.2 Basic definitions of graph theory

Definitions of graph theory are taken from the books by Béla Bollobás, "Modern Graph Theory" [4].

Throughout this paper we consider only finite undirected simple graphs.

Definition 1 (Graph)**.** *A* graph G *is an ordered pair of disjoint sets* (V, E) *such that* E *is the subset of the set* $\binom{V}{2}$ *of unordered pairs of* V*.*

We call the graph trivial if it consists at most one vertex.

Definition 2 (Subgraph)**.** *A graph* $G' = (V', E')$ *is a* subgraph *of* $G = (V, E)$ *if* $V' \subseteq V$ *and* $E' \subseteq E$*.*

Definition 3 (Induced subgraph)**.** *If* $G' = (V', E')$ *is a subgraph of* G *and forall* x *and* $y \in V$ *it holds that both* $V' \subseteq V$ *and* $\{x, y\} \in E'$ *if and only if* $\{x, y\} \in E'$ *, then* G' *is an* induced subgraph *of* G *and is denoted* $G[V']$*.*

Definition 4 (H-free graph)**.** *For graphs* H *and* G*, we say that* G *is* H-free *if and only if for every induced subgraph* G' *of* G*,* $G' \neq H$*.*

Definition 5 (Path)**.** *A* path *is a graph* $P = (V, E)$ *of the form*

$$V = \{x_1, x_2, \ldots, x_m\}, \quad E = \{\{x_1, x_2\}, \{x_2, x_3\}, \ldots, \{x_{m-1}, x_m\}\}$$

where for all distinct i *and* $j \in V$ *we have* $x_i \neq x_j$

This path P *is usually denoted by* $(x_1, x_2, \ldots, x_m)$*.*

We call a P_i a path consisting of i vertices.

Definition 6 (Cycle)**.** *If a graph $W = (V, E)$ where $V = (x_0, x_1 \dots, x_l)$ and*

$$E = \bigcup_{i=0}^{l-1} \{x_i, x_{i+1}\}$$

is such that $l \geq 3$, $x_0 = x_l$, and for all distinct $i, j \in [0, l-1]$ it hold that $x_i \neq x_j$, then W is called a cycle*.*

Definition 7 (Connected graph)**.** *A graph $G = (V, E)$ is* connected *if for every pair $(x, y) \subseteq V$ of distinct vertices, there is a* path *from x to y.*

Definition 8 (Forest, Tree)**.** *A graph without any cycles is a* forest*, or equivalently an acyclic graph. A* tree *is a connected forest.*

Definition 9 (Neighbours, $N(x)$)**.** *For a given vertex x in graph $G = (V, E)$, let $N(x)$ denote the neighborhood of vertex, that is $\{y \in V \colon \{x, y\} \in E\}$.*

Let us define the set $\bar{N}(x)$ as the set of all non-neighbors of vertex x in the graph that is $\{y \in V \colon y \notin N(x)\}$.

Definition 10 (Chromatic number, $\chi(G)$)**.** *The chromatic number $\chi(G)$ of a graph $G = (V, E)$ is the smallest number of colors for V so that adjacent vertices are colored differently.*

Here we introduce key definitions necessary for the analysis of trees and, in particular, cotrees.

Definition 11 (True twins and False twins)**.** *Vertices x, y are called* true twins *(*false twins*) if $N(x) \cup \{y\} = N(y) \cup \{x\}$ (respectively, if $N(x) = N(y)$).*

Definition 12 (Rooted tree)**.** *A* rooted tree *is a tree in which a special labeled node is singled out. This node is called the* root*.*

Definition 13 (Depth)**.** *The* depth *of vertex v in a rooted tree T is the distance between v and root of T.*

Definition 14 (Child)**.** *In a rooted tree $T = (V, E)$, a vertex y is a* Child *of vertex x if and only if $\{x, y\} \in E$ and vertex x belongs to the path from vertex y to the root of the tree.*

Definition 15 (Descendants, $D(x)$)**.** *For a given vertex x in graph $G = (V, E)$, let $D(x)$ denote the set of descendants of a vertex, that is, $D(x) = \{y \in V \colon$ vertex x belongs to the path from vertex y to the root of the tree $\}$*

Definition 16 (Subtree)**.** *In a tree T the induced subgraph of a vertex x and the set of all its descendants is called the* subtree *of vertex x.*

We denote the subtree of vertex x in tree T as T_x.

Definition 17 (Diameter)**.** *A* diameter *of graph G is denoted by $diam(G)$ and is maximum distance between any two vertices in G.*

Definition 18 (Least Common Ancestor, LCA)**.** *The* least common ancestor *of two nodes x and y in a tree T is the vertex with maximum depth that has both x and y as descendants.*

Definition 19 (Complement)**.** *For a graph $G = (V, E)$ the* complement *of G is a graph $\bar{G} = (V, \binom{V}{2} \setminus E)$; thus, two vertices are adjacent in $\bar{G}$ if and only if they are not adjacent in G.*

1.3 Cograph

Definitions are taken from the article "A simple linear time algorithm for cograph recognition" [8].

There are many different definitions of the class of cographs. The Theorem 1 on their equivalence was proven in the paper "Complement reducible graphs" [7].

Theorem 1 (Theorem 2 in [7])**.** *Given a graph G, the foIlowing statements are equivalent:*

1. *G is a cograph.*
2. *Any nontrivial subgraph of G has at least one pair of twins.*
3. *G does not contain P_4 as a subgraph.*
4. *The complement of any nontrivial connected subgraph of G is disconnected.*

The second statement of this theorem is proven below.

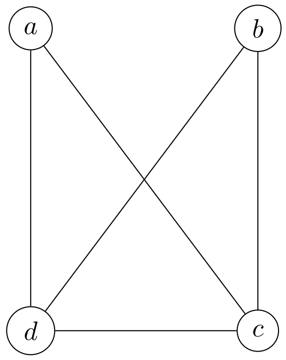
We also use the definition of a cograph through parallel and series composition operations.

Definition 20 (Parallel composition, $\cup$)**.** *A* parallel composition *of two graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$ is denoted by $G \cup H = (V_1 \cup V_2, E_1 \cup E_2)$.*

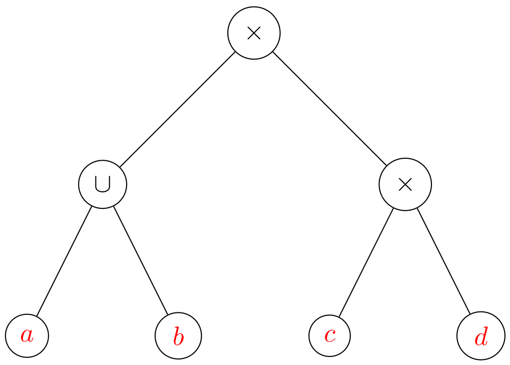
Definition 21 (Series composition, $\times$)**.** *A graph G is the* series composition *of G_1 and G_2 if $V = V_1 \cup V_2$ and $E = E_1 \cup E_2 \cup \{\{x_1, x_2\} \colon x_1 \in E_1$ and $x_2 \in E_2\}$*

Theorem 2 (Definition 1 in [8])**.** *A graph $G = (V, E)$ is a cograph if and only if one of the following conditions holds:*

- $|V| = 1$
- *There are cographs $G_1, \ldots, G_k$ such that $G_1 \cup G_2 \cup \cdots \cup G_k = G$.*
- *There are cographs $G_1, \ldots, G_k$ such that $G_1 \times G_2 \times \cdots \times G_k = G$.*



(a) Graph and it's cotree (left)



(b) Graph and it's cotree (right)

Figure 1.1: Example

Every graph can be represented as a formula using parallel compositions and series compositions of single-vertex graphs, often in multiple ways. This formula can be depicted as a tree, where each leaf corresponds to a single-vertex graph and each internal node corresponds to a parallel or series composition . In other words, each subtree of a tree corresponds to a certain cograph, and the internal nodes of the tree determine the order and type of compositions. Such a tree is called a cotree. In a canonical cotree, there are no adjacent nodes of the same type. See on an example of a graph and its corresponding cotree in Section 1.3.

The following fact follows from the construction of the cotree:

Remark 1. *In a cograph, two vertices x and y are adjacent if and only if their least common ancestor in the cotree is a series node.*

For example, in the cograph shown in the Figure 1.1a, vertices a and d are adjacent, and their least common ancestor in the cotree is a series node. On the other hand, vertices a and b are not adjacent, and their least common ancestor is a parallel node.

Theorem 3 (Lemma 1 in [7])**.** *Every subgraph of a cograph is a cograph.*

Proof. To prove this, it is sufficient to show that a cograph remains a cograph even after the removal of any vertex. Let T be the cotree corresponding to the cograph G. Let G_1 be the induced subgraph of G without vertex x. We construct the cotree T_1 corresponding to the graph G_1. Suppose vertex y is the parent of vertex x in tree T. If vertex y had more than two children, then tree T_1 is obtained from tree T by removing vertex x and the edge $\{x, y\}$. Otherwise, let z be the second child of vertex y, and let w be the parent of vertex y. Tree T_1 is obtained from tree T by removing vertices x and y and all edges connected to them, and by adding the edge $\{w, z\}$. Thus, for any vertex x and any cotree T, exists a cotree T_1. Therefore, graph G_1 is a cograph, and consequently, any induced subgraph of a cograph is also a cograph. □

Knowing this fact, we can prove statement 2 of Theorem 1. A graph is a cograph if and only if any of its nontrivial subgraphs has a pair of twin vertices. Every nontrivial subgraph of a cograph is a cograph. Therefore, a cotree exists for any subgraph. In any nontrivial tree, there are two sibling vertices. If the vertices are siblings in the cotree, then they are twins in the cograph. If there are two twin vertices in every nontrivial subgraph of a graph, a cotree can be constructed for such a graph. Hence, this graph is a cograph. A more detailed proof of these facts is in the section 2.3.3.

From Remark 1, it follows that for each vertex n and for each vertex in T_n, the set of neighbors outside T_n is the same because least common ancestor is the same for any vertex x in T and a fixed vertex y not in T_n .

A set of vertices in the graph that possesses this property is called a module.Formally:

Definition 22 (Module Definition 1 in [8])**.** *For a graph $G = (V, E)$. A set of vertices $M \subseteq V$ is a* module *if and only if for any z and $t \in M$ it holds that $N(z)/M = N(t)/M$.*

Definition 23 (Strong module Definition 1 in [8])**.** *A module M is a strong* module *if and only if for any module M_1 it holds that either $M_1 \subseteq M$ or $M \subseteq M_1$ or $M_1 \cap M = \emptyset$.*

If the set of vertices M of the cograph is a strong module and is not the set of leaves of some subtree of the cotree, then exist a subtree of the cotree T_r and $T_r \cap M \neq T_r$, $T_r \cap M \neq M$, $T_r \cap M \neq \omega$. This, for example, holds if r is vertex from M with maximum depth such that it has a child $y \notin M$.

Consequently:

Remark 2 (Remark 6 in [8])**.** *For any strong module M, there is an internal node n of the cotree T such that M is exactly the set of leaves of T_n.*

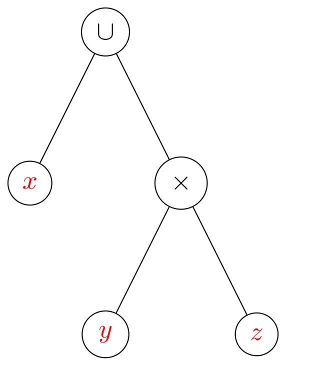
The first step of the cograph recognition algorithm is the construction of the factorizing permutation of the graph's vertices. Let us define it as follows :

Definition 24 (Definition 7 in [8])**.** *A factorizing permutation of a graph $G = (V, E)$ is a permutation σ of the vertex set V such that the vertices of any strong module of G appears consecutively in σ.*

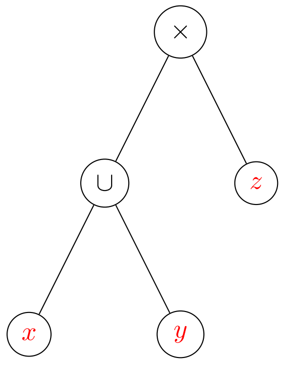
Factorizing permutations are convenient to use due to the existing bijection between them and the planar representations of cotrees.

Theorem 4 (Lemma 9 in [8])**.** *For each planar representation of a cotree, the leaves in the order of a left-to-right depth-first search forms a factorizing permutation. This is because the leaves of each subtree appear consecutively in the formed permutation. Consequently, the vertices of each strong module appear consecutively.*

The existence of a unique planar representation of a cotree for each factorizing permutation is proven by induction on the size of the permutation.



(a) Possible vertex order in Remark 3



(b) Alternative vertex order in Remark 3

Figure 1.2: Examples

Definition 25 (Definition 8 in [8])**.** *Let x, y, z be three distinct vertices in a graph $G = (V, E)$. Then x separates y and z if either $\{x, y\} \in E$ and $\{x, z\} \notin E$ or $\{x, y\} \notin E$ and $\{x, z\} \in E$.*

Now, using the properties of the cotree, we derive an important fact for the cograph recognition algorithm. To do this, let us introduce another definition :

Remark 3 (Corollary 11 in [8])**.** *Let x, y, z be distinct vertices appearing in that order in a factorizing permutation σ such that $\{x, y\} \notin E$ and $\{y, z\} \in E$. Then $\{x, z\} \in E$ if and only if $LCA(x, y)$ is a descendant of $LCA(y, z)$.*

Using Remark 1, we can uniquely determine the type of vertices for LCA of x and y and LCA of y and z. If the LCA of vertices x and y is a descendant of the LCA of vertices y and z, then the LCA of vertices x and y and the LCA of vertices x and z is the same vertex. The second case is considered similar. Both possible orders of the vertices are shown in the Section 1.3.

1.4 Partition

All definitions below are taken from [8]. Let us introduce the critical concept of this section:

Definition 26 (Partition)**.** *A* partition *$P = X_1, \dots, X_k$ of a set V is a set of disjoint subsets of V such that $X_1 \cup \dots \cup X_k = V$. We call $X_1, \dots, X_k$ the* parts *of P.*

Let us denote a part containing vertex x by H_x [8].

Definition 27 (Thinner relation)**.** *Let P and Q be two partitions of V. If for each part X of P there exists a part Y of Q such that $X \subseteq Y$, then we say that P is* thinner *than Q (or Q is coarser than P).*

Notation 1. *Let P be the partition $X_1, ..., X_k$ of the set V. Then we denote $X_i <_P X_j$ if and only if $i < j$. And let $u \in X_i$ and $v \in X_j$ be any two arbitrary vertices from different parts. Then $u <_P v$ if and only if $i < j$.*

Let us define a partial order on the partitions of set V.

Definition 28 (Compatibility)**.** *Let P and Q be two partitions of V, then P is* compatible *with Q, denoted by $P \leq Q$ if and only if both conditions hold:*

1. *P is thinner than Q,*
2. *for any x and $y \in V$ such that $x <_P y$ it holds that $x \leq_Q y$.*

Additionally $P < Q$ if both $P \leq Q$ and $P \neq Q$.

Definition 29 (Strict intersection, Definition 13 in [8])**.** *A set S* strictly intersects *another set S_1 if and only if $S \cap S_1 \neq \emptyset$ and the symmetric difference of sets S and $S_1 \neq \emptyset$.*

Definition 30 (Part refining)**.** *The* refinement *of the part H of the partition P using a pivot set S is replacing part H of the partition P with two parts $H \cup S$ and $H - S$.*

Definition 31 (Partition refining)**.** *The* refinement *of the partition P using a pivot set S is replacing each part X_i of P with two parts $X_i \cup S$ and $X_i - S$. The resulting partition after refinement is called Refine(P, S).*

In subsequent algorithms, the pivot set often denotes the set of neighbors of some vertex in the graph.

Notation 2 (Pivot)**.** *If $S = N(x)$ where $N(x)$ is the neighborhood of vertex x, we call x the* pivot*.*

Notation 3 (Splitter)**.** *If $N(x)$ strictly intersects P, we call vertex x a* splitter *of the partition P.*

Note that a vertex x is a splitter of P if and only if x separates at least two vertices within some part of P.

1.5 Treewidth and pathwidth

Treewidth and pathwidth are important structural parameters of graphs introduced in [13].

A cotree T can easily be transformed to an equivalent cotree T_1 such that every internal vertex in T has exactly two sons. It holds from the associativity of the operation : $G = G_1 \times G_2 \times \cdots \times G_k = G_1 \times (G_2 \times (\cdots \times (G_{k-1} \times G_k) \dots))$.

Definition 32 (Definition 2.2 in [3])**.** *A tree-decomposition of a graph $G = (V, E)$ is a pair $(\{X_i : i \in I\}, T = (I, F))$ with a family of subsets $\{X_i \subseteq V : i \in I\}$, and a tree T, such that*

1. *$\bigcup_{i \in I} X_i = V$,*

2. *There exists an appropriate $i \in I$ such that $v \in X_i$ and $w \in X_i$, for all $\{v, w\} \in E$,*

3. *For all $v \in V$, each set $\{i \in I \colon v \in X_i\}$ induced a subtree of T.*

Note that the third condition can be replaced by the following:

$3'$ For all $i, j, q \in I$ if q is on the path from i to j in T, then $X_i \cap X_j \subseteq X_q$.

Definition 33 (Treewidth, $tw(G)$)**.** *The treewidth of a tree-decomposition $(\{X_i \colon i \in I\}, T = (I, F))$ is defined as $\max_{i \in I} |X_i| - 1$. The treewidth of G, $tw(G)$, is the minimum treewidth over all possible tree-decompositions of G.*

Definition 34 (Definition 2.3 in [3])**.** *A path-decomposition of a graph $G = (V, E)$ is a pair $(\{X_i \colon i \in I\}, I)$, with a family of subsets $\{X_i \subseteq V \colon i \in I\}$, and there exists $r \in \mathbb{N}$ with $I = \{1, 2, \ldots, r\}$ such that*

1. *$\bigcup_{i \in I} X_i = V$,*

2. *There exists $i \in I$ such that $v \in X_i$ and $w \in X_i$, for all $\{v, w\} \in E$,*

3. *There exists an appropriate $b_v, e_v \in \{1, 2, \ldots, r\}$, such that for all $v \in V$ and i such that $v \in X_i$ it holds that $b_v \leq i \leq e_v$. And for all i such that $b_v \leq i \leq e_v$ it holds that $v \in X_i$.*

Definition 35 (Pathwidth, $pw(G)$)**.** *The pathwidth of $(\{X_i \colon i \in I\}, I)$ is defined as $\max_{i \in I} |X_i| - 1$. The pathwidth of graph G, $pw(G)$, is the minimum pathwidth over all possible path-decompositions of G.*

Chapter 2

Recognition algorithm

2.1 General idea

We are given a graph G. To determine if it is a cograph, we use the second fact from Theorem 1: a graph is a cograph if and only if its every nontrivial induced subgraph has a pair of twin vertices. In the second part of this section, we present a linear algorithm that checks whether every induced subgraph of the graph has a pair of twin vertices. For this, the algorithm requires not only the graph G but also a factorizing permutation of its vertices.

Thus, we need an algorithm that constructs a factorizing permutation for a cograph and returns a random permutation for a non-cograph.

2.2 Factorizing permutation

2.2.1 Idea

The algorithm maintains a partition P of the vertices of graph G. At each step, we refine the partition. To do this, we use two types of operations, which we call Rule 1 and Rule 2. These operations are chosen such that if G is a cograph and factorizing permutation of the vertices of G compatible with P before using the rule exist, then a factorizing permutation compatible with P after using the rule exist. After each partitioning step, the number of parts cannot decrease. Due to the properties of the rules, the number of parts increases. Consequently, at some point, the partition P consists of $|V|$ parts, each with one vertex. Later, we prove that for any graph, the number of rules applied cannot exceed $2 \cdot |E|$. If the partition P consists of $|V|$ parts, each with one vertex, then P is a permutation of the vertices of graph G. If G is a cograph, a factorizing permutation compatible with the partition P always exist, and therefore P itself is a factorizing permutation. If G is not a cograph, the algorithm returns a random permutation.

2.2.2 Rule 1

Definition 36 (Rule 1, Initialization rule in [8]). *Let ρ be a part in a partition, then pick an arbitrary vertex $x \in \rho$ hereafter called the* origin *of ρ, and refine ρ into $\{\bar{N}(x) \cap \rho, \{x\}, N(x) \cap \rho\}$.*

This rule has the property that if there is a compatible with partition P factorizing permutation before the operation, there will be also a compatible with partition P factorizing permutation after the operation.

Let us prove that we can use Rule 1 in the first step of the algorithm.

Theorem 5 (Lemma 14 in [8]). *Let x be an arbitrary vertex of a cograph, then there is a factorizing permutation compatible with partition $P = \{\bar{N}(x), \{x\}, N(x)\}$.*

Proof. For every cograph there is a cotree. Let T be any planar representation of the cotree of the cograph G. Let us prove that it is always possible to change the order of the subtrees of the vertices such that the order of the leaves in the left-to-right traversal of the new planar representation of the cotree is compatible with the partition $\{\bar{N}(x), \{x\}, N(x)\}$.

For each vertex u an ancestor of vertex x in the planar representation of the cotree T, we change the order of its children. Let $Y = \{y_0, y_1, \ldots, y_m\}$ be the set of all ancestors of vertex x in the cotree, such that vertex y_i is the parent of vertex y_{i+1} for each $i = 0, 1, \ldots, m-1$. If vertex y_i corresponds to a parallel composition, we make vertex y_{i+1} its right child. If vertex y_i corresponds to a series composition, we make vertex y_{i+1} its left child.

We call the such representation of the cotree T'. Then for any vertex z in the cograph, if vertices x and z are adjacent, Remark 1 proves that their least common ancestor in the cotree T': vertex u is a vertex corresponding to a series composition. Additionally, u belongs to the set Y. Then vertex x is in the subtree of a left child of u. Therefore,in a left-to-right depth-first search of the tree T' vertex z is visited before vertex vertex x. The case where vertex z is not adjacent to vertex x can be proved similarly. □

Now, let us prove the property of Rule 1 after the start of the algorithm.

Theorem 6 (Lemma 19 in [8]). *Let $G = (V, E)$ be a cograph and P be a partition with vertex x as origin, that can be refined into a factorizing permutation. Let $v \in$ part H_v be a pivot. Let us assume that any vertex z such that $LCA(x, z)$ is a descendant of $LCA(x, v)$ belongs to a singleton part. Let P_1 be the partition obtained from P by splitting H_v into $\{\bar{N}(v) \cap H_v, \{y\}, N(v) \cap H_v\}$. Then there is a factorizing permutation compatible with P_1.*

Proof. Let T be the planar representation of the cotree corresponding to a factorizing permutation compatible with partition P. Let the least common ancestor of vertices x and v in tree T be vertex y_i. Let $g_0, g_1, \ldots, g_k$ be the path from vertex y_i to vertex v. We make vertex g_0 the next child after vertex y_{i+1}. For vertices $g_0, g_1, \ldots, g_k$ the same algorithm as in Theorem 5 is used. That is, if vertex g_q corresponds to a parallel composition,let us make the subtree with vertex v the rightest subtree. If vertex g_q corresponds to a series composition, let us make the subtree with vertex v the leftest subtree.

Let us prove that the factorizing permutation obtained by a left-to-right depth-first search of the new planar representation of the cotree T' is compatible with the partition P_1. Let vertex $w \in H_v$ and its least common ancestor with vertex x be a vertex y_j. Then, according to the theorem conditions, either $j \leq i$ so the least common ancestor of vertices x and v is a descendant of the least common ancestor of vertices x and w.

If $j < i$, then LCA(x, w) =LCA$(v, w) = y_j$. According to Remark 1, x and w are adjacent if and only if v and w are adjacent. Therefore, a left-to-right depth-first search visits vertex w before vertex x if and only if it visits vertex w before vertex v.

If $j = i$, then LCA$(v, w) = g_q$ as already proven in Theorem 5, a left-to-right depth-first search visits vertex w before vertex v if and only if w and v are not adjacent.

Therefore, a factorizing permutation compatible with partition P_1 exists after using Rule 1. $\square$

2.2.3 Rule 2

Definition 37 (Rule 2, Refinement rule 2 in [8])**.** *If a vertex $x \notin H$ separates two vertices of a part H, then $N(x)$ refines H into $\{H \cap \bar{N}(x), H \cap N(x)\}$.*

First, let us prove a simpler version of the statement.

Theorem 7 (Lemma 16 in [8])**.** *Let x be a vertex $x \notin H$ for a part H. Let y and z be two vertices of cograph such that $y \in N(x)$ and $z \in \bar{N}(x)$ and let P be a partition thinner than $\{\bar{N}(x), \{x\}, N(x)\}$ such that there is a factorizing permutation compatible with P. Moreover,*

- *If y splits a part $H \subseteq \bar{N}(x)$ then there is a factorizing permutation compatible with P_1 that is obtained from P by refining H into $\{H \cap \bar{N}(y), H \cap N(y)\}$.*
- *If z splits a part $H \subseteq N(x)$ then there is a factorizing permutation compatible with P_1 that is obtained from P by refining H into $\{H \cap \bar{N}(z), H \cap N(z)\}$.*

Proof. Let us prove only the second statement, as the first one can be done similarly.

Let T be a planar representation of the cotree such that the corresponding factorizing permutation is compatible with the partition P. Vertex z separates part H. Therefore, there is a vertex $w \in H$ such that w and z are adjacent, and there is a vertex $u \in H$ such that z and u are not adjacent. Then, using Remark 1, we obtain the following facts:

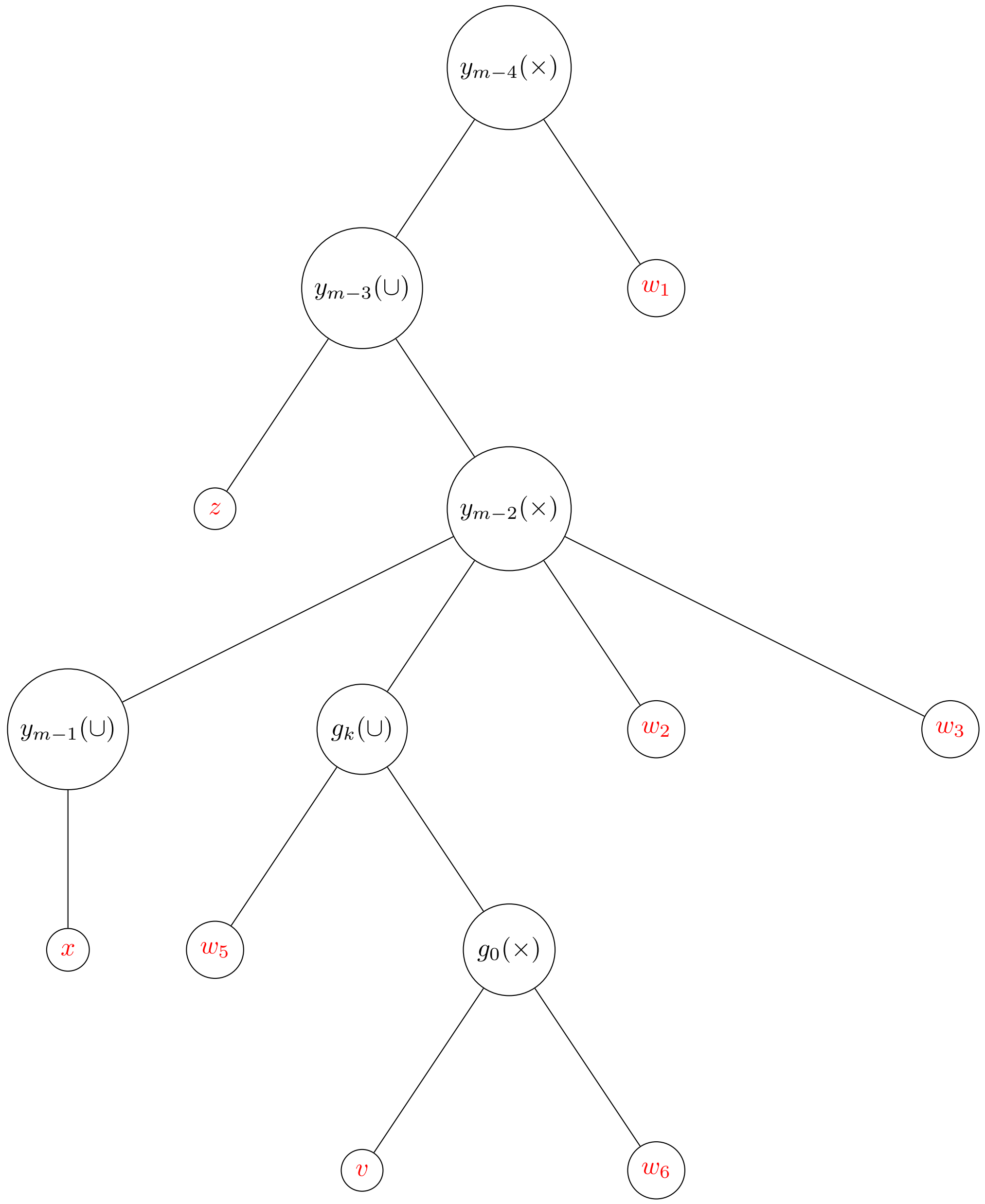


Figure 2.1: An example of planar representation of a cotree after applying the Rule 1 for vertex v

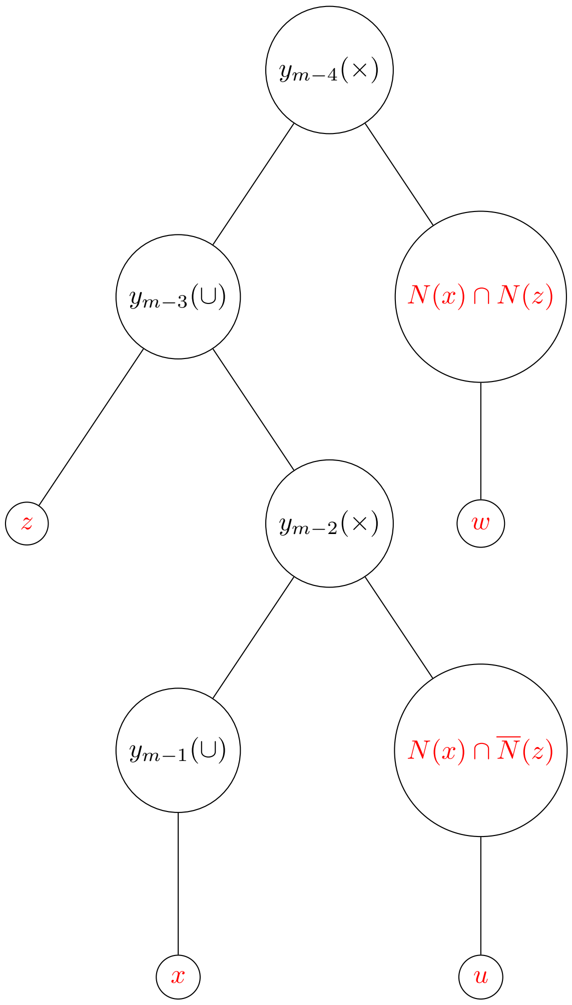


Figure 2.2: Vertex z splits the part formed by $N(x)$

1. LCA(x, u) is a descendant of LCA(x, z).
2. LCA(x, z) is a descendant of LCA(z, w).
3. LCA(w, z) = LCA(x, w).
4. LCA(x, w) is a series node.
5. LCA(x, u) is a series node.
6. LCA(x, z) is a parallel node.

Thus z and any vertex $v \in H$ are non-adjacent if and only if LCA$(v, z) =$ LCA(x, z). An example of a possible cotree is shown in Figure 2.2. From the theorem statement, we know that the factorizing permutation corresponding to the cotree is thinner than partition $\{\bar{N}(x), \{x\}, N(x)\}$. Therefore, the depth-first search visits vertex x before all vertices in set $N(x)$ and after all vertices in set $\bar{N}(x)$. Then, a left-to-right depth-first search visits all vertices in set $N(x) \cap \bar{N}(z)$ before any vertex in set $N(x) \cap N(z)$.

Suppose the opposite that the depth-first search visited $w \in N(x) \cap N(z)$ first, then vertex $u \in N(x) \cap \bar{N}(z)$. Then the order of visiting the vertices is as follows: z, x, w, u. Let the least common ancestor of vertices w and x be vertex y_{m-4}, and x be a descendant of vertex y_{m-3}, which is a child of vertex y_{m-4}. Since vertex x is visited before vertex w, it follows that vertex w is a descendant of a more right child of vertex y_{m-4}. In a left-to-right depth-first search, all vertices in the subtree $T_{y_{m-3}}$ must be visited before visiting vertex w. Vertex u belongs to the subtree $T_{y_{m-3}}$ so this leads to a contradiction.

Therefore, the factorizing permutation corresponding to cotree T is compatible with partition obtained from P by refining H into $\{H \cap \bar{N}(z), H \cap N(z)\}$. $\square$

The general idea is as follows: in the algorithm, we always use Rule 1 at least once before using Rule 2. Other relationship options between part H and vertex x are proven similarly. Therefore, if a factorizing permutation compatible with partition P existed before using Rule 2, then a factorizing permutation compatible with the partition P_1 obtained after using Rule 2 also exists.

2.2.4 Pseudocode

The algorithm is based on Algorithm 2 from [8]. Its idea for constructing a factorizing permutation is as follows: initially, the partition consists of one part. We use rules to increase the number of parts. And finally, when the number of parts becomes $|V|$, the algorithm stops.

The algorithm works in stages. At the beginning of each stage, a new vertex is chosen as the origin. Next, Rule 1 is applied to it. Then, as long as there is a part of the partition for which a pivot has not been chosen yet, we select any element of this part as the pivot and use it as a splitter for each other part of the partition.

```
recognition(graph G) {
     vector<parts> H;
     vector<parts> unusedParts;
     vector<nodes> N;
     H.push({V});
     Choose a random vertex of the graph as the origin
     while (|H| < |N|){
        if (H[origin].size() > 1) {
            Rule_1(origin);
            unusedParts.push(N(origin));
            unusedParts.push(origin.part / N(origin) / origin);
        }
        while (!unusedParts.empty()) {
            P = unusedParts[0];
            Choose a random vertex y of P as the pivot
            Rule_2(y);
            Mark P as used.
            for (p : newParts){
                if (The part p has no pivot){
                    unusedParts.push(p);
                }
            }
            unusedParts.erase(P);
        }
        Let l and r be the pivots of the nearest non-singleton parts to the vertex origin
        respectively, on the left and right sides.
        if ({l,r} belongs to the E) {
            origin = l;
        }
        else {
            origin = r;
        }
     }
}

Rule_1(node x) {
    H.erase(x.Part)
    H.push(origin.part - N(origin) - origin);
    H.push(origin);
    H.push(N(x));
}

Rule_2(node x) {
    for (p : H) {
        if (p != x.part) {
            if (N(x) strictly intersects p) {
                 part A = intersection of N(x) and p;
                H.push(A);
                H.push(p/A);
                H.erase(p)
            }
        }
    }
}
```

2.2.5 Correctness of the algorithm

Theorem 8. *The algorithm terminates on any graph.*

Proof. A part of the partition cannot be empty. Therefore, there cannot be more than $|V|$ parts in the partition. Each use of Rule 1 increases the number of parts by at least 1. Each use of Rule 2 increases the number of used parts in the partition. When all parts of the partition are used, Rule 1 is applied again. Therefore, the number of parts increases after $|V|+1$ operations. Consequently, the algorithm executes a finite number of steps. □

Invariant 1. *Let P be the current partition with origin $X \in V$. If G is a cograph, then there is a factorizing permutation compatible P.*

Proof. Let us prove the invariant by using induction on the number of used rules.

Base Case In the first step of the algorithm, we use Rule 1 to the partition consisting of one part containing all the vertices. The existence of a factorizing permutation after this refining is proven in Theorem 5.

Inductive Step Assume P_i is the partition obtained after used i rules with origin x_i. If G is a cograph, there is a factorizing permutation compatible with P_i. If in the next step we use Rule 1 for vertex v, then for each vertex w such that $\text{LCA}(x_1, w)$ is a descendant of $\text{LCA}(x_1, v)$ $H_w = \{w\}$. This is based on the case where we change the origin. We always choose the nearest vertex to the previous origin such that its part is not a singleton to be the next origin.

The existence of a factorizing permutation after this refining is proven in Theorem 6. If in the last step we used Rule 2, then the existence of a factorizing permutation after this refining is proven in Theorem 7 or can be proven similarly.

□

Theorem 9 (Theorem 20 in [8])**.** *Algorithm computes a factorizing permutation if the input graph is a cograph.*

Proof. When any part is a singleton part, from the invariant follows that a factorizing permutation exists. Therefore the partition P is a factorizing permutation. □

2.2.6 Implementation details of the algorithm

In this section, one of the variants of the linear implementation of the algorithm is described. We store the vertices of the graph and the parts of the partition as two doubly linked lists of structures.

Each part stores its first element, its last element, the number of elements, and its pivot (if it has one). Each element stores the part it belongs to and the next and previous element in its part.

The implementation of Rule 1 is simple. For each neighbor of vertex x, we check whether it is in the same part as vertex x.

```
Rule_1(node x){
   part newPart;
   for (y : N(x)) {
        if (y.part == x.part) {
            newPart.move(y);
        }
   }
   if (newPart.size() > 0) {
        H.push(newPart);
        x.part.next = newPart;
   }
   if (x.size() != 1) {
        part newPart1;
        newPart1.move(y);
        H.push(newPart1);
        x.part.next = newPart1;
   }
}
```

From all the neighbors of x that are in the same part as x, we create a new part. This new part is placed immediately after the part containing x. If there are any vertices in the part containing x, we split this part into two parts: the right part consists of vertex x, and the left part consists of all other vertices. In this way, we turn one part into 2 or 3 parts and use $O(|N(x))|)$ operations.

Now, let us show how to implement Rule 2 in linear time. First, for each part that contains a vertex that is a neighbor of vertex x and is not equal to $x.part$, we count how many vertices in this part are neighbors of x. Next, for each part that strictly intersects $N(x)$, we create a new part to the right of it. In the next loop, we move the elements into the new part. In the final loop, we clear all the values.

```

Rule_2(node x) {
   for (y : N(x)) {
        y.part.amount++
    }
    for (y : N(x)) {
        if (0 < y.part.amount < y.part.size() && !y.part.part_is_created){
            part newPart_y;
            H.push(newPart_y);
            y.part.next = newPart_y;
            y.part.part_is_created = true;
        }
    }
    for (y : N(x)) {
        if(y.part.part_is_created){
            y.part.next.move(y);
        }
    }
    for (y : N(x)) {
        y.part.previous.amount = 0;
        y.part.part_is_created = false;
    }
}
```

The total time of Rule 2 for vertex x is $O(|N(x)|)$.

2.2.7 Running time of the algorithm

Theorem 10. *Rule 1 cannot be used to vertex x more than once.*

Proof. After using the rule to vertex x, the part H_x becomes equal to $\{x\}$. Rule 1 is not applied to elements from singleton parts. □

Theorem 11. *Rule 2 cannot be applied to vertex x as a pivot more than once.*

Proof. After applying Rule 2, part H_x becomes used and remains used until the end of the algorithm. Rule 2 is not applied to the pivot if its part is already used. □

Theorem 12 (Theorem 24 in [8])**.** *The algorithm terminates in $O(n+m)$ time for any graph.*

Proof. As proven in the Section 2.2.6, the time complexity of applying the rule to element x is $O(|N(x)|)$. Each rule was applied to each vertex at most once. Therefore, the total time of the algorithm is at most $\sum\limits_{x \in V} 2 \cdot O(N(x)) = O(n+m)$. □

2.3 Permutation test

2.3.1 Idea

We use the definition of a cograph from Theorem 1. A graph is called a cograph if and only if for each of its non-trivial subgraphs, there are two twin vertices.

In this case, it is easy to prove that a graph is a cograph if and only if there is a twin-elimination ordering of its vertexs.

Definition 38 (Twin-elimination ordering [8])**.** *A vertex permutation σ is a twin-elimination ordering if and only if $\sigma = x_1, \ldots, x_n$ such that for any vertex x_i there are $j, i < j$ such that x_i and x_j are either true twins or false twins in graph $G_i = G[\{x_i, x_{i+1}, \ldots, x_{n-1}\}]$.*

Proof. If a twin-elimination ordering exists, then a cotree can be constructed for the graph. Thus, the graph is a cograph. More details on this can be found in Section 2.3.3.

On the other hand if the graph is a cograph, then such two twin vertices always exist. The proof of this follows from Theorem 13. Therefore, for any G_i, there is a respective x_i. □

In the permutation test algorithm, we attempt to construct a twin-elimination ordering using its vertices of permutation. If it is successful, it indicates that the permutation corresponds that graph is a cograph. Otherwise, the original graph is not a cograph.

We construct the twin-elimination ordering from left to right. Each time when we find a suitable vertex in the obtained permutation, we remove it. Then, we have successfully constructed the twin-elimination ordering if and only if there is exactly one number left in the permutation at the end of the algorithm.

2.3.2 Pseudocode

The algorithm is based on Algorithm 5 from [8].

Initially, the main vertex is the first vertex in the permutation. At each step, we check if the main vertex is a twin with any of its neighbors. If it is, we remove the left vertex of this pair. If we have not removed the vertex or if the main vertex is removed, then the next main vertex is the right neighbor of the previous main vertex.

The algorithm stops when there are no more vertices to be designated as the main vertex. In other words, it happens when the rightest vertex of the permutation has been the main vertex.

For an easier implementation, we add non-existent vertices to the beginning and to the end of the permutation. No vertex is a twin for these two added vertices. Then, the graph is a cograph if and only if, after the algorithm stops, there are exactly 3 vertices in the permutation.

```

Permutation test(x permutation of the vertex) {
    x[0] and x[n+1] added vertices
    mainVertex = x[1];
    while (mainVertex != x[n+1]) {
        if (mainVertex and mainVertex.previous are twins) {
            erase(mainVertex.previous);
        }
        else{
            if (mainVertex and mainVertex.next are twins) {
                mainVertex = mainVertex.next;
                erase(mainVertex.previous);
            }
            else{
                mainVertex = mainVertex.next;
            }
        }
    }
}
```

2.3.3 Constructing the cotree

Theorem 13 ([8]). *If two vertices are true (false) twins in a cograph, it is if and only if they are siblings in the cotree and their parent is a series (respective parallel) node.*

Proof. Assume the opposite: let vertices x and y be true twins, but their lowest common ancestor is a parallel node. Then, by Remark 1, they are not neighbors; thus, sets $N(x) \cup \{x\}$ and $N(y) \cup \{y\}$ are not equal. Contradiction. Assume vertices x and y are true twins, and their LCA is a series node, but the vertices are not siblings. Then, there is a vertex z in the cograph such that the LCA (x, z) is a descendant of the LCA(x, y), and LCA(x, z) is a parallel node. Then, vertices x and z are not neighbors, and vertices y and z are neighbors. Contradiction. □

Let us demonstrate an algorithm to construct a cotree from a cograph's twin-elimination ordering in linear time.

Let the twin-elimination ordering be denoted as $t = (t_1, \ldots, t_n)$. At each step, the algorithm adds the rightest unprocessed vertex of the ordering to the cograph. The cotree obtained after adding vertex t_i is called T_i.

Initially, the cotree is only the vertex $t_{|V|-1}$. At each subsequent step, we add vertex t_i. We know it is a twin of some other leaf t_j in the constructed cotree. We create a new internal vertex w that is parallel if t_i and t_j are false twins, or series if they are true twins. To form T_i, we replace vertex t_j with w in T_{i+1} and make t_i and t_j children of w. This replacement ensures that the LCA for each pair of leaves in T_{i+1} remains unchanged, maintaining the correctness of the cotree.

Note that the resulting cotree is binary but may not always have adjacent internal nodes of different types. If a canonical cotree is needed, a depth-first search can merge adjacent nodes of the same type.

```
Build cotree(t twin-elimination ordering) {
   for (i = |V| - 2; i >= 0; i--) {
        j = index of vertex i twin;
        node p = t[j].parent;
        node newNode;
        if (t[i] and t[j] are true twins) {
            newNode.type = series;
        }
        else {
            newNode.type = parallel;
        }
        p.addSon(newNode);
        p.eraseSon(t[j]);
        newNode.addSon(t[j]);
        newNode.addSon(t[i]);
    }
}
```

In order the algorithm to run in linear time, it is necessary for each vertex to store with which vertex it was removed in the twin-elimination ordering construction algorithm. Then, the time necessary to find the twin t_j for vertex t_i is $O(1)$.

2.3.4 Correctness of the algorithm

It is clear that if the algorithm constructs a twin-elimination ordering, then the graph is a cograph. Let us prove the correctness of the algorithm in the other direction.

Invariant 2 (Invariant 1 in [8]). *At each step of the twin-elimination ordering construction algorithm, the permutation x is a factorizing permutation of the vertices of the graph G_i.*

Proof. A factorizing permutation corresponds to some planar representation of the cotree. If we remove one leaf from this planar representation of the cotree, the leaves of each subtree appear consecutively in the permutation. Consequently, the vertices of each strong module of the new graph also appear consecutively in the permutation. Thus, the new permutation remains factorizing. □

Let m_i be the index of the vertex that was the main one at the i-th iteration of the while loop in the algorithm. Let G_i be the induced subgraph of the permutation t at the i-th iteration of the while loop in the algorithm.

Invariant 3 (Invariant 2 in [8]). *For any $k \geq 1$, the subsequence $t_k([m_0, \ldots, m_k])$ does not contain any twins vertices in G_k*

Proof. We distinguish three cases:

1. m_k and $m_k.previous$ are twins in G_k. But then $m_k.previous$ is deleted from G_k and $m_{k+1} = m_k$, $t_{k+1}([m_0, \ldots, m_{k+1}])$ is included in $t_k([z_0, \ldots, z_k])$, and therefore invariant is trivially true.

2. m_k and $m_k.next$ are twins in G_k. But then m_k is deleted from G_k and $m_{k+1} = m_k.next$, $t_{k+1}([m_0, \ldots, m_{k+1}]) = t_k([m_0, \ldots, m_k])$, and therefore invariant 2 is trivially true

3. In the last case, we move right on the circular list, and $m_{k+1} = m_k.next$, $t_{k+1}([m_0, \ldots, m_{k+1}]) = t_k([m_0, \ldots, m_k])$.

□

If the original graph is a cograph and after the algorithm finishes there is a permutation g and g has more than one vertex, then the following 3 statements are true:

- each vertex in g was the main vertex at the same point,
- g is a factorizing permutation,
- there are two vertices belonging to g such that they are twins in the graph $G[g]$.

This contradicts Invariant 2. Therefore, if the graph is a cograph, the size of the permutation after the testing is always equal to 1. Thus, we have proved the following theorem:

Theorem 14. *The size of the permutation after testing is equal to 1 if and only if it is a factorizing permutation of the cograph's vertices.*

2.3.5 Running time of the algorithm

It is clear that if the algorithm constructs a twin-elimination ordering, then the graph is a cograph. Let us prove the correctness of the algorithm in the other direction.

Theorem 15. *The permutation algorithm terminates in $O(n+m)$ time for any graph.*

Proof. Each vertex is the main vertex in no more than 2 twin checks. The twin check for vertices x and y takes a $\min\{|N(x)|, |N(y)|\}$ time. To get this, it is necessary to store the vertexs edges in sorted order. Then, the total time of the testing algorithm is no more than $\sum\limits_{u \in V} 2 \cdot O(|N(x)|) = O(n+m)$. □

Theorem 16. *The recognition algorithm terminates in $O(n+m)$ time for any graph.*

Proof. By Theorem 12 and Theorem 15 both parts of the algorithm run in linear time; therefore, the entire algorithm runs in linear time. □

Chapter 3

Algorithmic properties of cographs

3.1 Idea

The algorithms are inspired by [7]. Having a cotree corresponding to the given cograph, we can compute various graph parameters using dynamic programming, e. g; treewidth, maximum clique or similar parameters. Typically, we calculate them in a bottom-up manner using e.g depth-first search on the cotree in binary form. Initially, the parameter value is computed for all descendants of a vertex, and then for the vertex itself. For recalculation, we use the fact that knowing the type of vertex (parallel or series), we know which edges exist in the cograph between the leaves of different subtrees of the cotree.

3.2 Maximum clique

The idea of calculating the maximum clique is as follows:

- If the cograph consists of only one vertex, then the size of the maximum clique is 1.
- If If the root of the cotree is a parallel vertex, then the graph $G = G_1 \cup G_2 \cup \cdots \cup G_k$. Hence, the maximum clique of the cograph is the largest maximum clique among the graphs $G_1, \ldots, G_k$. Therefore, the maximum clique of the graph equals the maximum clique among all the children of the root.
- If the root of the cotree is a series vertex, then the graph $G = G_1 \times G_2 \times \cdots \times G_k$. Consequently, all the leaves from different subtrees of the root's children are adjacent in the cograph. Then, the set of vertices $v_1, \ldots, v_f$ forms a clique if and only if for every pair of vertices v_i and v_j, either v_i and v_j are descendants of different children of the root, or v_i and v_j are

neighbors. Therefore, the maximum clique of the graph equals the sum of the maximum cliques of its children.

Such a formula can be computed in linear time using a depth-first search.

The first depth-first search calculates the size of the maximum clique for each cograph corresponding to a subtree of the cotree. The second depth-first search finds any set of vertices in the graph that form a maximum clique.

```

Maximum clique(cotree T) {
   Maximum subtree clique size(tree root);
   Add maximum subtree clique(tree root);
}
Maximum subtree clique size(node x) {
    if (x.type == leaf) {
        return 1;
    }
    if (x.type == parallel) {
        return max(Maximum subtree clique(x.leftSon), Maximum subtree clique(x.rightSon));
    }
    if (x.type == series) {
         return Maximum subtree clique size (x.leftSon) + Maximum subtree clique size(x.rightSon);
    }
    return x.subtree_clique_size;
}

Add maximum subtree clique (node x) {
    if (x.type == leaf) {
        Clique.add(x);
    }
    if (x.type == parallel) {
        if (x.leftSon.subtree_clique_size >= x.rightSon.subtree_clique_size){
            Add maximum subtree clique(x.leftSon);
        }
        else {
            Add maximum subtree clique(x.rightSon);
        }
    }
    if(x.type == series) {
        Add maximum subtree clique(x.leftSon);
        Add maximum subtree clique(x.rightSon);
    }
}

```

3.3 Graph coloring

The coloring algorithm for a cograph use a similar scheme to the previous algorithms. We color the subtrees of the cograph using the properties of parallel and series vertices. If the graph is a parallel composition of cographs $G_1, \ldots, G_k$ then the cographs $G_1, \ldots, G_k$ can be colored independently. If the graph is a series composition of cographs $G_1, \ldots, G_k$ then the leaves that are descendants of different children of the root cannot have the same color. Therefore, the chromatic number of the cograph is equal to the sum of the chromatic numbers of the cographs $G_1, \ldots, G_k$. The first cograph can be colored with colors $1 \ldots \chi(G_1)$, the second cograph with colors $\chi(G_1)+1, \ldots, \chi(G_1)+\chi(G_2)$ and so on.

```

void Graph coloring (){
    Subtree coloring(tree root, 0);
}

int Subtree coloring(node x ,number of occupied colors k){
    if (x.type == leaf) {
        x.color = k + 1;
    }
    int l = Subtree coloring(x.leftSon, k);
    if (x.type == parallel) {
       int r = Subtree coloring(x.rightSon, k);
       return max(l, r);
    }
    if (x.type == series) {
       int r = Subtree coloring(x.rightSon, l);
       return r;
    }
}
```

3.4 Independent set

The idea of finding an independent set is similar to the one presented above, since the maximum independent set is the maximum clique of the complement graph. The algorithm for finding the maximum clique was discussed in Section 3.3. It is sufficient to build the cotree of the complement cograph in linear time.

Theorem 17. *If T is the cotree of graph G and T' is the cotree of graph G', where T' is obtained from T by changing the types of vertices (parallel nodes to series nodes and vice versa), then $G = \bar{G}'$.*

Proof. Vertices x and y are neighbors in the cograph G if and only if their lowest common ancestor in the cotree T, denoted by w, is a series vertex. From the definition of cotree T', the vertex w in cotree T' is a parallel vertex. Therefore, vertices x and y are neighbors in cograph G if and only if they are not neighbors in graph G'. $\square$

3.5 Treewidth and pathwidth

Theorems and algorithms are taken from [3].

The basis of the algorithm is the theorem on the relationship between pathwidth and treewidth with parallel and series composition.

Theorem 18 (Lemma 3.4 in [3])**.** *Let $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ be graphs. Then the following conditions hold:*

- $tw(G_1 \cup G_2) = \max\{tw(G_1), tw(G_2)\}$.
- $pw(G_1 \cup G_2) = \max\{pw(G_1), pw(G_2)\}$.
- $tw(G_1 \times G_2) = \min\{tw(G_1) + |V_2|, tw(G_2) + |V_1|\}$.
- $pw(G_1 \times G_2) = \min\{pw(G_1) + |V_2|, pw(G_2) + |V_1|\}$.

The complete proof of Theorem 18 can be found in [3]. The idea of the proof is as follows. If a cograph G is a parallel composition of two cographs G_1, G_2, then the tree decomposition (I, T) of the graph G is simply the union of the tree decompositions of graphs G_1 and G_2. $I = (I_1 \cup I_2), T = (T_1 \cup T_2)$ hence $tw(G) = \max\{(tw(G_1), tw(G_2)\}$.

If a cograph is a series composition of cographs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and (I_1, T_1) is tree decomposition of G_1 and (I_2, T_2) is tree decomposition of G_2, then it is sufficient to prove the inequality in two directions:

1. $tw(G) \leq \min\{tw(G_1) + |V_2|, tw(G_2) + |V_1|\}$.
2. $tw(G) \geq \min\{tw(G_1) + |V_2|, tw(G_2) + |V_1|\}$.

Let $tw(G_1) + |V_2| \leq tw(G_2) + |V_1|$. Then the pair $(X_i \cup |V_2| \colon i \in I_1, T_1)$ is a tree decomposition of the cograph G. It holds that $\max\limits_{i \in I_1} |X_i \cup V_2| \leq tw(G_1) + |V_2|$ hence $tw(G) \leq \min\{tw(G_1) + |V_2|, tw(G_2) + |V_1|\}$.

The proof of the inequality in the opposite direction can be found in [3].

Theorem 19 (Theorem 3.2 [3])**.** *For every cograph G, it holds that $tw(G) = pw(G)$.*

Proof. We can prove the equality of pathwidth and treewidth for every cograph using induction on the number of vertices.

Base case For a graph consisting of a single vertex, the pathwidth and treewidth are the same and equal to 0.

Induction step Assume that for all cographs with less than $n+1$ vertices, the pathwidth and treewidth are the same. Let $G = (V, E)$ be any cograph with $n+1$ vertices. Let T be the cotree corresponding to G. If the root of T is a parallel vertex, then there exist two cographs G_1 and G_2 such that $G = G_1 \cup G_2$. From Theorem 18, it follows that $tw(G_1 \cup G_2) = \max\{tw(G_1), tw(G_2)\}$ and $pw(G_1 \cup G_2) = \max\{pw(G_1), pw(G_2)\}$. However, the number of vertices in G_1 and G_2 is less than $n+1$. Therefore, $tw(G_1) = pw(G_1)$ and $tw(G_2) = pw(G_2)$. Then $tw(G) = pw(G)$. The case where the root of the cotree is a series vertex is proved similarly.

□

Next, using dynamic programming on the cotree, we find the number of vertices in the subtree of each vertex. In the following depth-first search, we find the pathwidth and treewidth for each subtree.

```

int Graph pathwidth(cotree T) {
   Subtree sizes(tree root);
   return Subtree pathwidth(tree root);
}

void Subtree sizes(node x) {
    if (x.type == leaf) {
        x.subtree_size = 1;
    }
    else {
        Subtree sizes(x.leftSon);
        Subtree sizes(x.rightSon);
        x.subtree_size = x.leftSon.subtree_size + x.rightSon.subtree_size
    }
}

int Subtree pathwidth(node x) {
    if (x.type == leaf){
        return 1;
    }
    if (x.type == parallel) {
        return max(Subtree pathwidth(x.leftSon), Subtree pathwidth(x.rightSon));
    }
    if (x.type == series) {
         return min(Subtree pathwidth(x.leftSon) + x.rightSon.subtree_size,
         Subtree pathwidth(x.rightSon) + x.leftSon.subtree_size);
    }
    return x.subtree_clique_size;
}

```

Chapter 4

Implementation

4.1 Introduction

The algorithms described above were implemented in C++. For graph operations, the NetworKit and Koala libraries were used. The code is open source and available in the Koala-NetworKit GitHub repository at `https://github.com/krzysztof-turowski/koala-networkit/pull/23`.

4.2 Implementation details

The repository contains separate classes for each type of problem, such as cograph recognition, treewidth calculation, maximum clique calculation, and so on. Below is a brief description of the files with their purposes.

- `koala-networkit/include/recognition/CographRecognition.hpp`

  The main header file defining the cograph recognition class.

- `koala-networkit/include/recognition/cograph/Part.hpp`

  Header file for the class implementing the Part and Element classes used by the recognition algorithm.

- `koala-networkit/include/recognition/cograph/FactorizingPermutation.hpp`

  Header file for the class implementing the creation and manipulation of a factorizing permutation. For example, adding or removing parts or elements, moving elements to different parts, and so on. It is used in the factorizing permutation construction algorithm.

- `koala-networkit/include/recognition/cograph/Twins.hpp`

  Header file for the class implementing operations with twin vertices in a cograph. For example, checking if vertices are twins. It is used in the single permutation testing algorithm.

- `koala-networkit/include/structures/Cotree.hpp`

  Header file for the class implementing operations on the cotree and its vertices.

- `koala-networkit/include/pathwidth/CographPathwidth.hpp`

  Header file for the class implementing the calculation of graph pathwidth.

- `koala-networkit/include/coloring/CographVertexColoring.hpp`

  Header file for the class that finds the graph coloring with the minimum number of colors.

- `koala-networkit/include/independent_set/CographIndependentSet.hpp`

  Header file for the class that finds the independent set of a graph.

- `koala-networkit/include/max_clique/CographMaxClique.hpp`

  Header file for the class that finds the maximum clique of a graph.

- `koala-networkit/benchmark/`

  Folder containing benchmarking programs to evaluate the performance of the algorithms on various graph instances.

- `koala-networkit/test/`

  Folder containing unit tests for checking correctness of the implemented algorithms.

4.3 Testing

The correctness of each algorithm was tested in two ways:

1. Unit tests on all small-sized graphs, i.e. graphs with less than 15 vertices.
2. On graphs from the benchmark containing random cographs and non-cographs with less than 10^6 vertices.

The following iterative algorithm generates random cographs. First, create n cographs, each consisting of a single vertex. At each step, select and merge two random cographs into one. Then, with a given probability p, replace the cograph with its complement. The algorithm stops when all cographs merge into one. This algorithm can produce any cograph with n vertices. At step i, merge cographs G_i^1 and G_i^2. Let the set $B_{x,y} = \{i \colon x, y \in (G_i^1 \cup G_i^2)\}$ include all the steps in which vertices x and y participated in the merging. Vertices x and y become neighbors if and only if the number of times the complement is used during these operations is odd. The probability of this is $1 - \sum_{i=0}^{|B|/2} \binom{|B|}{2 \cdot i}$. This fact allows generating graphs with a specific number of edges. For example, if $p = \frac{1}{2}$, then the probability of an edge existing between any pair of vertices is $\frac{1}{2}$. This follows from the well-known fact that forall n it is true that $\sum_{i=0}^{i=n} (-1)^i \cdot \binom{n}{i} = 0$.

Id	Algorithm
$A1$	Cograph recognition
$A2$	Independent set
$A3$	Max clique
$A4$	Pathwidth
$A5$	Vertex coloring

Table 4.1: The algorithms numbers with their identifiers

4.4 Running time

The explanations of the algorithm names are provided in Table 4.1. The algorithms were tested on several subgroups of tests. Each test consists of 5 random graphs generated with the same number of edges and vertices. The average runtime, in milliseconds, are presented in Table 4.2 and Table 4.3.

n	m	$A1$
10^3	$5 \cdot 10^3$	2
$7 * 10^4$	10^5	7100
10^5	$2 \cdot 10^6$	49000
10^6	$4 \cdot 10^7$	$17 \cdot 10^5$

Table 4.2: The running time of the cograph recognition algorithm

n	$A2$	$A3$	$A4$	$A5$
10^5	3	15	5	12
$5 \cdot 10^5$	16	108	26	90
10^6	33	238	53	189
$4 \cdot 10^6$	144	1369	212	910
$8 \cdot 10^6$	292	3157	425	2079
$2 \cdot 10^7$	732	9584	1105	6039
$5 \cdot 10^7$	1993	28287	2741	18316

Table 4.3: The running time of different algorithms on cographs

4.5 Conclusions

It is clear that the number of edges in the graph does not affect the performance of algorithms $A2, A3, A4, A5$. This is because these algorithms do not operate on the original cograph, but only on the constructed cotree. The algorithm from Section 2.3.3 always constructs a cotree with $O(n)$ edges. The Figure 4.1 shows that these algorithms run in linear time and, within the tested range.

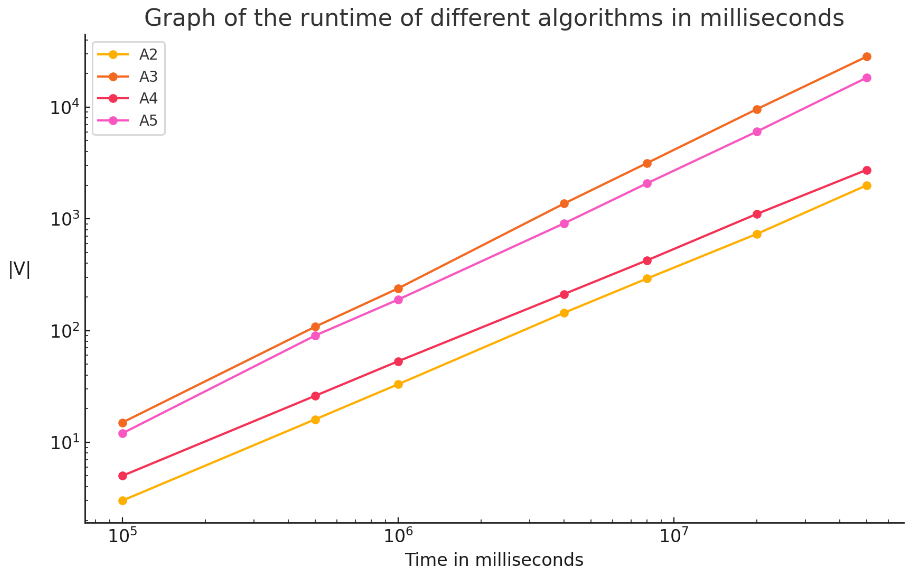


Figure 4.1: Graph of the runtime of different algorithms in milliseconds.

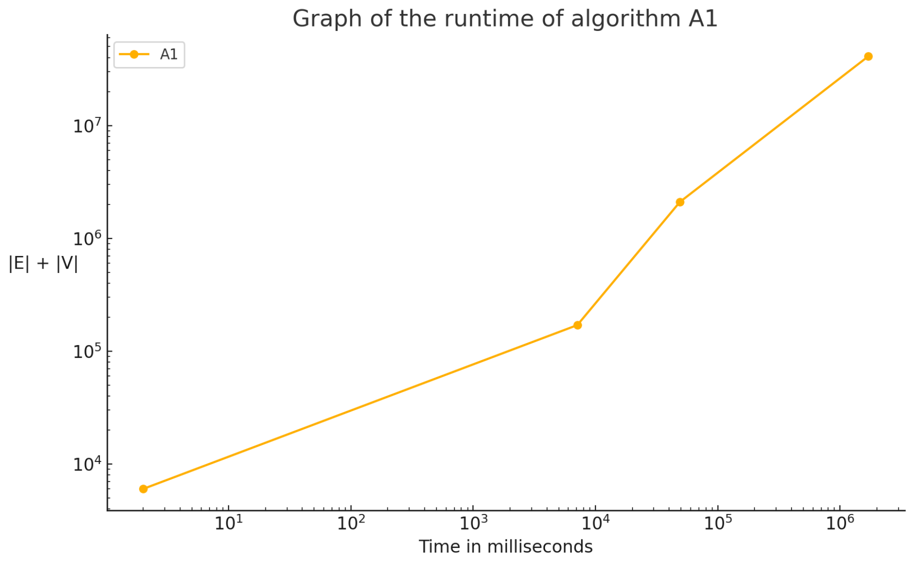


Figure 4.2: Graph of the runtime of $A1$ algorithm in milliseconds.

The running time of the algorithm A1 depends on both the number of edges and the number of vertices. From the Figure 4.2 and Table 4.2 values, it is clear that the algorithm is linear within the tested range.

Bibliography

[1] Hans Bodlaender. “Achromatic Number Is NP-Complete For Cographs And Interval Graphs”. In: *Information Processing Letters* 31 (1989), pp. 135–138. URL: `https://api.semanticscholar.org/CorpusID:204999731`.

[2] Hans Bodlaender and Klaus Jansen. “On The Complexity Of The Maximum Cut Problem”. In: *Nordic Journal of Computing* 775 (1994), pp. 769–780. URL: `https://api.semanticscholar.org/CorpusID:2648793`.

[3] Hans Bodlaender and Rolf Möhring. “The Pathwidth and Treewidth of Cographs”. In: *SIAM Journal on Discrete Mathematics* 6.2 (1993), pp. 181–188. URL: `https://doi.org/10.1137/0406014`.

[4] Béla Bollobás. “Modern Graph Theory”. In: *Graduate Texts in Mathematics.* Springer, 2002. URL: `https://api.semanticscholar.org/CorpusID:120529008`.

[5] Leizhen Cai. “Fixed-Parameter Tractability Of Graph Modification Problems For Hereditary Properties”. In: *Information Processing Letters* 58 (1996), pp. 171–176. URL: `https://api.semanticscholar.org/CorpusID:263880357`.

[6] Derek Corneil, Helmut Lerchs, and Burlingham Stewart. “Complement Reducible Graphs”. In: *Discrete Applied Mathematics* 3 (1981), pp. 163–174.

[7] Derek Corneil, Helmut Lerchs, and Burlingham Stewart. “Complement reducible graphs”. In: *SIAM Journal on Discrete Mathematics* 6.2 (1981), pp. 163–174. URL: `https://www.sciencedirect.com/science/article/pii/0166218X81900135`.

[8] Michel Habib and Christophe Paul. “A simple linear time algorithm for cograph recognition”. In: *Discrete Applied Mathematics* 145.2 (2005), pp. 183–197. URL: `https://www.sciencedirect.com/science/article/pii/S0166218X04002446`.

[9] Nastos James and Gao Yong. “A Novel Branching Strategy For Parameterized Graph Modification Problems”. In: *Lecture Notes in Computer Science* 6509 (2010), pp. 332–346.

[10] Klaus Jansen and Petra Scheffler. “Generalized Coloring For Tree-like Graphs”. In: 75.2 (1992), pp. 135–155. URL: https://api.semanticscholar.org/CorpusID:29823658.

[11] Yunlong Liu et al. “Complexity And Parameterized Algorithms For Cograph Editing”. In: *Theoretical Computer Science* 461 (2012), pp. 45–54. URL: https://api.semanticscholar.org/CorpusID:36105533.

[12] Damaschke Peter. “Induced Subraph Isomorphism For Cographs Is NP-complete”. In: *Lecture Notes in Computer Science* 484 (1991), pp. 72–78.

[13] Neil Robertson and Paul Seymour. “Graph minors. II. Algorithmic aspects of tree-width”. In: *Journal of Algorithms* 7.3 (1986), pp. 309–322. URL: https://www.sciencedirect.com/science/article/pii/0196677486900234.

[14] Dieter Seinsche. “On A Property Of The Class Of N-colorable Graphs”. In: *Journal of Combinatorial Theory* 16 (1974), pp. 191–193. URL: https://api.semanticscholar.org/CorpusID:122690712.