Faster and Enhanced Inclusion-Minimal Cograph Completion

Christophe Crespelle[1], Daniel Lokshtanov[2], Thi Ha Duong Phan[3], and Eric Thierry[1]

[1] Univ Lyon, UCB Lyon 1, ENS de Lyon, CNRS, Inria, LIP UMR 5668,
15 parvis René Descartes, F-69342, Lyon, FRANCE
christophe.crespelle@inria.fr, eric.thierry@ens-lyon.fr
[2] University of Bergen, Department of Informatics, N-5020 Bergen, NORWAY
daniello@ii.uib.no
[3] Institute of Mathematics, Vietnam Academy of Science and Technology,
18 Hoang Quoc Viet, Hanoi, Vietnam
phanhaduong@math.ac.vn

Abstract. We design two incremental algorithms for computing an inclusion-minimal completion of an arbitrary graph into a cograph. The first one is able to do so while providing an additional property which is crucial in practice to obtain inclusion-minimal completions using as few edges as possible : it is able to compute a minimum-cardinality completion of the neighbourhood of the new vertex introduced at each incremental step. It runs in $O(n + m')$ time, where m' is the number of edges in the completed graph. This matches the complexity of the algorithm in [24] and positively answers one of their open questions. Our second algorithm improves the complexity of inclusion-minimal completion to $O(n + m \log^2 n)$ when the additional property above is not required. Moreover, we prove that many very sparse graphs, having only $O(n)$ edges, require $\Omega(n^2)$ edges in any of their cograph completions. For these graphs, which include many of those encountered in applications, the improvement we obtain on the complexity scales as $O(n/\log^2 n)$.

1 Introduction

We consider the problem of completion of an arbitrary graph into a *cograph*, i.e. a graph with no induced path on 4 vertices. This is a particular case of *graph modification problem*, in which one wants to perform elementary modifications to an input graph, typically adding and removing edges and vertices, in order to obtain a graph belonging to a given target class of graphs, which satisfies some additional property compared to the input. Ideally, one would like to do so by performing a minimum number of elementary modifications. This is a fundamental problem in graph algorithms, which determines how far is a given graph to satisfy a property.

Here, we consider the modification problem called *completion*, where only one operation is allowed: adding an edge. In this case, the distance between

graphs, called the *cost* of the completion, is the number of edges added, which are called *fill edges*. The particular case of completion problems has been shown very useful incomputer science and other disciplines such as archaeology [22], molecular biology [3] and genomics [14].

Unfortunately, finding the minimum number of edges to be added in a completion problem is NP-hard for most of the target classes of interest (see, e.g., the thesis of Mancini [25] for further discussion and references). To deal with this difficulty of computation, the domain has developed a number of approaches. This includes, approximation, restricted input, parameterizationand inclusion-minimal completions. In the latter approach, one does not ask for a completion having the minimum number of fill edges but only ask for a set of fill edges which is minimal for inclusion, i.e. which does not contain any proper subset of fill edges that also results in a graph in the target class. This is the approach we follow here. In addition to the case of cographs [24], it has been followed for many other graph classes, including chordal graphs [17], interval graphs [10, 26], proper interval graphs [28], split graphs [18], comparability graphs [16] and permutation graphs [9].

The rational behind the inclusion-minimal approach is that minimum-cardinality completions are in particular inclusion-minimal. Therefore, if one is able to sample[4] efficiently the space of inclusion-minimal completions, one can computes several of them, pick the one of minimum cost and hope to get a value close to the minimum one. One of the reason of the success of inclusion-minimal completion algorithms is that this heuristic approach was shown to perform quite well in practice [2]. The second reason of this success, which is a key point for the approach, is that it is usually possible to design algorithms of low complexity for the inclusion-minimal relaxation of completion problems.

Modification problems into the class of cographs have already received a great amount of attention [15, 19, 23, 24], as well as modification problems into some of its subclasses, such as *quasi-threshold graphs* [4] and *threshold graphs* [12]. One reason for this is that cographs are among the most widely studied graph classes, they have been discovered independently in many contexts [6] and they are known to admit very efficient algorithms for problems that are hard in general. Moreover, very recently, cograph modification was shown a powerful approach to solve problems arising in complex networks analysis, e.g. community detection [21] and inference of phylogenomics [19]. The modification problem into the class of quasi-threshold graphs has also been used and it revealed that complex networks encountered in some contexts are actually very close to be quasi-threshold graphs [4], in the sense that only a few modifications are needed to transform them into quasi-threshold graphs. This growing need for treating real-world datasets, whose size is often huge, asks for more efficient algorithms both with regard to the running time and with regard to the quality (number of modifications) of the solution returned by the algorithm.

[4] Usually, minimal completion algorithms are not fully deterministic. There are some choices to be made arbitrarily along the algorithm and different choices lead to different minimal completions.

Our results. Our main contribution is to design two algorithms for inclusion-minimal cograph completion. The first one (Section 4) is an improvement of the incremental algorithm in [24]. It runs in the same $O(n + m')$ complexity, where m' is the number of edges in the completed graph, and is in addition able to select one minimum-cardinality completion of the neighbourhood of the new incoming vertex at each incremental step of the algorithm, which is an open question in [24] (Question 3 in the conclusion) which we positively answer here. It must be clear that this does not guarantee that the completion computed at the end of the algorithm has minimum cardinality (recall that the problem is NP-hard) but this feature is highly desirable in practice to use the inclusion-minimum algorithm as heuristics to obtain a completion using as few fill edges as possible.

When this additional feature is not required, our second algorithm (Section 5)solves the insertion-minimal problem in $O(n + m\,log^2 n)$ time, which only depends on the size of the input and almost linearly. Furthermore, we prove that many sparse graphs, namely those having mean degree fixed to a constant (as most of the graphs encountered in applications), require $\Omega(n^2)$ edges in a minimum-cardinality completion. This result is worth of interest in itself and show that, for such graphs, which have only $O(n)$ edges, the improvement of the complexity we obtain here is quite significant, namely a factor $n/log^2 n$.

2 Preliminaries

All graphs considered here are finite, undirected, simple and loopless. In the following, G is a graph, V (or $V(G)$) is its vertex set and E (or $E(G)$) is its edge set. We use the notation $G = (V, E)$, $n = |V|$ stands for the cardinality of V and $m = |E|$ for the cardinality of E. An edge between vertices x and y will be arbitrarily denoted by xy or yx. The neighbourhood of x is denoted by $N(x)$ (or $N_G(x)$) and for a subset $X \subseteq V$, we define $N(X) = (\bigcup_{x \in X} N(x)) \setminus X$. The subgraph of G induced by some $X \subseteq V$ is denoted by $G[X]$.

For a rooted tree T and a node $u \in T$, we denote $parent(u)$, $\mathcal{C}(u)$, $Anc(u)$ and $Desc(u)$ the *parent* and the set of *children*, *ancestors* and *descendants* of u respectively, using the usual terminology and with u belonging to $Anc(u)$ and $Desc(u)$. The *lowest common ancestor* of two nodes u and v, denoted $lca(u, v)$, is the lowest node in T which is an ancestor of both u and v. The subtree of T rooted at u, denoted by T_u, is the tree induced by node u and all its descendants in T. We use two other notions of subtree, which we call *upper tree* and *extracted tree*. The upper tree of a subset of nodes S of T is the tree, denoted T_S^{up}, induced by the set $Anc(S)$ of all the ancestors of the nodes of S, i.e. $Anc(S) = \bigcup_{s \in S} Anc(s)$. The tree extracted from S in T, denoted T_S^{xtr}, is defined as the tree whose set of nodes is S and whose parent relationship is the transitive reduction of the ancestor relationship in T. More explicitly, for $u, v \in S$, u is the parent of v in T_S^{xtr} iff u is an ancestor of v in T and there exist no node $v' \in S$ such that v' is a strict ancestor of v and a strict descendant of u in T.

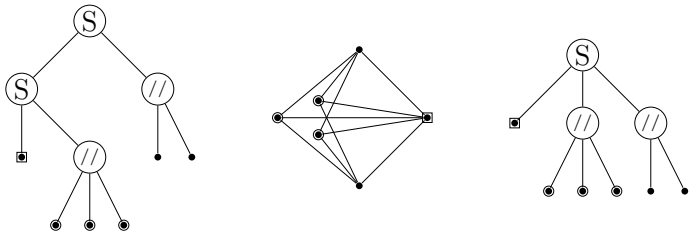

Fig. 1. Example of a labelled construction tree (left), the cograph it represents (centre), and the associated cotree (right). Some vertices are decorated in order to ease the reading.

Cographs. One of their simpler definitions is that they are the graphs that do not admit the P_4 (path on 4 vertices) as induced subgraph. This shows that the class is *hereditary*, i.e., an induced subgraph of a cograph is also a cograph. Equivalently, they are the graphs obtained from a single vertex under the closure of the *parallel* composition and the *series* composition. The parallel composition of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is their disjoint union, i.e., the graph $G_{par} = \big(V_1 \cup V_2, E_1 \cup E_2\big)$. The series composition of G_1 and G_2 is their disjoint union plus all possible edges between vertices of G_1 and vertices of G_2, i.e., the graph $G_{ser} = \big(V_1 \cup V_2, E_1 \cup E_2 \cup \{xy \mid x \in V_1, y \in V_2\}\big)$. These operations can naturally be extended to an arbitrary finite number of graphs.

This gives a nice representation of a cograph G by a tree whose leaves are the vertices of G and whose internal nodes (non-leaf nodes) are labelled $//$, for parallel, or S, for series, corresponding to the operations used in the construction of G. It is always possible to find such a labelled tree T representing G such that every internal node has at least two children, no two parallel nodes are adjacent in T and no two series nodes are adjacent. This tree T is unique [6] and is called the *cotree* of G, see example in Fig. 1. Note that the subtree T_u rooted at some node u of cotree T also defines a cograph, denoted G_u, whose set of vertices is the set of leaves of T_u, denoted $V(u)$ in the following. The adjacencies between vertices of a cograph can easily be read on its cotree, in the following way.

Remark 1 *Two vertices x and y of a cograph G having cotree T are adjacent iff the lowest common ancestor u of leaves x and y in T is a series node. Otherwise, if u is a parallel node, x and y are not adjacent.*

The incremental approach. Our approach for computing a minimal cograph completion of an arbitrary graph G is incremental, in the sense that we consider the vertices of G one by one, in an arbitrary order $(x_1, \ldots, x_n)$, and at step i we compute a minimal cograph completion H_i of $G_i = G[\{x_1, \ldots, x_i\}]$ from a minimal cograph completion H_{i-1} of G_{i-1}, by adding only edges incident to x_i. This is possible thanks to the following observation that is general to all hereditary graph classes that are also stable by addition of a universal vertex, which holds in particular for cographs.

Lemma 1 (see e.g. [26]). *Let $G = (V, E)$ be an arbitrary graph and let H be a minimal cograph completion of G. Consider a new vertex $x \notin V$ adjacent to an arbitrary subset $N(x) \subseteq V$ of vertices and denote $G' = G + x$ and $H' = H + x$ the graphs obtained by adding x to G and H respectively. Then, there exists a subset $M \subseteq V \setminus N(x)$ of vertices such that $H'' = (V, E(H') \cup \{xy \mid y \in M\})$ is a cograph. Moreover, for any such set M which is minimal for inclusion, H'' is an inclusion-minimal cograph completion of G'. We call such completions* (minimal) constrained completions *of $G + x$.*

For any subset $S \subseteq V$ of vertices, we say that we *fill* S in H'' if we make all the vertices of $S \setminus N(x)$ adjacent to x in the completion H'' of $G + x$. The edges added in a completion are called *fill edges* and the *cost* of the completion is its number of fill edges.

The new problem. From now on, we consider the following problem, with slightly modified notations. $G = (V, E)$ is a cograph, and $G + x$ is the graph obtained by adding to G a new vertex x adjacent to some arbitrary subset $N(x)$ of vertices of G. Both our algorithms take as input the cotree of G and the neighbourhood $N(x)$ of the new vertex x. They compute the set $N'(x) \supseteq N(x)$ of neighbours of x in some minimal constrained cograph completion H of $G + x$, i.e. obtained by adding only edges incident to x (cf. Lemma 1). Then, the cotree of G is updated under the insertion of x with neighbourhood $N'(x)$, in order to obtain the cotree of H which will serve as input in the next incremental step.

We now introduce some definitions and characterisations we use in the following.

Definition 1 (Full, hollow, mixed). *Let G be a cograph and let x be a vertex to be inserted in G with neighbourhood $N(x) \subseteq V(G)$. A subset $S \subseteq V(G)$ is* full *if $S \subseteq N(x)$,* hollow *if $S \cap N(x) = \varnothing$ and* mixed *if S is neither full nor hollow. When S is full or hollow, we say that S is* uniform.

We use these notions for nodes u of the cotree as well, referring to their associated set of vertices $V(u)$. We denote $\mathcal{C}_{nh}(u)$ the subset of non-hollow children of a node u.

Theorem 1 below gives a characterisation of the neighbourhood of a new vertex x so that $G + x$ is a cograph.

Theorem 1 ([7, 8]). *(Cf. Fig. 2) Let G be a cograph with cotree T and let x be a vertex to be inserted in G with neighbourhood $N(x) \subseteq V(G)$. If the root of T is mixed, then $G + x$ is a cograph iff there exists a mixed node u of T such that:*

1. *all children of u are uniform and*
2. *for all vertices $y \in V(G) \setminus V(u)$, $y \in N(x)$ iff $lca(y, u)$ is a series node.*

Moreover, when such a node u exists, it is unique and it is called the insertion node.

Remark 2 *In all the rest of the article, we do not consider the case where the new vertex x is adjacent to none of the vertices of G or to all of them. Therefore, the root of the cotree T of G is always mixed wrt. x.*

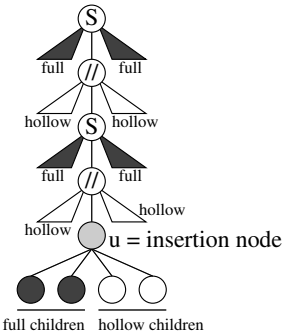

Fig. 2. Illustration of Theorem 1: characterisation of the neighbourhood of a new vertex x so that $G+x$ is a cograph. The nodes and triangles in black (resp. white) correspond to the parts of the tree that are full wrt. x (resp. hollow wrt. x). The insertion node u, which is mixed, appears in grey colour.

The reason for this is that the case where the root is uniform is straightforward: the only minimal completion of $G + x$ adds an empty set of edges and the update of cotree T is very simple. By definition, inserting x in G with its neighbourhood $N'(x)$ in some constrained cograph completion H of $G+x$ results in a cograph, namely H. Therefore, to any such completion H we can associate one insertion node which is uniquely defined, from Theorem 1 and from the restriction stated in Remark 2.

Definition 2. *Let G be a cograph with cotree T and let x be a vertex to be inserted in G. A node u of T is called a* completion-minimal insertion node *iff there exists a minimal constrained completion H of $G + x$ such that u is the insertion node associated to H.*

From now and until the end of the article, G is a cograph, T is its cotree, x is a vertex to be inserted in G and we consider only constrained cograph completions of $G + x$. We therefore omit to precise it systematically.

3 Characterisation of minimal constrained completions

The goal of this section is to give necessary and sufficient conditions for a node u of T to be a completion-minimal insertion node. From Theorem 1, the subtrees attached to the parallel strict ancestors of the insertion node u must be hollow. As we can modify the neighbourhood of x only by adding edges, it follows that if u is the insertion node of some completion, then u is *eligible*, as defined below.

Definition 3 (eligible). *A node u of T is* eligible *iff for all the strict ancestors v of u that are parallel nodes, all the children of v distinct from its unique child $u' \in \mathcal{C}(v) \cap Anc(u)$ are hollow.*

When a node u is eligible, there is a natural way to obtain a completion of the neighbourhood of x, which we call the completion anchored at u.

Definition 4 (Completion anchored at u). *Let u be an eligible node of T. The* completion anchored at *u is the one obtained by making x adjacent to all the vertices of $V(G) \setminus V(u)$ whose lowest common ancestor with u is a series node and by filling all the children of u that are non-hollow.*

The completion anchored at some eligible node u may not be minimal but, on the other hand, all minimal completions H are completions anchored at some eligible node u, namely the insertion node of H.

Lemma 2. *For any completion-minimal insertion node u of T, there exists a unique minimal completion H of $G+x$ such that u is the insertion node associated to H and this unique completion is the completion anchored at u.*

Sketch of proof.(complete proof in Appendix A.1) Consider a minimal completion H that has u as insertion node (there exists one by definition). Note that the set of edges added between x and the vertices of $V(G) \setminus V(u)$ is necessarily the same for all such completions: this is the set given in the definition of the completion anchored at u. Moreover, from Theorem 1, H must make each child of u either full or hollow. Consequently, the children of u that were non-hollow before completion must be full after completion. Since letting the children of u that were hollow before completion remain hollow results in a valid completion, it follows that, by minimality, this is what H does. Thus, H is the completion anchored at u. □

To characterise completion-minimal insertion nodes, we will use the notion of *forced nodes*. Their main property (see Lemma 3 below) is that they are full in any completion of $G + x$.

Definition 5 (Completion-forced). *Let G be a cograph with cotree T and let x be a vertex to be inserted in G. A* completion-forced *(or simply* forced*) node u is inductively defined as a node satisfying at least one of the three following conditions:*

1. *u is full, or*
2. *u is a parallel node with all its children non-hollow, or*
3. *u is a series node with all its children completion-forced.*

Lemma 3. *Let u be a completion-forced node of T. Then, u is filled in all the completions of $G + x$.*

Sketch of proof.(complete proof in Appendix A.2) It can be proven by induction on $|V(u)|$. The only interesting case is when u is parallel. Since all the children of u are non-hollow, it follows that no node of $T_u \setminus \{u\}$ is eligible. Moreover, u itself cannot be the insertion node of some completion since, in this completion, u should have at least one hollow child, which is impossible as all the children of u are already non-hollow before completion. As a consequence, the insertion node v of any completion H is necessarily out of T_u. And since u is not hollow, Theorem 1 implies that u is full in H. □

The next remark directly follows from Theorem 1 and Lemma 2.

Remark 3 *The insertion node u of any minimal completion of $G + x$ has at least one hollow child and at least one non-hollow child. Then, u is non-hollow and non-completion-forced.*

We now characterise the nodes u that contain some minimal-insertion node in their subtree T_u (including u itself). In our algorithms, we will use this characterisation to decide whether we have to explore the subtree of a given node.

Lemma 4. *For any node u of T, T_u contains some completion-minimal insertion node iff u is eligible, non-hollow and non-completion-forced.*

Sketch of proof.(complete proof in Appendix A.3) A minimal insertion node is eligible and, from Remark 3, non-hollow and non-completion forced. So are all its ancestors, proving that the conditions of the Lemma are necessary for T_u to contain a minimal insertion node. To show that they are sufficient, consider a node $v \in T_u$ that satisfies these three conditions and that is lower possible in T_u. By definition, the children of v are non-eligible or hollow or forced. One can show that, whether v is parallel or series, in both cases, v must have at least one hollow child and at least one non-hollow child. Therefore, the completion H' anchored at v leaves v mixed, and so does a minimal completion H included in H'. Then, H also leaves u mixed, which, from Theorem 1, is true only for completions whose insertion node is in T_u. □

Lemma 5 below gives additional conditions for u itself to be an insertion node.

Lemma 5. *A node u of T is a completion-minimal insertion node iff u is eligible, non-hollow and non-completion-forced and u satisfies in addition one of the two following conditions:*

1. *u is a series node and u has at least one hollow child, or*
2. *u is a parallel node and u has no eligible non-completion-forced child.*

Sketch of proof.(complete proof in Appendix A.4) To show that the conditions of the lemma are sufficient, we have to show that the completion H anchored at u is minimal (cf. Lemma 2). Because of Condition 1 when u is series and because u is non-completion forced when u is parallel, in both cases, H leaves some child v of u hollow. Since u is not hollow, the completions H' whose insertion node u' is out of T_u must fill u, and so v. It follows that such completions H' are not strictly included in H. The same holds if $u' = u$, since in this case $H' = H$ from Lemma 2. Then, the only possibility for H' to be strictly included in H is that its insertion node u' is a strict descendant of u. But, in that case, if u is a parallel node, u does not satisfy Condition 2. Consequently, u must be a series node and therefore v is not hollow in H', which shows that H' is not included in H. Thus, H is minimal.

Conversely, if u is a minimal insertion node, then Lemma 4 gives that u is eligible, non-hollow and non-completion-forced. Moreover, from Remark 3, Condition 1 is satisfied. Finally, if u is parallel and does not satisfy Condition 2,

then u has a child v which is eligible non-hollow and non-completion forced. By Lemma 4, T_v contains a completion-minimal insertion node. Then, the completion H' anchored at v is included in the completion anchored at u, which implies that u is not a completion-minimal insertion node. By contraposition, if u is a parallel node, then u satisfies Condition 2. □

4 An $O(n + m')$ algorithm with incremental minimum

In this section (see a more detailed version in Appendix B), we design an incremental algorithm whose overall time complexity is $O(n + m')$, where m' is the number of edges in the output completed cograph. We concentrate on one incremental step, whose input is the cotree T of some cograph G (the completion computed so far) and a new vertex x together with the list of its neighbours $N(x) \subseteq V(G)$. Each node $u \in T$ stores its number $|\mathcal{C}(u)|$ of children and the number $|V(u)|$ of leaves in T_u. One incremental step takes time $O(d')$, where d' is the degree of x in the completion of $G + x$ computed by the algorithm. Within this complexity, our algorithm scans all the minimal completions of the neighbourhood of x and select one of minimum cardinality. Our description is in two steps.

First step. For each non-hollow node u of T we determine: i) the list of its non-hollow children $\mathcal{C}_{nh}(u)$, ii) the number of neighbours of x in $V(u)$ and iii) whether it is completion forced or not. To this purpose, we perform two bottom-up searches of T from the leaves of T that are in $N(x)$ up until the root of T. Note that each of these searches discovers exactly the set $\mathcal{NH}(T)$ of non-hollow nodes of T (for which we show later that their number is $O(d')$). In the first search, we label each node encountered as non-hollow, we build the list of its non-hollow children and count them. In the second search, for each non-hollow node u we determine the rest of its information, that is items ii) and iii) above.

For the leaves of T in $N(x)$, it is straightforward to get this information. Then, the bottom-up search starts in an asynchronous manner: as soon as a node determines its information, it forwards its to its parent. When a node has received the information from all its non-hollow children (we determined their number in the first search), it can easily determine its own information and the process goes on, until the root of T has determined its information.

Second step. We search the set of all non-hollow, eligible and non-completion-forced nodes of T. For each of them, we determine whether it is a minimal insertion node and, in the positive, we compute the number of edges to be added in its associated minimal completion. Then, at the end of the search we select the completion of minimum cardinality.

Since, all the ancestors of a non-hollow eligible non-completion-forced node also satisfy these three properties, it follows that the part of T we have to search is a connected subset of nodes containing the root. Then, our search starts by determining whether the root is non-completion-forced. In the negative, we are done: there exists one unique minimal completion of $G + x$ which is obtained by

adding all missing edges between x and the vertices of G. Otherwise, if the root is non-completion-forced (it is always eligible and non-hollow, cf. Remark 2), we start our search. For all the non-hollow children of the current node (we built their list in the first step), we check whether they are eligible and non-completion-forced and search, in a depth-first manner, the subtrees of those for which the test is positive (cf. Lemma 4).

During this depth-first search, we compute for each node u encountered the number of edges, denoted $cost-above(u)$, to be added between x and the vertices of $V(G) \setminus V(u)$ in the completion anchored at u. It can be determined as follows: if the parent v of u is a parallel node, then $cost - above(u) = cost - above(v)$; otherwise, if the parent v of u is a series node, then $cost - above(u) = cost - above(v) + |V(v) \setminus N(x)| - |V(u) \setminus N(x)|$. We also determine whether u is a minimal insertion node by testing whether it satisfies Condition 1 or 2 of Lemma 5. Importantly for the complexity, this can be done by scanning only the list of its non-hollow children, and by using the information collected in the first step. If u is a minimal insertion node, then we determine the number of edges $cost(u)$ to be added in the completion anchored at u as $cost(u) = cost - above(u) + \sum_{v \in \mathcal{C}_{nh}(u)} |V(v) \setminus N(x)|$.

From Lemma 5, minimal insertion nodes are non-hollow, eligible and non-completion-forced. Therefore, our search discovers all of them and returns one that achieves the minimum cost among all completions of the neighbourhood of x. Finally, we need to update the cotree T for the next incremental step of the algorithm, as explained below.

Complexity. The key of the $O(d')$ time complexity is that we search and manipulate only the set $\mathcal{NH}(T)$ of non-hollow nodes of T. For each of them u, we need to scan the list of its non-hollow children $\mathcal{C}_{nh}(u)$ and to perform a constant number of tests and operations that all take $O(1)$ time (thanks to the information collected in the first step). Thus, the execution of the two steps takes $O(|\mathcal{NH}(T)|)$ time, which is also $O(d')$ as shown in [24]. Indeed, one can observe that during completion, all non-hollow nodes are filled (see Definition 4), except the ancestors of the insertion node u, but their number is also $O(d')$.

When, the insertion node u has been determined, the completed neighbourhood $N'(x)$ of x can be computed in extension by a search of the part of T that is filled, which takes $O(d')$ time. Then, the cotree of the completion H of $G+x$ is obtained from the cotree of G (as depicted in Figure 3) in the same time complexity thanks to the algorithm of [7]. Overall, one incremental step takes $O(d')$ time and the whole running time of the algorithm is $O(n + m')$.

5 An $O(n + m \log^2 n)$ algorithm

Even though it is linear in the number of edges in the output cograph, the $O(n+m')$ complexity achieved by the algorithm in [24] and the one we presented in Section 4 is not necessarily optimal, as the output cograph can actually be represented in $O(n)$ space using its cotree. We then design a refined version of the inclusion-minimal completion algorithm that runs in $O(n + m \log^2 n)$ time,

when no additional condition is required on the completion output at each incremental step. This improvement is further motivated by the fact that, as we show below, there exist graphs having only $O(n)$ edges and which require $\Omega(n^2)$ edges in any of their cograph completions. For such graphs, the new complexity we achieve also writes $O(n \log^2 n)$ (since $m = O(n)$) and constitutes a significant improvement over the $O(n^2)$ complexity of the previous algorithm (since $m' = \Omega(n^2)$).

Worst-case minimum-cardinality completion of very sparse graphs. Our proof is based on vertex expander graphs (see [20] for a survey on the topic), which require $\Omega(n^2)$ edges in any of their cograph completions, as stated by Theorem 2 below.

Definition 6. *A graph G is a* c-expander *if, for every vertex subset $S \subseteq V(G)$ with $|S| \leq \frac{|V(G)|}{2}$ we have $|N(S)| \geq c \cdot |S|$.*

Theorem 2. *Let $c > 0$ be a real number and G a c-expander. For any cograph completion H of G, $|E(H)| \geq \Omega(c^2.n^2)$.*

Sketch of proof.(complete proof in Appendix C) Cographs are known to be also distance hereditary graphs, and therefore totally decomposable by split decomposition, or equivalently of rank-width 1. It implies (for example from a result of [27]) that H contains a split $(S, V \setminus S)$ (meaning that the graph induced by the edges crossing the bipartition $(S, V \setminus S)$ is a complete bipartite graph) such that $n/3 \leq |S| \leq n/2$.

Since G is a c-expander, then $|N(S)| \geq c.|S|$. One can show that the expansion property of G also implies that $|N(V \setminus S)| \geq c.|S|/3$. Consequently, in H, the complete bipartite graph induced by the edges crossing the split $(S, V \setminus S)$ has at least $c.|S|/3$ vertices in S and at least $c.|S|$ in $V \setminus S$. Thus, it contains at least $c^2.|S|^2/3 \geq c^2.n^2/27$ edges. □

There exist deterministic constructions of very sparse graphs that are c-expanders, see for example the construction of 3-regular c-expanders by Alon and Boppana [1], for some fixed c. Such graphs have only $O(n)$ edges but, from Theorem 2, require $\Omega(n^2)$ edges in any of their cograph completions. More generally, it is part of the folklore that, for any constant $a > 1$, there exist $c > 0$ and $p > 0$ such that, for any $n \in \mathbb{N}$ sufficiently large, the proportion of graphs on n vertices and $a.n$ edges that are c-expanders is at least p. This means that many graphs of fixed mean degree have the vertex expansion property and therefore require $\Omega(n^2)$ edges in any cograph completion. Motivated by this frequent worst-case for the $O(n + m')$ complexity, we now describe an $O(n + m \log^2 n)$-time algorithm for inclusion-minimal cograph completion of arbitrary graphs (see Appendix D for a more detailed description).

Data structure. We store two distinct copies of the cotree T of G. The first one is a basic data structure in which each node u stores its number of children $|\mathcal{C}(u)|$ and a bidirectional couple of pointers to the corresponding node in the second copy of T, so that we can move from one copy to the other one in $O(1)$

time. In addition, in the first copy of T, each node u stores a copy of the list of its children using the *order data structure* of [11]. It allows to determine which of two given children of u precedes the other one in the list, in $O(1)$ time. It also supports two update operations, `delete` and `insert`, that respectively remove and insert an element in the list (just after a specified element), in $O(1)$ time as well.

The second copy of T is stored using the dynamic data structure developed in [29]. This data structure allows to answer two kinds of query: `lowest-common-Ancestor?`, which provides $lca(u, v)$ for two given nodes u, v, and `next-step-to-descendant?`, which given a node u of T and one of its strict descendants v, provides the child of u which is an ancestor of v. These two queries are treated in $O(\log n)$ worst-case time. To be precise, the latter query is obtained as a combination of three other basic operations provided by [29] that we do not use here, namely `root?`, `evert` and `parent?`. This data structure is dynamic, meaning that it supports update operations on the structure of the tree (which is actually a forest, as it is allowed to be disconnected). Operation `cut` removes the edge between one given node and its parent and `link` makes the root of one tree of the forest become the child of a given node in another tree. These update operations also have $O(\log n)$ worst-case time complexity.

Algorithm. Our algorithm determines the set W of the eligible non-hollow non-completion-forced nodes that are minimal for the ancestor relationship (i.e. none of their descendants satisfies the considered property), and arbitrarily picks one of them to be the insertion node of the minimal completion returned at this incremental step. Indeed, since nodes of W satisfy the conditions of Lemma 4 and none of their children does, it follows that nodes of W are completion-minimal insertion nodes. In order to get the improved $O(n + m \log^2 n)$ complexity, we avoid to completely search the upper tree $T^{up}_{N(x)}$ to determine W. Instead, we use a limited number of `lowest-common-Ancestor?` queries.

Clearly, if a parallel node u of T is the lca of two leaves in $N(x)$ then $T_u \setminus \{u\}$ contains no eligible node. Let P_{max} be the set of parallel common ancestors of vertices of $N(x)$ that are maximal for the ancestor relationship and let us denote $W' = P_{max} \cup N_{out}$, where N_{out} is the set of vertices of $N(x)$ that are not descendant of any node of P_{max}, i.e. $N_{out} = N(x) \setminus \bigcup_{p \in P_{max}} V(p)$. Note that all the nodes $w' \in W'$ are eligible, and so are their ancestors. It follows that the set W we aim at determining is the set of the lowest non-completion-forced nodes in the upper tree $T^{up}_{W'}$.

In order to compute the set W, we start by computing the tree $\widetilde{T} = T^{xtr}_{N(x) \cup A_x}$ extracted from (see Section 2) the leaves that belong to $N(x)$ and the set A_x of their lowest common ancestors, i.e. nodes u such that $u = lca(l_1, l_2)$ for some leaves $l_1, l_2 \in N(x)$. Then, we search $\widetilde{T}$ to find its parallel nodes P_{max} that are maximal for the ancestor relationship and thereby obtain the set W'. Finally, for each node $w' \in W'$ we determine its lowest non-completion-forced ancestor $nfa(w')$ in T and we keep only the $nfa(w')$'s that are minimal for the ancestor relationship: this is the set W. It is worth noting from the beginning that since

$\widetilde{T}$ has exactly d leaves and since all its internal nodes have degree at least 2, then the size of $\widetilde{T}$ is $O(d)$.

We now show how to compute $\widetilde{T}$ in $O(d \log^2 n)$ time. To this purpose, we sort the neighbours of x according to a special order of the vertices of the cograph G called a *factorising permutation* [5], which is the order π in which the vertices of G (which are the leaves of T) are encountered when performing a depth-first search of T. One can determine whether a vertex y_1 is before or after a vertex y_2 in π as follows: 1) find $u = lca(y_1, y_2)$ and the two children u_1 and u_2 of u that are respectively the ancestor of y_1 and y_2, and 2) determine whether u_1 is before or after u_2 in the list of children of u. Operation 1) can be implemented by one `lowest-common-ancestor?` query and two `next-step-to-descendant?` queries, which takes $O(\log n)$ time. Operation 2) can be executed in $O(1)$ time using the order data structure of [11]. Therefore, since one comparison takes $O(\log n)$ time, the neighbours of x can be sorted according to π in $O(d \log d \log n) = O(d \log^2 n)$ time.

The benefit of doing so is that $\widetilde{T}$ can be built efficiently by considering the neighbours of x one by one in the order $x_1, x_2, \ldots, x_d$ in which they appear in π (we say from left to right). At each step, we build the tree T_i extracted from $\{x_1, \ldots, x_i\}$ and their lowest common ancestors, then, at the end $T_d = \widetilde{T}$. When inserting x_{i+1} in T_i, at most one new internal node v_{i+1} is created in T_{i+1}. Because we consider the x_i's in the order they appear in π, we have necessarily $v_{i+1} = lca_T(x_i, x_{i+1})$ and v_{i+1} must be inserted in the rightmost branch of T_i, or it already appears in this branch if $v_{i+1} \in T_i$. Consequently, we climb up the rightmost branch of T_i, starting from the father of x_i, and for each node v encountered we determine whether v_{i+1} is above v by computing $lca(v, v_{i+1})$. The total number of lca queries needed to build the whole tree $\widetilde{T}$ is proportional to its size, since every time we pass above a node v on the rightmost branch, v leaves this branch for ever and will then never participate again to any lca query (cf. [13]). Since the size of $\widetilde{T}$ is $O(d)$ and each query takes $O(\log n)$ time, building $\widetilde{T}$ from the sorted list of neighbours of x takes $O(d \log n)$ time.

Once $\widetilde{T}$ is built, a simple search starting from its root determines the set P_{max} of its parallel nodes that are maximal for the ancestor relationship, and we cut off from $\widetilde{T}$ all the subtrees rooted at the children of nodes in P_{max}. The leaves of the resulting tree are precisely the nodes of W'. As $\widetilde{T}$ has size $O(d)$, this step takes $O(d)$ time. Then, for each $w' \in W'$, we determine its lowest non-completion-forced ancestor $nfa(w')$ in T. From the definition of P_{max}, the lowest parallel ancestor of w' is non-completion-forced. Then, $nfa(w')$ cannot be higher in T than the grand-parent of w'. It follows that we have to check the non-completion-forced condition only for w' and its parent, which can be done, for each of them u, in $O(|\mathcal{C}_{nh}(u)|)$ time. Then, we remove the $nfa(w')$'s that are not minimal for the ancestor relationship to obtain the set W, this takes $O(d)$ time, and we arbitrarily pick one node w in W. The minimal completion of the neighbourhood of x returned is the one anchored at w and the total complexity of one incremental step is $O(d + d \log n + d \log^2 n) = O(d \log^2 n)$.

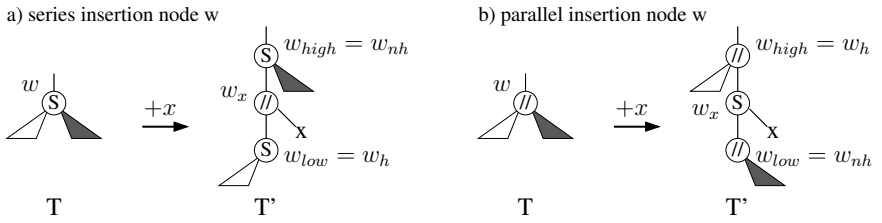

Fig. 3. Modification of the cotree under the insertion of x at insertion node w. The triangles in black (resp. white) correspond to the parts of the tree that are filled (resp. that remain hollow) in the completion anchored at w.

Updating the data structure. After the insertion node w has been determined, the cotree T must be modified as shown in Figure 3, and the data structure of [29] must be updated accordingly. The key for preserving the complexity is to perform operations involving only the non-hollow children of w. After the insertion of x, w is split into two new nodes w_h and w_{nh} in T', which are parent of respectively the hollow children of w and the non-hollow children of w. To form these two nodes, we cut from w its non-hollow children to obtain w_h, still linked to the hollow children, and we link the non-hollow children to a new node w_{nh}. This takes $O(d \log n)$ as it requires $O(d)$ `cut` and `link` operations. The rest of the operations to build T' are less sensitive and, for lack of space, we do not describe them.

As a conclusion, the complexity of one incremental step of the algorithm is $O(d \log^2 n)$ and overall, the complexity of the whole algorithm is $O(n+m \log^2 n)$.

References

1. Alon, N.: Eigenvalues and expanders. Combinatorica 6(2), 83–96 (1986)
2. Berry, A., Heggernes, P., Simonet, G.: The minimum degree heuristic and the minimal triangulation process. In: 29th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2003). LNCS, vol. 2880, pp. 58–70. Springer (2003)
3. Bodlaender, H., Downey, R., Fellows, M., Hallett, M., Wareham, H.: Parameterized complexity analysis in computational biology. Comput. Appl. Biosci. 11, 49–57 (1995)
4. Brandes, U., Hamann, M., Strasser, B., Wagner, D.: Fast quasi-threshold editing. In: 23rd European Symposium on Algorithms (ESA 2015). LNCS, vol. 9294, pp. 251–262. Springer (2015)
5. Capelle, C., Habib, M., de Montgolfier, F.: Graph decompositions and factorizing permutations. Discrete Mathematics and Theoretical Computer Science 5(1), 55–70 (2002)
6. Corneil, D., Lerchs, H., Burlingham, L.S.: Complement reducible graphs. Discrete Applied Mathematics 3(3), 163–174 (1981)
7. Corneil, D., Perl, Y., Stewart, L.: A linear time recognition algorithm for cographs. SIAM Journal on Computing 14(4), 926–934 (1985)

8. Crespelle, C., Paul, C.: Fully dynamic recognition algorithm and certificate for directed cographs. Discrete Applied Mathematics 154(12), 1722–1741 (2006)
9. Crespelle, C., Perez, A., Todinca, I.: An $O(n^2)$-time algorithm for the minimal permutation completion problem. In: 41st International Workshop on Graph-Theoretic Concepts in Computer Science (WG 2015). LNCS, Springer (2015), to appear
10. Crespelle, C., Todinca, I.: An $O(n^2)$-time algorithm for the minimal interval completion problem. Theor. Comput. Sci. 494, 75–85 (2013)
11. Dietz, P., Sleator, D.: Two algorithms for maintaining order in a list. In: 19th ACM Symposium on Theory of Computing (STOC 1987). pp. 365–372. ACM (1987)
12. Drange, P.G., Dregi, M.S., Lokshtanov, D., Sullivan, B.D.: On the threshold of intractability. In: 23rd European Symposium on Algorithms (ESA 2015). LNCS, vol. 9294, pp. 411–423. Springer (2015)
13. Gabow, H., Bentley, J., Tarjan, R.: Scaling and related techniques for geometry problems. In: 16th ACM Symposium on Theory of Computing (STOC 1984). pp. 135–143. ACM (1984)
14. Goldberg, P., Golumbic, M., Kaplan, H., Shamir, R.: Four strikes against physical mapping of DNA. J. Comput. Biol. 2, 139–152 (1995)
15. Guillemot, S., Havet, F., Paul, C., Perez, A.: On the (non-)existence of polynomial kernels for P_l-free edge modification problems. Algorithmica 65(4), 900–926 (2012)
16. Heggernes, P., Mancini, F., Papadopoulos, C.: Minimal comparability completions of arbitrary graphs. Discrete Applied Mathematics 156(5), 705–718 (2008)
17. Heggernes, P., Telle, J.A., Villanger, Y.: Computing minimal triangulations in time $O(n^{\alpha \log n}) = o(n^{2.376})$. SIAM J. Discrete Math. 19(4), 900–913 (2005)
18. Heggernes, P., Mancini, F.: Minimal split completions. Discrete Applied Mathematics 157(12), 2659–2669 (2009)
19. Hellmuth, M., Wieseke, N., Lechner, M., Lenhof, H.P., Middendorf, M., Stadler, P.F.: Phylogenomics with paralogs. PNAS 112(7), 2058–2063 (2015)
20. Hoory, S., Linial, N., Wigderson, A.: Expander graphs and their applications. Bulletin of the American Mathematical Society 43(4), 439–561 (2006)
21. Jia, S., Gao, L., Gao, Y., Nastos, J., Wang, Y., Zhang, X., Wang, H.: Defining and identifying cograph communities in complex networks. New Journal of Physics 17(1), 013044 (2015)
22. Kendall, D.: Incidence matrices, interval graphs, and seriation in archeology. Pacific J. Math. 28, 565–570 (1969)
23. Liu, Y., Wang, J., Guo, J., Chen, J.: Complexity and parameterized algorithms for cograph editing. Theoretical Computer Science 461, 45–54 (2012)
24. Lokshtanov, D., Mancini, F., Papadopoulos, C.: Characterizing and computing minimal cograph completions. Discrete Appl. Math. 158(7), 755–764 (2010)
25. Mancini, F.: Graph Modification Problems Related to Graph Classes. Ph.D. thesis, University of Bergen, Norway (2008)
26. Ohtsuki, T., Mori, H., Kashiwabara, T., Fujisawa, T.: On minimal augmentation of a graph to obtain an interval graph. Journal of Computer and System Sciences 22(1), 60–97 (1981)
27. Oum, S., Seymour, P.D.: Testing branch-width. J. Comb. Theory, Ser. B 97(3), 385–393 (2007)
28. Rapaport, I., Suchan, K., Todinca, I.: Minimal proper interval completions. Inf. Process. Lett. 106(5), 195–202 (2008)
29. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. 26(3), 362–391 (1983)

A Characterisation of minimal constrained completions

A.1 Completion anchored at u

Proof of Lemma 2: First, note that the modified neighbourhood $N'_{\bar{u}}(x)$ of x in $V(G) \setminus V(u)$ is given by Theorem 1 and is the same for every completion having u as insertion node. Moreover, as in any such completion, the children of u in T are uniform, then any non-hollow child v of u must be filled. Then, the completion H_{min} defined by the modified neighbourhood $N'_{min}(x) = N'_{\bar{u}}(x) \cup \bigcup_{v \in \mathcal{C}(u) \text{ and } v \text{ is non-hollow}} V(v)$ of x is included in every completion having u as insertion node. As there exists some minimal completion having u as insertion node, then from Theorem 1, u is left mixed after completion and so u has some hollow child with regard to $N(x)$. Consequently, u is also mixed with regard to $N'_{min}(x)$. Finally, since the insertion of x with neighbourhood $N'_{min}(x)$ satisfies conditions 1 and 2 of Theorem 1, then the completion H'_{min} has u as insertion node. And since H'_{min} is included in all such completions, it follows that H'_{min} is the unique minimal completion having u as insertion node. □

A.2 Completion-forced nodes

Lemma 3 is a direct consequence of the following stronger statement.

Lemma 6. *Let G be a cograph with cotree T and let x be a vertex to be inserted in G. A node u of T is completion-forced iff there exists a unique cograph completion of $G_u + x$, which is the one where all missing edges between x and $V(u)$ are added.*

Proof. Let us show the result by induction on $|V(u)|$. First, consider a completion-forced node u of T and a completion H of $G_u + x$. If u satisfies Condition 3 of Definition 5, then, by induction hypothesis, all its children are full in H (as H is also a cograph completion of $G_v + x$, for any child v of u) and so is u. If u satisfies Condition 1, then since u is full before completion, it is also full after. Consider now the case where u is completion-forced because it satisfies Condition 2 of Definition 5, i.e. u is parallel and all its children are non-hollow.

Assume for contradiction that H does not fill u. Then, denote u' the insertion node associated to H in T_u. Theorem 1 implies that u' is eligible, and since all the children of u are non hollow, it follows that u' is not a strict descendant of u. Consequently, $u' = u$ and since all the children of u are non hollow, Lemma 2 implies that H fills all of them, and so H fills u as well: contradiction. Thus, u is filled in any completion H of $G_u + x$ and therefore, there exists a unique such completion.

Conversely, consider a non-completion-forced node u of T. If u is a series node, then u has at least one non-completion-forced child v. By induction hypothesis, there exists a completion H' of $G_v + x$ that does not fill v. Then, the completion H of $G_u + x$ that coincides with H' on $V(v)$ and that fills all the other children of u is a cograph completion of $G_u + x$ that does not fill u.

Now, if u is a parallel node, then u has at least one hollow child v. As u is clearly eligible in T_u, the cograph completion H anchored at u is properly defined. Since H leaves v hollow, then H does not fill u, which achieves the proof. □

A.3 Nodes whose subtree contains a minimal insertion node

Proof of Lemma 4: If u is eligible non-hollow and non-completion-forced, consider such a node v of T_u which is lower possible in T_u. If v is a series node, as v is eligible so are all its children. It follows that all the children of v are either completion-forced or hollow. Since v is non-completion-forced, at least one of its children is hollow and since v is non-hollow at least one of its children is non-hollow. The same holds if v is a parallel node: since v is non-completion-forced, at least one of its children is hollow and since v is non-hollow at least one of its children is non-hollow. Then, in both cases, in the completion H' anchored at v, v is mixed and so is u. Consequently, there exists a minimal completion H included in H' and necessarily u is mixed in H as well. From Theorem 1, it is straightforward to see that all minimal completions having an insertion node out of T_u leaves u full or hollow. It follows that the insertion node associated to H belongs to T_u.

Now, conversely, if there exists $v \in T_u$ which is a completion-minimal insertion node, let us denote H the minimal completion anchored at v. From Remark 3, v is non-hollow in $G+x$, and so is u. Moreover, from Theorem 1, it is straightforward to see that v is eligible and so is u. From Theorem 1 again, v is mixed in H and so is u. Then, Lemma 3 implies that u is non-completion-forced, which achieves the proof of the lemma. □

A.4 Characterisation of minimal insertion nodes

Proof of Lemma 5: We first show that if the conditions of the lemma are satisfied, then u is a completion-minimal insertion node. From Lemma 2, if u is a completion-minimal insertion node, then there exists a unique minimal completion H such that u is the insertion node associated to this completion. From Lemma 2 again, this completion H is the completion anchored at u, which is properly defined here as u is eligible, see Definition 4. We will now show that H is minimal.

If u is a parallel node, as u is non-completion-forced, u has at least one hollow child v, and the same holds if u is a series node because of Condition 1. From Definition 4, v is hollow in H. Let H' be a minimal completion of $G+x$ and let u' be its insertion node u'. We will show that H' is not strictly included in H. From Lemma 2, if $u' = u$, then $H' = H$ and therefore, from now, we consider only the case where $u' \neq u$. Note that, from Theorem 1, the only nodes of T that remain mixed after completion into H' are the ancestors of u'. All the non-hollow nodes of T that are not ancestors of u' are filled in H'. Then, if u' is not

a descendant of u, node u is filled in H' and so is node v. It follows that, if u' is not a descendant of u, H' is not included in H.

Now, consider the case where u' is a strict descendant of u (remember that $u' \neq u$) and suppose for contradiction that u is a parallel node. Lemma 4 implies that u' is eligible. Since u' is a strict descendant of u, then all the children of u, except its child w that is an ancestor of u', are hollow. Then, from Condition 2 of the present lemma, it follows that w must be completion-forced. Lemma 3 implies that w, and so u', is filled in H'. This contradicts the fact that u' is the insertion node, as from Theorem 1, this node remains mixed after completion. Thus, u is not a parallel node, but a series node. From Remark 3, u' is non-hollow in $G + x$ and consequently, u' is not a descendant of v (the hollow child of u). Since u is a series node, it follows that v is filled in H', which is therefore not included in H. This achieves the proof that the conditions of the lemma are sufficient.

Let us now show that they are necessary. Consider a completion-minimal node u and let us show that it satisfies the conditions of the lemma. Firstly, because T_u contains some completion-minimal insertion node, namely u, Lemma 4 implies that u is mixed, eligible and non-completion-forced. Let H be the completion anchored at u. From Theorem 1, u is mixed in H. Then, from Lemma 2, it follows that u has at least one hollow child. Condition 1 is satisfied.

We now show that if u is parallel and does not satisfy Condition 2, then the completion H anchored at u is not minimal, which implies that u is not a completion-minimal insertion node. Since u is mixed, it has at least one non-hollow child v. Moreover, since u does not satisfy Condition 2, v is the unique non-hollow child of u (then v is eligible) and v is non-completion-forced. As v is eligible, non-hollow and non-completion-forced, it follows from Lemma 4 that T_v contains some completion-minimal insertion node. The corresponding minimal completion H' is included in H and even strictly included as H' leaves v mixed, while H fills it (since v is not hollow). Thus, H is not minimal. By contraposition, if H is minimal, Condition 2 is satisfied. This achieves the proof of the lemma. □

B An $O(n + m')$ algorithm

In this section, we design an algorithm that runs in linear time in the size of the output graph (encoded by adjacency lists), that is $O(n + m')$ time, by denoting m' the number of edges in the cograph output by the algorithm. The input of one incremental step of our algorithm is the cotree T of G and the new vertex x together with the list of its neighbours $N(x)$. Each node $u \in T$ stores its number of children and the number of leaves in T_u. Our algorithm works in two steps.

First step: collecting information on nodes of T. We compute some basic information on the nodes of T with regard to $N(x)$. To this purpose, we perform two bottom-up searches of T from the leaves of T that are in $N(x)$ up until the root of T. Consequently, each of these searches discovers exactly the non-hollow nodes of T.

In the first search, we simply label all visited nodes as non-hollow and for each of these nodes, we count their number of non-hollow children. The nodes that are not visited, and therefore not labelled are exactly the hollow nodes of T.

In the second bottom-up search, for each non-hollow node u, we determine:

- the number of neighbours of x in $V(u)$ and
- whether u is completion-forced.

It is straightforward to get this information for the leaves l of T that belong to $N(x)$: there is exactly one neighbour of x in $V(l)$ and l is forced. Then, all the leaves in $N(x)$ forward their information to their parents in an asynchronous way. Along this process, each non-hollow node u of T is able to know whether it has received the information from all its non-hollow children, as we determined their number in the first search. When it happens, when u has received the information from all its non-hollow children, u is able to determine its own information: u makes the sum of $|V(v) \cap N(x)|$ for all its non-hollow children v, and u determines whether it is completion-forced as follows. If u is parallel, then u is completion-forced iff all its children are non-hollow, and if u is series, then u is completion-forced iff all its children are completion-forced. Then, u forwards its information to its parent and the process goes on until the root of the tree itself has determined its information. At that time, the process ends as all the non-hollow nodes of T have already determined their information.

Second step: finding all completion-minimal insertion nodes of T. Starting from the root of T, we search one upper tree of T that contains all the completion-minimal insertion nodes. For each completion-minimal insertion node encountered during this search, we determine the number of edges to be added in its associated minimal completion. Then, at the end of the search we can select the completion of minimum cardinality encountered.

From Lemma 4, it is necessary and sufficient to search all the nodes of T that are non-hollow, eligible and non-completion-forced. Moreover, note that all the ancestors of such nodes are themselves non-hollow, eligible and non-completion-forced. Then our algorithm proceeds as follows.

It starts by determining whether the root of T is non-completion-forced (the root is always eligible, by definition, and non-hollow from Remark 2). If this is not the case, then there exists one unique minimal completion of $G + x$ which is obtained by making x a universal vertex.

Otherwise, if the root is non-completion-forced (and eligible and non-hollow, necessarily), then we search T, starting from the root. For all the non-hollow children of the current node, we check whether they are eligible and non-completion-forced and search, in a depth-first manner, the subtrees of those for which the test is positive. During this depth-first search, we compute for each node u encountered the number of edges, denoted $cost - above(u)$, to be added between x and the vertices of $V(G) \setminus V(u)$ in the completion anchored at u. This can be computed during the search as follows:

– if the parent v of u is a parallel node (necessarily eligible, since we parse only eligible nodes), then $cost - above(u) = cost - above(v)$; and
– if the parent v of u is a series node, then $cost - above(u) = cost - above(v) + \sum_{u' \in \mathcal{C}(v), u' \neq u} |V(u') \setminus N(x)|$.

In addition, for each node u encountered during the search (which is non-hollow, eligible and non-completion-forced), we test whether it satisfies the conditions of Lemma 5, i.e. whether u is a minimal insertion node. This can be done thanks to the information collected in the first step. For complexity reasons, note that Condition 2 of Lemma 5 can be tested by scanning only the non-hollow children of u. In the positive, if u is a minimal insertion node, then we determine the number of edges, denoted $cost(u)$, to be added in the completion anchored at u, as $cost(u) = cost - above(u) + \sum_{v \in \mathcal{C}_{nh}(u)} |V(v) \setminus N(x)|$.

As the search encounters all non-hollow eligible non-completion-forced nodes of T, then it discovers all the completion-minimal insertion nodes, and computes the cost of their associated minimal completion. We keep track of the minimum cost completion encountered during the search and outputs the corresponding insertion node at the end.

Finally, we need to update the cotree T for the next incremental step of the algorithm (as depicted in Figure 3). To this purpose, we use the algorithm of [7] as explained below.

Complexity. All tests on node u can be done in $O(|\mathcal{C}_{nh}(u)|$. Note that we never manipulate the hollow nodes: we search only non-hollow nodes and when we need to test the number of hollow children of u we avoid to count them by computing their number as $|\mathcal{C}(u)| - |\mathcal{C}_{nh}(u)|$. Moreover, the computation of $cost - above(u)$ can be done in $O(1)$ time by noting that the sum $\sum_{u' \in \mathcal{C}(v), u' \neq u} |V(u') \setminus N(x)|$ can rather be computed as $|V(v) \setminus N(x)| - |V(u) \setminus N(x)|$. Finally, $cost(u)$ can be computed in $O(|\mathcal{C}_{nh}(u)|$ as well.

Therefore all the searches take a time proportional to the number of non-hollow nodes of T. As shown in [24], this number is bounded by $|N'(x)|$, the completed neighbourhood of x. Indeed, from Theorem 1, all non-hollow nodes are filled except the ancestors of the insertion node u. Let v be a non-hollow child of one ancestor of u, then v is filled and it follows that the sum of the sizes of T_v for all such v is bounded by $N'(x)$. The number of ancestors of v is also linearly bounded by $N'(x)$ as half of these ancestors are series and therefore have a child v which is filled.

When, the insertion node u has been determined, the completed neighbourhood $N'(x)$ of x can be easily computed in extension by a search of the tree, which takes $O(|N'(x)|)$ time. Then, the cotree of H can be computed from the cotree of G in the same complexity thanks to the algorithm of [7]. Overall, one incremental step of the algorithm takes $O(|N'(x)|)$ and the whole running time of the algorithm is $O(n + m')$, where m' is the number of edges in the output cograph.

C Worst-case minimum-cardinality completion of very sparse graphs

The aim of this section is to prove Theorem 2.

Definition 7 (Vertex expander, see [20] for a survey). *A graph G is a c-expander if, for every vertex subset $S \subseteq V(G)$ with $|S| \leq \frac{|V(G)|}{2}$ we have $|N(S)| \geq c \cdot |S|$.*

Proposition 1 ([27]). *Let k be an integer and let G be a graph whose rank-width is at most k. Then there exists a subset $S \subseteq V(G)$ of vertices, such that $\frac{n}{3} \leq |S| \leq \frac{n}{2}$ and* cutrank$(S) \leq k$.

We remark that Proposition 1 is stated by Oum and Seymour [27] in terms of symmetric submodular functions. Also see [30] for definitions of *rank-width* and *cutrank*.

Lemma 7. *Let G be a c-expander and $S \subseteq V(G)$ be a subset of vertices of size at most $\frac{n}{2}$. Then, $|N(V \setminus S)| \geq c.|S|/3$.*

Proof. Denote $S' = N(V \setminus S)$ and suppose for contradiction that $|S'| < c.|S|/3$. Note that from the definition of a c-expander, we necessarily have $c \leq 2$. It follows that $|S'| < 2|S|/3$, and so $|S \setminus S'| > |S|/3$. Moreover, $N(S \setminus S') \subseteq S'$ and therefore $|N(S \setminus S')| \leq |S'| < c.|S|/3$. This contradicts the expansion property for $S \setminus S'$. □

We can now prove Theorem 2.

Proof of Theorem 2: Cographs have rank-width 1. Then, by Proposition 1, there exists a set $S \subseteq V(H) = V(G)$, such that $\frac{n}{3} \leq |S| \leq \frac{n}{2}$ and $(S, V \setminus S)$ is a split of H (or equivalently cutrank$_H(S) = 1$). This means that the edges between S and $V \setminus S$ induce a complete bipartite graph in H. Since G is a c-expander, $|N(S)| \geq c.|S|$. By Lemma 7, we also have $|N(V \setminus S)| \geq c.|S|/3$. Consequently, the number of edges crossing the split $(S, V \setminus S)$ of H is at least $c^2.|S|^2/3 \geq c^2.n^2/27$. □

D An $O(n + m \log^2 n)$ algorithm

D.1 Data structure

Our data structure is composed of two copies of the cotree: one stored in a basic data structure and one using the advanced dynamic data structure of [29] named *dynamic trees*. Actually, we could use only the advanced data structure, as it can be patched to contain the additional information we need (which we store in the basic data structure). But to avoid questions about the compatibility of such a patch with the performances of the data structure of [29], we prefer to store the additional information we need and to perform the related operations

independently in another structure. This is the reason why we describe our algorithm using two structures.

Moreover, we also enhance the basic data structure storing the cotree with one additional feature: given a node u and two of its children u_1, u_2, we can determine which of u_1, u_2 appears first in the list of children of u in $O(1)$ time. To this purpose, the set of children of a node u is not only stored in a doubly linked list, as in the classical version of the tree, but a copy of this list is also stored using the *order data structure* of [31, 11]. This data structure allows to answer *order queries*, i.e. which of two given elements of the list precedes the other one, and supports two update operations, `insert` and `delete`. The delete operation removes a given element from the order data structure while the insert operation insert a new element in the order data structure just after a specified element. The order query and the two update operations all take $O(1)$ worst-case time.

Finally, a node u of the tree stores its number of children $|\mathcal{C}(u)|$ and a bidirectional couple of pointers to the corresponding node of the data structure of [29], so that we can move from one element in one copy of the cotree to the same element in the other copy in $O(1)$ time.

Dynamic trees [29] In addition to the classical data structure described above, we also use the data structure developed in [29] to store a copy of the cotree T and maintain it at each incremental step. This data-structure maintains a dynamic forest rather than a single tree. It allows to answer two kinds of query:

`lowest-common-ancestor?` Given two nodes u and v of T, provide the lowest common ancestor $lca(u, v)$ of u and v.

`next-step-to-descendant?` Given a node u of T and one of its strict descendants v, provide the (unique) child of u which is an ancestor of v.

These two kinds of query are handled in $O(\log n)$ worst-case time in the data structure of Sleator and Tarjan [29]. To be precise, the second operation is not described in [29], but it can be obtained as a combination of other operations they provide. Indeed, their data structure also supports, in the same complexity:

- an update operation called `evert`(u) which, given a vertex u of T, makes u become the root of T, and
- a query operation named `root?`(u) that provides the root of the tree T to which node u belongs.

Then, the query `next-step-to-descendant?`(u, v) we use here can be resolved by the sequence of operations (two updates and two queries): $r =$`root?`(u), `evert`(v), `parent?`(u), `evert`(r), which takes $O(\log n)$ time.

Along our incremental algorithm, we need to maintain the dynamic data structure of [29], which can be done thanks to the following update operations.

`cut.` Given a node u in a tree T of the forest F such that u is not the root of T, remove the edge between u and $parent(u)$. Then, u becomes the root of its new tree T' in forest F.

`link`. Given a node u in a tree T of the forest F such that u is not the root of T and given the root v of a tree $T' \neq T$, make u the parent of v.

Note that operations `cut` and `link` are converse of each other. As for queries, all update operations takes $O(\log n)$ worst-case time.

D.2 Algorithm

Our algorithm determines the set W of the nodes that are simultaneously eligible, non-hollow and non-completion-forced and that are minimal for the ancestor relationship among nodes having these three properties. Then, it picks any of them to be the insertion node of the minimal completion returned at this incremental step. Indeed, since nodes of W satisfy the conditions of Lemma 4 and none of their children does (because nodes of W are minimal for the ancestor relationship), it follows that nodes of W are completion-minimal insertion nodes. In order to get the improved $O(n+m\log^2 n)$ complexity, we avoid to completely search the upper tree $T^{up}_{N(x)}$ to determine W. Instead, we use a limited number of *lca* queries.

Clearly, if a parallel node u of T is the *lca* of two leaves in $N(x)$ then $T_u \setminus \{u\}$ contains no eligible node. Let P_{max} be the set of parallel common ancestors of vertices of $N(x)$ that are maximal for the ancestor relationship and let us denote $W' = P_{max} \cup N_{out}$, where N_{out} is the set of vertices of $N(x)$ that are not descendant of any node in P_{max}, i.e. $N_{out} = N(x) \setminus \bigcup_{p\in P_{max}} V(p)$. Note that all the nodes $w' \in W'$ are eligible, and so are their ancestors. It follows that the set W that we want to compute is the set of the non-completion-forced nodes in the upper tree $T^{up}_{W'}$ that are minimal for the ancestor relationship.

Finding an inclusion-minimal insertion node. In order to compute W, we start by computing the tree $\widetilde{T} = T^{xtr}_{N(x)\cup A_x}$ extracted from the leaves in $N(x)$ and the set A_x of their lowest common ancestors, i.e. nodes of T that are the lowest common ancestor of two nodes of $N(x)$. Then, we remove from $\widetilde{T}$ all the strict descendants of the maximal parallel nodes of $\widetilde{T}$. The leaves of the resulting tree are exactly nodes of W'. Note that since $\widetilde{T}$ has exactly d leaves and since all its internal nodes have degree at least 2, then the size of $\widetilde{T}$ is $O(d)$.

Let us now show how to compute $\widetilde{T}$ in $O(d\log^2 n)$ time. To this purpose, we use a special order on the vertices of the cograph G which is called a *factorising permutation* [5] and we sort the neighbours of x respectively to this order. A factorising permutation is the order in which the vertices of G (which are the leaves of the cotree) are encountered when performing a depth-first search of the cotree T. There are as many different factorising permutations as different depth-first search of T. Here, we use the factorising permutation π which is obtained by visiting the children of one node u of T in the order they are stored in the list of the children of u used in the implementation of the cotree. To determine whether a vertex y_1 is before or after a vertex y_2 in the factorising permutation π, we can proceed as follows: 1) find $u = lca(y_1, y_2)$ and find the two children u_1 and u_2 of u that are respectively the ancestor of y_1 and y_2, and 2) determine

whether u_1 is before or after u_2 in the list of children of u. Operation 1) can be executed in $O(\log n)$ time thanks to the data structure of [29] by performing one `lowest-common-ancestor?` query and two `next-step-to-descendant?` queries. Operation 2) can be executed in $O(1)$ time using the order data structure of [31, 11]. Then, comparing the order of occurrence of two vertices y_1 and y_2 in π takes $O(\log n)$ time and totally, sorting all the neighbours of x respectively to order π takes $O(d \log d \log n) = O(d \log^2 n)$ time.

Once the neighbours of x are sorted in the order $x_1, x_2, \ldots, x_d$ in which they appear from left to right in π, we consider them one by one in this order and at each step we compute the tree T_i extracted from $\{x_1, \ldots, x_i\}$ and their lowest common ancestors. Then, at the end of the computation, when $i = d$, we obtain $T_d = \widetilde{T}$. For each i between 2 and k, we obtain T_i from T_{i-1} as follows: we compute $v_i = lca(x_{i-1}, x_i)$ and we insert it at its correct position in the tree T_{i-1} built so far.

Note that, since we consider the x_i's from left to right in the order of the factorising permutation π, the newly computed common ancestor v_i is the only node that may be in T_i but not in T_{i-1}. Moreover, for the same reason, if v_i is not yet a node of T_{i-1} then v_i has to be inserted on the rightmost branch of the tree T_{i-1}, and if v_i is already a node of T_{i-1} then v_i already belongs to this branch, and so we discover it when we try to insert it on this branch. In order to do so, we climb up the rightmost branch of T_i, starting from the father of x_{i-1}, and for each node v encountered on this branch we determine whether v_i is higher or lower than v in the tree (or eventually equal) by computing $lca(v, v_i)$. The total number of comparisons (treated by lca queries) made along the computation of T_d is $O(d)$. Indeed, as explained in [13], every time we pass above a node v on the rightmost branch, v leaves the rightmost branch for ever and will then never participate again to any comparison. Then, the total number of lca queries we need to built $\widetilde{T}$ (including the $d-1$ queries made on the pairs of neighbours of x appearing consecutively in the order of the factorising permutation) is proportional to its size, that is $O(d)$. Since each of these queries takes $O(\log n)$ time thanks to the data structure of [29], the complexity of building $\widetilde{T}$ from the sorted list of neighbours of x is $O(d \log n)$.

Once $\widetilde{T}$ is built, a simple search starting from its root determines the set P_{max} of its parallel nodes that are maximal for the ancestor relationship and we cut off from $\widetilde{T}$ all the strict descendants of nodes in P_{max}, which takes $O(d)$ time. The leaves of the resulting tree are the nodes of W' that we wanted to determine. As $\widetilde{T}$ has size $O(d)$, this step takes $O(d)$ time.

We now need to find the non-completion-forced nodes of the upper tree T_W^{up} that are minimal for the ancestor relationships. To that purpose, for each $w' \in W'$, we determine its lowest non-completion-forced ancestor $nfa(w')$ in T. From the definition of P_{max}, the lowest parallel ancestor of w' is non-completion-forced. Therefore, $nfa(w')$ cannot be higher in T than the grand-parent of w'. It follows that we have to check the non-completion-forced condition only for w' and its parent, which can be done as follows. If w' is a leaf of T, i.e. $w' \in N_{out}$, then w' is forced. If w' is a parallel node of T, i.e. $w' \in P_{max}$, then w' is forced iff

its number of children in $\widetilde{T}$ equals its number of children in T. Now, for the parent v of W', if v is a parallel node, as noted above, v is necessarily non-completion-forced. Otherwise, if v is a series node, v is completion-forced iff i) v belongs to $\widetilde{T}$ and ii) its number of children in $\widetilde{T}$ equals its number of children in T, and iii) all its children in T belong to W' and iv) all its children in T are forced (cf. conditions given above for w'). If none of w' and $parent(w')$ is non-completion-forced, then necessarily $parent(parent(w'))$ is. As testing these conditions for one node u takes $O(|\mathcal{C}_{nh}(u)|)$ time, then determining all the $nfa(w')$'s takes $O(d)$ time. Finally, we determine the $nfa(w')$'s that are minimal for the ancestor relationship, i.e. nodes of W, by searching T upward on at most two levels, starting from each of the nodes in W'. This also takes $O(d)$ and the total complexity of finding one completion-minimal insertion node, i.e. an arbitrary node w in W, is $O(d + d\log n + d\log^2 n) = O(d\log^2 n)$.

D.3 Updating the data structure under the insertion of x

In the previous section, we showed how to determine the insertion node w and the list of its children to be filled. Then, depending on whether w is a parallel or a series node, the cotree T must be modified as shown in Figure 3, and the data structure of [29] associated to the cotree must be updated accordingly. The key for doing so while preserving the $O(d\log^2 n)$ time complexity is to perform operations involving only the non-hollow children of w. Indeed, their number is $O(d)$, while the number of the hollow children of w can be up to $\Omega(n)$ and arbitrarily large compared to d.

After the insertion of x, the insertion node w is replaced by three nodes, see Figure 3. Two of them have the same label as w: one w_h has for children the hollow children of w and the other one w_{nh} has for children the non-hollow children of w. In order to preserve the complexity, it is important to form these two nodes as follows. We cut from w its non-hollow children and its parent, we then obtain w_h, still linked to its children. Then, we link to a new node all the non-hollow children of w. This takes $O(d\log n)$ as it requires $O(d)$ `cut` and `link` operations, each of which is supported in $O(\log n)$ time by the data structure of [29], and the corresponding `delete` and `insert` operations in the order data structure storing the lists of children of the nodes in the tree take $O(1)$ time. The rest of the transformations in order to get the new tree as depicted in Figure 3 only requires 4 `link` operations. Thus, the time complexity of the update step is $O(d\log n)$.

References cited only in the appendices

[30] Oum, S., Seymour, P.: Approximating clique-width and branch-width. Journal of Combinatorial Theory, Series B 96(4), 514–528 (2006)

[31] Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two simplified algorithms for maintaining order in a list. In: 10th European Symposium on Algorithms (ESA 2002), LNCS, vol. 2461, pp. 152–164. Springer (2002)