Jagiellonian University
Department of Theoretical Computer Science

Milana Kananovich

Implementation and comparison of cograph recognition algorithms

Bachelor Thesis

Supervisor: dr hab. Krzysztof Turowski

June 2024

Contents

Chapter 1

Introduction

1.1 Cograph information

The discoverers of cographs include Jung (1978) "On a Class of Posets and the Corresponding Comparability Graphs" [17], who called them *D^*-graphs*; Lerchs (1971) "On cliques and kernel" [8], who reffered to them as *hereditary Dacey graphs*; Seinsche (1974) "On a property of the class of n-colorable graphs" [23], who named them 2-*parity graphs*; and Sumner (1974) "Dacey graphs" [25]. Cographs recognition can be done in linear time. The different recognition algorithms are listed in tables (see Table 1.1 and Table 1.2). This work will discuss some of these recognition algorithms, compare them, and describe their respective advantages and disadvantages.

The subclasses of cographs include complete graphs, cluster graphs, threshold graphs, and Turán graphs. An example of the complexity of a problem for cographs is that the weighted maximum cut is NP-complete for threshold graphs, and consequently NP-complete for cographs.

Regarding the superclasses of cographs, they encompass distance-hereditary graphs, permutation graphs, comparability graphs, perfectly orderable graphs, even hole-free graphs, Meyniel graphs, and strongly perfect graphs. An example is that the domination problem is linear for distance-hereditary graphs, and therefore linear for cographs.

Various results are known for cographs:

- the ability to find in linear time the maximum clique, maximum independent set, vertex coloring number, maximum clique cover, Hamiltonicity [8], pathwidth and treewidth, which are equal for cographs [4]. Additionally, there is $O(n^2)$ algorithm for simple max cut [3],

- NP-complete problems include determining whether a given cograph G is a subgraph of a given cograph H [11], computing achromatic number [2] and list coloring [16],

- fixed parameter tractable (FPT) problems are determining whether a

Recognition algorithm	Authors	Year	Ref.	Complexity
Complement reducible graphs	Corneil, Lerchs, Stewart	1981	[8]	$O(n^2)$
A linear recognition algorithm for cographs	Corneil, Perl, Stewart	1985	[9]	$O(n+m)$
A simple Linear Time LexBFS Cograph Recognition Algorithm	Bretscher, Corneil, Habib, Paul	2003	[6]	$O(n+m)$
A fully dynamic algorithm for modular decomposition and recognition of cographs	Shamir, Sharan	2004	[24]	$O(n+m)$
A simple linear time algorithm for cograph recognition(partition refinement)	Habib, Paul	2005	[13]	$O(n+m)$
Linear-time certifying recognition algorithms and forbidden induced subgraphs	Heggernes, Kratsch	2007	[14]	$O(n+m)$
Split decomposition and graph-labelled trees: characterizations and fully dynamic algorithms for totally decomposable graphs	Gioan, Paul	2008	[12]	$O(n+m)$

Table 1.1: Consecutive recognition algorithms for cographs. The implemented algorithms are marked in bold.

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 [7], k-edge-deletion to cographs is solvable in $O(2.562^k \cdot (m+n))$ time and k-vertex-deletion algorithm is solvable in $O(3.303^k \cdot (m+n))$ time [15], k-edge-edited to cographs is solvable in $O(4.612^k \cdot (m+n))$ time [21].

1.2 Basic definitions

The following standard graph theory definitions are sourced from Bollobás "Modern Graph Theory" [5].

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.*

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

Recognition algorithm	Authors	Year	Ref.	Complexity
An NC recognition algorithm for cographs	Lin, Olariu	1991	[20]	$O(\log n)$ time $O(\frac{n^2+mn}{\log n})$ processors
Efficient parallel recognition algorithms of cographs and distance hereditary graphs	Dahlhaus	1995	[10]	$O(\log^2 n)$ time $O(n+m)$ processors
Efficient parallel recognition of cographs	Nikolopoulos, Palios	2005	[22]	$O(\log^2 n)$ time $O(\frac{n+m}{\log n})$ processors

Table 1.2: Parallel recognition algorithms for cographs. The implemented algorithm is marked in bold.

Definition 3 (induced subgraph)**.** *If $G' = (V', E')$ is a subgraph of G and G' contains all the edges of G that join two vertices in V', then G' is said to be an* induced subgraph *of G or spanned by V' and it is denoted by$G[V']$.*

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 (neighbours)**.** *We denote a set of* neighbours *of vertex v in a graph $G = (V, E)$ by $N(v) = \{x : xv \in E\}$.*

Definition 6 (path)**.** *A* path *is a graph P of the form*

$$V(P) = \{x_1, x_2, \ldots, x_l\}, \quad E(P) = \{x_1x_2, x_2x_3, \ldots, x_{l-1}x_l\}$$

This path P is usually denoted by $(x_1, x_2, \ldots, x_l)$ or x_1-x_2-$\ldots$-x_l or $x_1x_2x_3$.

We denote as P_L a path (graph or subgraph), consisted of exactly L vertices.

Definition 7 (complement)**.** *For a graph $G = (V, E)$ the* complement *of G is a graph $\overline{G} = (V, \binom{V}{2} \setminus E)$; Thus, two vertices are adjacent in $\overline{G}$ if and only if they are not adjacent in G.*

Definition 8 (cycle)**.** *If a walk $W = x_0x_1 \ldots x_l$ is such that $l \geq 3$, $x_0 = x_l$, and $x_i \neq x_j$ for all other $0 \leq i < j \leq l$, then W is said to be a* cycle*.*

Definition 9 (connected graph)**.** *A graph G is* connected *if for every pair $x, y \in V(G)$ of distinct vertices, there is a path from x to y.*

Definition 10 (forest, tree)**.** *A graph without any cycles is a* forest*, or an acyclic graph.*

A tree *is a connected forest.*

Definition 11 (union)**.** *A* union *of two graphs G and H is denoted by $G \cup H = (V(G) \cup V(H), E(G) \cup E(H))$.*

Definition 12 (join)**.** *Given disjoint subsets U and W of the vertex set of a graph, we write $E(U, W)$ for the set of $U - W$ edges, that is, for the set of edges joining every vertex in U to every vertex in W. The* join *of two graphs G and H is a graph $G \nabla H = (V(G) \cup V(H), E(G) \cup E(H) \cup E(G, H))$.*

Let us also introduce the following definitions, related to the trees.

Definition 13 (rooted tree)**.** *A* root *in tree T is a chosen special vertex in T. A* rooted tree *is a tree, which has a root.*

The *depth* of vertex v in a rooted tree T is the distance between v and root of T.

Definition 14 (children)**.** *The* children *of vertex v in a rooted tree T is $N(v) \cap \{x : depth[x] = depth[v] + 1\}$*

Definition 15 (subtree)**.** *A* subtree *of vertex x is a subgraph consisting of a set of vertices: x and vertices of the subtrees of children of x and edges: all edges in the subgraph induced by this set of vertices.*

Definition 16 (diameter)**.** *A* diameter *of graph G is denoted by $diam(G)$ and it is the maximum distance between any two vertices in G.*

Definition 17 (Least Common Ancestor, LCA)**.** *The* least common ancestor *of vertices x and y in rooted tree T is a vertex z with maximum depth and which is lying on both paths from x to root and from y to root.*

1.3 Cograph

We start by defining the class of complement reducible graphs, or cographs, for short:

Definition 18 (Theorem 11.3.1 [19])**.** *Let $G = (V, E)$ be a graph. Then,*

1. *if $|V| = 1$, then G is a* cograph*,*
2. *the union of two* cographs *is a* cograph*: if $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are vertex disjoint cographs, then $G = (V_1 \cup V_2, E_1 \cup E_2)$ is a cograph,*
3. *the complement of a* cograph *is a* cograph*: if G is a cograph, then $\overline{G}$ is a cograph.*

There are also alternative definitions that are equivalent to the above one.

Theorem 1. *Cograph can be built from one-vertex graphs in a series of union and join operations.*

Proof. If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are vertex disjoint graphs, then $G_1 \nabla G_2 = \overline{\overline{G_1} \cup \overline{G_2}}$.

On the other hand, let us make the induction assumption that $\overline{G}$ can be replaced by a series of join and union operations. The induction base is a trivial example of a cograph, i.e. one-vertex graph. The induction step is that we have two options what the previous operation before complement operation was:

- a complement operation – then these two operations can be erased and nothing will change,
- an union operation – then $G \cup H = \overline{\overline{G} \nabla \overline{H}}$, so the result of these two operations is $\overline{G} \nabla \overline{H}$, but G and H are smaller than $G \cup H$, so here induction assumption can be applied.

□

Theorem 2. *For every graph G it is a cograph if and only if it is a P_4-free graph.*

Proof. If none of two graphs we unite have an induced P_4, then of course we cannot find it in the new graph, because we have not any new edges. If we have the graph G, and $\overline{G}$ contains P_4 *a*-*b*-*c*-*d* as an induced subgraph, then $\{a, b\}, \{b, c\}, \{c, d\} \in E(G)$ and $\{a, c\}, \{a, d\}, \{b, d\} \notin E(G)$, but this means that *b*-*d*-*a*-*c* form the induced P_4 in G, the contradiction. □

Theorem 3. *For every graph G it is a cograph if and only if $diam(G) \leq 2$.*

Proof. If we unite two graphs, $diam(G \cup H) = \max\{diam(G), diam(H)\}$. If we join two graph G and H, then if we look at the distance between some vertices x and y, we must consider several cases:

- If x, $y \in V(G)$ or $x, y \in V(H)$ and the path between them existed before union, then nothing was changed,
- If $x \in V(G)$ and $y \in V(H)$, then the distance between them is 1,
- If x, $y \in V(G)$ or $x, y \in V(H)$ and the path between them did not exist before the union, then, without loss of generality, assume $x, y \in V(G)$, now it is sufficient to take vertex $z \in V(H)$ and we know $\{x, z\}, \{z, y\} \in V(G \nabla H)$, so the distance between x and y equals 2.

□

1.4 Cotree

Every cograph G has an alternative representation in the form of a so-called cotree.

Definition 19. *([9]). Let T be a* cotree *for some graph, then T is a valid cotree for G if and only if:*

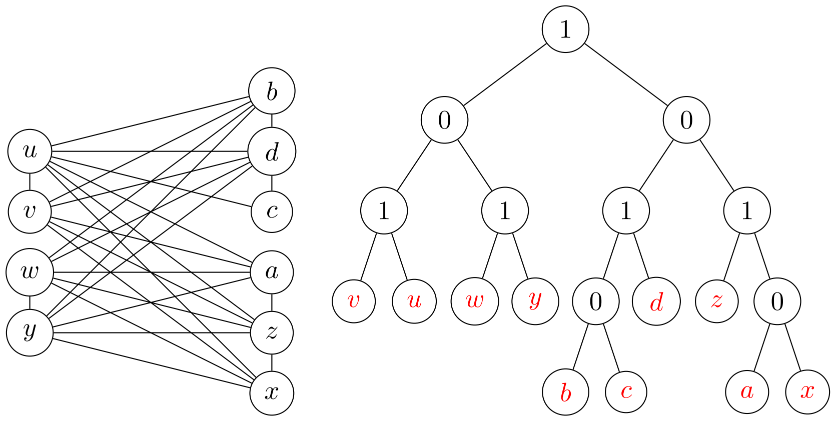


Figure 1.1: Cograph G and its cotree (taken from [6])

1. *T consists of* (0)- *and* (1)*-nodes, alternating with layers of the tree. The root is always* (1)*-node. In the other words, it consists of: on the 1st layer (root) of* (1)*-nodes, on the 2nd layer of* (0)*-nodes, on the 3rd layer of* (1)*-nodes, and so on,*
2. *The leaves of T are exactly the vertices from the cograph G,*
3. *The LCA of two leaves is a* (1)*-node if and only if these two nodes are neighbours in the cograph G.*

An example cograph G and its cotree T is presented in Figure 1.1.

Chapter 2

A linear time cograph recognition algorithm of Corneil, Perl and Stewart

2.1 Introduction

Let us consider a linear recognition algorithm by Corneil, Perl, and Stewart [9]. Historically, this was the first linear algorithm for cograph recognition that was ever designed. Its main ideas are intuitive, but it is very difficult to implement due to the large number of cases. The proof of this algorithm is difficult for the same reason: there are many cases with many subcases.

2.2 Idea

The algorithm idea is that any induced subgraph of a cograph is a cograph, which follows, for example, from the definition about P_4-free graph. So we will process the vertices one by one and modify the cotree, and if it ceases to be a cograph, the algorithm stops. Our algorithm will check this condition after every inserting of vertex from V.

The algorithm consists of several parts: `MARK`, `FIND-LOWEST`, `ADD-VERTEX`. Let us look at each of the parts in turn.

2.3 Function `MARK`

2.3.1 Idea

Consider an algorithm iteration for vertex x. Assume we have a cotree T for some induced subgraph of our graph G. We want to give the vertices of T some states, which are: “not-marked”, “marked” and “marked-and-unmarked”. At the beginning all vertices are “not-marked” and then immediately we change the state of every neighbour of x to “marked”.

There are two other cases when we change vertex state, and if vertex satisfies the given condition, then the changing must be applied. We change vertex state from “not-marked” to “marked” when at least one child of this vertex has state “marked-and-unmarked”. We change vertex state from “marked” to “marked-and-unmarked” when all children of this vertex have state “marked-and-unmarked”.

Let us denote *subtree leaves* by $L(x) = \{y : y \text{ is a leaf and is in the subtree rooted in } x\}$. So, we can create a table explaining what the vertex state means after `MARK` ends.

Vertex y state	How many neighbours of x are in $L(y)$:
“not-marked”	zero
“marked”	at least one
“marked-and-unmarked”	$\lvert L(y)\rvert$, i.e. all

2.3.2 Implementation

We denote by $d(w)$ the number of children of w in the cotree and by $md(w)$ the number of “marked-and-unmarked” children of w if w is “marked” – otherwise we assume it is equal to 0.

When we consider a new vertex x, then we need to change the state of every y from $N(x)$, that already exists in cotree, to “marked”. Further, while there exists a “marked“ node v for which $d(v) = md(v)$, we change the state of v to “marked-and-unmarked” and if its parent p state is “unmarked”, then we change its state to “marked”. Otherwise, we just increase $md(p)$ by one. Note that when we change the state of v to “marked-and-unmarked”, then its parent p now can have $d(p) = md(p)$, so we cannot just remember once all “marked” nodes v with $d(v) = md(v)$, but we need to constantly update our list of such vertices.

The root has only one child if and only if the graph is disconnected. If after the end of the algorithm there is a marked node and the graph is disconnected, then we change the root of the tree state to “marked”.

2.3.3 Pseudocode

```
M(v) = linked list of children of v;
MARK(x) {
  Mark all leaves of T which are adjacent to x;
  For (each marked node u of T with d(u) == md(u)) {
    unmark(u);  // change state from marked to marked-and-unmarked
    md(u) = 0;
    if (u != root) {
      mark (parent (u));
      md(w) += 1;
      insert u at the head of M(w);
    }
  }
  If (any vertex is marked and d(root) == 1) {
    mark(root); // change state from unmarked to marked-and-unmarked
  }
}
```

2.4 Function `FIND-LOWEST`

2.4.1 Idea

The next part of our algorithm is a function `FIND-LOWEST`. Its idea is that by relying on states of vertices in cotree T after calling `MARK`(x), we check whether the graph ceases to be a cograph when adding our vertex. The theorems in Section 2.5.2 help us to determine whether after adding x to G it ceases to be cograph. We either return that $G+x$ is not a cograph or we return a node α – the lowest marked node of cotree.

2.4.2 Implementation

Let us introduce the following definitions and notation.

Definition 20. *A node y is* properly marked *if and only if it is a* (1)*-node, it is marked and $md(y) = d(y) - 1$.*

Definition 21. *A* legitimate alternating path *is a path of adjacent alternating properly marked* (1)*-nodes and not "marked"* (0)*-nodes, the extreme points of which are* (1)*-nodes.*

Let us denote the graph by G, a cotree of G by T. Let x be the current vertex we are trying to insert to G, u be the lowest marked node so far examined, w be the previous value of u; i.e. the lowest marked node examined before u. Let y be a marked (1)-node which is not properly marked or a marked (0)-node, if either exists in cotree.

At each step of the algorithm we choose u and w from T and check if the path between them is a legitimate alternating path. At the first step w is initialized to the root and u initialized to any marked node. At another step we make w equal to u, and u equal to another arbitrary marked node. At the end of each step we change the state of u to "marked-and-unmarked".

Let us divide the function into phases. The first phase is initialization. The second phase is choosing arbitrary marked vertex u and checking some conditions for it. The third phase is checking if the path between u and w is a legitimate alternating path and unmarking vertices along this path.

2.4.3 Pseudocode

```
FIND-LOWEST() {
  // Phase 1.
  y = trash;
  If (R is not marked) {
    // G+x is not a cograph: condition (iii)
    return NOT_COGRAPH;
  }
  if (md(R) != d(R) - 1) {
    y = R;
  }
  unmark(R);  // change state from marked to marked-and-unmarked
  md(R) = 0;
  u = w = R;
  while (marked vertex exists in T) {
  // Phase 2
  u = arbitrary marked vertex;
  if (y != trash) {
    // G+x is not a cograph: condition (i) or (ii)
    return NOT_COGRAPH;
  }
  if (label (u) == 1) {
    if (md(u) != d(u) - 1) {
      y = u;
    }
    if (parent (u) is marked) {
      // G+x is not a cograph: conditions (i) and (vi)
      return NOT_COGRAPH;
    }
    else {
      t = parent (parent (u));
    }
  }
  else {
    y = u;
    t = parent (u);
  }
  unmark (u);
  md (u) = 0;
  // phase 3
  while (t != w) {
    if (t == R) {
      // G+x is not a cograph: condition (iv)
      return NOT_COGRAPH;
    }
    if (t is not marked) {
        // G+x is not a cograph: condition (iii) or (v) or (vi)
        return NOT_COGRAPH;
      }
      if (md(t) != d(t) - 1) {
        // G+x is not a cograph: condition (ii)
        return NOT_COGRAPH;
      }
      if (parent (t) is marked) {
        // G+x is not a cograph: condition (i)
        return NOT_COGRAPH;
      }
      unmark(t);
      md(t) = 0;
      t = parent (parent (t));
    }
    // Reset w for next choice of marked vertex
    w = u;
  }
}
```

2.5 Function ADD-VERTEX

2.5.1 Implementation

This method builds the next part of cotree T from the bottom by combining everything that was said above. It adds vertices one at a time by calling at the beginning MARK function and then, after handling edge cases, it calls the second function FIND-LOWEST. Next, it adds a vertex to the cotree in such a way that the rules of the cotree are fulfilled, in particular, rule 3 about LCA.

2.5.2 Pseudocode

```
ADD-VERTEX(V = (v1, .., vn), E) {  // V-vertices, E-edges
  // initialization
  Create a new (1) cotree node R;
  if (v1 and v2 are neighbours) {
    add v1, v2 as children of R;
  }
  else{
    create a new (0) cotree node N;
    add N as a child of R;
    add v1 and v2 as children of N;
  }
  // iteratively incorporate v3, .., vn into T
  for (x in (v3,...,vn)) {
    MARK (x);
    if (all nodes of T were marked and unmarked) {
      add x as a child of R;
      continue;
    }
    if (no nodes of T were marked) {
      if (d(R) == 1) {
        add x as a child of the only child of R;
      }
      else {
        create a new (label(u) ^ 1) node R with one child and
        two grandchildren: x and the old root;
      }
      continue;
    }
    u = FIND-LOWEST();
    A[0] = marked-and-unmarked children of u;
    A[1] = unmarked children of u;
    if (A[label(u)].size == 1) {
      if (w is a leaf and A[label(u)] == {w}) {
        add a new (1) cotree node ((0) cotree node) in place of w
        and make w and x children of this node;
      }
      else {
        add x as a new child of w;
      }
    }
    else {
      remove all elements of A[0] from u;
      add these removed elements as children of a new (label(u)) node y;
      if (u is a (0)-node) {
        add a new (1) cotree node as a child of u;
        add x and y as children of this new (1)-node;
      }
      else {
        remove u from its parent and add y in its place;
        add a new (0)-node as a child of y;
        add x and u as children of this new (0)-node;
      }
    }
  }
}
```

2.6 Correctness

2.6.1 `MARK`

Note that for vertex v the condition $d(v) = md(v)$ means that all children of v are "marked-and-unmarked". For every child w of v it means that it once had $d(w) = md(w)$, so all its children are "marked-and-unmarked". And applying such logic recursively to children of w and so on, we get that every y from $L(v)$ is "marked-and-unmarked", so once it was "marked", but a leaf in the tree was once "marked" if and only if it is the neighbour of x and hence y is a neighbour of x.

Accordingly, if we look again at the two types of state changes that were declared in the section `MARK` we can see that vertex can become "marked" if it is a leaf and a neighbour of x, or if at least one of its children is "marked-and-unmarked". Hence, for this "marked-and-unmarked" child w of v, as it was said above, we know that $L(w)$ consists only from children of x.

2.6.2 `FIND-LOWEST` implementation

As the first step we checked if the path between u and root is correct, at the second step we made w equals to u and u equals to any marked node. And we again checked if the path between u and w is correct; Thus, since the path between u and w is correct and the path between w and *root* is correct, then the path between u and *root* is correct, and so on. In this way we checked the condition 2.1 from the Theorem 4.

2.6.3 Theorem for `FIND-LOWEST`

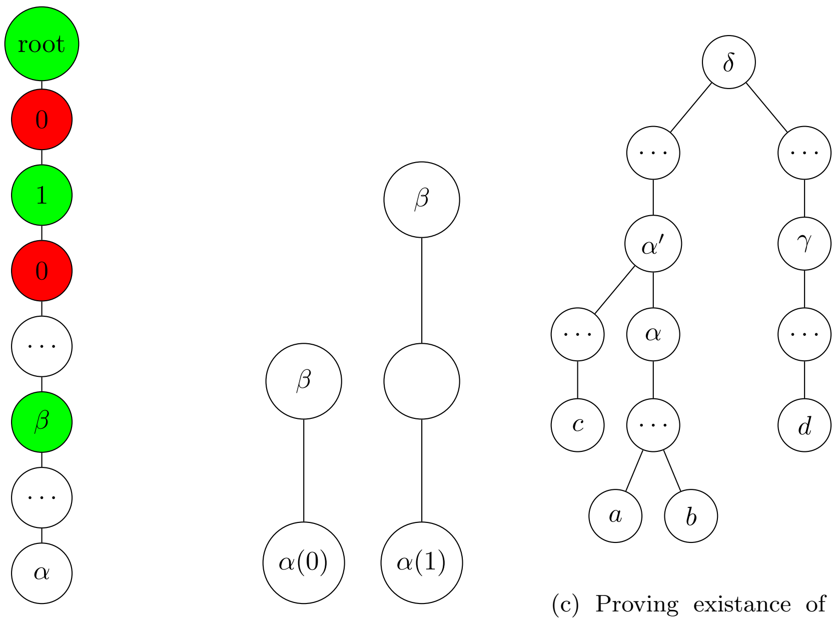
In order to state the theorem, we need to introduce the following definitions and notation.

Let M denote the set of (0)- and (1)-nodes of T which are marked after the procedure `MARK` has been performed, and let α be the lowest node in M and let β be the second lowest node.

Theorem 4 (Theorem 1 [9]). *If G is a cograph with cotree T then $G + x$ is a cograph if and only if one of the following holds:*

1. *M is empty,*
2. *M is not empty and:*
   - *$M \backslash \alpha$ consists of exactly the (1)-nodes of a (possibly empty) legitimate alternating path which ends at R. (See Figure 2.1a),*
   - *α is either a (0)-node whose parent is β, or α is a (1)-node whose grandparent, if it exists, is β. (see Figure 2.1b)*

Theorem 5 (Theorem 1 [9]). *The given graph G is a cograph if and only if none of the following conditions are true:*



(a) Legitimate alternating path

(b) Parent or grandparent of α is β

(c) Proving existance of P_4 in condition 1 of the theorem 5

Figure 2.1: Examples of cograph structure

1. *there exists a* (0)*-node in* $M \setminus \{\alpha\}$,
2. *there exists* $v \in M \setminus \{\alpha\}$*, such that* v *is a* (1)*-node and is not properly marked,*
3. *there exists* $y \in M \setminus \{\alpha\}$*, such that* $y \neq R$ *and the grandparent of* $y \notin M \setminus \{\alpha\}$,
4. *The vertices of* $M \setminus \{\alpha\}$ *do not lie on one path to* R*,*
5. α *is a* (0)*-node whose parent is not* β*,*
6. α *is a* (1)*-node which has a grandparent which is not* β*.*

It is easy to prove that this theorem is equivalent to the Theorem 4.

2.6.4 Theorem 5

Now let us prove that if condition 1 of the Theorem 5 is true, then G is not a cograph. The other cases are similar.

From now on let $des(x)$ denote the leaves of the subtree of x, γ be some (0)-node in $M \setminus \{\alpha\}$ and δ be the LCA of α and γ.

We have to consider four cases, depending of what label do α and δ have. Here we consider their labels are both 0 (see Figure 2.1c). The other cases follow similarly.

Let α' be the parent of α. As it was said above, if all leaves in the subtree are the neighbours of x, then this node is "marked-and-unmarked", but we look at the nodes in M. Such nodes have at least one neighbour of x and one not neighbour of x in their subtrees. So, $a \in des(\alpha)$ is the neighbour of x,$b \in des(\alpha)$ is not a neighbour of x, $c \in des(\alpha') \setminus des(\alpha)$, $d \in des(\gamma)$ (except one case under) is the neighbour of x. If c is the neighbour of x, the P_4 is $b - c - x - d$, else $b - c - a - x$. But if $\delta = \gamma$, $d \in des(y) \setminus des(\theta)$, where θ is the child of γ on the $\alpha - \gamma$ path.

2.6.5 `FIND-LOWEST`

Function `FIND-LOWEST` finds α, the lowest marked node of cotree.

2.6.6 `ADD-VERTEX`

Let us prove of one of many cases of the function `ADD-VERTEX` – when u is a (0)-node and the size of $A[0]$ does not equal to 1. The other cases are similar. We use the notation from function `ADD-VERTEX` implementation.

Theorem 6. *For each pair of leaves (a, b) in the modified cotree T', where $x \notin \{a, b\}$, the labels of $LCA_T(a, b)$ and $LCA_{T'}(a, b)$ are the same.*

Proof. We call a vertex *beautiful* if it is from the subtree of some vertex from $A[0]$ and *ugly* if it is from the subtree of some child of u, such that u is not in $A[0]$.

There are two states of every vertex – is in the subtree of u, which in turn divided into beautiful and ugly, and is not in the subtree of u.

- if a and b are not from the subtree of u, then their LCA stays the same,
- If exactly one node (only a or only b) is from the subtree of u and the other is not from subtree of u, then their LCA is still the same,
- if a and b are both beautiful or both ugly, then if their LCA is not u, then it remains the same. If their LCA is u, then if a and b are beautiful, it becomes y, that has the same label as u; and if a and b are ugly, then their LCA is still u.
- if exactly one between a and b is beautiful and the other is ugly, then their LCA is still u.

From the description of the algorithm, all vertices from $A[0]$, i.e. beautiful ones are now the children of y. Now we consider a beautiful node l and an ugly node r (see Figure 2.2). $L(l)$ are all neighbours of x, because l is "marked-and-unmarked". There are no neighbours of x in $L(r)$, because r is "unmarked". □

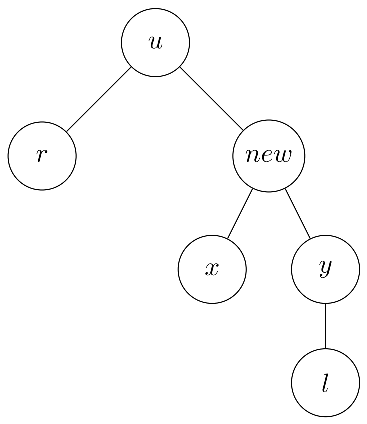


Figure 2.2: `ADD-VERTEX` special case proof

Now it's time to consider the case when $a = x$ or $b = x$.

Theorem 7. *For every $b \neq x$ from $L(u)$, the label of the LCA of x (the vertex that we are inserting to the cograph) and b, is correct.*

Proof. Let us split the proof into the several cases:

- If b is from $L(l)$ then their LCA is a new (1)-node and it is correct, because in $L(l)$ are only neigbours of x,
- If b is from $L(r)$ then their LCA is u and it is correct, because u is (0)-node and in $L(r)$ there are no neighbours of x,
- If b is not from $L(u)$, then we know that the path between u and R is a legitimate alternating path.
  - If the LCA of x and b is a (0)-node, and we know it is not marked, i.e. among its children there are not "marked-and-unmarked" vertices. Due to the existence of a legitimate alternating path we know than in the subtree of LCA, excluding the subtree of its properly marked (1)-node, there are no "marked" vertices. Therefore, every such child of LCA is "unmarked", and this logic can be recursively applied conclude that $L(LCA)$ has no neighbours of x, making it the correct LCA,
  - If the LCA is a (1)-node, and we know it is properly marked, it has exactly one not "marked-and-unmarked" child, and this child is an "unmarked" (0)-node and the other children are "marked-and-unmarked". For every such child c we know that $L(c)$ consists only of neighbours of x, making it the correct LCA too.

□

2.7 Complexity

Theorem 8. *Each iteration takes $O(\deg(x))$ time.*

Proof. `MARK`: each node except root has a minimum of two children. If we imagine that all vertices, which are not neighbours of x don't exist, then the tree has $\deg(x)$ leaves and the previous layer has at most $\frac{\deg(x)}{2}$ vertices, the one before that has at most $\frac{\deg(x)}{4}$ and so on. To calculate the total complexity we need to sum $\deg(x) + \frac{\deg(x)}{2} + \frac{\deg(x)}{4} + \dots$, which forms a geometric progression. The sum is at most $2 \cdot \deg(x)$, so it is $O(\deg(x))$. Therefore, the size of M is also $O(\deg(x))$. For each node, processing time is $O(1)$, and the total time complexity of this function is $O(\deg(x))$.

`FIND-LOWEST`: each node from M is processed in $O(1)$ time, the total complexity of this function is also $O(\deg(x))$.

`ADD-VERTEX`: all cases of inserting require $O(1)$ time, but the operation of removing elements of A from u to y is $O(\deg(x))$, because $|A| \leq |M|$.

□

Therefore, the total time complexity of the algorithm is $O(\sum\limits_{v \in V} \deg(v)) = O(n + m)$.

Chapter 3

A linear time cograph recognition algorithm of Bretscher, Corneil, Habib and Paul

3.1 Introduction

Let us consider a linear recognition algorithm by Bretscher, Corneil, Habib and Paul [6]. The correctness of this algorithm may be difficult to understand, but the idea of the algorithm is quite easy. The hardest part of the linear implementation is to choose the right data structures. The coding part is long enough.

3.2 Idea

We choose an arbitrary permutation of our graph vertices. We define a LexBFS on it, then LexBFS-, which can be implemented using LexBFS. Finally, we define `Neighbourhood Subset Property` on a permutation. Having these definitions, our algorithm is short and simple:

```
LexBFS_recognition_cograph(G) {
  a = arbitrary_permutation_of_vertices(G);
  b = LexBFS(G, a);
  GC = complement(G);
  c = LexBFS(GC, b);
  d = LexBFS(G, c);
  if (Neighbourhood_Subset_Property(c) and
    Neighbourhood_Subset_Property(d)) {
    return is_cograph;
  } else {
    return not_cograph;
  }
}
```

3.3 LexBFS

3.3.1 Idea

We process vertices one by one and divide them into groups according to the property that the vertices in the same group have the same set of already processed neighbours.

3.3.2 Implementation

Let us introduce the following notation: $N'(x) = \{v \colon v \in N(x)$ and v has not been processed$\}$.

We only store groups consisting of unprocessed vertices. A list is maintained comprising these groups. Initially, the list contains only one group that includes all graph vertices. In each step we choose the leftmost group A, then choose the leftmost vertex x in A. For every group B we select all neighbours of x in B, remove them from B and place them in a new group C, which is inserted directly before B in the list. Subsequently, we eliminate x from A and mark it as "processed". It is important to note that B may be the same as A. Refer to the pseudocode in Section 3.3.3 and the example in the Table 3.1 for further details.

3.3.3 Pseudocode

```
LexBFS (G, a) {
  L = a; // L is list of groups of vertices
  i = 0;
  while (L.not_empty()) {
    x = L[0][0]; // the leftmost vertex in the leftmost group
    ans[x] = i++; // the number of x in final permutation is i
    for (g in L) {
      q = neigbours of x in g;
      remove all vertices of q from g;
      insert q in L before g;
    }
    remove x from L[0];
  }
  return ans;
}
```

$\sigma(v)$	v	$N'(v)$	Partitions
			$\{x\ d\ y\ u\ v\ w\ c\ a\ z\ b\}$
1	x	$\{u\ v\ w\ y\ z\}$	$\{y\ u\ v\ w\ z\}\ \{d\ c\ a\ b\}$
2	y	$\{a\ b\ c\ d\ w\ z\}$	$\{w\ z\}\ \{u\ v\}\ \{d\ c\ a\ b\}$
3	w	$\{a\ b\ c\ d\ z\}$	$\{z\}\ \{u\ v\}\ \{d\ c\ a\ b\}$
4	z	$\{a\ u\ v\}$	$\{u\ v\}\ \{a\}\ \{d\ c\ b\}$
5	u	$\{a\ b\ c\ d\ v\}$	$\{v\}\ \{a\}\ \{d\ c\ b\}$
6	v	$\{a\ b\ c\ d\}$	$\{a\}\ \{d\ c\ b\}$
7	a	$\{\}$	$\{d\ c\ b\}$
8	d	$\{b\ c\}$	$\{c\ b\}$
9	c	$\{b\}$	$\{b\}$
10	b	$\{\}$	

Table 3.1: LexBFS example for the cograph G from Figure 1.1

3.4 LexBFS-

3.4.1 Idea

LexBFS- is similar to LexBFS with one difference – we specify further how we divide one group into two ones.

Definition 22 (Definition 1 from [6]). *The vertices in the leftmost partition of line* 5 *of Algorithm LexBFS(G, V) are tied when their neighbourhoods of processed vertices are identical. We refer to such a set of tied vertices as a* slice, *S.*

When vertices from one slice are divided into two slices, we call it a tie breaking mechanism*.*

The borders *of the slice S are called two numbers l and r, such that the set of vertices of S is exactly the same as the set of vertices located at indexes between l and r in the answer permutation of LexBFS.*

The tie breaking mechanism in LexBFS is very simple – by taking the leftmost vertex of the slice. As for LexBFS-, it differs from LexBFS in the way it chooses the vertex from the slice – it selects the leftmost vertex in the given permutation among all other vertices in the slice.

3.4.2 Implementation

Let us introduce the definitions that are required to understand this part of the algorithm.

Definition 23 (Definition 2 from [6])**.** *Let $S = [x, S^A(x), \langle S^N(x)\rangle]$ be an arbitrary slice constructed during a LexBFS sweep where x is the first vertex of S. We consider only the first level of nested subslices of S. Let $S^A(x) =$ [the subslice of S of vertices adjacent to x]. Each subsequent subslice of S contains vertices not adjacent to x. Let $\langle S^N(x)\rangle = (S_1^N(x), S_2^N(x), \ldots, S_k^N(x))$ denote the subslices not adjacent to x in S.*

We similarly define $\overline{S} = [x, \overline{S^A(x)}, \overline{\langle S^N(x)\rangle}]$ with respect to the complement $\overline{G}$.

For the previous example (see Figure 1.1), $S^A(x) = \{y, u, v, w, z\}$, $S_1^N(x) = \{a\}$, $S_2^N(x) = \{d, c, b\}$. If initial ordering is $xyuwzadcb$, then the slices for the $\overline{G}$ are $x[dacb][z][uwyv]$.

It is worth to say that when we write $S^A(x)$ and $\langle S^N(x)\rangle$, it does not mean that we start LexBFS from the beginning with x as a first vertex – but we do it inside its slice. With fixed initial permutation the total size of $S^A(x)$ and $\langle S^N(x)\rangle$ is the size of the slice of x minus 1. For the first element of permutation, this sum is equal to $n - 1$.

The example of nested slices for vertex y for LexBFS of G is $x[y[wz][uv]][a][dcb]$, so $S^A(y) = [wz]$ and $S_1^N(y) = [uv]$. For vertices y and a for LexBFS of $\overline{G}$ is $x[a[d][][cb]]][z][y][y[uv][w]]$, so $S^A(y) = [uv]$ and $S_1^N(y) = [w]$.

Implementation and complexity

The optimal implementation LexBFS- on G runs in $O(n+m)$ time, that is, if we implement L as a list of lists, representing the groups of unprocessed vertices. For each vertex, we can maintain a pointer to a list(a group) it belongs to. Removing the neighbours of x from some group A can be done in $O(\deg(x))$ time. This is achieved by examining the pointer to the group of each unprocessed neighbor y and removing y from A. If y is the first element removed from A during the iteration of the algorithm, a new list containing only y is created and placed before A in L. If y is not the first element removed from A, it is inserted into the list before A in L. This operation takes $O(1)$ time for each y.

While processing vertex x we store the number of vertices in $S^A(x)$ and the total capacity of $S_i^N(x)$, which indicate the number of neighbours and not neighbours of its slice respectively. With this information, for every index i we know for each vertex y if i is the end of $S^A(y)$. The list of such vertices is denoted as D. For each z in D we know that at this moment the neighbours of z, i.e. $S^A(z)$ are processed. This implies that the non-neighbours of z have already been partitioned into the slices now. Therefore, we know $S_j^N(z)$ for every j, enabling us to outline the borders of each $S_j^N(y)$ in an array, which proves beneficial in the subsequent function.

3.5 Neighbourhood Subset Property

To understand this property we need to introduce it first formally via the following definitions:

Definition 24 (Definition 4 from [6])**.** *Given a slice $S_i^N(v)$ we define the* processed neighbourhood *of $S_i^N(v)$ to be $N_i(v) = \{y\colon y$ is processed and for every z from $S_i^N(v)$ holds $y \in N(z)\}$.*

Definition 25 (Definition 5 from [6])**.** *LexBFS satisfies the* Neighbourhood Subset Property *if and only if $N_{i+1}(v) \subseteq N_i(v)$, for all $v \in V$ and for all $i \geq 1$.*

Before we elaborate on the meaning of this property (see Section 3.8), let us proceed with its implementation details.

3.5.1 Implementation

We use the following notation: let c be the resulting ordering from the second LexBFS pass in the algorithm and d be the resulting ordering from the third LexBFS pass in the algorithm.

For every x and i we know the borders of $S_i^N(x)$ in c and $\overline{S_i^N(x)}$ in d. Therefore, for each x we can compare the set of processed neighbours of first vertices a and b of $S_i^N(x)$ and $S_{i+1}^N(x)$ respectively. We can check only the first vertices in these slices because due to the definition of a slice, all vertices in it have the same set of processed vertices.

For every vertex v let us define $used[v] = 1$ if v is a neighbour of a in G, and $used[v] = 0$ otherwise. And for every neighbour w of b we just need to verify if $used[w] = 1$.

When we do it for the complement of the graph and the result permutation d, the procedure is similar. We need every processed neighbour of $\overline{S_i^N(x)}$ to be a neighbour of $\overline{S_{i+1}^N(x)}$, because a neighbour in G is not a neighbour in $\overline{G}$. Additionally, we need to separately verify that $\overline{S_{i+1}^N(x)}$ does not contain any element of $\overline{S_i^N(x)}$ as a neighbour.

3.6 Cotree

3.6.1 Idea

The wonderful fact is if we start building a cotree with some vertex x, then we just need to build a path of (0)-nodes and (1)-nodes and cotree for slice $S_i^N(x)$ corresponds to the i^{th} (0)-node and the cotree for slice $\overline{S_i^N(x)}$ corresponds to the i^{th} (1)-node. Thus, we can just build these parts of the cotree recursively, using the fact, that we already know every $S_i^N(x)$ and $\overline{S_i^N(x)}$ recursively.

3.6.2 Implementation

```
Construct_Cotree (borders [l,r] of slice S) {
  S = result_permutation[l..r];
  v = S[0];
  S(v) = slices borders for v in G;
  /* S(v)[i].first and S(v)[i].second
  are the borders for i-th slice of v */
  SC(v) = slices borders for v in complement of G
  create empty cotree T;
  set root of T to v;
  m = max(number of S(v), number of SC(v));
  /* if number of S(v) is greater than the number of SC(V),
  then S(v)[m] is not empty and S(v)[m + 1] is empty */
  for (i in [0..m]) {
    z = new (0)-vertex;
    z.add_child(T.root);
    if (S(v)[i] is not empty) {
      z.add_child(Construct_Cotree(S(v)[i]));
    }
    y = new (1)-vertex;
    y.add_child(z);
    set root of T to y;
    if (SC(v)[i] is not empty) {
      p = new (0)-vertex;
      y.add_child(p);
      p.add_child(Construct_Cotree(SC(v)[i]));
    }
  }
  return T.root;
}
```

3.7 Complexity

3.7.1 LexBFS and LexBFS-

LexBFS on $\overline{G}$ also runs in $O(n+m)$ time, because in the line 10 of LexBFS implementation on G (see Section 3.3.3) we can just move q to the right of g. Although there are the neighbours of x in the graph G within the group g, then there are no neighbours of x in $\overline{G}$ within the group g. Similarly, there are only neighbours of x in $\overline{G}$ within q. So when we insert g not to the left of q, but to the right, then their order now is (q, g) and in q there are no neighbours of x in $\overline{G}$, while in g there are only neighbours of x in $\overline{G}$. Thus, the list was partitioned into two lists: the left list are neighbours and the right one are not the neighbours. This is what we wanted to show.

LexBFS- on G can also be implemented in $O(n+m)$ time. We achieve this by making the initial permutation equal to the given parameter permutation π and running the usual LexBFS algorithm. So when we select the leftmost element, it is processed earliest with respect to other elements in its slice. This behavior arises because the initial permutation is π and the elements inside the slice do not change places relative to each other. Our only modification involves removing some vertices from the slice in the same order and inserting them into a new slice.

3.7.2 Neighbourhood Subset Property

Let us denote by $NS(S)$ the number of all non-empty $S_i^N(x)$ for every i and x for some slice S.

Theorem 9. $NS(S) \leq |S| - 1$.

Proof. It can be proved by induction. The base case for $n = 1$ holds since $NS(S) = 0$. In the induction step we know that for every i $NS(S_i^N(x))$ by induction is no more than $|S_i^N(x)| - 1$. Therefore, if m is the number of non-empty slices $S_i^N(x)$, then:

$$NS(S) = m + \sum_{i=1}^{m} NS(S_i^N(x)) \leq m + \sum_{i=1}^{m}(|S_i^N(x)| - 1) = m + \sum_{i=1}^{m} |S_i^N(x)| - m$$
$$= \sum_{i=1}^{m} |S_i^N(x)| \leq |S| - 1$$

The last inequality is in fact equality if and only if x has no neighbours in S. □

Theorem 10. *For every vertex v it is the first vertex for no more than one $S_i^N(x)$ for some x and i.*

Proof. Let us consider the slice $S_i^N(x)$ with the largest size, in which v is the first vertex. $S_i^N(x)$ is recursively divided into v, $S^A(v)$ and $\langle S^N(v)\rangle$, but v cannot be the first vertex of $S_i^N(v)$ for any i. And we know that $S_i^N(x)$ is the largest slice, so $S_i^N(v)$ for some i is the only option to v to be the first vertex in the slice. □

Due to the previous theorems, we can build an injective mapping between the set of non-empty slices and the set of vertices. All representatives, i.e. first vertices of slices, are unique, so the sum of their degrees is no more than $2 \cdot m$.

Therefore, we can just iterate through every existing non-empty slice $S_i^N(x)$ and check if its neighbourhood of processed vertices is a superset of $S_{i+1}^N(x)$, if it exists. As mentioned above, we have at most $n-1$ such non-empty slices $S_i^N(x)$ for every x and i and the first vertex of each slice is unique. Consequently, they have no more than $2 \cdot m$ neighbours in total. Let v be the first vertex of $S_i^N(x)$ and w be the first vertex of $S_{i+1}^N(x)$. Comparing the sets of processed neighbours of v and w takes $O(\deg(v) + \deg(w))$ time. Therefore, the total time complexity is $O(n + m)$ time.

3.8 Correctness

The idea of the proof is based on the fact, that there is a bijection between cographs and cotrees. Therefore, if a valid cotree exists for this graph, then this graph is a cograph.

We need to understand when we can build a cotree from the slice sequences. Slices $S_i^N(x)$ corresponds to sub-cotrees, as mentioned in the Cotree section. The processed neighbours of $S_{i+1}^N(x)$ form a subset of processed neighbours of $S_i^N(x)$. Equivalently, the processed neighbours of vertices of $(i+1)^{th}$ (0)-node subtree, are a subset of the processed neighbors of vertices in the i^{th} (0)-node subtree. Consider any vertex z from $S_{i+1}^N(x)$. If some y is a processed neighbour of z, then their LCA is (1)-node due to the cotree property. But some vertex t from $S_i^N(x)$ is in the subtree of the i^{th} (0)-node, i.e. in the subtree of grandchild of the $(i+1)^{th}$ (0)-node. This implies that LCA$(t, y) =$ LCA (y, z). Therefore, it is (1)-node.

Furthermore, LCA of a vertex from $S_i^N(x)$ and a vertex from $S_{i+1}^N(x)$, is the $(i+1)^{th}$ (0)-node. Consequently, they can not be neighbours if this graph is a cograph.

In summary, the statement "a valid cotree exists if the neighbourhood subset property is true for both results permutations" aligns with the reasoning presented above.

Similar reasoning also holds also for $\overline{G}$ and its corresponding slices, i.e. $\overline{S_i^N(x)}$ instead of $S_i^N(x)$.

Chapter 4

A $O((m+n)\cdot\log n)$ time cograph recognition algorithm of Dahlhaus

4.1 Introduction

Let us consider a parallel recognition algorithm in $O(\log^2 n)$ time and $O(n+m)$ processors by Dahlhaus [10]. We will consider the sequential version of this algorithm in $O((m+n)\cdot\log n)$ time. This algorithm is easier than the previous ones, but it has a big constant hidden in the asymptotic notation, because of many operations performed in every iteration.

4.2 Idea

We proceed with finding the cotree for our graph by dividing the graph into groups of induced subgraphs and recursively computing cotrees on them and merging them into one cotree. Once we have the cotree, we can determine whether the graph is a cograph. Let us assume the cotree T that we have found corresponds to $G' = (V, E')$. Our goal is to check that $E = E'$. To check this we perform two actions:

- for every connected pair of vertices $\{x,y\} \in E$ in our graph we check that $\mathrm{LCA}(x, y)$ in the cotree is a (1)-node,
- for every (1)-node z in the cotree, we compute the number of pairs of leaves x, y, such that $\mathrm{LCA}(x, y) = z$. It is equivalent to getting the number of edges in the graph corresponding to the cotree.

Suppose that there exists a vertex v such that $\frac{n}{a} \le \deg v \le \frac{n(a-1)}{a}$. The case when such vertex does not exists considered in the Section 4.6. Next, we

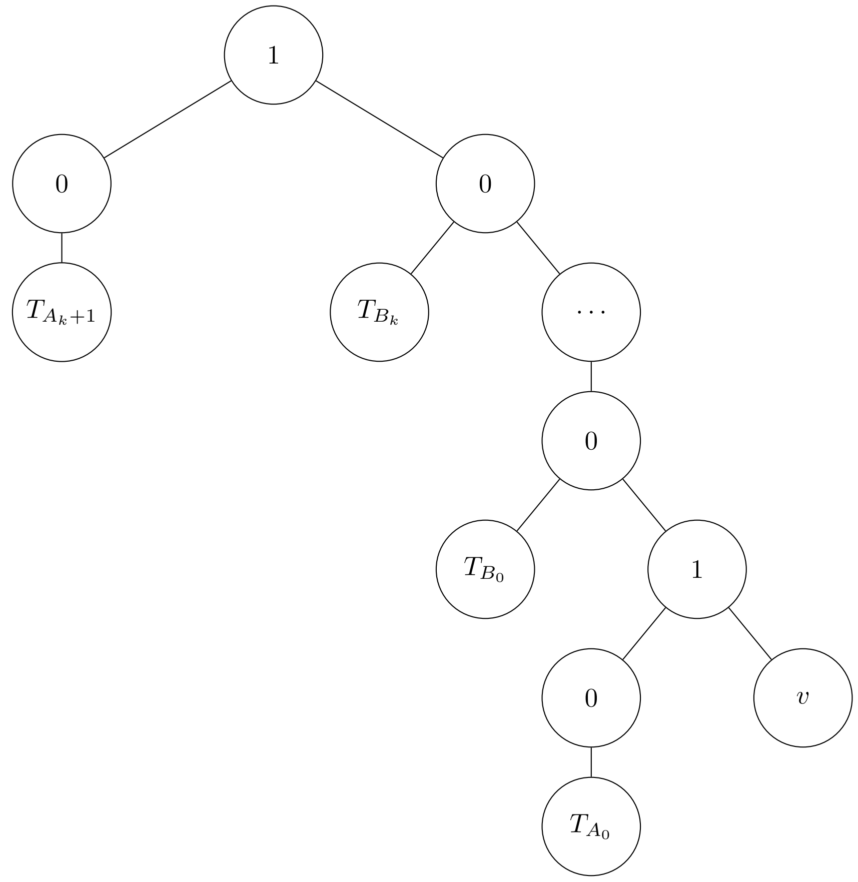


Figure 4.1: Cotree building procedure

divide the remaining vertices into two induced subgraphs – A, which consists of the neighbours of v, and B, which consists of non-neighbours of v. Then we partition B into connected components, i.e. induced subgraphs B_0, B_1, …, B_k, and reorder them according to some rules. After that, by using this reordered partitioning we divide A into induced subgraphs A_0, A_1, …, A_{k+1}. Finally, we recursively compute the cotrees for every A_i and B_i and connect them. We do it by connecting the root of the cotree of B_i to the i^{th} (0)-node and connect the root of the cotree of A_i to the i^{th}(1)-node (see Figure 4.1).

```
add(type t, graph H) {
  x = new (t)-node;
  x.add_child(T.root);
  T.root = x;
  if (t == 0) {
    x.add_child(build_cotree(H).root);
  } else {
    y = new (0)-node;
    x.add_child(y);
    y.add_child(build_cotree(H).root);
  }
}

build_cotree(G) {
  v = arbitrary chosen vertex from G;
  compute B[0], .., B[k];
  reorder B[0], .., B[k];
  compute A[0], .., A[k + 1];
  create empty cotree T;
  T.root = v;
  for (i in [0..k+1]) {
    add(1, A[i]);
    if (i == k + 1) {
      break;
    }
    add(0, B[i]);
  }
}
```

4.3 Implementation

Now all that remains is to understand how to reorder $\{B_i\}$ and compute $\{A_i\}$.

4.3.1 Reordering B_i

Definition 26. *For a set $W \subseteq V$, we denote by $\Gamma(W) = \{v \in V \setminus W\colon$ there exists $w \in W$ such that $\{v, w\} \in E\}$ the set of neighbours of W which are not in W.*

When we compute B_i, let us take one element $x \in B_i$. For x we compute the set $C_i = \{v\colon v \notin B_i$ and $\{x, v\} \in E\}$. Later we will show that $C_i = \Gamma(B_i)$. It is sufficient to sort the set $\{B_i\}$ by the size of $\{C_i\}$ in descending order using bucket sort.

4.3.2 Computing A_i

Next, for all $i = 0, 1, \dots, k-1$ we compute $C_i \setminus C_{i+1}$. The induced subgraph A_{i+1} is formed by the vertices in $C_i \setminus C_{i+1}$. Similarly, A_0 is induced by $V \setminus (\{v\} \cup C_0)$ and A_{k+1} is induced by C_k. The effective implementation requires to find the largest j, such that element y appears in C_j, then $y \in A_{j+1}$. If y does not appear in any C_j, then we know that $y \in A_0$.

4.3.3 Checking the correctness of the cotree

The algorithm for finding LCA in $O(\log n)$ time for query and $O(n \cdot \log n)$ pre-calculation time, where n is the size of the tree is well-known, so it will not be discussed here. The curious reader can be referred to [18].

Let us find, for every (1)-node z the number of leaves x, y, such that $\mathrm{LCA}(x, y) = z$. Assume that z has p children denoted by $z_0, \ldots, z_{p-1}$. For every i it holds that LCA of two leaves from subtree rooted in z_i is different from z. But $\mathrm{LCA}(x, y) = z$, for every x, y where x is a leaf from z_i and y is a leaf from z_j with $i \neq j$. Therefore, we just need to calculate the number of such pairs.

Definition 27. *For every vertex q, let us denote by $leaves[q]$ the number of leaves in subtree of q.*

It is easy to spot that for every q $leaves[q]$ can be calculated using well-known techniques of dynamic programming and depth first search, so we skip its details. Let us instead continue with the main algorithm. To get the number of pairs x, y, such that $\mathrm{LCA}(x, y) = z$ we initialize a variable *sum* by zero. Then, for every $i = 0, \ldots, p-1$ we just multiply $leaves[z_i]$ by $\sum_{j \neq i} leaves[z_j]$ and add this value to *sum*. Finally, we divide the *sum* by 2, because each pair was counted twice.

4.4 Complexity

Computing the connected components, i.e. $\{B_i\}$ can be done in $O(n+m)$ time. Counting $\{C_i\}$ also can be done in $O(n+m)$ time, because $\sum_{v \in V} \deg(v) = 2 \cdot m$. The maximum size of C_i is n, so bucket sort takes $O(n)$ time. The effective implementation of finding $\{A_i\}$ in Section 4.3.2 can be done in $O(n+m)$ time, because $O(\sum_{i=0}^{i=k} |C_i|) = O(n+m)$ and we can iterate through the elements of every C_j just updating the maximum index position where element the element appears.

Furthermore, it holds that $A_i, B_i \leq \frac{n(a-1)}{a}$. Therefore, we divide the main problem into smaller ones and such small problems have sizes at most $\frac{n}{c}$ for some $c > 1$. As mentioned earlier, computing $\{A_i\}$, $\{B_i\}$ and $\{C_i\}$ is $O(n+m)$ time. Therefore, we can say that processing of each vertex and each edges takes $O(1)$. But one vertex or edge can be in at most $O(\log n)$ problems, because if the last segment where it was has size x, then the previous one has size at least xc, the previous one has size at least xc^2, etc. The last one has size n.

Checking the cotree consists of counting array *leaves* and m queries. Counting array *leaves* can be computed in $O(n+m)$ time, because we run one depth first search and pre-calculation is $O(n \cdot \log n)$ time. We have m edges, so we have m queries and each query is $O(\log n)$. So, checking the cotree runs in $O((m+n) \cdot \log n)$ time. Therefore, the total time complexity of the algorithm is $O((m+n) \cdot \log n)$.

4.5 Correctness

For every pair of vertices we need to show that if G is a cograph, then their LCA label is correct.

None of the vertices of B_i for every i are connected with v, so it is correct to connect the cotree to (0)-node, because the LCA of v and every vertex of B_i is this (0)-node. The same also holds for A_i and (1)-node.

The LCA of $x \in B_i$ and $y \in B_j$ is the $\max\{i, j\}^{th}$ (0)-node. This is correct because B_i and B_j are the connected components of B, so they do not have any edges between each other.

Since we build the cotrees recursively, so the LCA of a pair of leaves in each recursively computed cotree is correct.

Therefore, it remains to show only that if G is a cograph, then the LCA of $x \in A_i$ and $y \in A_j$ is correct and the LCA of $x \in B_i$ and $y \in A_j$ is correct too.

Theorem 11. *For every i, j such that $0 \leq i < j \leq k + 1$ and every $x \in A_i$ and every $y \in A_j$ it holds that $\{x, y\} \in E$.*

Proof. If for some i, j such that $0 \leq i < j \leq k + 1$ there exists $x \in A_i$ and $y \in A_j$ and $\{x, y\} \notin E$, then for every $z \in B_{i-1}$ and for every $t \in B_{j-1}$ holds $\{z, x\}, \{y, t\}, \{z, y\} \in E$ and $\{x, t\} \notin E$, due to the definition of $\{A_i\}$ and $\{z, t\} \notin E$ due to the definition of $\{B_i\}$. Therefore, $x - z - y - t$ is P_4, the contradiction. Therefore, $\{x, y\} \in E$. $\square$

Thus, the LCA of $x \in A_i$ and $y \in A_j$ where $i < j$, is correct, because it is exactly the j^{th} (1)-node.

Therefore, it remains to show only that if G is a cograph, then the LCA of $x \in B_i$ and $y \in A_j$ is correct too. To prove it we need to introduce the following definition and theorems.

Definition 28. *For every i and for every $x \in B$ let us denote by $NB(B, x)$ the set of neighbours of x, which are not in B.*

Theorem 12. *For every i and for every $x, y \in B_i$ it holds that $NB(B_i, x) = NB(B_i, y) = \Gamma(B_i)$*

Proof. If for some i, j there exist $x \in B_i$ and $y \in B_i$ and $z \in A_j$, such that $\{x, z\}, \{x, y\} \in E$ and $\{y, z\} \notin E$, i.e. there exists a vertex where adjacent vertices x and y are not agreed. Then there exists $P_4 = v - z - x - y$, so G is not a cograph. Therefore, for every i and every pair of x and y from B_i, such that $x, y \in E$, has the same neighbourhood among the neighbours of v. And there do not exist $i \neq j$, a from B_i and b from B_j, such that $\{a, b\} \in E$. So, $NB(B_i, x) = NB(B_i, y)$ if $\{x, y\} \in E$ and x, y are from B_i. This implies that $NB(B_i, x) = NB(B_i, y) = \Gamma(B_i)$ for every pair x, y from B_i, because B_i is connected. $\square$

So, it is correct to look for the neighbours of only one vertex from B_i.

Theorem 13. *For $0 \leq i < j \leq k$ it holds that either $\Gamma(B_i) \subseteq \Gamma(B_j)$ or $\Gamma(B_j) \subseteq \Gamma(B_i)$, if G is a cograph.*

Proof. Assume it is not true, then for $x \in B_i$ and $y \in B_j$ there exist z, t, such that $\{x, z\}, \{y, t\} \in E$ and $\{x, t\}, \{y, z\} \notin E$, but $\{x, y\} \notin E$ and $\{z, t\} \in E$, as mentioned earlier. Consequently, $x - z - t - y$ is P_4 in G, the contradiction. $\square$

Theorem 14. *For every i, j and every $x \in A_i$, $y \in B_j$ it holds that $\{x, y\} \in E$ and $LCA(x, y)$ is (1)-node if and only if $j \geq i$.*

Proof. For every $i = 0, \ldots, k+1$, every $x \in A_i$, every $j = i, \ldots, k$ and every y in B_j, $\{x, y\} \notin E$ due to the definition of A_i. So, the LCA(x, y) is correct, because the i^{th} (0)-node, to which the cotree of A_i is connected is lower than the j^{th} (1)-node, to which the cotree of B_j is connected, so their LCA is the j^{th} (1)-node. The same argument also holds for every $j = 0, \ldots, i-1$. $\square$

4.6 High and low vertices

Definition 29. *A vertex v is* high *if its degree is greater than $\frac{n(a-1)}{a}$. A vertex v is* low *if its degree is smaller than $\frac{n}{a}$.*

We do not have a vertex in G which degree is in the range $[\frac{n}{a}, \frac{n(a-1)}{a}]$, so now all vertices are high or low. Let us define by L the set of all low vertices. It can be proved similarly to the previous case that every component of L forms its subtree and that parents of these subtrees are on one path to root.

Similar to the previous case, we calculate $\Gamma(C_i)$ for each component C_i of L, sort the components by the size of $\Gamma(C_i)$ and compute $D_i = \Gamma(C_i) \backslash \Gamma(C_{i+1})$. The only difference is that if for some j, $|D_j| > \frac{n(a-1)}{a}$, we run another algorithm. We know that there exists at most one such j. Let H be the subgraph induced by D_j . If in $\overline{H}$ there exists a non-high and non-low vertex, then we do not need to change anything, because when the recursion is called on H, it is divided into problems at least $\frac{a}{a-1}$ times smaller. If not, then there are only low vertices in $\overline{H}$, because in D_j there only high vertices and in $\overline{H}$ there are no non-high and non-low vertices, so they are all low. So, we can build for each component of $\overline{H}$ a cotree and reverse it, so we can get a cotree for H.

4.6.1 Correctness

Similar to the previous case, we need to show, that the label of LCA of every pair of vertices is correct.

Theorem 15. *For every i,j such that $0 \leq i < j \leq k$, every $x \in C_i$ and every $y \in C_j$ it holds that $\{x, y\} \notin E$.*

Proof. C_i and C_j are disjoint connected components of L, so x and y cannot be connected. $\square$

Theorem 16. *For every i,j such that $0 \leq i < j \leq k+1$, every $x \in D_i$ and every $y \in D_j$ it holds that $\{x, y\} \notin E$.*

Proof. The proof is exactly the same as in the Theorem 11. □

Theorem 17. *For every i and every $\{a, b\} \in D_i$ such that $\{a, b\} \in E$, $NB(D_i, a) = NB(D_i, b)$.*

Proof. Assume, to the contrary, that there exists a vertex $z \in D_j$ for some j, such that $\{a, z\} \in E$ and $\{b, z\} \notin E$.

There also exists a vertex t, such that $\{t, z\} \in E$ and $\{t, a\}, \{t, b\} \notin E$. If it is not true, than for every t either $\{t, z\} \notin E$, or $\{t, a\} \in E$ or $\{t, b\} \in E$. We know that z is high, so the number of non-neighbours of it is not greater than $\frac{n}{a}$. Since a, b are low, the number of their neighbours does not exceed $\frac{n}{a}$. Therefore, there are at least $\frac{n(a-3)}{a}$ vertices not meeting any of these criteria, i.e. at least $\frac{n(a-3)}{a}$ vertices t such that $\{t, z\} \in E$, and $\{t, a\} \notin E$ and $\{t, b\} \notin E$.

Thus, there is an induced P_4 $b - a - z - t$ - a contradiction. □

Theorem 18. *For every i,j such that $0 \leq i < j \leq k+1$ and every $x \in C_i$ and $y \in D_j$ it holds that $\{x, y\} \notin E$.*

Proof. The proof is exactly the same as in the Theorem 14. □

Theorem 19. *For every i,j such that $0 \leq i < j \leq k$ and every $x \in D_i$ and $y \in C_j$ it holds that $\{x, y\} \notin E$.*

Proof. The proof is exactly the same as in the Theorem 14. □

4.6.2 Complexity

Theorem 20 (Lemma 6 from [10])**.** *Every connected component in the graph induced by low vertices has at most $\leq \frac{2n}{a} + 1$ vertices.*

Proof. Let t be LCA of all vertices from C_i for some i. Clearly, t is (1)-node, because C_i is connected. Assume t has k children denoted as $t_1, ..., t_k$. We know that $k \geq 2$, because C_i is connected. For some descendant leaf $v \in V$ of t_1, it must be connected with all $u \in V$, which are descendant leaves of $t_2, ..., t_k$, because the label of their LCA is 1. But v is low, so its degree is at most $\frac{n}{a}$, so there are no more than $\frac{n}{a}$ vertices in V among descendant leaves of $t_2, ..., t_k$. We do the same for t_2, so the total amount of vertices in the subtree of t, i.e. in C_i, does not exceed $\frac{2n}{a} + 1$. □

Theorem 21. *$\overline{H}$ can be computed in $O(n + m)$ time.*

Proof. H is a set of high vertices, because components of low vertices have sizes at most $\frac{2n}{a} + 1$. In $\overline{H}$ the degree of each vertex is at most $\frac{n}{a}$, because its degree in G is at least $\frac{n(a-1)}{a}$. So, the total number of edges in $\overline{H}$ is bounded from above by $\frac{n \cdot |H|}{a} \leq \frac{(a-2)n \cdot |H|}{a}$. On the other hand, $|H| \geq \frac{(a-1)n}{a}$, so the cardinality of

$G \setminus H$ is bounded by $\frac{n}{a}$, but every $v \in H$ has at least $\frac{(a-1)n}{a} - \frac{n}{a} = \frac{(a-2)n}{a}$ neighbours in H, so the number of edges in H is at least $\frac{(a-2)n \cdot |H|}{a}$.

Therefore, H contains more edges than $\overline{H}$ and it contains at least half of the pairs from $\frac{|H|(|H|-1)}{2}$. So, we can iterate through every pair of vertex and check if they are connected, and they are not connected we add this edge to the complement of graph. $\square$

So, like in the previous case, the problem is divided into the smaller ones with sizes at most $\frac{n}{c}$ for some constant $c > 1$. The time complexity remains the same as in the previous case, i.e. it is equal to $O((n+m) \cdot \log n)$.

Chapter 5

Experiments

All discussed algorithms and the other cograph recognition algorithm [13] were implemented and added to the Koala-NetworKit library [26]. They were implemented in C++ using the NetworKit [1] library and the Koala [26] library.

5.1 Implementation details

Let us show the file structure.

CoTree files:

- `include/recognition/CoTree.hpp` – header file, which defines cotree interface
- `cpp/recognition/cograph/CoTree.cpp` – implementation of cotree interface

Cograph recognition algorithm files:

- `include/recognition/CographRecognition.hpp` – main header file, which defines default cograph recognition and other algorithm interfaces.
- `cpp/recognition/CographRecognition.cpp` – implementation of default cograph recognition algorithms' interfaces.
- `cpp/recognition/cograph/CorneilStewartPerlCographRecognition.cpp` – Corneil, Perl and Stewart algorithm implementation.
- `cpp/recognition/cograph/BretscherCorneilHabibPaulCographRecognition.cpp` – Bretscher, Corneil, Habib, Paul algorithm implemenation.
- `cpp/recognition/cograph/DahlhausCographRecognition.cpp` – Dahlhaus algoritm implementation.
- `benchmark/benchmarkCographs.cpp` – benchmarking programs to evaluate the correctness of every algorithm on different graphs.

- `test/testCographRecognition.cpp` – unit tests for evaluating the correctness of every algorithm.

5.2 Benchmarking and performance evaluation

Our benchmarking suite allows for a robust assessment of algorithm correctness and performance, testing on randomly generated graphs, as well as on all known fixed-size graphs and cographs and comparing the results.

The performance of the algorithms was thoroughly evaluated across a wide range of graph instances. By testing graphs of varying sizes and densities, we assessed the efficiency of each algorithm in different scenarios. Table 5.1 displays the identifier assigned to each algorithm in the tables below.

We choose four random graph models for performance testing:

- Erdős-Renyi model – it provides a wide variety of graph structures, ensuring comprehensive testing across different configurations of node connections. It is a standard model for testing graph algorithms,
- Barabási-Albert model – it produces a network with a few vertices of high degree and many vertices of low degree. If we initiate our algorithm with vertices of low degree, the algorithm can terminate much sooner than if we start with vertices of high degree. For instance, $A1$ is iterative and its performance is significantly influenced by the degrees of the processed vertices. So, this test allows us to obtain the "best case" scenario.
- Watts-Strogatz model – all vertex degrees are identical and all paths are short, providing a more useful test than the previous one, allowing us to obtain the "average" scenario.
- Randomly generated cographs – unlike graphs that are not cographs, the execution of the algorithms on a cograph cannot end earlier with a result "not a cograph". Therefore, this test allows us to maintain the "worst-case" scenario.

All restrictions follow from the capabilities of the algorithms or the capabilities of the generator functions, some of which end with a SIGKILL error or hang due to large input data. Generators work longer than all algorithms. For each data set, only one graph was tested, since generating such large data is very time-consuming.

5.3 Erdős-Renyí model

The Erdős-Renyí model, denoted as $G(n, p)$ is a random graph model used to generate graphs with a given number of nodes n and probability p. Each edge

Number	Algorithm
$A1$	A linear recognition algorithm for cographs by Corneil, Perl, Stewart [9]
$A2$	A simple Linear Time LexBFS Cograph Recognition Algorithm by Bretscher, Corneil, Habib, Paul [6]
$A3$	Efficient parallel recognition algorithms of and distance hereditary graphs by Dahlhaus [10]
$A4$	A simple linear time algorithm for cograph recognition by Habib, Paul [13]

Table 5.1: The algorithms with their identifiers

n	p	m	$A1$	$A2$	$A3$	$A4$
10000	0.1	4997813	897	116567	3643	129
10000	0.3	15001181	2350	926327	4719	436
10000	0.5	24997283	3392	$>$1000000	5654	680
10000	0.7	34997994	4036	$>$1000000	8650	960
10000	0.9	44994097	4998	$>$ 1000000	27986	1122

Table 5.2: The running time for the Erdős-Renyí model(time in ms)

is added to the resulting graph with a probability of p. Every edge addition is independent from the other edges additions.

The Table 5.2 below shows the results for graphs randomly generated by this model.

5.4 Barabási-Albert model

The Barabási-Albert model generates scale-free networks using a preferential attachment mechanism. It starts with a small connected network, forming connections to existing nodes, one node at a time. The probability of connecting to a particular existing node is proportional to its degree, meaning that nodes with higher degrees are more likely to receive new links. The parameters for this model are the number of vertices n and the number of attachments per node k.

The Table 5.3 below shows the results for graphs randomly generated by this model.

n	k	m	$A1$	$A2$	$A3$	$A4$
10000	10	99910	35	112	3	5
10000	100	990100	225	4405	181	23
10000	1000	9001000	1554	269225	3313	240
100000	10	999910	506	1609	46	128
100000	100	9990100	4288	56588	175	795
100000	1000	99001000	37708	4132689	31877	17512

Table 5.3: The running time for the Barabási-Albert model(time in ms)

n	k	p	m	$A1$	$A2$	$A3$	$A4$
10000	40	0.1	400000	41	669	7	10
10000	400	0.1	4000000	468	65170	94	106
10000	4000	0.1	40000000	4637	4301795	21035	958
10000	40	0.7	400000	96	983	8	10
10000	400	0.7	4000000	788	73502	1281	94

Table 5.4: The running time for the Watts-Strogatz model(time in ms)

5.5 Watts-Strogatz model

The Watts-Strogatz model creates small-world networks that exhibit high clustering and short average path lengths. It starts with a regular lattice where each node is connected to its k nearest neighbors. Then, with probability p, each edge is randomly rewired, introducing shortcuts that reduce the path length between nodes while maintaining the high clustering characteristic.

The Table 5.4 below shows the results for graphs randomly generated by this model.

5.6 Random cographs

Random cographs are generated by applying union and complement operations with given probabilities. We generate n trivial cographs. Next, with a given probability p, we choose the complement or union operation. Then we randomly choose a graph or two graphs respectively and do the chosen operation. The results are shown in the Table 5.5.

n	p	m	$A1$	$A2$	$A3$	$A4$
1000	0.1	8508	3	6	5	4
1000	0.3	85975	8	184	24	53
1000	0.5	216736	20	1011	24	584
1000	0.7	372950	37	3219	49	581
1000	0.9	275882	27	1541	61	524
5000	0.1	53793	142	69	33	57
5000	0.3	241943	197	1009	81	256
5000	0.5	5369758	865	116818	1458	24276
5000	0.7	1554688	414	13911	384	3350
5000	0.9	5411969	936	139731	924	26561
10000	0.1	2311377	973	34296	752	8257
10000	0.3	22918654	3765	1418090	10427	306288
10000	0.5	19424286	3352	830930	4376	223273
10000	0.7	14658457	2988	633056	3020	122469
10000	0.9	16429436	3441	682039	2606	196761

Table 5.5: The running time for the random cographs(time in ms)

5.7 Conclusion

As we can see, $A2$ is quadratic time, since the function hasEdge from [1] is linear time. If we get rid of this function in favor of a hash table, then $A2$ will be linear time. $A3$ is $O((n+m)\log n)$, $A1$ and $A4$ are linear time. The results can be explained it the following way. The $A4$ algorithm has a small constant, the other algorithms have significantly larger constants. Additionally, the $A1$ and the $A3$ algorithms tend to terminate before examining every edge and node – they just told us it is not a cograph already, $A3$ has more such posibilities, so in some cases it is definitely the best. The $A2$ always goes through the entire algorithm and never stops before.

Bibliography

[1] Humboldt-Universität zu Berlin - Department of Computer Science - Modeling, Analysis of Complex Systems, and contributors. *NetworKit*. 2023. URL: `https://networkit.github.io/index.html` (visited on 06/2024).

[2] 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`.

[3] 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`.

[4] Hans Bodlaender and Rolf Möhring. "The Pathwidth And Treewidth Of Cographs". In: *SIAM Journal on Discrete Mathematics* 6 (1990), pp. 301–309. URL: `https://api.semanticscholar.org/CorpusID:12667917`.

[5] Béla Bollobás. *Modern Graph Theory*. Springer, 2002. URL: `https://api.semanticscholar.org/CorpusID:120529008`.

[6] Anna Bretscher et al. "A Simple Linear Time LexBFS Cograph Recognition Algorithm". In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Vol. 22. 4. 2008, pp. 1277–1296. URL: `https://api.semanticscholar.org/CorpusID:1348333`.

[7] 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`.

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

[9] Derek Corneil, Yehoshua Perl, and Lauren Stewart. "A Linear Recognition Algorithm For Cographs". In: *Society for Industrial and Applied Mathematics* 14.4 (1985), pp. 926–934.

[10] Elias Dahlhaus. "Efficient Parallel Recognition Algorithms Of Cographs And Distance Hereditary Graphs". In: *Discrete Applied Mathematics* 57 (1995), pp. 29–44. URL: `https://api.semanticscholar.org/CorpusID:15501076`.

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

[12] Emeric Gioan and Christophe Paul. “Split Decomposition And Graph-labelled Trees: Characterizations And Fully-dynamic Algorithms For Totally Decomposable Graphs”. In: *Discrete Applied Mathematics* 160 (2008), pp. 708–733. URL: `https://api.semanticscholar.org/CorpusID:6528410`.

[13] Michel Habib and Christophe Paul. “A Simple Linear Time Algorithm For Cograph Recognition”. In: *Discrete Applied Mathematics* 145 (2005), pp. 183–197. URL: `https://api.semanticscholar.org/CorpusID:3041768`.

[14] Pinar Heggernes and Dieter Kratsch. “Linear-time Certifying Recognition Algorithms And Forbidden Induced Subgraphs”. In: *Nordic Journal of Computing* 14 (2007), pp. 87–108. URL: `https://api.semanticscholar.org/CorpusID:17139705`.

[15] 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.

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

[17] Heinz Jung. “On A Class Of Posets And The Corresponding Comparability Graphs”. In: *Journal of Combinatorial Theory, Series B* 24 (1978), pp. 125–133. URL: `https://api.semanticscholar.org/CorpusID:19941747`.

[18] *LCA computing algorithm.* URL: `https://cp-algorithms.com/graph/lca.html`.

[19] Van Bang Le, Andreas Brandstädt, and Jeremy Spinrad. *Graph Classes: A Survey.* Monographs on Discrete Mathematics and Applications. Society for Industrial Mathematics, 1987. ISBN: 9780898714326.

[20] Rong Lin and Stephan Olariu. “An NC Recognition Algorithm For Cographs”. In: *Journal of Parallel and Distributed Computing* 13 (1991), pp. 76–90. URL: `https://api.semanticscholar.org/CorpusID:36868728`.

[21] 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`.

[22] Stavros Nikolopoulos and Leonidas Palios. “Efficient Parallel Recognition Of Cographs”. In: *Discrete Applied Mathematics* 150 (2005), pp. 182–215. URL: `https://api.semanticscholar.org/CorpusID:933896`.

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

[24] Ron Shamir and Roded Sharan. “A Fully Dynamic Algorithm For Modular Decomposition And Recognition Of Cographs”. In: *Discrete Applied Mathematics* 136 (2004), pp. 329–340. URL: `https://api.semanticscholar.org/CorpusID:7504495`.

[25] David Sumner. “Dacey Graphs”. In: *Journal of the Australian Mathematical Society* 18 (1974), pp. 492–502. URL: `https://api.semanticscholar.org/CorpusID:249898333`.

[26] Krzysztof Turowski et al. *koala-networkit*. 2023. URL: `https://github.com/krzysztof-turowski/koala-networkit` (visited on 06/2024).