Linear-Time Minimal Cograph Editing

Christophe Crespelle

University of Bergen, Department of Informatics, N-5020 Bergen, NORWAY
christophe.crespelle@uib.no

Abstract. We present an algorithm for computing a minimal editing of an arbitrary graph G into a cograph, i.e. a set of edits (additions and deletions of edges) that turns G into a cograph and that is minimal for inclusion. Our algorithm runs in linear time in the size of the input graph, that is $O(n+m)$ time where n and m are the number of vertices and the number of edges of G, respectively. Our algorithm is incremental on the vertices and has two remarkable properties: (1) at each incremental step, it is able to provide a set of edits incident to the newly considered vertex which is not only minimal for inclusion but also has minimum cardinality and (2) the total number of edits output at the end of the algorithm is never more than m. These properties are very useful in practice for using our inclusion-minimal algorithm as a heuristic for solving the minimum-cardinality version of the problem, which is NP-hard.

1 Introduction

We consider the problem of *editing* 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 corresponds to the notion of projection in geometry: given an element a of a ground set X equipped with a distance and a subset $S \subseteq X$, find an element of S that is closest to a for the provided distance (here, the number of elementary modifications performed on the graph). This is also the meaning of modification problems in algorithmic graph theory: they answer the question to know how far is a given graph from satisfying a target property.

Here, we consider the *edge modification problem* called *editing*, where two operations are allowed: adding an edge and deleting an edge. In other words, given a graph $G=(V,E)$, we want to find a set $M \subseteq \{\{x,y\} \mid x,y \in V\}$ of pairs of vertices, called *edits*, such that the edited graph $H=(V, E \Delta M)$ belongs to the target class. In this case, the quantity to be minimised, called the *cost* of the editing, is the number $|M|$ of adjacencies that are modified, i.e. the number of edges that are added plus the number of edges that are deleted. There exist two other edge modification problems, called *completion* and *deletion*, which are particular cases of editing where only addition of edges or only deletion of edges is allowed, respectively. Edge modification problems play an essential role in algorithmic graph theory, where they are closely related to some important graph parameters, such as treewidth [1]. They are also useful for many problems arising in computer science, e.g. sparse matrix multiplication [53], anonymisation in networks [42] and clustering [30, 3], and in other disciplines such as archaeology [39], molecular biology [6, 11] and genomics, where they played a key role in the mapping of the human genome [27, 38]. Recently edge modification problems into the class of cographs and some of its subclasses has become a powerful approach to solve problems arising in complex networks analysis, such as inference of phylogenomics [35, 34], identification of groups in social networks [37, 48] and measures of centrality of nodes in networks [54, 9]. For these applications, the need to treat real-world datasets, whose size is often huge and constantly growing, asks for more efficient algorithms both with

regard to the running time and with regard to the quality (number of edits) of the solution returned.

Unfortunately, finding the minimum number of edits to be performed in an editing problem is NP-hard for most of the target classes of interest (see, e.g., the thesis of Mancini [45] for further discussion and references). To deal with this difficulty of computation, the domain has developed a number of approaches, including approximation [49, 36], restricted input [40, 10, 8, 50, 46, 7, 41], parameterization [12, 22, 5, 26, 14, 2, 13] and exact exponential algorithms [4]. Another popular approach, called *minimal editing*, is based on a relaxation of the problem which does not ask for the minimum number of edits but only ask for a set of edits which is minimal for inclusion, i.e. which does not contain any proper subset of edits that also results in a graph in the target class. This approach has been extensively used for completion problems, for the class of cographs itself [44, 19], as well as for many other graph classes, including chordal graphs [32], interval graphs [21, 51], proper interval graphs [52], split graphs [33], comparability graphs [31] and permutation graphs [20]. The main reason for the success of inclusion-minimal completion is that it provides a heuristic for minimum-cardinality completion that has usually a low polynomial complexity. This heuristic approach exploits the fact that different minimal completions can be obtained by making different choices along the algorithm. Making these choices at random and repeating the algorithm several times perform a sampling of the set of minimal completions. One can then keep the best solution obtained and hope that it is close enough from the optimal one.

Surprisingly, it seems that the inclusion-minimal approach has never been used for editing problems, where both addition and deletion are allowed. We presume that this is due to the conjunction of two reasons: dealing with both addition and deletion of edges is usually harder for algorithm design and at the same time, from a purely theoretic point of view, minimal editing has no particular interest compared to minimal completion for instance, since an inclusion-minimal set of added edges is also an inclusion-minimal set of edits. Nevertheless, from a practical point of view, when the object of interest is the edit distance to the target class, dealing with the general version of editing is essential as the number of edits obtained is often much lower when both operations are used. Moreover, we show in this paper that dealing with both operations may be beneficial not only for the number of edits output, but also for the time complexity of the algorithm.

Related work. Edge modification problems into the class of cographs and some of its subclasses, such as *quasi-threshold graphs* (also known as *trivially perfect graphs*) and *threshold graphs*, have already received a great amount of attention. In particular, in the parameterized complexity framework with the editing distance as parameter, [24, 23, 43, 47] obtained FPT algorithms while [23, 25, 29, 28] designed polynomial kernels.

Unlike the minimal editing problem, minimal cograph completion has already been studied. [44] designed an incremental algorithm that gives an inclusion-minimal cograph completion in time $O(n + m')$, where n is the number of vertices and m' the number of edges in the output cograph. Later, [19] improved the running time to $O(n + m \log^2 n)$ for the inputs where the number of edges m is small and m' is large. They also show that within the $O(n + m')$ time complexity, it is possible to determine the minimum number of edges to be added at each step of the incremental algorithm.

[37] and [34] designed heuristics, for cograph deletion and cograph editing respectively, that are not intended to output a minimal set of modifications. In the worst case, the algorithm of [37] runs in time at least $O(m^2)$ and the algorithm of [34] in time greater than $O(n^2)$. Finally, let us mention that for the class of quasi-threshold graphs, there exist two heuristics dedicated to the editing problem. The one in [48] runs in cubic time in the worst case while [9] obtains a complexity which is close to linear, but which remains quadratic in the worst case.

Our results. We design an algorithm that computes a minimal cograph editing of an arbitrary graph in linear time in the size of the input graph, i.e. $O(n+m)$ time. This is the first algorithm for a cograph edge modification problem that runs in linear time. Even compared to the $O(n+m')$-time algorithm of [44] for the pure completion problem, the $O(n+m)$ complexity we obtain here for the editing problem is a significant improvement since, as shown in [19], many instances with $m = O(n)$ edges require $m' = \Omega(n^2)$ edges in any of their cograph completions. Note that, as a particular case, our minimal cograph editing algorithm solves the cograph recognition problem in linear time, and the technique we use can actually be seen as an extension of the seminal work of [17].

Like the algorithms in [44, 19], our algorithm is incremental on the vertices and as the one in [19], it is able to provide a minimum number of edits to be performed at each incremental step and even able to list all the sets of edits having this minimum cardinality, within the same complexity. Moreover, all the different solutions that can be obtained at the end of the algorithm never contain more than m edits.

To obtain an $O(n + m)$ time complexity, we use the fact that there always exists an editing of cost m that deletes all the edges of the graph. Because of this, at each incremental step, we can ignore all the minimal editings that have cost more than d, the degree of the new vertex. This allows us to limit our exploration of the cotree to a part of it that has size $O(d)$. Our main technical contribution is to show how to identify this part in $O(d)$ time.

2 Preliminaries

All graphs considered here are finite, undirected, without multiple edges 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 $G = (V, E)$ and n stands for the cardinality $|V|$ of $V(G)$ and m for $|E|$. An edge between vertices x and y will be arbitrarily denoted by xy or yx. The (open) neighbourhood of x is denoted by $N(x)$ (or $N_G(x)$) and its closed neighbourhood by $N[x] = N(x) \cup \{x\}$. The subgraph of G induced by the set of vertices $X \subseteq V$ is denoted by $G[X] = (X, \{xy \in E \mid x, y \in X\})$.

For a rooted tree T and a node $u \in T$, the depth of u in T is the number of edges in the path from the root of T to u (the root has depth 0). We employ the usual terminology for *children*, *parent*, *ancestors* and *descendants* of a node u in T (the two later notions including u itself). We denote by $\mathcal{C}(u)$ the set of children of u and by $parent(u)$ its parent. The subtree of T rooted at u, denoted T_u, is the tree induced by the descendants of node u in T (which include u itself). We denote $lca(u, v)$ the least common ancestor of nodes u and v in T.

Two sets A and B overlap if $A \cap B \neq \varnothing$ and $A \setminus B \neq \varnothing$ and $B \setminus A \neq \varnothing$. If A and B do not overlap, then either they are disjoint or one is included in the other.

2.1 Cographs

There are several characterizations of the class of *cographs.* They are often defined as the graphs that do not admit the P_4 (path on 4 vertices) as induced subgraph. 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 the disjoint union of G_1 and G_2, i.e., the graph $G_{par} = \big(V_1 \cup V_2, E_1 \cup E_2\big)$. The series composition of two graphs G_1 and G_2 is the disjoint union of G_1 and G_2 plus all possible edges from a vertex of G_1 to one 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 a finite number of graphs.

This gives a nice representation of a cograph G by a tree whose leaves are the vertices of the graph and whose internal nodes (non-leaf nodes) are labelled P, 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 [16] and is called the *cotree* of G, see the example on Figure 1. Note that the subtree T_u rooted at some node u of cotree T also defines a cograph whose vertex set, denoted $V(u)$, is the set of leaves of T_u. We denote Ser and Par the set of series and parallel nodes of T respectively. The adjacencies between vertices of a cograph can easily be read off its cotree, in the following way.

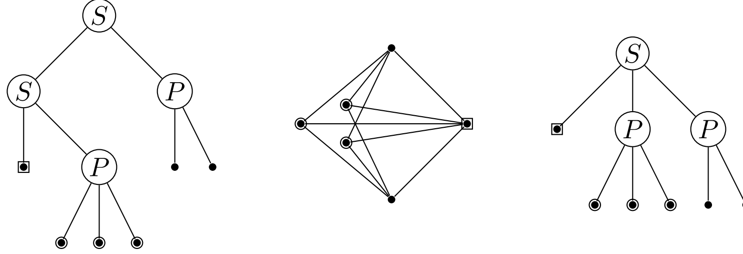

Fig. 1. Example of a labelled construction tree (left), the cograph it represents (center), and the associated cotree (right). Some vertices are decorated in order to ease the reading.

Remark 1 *Two vertices x and y of a cograph G having cotree T are adjacent iff the least 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.*

Let us emphasize that the class of cographs is *hereditary*, i.e., an induced subgraph of a cograph is also a cograph (since the class admits a definition by forbiden induced subgraphs).

2.2 Incremental minimal cograph modification

Note that every graph G has a cograph completion and a cograph deletion, and consequently a cograph editing as well. For completion, one can simply add all the missing edges to G and observe that the complete graph is a cograph. For deletion, one can delete all the edges and observe that the empty graph is also a cograph.

Our approach for computing a minimal editing of an arbitrary graph G is incremental, in the sense that we take the vertices of G one by one, in an arbitrary order $(x_1, \dots, x_n)$, and at step i we compute a minimal cograph editing H_i of $G_i = G[\{x_1, \dots, x_i\}]$ from a minimal cograph editing H_{i-1} of G_{i-1}, by modifying only adjacencies involving x_i. This is possible thanks to the following observation that is general to all hereditary graph classes that are stable under the addition of universal vertices and isolated vertices, like cographs.

Lemma 1 (see e.g. [51]). *Let G be an arbitrary graph and let H be a minimal cograph editing (resp. completion or deletion) of G. Consider a new graph $G' = G + x$, obtained by adding to G a new vertex x adjacent to an arbitrary set $N(x)$ of vertices of G. There is a minimal cograph editing (resp. completion or deletion) H' of G' such that $H' - x = H$.*

Let us mention that the conditions of application of the above Lemma for editing are actually lighter: in addition to the fact that the class is hereditary, it is sufficient that any graph in the class is an induced subgraph of another graph in the class (i.e. there is no maximal element for the induced subgraph relationship).

The new problem. From now on, we consider the following problem, with slightly modified notations.

Let G be a cograph, and let $G+x$ be the graph obtained by adding to G a new vertex x adjacent to some set $N(x)$ of vertices of G. Our goal is to compute a minimal cograph editing H of $G+x$ such that $H-x=G$. We sometimes write $G+(x,N(x))$ instead of $G+x$, when we want to make the neighbourhood of x in $G+x$ explicit, and we denote $d=|N(x)|$.

Definition 1 (Full, hollow, mixed). *Let G be 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 S is* uniform. *We use the same vocabulary for nodes u of T, referring to their associated set of vertices $V(u)$.*

From [17, 18], we have the following characterisation of the case where the insertion of x in cograph G yields a cograph $G+x$.

Theorem 1 (Reformulated from [17, 18]). *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.

3 Characterisation of minimal cograph editings of $G+x$

In this section, we build upon Theorem 1 to get a characterisation of all the minimal cograph editings of $G+x$ (Lemmas 7 and 8 below), extending the work of [19] for pure completion. Note that from Theorem 1, any minimal editing defines a unique insertion node.

Lemma 2. *Let G be a cograph with cotree T and let x be a vertex to be inserted in G. The insertion node u of a minimal cograph modification H (editing, completion or deletion) of $G+x$ is a mixed node of T wrt. x in $G+x$.*

Proof. Let us denote $N'(x)$ the neighbourhood of x in H and let $Mod'(x)$ denote the subset of vertices of G whose adjacency with x is modified between $G+x$ and H, i.e. $Mod'(x)=N(x)\Delta N'(x)$. From Theorem 1, after modification, node u is mixed wrt. $N'(x)$. Suppose for contradiction that u is not mixed before modification, that is with regard to $N(x)$. Then, either u is full w.r.t. $N(x)$ and some edges between x and $V(u)$ have been deleted, or u is hollow w.r.t. $N(x)$ and some edges between x and $V(u)$ have been added. In both cases, the set $Mod''(x)$ defined as $Mod''(x)=Mod'(x)\setminus V(u)$ is strictly included in $Mod'(x)$, and the resulting modified neighbourhood $N''(x)=N(x)\Delta Mod''(x)$ of x is such that $G+(x,N''(x))$ is also a cograph. It follows that the modification that results in $N'(x)$ is not a minimal cograph modification: contradiction. Thus, u is mixed with regard to $N(x)$. □

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* minimal insertion node *iff there exists a minimal editing H of $G+x$ such that u is the insertion node associated to H.*

Definition 3 (Completion-forced [19]). *Let G be a cograph with cotree T and let x be a vertex to be inserted in G. A* completion-forced *node u is inductively defined as a node satisfying at least one of the three following consditions:*

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 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 the missing edges between x and $V(u)$ are added.*

Proof. This lemma is somehow a stronger version of Lemma 3 in [19] (because it gives an equivalence instead of a simple implication), which we use for part of our proof. Let u be a completion-forced node in T. Then, u is also completion-forced in T_u and Lemma 3 in [19] implies that u is made full in all the completions of $G_u + x$, which is what we wanted to prove.

We prove the converse implication by induction of $|V(u)|$. Consider a non-completion-forced node u of T, we will show that there exists a completion of $G_u + x$ that does not make $V(u)$ full. 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 make $V(v)$ full. Then, the completion H of $G_u + x$ that coincides with H' on $V(v)$ and that makes all the other children of u full is a cograph completion of $G_u + x$ that does not make $V(u)$ full. Now, if u is a parallel node, then u has at least one hollow child v. Leave v hollow and make all the other children of u full: this yields a cograph and again, $V(u)$ is not full in this completion. □

Definition 4 (Deletion-forced). *A* deletion-forced *node u is inductively defined as a node satisfying at least one of the three following conditions:*

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

Lemma 4. *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 deletion-forced iff there exists a unique cograph deletion of $G_u + x$, which is the one where all the edges between x and $V(u)$ are deleted.*

Proof. Note that since the inductive definition of a deletion-forced node is exactly the complement of the inductive definition of a completion-forced node (Definition 3), it follows that u is deletion-forced in $G + x$ iff u is completion-forced in $\overline{G} + \bar{x}$ where $\bar{x}$ is a new vertex such that the neighbourhood of $\bar{x}$ in $V(\overline{G}) = V(G)$ is the complement of the neighbourhood of x in $V(G)$. With this observation, the current lemma appears to be simply a rewriting of Lemma 3 in the complement graph. □

Definition 5 (Consistent and settled). *An editing H of $G + x$ is* consistent *at a node u of T iff all the following conditions are satisfied:*

1. *H makes x adjacent to all the vertices $y \notin V(u)$ that have a series least common ancestor with u and H makes x non-adjacent to all the vertices $z \notin V(u)$ that have a parallel least common ancestor with u, and*
2. *all the children of u are uniform in H, and*

3. all the children of u that are hollow in $G+x$ are hollow in H as well and all the children of u that are full in $G+x$ are full in H as well.

If, in addition, u is mixed in H, we say that H is settled *at u. A* minimum consistent editing *(resp.* minimum settled editing*) is an editing consistent with some node u (resp. settled at u) and having minimum cost among such editings. The minimum cost of an editing settled at u is denoted $mincost(u)$.*

Lemma 5. *If H is an editing settled at u, then the insertion node of H is u. If H is a minimal editing whose insertion node is u, then H is an editing settled at u.*

Proof. Clearly, if H is settled at u, then u fulfills all the conditions of Theorem 1 and u is then the insertion node of H. Conversely, if u is the insertion node of H, then, from Theorem 1, u satisfies conditions 1 and 2 of Definition 5. Moreover, assume for contradiction that u has one hollow child v in $G+x$ which is not hollow in H. Then, consider the completion H' obtained by removing from H the edges added between x and $V(v)$. In H', either u or the lowest series ancestor of u satisfies the conditions of Theorem 1. Therefore, H' is a cograph completion of $G+x$ and so H is not minimal: contradiction. Thus, v is hollow in H. The case where v is a full child of u can be treaten similarly and shows that Condition 3 of Definition 5 always holds, which achieves the proof of the lemma. □

Lemma 6. *Let G be a cograph with cotree T and let x be a vertex to be inserted in G. For any mixed node u of T, T_u contains some minimal insertion node.*

Proof. Consider a mixed node v lower possible in T_u, that is having all its children uniform. From Definition 5, there exists a unique editing of $G+x$ settled at u, we denote it H and we show that it is minimal.

Let H' be a cograph editing of $G+x$ whose insertion node is v' and let us show that the set of modifications defining H' is not strictly included in the one defining H. First, from Lemma 2, v' has to be a mixed node, and since there are no mixed node in $T_v \setminus \{v\}$, it follows that $v' \notin T_v \setminus \{v\}$. Moreover, since H does not modify any adjacency between x and vertices of $V(v)$ then clearly H is minimal among editings that have v as insertion node. Finally, from Theorem 1, the only nodes of T that are mixed in H' are the ancestors of v'. Then, if $v' \notin T_v$, v is uniform in H'. Since v is mixed before editing, this implies that some adjacencies between x and vertices of $V(v)$ are changed in H', while H changes none of them: the set of modifications of H' is not included in the one of H. This shows that H is minimal. Since H is settled at v then the insertion node of H is $v \in T_u$ (see Lemma 5) and the statement of the lemma follows. □

Lemma 7. *An editing H settled at a mixed node u is a minimal editing iff exactly one of the two following conditions is satisfied:*

- *u is a parallel node and either at least two children of u are full in H or exactly one child of u is full in H and this child is completion-forced.*
- *u is a series node and either at least two children of u are hollow in H or exactly one child of u is hollow in H and this child is deletion-forced.*

Proof. We prove it in the case where u is a parallel node. First, let us show that if there exists an editing H' whose set of modifications is strictly included in the one of H, then Condition 1 of the lemma is not satisfied. First, we show that the insertion node u' of H' is necessarily in $T_u \setminus \{u\}$. Note that if $u' \notin T_u$, then u is uniform in H' (see Theorem 1). Moreover, as H is settled

at u, by definition, it both leaves some edges between x and $V(u)$ unchanged and leaves some non-edges between x and $V(u)$ unchanged. Thus, if $u' \notin T_u$, H' is not included in H. Moreover, if the insertion node of H' is u, H' is not included in H. Indeed, since H' and H are different, from Definition 5, they can differ only on one mixed child v of u: one of H or H' makes v full while the other one makes it hollow. As v is mixed in $G + x$, none of H and H' is included in the other one. Thus, since H' is strictly included in H, its insertion node u' belongs to $T_u \setminus \{u\}$.

Let v be the child of u that is an ancestor of u'. Because u is parallel and is an ancestor of the insertion node u' of H', from Theorem 1, it follows that all the children of u different from v are hollow in H'; and they are uniform in H, because H is settled at u. Then, since the set of modifications of H' is included in the one of H, the children of u distinct from v must also be hollow in H, while v must be full, as u is mixed in H from Definition 5. Then, since the set of modifications of H' is included in the one of H, H' uses only addition of edges between x and $V(v)$. Moreover, since u' belongs to T_v, v is mixed in H'. Consequently, H' restricted to $V(v)$ is a completion of $G_v + x$ that does not fill $V(v)$. Thus, v is not completion-forced and Condition 1 of the lemma is not satisfied.

Conversely, let us now show that if Condition 1 of the lemma is not satisfied, then H is not minimal. In this case, u is a parallel node with exactly one child v full in H and this child v is not-completion forced. Then, from Lemma 3, there exists a completion H''_v of $G_v + x$ that does not fill $V(v)$. Let us denote $N''(x)$ the neighbourhood of x in H'' and $N_H(x)$ the neighbourhood of x in H. Clearly, the graph H' obtained from G by adding vertex x with neighbourhood $N'(x) = (N_H(x) \setminus V(v)) \cup N''(x)$ is a cograph editing of $G + x$. Moreover, as $N''(x) \subsetneq V(v)$, the set of modifications of H' is strictly included in the one of H. Therefore, H is not a minimal editing of $G + x$.

This ends the proof in the case where u is a parallel node. The case where u is a series node is very similar, by taking the complement. □

Definition 6 (Clean). *A mixed node u of T is* clean *iff one of the two following conditions holds:*

- *the parent v of u is a parallel node and u is the unique non-hollow child of v, or*
- *the parent v of u is a series node and u is the unique non-full child of v.*

Lemma 8. *Let G be a cograph with cotree T and let x be a vertex to be inserted in G. A mixed node u of T is a minimal insertion node iff exactly one of the two following conditions are satisfied:*

1. *u is parallel and either u has at least 3 children and no clean non-completion-forced child or u has exactly 2 children, at least one of which is completion-forced.*
2. *u is series and either u has at least 3 children and no clean non-deletion-forced child or u has exactly 2 children, at least one of which is deletion-forced.*

Proof. Let us prove the lemma in the case of a parallel node. First, let us show that if the conditions of the lemma are satisfied, then there exists one editing settled at u which is minimal. If u has at least 3 children, then either u has no clean child or u has one clean child v which is completion forced. In the latter case, the completion H settled at u obtained by filling v satisfies Condition 1 of Lemma 7 and is therefore minimal. In the former case, since u is mixed, u has at least one child v non-full and since u has no clean child, u has at least two children v_1, v_2 that are non-hollow. Moreover, v, v_1, v_2 can always be chosen pairwise distinct. The completion that empties v and fill v_1 and v_2 again satisfies Condition 1 of Lemma 7 and is therefore minimal.

If u has exactly 2 children, at least one of them v_1 is completion forced and at least one v_2 is non-full (since u is mixed), and v_1 and v_2 can always be chosen distinct. Define H as the

editing settled at u that fills v_1 and makes v_2 hollow: H satisfies Condition 1 of Lemma 7, and so H is minimal.

Conversely, we show that if the conditions of the lemma are not satisfied, then there is no editing settled at u that is minimal. If u has at least 3 children and one clean non-completion-forced child v, then all children of u different from v are hollow and v must be mixed (as u is). It follows that there is only one editing H settled at u: the one that fills v and leave the other children of u hollow. As v is non-completion-forced, H does not fulfill the conditions of Lemma 7. Consequently, H is not minimal and there is no minimal editing settled at u.

If u has exactly 2 children, as u does not satisfies Condition 1 of the present lemma, then its 2 children are non-completion-forced. Because u has 2 children, there exist at most two editings settled at u: the editings H who fill one non-hollow child of u and makes the other one hollow. Again, because the unique child filled in H is non-completion-forced, H does not satisfy conditions of Lemma 7, which implies that there is no minimal editing settled at u in this case as well.

This achieves the proof for a parallel node. The case of a series node is very similar by taking the complement. □

4 An $O(n)$-time algorithm for minimal cograph editing of $G + x$

In this section, we denote NH the set of non-hollow nodes of T. For a node $u \in T$, we also denote $\mathcal{C}_{nh}(u)$ its set of non-hollow children. Our algorithm takes as input the cotree T of G, where each node u stores the number of leaves of T_u, and the new vertex x together with the list of its neighbours $N(x)$. It works in two steps.

First step. In the data structure we use, each node u of T stores the number $|V(u)|$ of leaves in T_u. In addition, we compute some basic information about the nodes of T with regard to the new vertex x, in the same way as [19] does. More explicitely, for each non-hollow node u of T, we determine the list of its non-hollow children, their number, the number of neighbours of x in $V(u)$, whether u is completion-forced or not, whether u is full or mixed. In addition, we also determine for each non-hollow node u whether it is deletion-forced. This can be easily done as follows: for a series node, we just check that its number of full children is 0 and for a parallel node, we check that all its non-hollow children are deletion-forced.

Second step. We parse the mixed nodes of T, which, from Lemma 6, are excatly the nodes whose subtree contain some minimal insertion node. Observe, that when the set of mixed nodes is not empty, it is connected and contain the root of T. Therefore, the algorithm first checks that the root is mixed, otherwise there is no mixed node in T and the only minimal editing is the one having an empty set of modifications. Then, the algorithm checks for all the children of the current node (initially the root) whether they are mixed and parse, in a depth-first manner, those for which the test is positive (recall that this information is directly available after the first step).

During this depth-first search, we compute for each node u encountered the number of changes, denoted cost-above(u), to be performed on the ajacencies between x and the vertices of $V(G) \setminus V(u)$ in any editing settled at u. This can be easily computed[1] during the search by noticing that:

- if the parent v of u is a parallel node, then cost-above(u) $=$ cost-above(v) $+$ $\sum_{u' \in \mathcal{C}(v) \setminus \{u\}} |V(u') \cap N(x)|$; and

[1] Actually, for complexity reasons, the values of these expressions are computed by avoding to perform the sum, see paragraph *complexity* below.

– if the parent v of u is a series node, then cost-above(u) = cost-above$(v) + \sum_{u' \in \mathcal{C}(v) \setminus \{u\}} |V(u') \setminus N(x)|$.

Then, for each mixed node u encountered during the depth-first search, the algorithm determines whether u is an minimal insertion node by checking the conditions of Lemma 8. A difficulty is that, according to Definition 5 and Lemma 7, the number of minimal editings having a given insertion node u may be exponential in the number of children of u. This implies that our algorithm cannot parse them one by one and preserve its linear complexity. Nevertheless, note that, in linear complexity, our algorithm parses all minimal insertion node and that Lemma 7 somehow gives a way to consider at once all the minimal editings having insertion node u. Here we use this possibility for determining, in $O(|\mathcal{C}_{nh}(u)|)$ time, one editing H settled at u that is minimum for the number of edits. In the case where u has exactly two children, this is very easy as there are at most two editings settled at v: for each of them, we test whether it satisfies conditions of Lemma 7, and we keep the one that has the minimum number of edits among those satisfying these conditions. If u has at least 3 children, we proceed as follows.

As required by Definition 5, the children of u that are hollow in $G + x$ must be hollow in H and the children of u that are full in $G+x$ must be full in H. Then, in order to define H, we just have to decide for each mixed child of u whether it will be hollow or full in H. To that purpose, we color the mixed children of u as red or blue, with the meaning that the children colored red are full in H and the children colored blue are hollow in H. The following procedure assigns the colors to children of u in three steps. We describe it for u a parallel node, the series case is very easy to deduct by complement. First step: we color red all the mixed children v of u such that $|V(v) \cap N(x)| \geq |V(v) \setminus N(x)|$, all the other mixed children are colored blue. Second step: if u has no hollow child and no mixed child has been colored blue in the first step, then we select one red mixed child v realising the minimum of the quantity $|V(v) \cap N(x)| - |V(v) \setminus N(x)|$ and we color v blue. Third step: if the number nb_{filled} of children of u that are either full or colored red is such that $nb_{filled} < 2$ (note that this can happen only when the test of the second step failed and no color was changed at this step), then we select $2 - nb_{filled}$ blue mixed children of u that realise the minimum of the quantity $|V(v) \setminus N(x)| - |V(v) \cap N(x)|$ and we color them red (recall that u as at least 3 children). At the end of these steps, the editing H filling the red mixed children of u and making its blue mixed children hollow realises the minimum number of edits among the editings settled at u and satisfying conditions of Lemma 7, i.e. the minimal editings whose insertion node is u.

The cost of the minimum completion H settled at node u is computed as $mincost(u)$ = cost-above$(u) + \sum_{v \in red(u)} |V(v) \setminus N(x)| + \sum_{v \in blue(u)} |V(v) \cap N(x)|$ where $red(u)$ is the set of mixed children to be filled in H (colored red) and $blue(u)$ is the set of mixed children to be made hollow in H (colored blue). As our algorithm visits all the minimal insertion nodes u, it is able to determine the minimum of the values computed for $mincost(u)$ along the search, which is the minimum number of edits among all the cograph editings of $G + x$.

Complexity.

For the first step of the algorithm, all the information we need can be computed by seraching the tree top-down from the leaves that are neighbour of x until the root of the tree. Therefore, the nodes of u we encounter during the search are exactly the non-hollow nodes of u. Furthermore, all the treatments we need to perform on a node u take a time $O(|\mathcal{C}_{nh}(u)|)$. Then, the total time spent by such a search of the tree is $O(|NH|)$ and this is also the complexity of the first step of the algorithm.

In the second step of the algorithm, all the tests we need to perform on a node u can be done in $O(|\mathcal{C}_{nh}(u)|$, because we never manipulate the hollow children of one node. We only need to determine their number, which can be done by counting the number of non-hollow

nodes instead. The computation of cost-above(u) can be done in $O(1)$ time by noting that the sums $\sum_{u'\in\mathcal{C}(v)\setminus\{u\}} |V(u')\setminus N(x)|$ and $\sum_{u'\in\mathcal{C}(v)\setminus\{u\}} |V(u')\cap N(x)|$can rather be computed as $|V(v)\setminus N(x)| - |V(u)\setminus N(x)|$ and $|V(v)\cap N(x)| - |V(u)\cap N(x)|$ respectively. Finally, for a minimal insertion node u, once we have determined which of the mixed children of u should be filled and which of them should be made hollow in an editing H settled at u and realising the minimum number of changes, $cost(u)$ can be computed very easily in $O(|\mathcal{C}_{nh}(u)|$. Then one key issue for preserving linear complexity is to be able to determine which children to fill and which should be made hollow in H in $O(|\mathcal{C}_{nh}(u)|$ time. If u has only 2 children, this is clear as there are only two editings settled at u (see algorithm above), and for each of them their cost can be computed in constant time. In the case where u as at least 3 children, the algorithm works in three steps. In the first step, it parses all the mixed children v and uses the sign of the quantity $|V(v)\cap N(x)| - |V(v)\setminus N(x)|$ to decide for each of them, in $O(1)$ time, whether they should be made full or hollow in H, by coloring them red or blue. During this search which takes $O(|\mathcal{C}_{nh}(u)|$, one can also determine the minimum value of the quantities $|V(v)\cap N(x)| - |V(v)\setminus N(x)|$ and its two maximum values. This information allows to perform, in constant time, the at most 2 corrections made on the colors assigned to mixed children of u during the second and the third step. Overall, the three steps to determine one minimum cost editing settled at u execute in $O(|\mathcal{C}_{nh}(u)|$ time, as all the other treatments performed on a mixed node u. Consequently, the total complexity of the second step of the algorithm is $O(\sum_{u\in T \text{ mixed}} |\mathcal{C}_{nh}(u)|) = O(|NH|)$.

Once the two steps of the algorithm are over, we have determined the insertion node u and the set $filled(u)$ of children of u that will be full in H (namely, the children of u that are full in $G+x$ and the mixed children of u in $G+x$ that have been colored red), the others become hollow. To finish one incremental step, we just need to update the tree. First, we insert x in the tree as done in the algorithm of [17], this part of their algorithm takes $O(|filled(u)|) = O(|\mathcal{C}_{nh}(u)|)$. Then, for all the ancestors of x in the new tree T_H, we need to update their number of children, this takes $O(NH)$ as all the ancestors of x in T_H are mixed nodes in T. Finally, the updating step has complexity $O(|NH|)$ and this is the whole complexity of one incremental step of the algorithm.

Let us note that, unfortunately, $|NH|$ is not dominated by the degree d of x, as one unique neighbour of x can gives rise to a branch of non-hollow nodes of T that has size $\Omega(n)$. Therefore, for expressing the complexity of the whole editing algorithm, we can only observe that $O(|NH|) = O(n)$ and consequently obtain an $O(n^2)$ total complexity.

5 An $O(n+m)$-time algorithm for minimal cograph editing

As observed above, the reason why the previous algorithm has a quadratic time complexity is that the part of the cotree that it needs to search at each incremental step may be up to $\Omega(n)$, indepently of what is the degree d of the new vertex x. On the positive side, despite of this, the minimum-cost insertion nodes are all located in some part of the tree containing mostly neighbours of x, and whose size is consequently bounded by $O(d)$. We formalise this intuition with the following definition.

Definition 7 (Preponderant nodes). *A node u of T is* preponderant *if $|V(u)\cap N(x)| \geq |V(u)\setminus N(x)|$ and u is a* maximal preponderant *node if u is preponderant and none of its strict ancestors in T is.*

The main observation we exploit in the $O(n+m)$-time algorithm we design in this section is the following.

Lemma 9. *For any node $v \in T$, if $mincost(v)$ is minimum among all nodes of T then there exists some maximal preponderant node u such that $v \in T_u$ or v is the parent of u.*

Proof. Consider the minimum editing settled at v. From Lemma 1, it makes some child v_f of v full. Then, v_f is necessarily a preponderant node, because otherwise, making v_f hollow would result in an editing consistent with v and having strictly smaller cost, which is impossible since $mincost(v)$ is minimum among all nodes of T. $\square$

Therefore, following Lemma 9, the first task of our algorithm will be to discover all the maximal preponderant nodes of T, by paying attention to search only their subtrees, whose total size is $O(d)$, by definition. We can then use the previous algorithm to determine what is the best editing settled in the subtree of each maximal preponderant node u. The total time needed to do so is only $O(\sum_{u \text{ max. preponderant}} |T_u|) = O(d)$. It remains that we need to compare the minimum editings obtained in each subtree of maximal preponderant node and to take into consideration the editings settled at their parents. This requires to search the part of the tree which is outside of the subtrees of preponderant nodes. Fortunately, we can restrict ourselves to searching again an $O(d)$-size part of T by exploiting the fact that we are interested only in editings that are not more costly than the *delete-all editing*, i.e. the one making x an isolated vertex in $G + x$. To express this condition more precisely, we first need some more definitions and notations.

We denote Γ the set of parents of the maximal preponderant nodes of T. The algorithm colours the neighbours of x in black and leaves the other vertices white. We denote $B(u) = |V(u) \cap N(x)|$ the number of black leaves of T_u and $W(u) = |V(u) \setminus N(x)|$ its number of white leaves. We denote diff-above(u) = cost-above$(u) - (d - B(u))$, which is the difference of cost, restricted to the adjacencies between x and $V(G) \setminus V(u)$, between any editing settled at u and the delete-all editing. For a node $u \in T$, we denote $\mathcal{C}_{prep}(u)$ its set of preponderant children and we denote $B_{prep}(u) = \sum_{v \in \mathcal{C}_{prep}(u)} B(v)$ and $W_{prep}(u) = \sum_{v \in \mathcal{C}_{prep}(u)} W(v)$. Similarly, we denote $\mathcal{C}_{min}(u)$ the set of non-preponderant children of u and $B_{min}(u) = \sum_{v \in \mathcal{C}_{min}(u)} B(v)$ and $W_{min}(u) = \sum_{v \in \mathcal{C}_{min}(u)} W(v)$.

Deleting all the edges between the new vertex x and the vertices in $N(x)$ is always a valid cograph editing of $G + x$, which has cost d. Consequently, when we look for an editing of minimum cost, we can consider only these editings that have cost no more than d. From Lemma 9 above, we know that minimum editings are settled in the subtrees of maximal preponderant nodes or at their parents. Consider first the case of an editing settled at, or just consistent with, some node $w \in T_v$, where v is a maximal preponderant node. Compared to the delete-all editing, restricted to the edits between x and the vertices of $V(v)$, the editing consistent with w can save at most $B(v)$ edits, because this is the cost of the delete-all editing restricted to the vertices of $V(v)$. On the other hand, restricted to the edits between x and $V \setminus V(v)$, the editing consistent with w looses exactly diff-above(v) edits (which may be negative) compared to the delte-all editing. It follows that if the editing consistent with w is no more costly than the delete-all editing, then we must have diff-above$(v) \leq B(v)$. Consequently, the editings we are interested in are settled in the subtree T_v of some maximal preponderant node v satisfying this condition. For the editings consistent with the parent u of some maximal preponderant node, we also have a similar condition that writes diff-above$(u) \leq B_{prep}(u)$. Indeed, the minimum editing consitent with u makes all the preponderant children of u full and its non-preponderant children hollow. Then, restricted to the edits between x and vertices of $V(u)$, it can save at most $B_{prep}(u)$ edits compared to the delete-all editing, which makes all the children of u hollow. Again, if the cost of the minimum editing consistent with u is no more than d, then we must have diff-above$(u) \leq B_{prep}(u)$. Moreover, note that if some maximal preponderant node v satisfies the condition diff-above$(v) \leq B(v)$, then its parent u necessarily satisfies

the condition diff-above(u) $\leq B_{prep}(u)$, because by definition diff-above(u) $\leq$ diff-above(v) and $B(v) \leq B_{prep}(u)$.

For this reason, in the rest of the algorithm we will concentrate on finding the nodes $u \in \Gamma$ that satisfy the condition diff-above(u) $\leq B_{prep}(u)$. Later, we will show how to determine which of their preponderant children satisfy the condition diff-above(v) $\leq B(v)$, which is much more easy when we know that u satisfies the previous condition. Concerning the $O(d)$ complexity, roughly speaking, it comes from the fact that we are able to check whether $u \in \Gamma$ satisfies the condition diff-above(u) $\leq B_{prep}(u)$ by a limited search of the branch between u and the root of T that takes only $O(B_{prep}(u))$ time. In addition, when this condition is satisfied, we can determine the exact value of diff-above(u) within the same time complexity. This allows us to determine the exact values of the minimum editings settled in the subtrees of preponderant children of u, among which we select the minimum. The outline of our algorithm is as follows.

General scheme of the algorithm. It is in three steps:

1. determine the maximal preponderant nodes of T,
2. for every node u that is the parent of some maximal preponderant node, determine whether diff-above(u) $\leq B_{prep}(u)$ and determine the exact value of diff-above(u) when this condition is satisfied.
3. for such nodes u, for each of their preponderant children u', determine one minimum settled editing for each node $v \in T_{u'}$.

Data structure. Each node u of the cotree T stores the number $|V(u)|$ of leaves in T_u. Together with T, we store a *factorising permutation* π of G (see [15]), which is the order in which the leaves are encountered in some depth-first search of T (say the search where the children of each node u are searched in the order they appear in the list of children of u). The main property of a factorising permutation π is that for any node $u \in T$, the leaves of T_u form an interval of π, denoted I_u. π is stored as a doubly-linked list and each node of T stores two pointers, one to the left bound of I_u in π and one to its right bound. Reciprocally, each cell y of π keeps two lists: the list of nodes u such that y is the left bound of I_u and the list of nodes u such that y is the right bound of I_u. Both of these lists are sorted according to the inclusion order on the intervals I_u, smaller intervals appearing first in the list. The predecessor and successor of a vertex y in π are denoted $pred(y)$ and $succ(y)$ respectively.

We now detail each of the three steps of our algorithm, plus a preprocessing step that is performed before Step 2.

5.1 Step 1: finding maximal preponderant nodes

We perform this task thanks to the factorising permutation π. As for nodes, for an interval I of π, we denote $B(I)$ and $W(I)$ its number of black vertices and white vertices respectively and we say that I is preponderant if $B(I) \geq W(I)$. The left bound of interval I is denoted $l(I)$ and its right bound $r(I)$. We first determine a family $\mathcal{I}$ of pairwise disjoint intervals of π whose union contains all the preponderant intervals of π.

Definition 8. *A* direct interval *I is an interval such that $l(I)$ is black and $r(I)$ is the leftmost position on the right of $l(I)$ such that $B(I) = W(I)$. Simetrically, an* indirect interval *I is an interval such that $r(I)$ is black and $l(I)$ is the rightmost position on the left of $r(I)$ such that $B(I) = W(I)$. A maximal direct interval (resp. maximal indirect interval) is a direct (resp. indirect) interval which is maximal for inclusion among all direct (resp. indirect) intervals.*

Remark 1. Any black vertex belong to some direct interval and to some indirect interval.

Remark 2. For any vertex a in some direct (resp. indirect) interval I such that $a \neq l(I)$ (resp. $a \neq r(I)$), we have $B([\![a, r(I)]\!]) < 0$ (resp. $B([\![l(I), a]\!]) < 0$).

Remark 3. The direct intervals do not overlap and neither do the indirect intervals. Consequently, the maximal direct intervals are pairwise disjoint and so are the maximal indirect intervals.

Lemma 10. *Any preponderant interval I is included in the union of some direct and indirect intervals.*

Proof.

Assume for contradiction that there exists some vertex $a \in I$ that is neither in a direct interval nor in an indirect interval. Note that necessarily, from Remark 1, a is white. Then, the interval $[\![l(I), pred(a)]\!]$ (possibly empty), where $pred(a)$ is the predecessor of a in π, is the union of some white intervals, some maximal direct intervals and possibly the right piece of a maximal direct interval. Therefore, $B([\![l(I), pred(a)]\!]) \leq W([\![l(I), pred(a)]\!])$. Symetrically, we have $B([\![succ(a), r(I)]\!]) \leq W([\![succ(a), r(I)]\!])$. Since a is white, we obtain $B(I) < W(I)$, a contradiction with the fact that I is a preponderant interval.

□

The algorithm to find all the maximal preponderant nodes of T is in four steps: i) it computes all the maximal direct intervals and all the maximal indirect intervals, ii) it makes the union of them and obtain an interval partition $\mathcal{I}$ of this union, iii) for each interval $I \in \mathcal{I}$, it searches for the nodes of the tree that have their interval entirely included in I and iv) for each such node it determines whether it is preponderant or not and whether it is maximal or not.

In step i), for identifying maximal direct intervals, build the list L of all the black vertices that are preceeded by a white one. Start a scan of π (the factorising permutation) from each of the vertex y in L, moving toward the right of π. In this scan, maintain the difference diff between the number of black vertices and the number of white vertices encountered so far in the scan and stop the scan when diff becomes null. During this scan, mark all the vertices encountered with a green mark. At the end of the scan, put one pointer from the cell of y to the rightmost cell reached by the scan from y (the cell where diff became 0). Withdraw y from L and place it in the list M of the left bounds of maximal direct intervals found so far. When the vertex y considered in L is already marked in green, do not perform its scan and simply remove y from L. During a scan, if a green-marked vertex y' is encountered, jump directly to the cell pointed by its associated pointer and remove y' from the list M (the direct interval starting at y' is no longer maximal in the ones we have found, it is included in the direct interval of y being currently scanned). At the end, list M exactly contains the vertices y such that the direct interval D_y starting at y is maximal and we have a pointer to the last vertex of this interval. Since we do not scan twice a same part of π (thanks to the green marks), the total time taken by all these scans is $O(\sum_{y \in M} |D_y|) = O(d)$ since the D_y are disjoints (see Remark 3) and each of them is preponderant. Proceed in the same way to obtain the list of all maximal indirect intervals, but use a distinct marking color, say yellow marks instead of green marks.

For step ii), we can procced as follows. We denote M_{dir} and M_{ind} the lists of maximal direct intervals and maximal indirect intervals respectively. We start from any interval in $M_{dir} \cup M_{ind}$ and extend it to find the maximal union of intervals of $M_{dir} \cup M_{ind}$ containing it. Let say we start from a maximal direct interval I, which we immediately withdraw from M_{dir}, and we first try to extend it to the right. Notice that since the maximal indirect intervals are pairwise disjoint, there exists at most one of them that overlaps I on the right of I. In order to try to

discover it, we scan I from left to right and everytime we encounter the left bound of some maximal indirect interval J, we remove J from M_{ind} and we check whether the right bound of J belongs to I or not. If it does not belong to I, we change the current interval $I_{cur} = I$ to be extended to the right for the new interval $I_{cur} = [\![succ(r(I)), r(J)]\!]$. Then we try to extend I_{cur} to the right again, but with a direct interval instead of an indirect one. We continue this process until we cannot extend the current interval I_{cur} to the right anymore. Then, we come back to $I_{cur} = I$ and we now try to extend I_{cur} to the left, until it is not possible anymore. When both the right extension process and the left extension process terminate, the intervals that we have discovered are exactly the intervals in the maximal overlapping union containing I. As long as at least one of M_{dir} and M_{ind} is not empty, we pick a new interval I in them and perform the extension process to compute the maximal overlapping union containing I. Then, we can check the cell preceding and following each overlapping union in order to merge those intervals that are next to each other without overlapping. At the end of this step, we obtain a set $\mathcal{I}$ of disjoint intervals that are unions of both direct and indirect intervals and that are maximal for this property.

In step iii), we consider an interval $I \in \mathcal{I}$ and we determine all the nodes of T whose interval is included in I. Let us recall that in the data structure, the left bounds of the intervals of the nodes of the cotree are stored explicitely in the cells of the factorising permutation, sorted by increasing right bound. We make a first scan of I and mark all its cells using pink color. In a second scan, for each cell, we go through its list of left bounds and check that the corresponding right bound is also in I (i.e. marked in pink). We stop to scan the list of left bounds the first time that we encounter a right bound out of I, because we are sure that all the other right bounds in this list will also be out of I, as the list is sorted by increasing right bound. At the end of the second scan, we obtain the set S of all the nodes that have their two bounds in I. Note that there are $O(B(I))$ vertices in I and that there are then $O(B(I))$ nodes in S.

For step iv), we perform a scan of I from left to right, in which we compute, for each cell y, the difference $\text{diff}(y)$ between the number of black vertices and white vertices encountered since the beginning of the scan until y. Then, for each node u in S, u is preponderant iff $B(I_u) - W(I_u) = \text{diff}(pred(l(I_u))) - \text{diff}(r(I_u)) \geq 0$. Finally, a last scan of I allows to determine among the nodes in D that are preponderant nodes, which one are maximal. All these computations take $O(B(I))$ time, because I has size $O(B(I))$.

At the end of these four steps, we obtain the set of all maximal preponderant nodes of T, in time $O(d)$.

5.2 Preprocessing step: assigning a shorcut to the parent of each maximal preponderant node

In this section we show how to decide for any non-preponderant node u whether $W(u) - B(u) \leq s$, where s is any positive integer, in time $O(\min\{W(u) - B(u), s\})$. This is done by counting the white leaves and the black leaves encountered in a (partial) depth-first search of the subtree T_u that performs a limited number of edge traversals, controlled by some parameter of the search. In order to achieve the desired $O(\min\{W(u) - B(u), s\})$ time complexity, we need to precompute some parts of the search and to keep track of the result of these precomputed searches by assigning a shorcut to each node in Γ. The routine that decides whether $W(u) - B(u) \leq s$, for any $u \in T$, and that assigns its shorcut to some $u \in \Gamma$ is the same. It is named `Search-tree` and it is described in Algorithm 1 below.

The shorcut we assign to $u \in \Gamma$ is the node on which stops the call to `Search-tree`$(u, exc(u))$, where $exc(u) = \sum_{u_i \in \mathcal{C}_{prep}(u)} (B(u_i) - W(u_i))$. These shorcuts are crucial later in order to obtain a linear complexity for our algorithm. Indeed, we will need to search some subtrees of T several

times and these shorcuts avoid to repeat some portions of these searches that would threaten the linear complexity. We now describe generically how Routine `Search-tree`(u, s) searches the subtree of any non-preponderant node u with a budget s.

Before performing any call to Routine `Search-tree`, we color in grey the parents u of maximal preponderant nodes of T and we assign them an excess $exc(u) = \sum_{u_i \in \mathcal{C}_{prep}(u)} (B(u_i) - W(u_i))$, as above. We also shorcut their list of children in order to exclude all preponderant children from it. We do this by redirecting the pointers in the list so that they jump over the ranges of consecutive preponderant children. The resulting list of children of u, containing only its non preponderant children, is denoted $\mathcal{C}'(u)$ and the first child and the last child in this list are denoted $first'(u)$ and $last'(u)$ respectively. The tree resulting from these modifications of the children lists is denoted T'.

Routine `Search-tree`(u, s) performs a depth-first search of the modified subtree T'_u of u, in which the number of edge traversals is controlled by the variable named *ttl* in Algorithm 1, which stands for *time to live*. Everytime an edge of T'_u is traversed, either upward or downward, the *ttl* is decreased by 1. and the search stops when the *ttl* becomes negative. The value of this paramater *ttl* is initially set at $2 + 5s$ (Line 1 of Algorithm 1), where s is the budget of the search. In addition to its initial budget s, the search also uses the excesses $exc(v)$ of the grey nodes v it encounters in order to parse their subtrees T_v. The goal of Routine `Search-tree`(u, s) is to return a pointer on the node of T_u where the search stopped, that is where the *ttl* reached a negative value for the first time, and the difference (stored in the variable *cpt* in Algorithm 1) between the number of black leaves and the number of white leaves encountered during the search before it stopped. In order to be able to resume the search later, the routine also returns the node visited at the previous step, just before it stopped. Indeed, knowing the current node of the search and the previous one is enough to determine unambiguously the next node to be visited by the depth-first search (this is the purpose of Function `Next`, see Algorithm 2). In the case where the *ttl* does not become negative, the routine stops when T_u has been entirely searched and returns a pointer to node u itself. In this case, the value of *cpt* at the end of the search is precisely $W(u) - B(u)$, which is what we aim at determining.

Initially, for all grey nodes u of T, *shortcut*(u), *precshort*(u) and *weightshort*(u) are all set to $\perp$. Routine `Search-tree`(u, s) (see Algorithm 1) mainly relies on the function `Next`(v, v_{prec}, cpt, ttl) (see Algorithm 2). The purpose of function `Next` is to determine the next node to be visited by the depth-first search and to update the *ttl* of the search and the difference *cpt* between the number of white leaves and the number of black leaves encountered during the search. To this purpose, it takes as parameter the current node $v \in T_u$ on which the search is, the preceeding node v_{prec} on which the search was at the previous step, the current *ttl* and the current difference of leaves *cpt*. One basic and important property of function `Next` is that it moves from v to a neighbour of v in T', either its parent or one of its children, except when v is a grey node. In this later case, `Next` moves to the other extremity of the shortcut of v, which can be any node in the subtree T'_u.

The following lemma states the validity of the way we use Routine `Search-tree` throughout the paper in order to decide whether $W(u) - B(u) \leq s$ for some non-preponderant node u of T.

Lemma 11. *After all the shorcuts of the nodes in Γ have been assigned, for any non-preponderant node u and any positive integer s, we have $W(u) - B(u) \leq s$ iff `Search-tree`(u, s) scans all the subtree of u and returns a value $cpt \geq -s$.*

Proof. It is not difficult to see that *cpt* exactly counts the difference between the number of black leaves and white leaves encountered in the part of the subtree of u that is scanned[2] by the

[2] Note that the subtrees of preponderant nodes do not actually need to be searched as we already know their number of black and white leaves.

Algorithm 1: `Search-tree`(u, s)

```
1  Pre-conditions: u ∈ Γ and u is not the root of T.
2  Post-conditions: returns a triple (end, prec, weight) which is made of the node end of T_u on which ends
     the search, the preceding node prec visited by the search and the difference weight between the number
     of white leaves and te number of black leaves encountered during the search.
3  begin
4      cpt ← 0; ttl ← 2 + 5s;
5      (v, v_prec, cpt, ttl) ← next(u, parent(u), cpt, ttl);
6      while ttl ≥ 0 and (v ≠ u or v_prec ≠ last'(u)) do
7          if (v_prec = parent(v) and color(v) = grey and shortcut(v) = ⊥) then
8              (shortcut(v), precshort(v), weightshort(v)) ← Search-tree(v, exc(v));
9          end
10         (v, v_prec, cpt, ttl) ← next(v, v_prec, cpt, ttl);
11     end
12     (end, prec, weight) ← (v, v_prec, cpt);
13     return (end, prec, weight);
14 end
```

call to `Search-tree`(u, s). Therefore, if `Search-tree`(u, s) goes through all the subtree of u, the statement of the lemma clearly holds. The difficulty here is to show that conversely, when the search terminates before having entirely scanned the subtree of u, then we have $W(u)-B(u) > s$.

Let A be the set of grey nodes in T'_u and let $A' \subseteq A$ be the subset of nodes $v \in A$ such that the call to `Search-tree`$(v, exc(v))$ that assigns its shorcut to v entirely discovers the tree T'_v, i.e. $shortcut(v) = v$ and $precshort(v) = last'(v)$. Let $\tilde{A} \subseteq A'$ be the nodes of A' that are maximal in T' and let us denote $A^+ \subseteq A$ the subset of nodes in A that are not a strict descendant of a node in $\tilde{A}$. Note that by definition $\tilde{A} \subseteq A^+$. In the rest of the proof, we consider only the case where $u \notin \tilde{A}$ since otherwise the `Search-tree`(u, s) discovers all T'_u and the statement directly holds as explained above.

Observe that by construction of Routine `Search-tree`, a call to `Search-tree`(u, s) terminates on the same node of T_u independently of whether the shorcut of nodes $v \in A^+$ have been assigned before the time when the call is made or not. Therefore, we can choose to assign their shorcut to the nodes in $\tilde{A}$ and not to assign their shorcut to the nodes in $A^+ \setminus \tilde{A}$. Now, uncolor the nodes of $A^+ \setminus \tilde{A}$ (that were colored grey by definition) and observe that `Search-tree`(u, s) in T', which runs with an initial ttl value of $2 + 5s$, terminates on the same node as the call to `Search-tree` in the uncolored version of T'_u with an initial ttl value of $2 + 5s + SUM$, where $SUM = \sum_{v \in A^+ \setminus \tilde{A}} (2 + 5exc(v))$. We denote $\tilde{T}$ the modification of T' where all the strict descendants of nodes in $\tilde{A}$ have been removed, i.e. nodes of $\tilde{A}$ become leaves in $\tilde{T}$. It should be clear that the minimum initial value of ttl needed for a call to `Search-tree`(u, s) in order to entirely scans the uncolored version of T'_u, where nodes of $\tilde{A}$ have been assigned their shorcut and nodes of $A^+ \setminus \tilde{A}$ have not, is exactly $2|E(\tilde{T})| + |\tilde{A}|$. Therefore, in order to prove the lemma, we just need to prove that $2|E(\tilde{T})| + |\tilde{A}| \le 2 + 5(W(u) - B(u)) + SUM$, which, in the case where $W(u) - B(u) \le s$, implies that $2|E(\tilde{T})| + |\tilde{A}| \le 2 + 5s + SUM$.

First, note that the internal nodes of $\tilde{T}$ have at least two children, except the grey nodes in $A^+ \setminus \tilde{A}$, which may have only one child because their preponderant children have been removed to obtain tree T'. Denoting $\tilde{L}$ the set of leaves of $\tilde{T}$, we then have $|E(\tilde{T})| \le |A^+ \setminus \tilde{A}| + 2|\tilde{L}|$. Now, observe that $\tilde{L}$ contains only leaves of T', which are all white, and the nodes in $\tilde{A}$. Therefore, we have $W(u) - B(u) = |\tilde{L} \setminus \tilde{A}| - \sum_{v \in A^+ \setminus \tilde{A}} exc(v) + \sum_{v \in \tilde{A}} (W(v) - B(v)) \ge |\tilde{L} \setminus \tilde{A}| - \sum_{v \in A^+ \setminus \tilde{A}} exc(v) + |\tilde{A}| = |\tilde{L}| - \sum_{v \in A^+ \setminus \tilde{A}} exc(v)$, which gives $|\tilde{L}| \le W(u) - B(u) + \sum_{v \in A^+ \setminus \tilde{A}} exc(v)$. Combining this with $|E(\tilde{T})| \le |A^+ \setminus \tilde{A}| + 2|\tilde{L}|$, we get $2|E(\tilde{T})| + |\tilde{A}| \le 2|A^+ \setminus \tilde{A}| + 4(W(u) - B(u)) + 4\sum_{v \in A^+ \setminus \tilde{A}} exc(v) + |\tilde{A}|$. As we also have $|\tilde{A}| \le \sum_{v \in \tilde{A}} (W(v) - B(v)) = W(u) - B(u) +$

Algorithm 2: `Next`(v, v_{prec}, cpt, ttl)

```
1  Pre-conditions: v_prec ∈ C(v) ∪ {parent(v)}.
2  begin
3      if v is a leaf or v_prec = last'(v) then
4          v_next ← parent(v);
5      end
6      else
7          if v = parent(v_prec) then
8              v_next ← succ_C'(v)(v_prec);
9              if v_next is a leaf then cpt ← cpt − 1;
10         end
11         else
12             if color(v) = grey and shortcut(v) ≠ ⊥ then
13                 cpt ← cpt + weightshort(v); v_next ← shortcut(v); v ← precshort(v);
14             end
15             else
16                 v_next ← first'(v);
17                 if v_next is a leaf then cpt ← cpt − 1;
18             end
19         end
20     end
21     ttl ← ttl − 1;
22     return (v_next, v, cpt, ttl);
23 end
```

$\sum_{v\in A^+\setminus\tilde{A}} exc(v) - |\tilde{L}\setminus\tilde{A}| \le W(u) - B(u) + \sum_{v\in A^+\setminus\tilde{A}} exc(v)$, we finally obtain $2|E(\tilde{T})| + |\tilde{A}| \le 5(W(u) - B(u)) + 2|A^+\setminus\tilde{A}| + 5\sum_{v\in A^+\setminus\tilde{A}} exc(v) = 5(W(u) - B(u)) + SUM \le 2 + 5(W(u) - B(u)) + SUM$.

□

Complexity of the preprocessing step. As mentionned above, the shorcut assigned to node u is the node $end \in T_u$ on which ends the call to `Search-tree`$(u, 2 + 5exc(u))$. Because Routine `Search-tree` recursively calls itself (Line 8 of Algorithm 1) on all the nodes in Γ that it encounters and that have not been searched yet, at the end of the call to Routine `Search-tree`$(u, 2 + 5exc(u))$ that we use to assign its shorcut to u, the shorcuts of all the nodes in $\Gamma \cap T_u$ are assigned. Therefore, in order to assign their shorcuts to all the nodes in Γ, we start with a list containing all of them and we call Routine `Search-tree` on the first node in the list. Everytime one call to `Search-tree` is made on a node in Γ, this node is withdrawn from the list. When all calls that have been launched terminate, if the list is not empty then we call Routine `Search-tree` on its first element, until the list become empty and all the necessary shorcuts have been assigned.

The time spent by one call to `Search-tree`$(u, 2 + 5exc(u))$, if we exclude the time spent in the recursive calls that occur inside this one, is $O(2 + 5exc(u)) = O(exc(u))$ as the search stops after traversing $2 + 5exc(u)$ edges or shorcuts. Therefore, as each node $u \in \Gamma$ is searched only once, thanks to the shorcut that is assigned to it and that avoid to search T_u again in the subsequent calls to `Search-tree`$(v, 2 + 5exc(v))$ where v is an ancestor of u, the total time complexity of assigning their shorcuts of all the nodes in Γ is $O(\sum_{u\in\Gamma} exc(u)) = O(d)$.

Complexity of a call to `Search-tree`(u, t) after the preprocessing step. After all the shorcuts have been assigned to the nodes in Γ, a call to `Search-tree`$(u, 2 + 5s)$, where u is a

non-preponderant node, takes time $O(\min\{W(u) - B(u), s\})$. Indeed, after preprocessing, an execution of Routine `Search-tree` no longer calls itself recursively, it only follows the shorcuts that have already been assigned. Therefore, as observed before, the execution time of the routine is bounded by the initial value of the *ttl* variable, that is $2 + 5s = O(s)$. Moreover, from Lemma 11, if the call to `Search-tree`$(u, 2+5s)$ does not entirely discover the subtree of u then $W(u) - B(u) > s$ and we then have $O(s) = O(W(u) - B(u))$. On the other hand, when the search entirely spans T_u, as explained in the proof of Lemma 11, because of the shorcuts of the nodes in $\tilde{A}$, the search is contained into a part of T_u that is denoted $\tilde{T}$ in the proof. Moreover, we also proved that if the nodes in $A+\setminus\tilde{A}$ have no shorcuts, then the number of times the search needs to traverse an edge or a shorcut (of a node in $\tilde{A}$) is at most $5(W(u) - B(u)) + SUM$. As the shorcuts of nodes in $A^+ \setminus \tilde{A}$ save exactly $\sum_{v\in A^+\setminus\tilde{A}}(2 + 5exc(v)) = SUM$ edge traversals, then the number of edge and shorcut traversals that `Search-tree` actually uses to entirely scan T_u does not exceed $5(W(u) - B(u)) = O(W(u) - B(u))$. As this number is also always $O(s)$ (see above), the time complexity of a call to `Search-tree`$(u, 2+5s)$ after all the shorcuts have been assigned is $O(\min\{W(u) - B(u), s\})$.

5.3 Step 2: determining diff-above for the parents of maximal preponderant nodes

The purpose of this section is to determine for any parent u of some maximal preponderant node, whether it satisfies the condition diff-above$(u) \leq B_{prep}(u)$ and for any maximal preponderant node u, whether it satisfies the condition diff-above$(u) \leq B(u)$. For all those nodes, we also want to determine the exact value of diff-above(u). Before doing this, we explain why we can first focuss exclusively on the parents of maximal preponderant nodes, whose set is denoted Γ, and even restrict ourselves to a subset of them, denoted Γ^*, that we call *traceable*. We will need the following definitions that are extensions of the notions cost-above and diff-above that we used before.

Definition 9. *For $u \in T$ and $v \in Anc(u) \setminus \{u\}$, we denote $]u, v] = (Anc(u) \setminus \{u\}) \cap Desc(v)$ and we define:*
cost-above$_v(u) = \sum_{w\in Ser\cap]u,v]} \sum_{w'\in\mathcal{C}(w)\setminus Anc(u)} W(w') + \sum_{w\in Par\cap]u,v]} \sum_{w'\in\mathcal{C}(w)\setminus Anc(u)} B(w')$,
del-above$_v(u) = \sum_{w\in]u,v]} \sum_{w'\in\mathcal{C}(w)\setminus Anc(u)} B(w')$ and
diff-above$_v(u) =$ cost-above$_v(u) -$ del-above$_v(u) = \sum_{w\in Ser\cap]u,v]} \sum_{w'\in\mathcal{C}(w)\setminus Anc(u)}(W(w') - B(w'))$.

Focussing on the set Γ of the parents of maximal preponderant nodes. As explained above, our goal is to determine diff-above both for the maximal preponderant nodes u_i such that diff-above$(u_i) \leq B(u_i)$ and for their parents u such that diff-above$(u) \leq B_{prep}(u)$. We can first focuss on achieving this task only for the parents u of maximal preponderant nodes, because from the result for u we can deduce the result for its preponderant children u_i.

Indeed, if diff-above$(u) > B_{prep}(u)$ then we have diff-above$(u_i) > B(u_i)$, because diff-above$(u_i) \geq$ diff-above(u) and $B_{prep}(u) \geq B(u_i)$. Therefore if some node u in Γ does not satisfy the condition required for the parents of maximal preponderant nodes, namely diff-above$(u) \leq B_{prep}(u)$, then its preponderant children u_i do not satisfy the condition required for the maximal preponderant nodes, namely diff-above$(u_i) \leq B_(u_i)$.

On the other hand, if a node $u \in \Gamma$ satisfies the condition diff-above$(u) \leq B_{prep}(u)$ and if we have then determined the value of diff-above(u), we can also determine diff-above(u_i) for all its preponderant children u_i by observing that diff-above$(u_i) =$ diff-above$_u(u_i) +$ diff-above(u). If u is a parallel node, diff-above$_u(u_i) = 0$ and we immediately get the value of diff-above(u_i) as it is equal to diff-above(u). If u is a series node, this requires more computation, but it can be done within the linear complexity we aim at, as follows.

Using the notations we set above, we have diff-above$_u(u_i) = W_{min}(u) + W_{prep}(u) - W(u_i) - (B_{min}(u)+B_{prep}(u)-B(u_i))$. In this quantity, everything is known except $W_{min}(u)$ and $B_{min}(u)$. We want to determine whether diff-above$(u_i) \leq B(u_i)$, which writes, with what precedes, $W_{min}(u) + W_{prep}(u) - W(u_i) - (B_{min}(u) + B_{prep}(u) - B(u_i)) \leq B(u_i)$. This gives $W_{min}(u) - B_{min}(u) \leq B_{prep}(u) - W_{prep}(u) + W(u_i)$. Since $W(u_i) \leq W_{prep}(u)$, we obtain $W_{min}(u) - B_{min}(u) \leq B_{prep}(u)$, which is a necessary condition for diff-above$(u_i) \leq B(u_i)$. Consequently, we first check this necessary condition by searching, thanks to Routine `Search-tree`, the subtrees of the non-preponderant children of u with a global budget of $B_{prep}(u)$ (the subtree of each non-preponderant child of u is searched with the budget remaining from the previous searches). If the condition does not hold, then diff-above$(u_i) > B(u_i)$ for all the preponderant children u_i of u. As a consequence, as explained above, we can safely discard all the editings settled in the subtrees of the preponderant children of u, as none of them has a cost smaller than the delete-all editing. On the other hand, if the condition $W_{min}(u) - B_{min}(u) \leq B_{prep}(u)$ holds, then we have searched entirely the subtrees of all the non-preponderant children of u and consequently, we have determined the value of $W_{min}(u)$ and $B_{min}(u)$. We can then determine the exact value of diff-above(u_i) for each preponderant child u_i of u and keep only those for which the condition diff-above$_u(u_i) \leq B(u_i)$ is satisfied, if any.

Considering only the subset Γ^* of the traceable nodes in Γ. For complexity reason, we will not be able to determine all the nodes u in Γ that satisfy the condition diff-above$(u) \leq B_{prep}(u)$. We will discard some of them, for which we know for sure that they have an ancestor v such that the minimum editing consistent with v is less costly than any editing settled at u or in the subtree of some preponderant child of u. These nodes that we will discard are called *non traceable*, as defined below. They can be discard safely from the set of nodes in Γ we consider as none of the editings settled at such nodes or in the subtrees of their preponderant children is minimum.

Definition 10 (Traceable). *A node $u \in \Gamma$ is* traceable *iff for all ancestors v of u, diff-above$_v(u) \leq B_{prep}(u)$.*

From the definition, one can immediately see that all taceable nodes satisfy the condition diff-above$(u) \leq B_{prep}(u)$. We now prove the property we use in order to discard non traceable nodes.

Lemma 12. *For $u \in \Gamma$ a non traceable node, if there exists $u' \in \{u\} \cup \bigcup_{u_i \in \mathcal{C}_{prep}(u)} V(T_{u_i})$ such that $mincost(u') < d$, then there exists $v \in \Gamma \cap Anc(u) \setminus \{u\}$ such that for any $u' \in \{u\} \cup \bigcup_{u_i \in \mathcal{C}_{prep}(u)} V(T_{u_i})$, we have $mincost(u') > mincost(v)$.*

Proof. Let w be a necessarily strict ancestor of u such that diff-above$_w(u) > B_{prep}(u)$. Observe that a minimum editing $H_{u'}$ consistent with u' always make all the non-preponderant children of u hollow. Then, compared to the delete-all editing, and restricted to the subset of vertices $V(u)$, $H_{u'}$ can save at most $B_{prep}(u)$ edits. Moreover, since diff-above$_w(u) > B_{prep}(u)$, then, restricted to the subset of vertices $V(w)$, $H_{u'}$ is more costly than the editing $\hat{H}_w$ consistent with w that makes all the children of w, and so w itself, hollow. By definition, $\hat{H}_w$ is at least as costly as the minimum editing H_w consistent with w, which makes all the preponderant children of w full and all the non-preponderant children of w hollow. It follows, that $cost(H_{u'}) > mincost(w)$. If w has at least one preponderant child, then $v = w$ belongs to Γ and then satisfies the statement of the lemma.

On the other hand, if w has no preponderant child, its minimum consistent editing H_w makes all the children of w, and so w itself, hollow. This means that restricted to the vertices

of $V(w)$, H_w is the same as the delete-all editing. But we know from the hypothesis of the lemma that there exists some $u' \in \{u\} \cup \bigcup_{u_i \in \mathcal{C}_{prep}(u)} V(T_{u_i})$ such that $mincost(u') < d$. As we showed previously that $mincost(u') > mincost(w)$, this implies that $cost(w) < d$. Consequently, it must be that diff-above$(w) < 0$. From the definition of diff-above, this gives that there exists a series ancestor w' of w such that $\sum_{w'' \in \mathcal{C}(w') \setminus Anc(u)} (W(w'') - B(w''))$. Then, w' has at least one preponderant child and so $w' \in \Gamma$ and $w' \neq w$. Take the lowest ancestor v of w that belongs to Γ. v may be series or parallel, but in any case, the minimum editing H_v consistent with v cannot be more costly than H_w, because H_w either makes all the children of v not ancestor of w hollow or all of them full, while H_v makes the preponderant children of v full and the others hollow. Consequently, we have $mincost(v) \leq mincost(w) < mincost(u')$, which ends the proof. □

Let us denote T_u^{prep} the tree made of u and all the subtrees of preponderant children of u. In other words, Lemma 12 states that any non traceable node $u \in \Gamma$ such that T_u^{prep} contains some minimum consistent editing better than the delete-all editing has some ancestor v such that $mincost(v)$ is less than any minimum consistent editing in T_u^{prep}. This directly implies that u also has such an ancestor v that is traceable. This is the reason why we can safely discard a non traceable node in Γ when we identify one: its tree T_u^{prep} does not contain any minimum-cost insertion node.

Searching up the branch of $u \in \Gamma$. From now, we concentrate on the core of the problem, which is determining for which parents u of maximal preponderant nodes we have diff-above$(u) \leq B_{prep}(u)$ and in the positive, determining the exact value of diff-above(u). This is done by searching, upward in the tree, the branch from u to the root of T, for each u parent of some maximal preponderant node. This search stops either when:

1. we encounter an ancestor v of u such that diff-above$_v(u) > B_{prep}(u)$; then we can discard u as the cost of the editing settled at any node of T_u is greater than the cost of some editing settled at some ancestor of v.
2. we encounter an ancestor v of u such that v is the parent of some maximal preponderant node, i.e. $v \in \Gamma$, and $B_{prep}(v) > B_{prep}(u) -$ diff-above$_v(u)$; then, for complexity reasons, we stop the search from u and instead starts the search from v. Afterwards, when we have determined whether v satisfy the condition diff-above$(v) \leq B_{prep}(v)$, we can decide whether u satisfies this condition.
3. we reach the root of the tree; then we have diff-above$(u) \leq B_{prep}(u)$ and we will check afterwards whether there exist some node w in T_u such that the cost of the editing settled at w is at most d and we will determine the set of all such nodes $w \in T_u$.

The main idea of the search is contained in Rules 3 and 1 above. When the search terminates because of Rule 3, this is a success: u satisfies the condition diff-above$(u) \leq B_{prep}(u)$. On the opposite, when the search terminates because of Rule 1, this is a fail as u is not traceable, so we discard it. Rule 2 is used for complexity reasons. In this case, we cannot yet decide whether u satisfies the desired condition, but we will do this later thanks to the result of the search initiated at v, which will preserve the $O(d)$ time complexity.

We perform the search as follows. We use a budget *bud* that is initialesd on node u with the value $bud = B_{prep}(u)$. Then, we climb the branch from u to the root r of T, the current node of the search being denoted v. Along the search we maintain *bud* such that its value on node v is always $B_{prep}(u) -$ diff-above$_v(u)$ (note that this equality holds initially when $v = u$). To this purpose, when the search moves from node v to node $p = parent(v)$, we update *bud* by withdrawing from it the quantity diff-above$_p(v)$. If p is a parallel node then diff-above$_p(v) = 0$

and the value of *bud* remains unchanged. If p has at least one preponderant node, then we check whether $bud \geq B_{prep}(p)$. In the positive, the search started from u continues, otherwise it stops and u is made a child of p in an auxiliary forest F, whose role is explained later.

If p is a series node, we have diff-above$_p(v) = W_{prep}(p) - B_{prep}(p) + \sum_{v_i \in \mathcal{C}_{min}(p)\setminus\{v\}}(W(v_i) - B(v_i))$, as v is necessarily a non-preponderant child of p. We distinguish two cases. First, if p has no preponderant children then diff-above$_p(v)$ writes diff-above$_p(v) = \sum_{v_i \in \mathcal{C}_{min}(p)\setminus\{v\}}(W(v_i) - B(v_i))$ and the search started from u will continue iff the budget remaining on v is non-negative, i.e. $bud -$ diff-above$_p(v) \geq 0$. This also writes $bud \geq \sum_{v_i \in \mathcal{C}_{min}(p)\setminus\{v\}}(W(v_i) - B(v_i))$. In order to determine whether this condition holds, we search the subtrees of the non-preponderant children of p distinct from v thanks to the routine `Search-tree` with a total budget of *bud*. This means that we start the search of the first non preponderant child of p with a budget *bud* and the searches of the other children with the budget remaining from the previous searches. This allows to determine in time $O(bud)$ whether the budget remaining on p, namely $bud -$ diff-above$_p(v)$ is non-negative. If this holds, then *bud* is updated to this new value, otherwise, the search started from u stops and u is discarded.

Now, we examine the case where p has at least one preponderant child. There are then two conditions for the search started from u to continue. First, as previously, the remaining budget on p must be non-negative, and second, this remaining budget must be at least $B_{prep}(p) - W_{prep}(p)$. The second condition is stronger and writes $bud - (W_{prep}(p) - B_{prep}(p) + \sum_{v_i \in \mathcal{C}_{min}(p)\setminus\{v\}}(W(v_i) - B(v_i))) \geq B_{prep}(p) - W_{prep}(p)$, i.e. $bud \geq \sum_{v_i \in \mathcal{C}_{min}(p)\setminus\{v\}}(W(v_i) - B(v_i))$. In order to check this condition, again, we simply have to search the subtrees of the non-preponderant children of p distinct from v with a total budget of *bud*, which we do using Routine `Search-tree`. The time needed for this search is again $O(bud)$. If the search goes until the end, then the condition is fulfilled and as we completely searched all the subtrees of the non-preponderant children v_i of p distinct from v, we then know for each of them the value of $W(v_i) - B(v_i)$. We can therefore determine the value of diff-above$_p(v)$ and update the value of *bud*, as $bud \leftarrow bud - \sum_{v_i \in \mathcal{C}_{min}(p)\setminus\{v\}}(W(v_i) - B(v_i)) + (B_{prep}(p) - W_{prep}(p))$, in order to continue the search started from u. If the condition to continue is not fulfilled, then we stop the search started from u and make u a child of p in the auxiliary forest F that we already mentionned above.

Synchronising the searches initiated at nodes $u \in \Gamma$. The main idea for getting an $O(d)$ time complexity is to ensure that each black leaf will participate to the budget of only one search. This is true for the initial budgets $B_{prep}(u)$ of the searches. But unfortunately, when the next node p in the search is a series node with some preponderant child, an additional quantity $B_{prep}(p) - W_{prep}(p)$ is added to the current budget of the search. This means that this quantity $B_{prep}(p) - W_{prep}(p)$ will contribute to the budget of all the searches that were initiated at some node in the subtree of p and that continue after p, which threatens the linear complexity we aim at. Therefore, for preserving the complexity, instead of continuing all the searches that should continue beyond p, we only continue one of them, say f, that reached p with maximum remaining budget and we make the other searches children of this search f in F. In this way, the quantity $B_{prep}(p) - W_{prep}(p)$ contributes only to search f and the total time complexity of all the searches is bounded by $O(\sum_{u \in \Gamma} B_{prep}(u)) = O(d)$.

There is an additional difficulty to be solved: we have to be sure that at the time we determine the search reaching p with maximum remaining budget, all the searches that will eventually reach p have done so already. To ensure this, we launch the searches in an order such that all the searches initiated at nodes $v \in Desc(u)$ are launched before the search initiated at node $u \in \Gamma$ is launched, i.e. we use an order σ that is a linear extension of the descendant relationship on nodes of Γ. In order to compute σ in $O(d)$ time, we need to discard some irrelevant nodes of Γ as before. Indeed, as we already noted, we can safely discard the nodes

of $u \in \Gamma$ that have an ancestor v such that diff-above$_v(u) > B_{prep}(u)$. So for all ancestors v of u, we must have diff-above$_v(u) \leq B_{prep}(u)$. Since diff-above$_v(u) \geq |(Anc(u) \cap Desc(v) \cap Ser) \setminus \Gamma| - \sum_{w \in (Anc(u) \setminus \{u\}) \cap Desc(v) \cap Ser} B_{prep}(w)$, where Ser is the set of series nodes of the cotree T, we obtain $|(Anc(u) \cap Desc(v) \cap Ser) \setminus \Gamma| \leq \sum_{w \in Anc(u) \cap Desc(v) \cap Ser} B_{prep}(w)$. We denote $\tilde{\Gamma}$ the subset of nodes $u \in \Gamma$ that satisfy this condition, these are the only relevant nodes for our purpose and we can safely discard the other nodes of Γ. Now, our goal is then to determine all the nodes of $\tilde{\Gamma}$ and the decendant relationship between them.

We proceed as follows. We place all the nodes $u \in \Gamma$ in a queue, denoted Q, and we set them active with an associated budget of $B_{prep}(u)$. Then, for each node u in Q, we search the branch of u up toward the root of T with a budget initially set at $B_{prep}(u)$. During this search we jump over paralallel nodes and consider only the series nodes. The current budget of the search is decreased by one everytime a series node not in Γ is encountered. The search stops when the budget become null or a node in Γ is encountered or the root of T is encountered. Then, node u is removed from Q and become inactive. If the search stopped on a node v in γ with a non-null budget, then u is made a child of v in an auxiliary forest denoted $\tilde{F}$ and we keep track of the remaining budget of the search when it reached v. When all the nodes in Q have been processed, i.e. queue Q is empty, we search each tree $\tilde{F}_i$ of the forest $\tilde{F}$ starting from its root. By this search, we determine what is the maximum Max_{rem}, over all the leaves l of $\tilde{F}_i$, of the sum of the remaining budgets of all nodes on the branch of $\tilde{F}_i$ between leaf l and the root of $\tilde{F}_i$. If $Max_{rem} > 0$, then the node $\tilde{u}$ on which stopped the search initiated at the root of F_i is made active and placed in queue Q with an associated budget Max_{rem}. When all the subtrees of $\tilde{F}_i$ have been treated and for each of them a new active node has been eventually placed in Q, a new round starts. As previously, for every node u in Q we perform the search initiated from u with its associated budget. If at the beginning of one round, Q is empty then the process stops. During this process, all the nodes encountered during the searches (including the parallel nodes over which we jump) are marked and each node maintain a list of its marked children. If the root of T is not marked, then $\tilde{\Gamma} = \varnothing$ and the only minimum editing is the one making x an isolated vertex. Otherwise, the nodes of $\tilde{\Gamma}$ are all marked, as well as all their ancestors. We can therefore discover all of them by searching the set of marked nodes accessible from the root. Overall, the complexity of this process is $O(d)$. Indeed, each neighbour of x contributes to the initial budget $B_{prep}(u)$ of exactly one search. Moreover, the budgets used in one round are the budgets remaining from the previous round. Therefore, the part of the tree T that is discovered has size $O(d)$.

Using the forest F. Afterwards, when the searches of all the branches are over, we use the auxiliary forest F to determine for each node u in F whether u satifies the condition diff-above$(u) \leq B_{prep}(u)$. We proceed as follows. We examine each tree F_i in F, starting from its root. If the root r_i of F_i was discarded during the algorithm, then we discard all the nodes in the tree F_i. Otherwise, the serach started from r_i has reached the root r of T and r_i satisfies the condition diff-above$(r_i) \leq B_{prep}(r_i)$. We then search the tree F_i starting from r_i and for every node v in F_i we determine whether it satisfies the condition diff-above$(v) \leq B_{prep}(v)$ and determine the value of diff-above(v) in the positive. If v does not satisfies this condition, then we discard all the nodes in the subtree rooted at v in F_i, beacuse none of them satisfies the condition. If v satisfies the condition, we check whether its children u in F_i do as well.

If v is a parallel node, this is straightforward: we just need to check whether $bud_u(v) \geq$ diff-above(v), where $bud_u(v)$ is the remaining budget when the search started from u reached v (i.e. $bud_u(v) = B_{prep}(u) -$ diff-above$_v(u)$). The key point here is that when v is a parallel node, we have already determined the value of $bud_u(v)$ at the moment when the search initiated at u

stopped at v. So we have all the necessary knowledge to test that the condition is satisfied for u.

When v is a series node, the search initiated at u stopped at v without determining the exact value of $bud_u(v)$. We only know that this value is less than $B_{prep}(v)$. We then have to determine whether it is non negative and determine it precisely in this case. Let v' be the child of v that is an ancestor of u. We have $bud_u(v) = bud_u(v') - \text{diff-above}_v(v')$, and $\text{diff-above}_v(v') = W_{prep}(v) - B_{prep}(v) + \sum_{v_i \in \mathcal{C}_{min}(v) \setminus \{v'\}}(W(v_i) - B(v_i))$. So the condition $bud_u(v) \geq 0$ also writes $bud_u(v') \geq W_{prep}(v) - B_{prep}(v) + \sum_{v_i \in \mathcal{C}_{min}(v) \setminus \{v'\}}(W(v_i) - B(v_i))$, that is $bud_u(v') + B_{prep}(v) - W_{prep}(v) \geq \sum_{v_i \in \mathcal{C}_{min}(v) \setminus \{v'\}}(W(v_i) - B(v_i))$. In order to determine whether this holds, we should search the subtrees of the non preponderant children of v in T that are different from v' with a total budget of $bud_u(v') + B_{prep}(v) - W_{prep}(v)$. But in order to keep a linear complexity, we cannot afford to do so for all the children u of v in F_i. Instead, we show that performing two slightly different searches is enough. First, we take the maximum value bud_{max} of $bud_u(v')$ among all children u of v in F_i. then, we come back to T and we search the subtrees rooted at the non preponderant children of v in T with a total budget of $bud_{max} + B_{prep}(v) - W_{prep}(v)$. Let us denote the non preponderant children of v in T as $v_1, v_2, \ldots, v_k$, indexed in the order in which we search them. Denote v_s the last child in this order that is searched by our searches (of total budget $bud_{max} + B_{prep}(v) - W_{prep}(v)$). Observe that necessarily, for all $v_i > v_s$, the search of the subtrees rooted at the nodes of $\mathcal{C}_{min}(v) \setminus \{v_i\}$ with total budget $bud_{u_i}(v_i) + B_{prep}(v) - W_{prep}(v)$ does not terminate, because it needs to search all the subtrees of $v_1, v_2 \ldots, v_s$ with a budget lower than the previous search (since $bud_{u_i}(v_i) \leq bud_{max}$). Now, perform the search with budget $bud_{max} + B_{prep}(v) - W_{prep}(v)$ considering the children of v in the reverse order $v_s, v_{s-1}, \ldots, v_1$ and denote v_t the last child that we reach. Again, for all $v_i < v_t$, the search corresponding to v_i does not terminate. Therefore, if $s < t$ then we can safely discard all the children of v in F_i as none of them satisfies the condition. On the opposite, if $t < s$ then, during the two searches, we have entirely searched all the subtrees rooted at the children v_i of v in T and for all of them we precisely know the values of $W(v_i)$ and $B(v_i)$, which is enough to determine which are the children of v in F_i that satisfy the desired condition. Finally, in the case where $v_s = v_t$, v_s is the only pretendent to satisfy the condition, which we can easily check as we know the values of $W(v_i)$ and $B(v_i)$ for all the other children of v in T.

Finally, after examining all the trees of the forest F, we obtain the set $\Gamma^* \subseteq \Gamma$ of all the parents u of some maximal preponderant nodes such that u satisfies the condition $\text{diff-above}(u) \leq B_{prep}(u)$ and for all $u \in \Gamma^*$, we know the exact value of $\text{diff-above}(u)$. The fundamental property of the set Γ^* is that all the minimal cograph editings of $G + x$ that have cost at most d and at most the cost of the minimum editings settled at their ancestors are settled either at some node $u \in \Gamma^*$ or at some node in the subtree of some preponderant child of some $u \in \Gamma^*$.

5.4 Step 3: listing the minimum settled editings in the subtree of u

Let us start with the editing settled at some node $u \in \Gamma^*$. We have $\text{cost}(u) - d = \text{diff-above}(u) + \text{cost}_{T_u}(u) - B(u)$, where $\text{cost}_{T_u}(u)$ is the cost restricted to $V(u)$ of the editing settled at u. As we have $\text{cost}_{T_u}(u) = B_{min}(u) + W_{prep}(u)$, this gives $\text{cost}(u) - d = \text{diff-above}(u) + B_{min}(u) + W_{prep}(u) - (B_{min}(u) + B_{prep}(u)) = \text{diff-above}(u) + W_{prep}(u) - B_{prep}(u)$. So the condition $\text{cost}(u) \leq d$ also writes $\text{diff-above}(u) \leq B_{prep}(u) - W_{prep}(u)$. This condition is easy to test as we already determined all the quantities in its expression. If the condition is satisfied we keep the editing settled at u in the output set of solutions and we obtain its cost as $d + \text{diff-above}(u) + W_{prep}(u) - B_{prep}(u)$.

We now turn to the editings settled at a node w in the subtree of some preponderant child u' of $u \in \Gamma^*$. Remember that we showed already how to determine whether $\text{diff-above}(u') \leq$

$B(u')$, so we consider only such children u' of u. Again, we have $\text{cost}(w) - d = \text{diff-above}(w) + \text{cost}_{T_w}(w) - B(w) = \text{diff-above}(u') + \text{diff-above}_{u'}(w) + \text{cost}_{T_w}(w) - B(w)$. Since we also have $\text{diff-above}_{u'}(w) = \text{cost-above}_{u'}(w) - (B(u') - B(w))$, we obtain $\text{cost}(w) - d = \text{diff-above}(u') + \text{cost-above}_{u'}(w) + \text{cost}_{T_w}(w) - B(u') = \text{diff-above}(u') + \text{cost}_{T_{u'}}(w) - B(u')$. As we want $\text{cost}(w) \leq d$, this gives $\text{cost}_{T_{u'}}(w) \leq B(u') - \text{diff-above}(u')$. We already know the exact value of $B(u')$ and $\text{diff-above}(u')$, so we can apply the algorithm of Section 4 to $G[V(u')] + x$ in order to determine all the minimal insertion nodes w of $T_{u'}$ and keep only those such that $\text{cost}_{T_{u'}}(w)) \leq B(u') - \text{diff-above}(u')$. This takes time $O(|V(u')|) = O(B(u'))$, since u' is a preponderant node. So overall, doing so for all the maximal preponderant nodes of T takes $O(d)$ time.

6 Conclusion and perspectives

We designed an $O(n+m)$-time algorithm for minimal cograph editing using the fact that there always exists an editing of cost at most m. This is not true for pure completion but also holds for pure deletion. Therefore, the most immediate question is whether the same approach can be used in order to obtain a linear-time algorithm for the minimal cograph deletion problem. More generally, our result, which, to the best of our knowledge, is the first algorithm for an inclusion-minimal editing problem, suggests that considering minimal editing instead of minimal completion, its more popular restricted version, may allow to design faster algorithms. Since at the same time, the editing problem is likely to provide smaller sets of edits than the completion problem, this possibility should be explored for other target classes of graphs as well.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discrete Methods 8(2), 277–284 (1987)
2. Bliznets, I., Fomin, F.V., Pilipczuk, M., Pilipczuk, M.: Subexponential parameterized algorithm for interval completion. In: Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1116–1131. SODA '16, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2016)
3. Böcker, S., Baumbach, J.: Cluster editing. In: The Nature of Computation. Logic, Algorithms, Applications. LNCS, vol. 7921, pp. 33–44. Springer (2013)
4. Böcker, S., Briesemeister, S., Klau, G.W.: Exact algorithms for cluster editing: Evaluation and experiments. Algorithmica 60(2), 316–334 (Jun 2011)
5. Böcker, S., Damaschke, P.: Even faster parameterized cluster deletion and cluster editing. Information Processing Letters 111(14), 717 – 721 (2011), http://www.sciencedirect.com/science/article/pii/S0020019011001232
6. Bodlaender, H., Downey, R., Fellows, M., Hallett, M., Wareham, H.: Parameterized complexity analysis in computational biology. Comput. Appl. Biosci. 11, 49–57 (1995)
7. Bodlaender, H., Kloks, T., Kratsch, D., Müller, H.: Treewidth and minimum fill-in on d-trapezoid graphs. J. Graph Algorithms Appl. 2(5), 1–23 (1998)
8. Bouchitt, V., Todinca, I.: Treewidth and minimum fill-in: Grouping the minimal separators. SIAM Journal on Computing 31(1), 212–232 (2001)
9. 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)
10. Broersma, H., Dahlhaus, E., Kloks, T.: A linear time algorithm for minimum fill-in and treewidth for distance hereditary graphs. Discrete Applied Mathematics 99(1-3), 367–400 (2000)
11. Bruckner, S., Hüffner, F., Komusiewicz, C.: A graph modification approach for finding core–periphery structures in protein interaction networks. Algorithms for Molecular Biology 10(1), 1–13 (2015)
12. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Information Processing Letters 58(4), 171–176 (1996)
13. Cao, Y.: Unit interval editing is fixed-parameter tractable. Information and Computation 253, 109 – 126 (2017)
14. Cao, Y., Marx, D.: Chordal editing is fixed-parameter tractable. Algorithmica 75(1), 118–137 (May 2016)
15. Capelle, C., Habib, M., de Montgolfier, F.: Graph decompositions and factorizing permutations. Discrete Mathematics and Theoretical Computer Science 5(1), 55–70 (2002)

16. Corneil, D., Lerchs, H., Burlingham, L.: Complement reducible graphs. Discrete Applied Mathematics 3(3), 163–174 (1981)
17. Corneil, D., Perl, Y., Stewart, L.: A linear time recognition algorithm for cographs. SIAM Journal on Computing 14(4), 926–934 (1985)
18. Crespelle, C., Paul, C.: Fully dynamic recognition algorithm and certificate for directed cographs. Discrete Applied Mathematics 154(12), 1722–1741 (2006)
19. Crespelle, C., Lokshtanov, D., Phan, T.H.D., Thierry, E.: Faster and enhanced inclusion-minimal cograph completion. In: 11th International Conference on Combinatorial Optimization and Applications (COCOA'17). LNCS, vol. 10627, Part I, pp. 210–224. Springer (2017)
20. 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
21. Crespelle, C., Todinca, I.: An $O(n^2)$-time algorithm for the minimal interval completion problem. Theor. Comput. Sci. 494, 75–85 (2013)
22. Dom, M., Guo, J., Huffner, F., Niedermeier, R.: Error compensation in leaf power problems. Algorithmica 44(4), 363–381 (2005)
23. 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)
24. Drange, P.G., Fomin, F.V., Pilipczuk, M., Villanger, Y.: Exploring the subexponential complexity of completion problems. ACM Trans. Comput. Theory 7(4), 14:1–14:38 (Aug 2015), http://doi.acm.org/10.1145/2799640
25. Drange, P.G., Pilipczuk, M.: A polynomial kernel for trivially perfect editing. Algorithmica (Dec 2017)
26. Fomin, F., Villanger, Y.: Subexponential parameterized algorithm for minimum fill-in. SIAM Journal on Computing 42(6), 2197–2216 (2013)
27. Goldberg, P., Golumbic, M., Kaplan, H., Shamir, R.: Four strikes against physical mapping of DNA. J. Comput. Biol. 2, 139–152 (1995)
28. 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)
29. Guo, J.: Problem kernels for NP-complete edge deletion problems: Split and related graphs. In: Tokuyama, T. (ed.) Algorithms and Computation. pp. 915–926. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
30. Hartuv, E., Shamir, R.: A clustering algorithm based on graph connectivity. Information Processing Letters 76(4), 175 – 181 (2000)
31. Heggernes, P., Mancini, F., Papadopoulos, C.: Minimal comparability completions of arbitrary graphs. Discrete Applied Mathematics 156(5), 705–718 (2008)
32. 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)
33. Heggernes, P., Mancini, F.: Minimal split completions. Discrete Applied Mathematics 157(12), 2659–2669 (2009)
34. Hellmuth, M., Fritz, A., Wieseke, N., Stadler, P.F.: Techniques for the cograph editing problem: Module merge is equivalent to editing P4s. CoRR abs/1509.06983 (2015)
35. Hellmuth, M., Wieseke, N., Lechner, M., Lenhof, H.P., Middendorf, M., Stadler, P.F.: Phylogenomics with paralogs. PNAS 112(7), 2058–2063 (2015)
36. Hüffner, F., Komusiewicz, C., Nichterlein, A.: Editing graphs into few cliques: Complexity, approximation, and kernelization schemes. In: Algorithms and Data Structures. LNCS, vol. 9214, pp. 410–421. Springer (2015)
37. 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)
38. Karp, R.: Mapping the genome: some combinatorial problems arising in molecular biology. In: 25th ACM Symposium on Theory of Computing (STOC 1993). pp. 278–285. ACM (1993)
39. Kendall, D.: Incidence matrices, interval graphs, and seriation in archeology. Pacific J. Math. 28, 565–570 (1969)
40. Kloks, T., Kratsch, D., Wong, C.: Minimum fill-in on circle and circular-arc graphs. Journal of Algorithms 28(2), 272–289 (1998)
41. Kloks, T., Kratsch, D., Spinrad, J.: On treewidth and minimum fill-in of asteroidal triple-free graphs. Theoretical Computer Science 175(2), 309–335 (1997)
42. Liu, K., Terzi, E.: Towards identity anonymization on graphs. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. pp. 93–106. SIGMOD '08, ACM, New York, NY, USA (2008)
43. Liu, Y., Wang, J., Guo, J., Chen, J.: Complexity and parameterized algorithms for cograph editing. Theoretical Computer Science 461, 45–54 (2012)

44. Lokshtanov, D., Mancini, F., Papadopoulos, C.: Characterizing and computing minimal cograph completions. Discrete Appl. Math. 158(7), 755–764 (2010)
45. Mancini, F.: Graph Modification Problems Related to Graph Classes. Ph.D. thesis, University of Bergen, Norway (2008)
46. Meister, D.: Treewidth and minimum fill-in on permutation graphs in linear time. Theoretical Computer Science 411(40-42), 3685–3700 (2010)
47. Nastos, J., Gao, Y.: Bounded search tree algorithms for parametrized cograph deletion: Efficient branching rules by exploiting structures of special graph classes. Discrete Math., Alg. and Appl. 4 (2012)
48. Nastos, J., Gao, Y.: Familial groups in social networks. Social Networks 35(3), 439 – 450 (2013)
49. Natanzon, A., Shamir, R., Sharan, R.: A polynomial approximation algorithm for the minimum fill-in problem. SIAM J. Comput. 30(4), 1067–1079 (2000)
50. Natanzon, A., Shamir, R., Sharan, R.: Complexity classification of some edge modification problems. Discrete Applied Mathematics 113(1), 109 – 128 (2001)
51. 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)
52. Rapaport, I., Suchan, K., Todinca, I.: Minimal proper interval completions. Inf. Process. Lett. 106(5), 195–202 (2008)
53. Rose, D.J.: A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In: Graph Theory and Computing. pp. 183–217 (1972)
54. Schoch, D., Brandes, U.: Stars, neighborhood inclusion and network centrality. In: SIAM Workshop on Network Science (2015)