

Practical and Efficient Split Decomposition via Graph-Labelled Trees

Emeric Gioan,¹ Christophe Paul,¹ Marc Tedder,² Derek Corneil²

Abstract

Split decomposition of graphs was introduced by Cunningham (under the name join decomposition) as a generalization of the modular decomposition. This paper undertakes an investigation into the algorithmic properties of split decomposition. We do so in the context of graph-labelled trees (GLTs), a new combinatorial object designed to simplify its consideration. GLTs are used to derive an incremental characterization of split decomposition, with a simple combinatorial description, and to explore its properties with respect to Lexicographic Breadth-First Search (LBFS). Applying the incremental characterization to an LBFS ordering results in a split decomposition algorithm that runs in time $O(n+m)\alpha(n+m)$, where α is the inverse Ackermann function, whose value is smaller than 4 for any practical graph. Compared to Dahlhaus' linear time split decomposition algorithm [16], which does not rely on an incremental construction, our algorithm is just as fast in all but the asymptotic sense and full implementation details are given in this paper. Also, our algorithm extends to circle graph recognition, whereas no such extension is known for Dahlhaus' algorithm. The companion paper [25] uses our algorithm to derive the first sub-quadratic circle graph recognition algorithm.

1 Introduction

Split decomposition ranks among the classical hierarchical graph decomposition techniques, and can be seen as a generalization of modular decomposition [21, 33, 28] and the decomposition of a graph into 3-connected components [41]. It was introduced by Cunningham and Edmonds [14, 15] as a special case of the more general framework of bipartitive families. Since then, a number of extensions and applications have been developed. For example, the decomposition scheme used in the proof of the Strong Perfect Graph Theorem [6] and in the recognition of Berge graphs [5] is based in part on the 2-join decomposition, which generalizes split decomposition. Also, clique-width theory [13] and rank-width theory [34] can be considered generalizations of split decomposition theory. Indeed, split decomposition is one of the important subroutines in the polynomial-time recognition of clique-width 3 graphs [11]. Moreover, the graphs of rank-width one are precisely the graphs that are totally decomposable by split decomposition (i.e. the distance-hereditary graphs [30] or completely-separable graphs [29]).

As with distance hereditary graphs [29], parity graphs can be characterized by their split decomposition [3, 8]. In [7], split decomposition is used to define a hierarchy of graph families between

¹CNRS - LIRMM, Univ. Montpellier II France; {emeric.gioan,christophe.paul}@lirmm.fr; financial support was received from the French ANR project ANR-O6-BLAN-0148-01: *Graph Decomposition and Algorithms* (GRAAL).

²Department of Computer Science, University of Toronto; {mtedder,dgc}@cs.toronto.edu; financial support was received from Canada's Natural Sciences and Engineering Research Council (NSERC).

distance hereditary and parity graphs. Split decomposition also appears in the recognition of circular arc graphs [31] and in structure theorems of various graph classes (see e.g. [40]). One of the more important applications of split decomposition is with respect to circle graphs; these are the intersection graphs of chords inscribing a circle. Prime circle graphs – those indecomposable by split decomposition – have unique chord representations (up to reflection) [1] (see also [12]). All of the fastest circle graph recognition algorithms are based on this fact [1, 20, 36]. Recent work has focused on their connection to rank-width and vertex-minors [2, 34]. For a brief introduction to split decomposition, the reader may refer to [37].

The first polynomial-time algorithm for split decomposition appeared in [14], and ran in time $O(nm)$. Ma and Spinrad later developed an $O(n^2)$ algorithm [32], which yields an $O(n^2)$ circle graph recognition algorithm when combined with their prime testing procedure in [36]. The only linear time algorithms for split decomposition are due to Dahlhaus [16] and, more recently, Montgolfier et al. [4]. However, so far neither of these linear time algorithms seems to extend to circle graph recognition. This paper develops a split decomposition algorithm that runs in time $O(n+m)\alpha(n+m)$, where α is the inverse Ackermann function [9, 38] (we point out that this function is so slowly growing that it is bounded by 4 for all practical purposes.³) Hence, there is essentially no running time tradeoff in using our algorithm. Moreover, the algorithm presented here is used by the companion paper [25] to derive the first sub-quadratic circle graph recognition algorithm.

Our algorithm benefits from the recent reformulation of split decomposition in terms of graph-labelled trees (GLTs), introduced in [23, 24] (see Section 2). That paper enabled the authors to derive fully-dynamic recognition algorithms for distance-hereditary graphs and various subfamilies. GLTs are a combinatorial structure designed to capture precisely the underlying structure of split decomposition [14] and in other similar reformulations that have been considered in the literature, for instance in a logical context [12] or in a distance-hereditary graph drawing context [19]. GLTs can also be understood as a special case of a term in a graph grammar [18]. They are valuable here for greatly simplifying the consideration of split decomposition and providing the insight for the results in this paper.

The overview of our algorithm appears as Algorithm 1, where G_0 refers to the empty graph, G_i denotes the subgraph of G induced on $\{x_1, \dots, x_i\}$, and $ST(G_i)$ denotes the GLT (called the split-tree) that captures the split decomposition of G_i .

We use GLTs to derive a combinatorial incremental characterization of split decomposition, generalizing that given for distance-hereditary graphs in [23, 24] (see Section 4). Note that in Theorem 4.14 and its subsequent propositions, we characterize all possible ways in which $ST(G_{i-1})$ is modified to produce $ST(G_i)$. GLTs are also used to demonstrate properties of split decomposition with respect to Lexicographic Breadth-First Search (LBFS) [35] (see Section 3). Sections 3 and 4 are independent, and their content provides general results and constructions that may be useful on their own. Notably, the results of Section 4 easily yield an efficient split decomposition dynamic algorithm supporting vertex insertion and deletion.

By applying the incremental characterization to an LBFS ordering we achieve a split decomposition algorithm that is conceptually straightforward, but requires a careful and detailed explanation

³ Let us mention that several definitions exist for this function, either with two variables, including some variants, or with one variable. For simplicity, we choose to use the version with one variable. This makes no practical difference since all of them could be used in our complexity bound, and they are all essentially constant. As an example, the two variable function considered in [9] satisfies $\alpha(k, n) \leq 4$ for all integer k and for all $n \leq \underbrace{2^{\dots^2}}_{17 \text{ times}}$.

Algorithm 1: The Split Decomposition Algorithm

Input: A connected graph G with n vertices.

Output: $ST(G)$, the split-tree of G .

$ST(G_0) \leftarrow \text{null}$;

Using Algorithm 2, do an LBFS on G to produce ordering x_1, x_2, \dots, x_n ;

for $i = 1$ **to** n **do**

 | $ST(G_i) \leftarrow ST(G_{i-1}) + x_i$, using Algorithm 3;

end for

return $ST(G_n)$;

of the implementation in order to achieve the stated running time (see Section 5). We develop a charging argument based on the structure of GLTs that allows us to evaluate the amortized cost of inserting each vertex, according to an LBFS ordering. We use it to prove the $O(n+m)\alpha(n+m)$ running time (see Section 6). Furthermore, our algorithm extends to circle graph recognition; the companion paper [25] uses it to develop the first sub-quadratic circle graph recognition algorithm, which also runs in $O(n+m)\alpha(n+m)$ time. Note that different versions of both the split decomposition algorithm and the circle graph recognition algorithm appear in [39].

2 Preliminaries

All graphs in this document are simple, undirected, and connected. The set of vertices in the graph G is denoted $V(G)$ and the set of edges by $E(G)$. The graph *induced* on the set of vertices S is signified by $G[S]$. We let $N_G(x)$, or simply $N(x)$, denote the neighbours of vertex x , and for S a set of vertices $N(S) = (\cup_{x \in S} N(x)) \setminus S$. A vertex is *universal* to a set of vertices S if $S \subseteq N(x)$; it is *isolated* from S if $N(x) \cap S = \emptyset$. A vertex is *universal in* a graph if it is adjacent to every other vertex in the graph. We use $N[x] = N(x) \cup \{x\}$ to denote the *closed neighbourhood of a vertex*. Two vertices x and y are *twins* if $N(x) \setminus \{y\} = N(y) \setminus \{x\}$. A *pendant* is a vertex of degree one. A *clique* is a graph in which every pair of vertices is adjacent. A *star* is a graph with at least three vertices in which one vertex, called its *centre*, is universal, and no other edges exist; the vertices other than the centre are called its *degree-1 vertices*. The clique on n vertices is denoted K_n ; the star on n vertices is denoted S_n .

The graph $G + (x, N(x))$ is formed by adding the vertex x to the graph G adjacent to the subset $N(x)$ of vertices, its neighbourhood; when $N(x)$ is clear from the context, we simply write $G + x$. The graph $G - x$ is formed from G by removing x and all its incident edges.

The non-leaf vertices of a tree T are called its *nodes*. The edges in a tree not incident to leaves are its *internal* edges. If S is a set of leaves of T , then $T(S)$ denotes the smallest connected subtree spanning S . If T is a tree, then $|T|$ represents its number of nodes and leaves. In a rooted tree T , every node or leaf x (except the root) has a unique *parent*, namely its neighbour on the path to the root. A *child* of a node x is a neighbour of x distinct from its parent.

2.1 Split decomposition

This subsection recalls original definitions from [14].

Definition 2.1. A split of a connected graph $G = (V, E)$ is a bipartition (A, B) of V , where $|A|, |B| > 1$ such that every vertex in $A' = N(B)$ is universal to $B' = N(A)$. The sets A' and B' are called the frontiers of the split.

A graph not containing a split is called *prime*. A bipartition is *trivial* if one of its parts is the empty set or a singleton. Cliques and stars are called *degenerate* since every non-trivial bipartition of their vertices is a split:

Remark 2.2. Let (A, B) be a bipartition of the vertices in a clique or a star such that $|A|, |B| > 1$. Then (A, B) is a split.

Degenerate graphs and prime graphs represent the base cases in the process defining split decomposition:

Definition 2.3. Split Decomposition is a recursive process decomposing a given graph G into a set of disjoint graphs $\{G_1, \dots, G_k\}$, called split components, each of which is either prime or degenerate. There are two cases:

1. if G is prime or degenerate, then return the set $\{G\}$;
2. if G is neither prime nor degenerate, it contains a split (A, B) , with frontiers A' and B' . The split decomposition of G is then the union of the split decompositions of the graphs $G[A] + a$ and $G[B] + b$, where a and b are new vertices, called markers, such that $N_{G[A]+a}(a) = A'$ and $N_{G[B]+b}(b) = B'$.

Notice that during the split decomposition process, the marker vertices can be matched by so called *split edges*. Then given a split decomposition, provided the marker vertices and their matchings are specified, the input graph G can be reconstructed without ambiguity. The set of split edges merely defines the *split decomposition tree* whose nodes are the components of the split decomposition.

Cunningham showed that every graph has a canonical split decomposition tree [14]. As Cunningham's original work was on the decomposition of a graph by a family of bipartitions of the vertex set, his paper focuses on the tree representation of the family of splits to obtain a canonical tree rather than on how the graph's adjacencies can be retrieved from its split decomposition tree. At first sight, it is not immediately clear how the graph's adjacencies are encoded by the split decomposition tree, and what role the marker vertices play in determining them. Tellingly, the base case treats prime and degenerate graphs the same; looking at the tree, the viewer is left to guess which one applied. In recent papers [22, 12], split decomposition is represented by the *skeleton graph* which is the union of the split components connected by the split edges. The fact that G 's vertices and the marker vertices are mixed is a drawback of this representation.

A recent reformulation of split decomposition in terms of graph-labelled trees (GLTs) aims to clarify this [23, 24]. Our investigation of split decomposition takes place entirely in this new GLT setting, which is described below.

2.2 Graph-labelled trees

This subsection recalls definitions from [23, 24] and adds useful terminology.

Definition 2.4 ([23, 24]). A graph-labelled tree (GLT) is a pair (T, \mathcal{F}) , where T is a tree and \mathcal{F} a set of graphs, such that each node u of T is labelled by the graph $G(u) \in \mathcal{F}$, and there exists a bijection ρ_u between the edges of T incident to u and the vertices of $G(u)$. (See Figure 1.)

When we refer to a node u in a GLT, we usually mean the node itself in T (non-leaf vertex). We may sometimes use the notation u as a shortcut for its label $G(u) \in \mathcal{F}$; the meaning will be clear from the context. For instance, notation will be simplified by saying $V(u) = V(G(u))$. The vertices in $V(u)$ are called *marker vertices*, and the edges between them in $G(u)$ are called *label-edges*. For a label-edge $e = uv$ we may say that u and v are *the vertices of e* . The edges of T are *tree-edges*. The marker vertices $\rho_u(e)$ and $\rho_v(e)$ of the internal tree-edge $e = uv$ are called the *extremities* of e . Furthermore, $\rho_v(e)$ is the *opposite* of $\rho_u(e)$ (and vice versa). A leaf is also considered an *extremity* of its incident edge, and its opposite is the other extremity of the edge (marker vertex or leaf). For convenience, we will use the term *adjacent* between: a tree-edge and one of its extremities; a label-edge and one of its vertices; two extremities of a tree-edge, etc., as long as the context is clear. The most important notion for GLTs with respect to split decomposition is that of *accessibility*:

Definition 2.5 ([23, 24]). Let (T, \mathcal{F}) be a GLT. The marker vertices q and q' are accessible from one another if there is a sequence Π of marker vertices q, \dots, q' such that:

1. every two consecutive elements of Π are either the vertices of a label-edge or the extremities of a tree-edge;
2. the edges thus defined alternate between tree-edges and label-edges.

Two leaves are accessible from one another if their opposite marker vertices are accessible; similarly for a leaf and marker vertex being accessible from one another; see Figure 1 where the leaves accessible from q include both 3 and 15 but neither 2 nor 11. By convention, a leaf or marker vertex is accessible from itself.

Note that, obviously, if two leaves or marker vertices are accessible from one another, then the sequence Π with the required properties is unique, and the set of tree-edges in Π forms a path in the tree T .

Definition 2.6 ([23, 24]). Let (T, \mathcal{F}) be a GLT. Then its accessibility graph, denoted $Gr(T, \mathcal{F})$, is the graph whose vertices are the leaves of T , with an edge between two distinct leaves ℓ and ℓ' if and only if they are accessible from one another. Conversely, we may say that (T, \mathcal{F}) is a GLT of $Gr(T, \mathcal{F})$.

Accessibility allows us to view GLTs as encoding graphs; an example appears in Figure 1.

Let (T, \mathcal{F}) be a GLT, and let q be a marker vertex belonging to the node u of T and corresponding to the tree-edge e of T . Then we denote:

- $L(q)$ the set of leaves of T from which there is a path to u using e ;
- $A(q)$ the subset of leaves of $L(q)$ that are accessible from q ;
- $T(q) = T(L(q))$ the smallest subtree of T that spans the leaves $L(q)$; note that $q \notin T(q)$.

To unify our notation, for a leaf ℓ of T , the sets $L(\ell)$, $A(\ell)$, $T(\ell)$ can be similarly defined, so that $A(\ell) = N_G(\ell)$, ℓ 's neighbourhood in $G = Gr(T, \mathcal{F})$, and $L(\ell) = V(G) \setminus \{\ell\}$.

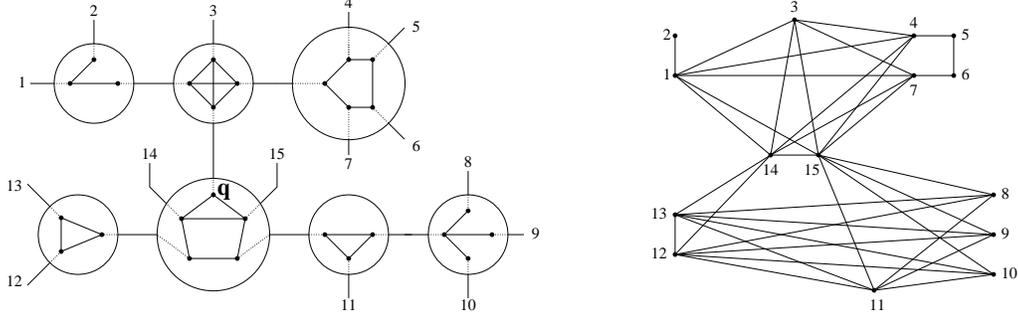


Figure 1: A graph-labelled tree (T, \mathcal{F}) and its accessibility graph $Gr(T, \mathcal{F})$.

Definition 2.7. Let (T, \mathcal{F}) be a GLT and let q and p be distinct marker vertices. Then p is a descendant of q if $L(p) \subset L(q)$, that is if $T(p)$ is a subtree of $T(q)$.

The above notation and definitions are illustrated in Figure 2. Also note that a leaf is never a descendant of a leaf or a marker vertex.

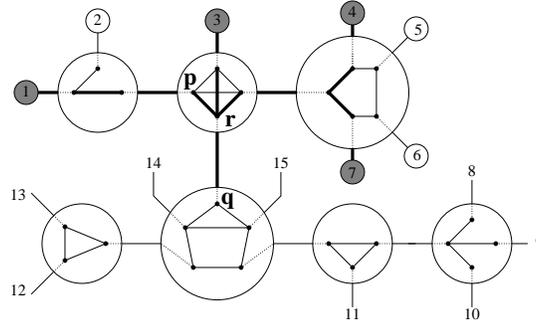


Figure 2: A marker vertex q , with $L(q) = \{1, 2, 3, 4, 5, 6, 7\}$ and $A(q) = \{1, 3, 4, 7\}$. Bold edges are those used in sequences certifying the accessibility between q and elements of $A(q)$. The vertices of the subtree $T(q)$ are the elements of $L(q)$ together with the three upper nodes. The marker vertex p with $L(p) = \{1, 2\}$ is a descendant of q . The marker vertex r is the opposite of q , and is not a descendant of q .

We conclude this subsection by a series of remarks following directly from the definitions.

Remark 2.8. If a graph G is connected, then every label in a GLT of G is connected.

Remark 2.9. For any marker vertex q in a GLT of a connected graph, $A(q) \neq \emptyset$.

As a consequence, by choosing one element of $A(q)$ for every marker vertex q in the label we see that every label in a GLT of a connected graph G is an induced subgraph of G .

Remark 2.10. Let p and q be two marker vertices of a GLT such that p is a descendant of q . If p and q are accessible from one another, then $A(q) \cap L(p) = A(p)$. If p and q are non-accessible from one another, then $A(q) \cap L(p) = \emptyset$.

2.3 The split-tree

This subsection reformulates split decomposition [14] in the GLT setting, as done in [23, 24].

Definition 2.11. Let e be a tree-edge incident to nodes u and u' in a GLT, and let $q \in V(u)$ and $q' \in V(u')$ be the extremities of e . The node-join of u, u' replaces u and u' with a new node v labelled by the graph formed from $G(u)$ and $G(u')$ as follows: all possible label-edges are added between $N(q)$ and $N(q')$, and then q and q' are deleted. See Figure 3.

Definition 2.12. The node-split is the inverse of the node-join. More precisely, let v be a node such that $G(v)$ contains the split (A, B) with frontiers A' and B' . The node-split with respect to (A, B) replaces v with two new adjacent nodes u and u' labelled by $G[A] + q$ and $G[B] + q'$, respectively, where q and q' are the extremities of the new tree-edge thus created, q being universal to A' , and q' being universal to B' . The extremities of the tree-edges incident to v remain unchanged. See Figure 3.

When a node-split or a node-join operation is performed, a marker vertex of the initial GLT is inherited by the resulting GLT through the operation if its corresponding tree-edge has not been affected by the operation, i.e. if its corresponding tree-edge is not created or deleted in one of the above definitions.

The key property to observe is:

Observation 2.13. The node-join operation and the node-split operation preserve the accessibility graph of the GLT.

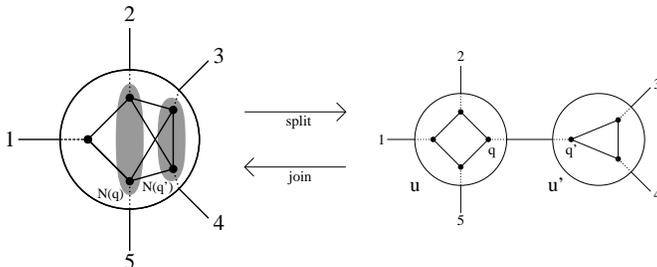


Figure 3: Example of the node-join and node-split.

Hence, GLTs do not uniquely encode graphs. In particular, recursive application of the node-join on every edge of a GLT of G leads to the GLT with a unique node labelled by the accessibility graph G . And conversely, any GLT of a graph G can be obtained by recursive application of the node-split from the GLT consisting of a unique node labelled by G .

Also, observe that, as a consequence, the accessibility graph G of a GLT and the tree structure of the GLT (with leaves labelled by $V(G)$) completely determine the node labels of the GLT. Therefore, transforming a GLT into another GLT using node-splits and node-joins can be done using any ordering for such operations. In particular, performing a set of node-joins can be done in any order without changing the result (the final tree structure is obtained by contracting edges from the initial tree). And concerning node-splits, creating two tree-edges using these operations can be done equally by creating first one tree-edge or the other. We emphasize these two remarks, as they will guarantee the consistency of further constructive statements.

Remark 2.14. Applying a sequence of node-joins on a GLT yields the same GLT, regardless of the order of the node-joins.

Remark 2.15. Recalling the notation of Definition 2.12, let v be a node of a GLT and let (A, B) and (C, D) be two splits of $G(v)$ such that $A \subset C$. Then applying the node-split on v with respect to (A, B) and then on node u' with respect to $((C \setminus A) \cup \{q'\}, D)$ or applying the node-split on v with respect to (C, D) and then on node u with respect to $(A, (B \setminus D) \cup \{q\})$ yields the same GLT.

Of special interest are those node-joins/splits involving degenerate nodes. The *clique-join* is a node-join involving adjacent cliques: its result is a clique node; the *clique-split* is its inverse operation. The *star-join* is a node-join involving adjacent stars whose common incident tree-edge has exactly one extremity that is the centre of its star: its result is a star node; the *star-split* is its inverse operation. Figure 4 provides examples.

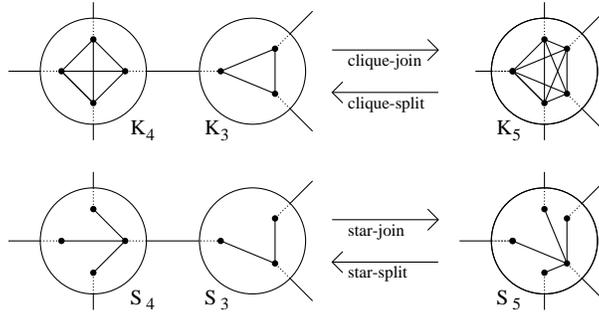


Figure 4: Examples of the clique-join/split and star-join/split.

Definition 2.16. A GLT is reduced if all its labels are either prime or degenerate, and no clique-join or star-join is possible.

We can now state the main result of [14], as reformulated in [23, 24].

Theorem 2.17 ([14, 23, 24]). For any connected graph G , there exists a unique, reduced graph-labelled tree (T, \mathcal{F}) such that $G = Gr(T, \mathcal{F})$.

The unique GLT guaranteed by the previous theorem is the *split-tree*, and is denoted $ST(G)$. The GLT in Figure 1 is the split-tree for the accessibility graph pictured there. The split-tree is the intended replacement for Cunningham’s split decomposition tree. The following theorem first appeared in Cunningham’s seminal paper [14] in an equivalent form. We phrase it in terms of GLTs and the split-tree:

Theorem 2.18 ([14]). Let G be a connected graph. A bipartition (A, B) is a split of G if and only if either there exists an internal tree-edge of $ST(G)$ with extremities p and q such that $A = L(p)$ and $B = L(q)$, or there exists a degenerate node u and a split (A_u, B_u) of $G(u)$ such that $A = \cup_{p \in A_u} L(p)$ and $B = \cup_{p \in B_u} L(p)$.

In other words, the split-tree can be understood as a compact representation of the family of splits of a connected graph. Indeed it is easy to show that the size of the split-tree $ST(G)$ of a

graph is linear in the size of G (the sum of the sizes of label graphs of $ST(G)$ is linear in the size of G), whereas a graph can have exponentially many splits (it is the case for the clique and the star). The following corollary is simply a rephrasing of Theorem 2.18 based on the node-split operation.

Corollary 2.19 ([14, 23, 24]). *Let $ST(G) = (T, \mathcal{F})$. Any split of G is the bipartition (of leaves) induced by removing an internal tree-edge from \tilde{T} , where $\tilde{T} = T$, or \tilde{T} is obtained from T by exactly one node-split of a degenerate node.*

Compared to Cunningham’s split decomposition tree or the skeleton graph representation (see the remarks following Definition 2.3), the advantage of the split-tree is manifest. The adjacency relation in the underlying graph is now explicitly represented by the accessibility relation, and the role played by the marker vertices (and their own adjacencies) is established. All this added information comes with no space-tradeoff:

Lemma 2.20 ([23, 24]). *Let $ST(G) = (T, \mathcal{F})$. If $x \in V(G)$, then $|T(N(x))| \leq 2 \cdot |N(x)|$.*

3 Lexicographic breadth-first search

As mentioned in the introduction, our algorithm incrementally builds the split-tree by adding vertices one at a time from the input graph. Now, adding a single vertex to the split-tree of a graph with n vertices can require $\Theta(n)$ changes, as demonstrated in Figure 5. However, later in the paper we prove that if vertices are added according to a Lexicographic Breadth-First Search (LBFS) ordering, then the total cost of inserting all vertices of G can be amortized to linear time up to inverse Ackermann function.

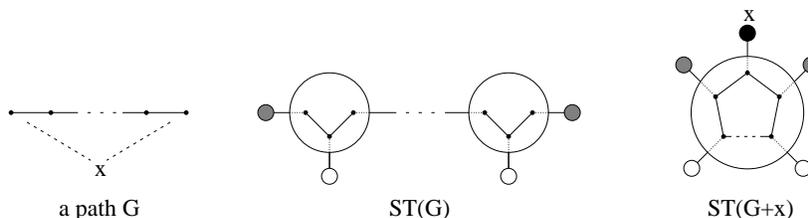


Figure 5: Adding a single vertex adjacent to the ends of a path requires $\Theta(n)$ changes to the split-tree (the neighbours of x appear as the grey leaves).

This section presents new LBFS results on the split decomposition and more generally on GLTs. We first present the LBFS algorithm and some known results.

3.1 LBFS orderings

Lexicographic Breadth-First Search (LBFS) was developed by Rose, Tarjan, and Lueker for the recognition of chordal graphs [35] and has since become a standard tool in algorithmic graph theory [10].

An *ordering* σ of a graph G is a linear ordering of its set of vertices $V(G)$. Formally, we can define it either as an injective mapping from $V(G)$ to the integers, or as an ordering binary relation.

We slightly abuse notation by allowing σ to represent such a mapping as well as the ordering, and we let $<_\sigma$ denote the binary relation: $x <_\sigma y$ is equivalent to $\sigma(x) < \sigma(y)$. In such a case, we say that “ x appears before y ”, or “earlier than y ”, in σ . Similarly, by “first”, “last” and “penultimate”, we denote respectively, the smallest element of $<_\sigma$, the greatest element and the element appearing immediately before the last one.

By an LBFS ordering of the graph G , we mean any ordering produced by Algorithm 2 on input graph G . Notice that such an ordering can be built in linear time (see e.g. [26, 27]).

Algorithm 2: Lexicographic Breadth-First Search

Input: A graph G with n vertices.

Output: An ordering σ defined by a mapping $\sigma : V(G) \rightarrow \{1, \dots, n\}$.

foreach $x \in V(G)$ **do** label(x) \leftarrow null;

for $i = 1$ **to** n **do**

pick an unnumbered vertex x with lexicographically largest label;
 $\sigma(x) \leftarrow i$; // **assign** x the number i
foreach unnumbered vertex $y \in N(x)$ **do** append $n - i + 1$ to label(y);

end for

The next result characterizes LBFS orderings:

Lemma 3.1 ([17, 26]). *An ordering σ of a graph G is an LBFS ordering if and only if for any triple of vertices $a <_\sigma b <_\sigma c$ with $ac \in E(G)$, $ab \notin E(G)$, there is a vertex $d <_\sigma a$ such that $db \in E(G)$, $dc \notin E(G)$.*

For a subset S of $V(G)$, $\sigma[S]$ denotes the restriction of σ to S . A *prefix* of an ordering σ is a set S such that $x <_\sigma y$ and $y \in S$ implies $x \in S$. One obvious result is the following:

Remark 3.2. *Let S be a prefix of any LBFS ordering σ of connected graph G . Then $\sigma[S]$ is an LBFS ordering of $G[S]$, and $G[S]$ is connected.*

3.2 LBFS and split decomposition

We now introduce a general lemma about split decomposition, followed by lemmas relating LBFS orderings and split decomposition.

Lemma 3.3. *Let G and $G + x$ be two connected graphs such that G is prime but $G + x$ is not. Then either x is a pendant vertex or x has a twin.*

Proof. Since $G + x$ is not prime, it has a split (A, B) . Let A' and B' be the frontiers of the split. Without loss of generality, assume that $x \in A$. Since $(A \setminus \{x\}, B)$ is not a split in G , we know that $|A| = 2$. If $A' = \{x\}$, then G is disconnected. If $A' = \{x, y\}$, then y is a twin of x . If $A' = \{y\}$, $y \neq x$, then $N(x) = \{y\}$, since $G + x$ is connected. Therefore x is a pendant. \square

Lemma 3.4. *Let G and $G + x$ be two connected graphs and let σ be an LBFS ordering of $G + x$ in which x appears last. If G is prime and x has a twin y , then y is either universal in G or is the penultimate vertex in σ .*

Proof. Observe that if $|V(G)| > 3$, then y is unique since G is prime. Consider an execution of Algorithm 2 that produced the ordering σ . Let S be the set of vertices with the same label as y at the time y is numbered by Algorithm 2 (of course $y \in S$). As x and y are twins, we must have $x \in S$. We can assume that $S \setminus \{y, x\} \neq \emptyset$ as otherwise y would be the penultimate vertex of σ . Let B be the set of vertices numbered before y by Algorithm 2. Observe that $|B| \leq 1$ as otherwise G would contain the split $(B, S \setminus \{x\})$.

Consider the case where $B = \emptyset$. Then y is the first vertex in σ and immediately following y are the vertices in $N(y)$. If y is not universal in G , then the set $Z = V(G) - N(y)$ is non-empty and in σ , all of its vertices appear after those in $N(y)$. We claim that there is a join between $N(y)$ and Z . Suppose for contradiction that there is no such join. Then there is a vertex $w \in N(y)$ that is not universal to Z . Consider some vertex $z \in Z$ such that $wz \notin E(G)$. With x and y twins, it follows that $wx \in E(G)$. Hence, $w <_\sigma z <_\sigma x$, and $wx \in E(G)$ but $wz \notin E(G)$. Therefore, by Lemma 3.1, there is a vertex $d <_\sigma w$ such that $dz \in E(G)$ but $dx \notin E(G)$. But x is universal to $N(y)$ since it is y 's twin, and thus $d \notin N(y)$. Given the restrictions on σ noted above, it follows that $d = y$. But then $dz \notin E(G)$, since $z \in Z$, providing the desired contradiction.

That means there is a join between $N(y)$ and Z . But now, unless $|N(y)| = 1$, $(N(y), Z \cup \{y\})$ is a split in G contradicting G being prime. When $|N(y)| = 1$, if $|Z| = 1$, then G is a star on three vertices and $G + x$ is a star on four vertices, where the penultimate vertex is a twin of x (note that by Lemma 3.1, $xy \notin E$); if $|Z| > 1$, $(N(y) \cup \{y\}, Z)$ is a split in G . Thus B is not empty.

So let s be the unique vertex of B . Now S , as ordered by σ , consists of y followed by $Z_1 = N(y) \cap S$, followed by Z_2 (vertices of $S \cap V(G)$ not adjacent to y); x is the last element of S . As $S \setminus \{y, x\} \neq \emptyset$, we have that $Z_1 \cup Z_2 \neq \emptyset$. Since x and y are twins, x is universal to Z_1 and not adjacent to any vertices in Z_2 . Since y is not universal in G , $|Z_2| > 0$ and thus by Lemma 3.1, x is not adjacent to y . If $|Z_1| = 0$, then unless $|Z_2| = 1$, G has the split $(\{y, s\}, Z_2)$. If $|Z_2| = 1$, then, as in the $B = \emptyset$ case, G is a star on three vertices and $G + x$ is a star on four vertices, where the penultimate vertex is a twin of x . Thus $|Z_1| > 0$ and $|Z_2| > 0$. By the application of Lemma 3.1 to (z_1, z_2, x) , where $z_1 \in Z_1, z_2 \in Z_2$, we see that $z_1 z_2 \in E(G)$ and thus $(\{s\} \cup Z_1, \{y\} \cup Z_2)$ is a split of G , contradicting G being prime. \square

Let u be a node in a GLT (T, \mathcal{F}) . Notice that the sets $L(q), q \in V(u)$ partition the leaves of T . In other words, each marker vertex can be associated with a distinct leaf in T . This allows us to define a type of induced LBFS ordering on $G(u)$ as demonstrated below.

Definition 3.5. Let u be a node of a GLT (T, \mathcal{F}) and let σ be an ordering of $G = Gr(T, \mathcal{F})$. For any marker vertex p , let x_p be the earliest vertex of $A(p)$ in σ . Define $\sigma[G(u)]$ to be the ordering of $G(u)$ such that for $q, r \in V(u)$, $q <_{\sigma[G(u)]} r$ if $x_q <_\sigma x_r$.

Lemma 3.6. Let σ be an LBFS ordering of a connected graph $G = Gr(T, \mathcal{F})$, and let u be a node in (T, \mathcal{F}) . Then $\sigma[G(u)]$ is an LBFS ordering of $G(u)$.

Proof. First observe that if we collect in a set S one leaf ℓ_q of $A(q)$ for every marker vertex $q \in V(u)$, then the induced subgraph $G[S]$ is isomorphic to $G(u)$. Notice that $\sigma[G(u)]$ is then the ordering $\sigma[S]$ if each selected leaf ℓ_q is chosen to be the earliest in σ . We prove by induction on the number of nodes in T that $\sigma[S] = \sigma[G(u)]$ is an LBFS ordering of $G(u)$. To that aim, we use Lemma 3.1.

As an induction hypothesis, assume the lemma holds for all graphs whose split-tree has fewer nodes than $ST(G)$. The lemma clearly holds if (T, \mathcal{F}) contains only one node, because $G[S]$ is isomorphic to G in this case.

So assume that (T, \mathcal{F}) contains more than one node. Then there is a $q \in V(u)$ such that $T(q)$ contains at least one node. Let $\ell_q \in A(q)$ be the leaf associated with q in $\sigma[G(u)]$. Let $G' = G[(V(G) \setminus L(q)) \cup \{\ell_q\}]$. Remove $T(q)$ from (T, \mathcal{F}) , choosing ℓ_q to be the leaf that replaces its nodes; let (T', \mathcal{F}') be the resulting GLT. Clearly $G' = Gr(T', \mathcal{F}')$. For simplicity, let $\sigma' = \sigma[V(G')]$. Suppose a, b and c form a triple of vertices of $V(G')$ as in Lemma 3.1. As σ is an LBFS ordering of G , there exists $d \in V(G)$ appearing earlier than a in σ which is adjacent to b but not to c . Suppose that d does not belong to $V(G')$, i.e. $d \neq \ell_q$ and $d \in L(q)$. Let p be q 's opposite in (T, \mathcal{F}) . As $(L(q), L(p))$ is a split of G , the vertex b either belongs to $L(p)$ or $L(q)$. In the former case, since b is adjacent to a vertex in $L(q)$, $b \in A(p)$ and thus $d \in A(q)$. By the choice of ℓ_q , it can replace vertex d . (Note that $c \in L(p) \setminus A(p)$ and thus ℓ_q and c are not adjacent.) In the latter case, since ℓ_q is the only $L(q)$ vertex in G' , $b = \ell_q$ and $b \in A(q)$. Moreover, by the choice of ℓ_q , d belongs to $L(q) \setminus A(q)$. We now prove that a cannot appear before b in σ , yielding a contradiction. As b is the only vertex of $L(q)$ present in $V(G')$, so vertex a belongs to $L(p) \setminus A(p)$. By the choice of ℓ_q , no vertex of $A(q)$ appears before a in σ . By Remark 3.2, the subgraph of G induced on the vertices of σ up to, and including a is connected. But, there can be no path in the subgraph connecting $d \in L(q) \setminus A(q)$ and a since $A(q)$ is a separator for d and a , and $b = \ell_q$ is the earliest vertex of $A(q)$ in σ . Thus a cannot appear before b in σ , thereby contradicting the existence of the triple $\{a, b, c\}$. It follows that σ' is an LBFS ordering of G' .

Of course, (T', \mathcal{F}') has fewer nodes than (T, \mathcal{F}) . We can therefore apply our induction hypothesis. Hence, $\sigma[S]$ is an LBFS ordering of $G'[S]$. But notice that $G'[S]$ is isomorphic to $G[S]$ which is isomorphic to $G(u)$. The induction step follows. \square

4 Incremental split decomposition

Throughout this section we assume that the graphs G and $G + x$ are both connected. We provide a simple combinatorial description of the updates required in $ST(G)$ to arrive at $ST(G + x)$. The proof is obtained by a case by case analysis of the properties of $ST(G)$ when removing x from $ST(G + x)$, which turns out to be easily invertible.

4.1 State assignment

Most results in the paper rely on the next definition. Intuitively, its aim is to allow a characterization of the portions of the split-tree that change or fail to change under the insertion of a new vertex.

Definition 4.1. *Let (T, \mathcal{F}) be a GLT, and let q be one of its leaves or marker vertices. Let S be a subset of T 's leaves. Then the state (with respect to S) of q is:*

- perfect if $S \cap L(q) = A(q)$;
- empty if $S \cap L(q) = \emptyset$;
- and mixed otherwise.

For a node u , define the sets $P(u) = \{q \in V(u) \mid q \text{ perfect}\}$, $M(u) = \{q \in V(u) \mid q \text{ mixed}\}$, and $NE(u) = P(u) \cup M(u)$ (“NE” for “Not-empty”). See Figure 6.

Remark 4.2. *The state of a marker vertex before and after a node-split or a node-join is the same.*

Notice that the opposite of any leaf l must be either perfect (if $l \in S$) or empty (if $l \notin S$). We extend the state definition to subtrees: if a marker vertex (or leaf) q is perfect (respectively

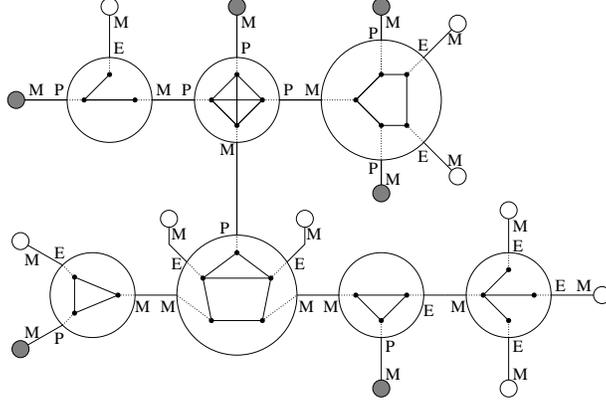


Figure 6: A GLT and states assigned according to the shaded leaves (“P” for “perfect”, “M” for “mixed”, and “E” for “empty”).

empty, mixed), then the *subtree* $T(q)$ is *perfect* (respectively *empty*, *mixed*) as well. A node u of T is *hybrid* if every marker vertex $q \in V(u)$ is either perfect or empty and q 's opposite is mixed. A tree-edge e of T is *fully-mixed* if both of its extremities are mixed. A *fully-mixed subtree* T' of T is one that contains at least one tree-edge, all of its tree-edges are fully-mixed, and it is maximal for inclusion with respect to this property. For a degenerate node u , we denote:

$$P^*(u) = \{q \in V(u) \mid q \text{ perfect and not the centre of a star}\},$$

$$E^*(u) = \{q \in V(u) \mid q \text{ empty, or } q \text{ perfect and the centre of a star}\}.$$

We now describe some basic properties. The first key lemma follows directly from Remark 2.10, and implies the subsequent corollary.

Lemma 4.3 (Hereditary property). *Let (T, \mathcal{F}) be a GLT marked with respect to a subset of leaves S . Then*

1. *a marker vertex q is perfect if and only if every accessible descendant of q is perfect and every non-accessible descendant of q is empty.*
2. *a marker vertex q is empty if and only if every descendant of q is empty.*

Corollary 4.4. *Let (T, \mathcal{F}) be a GLT marked with respect to a subset of leaves S .*

1. *If marker vertex q is mixed, then every marker vertex having q as a descendant is mixed.*
2. *If a tree-edge has a perfect or empty extremity q with a mixed opposite, then, for every tree-edge in $T(q)$, the extremity that is a descendant of q is perfect or empty and its opposite is mixed.*
3. *If there exists a hybrid node, then it is unique.*
4. *In a clique node, if every marker vertex is perfect, then every opposite of a marker vertex is also perfect.*
5. *In a star node, if every marker vertex is empty, except the centre which is perfect, then every opposite of a marker vertex is perfect, except the opposite of the centre which is empty.*

4.2 Lemmas deriving $ST(G)$ from $ST(G+x)$

This subsection is devoted to technical lemmas, which aim to enumerate and characterize in terms of states all possible cases for the deletion of x from $ST(G+x)$. Their proofs rely on an extensive use of the hereditary property (Lemma 4.3) and Corollary 4.4. These lemmas will only be used in the proofs of the next subsection to describe how to update $ST(G)$ when inserting a new vertex.

In this subsection, we let u denote the node of $ST(G+x)$ to which the leaf x is attached. Let (T_x, \mathcal{F}_x) be the GLT, obtained from $ST(G+x)$ by removing leaf x and x' its opposite marker vertex in the label of u , and let u_x be the node corresponding to u in (T_x, \mathcal{F}_x) , such that $G(u_x) = G(u) - x'$. Note that the accessibility graph of (T_x, \mathcal{F}_x) is G . For convenience, but contrary to the definition, the GLT (T_x, \mathcal{F}_x) is allowed to have a binary node u_x in the case where u was ternary; in this case, “contraction of u_x to e ” refers to the operation of replacing u_x and its two adjacent tree-edges by a single tree-edge e . To simplify, we may identify a marker vertex in (T_x, \mathcal{F}_x) with the corresponding marker vertex in $ST(G+x)$. Finally, we assume that $ST(G)$ and (T_x, \mathcal{F}_x) are marked with respect to $S = N_G(x)$. Notice we consider x to have the perfect state and thus the states of the descendants of x in (T_x, \mathcal{F}_x) are determined to be either perfect or empty by applying Lemma 4.3-1 in $ST(G+x)$. To shorten statements, a tree-edge is said to be PP , PE , PM , EM , or MM (i.e. fully-mixed), depending on the states of its two extremities, where P , E , and M , stands respectively for perfect, empty, and mixed.

In the following subsections, we deal with all possibilities of u , the node in $ST(G+x)$ to which x is adjacent.

4.2.1 u is a clique

Lemma 4.5. *Assume x is adjacent to a clique u in $ST(G+x)$. Then every tree-edge of (T_x, \mathcal{F}_x) incident to u_x is PP , and every other edge in (T_x, \mathcal{F}_x) is either PM or EM .*

Proof. Every marker vertex of u_x is a descendant of x in $ST(G+x)$ and hence it is perfect by the hereditary property (Lemma 4.3-1). Then by Corollary 4.4-4, every opposite p of a marker vertex t of u_x is perfect. So every tree-edge incident to u_x is PP .

Let v be a node adjacent to u_x by the tree-edge e , and let p and t be respectively the extremities of e in v and in u_x . Let r be the opposite of a marker vertex q of v distinct from p . Observe that $T(r)$ contains the node u_x and thus r has a perfect descendant. So by the hereditary property (Lemma 4.3-2), r cannot be empty.

We now prove that if r is perfect then, by the hereditary property (Lemma 4.3-1), v is a clique node. Observe first that Lemma 4.3-1 applied on r and p implies that p and q are adjacent. Since $G(v)$ is connected and contains at least 3 marker vertices, v contains a marker vertex distinct from p and q adjacent to at least one of p or q . As every such vertex s is a descendant of t and r (both being perfect), Lemma 4.3-1 implies that s is adjacent to both p and q . It follows that either $(\{p, q\}, V(G(v)) \setminus \{p, q\})$ forms a split in $G(v)$ or v is ternary. Since $ST(G+x)$ is reduced, in both cases v is degenerate and by the adjacencies between p , q and s , v is a clique node.

So we proved that if r is perfect, then $ST(G+x)$ contains two adjacent clique nodes: contradiction. It follows that r is mixed. By the hereditary property (lemma 4.3-1), since t is perfect, q is either perfect or empty. Hence, every tree-edge not incident to u_x is PM or EM by Corollary 4.4-2. \square

Lemma 4.6. *Assume x is adjacent to a clique node u in $ST(G+x)$.*

1. If u is ternary, let (T, \mathcal{F}) be the GLT resulting from the contraction of u_x to e in (T_x, \mathcal{F}_x) .
 - (a) If (T, \mathcal{F}) is reduced then $ST(G) = (T, \mathcal{F})$ and e is the unique PP tree-edge of $ST(G)$, every other tree-edge is PM or EM.
 - (b) Otherwise, $ST(G)$ results from the star-join in (T, \mathcal{F}) of the nodes incident to e (let v be the resulting node). Then v is the unique hybrid node of $ST(G)$, and every tree-edge is PM or EM.
2. If u is not ternary then $ST(G) = (T_x, \mathcal{F}_x)$, u_x is the unique clique node of $ST(G)$ whose marker vertices are all perfect, the tree-edges incident to u_x are PP and every other tree-edge is PM or EM.

Proof. The correctness of the construction of $ST(G)$ follows directly from the definition of the split-tree since the involved operations preserve the accessibility graph G and yield a reduced GLT. The state properties of tree-edges come directly from Lemma 4.5 since, by Remark 4.2, states of marker vertices are preserved by the involved operations. This concludes the proof for cases 1(a) and 2 since uniqueness follows in both cases from Lemma 4.5.

In case 1(b), let p and q denote the two marker vertices of u_x . Observe that as u is a clique node and as (T, \mathcal{F}) is not reduced, the two neighbours v_1 and v_2 of u_x are star nodes such that the centre of v_1 is the opposite of p , whereas the centre of v_2 is not the opposite of q . It follows that $ST(G)$ results from a star-join of v_1 and v_2 . Note that the node v (resulting from the star-join) inherits from v_1 and v_2 the descendants of x in $ST(G+x)$. It follows by the hereditary property (Lemma 4.3-1), that the marker vertices of v are perfect or empty. Finally observe that v contains empty and perfect marker vertices: the non-centre marker vertices inherited from v_2 are empty, all the others are perfect. It follows that v is a hybrid node and it is unique (by Corollary 4.4-3). \square

4.2.2 u is a star node

Lemma 4.7. *Assume x is adjacent to a star node u in $ST(G+x)$. Then every tree-edge of (T_x, \mathcal{F}_x) incident to u_x is PE, and every other edge in (T_x, \mathcal{F}_x) is either PM or EM.*

Proof. Let c be the centre of the star $G(u)$. Since G is connected, x' the opposite of x is a degree-1 marker vertex of $G(u)$. It follows that $S = A(c)$, and thus, c is perfect and, by Corollary 4.4-5, its opposite p is empty. Now let q be a marker vertex of u_x distinct from c and let r be its opposite. By the hereditary property (Lemma 4.3-2), as a descendant of p , q is empty. By Corollary 4.4-5, r is perfect. So we proved that every tree-edge incident to u_x is PE.

We now prove that every tree-edge non-incident to u_x is either PM or EM. Let v be a node adjacent to u_x by the tree-edge e , and let p and t be respectively the extremities of e in v and in u_x . Let r be the opposite of a marker vertex q of v distinct from p .

Assume first that $t \neq c$. Then by Lemma 4.3-2, since c is a perfect descendant of r , r is not empty. So suppose for contradiction that r is perfect. Observe first that Lemma 4.3-1 applied to r and p implies that p and q are adjacent and that by Lemma 4.3-2, as a descendant of t , q is empty. Since $G(v)$ is connected and contains at least 3 marker vertices, v contains a marker vertex s distinct from p and q adjacent to at least one of p or q . As $S = A(c)$, we have that $L(s) \cap S = \emptyset$ implying that s is empty. As s is a descendant of r , by Lemma 4.3-1, s is not adjacent to q and thereby it is adjacent to p . It follows that in $G(v)$, the marker vertex q has degree one. Then v has

to be a star node whose centre is p : this contradicts the fact that $ST(G+x)$ is reduced. It follows that r is mixed (it can not be perfect or empty).

Assume now that $t = c$. If r is empty, by definition $L(r) \cap S = \emptyset$. For every neighbour q' of p , there exists a leaf accessible from c in $T(q')$, and hence an element of S is in $T(q')$. But now, for every $q' \neq q$, $T(q') \subseteq T(r)$ which contradicts $L(r) \cap S = \emptyset$. Thus q is the only neighbour of p in $G(v)$, and v has to be a star node whose centre is q ; this contradicts the fact that $ST(G+x)$ is reduced. So r is perfect or mixed. Now assume r is perfect. Since p is an empty descendant of r , by Lemma 4.3-1, p and q are not adjacent in $G(v)$. Since $G(v)$ is connected and contains at least 3 marker vertices, v contains a marker vertex distinct from p and q adjacent to at least one of p or q . As every such vertex s is a descendant of r and c , both being perfect, Lemma 4.3-1 implies that s is adjacent to p and q . It follows that either $(\{p, q\}, V(G(v)) \setminus \{p, q\})$ forms a split in $G(v)$ or v is ternary. In both cases, v is degenerate and by the adjacencies between p , q and s , v is a star node whose centre is s . This contradicts the fact that $ST(G+x)$ is reduced. It follows in this case also that r is mixed (it can not be perfect or empty).

So r is always mixed and q is perfect or empty by the hereditary property (Lemma 4.3-1 applied to the marker vertices of u_x). Then, every tree-edge not incident to u_x is PM or EM by Corollary 4.4-2. \square

Lemma 4.8. *Assume x is adjacent to a star node u in $ST(G+x)$.*

1. *If u is ternary, let (T, \mathcal{F}) be the GLT resulting from the contraction of u_x to e in (T_x, \mathcal{F}_x) .*
 - (a) *If (T, \mathcal{F}) is reduced then $ST(G) = (T, \mathcal{F})$ and e is the unique PE tree-edge of $ST(G)$, and every other tree-edge is PM or EM.*
 - (b) *Otherwise, $ST(G)$ results from the star-join or a clique-join in (T, \mathcal{F}) of the nodes incident to e (let v be the resulting node). Then v is the unique hybrid node of $ST(G)$, and every tree-edge is PM or EM.*
2. *If u is not ternary then $ST(G) = (T_x, \mathcal{F}_x)$, u_x is its unique star node whose marker vertices are all empty except the centre, which is perfect, tree-edges adjacent to u_x are PE, and all other tree-edges are PM or EM.*

Proof. The proof follows the same lines as the proof of Lemma 4.6, using Lemma 4.7 instead of Lemma 4.5. Only the arguments to show that v is a hybrid node in case 1(b) differ slightly.

So assume case 1(b) holds. As u is ternary and (T, \mathcal{F}) is not reduced, the two neighbours v_1 and v_2 of u are either clique nodes or star nodes. Suppose that the centre c of u is the extremity of the tree-edge uv_1 . Let: p_1 be the opposite of c ; p_2 be the marker vertex of v_2 's extremity of the tree-edge uv_2 ; and q be the opposite of p_2 . Observe that p_1 is universal in $G(v_1)$: this is trivial if $G(v_1)$ is a clique node; if $G(v_1)$ is a star node, then the fact that $ST(G+x)$ is reduced implies that p_1 is the centre of the star. Now, due to their adjacency with x' (the opposite of x), c is perfect and q is empty. It follows by Lemma 4.3-1, that the marker vertices of v_1 distinct from p_1 are perfect (since they are accessible descendants of c). Similarly, by Lemma 4.3-2, the marker vertices of v_2 distinct from p_2 are empty (since they are descendant of q). As all these marker vertices are inherited by v , v is an hybrid node which is unique (by Corollary 4.4-3). \square

4.2.3 u is a prime node

Here we have two cases depending on whether or not $G(u_x)$ is also prime.

Lemma 4.9. *Assume that x is adjacent to a prime node u in $ST(G + x)$ such that $G(u_x)$ is also prime. Then $ST(G) = (T_x, \mathcal{F}_x)$, u_x is its unique hybrid node, and every tree-edge is PM or EM.*

Proof. Observe that by construction, every marker vertex of u_x is either perfect or empty. Let p be the opposite of a marker vertex t of u_x . Assume that p is perfect. Then, in $G(u)$, t is a twin of x' , the opposite of x : contradicting the fact that u is a prime node of $ST(G + x)$. So assume that p is empty. Then, by the hereditary property (Lemma 4.3-2), every marker vertex q distinct from t in u_x is empty. It follows that $A(t) = S$ and thereby t is the only neighbour of x' in $G(u)$. This is again a contradiction with the fact that u is a prime node of $ST(G + x)$, since a prime graph does not contain pendant vertices. Thus p is mixed and now the proof follows from Corollary 4.4 (-2 and -3). \square

It remains to analyse the case where x is adjacent to a prime node u in $ST(G + x)$ but $G(u_x)$ is not prime. To that aim, we describe a three step construction that computes $ST(G)$ from $ST(G + x)$. Note that this construction is not part of the Split Decomposition Algorithm itself.

Let us first recall that when a node-join or a node-split has been performed on an initial GLT, then a marker vertex is *inherited* by the resulting GLT if its corresponding tree-edge is not affected by the operation. We say that a tree-edge $e = uv$ of a GLT is *non-reduced* if a node-join on u and v yields a star node or a clique node.

The announced construction is the following; it uses (T_x, \mathcal{F}_x) as input:

1. While the current GLT contains a node v which is neither prime nor degenerate, find a split in $G(v)$ and perform the node-split accordingly.
2. While the resulting GLT contains a non-reduced tree-edge e both extremities of which are not inherited from (T_x, \mathcal{F}_x) , perform the corresponding node-join. Let (T'_x, \mathcal{F}'_x) be the resulting GLT.
3. While the current GLT contains a non-reduced tree-edge, perform the corresponding node-join. Let (T, \mathcal{F}) be the resulting GLT.

The rest of the results in this section should be interpreted in the context of the above construction, which will subsequently be referred to as the *prime-splitting construction* for simplicity. The following observation concerning the prime-splitting construction follows from the fact that a node-join and a node-split do not change the accessibility of a GLT and that (T, \mathcal{F}) is clearly reduced.

Observation 4.10. *The GLT (T, \mathcal{F}) resulting from the prime-splitting construction is the split-tree $ST(G)$.*

Intuitively, the GLT (T'_x, \mathcal{F}'_x) is obtained from T_x, \mathcal{F}_x by replacing the node u_x with the split-tree $ST(G(u_x))$. Such a replacement is obtained by accurately identifying the leaves of $ST(G(u_x))$ with the marker vertices opposite the marker vertices of u_x . Indeed, note that in the case where $G + x$ is prime, but not G , then $ST(G) = (T'_x, \mathcal{F}'_x) = (T, \mathcal{F})$. To help the intuition of the following lemmas, we state the summarizing lemma (Lemma 4.13) in the context of this special case.

We will now describe the properties of $ST(G)$ and of the intermediate GLT (T'_x, \mathcal{F}'_x) in terms of the states of their marker vertices. Recall that by Remark 4.2, the states of inherited marker vertices remain unchanged. Also observe that after a series of node-joins and node-splits, a tree-edge e of the resulting GLT has its two extremities either both inherited or both non-inherited.

In the former case, e is an *inherited tree-edge*, in the latter case a *non-inherited tree-edge*. Finally observe that if e is a non-inherited tree-edge, then it corresponds to a split (A_x, B_x) of $G(u_x)$ since it results from the first and second steps of the prime-splitting construction. Intuitively, the non-inherited tree-edges correspond to the internal tree-edges of $ST(G(u_x))$.

Lemma 4.11. *Consider the prime-splitting construction. Assume that x is adjacent to a prime node u in $ST(G+x)$. Then every non-inherited tree-edge of (T'_x, \mathcal{F}'_x) is MM and every inherited tree-edge is PM or EM.*

Proof. Note that if $G(u_x)$ is prime, the result is trivial. We first prove that inherited tree-edges are PM or EM. To that aim we first argue that every opposite q of a marker vertex p of u_x is mixed in (T_x, \mathcal{F}_x) marked with respect to S . Observe that q cannot be perfect, since otherwise p and x' are twins in $G(u)$, contradicting node u being a prime node (a prime graph cannot have a pair of twins). So assume that q is empty. Then p cannot be empty since otherwise we would have $S = \emptyset$ (as $L(q) = L(p) = \emptyset$). If p is perfect, then p has degree 1 in $G(u)$ (since $L(q) = \emptyset$). Thus p is a pendant vertex of $G(u)$: contradiction, a prime graph cannot have a pendant vertex. It follows by Corollary 4.4-2 that every tree-edge of (T_x, \mathcal{F}_x) is PM or EM. Now by Remark 4.2, the state of the inherited marker vertices are preserved under node-joins and node-splits. Thus every inherited tree-edge of (T'_x, \mathcal{F}'_x) is PM or EM.

We now deal with non-inherited tree-edges. Let p be the extremity of such a tree-edge e in (T'_x, \mathcal{F}'_x) . Denote by (A, B) the split of G corresponding to e with $L(p) = B$. As noticed before, e also corresponds to a split (A_x, B_x) of $G(u_x)$ (we see that $A = \cup_{q \in A_x} L(q)$ and $B = \cup_{q \in B_x} L(q)$). We prove that if p is not mixed, then $G(u)$ contains a split, a contradiction with u being a prime node. So assume first that p is empty. Then by definition $L(p) \cap S = B \cap S = \emptyset$. It follows that the bipartition $(A \cup \{x\}, B)$ is a split of $G+x$ and thus $(A_x \cup \{x'\}, B_x)$ is a split of $G(u)$. Assume now that p is perfect, then $L(p) \cap S = B \cap S = A(p)$. It follows that $(A \cup \{x\}, B)$ is a split of $G+x$ (here x belongs to the frontier of A). Thereby $(A_x \cup \{x'\}, B_x)$ is again a split of $G(u)$. Thus p is mixed, as required. \square

Lemma 4.12. *Consider the prime-splitting construction. Assume that x is adjacent to a prime node u in $ST(G+x)$. Let w be a degenerate node incident to a non-inherited tree-edge in (T'_x, \mathcal{F}'_x) .*

1. *If w is a star, the centre of which is perfect, then w has no empty marker vertex and at most two perfect marker vertices.*
2. *Otherwise w has at most one empty marker vertex and at most one perfect marker vertex.*

Proof. First observe that the result is trivial if $G(u_x)$ is prime. Note that every empty or perfect marker vertex p is inherited. Otherwise p would be the extremity of a tree-edge e corresponding to a split (A_x, B_x) of $G(u_x)$ and being empty or perfect would imply that $(A_x \cup \{x\}, B_x)$ is a split of $G(u)$, contradicting u being prime.

1. *w is a star node, the centre c of which is perfect:* Suppose that w has an empty marker vertex q (distinct from c). As q is inherited from u_x and not accessible from every marker vertex inherited from u_x , q is a pendant vertex in $G(u)$: contradicting u being a prime node. So no marker vertex of w is empty. Suppose now that w has two perfect marker vertices p and p' distinct from c . Again p and p' are inherited from u_x . Moreover, every inherited vertex from u_x accessible to p is accessible to p' (and vice versa). It follows that p and p' form a pair of

twins in $G(u)$: contradicting u being a prime node. So w contains at most two perfect marker vertices (including c).

2. *otherwise*: Suppose that w is a clique node containing two perfect (or two empty) marker vertices p and p' . Again p and p' are inherited from u_x and have the same accessibility set among the inherited marker vertices of u_x . Thereby p and p' are twins in $G(u)$: contradicting u being a prime node.

Assume that w is a star node, the centre c of which is not perfect. If c is mixed, then the same argument as for the clique node proves the property. So assume that c is empty. Then the same argument as in case 1 applies. If w contains an empty marker vertex q distinct from c , then q is pendant in $G(u)$. If w contains two perfect marker vertices p and p' (distinct from c by hypothesis), then p and p' are twins in $G(u)$. Both cases lead to a contradiction. □

The following lemma summarizes what we have found so far and completes the picture. Recall that $P^*(u) = \{q \in V(u) \mid q \text{ is perfect and not the centre of a star}\}$ and that $E^*(u) = \{q \in V(u) \mid q \text{ is empty, or } q \text{ is perfect and the centre of a star}\}$.

Lemma 4.13. *Consider the prime-splitting construction. Assume that x is adjacent to a prime node u in $ST(G+x)$ such that $G(u_x)$ is not prime. Then:*

1. $ST(G) = (T, \mathcal{F})$;
2. every tree-edge of $ST(G)$ both extremities of which are not inherited from (T_x, \mathcal{F}_x) is MM and all other tree-edges are PM or EM ;
3. a degenerate node v of (T, \mathcal{F}) incident to a non-inherited tree-edge results from at most two node-joins during step 3 of the construction and these node-joins respectively generate the split $(P^*(v), V(v) \setminus P^*(v))$ and/or $(E^*(v), V(v) \setminus E^*(v))$ of $G(v)$.

Proof. The first assertion is given by Observation 4.10 and the second follows from Lemma 4.11 and Remark 4.2. So it remains to prove the third property.

Let v be a degenerate node of (T, \mathcal{F}) incident to a non-inherited tree-edge (i.e. an MM tree-edge). Observe that if v results from the node-join of nodes w and w' during the third step, then the tree-edge $e = ww'$ is inherited (i.e. EM or PM) and exactly one of the two nodes, say w is incident to an MM tree-edge. Let p the extremity of e in w and q be the (mixed) extremity of e in w' . We need to examine all the possibilities for p and w . We provide all details for the first case; the others use similar arguments.

If w is a star node and p its perfect centre, then q is a degree-1 marker vertex of w' . The resulting star node v contains the split $(E^*(v), V(v) \setminus E^*(v))$ where $E^*(v)$ is the set of inherited marker vertices of w' ($E^*(v)$ contains the perfect centre and the empty degree-1 marker vertices of w'). This follows from Lemma 4.12-1 which shows that q is the only empty marker vertex and from Remark 4.2 that claims that the states of inherited marker vertices are unchanged under node-join.

The other cases follow from Lemma 4.12-2. If p is an empty marker vertex of w (in that case, the node w can be a star or a clique as well), then node v contains the split $(E^*(v), V(v) \setminus E^*(v))$ with $E^*(v)$ is the set of inherited marker vertices of w' . Now if p is a perfect marker vertex but not the centre of a star, then the resulting node contains the split $(P^*(v), V(v) \setminus P^*(v))$ where $P^*(v)$ is composed of the marker vertices inherited from w' .

Finally, by Lemma 4.12 a degenerate node incident to an MM tree-edge contains at most two non-mixed marker vertices, and thus at most two node-joins are required to generate node v . We mention that forthcoming Figure 9 illustrates the two inverse node-split operations on v . \square

4.3 Construction of $ST(G+x)$ from $ST(G)$

Having shown how $ST(G)$ can be derived from $ST(G+x)$, we now use these results to characterize how $ST(G+x)$ can be derived from $ST(G)$. The various cases of the following theorem drive our Split Decomposition algorithm in Section 5. Recall that by definition, a fully-mixed subtree is maximal.

Theorem 4.14. *Let $ST(G) = (T, \mathcal{F})$ be marked with respect to a subset S of leaves. Then exactly one of the following conditions holds:*

1. $ST(G)$ contains a clique node, whose marker vertices are all perfect, and this node is unique;
2. $ST(G)$ contains a star node, whose marker vertices are all empty except the centre, which is perfect, and this node is unique;
3. $ST(G)$ contains a unique hybrid node, and this node is prime;
4. $ST(G)$ contains a unique hybrid node, and this node is degenerate;
5. $ST(G)$ contains a PP tree-edge, and this edge is unique;
6. $ST(G)$ contains a PE tree-edge, and this edge is unique;
7. $ST(G)$ contains a unique fully-mixed subtree.

Moreover, in every case, the unique node/edge/subtree is obtained from T by deleting, for every tree-edge e with a perfect or empty extremity q whose opposite r is mixed, the tree-edge e and the node or leaf corresponding to r . In case 1 and case 2, the node, together with its adjacent edges, is obtained in this way.

Proof. By Lemmas 4.6, 4.8, 4.9 or 4.13, applied to $G+x$ with $N(x) = S$, we directly know that (at least) one condition holds. A more careful look at these lemmas also proves that exactly one condition holds, implying directly with Corollary 4.4-2 the given construction by deletion of PM and EM edges. First, notice that the following conditions are mutually exclusive: there exists a PP edge; there exists a PE edge; there exists an MM edge. Indeed, every time one of these conditions holds, all the tree-edges of another type are known to be PM or EM (Lemmas 4.6, 4.8, and 4.13). These three cases are mutually exclusive from the existence of a hybrid node (Lemmas 4.6, 4.8, and 4.9). Together, these four cases – existence of a PP edge, PE edge, MM edge, and hybrid node – determine the cases in the theorem: if there is exactly one (respectively at least two) PP edge(s), then case 5 (respectively case 1) holds; if there is exactly one (respectively at least two) PE edge(s), then case 6 (respectively case 2) holds; if there is a hybrid node, then it is either prime (case 3) or degenerate (case 4) but not both; if there is an MM edge, then case 7 holds. \square

Proposition 4.15 (Cases 1, 2 and 3 of Theorem 4.14). *Let $ST(G) = (T, \mathcal{F})$ be marked with respect to a subset $N(x)$ of leaves. If $ST(G)$ contains:*

- [case 1] a unique clique node u the marker vertices of which are all perfect, or
- [case 2] a unique star node u the marker vertices of which are all empty except its centre which is perfect, or
- [case 3] a unique hybrid node u which is prime,

then $ST(G+x)$ is obtained by adding to node u a marker vertex q adjacent in $G(u)$ to $P(u)$ and making the leaf x the opposite of q .

Proof. Trivial by the definition of the split-tree; the resulting GLT is reduced and its accessibility graph is $G+x$. Notice that each of these three cases is the converse construction of the one provided in Lemma 4.6-2, or Lemma 4.8-2, or Lemma 4.9, respectively. \square

Proposition 4.16 (Case 4 of Theorem 4.14). *Let $ST(G) = (T, \mathcal{F})$ be marked with respect to a subset $N(x)$ of leaves. If $ST(G)$ contains a unique hybrid node u which is degenerate, then $ST(G+x)$ is obtained in two steps:*

1. performing the node-split corresponding to $(P^*(u), E^*(u))$ thus creating a tree-edge e both of whose extremities are perfect or empty (see Figure 7);
2. subdividing e with a new ternary node adjacent to x and e 's extremities, such that the node is a clique if both extremities of e are perfect, and such that the node is a star whose centre is the opposite of e 's empty extremity otherwise (see Figure 8).

Proof. First, observe that $(P^*(u), E^*(u))$ is a split since $|P^*(u)| > 1$ and $|E^*(u)| > 1$, otherwise the degenerate node u would either be a clique adjacent to a PP edge or a star adjacent to a PE edge, contradicting u being hybrid. Then the construction follows easily from the definition of the split-tree: the resulting GLT is reduced and its accessibility graph is $G+x$. Notice that this construction is the converse of the one provided in Lemma 4.6-1(b) if the label is a clique, or Lemma 4.8-1(b) if the label is a star. \square

Proposition 4.17 (Cases 5 and 6 of Theorem 4.14). *Let $ST(G) = (T, \mathcal{F})$ be marked with respect to a subset $N(x)$ of leaves. If $ST(G)$ contains:*

- [case 5] a unique tree-edge e both of whose extremities are perfect, then $ST(G+x)$ is obtained by subdividing e with a new clique node adjacent to x and e 's extremities (see Figure 8);
- [case 6] a unique tree-edge e one of whose extremities is perfect and the other empty, then $ST(G+x)$ is obtained by subdividing e with a new star node adjacent to x and e 's extremities, such that the centre of the star is opposite e 's empty extremity (see Figure 8).

Proof. Direct by the definition of the split-tree; the resulting GLT is reduced and its accessibility graph is $G+x$. Notice that each of these two cases is the converse construction of the one provided in Lemma 4.6-1(a), or Lemma 4.8-1(a), respectively. \square

Definition 4.18. *Let (T, \mathcal{F}) be a GLT marked with respect to a subset of leaves and having a fully-mixed subtree. Cleaning the GLT consists of performing, for every degenerate node u of the fully-mixed subtree, the node-splits defined by $(P^*(u), V(u) \setminus P^*(u))$ and/or $(E^*(u), V(u) \setminus E^*(u))$ as long as they are splits of $G(u)$. The resulting GLT is denoted $cl(T, \mathcal{F})$.*

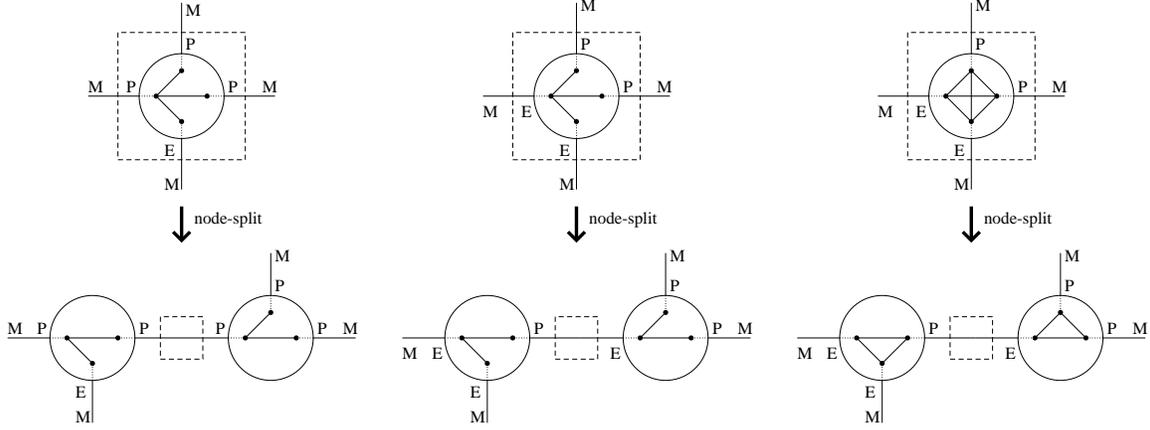


Figure 7: Node-split performed when there is a degenerate hybrid node (case 4 of Theorem 4.14, first step of Proposition 4.16). The dashed rectangle shows where the local transformation takes place, as described by the second step of Proposition 4.16 (Figure 8).



Figure 8: Insertion of vertex x in case 4 of Theorem 4.14, second step of Proposition 4.16 and when there is a unique edge with no mixed extremity (cases 5 or 6 of Theorem 4.14, Proposition 4.17). The dashed rectangle shows where the local transformation is made.

The above definition makes sense thanks to Remark 2.15 since $P^*(u) \cap E^*(u) = \emptyset$; the two node-splits corresponding to these splits can be done in any order with the same result. Figure 9 illustrates the possible local transformations at each node u .

Remark 4.19. *With Lemma 4.3 and the fact that u contains at least one mixed marker vertex whose opposite is mixed, one can easily show that $(P^*(u), V(u) \setminus P^*(u))$, respectively $(E^*(u), V(u) \setminus E^*(u))$, is a split of u if and only if $|P^*(u)| > 1$, respectively $|E^*(u)| > 1$.*

Proposition 4.20 (Case 7 of Theorem 4.14). *Let $ST(G) = (T, \mathcal{F})$ be marked with respect to a subset $N(x)$ of leaves. If $ST(G)$ contains a fully-mixed tree-edge e , then $ST(G+x)$ is obtained by:*

- *contracting, by a series of node-joins, the fully-mixed subtree of $cl(ST(G))$ into a single node u ;*
- *adding to node u a marker vertex q_x adjacent in $G(u)$ to $P(u)$ and making x, q_x 's opposite. The resulting node u is prime. See Figure 10 for an illustration of the whole process.*

Proof. First, observe that the series of node-joins is well defined by Remark 2.14. This construction is exactly the inverse of the prime-splitting construction referenced in Lemma 4.13. More precisely,

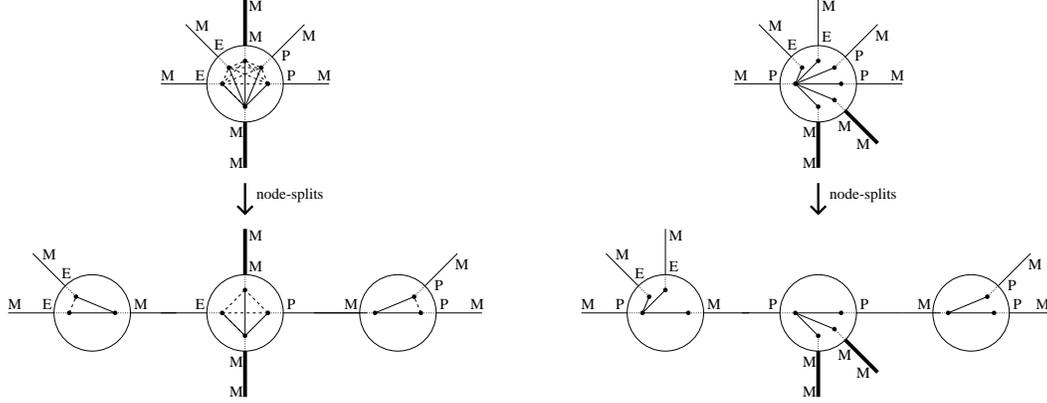


Figure 9: Cleaning of a degenerate node of the fully-mixed subtree (Definition 4.18). The left picture (as shown with dashed label-edges) applies equally to a clique or star node with a mixed centre. The right picture concerns a star node with a perfect centre. Bold edges are fully-mixed.

the GLT $cl(ST(G))$ here is exactly the GLT (T'_x, \mathcal{F}'_x) there. So the fully-mixed subtree of $cl(ST(G))$ here is the fully-mixed subtree induced by $ST(G(u_x))$ there. And the series of node-joins applied to this subtree leads to the node labelled by u_x , to which x is added naturally. \square

Throughout the rest of the paper, we will use the phrase *contraction step*, or simply *contraction*, to refer to the procedure involved in Proposition 4.20 which transforms the fully-mixed subtree of $cl(ST(G))$ into a prime node that has the new vertex x attached.

To end this section, we point out a number of observations that follow from the results in this section. First, the construction provided by Propositions 4.15, 4.16, and 4.17 applied to a distance hereditary graph (i.e. when every node is degenerate), amounts to the one provided in [23, 24]. Thus, the present construction is a generalization to arbitrary graphs.

Secondly, note that we chose to separate the cases in Theorem 4.14 for consistency with our next algorithm. But other shorter and equivalent presentations would have been possible; for instance: case 1 and case 5 (respectively case 2 and case 6) could be grouped and treated the same way as they are the only cases where there exists a PP (respectively PE) edge, with a clique-join or star-join after insertion if the edge was not unique; case 4 comes to cases 5 and 6 by making a PP or PE edge appear after splitting the node; case 1 could be considered as a trivial subcase of case 7.

Finally, the results of this subsection, together with Lemmas 4.11 and 4.12 yield the following theorem which plays an important role in our circle graph recognition algorithm [25].

Theorem 4.21. *A graph $G + x$ is a prime graph if and only if $ST(G)$, marked with respect to $N(x)$, satisfies the following:*

1. *Every marker vertex not opposite a leaf is mixed,*
2. *Let w be a degenerate node. If w is a star node, the centre of which is perfect, then w has no empty marker vertex and at most two perfect marker vertices; otherwise, w has at most one empty marker vertex and at most one perfect marker vertex.*

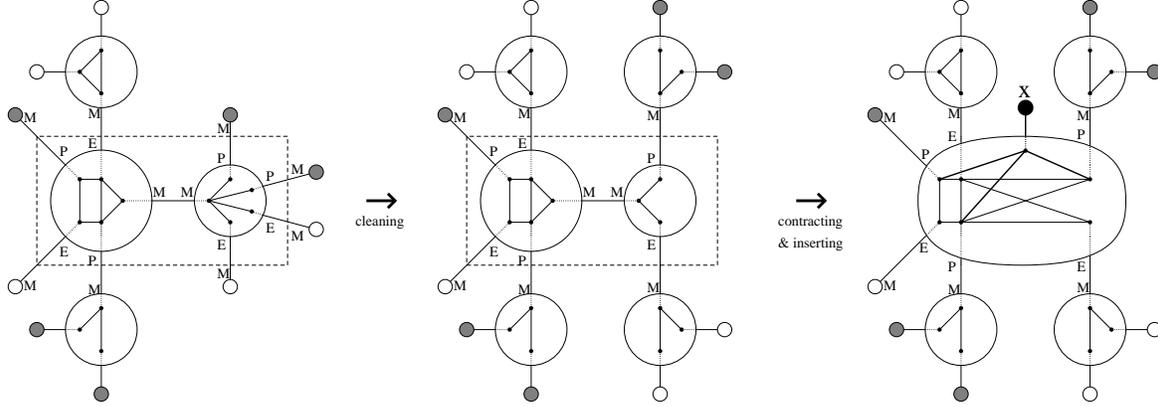


Figure 10: A complete example of the split-tree update where case 7 of Theorem 4.14 applies (Proposition 4.20). The dashed rectangle contains the fully-mixed subtree of $ST(G)$ in the left picture, and the fully-mixed subtree of $cl(ST(G))$ in the middle one.

5 An LBFS incremental split decomposition algorithm

Our combinatorial characterization, described by Theorem 4.14 and the subsequent Propositions 4.15, 4.16, 4.17 and 4.20, immediately suggests an incremental split decomposition algorithm. Our characterization makes no assumption about the order vertices are to be inserted. For the sake of complexity issues, we choose to add vertices according to an LBFS ordering σ , which we assume to be built by a preprocess. From Remark 3.2, such an ordering is compatible with the assumption made in Section 4: all iterations of the algorithm satisfy the condition that G and $G + x$ are connected. Roughly speaking, the LBFS ordering will play two crucial parts: first, it permits a costless twin test allowing us to avoid “touching” non-neighbours of the new vertex when identifying states (Subsection 5.2); second, it means that successive updates of the split-tree have an efficient amortized cost (Section 6).

As in the previous section, we assume throughout that $ST(G) = (T, \mathcal{F})$, and that leaves and marker vertices in $ST(G)$ are assigned states according to the set $N(x)$; then we consider the changes required to form $ST(G + x)$. Algorithm 3 outlines how the split-tree is updated to insert the last vertex of an LBFS ordering.

The first task consists of identifying which of the cases of Theorem 4.14 holds, at line 1 of Algorithm 3. The implementation of line 1 is by a procedure which also returns the states of the involved marker vertices (see Subsection 5.2), and hence allows us to apply the constructions provided by the propositions. At line 2, testing the uniqueness of the tree-edge e amounts to a check as to whether e is incident to a clique or a star node. This is required to discriminate between cases 1 and 5 or cases 2 and 6. More precisely, if e has two perfect extremities and is adjacent to a clique u , then all the marker vertices of u are perfect by Lemma 4.3, and then Proposition 4.15 is applied to this clique node. If e has a perfect extremity q and its opposite r is empty, and if q is the centre of a star or r is a degree-1 vertex of a star, then by Lemma 4.3, this star has all of its marker vertices empty except the centre, which is perfect, and then Proposition 4.15 is applied to this star node. These tests at line 2 can be done in constant time in the data-structure we use, as

Algorithm 3: Vertex insertion

Input: A graph G , a vertex $x \notin V(G)$ which is the last vertex in an LBFS ordering of $G + x$, and the split-tree $ST(G) = (T, \mathcal{F})$

Output: The split-tree $ST(G + x)$

- 1 Determine whether $ST(G)$ contains either a tree-edge neither of whose extremities is mixed, or a hybrid node, or a fully-mixed subtree;
if $ST(G)$ contains a tree-edge e neither of whose extremities is mixed **then**
 - 2 | **if** e is unique **then** update $ST(G)$ according to Proposition 4.17;
 | **else** update $ST(G)$ according to Proposition 4.15;

 - if** $ST(G)$ contains a hybrid node u **then**
 - 3 | **if** u is degenerate **then** update $ST(G)$ according to Proposition 4.16;
 | **else** update $ST(G)$ according to Proposition 4.15;

 - if** $ST(G)$ contains a fully-mixed subtree **then**
 - 4 | compute and update $cl(ST(G))$ according to Proposition 4.20;
-

well as the updates required in these simplest cases (Proposition 4.15 and Proposition 4.17). They will not be considered again in the implementation.

Now, this section fills out the framework by specifying procedures for the state assignment, node-split, node-join, cleaning, and contraction involved at lines 1, 3, and 4 of Algorithm 3. We first describe our data-structures which is partly based on *union-find* [9]. We then provide a complexity analysis of the insertion algorithm parameterized by elementary union-find requests. An amortized complexity analysis is developed in the next section.

5.1 The data-structure

In order to achieve the announced time complexity, we implement a GLT (T, \mathcal{F}) with the well-known union-find data-structure [9], making T a rooted tree. There are two reasons for this choice. First, identifying empty and perfect subtrees is easier if the tree T is rooted. Second, in the contraction step, we'll need to union the neighbourhoods of two nodes to perform a node-join. We first present how the tree T will be encoded with a union-find data-structure. We then detail how each node and its labels are represented.

A union-find data-structure maintains a collection of disjoint sets. Each set maintains a distinguished member called its *set-representative*. Union-find supports three operations:

1. **initialize**(x): creates the singleton set $\{x\}$;
2. **find**(x): returns the set-representative of the set containing x ;
3. **union**(S_1, S_2): forms the union of S_1 and S_2 , and returns the new set-representative, chosen from amongst those of S_1 and S_2 .

The initialization step takes $O(1)$ time, and a combination of k union and find operations takes time $O(\alpha(N) \cdot k + N)$, where N is the number of elements in the collection of disjoint sets and α is the inverse Ackermann function [9]. The complexity of the algorithms described in this section will be parameterized by `initialize-cost`, `find-cost`, `union-cost` the respective costs of the above requests.

Our algorithm will store a GLT (T, \mathcal{F}) as a *rooted GLT* where a leaf of T will serve as *the root* (it is the leaf corresponding to the first inserted vertex). Each node or leaf of T , except the root, has a *parent pointer* to its parent (which is a node or the root leaf) in T with respect to the root. To each prime node is associated a *children-set* containing the set of its children in T with respect to the root (nodes or leaves). The children-sets of prime nodes form the collection of disjoint sets maintained by the union-find data-structure. Every children-set has a set representative, which is a child of the node in T . A parent pointer may be active or not. The nodes or leaves with an *active* parent pointer are: the child of the root, the children of degenerate nodes (clique or star), and the nodes or leaves that are the set representative of the children-set of a prime node. A non-active parent pointer is just one that will never be used again; there is no need to update information for it.

Remark 5.1. *A traversal of a rooted GLT (T, \mathcal{F}) can be implemented in time $O((1 + \text{find-cost}) \cdot |T|)$.*

Union-find is required only to update the tree structure (child-parent relationship) efficiently. As the node-splits only apply to degenerate nodes (lines 3, 4), union-find is not required here. Union operations are performed after the cleaning step, during the contraction step (line 4).

Remark 5.2. *The data-structure described here concerns a split-tree, whose labels are either prime or degenerate, since after each step of the construction it is such a GLT that will be obtained. Still, we need to allow node-joins in the data-structure. In what follows, during the successive node-joins in the contraction step (Proposition 4.20), the GLT has one non-degenerate node whose label graph will eventually become prime only after the final insertion step. The data-structure for such a GLT remains the same by recording the type of this non-degenerate node as prime, and dealing with it the same way as a prime node.*

It is important to note that removing elements from sets is not supported in the union-find data-structure. It follows that when a node-join is performed and the children-set of a node u' is unioned with the children-set of its parent u , the node object corresponding to u' still exists in the children-set of u . As we will see later, the persistence of these *fake* nodes is not a problem. Indeed, their total number will be suitably bounded, they will never be selected again as set representatives, and the data-structure we develop below guarantees that they will never be accessed again. In particular, no active parent pointer points to a fake node. That is why the children-set of a prime node may strictly contain its set of children in T .

To complete our data-structure we define a data-object for every node and leaf in the rooted GLT. These data-objects will maintain several fields:

- We already mentioned that every node and leaf, except the root, has a parent pointer. Each node u has a distinguished marker vertex, called the *root marker vertex*, which is the extremity of the tree-edge between u and its parent. Every node maintains a pointer to its root marker vertex. As already implied, we need to store the *type* of every node (prime, clique, or star), and we also store the *number of its children*.

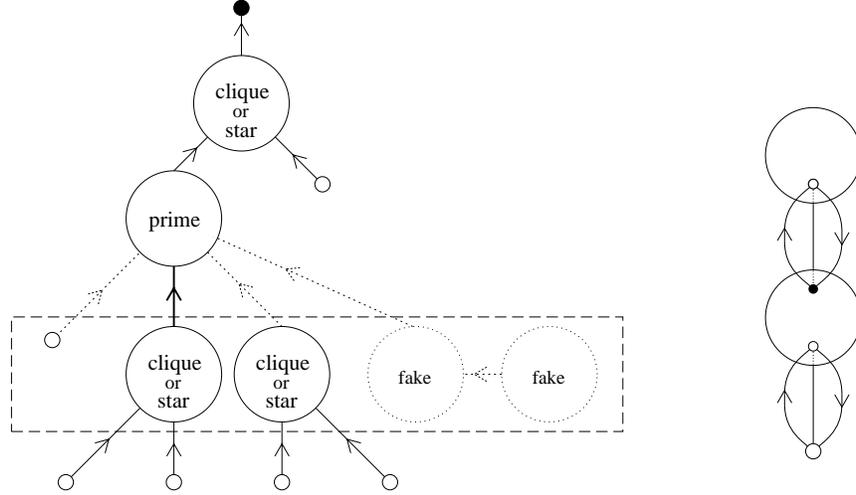


Figure 11: Representation of a rooted GLT with the union-find data-structure. In the left picture: full arrows going up represent active parent pointers, the dashed arrows are non-active; the dashed rectangle represents the children-set of the prime node; small circles outside nodes represent leaves, the black one is the root. In the right picture: small circles inside nodes represent marker vertices, the black one is a root marker vertex; curved arrows represent pointers to the opposite marker vertex or leaf.

On the top of that, depending on its type, every node u maintains the following fields:

- if u is prime: an *adjacency-list representation* of $G(u)$; a pointer to the *last marker vertex* in $\sigma[G(u)]$; a pointer to its *universal marker vertex* (if it exists);
- if u is degenerate: a *list of its marker vertices* $V(u)$; and a pointer to its *centre* if it is a star;

To each leaf and marker vertex, we associate:

- a pointer, called the *opposite pointer*, to its opposite marker vertex; a field for its *perfect-state* (at each new vertex insertion, perfect states, but no other state, will be computed and recorded, and the content of these fields from previous vertex insertions is not reused); and, for every root marker vertex, a pointer, called the *node pointer*, toward the node to which it belongs.

Figure 11 illustrates this rooted GLT data-structure. For instance, notice how a prime node accesses its children-set using this data-structure: pick a non-root marker vertex of the node, then its opposite marker vertex, then the node to which this marker vertex belongs, then the `find` on this node gives the set-representative of the corresponding children-set.

5.2 State assignment and case identification

Prior to any update, a preprocessing of the split-tree is required to identify which of the cases of Theorem 4.14 holds. This preprocessing is based on state assignment and tree traversals. The LBFS ordering will play an important role here. The procedure we use for the empty subtrees detection (Algorithm 4) differs from that for perfect subtrees (Algorithm 5).

5.2.1 Empty subtrees

If a marker vertex or leaf q is empty, then, by definition, $T(q)$ contains no leaf in $N(x)$, and thus it will be unchanged under x 's insertion. For the sake of complexity issues, we will want to avoid “touching” q and any part of $T(q)$. We are fortunate that identifying such empty subtrees can be simulated indirectly:

Lemma 5.3. *Consider $ST(G) = (T, \mathcal{F})$, and let $T(N(x))$ be the smallest connected subtree of T spanning the leaves $N(x)$. Then q is an empty leaf or an empty marker vertex if and only if $T(q)$ and $T(N(x))$ are node disjoint.*

Proof. It's important to recall that if $q \in V(u)$, then the node u is not part of $T(q)$. If q is empty, then $L(q) \cap N(x) = \emptyset$, meaning $T(q)$ and $T(N(x))$ must be disjoint. If $T(q)$ and $T(N(x))$ are disjoint, then $L(q) \cap N(x) = \emptyset$, meaning q is empty. \square

Algorithm 4: [23, 24] Detection of empty subtrees: computing the smallest connected subtree spanning a set of leaves

Input: A tree T rooted at a leaf and a subset $N(x)$ of its leaves (assuming $|N(x)| > 1$).

Output: The tree $T(N(x))$, the smallest connected subtree of T spanning the leaves $N(x)$.

Mark each leaf of $N(x)$ as *active* (other nodes and leaves are considered inactive);

while [the root is not visited and there are at least two active leaves or nodes] **OR** [the root is visited and there is at least one active leaf or node] **do**

 Let L be the current set of active leaves or nodes;

foreach element of L , u **do**

u is no longer *active*, it becomes *visited*;

if u is not the root and its parent is not visited **then** u 's parent is marked *active*;

end foreach

end while

Let T' be the subtree of T induced by the visited leaves and nodes;

if t' , the root of T' , has a unique visited child but t' does not belong to $N(x)$ **then**

 remove in T' the path from t' to the closest node with at least two visited children;

return T' ;

We can compute $T(N(x))$ using the procedure specified in [23, 24], which is repeated here as Algorithm 4. It was proved in [23, 24] that a call to Algorithm 4 runs in time $O(|T(N(x))|)$, assuming each node maintains a pointer to its parent. Therefore, given the data-structure proposed above, a `find()` request is needed to move from a node to its parent, when prime. So the following holds:

Lemma 5.4. *Given a GLT (T, \mathcal{F}) , Algorithm 4 returns a subtree of T that is node disjoint from every empty subtree, and runs in time $O((1+\text{find-cost}) \cdot |T(N(x))|)$.*

5.2.2 Perfect subtrees

As T is rooted, the subtree $T' = T(N(x))$ has a root which is a leaf or a node of T . For every node u of T' , the marker vertices of u which are the extremity of a tree-edge in T' form the set

$NE(u)$ (recall Definition 4.1). The next task is to identify the perfect subtrees and derive the case identification used at line 1 of Algorithm 3. Our procedure, described in detail as Algorithm 5, outputs either a tree-edge or a hybrid node, or the fully-mixed subtree of $ST(G)$. (Recall that a fully-mixed subtree is maximal, by definition.) It works in three main steps:

1. First it traverses the subtree $T(N(x))$ in a bottom-up manner to identify the pendant perfect subtrees: for each non-leaf node u , we test if the marker q opposite u 's root marker is perfect and if so remove u from T' and move to u 's parent.
2. Then, if the root of the remaining subtree T' has a unique child v in T' , we check whether v 's root marker vertex r is perfect and if so remove the root from T' and move to v . This test is repeated until the current root of T' neighbours at least two nodes in T' or is the unique remaining node of T' .
3. In the former case, we are done and the resulting tree T' is fully-mixed. In the latter case, we still need to test whether the remaining node is hybrid or contains a marker vertex opposite a perfect leaf, in which case the output is this edge. As we will see, using the LBFS ordering allows us to test only two marker vertices.

The next remark explains how one can test whether a given marker vertex is perfect and will be used in the bottom-up and top-down traversal of $T(N(x))$.

Remark 5.5. *Let $q \in V(u)$ be a marker vertex in $ST(G)$, and let r be q 's opposite. Then r is perfect if and only if:*

1. $P(u) = N_{G(u)}(q)$, or $P(u) = N_{G(u)}[q]$; and
2. $NE(u) \setminus P(u) = \emptyset$ or $NE(u) \setminus P(u) = \{q\}$.

Remark 5.5 must not be applied, at the third step of our procedure, to every marker vertex of the unique remaining node u . Indeed, consider $q, q' \in V(u)$, and let r and r' be their opposites, respectively. To test if r and r' are perfect using Remark 5.5 requires us to test if $P(u) = N(q)$ and $P(u) = N(q')$. But if $N(q) \cap N(q') \neq \emptyset$, then this involves ‘‘touching’’ marker vertices of u multiple times. In general, we cannot bound the number of times marker vertices in u will have to be ‘‘touched’’.

The solution for a degenerate node follows from the next lemma. In the case of a prime node, we will use the LBFS Lemmas 3.4 and 3.6 of Section 3.

Lemma 5.6. *Let u be a degenerate node of $ST(G)$, the marker vertices of which are all either perfect or empty (i.e. $P(u) = NE(u)$). There exists a marker vertex $q \in V(u)$ whose opposite r is perfect if and only if one of the following conditions holds:*

1. $P(u) = V(u)$ (in this case, if u is a clique then any $q \in V(u)$ is suitable, and if u is a star then q is its centre);
2. $P(u) = V(u) \setminus \{q\}$ and, when u is a star, q is the centre of u ;
3. $P(u) = \{c\}$ and u is a star with centre c (in this case any $q \in V(u) \setminus \{c\}$ is suitable);
4. $P(u) = \{c, q\}$ and u is a star with centre c .

Proof. Choose some $q \in V(u)$ and let r be its opposite marker vertex. Assume that r is perfect. Then if u is a clique or a star with centre q , all marker vertices in $V(u) - \{q\}$ are accessible descendants of r and are therefore perfect, by Lemma 4.3-1. Therefore either items 1 or 2 of the lemma hold.

So assume that u is a star with centre $c \neq q$. Then c is an accessible descendant of r , and all marker vertices in $V(u) - \{c, q\}$ are inaccessible descendants of r . Thus, c is perfect and the marker vertices in $V(u) - \{c, q\}$ are empty, by Lemma 4.3-1. It follows that either items 3 or 4 of the lemma hold.

Now assume that one of items 1-4 of the lemma holds. If items 1 or 2 hold, then all marker vertices in $V(u) - \{q\}$ are accessible descendants of r , and all are perfect. So by Lemma 4.3-1, r is perfect as well. And if items 3 or 4 hold, then only c is perfect, and only c is an accessible descendent of r . It follows that r is perfect, once again by Lemma 4.3-1. \square

Applying Lemma 5.6 at node u is straightforward as soon as $P(u)$ has been computed. Notice that in cases 2 and 3, the marker vertex q is empty, but it can be determined without considering other empty marker vertices. Hence at most one empty marker vertex is involved in this step of the procedure.

Let us now turn to prime nodes. First observe the following remark, which is a straightforward application of the definitions:

Remark 5.7. *Let $q \in V(u)$ be a marker vertex in $ST(G)$, and let r be its opposite. Let t be a marker vertex added to u , made adjacent precisely to $P(u)$. Then r is perfect if and only if q and t are twins.*

The next lemma merely translates Lemma 3.4 to the split-tree; its corollary is the important result for our purposes:

Lemma 5.8. *Let σ be an LBFS of the connected graph $G + x$ in which x appears last, and let u be a prime node in $ST(G)$. Let r be the opposite of some $q \in V(u)$. If r is perfect, then q is universal in $G(u)$ or q appears last in $\sigma[G(u)]$.*

Proof. Let u' be the same as u but with a new marker vertex t adjacent precisely to $P(u)$. Consider the GLT (T', \mathcal{F}') that results from replacing u with u' , and adding a new leaf ℓ opposite t . Let σ_ℓ be the same as σ but with x replaced by ℓ . Since t is only adjacent to $P(u)$, we have $N(\ell) \subseteq N(x)$. Therefore σ_ℓ is an LBFS of $G + \ell$ in which ℓ appears last, and $\sigma_\ell[G(u')]$ is an LBFS of $G(u')$ in which t appears last, by Lemma 3.6 applied to the split-tree.

Assume that r is perfect. Then q and t are twins, by Remark 5.7. Therefore u' is not prime. But recall that u was prime. So by Lemma 3.4, either q is universal in $G(u)$ or it is the penultimate vertex in $\sigma_\ell[G(u')]$. If it is the penultimate vertex in $\sigma_\ell[G(u')]$, then it must be the last vertex in $\sigma[G(u)]$. \square

Corollary 5.9. *Let σ be an LBFS of $G + x$ in which x appears last. Let ℓ be a leaf adjacent to a prime node u in $ST(G)$, and let $q \in V(u)$ be ℓ 's opposite. Then ℓ is perfect if and only if q is universal in $G(u)$ or q appears last in $\sigma[G(u)]$.*

Proof. A direct consequence of Remark 5.7 and Lemma 5.8. \square

Therefore to perform the third step of the case identification procedure, at most two marker vertices of the remaining prime node u can have opposites that are perfect. Thus, Remark 5.5 needs to be applied at most twice. Remember that our data-structure keeps track of these two marker vertices.

Lemma 5.10. *Given an LBFS ordering of a connected graph $G + x$ and the split-tree $ST(G)$, Algorithm 5 returns:*

- a tree-edge e , one of whose extremities is perfect and the other is either empty or perfect, if case 1, 2, 5 or 6 of Theorem 4.14 applies;
- a hybrid node u , if case 3 or 4 of Theorem 4.14 applies;
- the full-mixed subtree T' of T , if case 7 of Theorem 4.14 applies;

It can be implemented to run in time $O((1+\text{find-cost}) \cdot |T(N(x))|)$.

Proof. By Lemma 5.4, we know that $T(N(x))$, and thus T' , is node disjoint from every empty subtree of $ST(G)$. Clearly, before the while loop at line 3, the current subtree T' is disjoint from every pendant perfect subtree of $ST(G)$. Thus, if the root of the tree belongs to a perfect subtree $T(p)$, then the nodes of $T(p) \cap T'$ can only form a path of T' : otherwise $T(p)$ should contain a node u with two non-root marker vertices q and r which are neither perfect nor empty, as $T(q)$ and $T(r)$ have not been removed so far, contradicting the fact that $T(p)$ is perfect.

Concerning the correctness of the third step (case identification), first observe that if, at line 4, T' contains more than one node, then every tree-edge in T' is fully-mixed (otherwise one of its extremities would have been removed during the tree traversals). Thus case 7 of Theorem 4.14 holds. So assume that T' consists of one node. It follows from Lemma 5.6, Remark 5.7 and Corollary 5.9, that a single tree-edge is returned by the algorithm if there exists a tree-edge with a perfect extremity and the other extremity either perfect or empty. Notice that we do not test the uniqueness of such a tree-edge. If none of the previous cases applies, then, by Theorem 4.14, $ST(G)$ contains a hybrid node which is correctly identified by the algorithm. This corresponds to cases 3 and 4 of Theorem 4.14.

By Remark 5.1, the cost of performing the tree traversals is $O((1 + \text{find-cost}) \cdot |T(N(x))|)$. During the algorithm, either Remark 5.5 or Lemma 5.6 is applied a constant number of times at each node. The cost to apply each remark/lemma in the context of the data-structure presented above is clearly $O(|NE(u)|)$, where $NE(u) = P(u) \cup M(u)$ (recall Definition 4.1). But every $q \in NE(u)$ has its corresponding edge in $T(N(x))$, by Lemma 5.3 and the definition of $NE(u)$. So the total cost of applying Remark 5.5 and Lemma 5.6 is $O(|T(N(x))|)$. Of course, once q is found to be perfect, then u can be removed from T' in constant time. \square

Before describing how the split-tree is updated in each of the different cases, let us point out how the states (perfect, empty, mixed) of marker vertices are computed (or not) during Algorithm 5:

Remark 5.11. *Algorithm 5 assigns a state to a marker vertex q and updates the data-structure accordingly only if q is perfect. From these recorded perfect-state fields, the states of all marker vertices involved in the output of Algorithm 5 can be deduced.*

It is not a problem to avoid explicitly computing the state of all the marker vertices. Indeed, computing them would affect our complexity. Moreover, notice that in every case (see Propositions 4.15, 4.16, 4.17 and 4.20) the knowledge of the perfect marker vertices is enough to determine

Algorithm 5: Detection of perfect subtrees, and split-tree case identification

Input: The rooted split-tree $ST(G) = (T, \mathcal{F})$ and an LBFS ordering σ of $G + x$ where x is the last vertex.

Output: A tree-edge e , with one extremity perfect and the other empty or perfect (if one exists); or a hybrid node u (if one exists); or the fully-mixed subtree T' of T (if one exists).

```
1  $T' \leftarrow T(N(x))$ , the result of Algorithm 4 with input  $T$  and  $N(x)$ ;  
   Set the non-root marker vertices opposite leaves of  $N(x)$  to perfect (these markers belong to  
   nodes of  $T'$ );  
  
   // bottom-up traversal: discard the pendant perfect subtrees  
   if  $T'$  contains more than one node then  
2   | while there exists a non-processed node  $u$  in  $T'$  all of whose children in  $T'$  are leaves do  
   |   | let  $q$  be the (non-root) marker vertex opposite  $u$ 's root marker vertex;  
   |   | determine  $q$ 's state by applying Remark 5.5;  
   |   | if  $q$  is perfect then remove  $u$  from  $T'$ ;  
   |   end while  
  
   let  $u$  be the root of  $T'$ ;  
  
   // top-down traversal: discard the perfect subtree containing the root  
3 while node  $u$  has exactly one non-leaf child  $v$  in  $T'$  and no fully-mixed edge has been  
   identified do  
   | apply Remark 5.5 to determine the state of the root marker vertex  $q$  of  $v$ ;  
   | if  $q$  is perfect then remove  $u$  from  $T'$  and  $u \leftarrow v$ ;  
   | else the tree-edge  $uv$  is fully-mixed;  
   end while  
  
   // case identification  
4 if  $T'$  contains a unique node  $u$  then  
   | if  $u$  is degenerate then  
   |   | apply Lemma 5.6 to determine if there is a  $q \in V(u)$  whose opposite  $r$  is perfect;  
   |   | if such a  $q$  exists then return the tree-edge  $e$  of  $T$  whose extremities are  $q$  and  $r$ ;  
   |   | else return the hybrid node  $u$ ;  
   | else  
   |   | // twin-test in a prime node  
   |   | let  $q$  be the last marker vertex in  $\sigma[G(u)]$  and let  $r$  be its opposite;  
   |   | let  $q'$  be  $u$ 's universal vertex (if it exists), and let  $r'$  be its opposite;  
   |   | apply Remark 5.5 to determine the states of  $r$  and  $r'$ ;  
   |   | if  $r$  (respectively  $r'$ ) is perfect then  
   |   |   | return the tree-edge  $e$  of  $T$  whose extremities are  $q$  and  $r$  (respectively  $q'$  and  $r'$ );  
   |   |   | else return the hybrid node  $u$ ;  
   | else return the fully-mixed subtree  $T'$ ;
```

the state of every other marker vertex which will be affected in the successive steps of the updates. For example, in a hybrid node, the non-perfect marker vertices are by definition empty. Similarly if $ST(G)$ contains a fully-mixed subtree T' , then a marker vertex of a node of T' is empty if and only if it is not perfect and not incident to a tree-edge of T' . It follows that, once Algorithm 5 has been performed, we can conclude that the state of every useful marker vertex has been determined.

5.3 Node-split and cleaning

The node-split operation (see Definition 2.12) is required when cases 4 and 7 of Theorem 4.14 hold. Case 4, existence of a degenerate hybrid node u (Proposition 4.16), only requires the node-split of u according to $(P^*(u), V(u) \setminus P^*(u))$. Case 7 potentially implies a large number of node-splits since, before the contraction step, the cleaning of $ST(G)$ is necessary (Proposition 4.20). Notice that degenerate nodes are the only nodes that are ever node-split.

To be as efficient as possible, to perform a node-split we won't create two new nodes as seemingly required by the definition. Instead, we will reuse the node being split so that only one new node has to be created. This is presented in Algorithm 6.

Algorithm 6: Node-split(v, A, B)

Input: A rooted GLT with a node v such that $G(v)$ contains the split (A, B) having frontiers A' and B' .

Output: The rooted GLT with nodes u and u' resulting of the node-split of v with respect to (A, B) .

replace the vertices in A with a new marker vertex q adjacent precisely to B' ;
call the result u' ;

create a new node u consisting of the vertices in A , plus one new marker vertex r adjacent precisely to A' ;

add an internal tree-edge between u and u' having extremities q and r ;

if the root marker vertex of v belongs to A **then** make u' a child of u ;

else make u a child of u' ;

return the resulting GLT with u, u' and their child relation;

Lemma 5.12. *Algorithm 6 performs a node-split (A, B) for a degenerate node in time $O(|A|)$.*

Proof. The correctness follows from the definitions. The time complexity is obvious as well from a simple examination of our data-structure. Recall that every child of a degenerate node maintains a parent pointer (unlike for prime nodes which use the union-find). Depending on whether the root marker vertex of v belongs to A or B , u' becomes a child of u or vice-versa. The root marker vertices (and their respective node pointers) of the resulting nodes are updated accordingly. As these two nodes are degenerate, they need to have a list of their marker vertices: u' inherits the list of v in which A has been removed plus the new marker vertex q , while u 's list is created and contains $A \cup \{r\}$. Meanwhile, the marker vertices of A update their node pointer. This work requires $O(|A|)$ time. Other information such as type of the node, number of children, pointer to the centre (if it is a star) or opposite pointer, perfect-states, is easily updated in constant time. \square

Cleaning was introduced in Definition 4.18 along with the notation $cl(ST(G))$. Notice that it amounts to repeated application of the node-split operation. The cleaning step will proceed according to Algorithm 7. To determine if a degenerate node needs to be node-split according to $(E^*(u), V(u) \setminus E^*(u))$, Algorithm 7 takes advantage of the equivalence: $E^*(u) = V(u) \setminus (P^*(u) \cup M(u))$, where $P^*(u)$ and $M(u)$ are deduced directly from perfect-states fields and the fully-mixed subtree structure. As before, the reason is one of efficiency; we want to avoid “touching” empty subtrees. To this end, it is important that the node-split is performed only “touching” perfect and mixed marker vertices of $V(u) \setminus E^*(u)$. This is the reason for defining Algorithm 6 as we did. The rest of Algorithm 7 is a direct implementation of the definition.

Algorithm 7: Cleaning($(T, \mathcal{F}), T'$)

Input: The rooted split-tree $ST(G) = (T, \mathcal{F})$ marked with respect to $N(x)$ and the fully-mixed subtree T' of T .
Output: The rooted GLT $cl(ST(G))$ resulting from the cleaning of (T, \mathcal{F}) and the fully-mixed subtree T_c of $cl(ST(G))$.

Assume all nodes are marked *unvisited*;
 $T_c \leftarrow T'$;

foreach *unvisited degenerate node* v **in** T_c **do**

if $|P^*(v)| > 1$ **then**

 node-split v according to the split $(P^*(v), V(v) \setminus P^*(v))$, using Algorithm 6;
 mark *visited* the two nodes that result from the node-split;
 keep in T_c only the node containing the marker vertices in $V(v) \setminus P^*(v)$;

end foreach

reset all the marks in T_c ;

foreach *unvisited degenerate node* v **in** T_c **do**

if $|V(v) \setminus (|P^*(v)| + |M(v)|) > 1$ **then**

 node-split u according to the split $(V(v) \setminus E^*(v), E^*(v))$, using Algorithm 6;
 mark *visited* the two nodes that result from the node-split;
 keep in T_c only the node containing the marker vertices in $V(v) \setminus E^*(v)$;

end foreach

return the updated GLT, $cl(ST(G))$ together with its fully-mixed subtree T_c ;

Lemma 5.13. *Given the split-tree $ST(G) = (T, \mathcal{F})$ marked with respect to $N(x)$ and the fully-mixed subtree T' of T , Algorithm 7 computes $cl(ST(G))$ together with its fully-mixed subtree, and it runs in time $O((1 + \text{find-cost}) \cdot |T(N(x))|)$.*

Proof. Correctness is clear as Algorithm 7 traverses T' twice. In both traversals, a single test is performed at each node, and then if it succeeds, a node-split is applied. Recall that $(P^*(u), V(u) \setminus P^*(u))$, respectively $(E^*(u), V(u) \setminus E^*(u))$, is a split of u if and only if $|P^*(u)| > 1$, respectively $|E^*(u)| > 1$, by Remark 4.19. The resulting subtree T_c is the fully-mixed subtree of $cl(ST(G))$ by construction.

The traversals require time $O((1 + \text{find-cost}) \cdot |T(N(x))|)$. To perform the test required by Algorithm 7, we first need to compute $P^*(u)$. But this clearly can be done in time bounded by $O(|P(u)|)$, which is bounded by $O(|NE(u)|)$. So by the argument already used in the proof of Lemma 5.10, the total cost of computing the sets $P^*(u)$ is $O(|T(N(x))|)$. Once $P^*(u)$ is computed, the test performed at each node can be carried out in constant time. If the degenerate node u is node-split during the first pass, then the set $P^*(u)$ plays the role of A in the input to Algorithm 6; if u is split during the second pass, then the set $V(u) \setminus E^*(u)$ plays the role of A . The size of both of these sets is bounded by $|NE(u)|$. But every $q \in NE(u)$ has its corresponding edge in $T(N(x))$, by Lemma 5.3 and the definition of $NE(u)$. So the total cost of the node-splits performed during cleaning is $O(|T(N(x))|)$, by Lemma 5.12. \square

No `union()` operation has so far been needed; it has been sufficient to employ `find()` operations while traversing various trees.

5.4 Node-joins and contraction

Contraction amounts to repeated application of the node-join, which requires the `union()` operation. In the same way that we reused one node for the node-split, we will want to reuse one node for the node-join (we arbitrarily choose to reuse the parent node). Algorithm 8 provides the details of the implementation of the node-join between a node u' and its parent u . Notice that in the case where u' is a star and its root marker vertex q' has degree one, a node-join is performed differently. Here we reuse the marker vertex q of u adjacent to u' to play the role of the unique neighbour t of q' . We do so for reasons of efficiency that will become clear later. In other cases, the label-edges adjacent to the two marker vertices disappearing in the node-join are not reused.

The node resulting from the node-join of u and u' in Algorithm 8 may be neither prime nor degenerate. In the data-structure of the resulting rooted GLT, this resulting node is nevertheless marked as prime (standing for non-degenerate), as explained in Remark 5.2, since it will finally be prime after insertion of the new vertex. The children-set associated with this node is created at line 3, and may require several `initialize()` and `union()` operations when u and/or u' were degenerate and thus were not associated with a children-set.

Lemma 5.14. *Let u and u' be two adjacent nodes of a GLT and let q and q' be the respective extremities of the tree-edge between u and u' . Algorithm 8 computes the GLT resulting from the node-join of u and u' . It can be implemented to run in time $\text{join-cost} = O(\text{\#new-label-edge}) + \text{tree-update-cost}$ where:*

- *`\#new-label-edge` denotes the number of newly created label-edges (i.e. at line 2: the number of neighbours of q multiplied by the number of neighbours of q').*
- *the `tree-update-cost` amounts to*
 - $O(d \cdot (\text{union-cost} + \text{initialize-cost}))$ if u is a degenerate node with d children, plus
 - $O(d' \cdot (\text{union-cost} + \text{initialize-cost}))$ if u' is a degenerate node with d' children, plus
 - $O(\text{union-cost})$ to perform the union of children-sets of u and u' .

Proof. The correctness easily follows from the definitions. The resulting node is assigned the prime type and therefore is associated with a children-set, and its graph label has to be represented

Algorithm 8: Node-join(u, u')

Input: A rooted GLT with two adjacent nodes u and u' , where u' is the child of u .

Output: The rooted GLT resulting from the node-join of u and u' .

let $q \in V(u)$ and $q' \in V(u')$ be the extremities of the tree-edge between u and u' ;

if u' is a star node and its root marker vertex q' has degree one **then**

```
1 | foreach non-neighbour  $t'$  of  $q'$  in  $G(u')$  do  
  |   move  $t'$  to  $G(u)$  and make it adjacent to  $q$ ;  
  |   let  $v$  be the child of  $u'$  containing the marker vertex opposite  $t'$ ;  
  |   update  $v$ 's parent pointer to  $u$ ;  
  | end foreach
```

2 **else**

```
  | move all the marker vertices of  $G(u')$  except  $q'$  to  $G(u)$  and remove  $q$ ;  
  | add adjacencies in  $G(u)$  between every neighbour of  $q'$  and every neighbour of  $q$ ;
```

```
  | if  $u'$  is prime then
```

```
    | let  $v$  be the representative of the children-set of  $u'$ ;
```

```
    | update the parent pointer of  $v$  to  $u$ ;
```

```
  | else update the parent pointer of every child  $v$  of  $u'$  to  $u$ ;
```

```
3 create a single children-set containing the children of  $u$  and  $u'$  (by the way of a series of  
  initialize() and union());
```

```
return the resulting rooted GLT;
```

by an adjacency list. Concerning the series of `union()` and `initialize()` requests to build the children-set: if a degree d degenerate node is involved in a node-join, then the cost to create a set containing its children amounts to the cost of d `initialize()` and d `union()` requests. Concerning the adjacency-lists: we first create those not already present at degenerate nodes. Then the existing ones can be combined to create one for the result of the node-join. This can be done in the obvious way. \square

Recall that node-joins on a given set of tree-edges can be performed in any order (Remark 2.14). But to ease the amortized time complexity (developed in Section 6), during the contraction step, different types of node-joins are performed before others. This is reflected in Algorithm 9, which separates the node-joins into three phases, defined as *phase 1*, *phase 2*, and *phase 3* node-joins, and dealing with different types of node-joins.

Algorithm 9: Contraction($(T, \mathcal{F}), T'$)

Input: The rooted GLT $(T, \mathcal{F}) = \text{cl}(ST(G))$ and the fully-mixed subtree T' of T
Output: The rooted GLT resulting from the contraction of T' into a single node u to which the new leaf x has to be attached, with x 's opposite made adjacent in $G(u)$ precisely to $P(u)$.

```

// Phase 1 node-joins
foreach star node  $u$  in  $T'$  whose root marker vertex is its centre do
    perform node-join( $u, u'$ ) (Algorithm 8) for every non-leaf child  $u'$  of  $u$  in  $T'$ ;
    update  $T'$  accordingly;
end foreach

// Phase 2 node-joins
foreach node  $u'$  in  $T'$  whose root marker vertex  $r$  has degree 1 do
    if the parent node  $u$  of  $u'$  is not a leaf and is in  $T'$  then perform node-join( $u, u'$ )
        (Algorithm 8);
    update  $T'$  accordingly;
end foreach

// Phase 3 node-joins
recursively perform node-joins to contract  $T'$  into a single node  $u$ , using Algorithm 8;

// New vertex insertion
add a marker vertex  $q$  to  $u$ , adjacent precisely to  $P(u)$ , then make  $x$  opposite  $q$ ;
let  $u + x$  be the resulting node and mark  $q$  as the last marker vertex of  $\sigma[G(u + x)]$ ;
 $x$ 's parent is  $u + x$ ;
return the resulting rooted GLT;

```

Lemma 5.15. *If $ST(G)$ contains a fully-mixed subtree, given $\text{cl}(ST(G)) = (T, \mathcal{F})$ and its fully-mixed subtree T' , Algorithm 9 computes $ST(G + x)$. It can be implemented to run in time:*

$$O(|T(N(x))| \cdot (1 + \text{find-cost}) + k \cdot \text{join-cost} + \text{intialize-cost} + \text{union-cost})$$

where k is the number of fully-mixed tree-edges of T' .

Proof. From Remark 2.14, the GLT resulting from the node-joins between nodes incident to the tree-edges of T' is independent of the order in which they are applied. If u is the node resulting from the contraction of T' (together with the new vertex insertion), then observe that $\cup_{q \in P(u)} A(q) = N(x)$. It follows that the accessibility graph of the resulting GLT is $G + x$. Now from Proposition 4.20, u is prime thereby showing that Algorithm 9 computes $ST(G + x)$ since its result is reduced.

The nodes participating in each phase can be located by performing a single pass over the tree. This traversal can be performed in time $O((1 + \text{find-cost}) \cdot |T(N(x))|)$. Once the nodes participating in each phase have been located, the node-joins can proceed. The cost of each one is described by Lemma 5.14. The outcome of these node-joins is a single node u . At this point, all of u 's children are organized into a children-set.

The new vertex x is made a neighbour of u , which in our situation means it is made a child of u . This requires x to be added to the set that represents u 's children. To do so we need one `initialization()` operation and one `union()` operation. The opposite of x is a new marker vertex q added to $G(u)$ and made adjacent precisely to $P(u)$. Recall that our split-tree algorithm inserts vertices according to an LBFS ordering, say σ . Notice that q clearly becomes the last vertex in $\sigma[G(u) + q]$. So given the data-structure assumed earlier, the cost of adding q is $O(|P(u)|)$. But every marker vertex in $NE(u)$ has its corresponding edge in $T(N(x))$, by Lemma 5.3 and the definition of $NE(u)$. Therefore the total cost of adding q is $O(|T(N(x))|)$. \square

At this point of the paper, the reader can completely compute the split decomposition of a graph with our algorithm. It is the number of node-joins involved in successive uses of Lemma 5.15 for contraction that prevents us from concluding its running time. We have already seen one example where the number of node-joins required by contraction is linear in the size of the split-tree (see Figure 5). But later we emphasized that this was worst-case behaviour. We promised that our LBFS ordering would make it possible to amortize the cost of contraction. We finally prove this in the next section.

6 An amortized running time analysis

This section completes the proof for the running time of our algorithm, described completely in Section 5. Our main goal is to amortize the cost of contraction (Lemma 5.15), involving the number of updates and requests to the union-find data-structure, and the number of created label-edges and vertices involved in the adjacency lists of label graphs.

So far, the number of `find()` operations required by our algorithm has always been bounded by $O(T(|N(x)|))$. This will directly imply a suitable bound (by Lemma 6.18 in Subsection 6.3).

The `initialization()` routine is always performed just prior to a `union()` during a node-join operation and it involves a child of a degenerate node or the new vertex to be inserted (Algorithm 8 line 3). It follows that the number of `initialization()` operations is bounded by the number of non-root marker vertices belonging to a degenerate node that appear at some step of the algorithm (when x is inserted or when a node-split is performed). Bounding the number of such vertices is also required since they participate in the data-structure (see Subsection 6.1).

The `union()` operations are performed during a node-join (Algorithm 8 line 3) once together with each `initialization()` operation, and once to finalize the node-join. The total number for the first part is bounded the same way as `initialization()` operations, and the total number

for the second part is bounded by the total number of node-joins, which is bounded by the total number of created label-edges, since each node-join implies the creation of a label-edge (Lemma 5.14).

Therefore, bounding the number of created label-edges is the key to the complexity analysis and is the difficult part of our complexity argument. To count and bound the number of label-edges created during the whole algorithm, we use a charging argument in which the role of LBFS is critical (see Subsection 6.2.3).

6.1 Bounding the number of degenerate marker vertices

We prove that the number of non-root marker vertices that are created in some degenerate node during the process of building $ST(G)$ is linearly bounded by the number of vertices of the input graph G . The idea is to show that at each vertex insertion, only a constant number of such marker vertices are generated by our incremental algorithm.

Lemma 6.1. *Let G be a connected graph. The insertion of vertex x in the process of building $ST(G+x)$ creates at most two new non-root marker vertices belonging to a degenerate node in the rooted GLT data-structure.*

Proof. Consider forming the split-tree $ST(G+x)$, where x is some new vertex not already in G . We consider the changes required of $ST(G)$ to form $ST(G+x)$, as described by Theorem 4.14 and the subsequent propositions. Notice that the set of marker vertices belonging to some degenerate node is modified in three different ways:

- *the leaf x is attached to a degenerate node.* This occurs when cases 1, 2, 4, 5 and 6 of Theorem 4.14 apply. Two subcases are to be considered. If the degenerate node u neighbouring leaf x has degree 3 (that is case 4, 5 or 6 holds and u is a new node), then exactly two new non-root marker vertices have been created. It also follows that the degree in $G(u)$ of these two marker vertices is at most two. Otherwise (case 1 or 2), the only new non-root marker vertex $q \in V(u)$ is the opposite of x .
- *a node-split is performed on a degenerate node.* This occurs when cases 4 or 7 (during cleaning) of Theorem 4.14 applies. Observe that every split creates exactly two new marker vertices, one of which is the root of its node. In case 4, only one node-split is performed, thereby creating one extra non-root marker vertex in a degenerate node.

So let us consider the node-splits performed during the cleaning step when case 7 holds. Each degenerate node u of the fully-mixed subtree is involved in at most two node-splits (see Figure 9). Among the two nodes resulting from each node-split, one will eventually be node-joined to form a prime node and the other remains degenerate in $ST(G+x)$. Let us call v such a created degenerate node. All marker vertices inherited by v through the node-split are reused, and remain non-root marker vertices if they were non-root marker vertices in u . Hence the only case where a non-root marker vertex is created in v is when v inherits the root marker of u and thus a non-root marker vertex is created as the extremity of the new tree-edge resulting from the split. Of course, this case can happen at most once for any degenerate node u affected by a node-split. Moreover, this can only happen at the node at the root of the fully-mixed subtree. Thus at most one non-root degenerate marker is created during the series of node-split required by x 's insertion.

- a node-join is performed and involves a degenerate node. This only occurs during the contraction step while a prime node is being formed. In this case, marker vertices of a degenerate node are lost. The invariant trivially holds.

□

From the previous lemma, we can conclude the following:

Lemma 6.2. *The total cost of `initialization()` operations to the construction of $ST(G)$ is $O(n)$, where n is the number of vertices in G .*

Proof. Each `initialization()` operation takes constant time. They are only employed prior to a node-join involving a degenerate node (see Lemma 5.14), and when the vertex x to be inserted is made a neighbour of a newly formed prime node (see Lemma 5.15). Of the latter, there can be at most $O(n)$. The `initialization()` operations of the former are dealt with below. Every such `initialization()` operation corresponds to a child of a degenerate node, or equivalently to a non-root marker vertex of a degenerate node. Thus by Lemma 6.1, building $ST(G)$ requires $O(n)$ calls to `initialization()`. □

6.2 Bounding the number of label-edges

We shall first recall that, in our data-structure, degenerate nodes do not store any label-edge. label-edges belong to prime nodes, which are only formed by contraction. So, the label-edges in the resulting prime node either existed previously or were created by a node-join during contraction (Lemma 5.14). Recall also that label-edges adjacent to marker vertices that disappear during a node-join are lost since these are not reused, but of course they count in the total number of created label-edges.

To bound the number of created label-edges, we develop a charging argument driven by a stamping scheme. The stamps will help us to spread and distribute the charge over the successive steps of our algorithm. The charging argument depends on our LBFS ordering. Keep in mind that the split-tree construction algorithm does not involve the stamping scheme, nor the subsequent charging argument. These are only defined for the sake of counting created label-edges and of the amortized complexity analysis.

Let us sketch the construction. First, we need to show as a preliminary result that our LBFS ordering regulates how stars are formed during the construction of the split-tree: see Subsubsection 6.2.1. Second, every non-root marker vertex is associated with some *stamps* (0, 1 or 2 depending on its type), which are vertices of the input graph: see Subsubsection 6.2.2. Independently, marker vertices are associated with *Charge lists* such that spreading units of charge in the lists during the incremental process serves to count created label-edges: see Subsubsection 6.2.3. Lastly, the way stamps and Charge lists are associated with marker vertices will allow us to evaluate the total number of units of charge in terms of parameters of the input graph (number of edges and vertices), and hence to get the awaited complexity bound: see Lemma 6.16 in Subsection 6.3.

6.2.1 LBFS and stars

Let us introduce extra notation and definitions related to an LBFS ordering σ (see Section 3). First we will abusively use x_i instead of $\sigma^{-1}(i)$ to denote the i -th vertex in σ . Then G_i stands for

the subgraph induced by the subset $\{x_1, \dots, x_i\}$ of vertices and we denote by B_i the set of vertices appearing before x_i (not including x_i).

Definition 6.3. Let σ be an LBFS ordering of a connected graph G . A subset of consecutive vertices $S = \{x_i, \dots, x_j\}$ (with $i \leq j$) is a slice of σ if for every $y \in S$, $N(y) \cap B_i = N(x_i) \cap B_i$. The set $\text{Slice}(x_i)$ denotes the largest slice starting at vertex x_i .

In addition to the properties proved in Subsection 3.2, LBFS controls the formation of star nodes. During the split-tree construction process, some new marker vertices appear (e.g. the one opposite the new leaf, or when a node-split is performed), some disappear (when a node-join is performed) and some others are kept. More formally:

Definition 6.4. Let σ be an LBFS ordering of a connected graph G . Building on the definition given in Subsection 2.3, we say that $ST(G_{i+1})$ inherits a marker vertex q of a node in $ST(G_i)$ if q is not the extremity of a fully-mixed tree-edge of $ST(G_i)$ marked by the neighbourhood of x_{i+1} . By extension, $ST(G_j)$, with $j > i + 1$, inherits the marker vertex q from $ST(G_i)$ if $ST(G_{j-1})$ inherits q from $ST(G_i)$ and q is not the extremity of a fully-mixed tree-edge of $ST(G_{j-1})$ marked by the neighbourhood of x_j .

Recall that in a split-tree the centre of a star is never the opposite of a degree-1 marker vertex (since otherwise the split-tree wouldn't be reduced). For our charging argument, we need to extend this property over the life time of a marker vertex that was created as the centre of a star, assuming the vertex insertion follows an LBFS ordering.

Lemma 6.5. Let σ be an LBFS ordering of a connected graph G . Assume that to insert vertex x_i , a degree three node u_i labelled by a star has been created. Let c_i be the centre of u_i and q_i be the degree-1 marker vertex of u_i not opposite x_i . If $x_j \in \text{Slice}(x_i)$, then $ST(G_j)$ contains a star node u_j which contains c_i as centre and q_i as one of its degree-1 marker vertices. Moreover u_j contains a degree-1 marker vertex q_j such that $x_j \in L(q_j)$.

Proof. The result clearly holds if $i = j$. Consider the case $j = i + 1$. Notice that x_{i+1} is a twin of x_i since $x_{i+1} \in \text{Slice}(x_i)$. This implies that, if $ST(G_i)$ is marked by the neighbourhood of x_{i+1} , then c_i is perfect and q_i is empty. In other words the tree-edges respectively incident to c_i and q_i are not fully-mixed. So by definition, c_i and q_i are inherited by $ST(G_{i+1})$. Obviously, the opposite of x_i is either perfect or empty in $ST(G_i)$ and is inherited by $ST(G_{i+1})$. It follows that $ST(G_{i+1})$ contains the desired star node u_{i+1} .

Assume that for $i < k < j$, $ST(G_k)$ has a star node u_k in which c_i is the centre and q_i is the degree-1 marker vertex identified at the creation of u_i . Notice that by the definition of a star, $i > 2$. It follows that for every k such that $i < k < j$, $(B_i, \{x_i, \dots, x_k\})$ is a split of G_k . Moreover, observe that $B_i = L(c_i) \cup L(q_i)$. Consider the GLT obtained by a node-split of u_k , creating a tree-edge e corresponding to the split $(B_i, \{x_i, \dots, x_k\})$. Since $(B_i, \{x_i, \dots, x_k, x_{k+1}\})$ is also a split of G_{k+1} , the extremity q of e such that $L(q) = B_i$ is perfect or empty in this GLT marked with respect to the neighbourhood of x_{k+1} . So, by our incremental split-tree construction, the marker vertices in $T(q)$, and in particular c_i and q_i , are inherited by $ST(G_{k+1})$. Hence $ST(G_{k+1})$ contains the desired star node u_{k+1} . \square

Lemma 6.6. Let σ be an LBFS ordering of a connected graph G . Let c_i be the centre marker vertex of a star u_i in $ST(G_i)$. If c_i is inherited by $ST(G_j)$, with $i \leq j$, then it is not opposite a degree-1 marker vertex of a star in $ST(G_j)$.

Proof. Assume without loss of generality that c_i has been generated by x_i 's insertion, that is u_i is a degree three node u_i labelled by a star. Let q_i be the degree-1 marker vertex of u_i not opposite x_i in $ST(G_i)$. Let k be the smallest index such that c_i is inherited by $ST(G_k)$ and is opposite a degree-1 marker vertex. Clearly as $ST(G_i)$ is reduced, $i < k$. By assumption, in $ST(G_{k-1})$, c_i 's opposite has degree at least two. By Propositions 4.15, 4.16, 4.17 and 4.20, the only way to make c_i the opposite of a degree-1 marker vertex in $ST(G_k)$ is to subdivide the tree-edge e incident to c_i in $ST(G_{k-1})$ by a star node adjacent to x_k . That is, case 6 of Theorem 4.14 holds and e was the unique tree-edge with one perfect and one empty extremity (the perfect extremity being c_i). Observe that the node u containing c_i in $ST(G_{k-1})$ cannot be a star node (Corollary 4.4-5). We now contradict this fact. To that aim observe that as c_i is perfect in $ST(G_{k-1})$ and in $ST(G_i)$, we have $N(x_k) \cap B_i \subseteq N(x_i) \cap B_i$. As $i < k$, for σ to be a LBFS ordering, we have $N(x_k) \cap B_i = N(x_i) \cap B_i$; in other words, $x_k \in \text{Slice}(x_i)$. But now Lemma 6.5 implies that u is a star: contradiction. \square

The last results rely on the crucial assumption that an LBFS ordering is followed. One way of interpreting them is to say that once a star is created, it is then expanded maximally. The important consequence is what the last result says about the phase 1 node-joins defined by Algorithm 9. Recall that phase 1 node-joins involve a star whose root marker vertex is its centre, and one of its children. Lemma 6.6 therefore restricts the number of phase 1 node-joins a node can undergo. We need this fact to bound the number of new label-edges created during contraction.

6.2.2 Stamping schemes

The amortized complexity analysis relies on two stamping schemes. First, every non-root marker vertex q of a degenerate node is stamped with a vertex of the input graph G , called the *degenerate stamp* of q . As a consequence of Lemma 6.1, degenerate stamps can be assigned such that every vertex of G is used at most three times. Intuitively, the role of degenerate stamping is to amortize the cost of the creation of the label-edges of degenerate nodes prior to some node-join operation.

In addition, another stamping scheme is developed to amortize the cost of the creation of the label-edges generated by the node-join operations during contraction. Consider the following inductive procedure which, given a reduced GLT, assigns a stamp $s(q) = (s_1(q), s_2(q)) \in V(G)^2$ to every non-root marker vertex q that is not the centre of a star:

1. If q is opposite the leaf y , then $s(q) = (y, y)$.
2. Let uv be an internal tree-edge in the split-tree $ST(G)$ with extremities $q \in V(u)$ and $r \in V(v)$, where u is the parent of v :
 - (a) if $d(r) > 1$, then set $s(q) = (s_2(t), s_2(t'))$ for two (arbitrary) neighbours t and t' of r ;
 - (b) if $d(r) = 1$, and therefore v is a star with centre c , then set $s(q) = s(c)$, and then remove c 's stamp.

We will refer to $s_1(q)$ as q 's *primary stamp* and s_2 as q 's *secondary stamp*. We are interested in primary stamps; secondary stamps only exist to be "passed up" in step 2(a) above. The procedure guarantees the following properties of these stamps:

Lemma 6.7. *At the end of the procedure, centres of stars are the only non-root marker vertices without a stamp.*

Proof. This follows from the observation that only step 2(b) removes a stamp. \square

Lemma 6.8. *At the end of the procedure, if the leaf y is the primary stamp of the marker vertex q , then $y \in A(q)$.*

Proof. An easy inductive argument shows this, applying the fact that q only receives a stamp via its accessibility paths. \square

Lemma 6.9. *At the end of the procedure, every leaf is a primary stamp at most twice.*

Proof. Let uv be an arbitrary edge in T , where u is the parent of v , and let $q \in V(u)$ and $p \in V(v)$ be arbitrary non-root marker vertices, where p is accessible from q . We let y be an arbitrary vertex and examine how occurrences of y in $s(p)$ can be transmitted to $s(q)$. Note that the stamp (y, y) applies to the marker vertex opposite y , and thus a bottom-up argument starts with y having appeared once as a primary stamp.

First, we observe that no step of the algorithm allows a primary occurrence of y in $s(p)$ to be a secondary occurrence of y in $s(q)$. Suppose for contradiction that a primary occurrence of y in $s(p)$ is also a primary occurrence of y in $s(q)$. This can only happen by execution of step 2(b), but now the stamp is removed from p . Thus this case does not allow an increase in the number of times that x appears as a primary stamp.

Finally, suppose that a secondary occurrence of y in $s(p)$ becomes a primary occurrence of y in $s(q)$. Step 2(a) allows this to happen thereby increasing by one the number of times that y can appear as a primary stamp. The preceding argument shows that this cannot occur again. \square

What these properties will allow us to do, after the next subsection, is to transfer the charge assigned to marker vertices (that are not centres of stars) to their primary stamps. Lemma 6.8 allows us to associate charge with an edge (incident to the primary stamp) in the underlying accessibility graph. Lemma 6.9 allows us to associate the charge with a vertex (i.e. the primary stamp) in the underlying accessibility graph (and to do this at most twice for each vertex). Then we will be able to bound the total charge in terms of the input graph parameters.

6.2.3 The charging apparatus

The idea of the charging argument is to charge the creation of each new label-edge to one of its incident marker vertices. To that aim, each marker vertex q is associated throughout its lifetime with a list of vertices $Charge(q)$ that can be given units of charge.

Definition 6.10. *Let q be a marker vertex of a node u in a (rooted) split-tree $ST(G)$ of a connected graph G . The list $Charge(q)$ of vertices of G contains a set of vertices of G such that:*

- *each element in the list can be given a number of units of charge;*
- *the vertices are divided into groups, one for each of q 's neighbours in $G(u)$;*
- *the vertices in neighbour t 's group are the vertices in $A(t)$;*
- *the root marker vertex's group (if it exists) is called the root group.*

For a leaf x of $ST(G)$ (i.e. a vertex of G), the list $Charge(x)$ contains the vertices of $N(x) = A(x)$.

The way we assign charge during the algorithm is described precisely in forthcoming Lemmas 6.12, 6.13, 6.14 and 6.15. Of course, the Charge lists are not static during the construction of $ST(G)$: as new vertices are inserted, new elements must be added to some *Charge* lists; and as node-joins and node-splits occur, new groups are created and destroyed, respectively. However, throughout these changes, the following invariant will be maintained:

Invariant 6.11. *Let $ST(G)$ be the (rooted) split-tree of a connected graph G .*

- *if q is a marker vertex of a node u of $ST(G)$, then*
 1. *the list $Charge(q)$ is free of charge if*
 - (a) *u is a degenerate node or at some intermediate step of the contraction, q has degree one in $G(u)$;*
 - (b) *q is a root marker vertex and has never been the centre of a star at some prior step;*
 2. *if q is adjacent to the root of $G(u)$, then the root group in $Charge(q)$ is free of charge;*
 3. *at most one vertex in each group in $Charge(q)$ has been assigned charge;*
 4. *every vertex in $Charge(q)$ has been assigned at most one unit of charge if q is a root marker vertex, and at most three units of charge otherwise.*
- *if x is a leaf of $ST(G)$, then each vertex in $Charge(x)$ is assigned at most three units of charge.*

Moreover the number of label-edges created during the process of constructing $ST(G)$ is bounded by the total charge on all the $Charge()$ lists.

Let us observe that a marker vertex can have degree one (condition 1(a)) and not belong to a degenerate node only at some intermediate step of the contraction prior to the vertex insertion. Of course, once the new vertex is inserted, this is no longer possible since the current GLT is a split-tree (see Proposition 4.20).

Assume the invariants hold for the split-tree $ST(G)$. Now consider forming the split-tree $ST(G+x)$, where x is the last vertex in an LBFS ordering of $G+x$. We will consider the changes required of $ST(G)$ to form $ST(G+x)$, as described by Propositions 4.15, 4.16, 4.17 and 4.20.

Lemma 6.12. *Let x be the last vertex of an LBFS ordering of the connected graph $G+x$. If Invariant 6.11 is satisfied by $ST(G)$ and if $ST(G)$ does not contain a fully-mixed edge, then Invariant 6.11 is satisfied by $ST(G+x)$.*

Proof. In every case, except if $ST(G)$ contains a unique hybrid prime node (case 3 of Theorem 4.14), the modifications performed on $ST(G)$ to obtain $ST(G+x)$ only involve degenerate nodes. Thus no label-edge is created. By condition 1 of Invariant 6.11 every list $Charge(q)$ for a marker vertex of a degenerate node is free of charge, and Invariant 6.11 is still valid after x 's insertion.

In the case $ST(G)$ contains a unique hybrid prime node u , then by Proposition 4.15 new label-edges are created incident to x 's opposite, namely $q \in V(u)$ the new created marker vertex. Clearly q is not the root marker of u and $Charge(q)$ is divided in $|P(u)| = d(q)$ groups. One vertex of each of these groups receives a unit charge. If one of these group is the root group, then the charge assigned to one of its vertices is shifted to one of the other already charged vertices of $Charge(q)$. It follows that Invariant 6.11 it still satisfied. \square

We now deal with the case where $ST(G)$ contains a fully-mixed subtree T' . By the arguments used in the proof above, since the cleaning step only involves degenerate nodes and thus does not create any label-edges, the GLT $\text{cl}(ST(G))$ still satisfies Invariant 6.11. Our split-tree algorithm uses Algorithm 9 to perform contraction. Recall that it separates node-joins into three phases. Phase 1 node-joins involve star nodes whose root marker vertex is its centre. Phase 2 node-joins involve nodes whose root marker vertex has degree one. Phase 3 node-joins are all those remaining. No matter the phase, a node-join creates new label-edges. We need to assign charge to account for every one of these edges. However, this is done differently for each of contraction's three phases, as explained below.

Lemma 6.13 (Phase 1 node-joins). *Let x be the last vertex of an LBFS ordering of the connected graph $G + x$ and assume $\text{cl}(ST(G))$ satisfies Invariant 6.11. If the node-join(u, u') is performed on $\text{cl}(ST(G))$ between a star node u , whose root marker vertex is its centre, and a child u' of u , then the resulting GLT satisfies Invariant 6.11.*

Proof. Let c be the centre of u . Notice that $\text{Charge}(c)$ is free of charge, by condition 4 of Invariant 6.11. Let $t \in V(u)$ and $t' \in V(u')$ be the extremities of the tree-edge between u and u' . Notice that t is a degree one marker vertex.

Prior to the node-join, we need to create the label-edges of the graph $G(u)$ since it is degenerate and of $G(u')$ if u' is degenerate. Every such label-edge e is incident to a non-root marker vertex q whose degenerate stamp is a vertex z of G . Let y be a leaf of $A(q')$, where q' is the other marker vertex incident to e . Observe that y belongs to $\text{Charge}(z)$. Then one unit is charged to y 's entry in $\text{Charge}(z)$ for the cost of the creation of e . As z appears at most three times as a degenerate stamp (see Lemma 6.1), Invariant 6.11 is satisfied.

So assume the label-edges of $G(u)$ and $G(u')$ exist. If $d(t') > 1$, then the node-join of u and u' results in $d(t')$ extra label-edges being created. But notice that it also results in t 's group in $\text{Charge}(c)$ being replaced by $d(t')$ new groups, each free of charge. So to each of these new groups we assign one unit of charge. If t' is a degree one marker vertex, then recall the node-join is handled differently (see Algorithm 8). Only one new label-edge is added between c and t' 's unique neighbour. Again, for this we assign one unit of charge to what was t 's group in $\text{Charge}(c)$.

Now, since $d(t) = 1$, we know $\text{Charge}(t)$ is free of charge, by condition 1(a) of Invariant 6.11. We also know that t' is not the centre of a star, because $ST(G)$ is reduced. Moreover, t' can never have been the centre of a star, because of Lemma 6.6 and the fact that $d(t) = 1$. It follows from condition 1(b) of Invariant 6.11 that $\text{Charge}(t')$ is free of charge. In other words, no charge is lost in deleting $\text{Charge}(t)$ and $\text{Charge}(t')$ along with t and t' . It follows easily that the number of label-edges created so far is bounded by the total charge on all the $\text{Charge}()$ lists.

It is also easy to verify that every condition of Invariant 6.11 continues to hold, although we single out condition 1(b) for comment. The key for condition 1(b) is that c is no longer the centre of a star after the node-join is performed, and thus is allowed to have charge. \square

We can now assume that all phase 1 node-joins are complete. Let us turn to phase 2 node-joins. Recall from Algorithm 9 that a phase 2 node-join involves a node u and one of its children u' such that the root of u' is a degree one marker vertex. This type of node-join is implemented differently (see Algorithm 8). Observe also that u' could not have resulted from any node-join in phase 1.

Lemma 6.14 (Phase 2 node-joins). *Let x be the last vertex of an LBFS ordering of the connected graph $G + x$. Assume that all the phase 1 node-joins have been performed on $\text{cl}(ST(G))$ and that*

the resulting GLT satisfies Invariant 6.11. If the node-join(u, u') is performed between a node u and one of its children u' which is a star node rooted at a degree one marker vertex, then the resulting GLT satisfies Invariant 6.11.

Proof. First observe that as u' is a star node and u is possibly a clique node, the label-edges of $G(u)$ and $G(u')$ have to exist prior to the node-join. The cost of this creation can be charged, as described in the proof of Lemma 6.13, to the degenerate stamps of their non-root marker vertices.

Let r be the root of u' (r has degree one in $G(u')$) and let c be the centre of u' . Let q be the opposite of r . The node-join proceeds by deleting c 's neighbours (other than r) from u' , and adding them to u as neighbours of q . Suppose that k neighbours of c are moved in this way. Then k new label-edges are created by the node-join. But notice that adding the new edges incident to q creates k new groups in $Charge(q)$, each being free of charge by virtue of being new. So to each of these groups we assign one new unit of charge.

By condition 1(a) of Invariant 6.11, both $Charge(c)$ and $Charge(r)$ are free of charge. So no charge is lost deleting $Charge(c)$ and $Charge(r)$ along with c and r . Therefore the number of label-edges created so far is bounded by the total charge on all the $Charge()$ lists. \square

We now consider phase 3 node-joins. This means that all the node-join involving a star node have been performed.

Lemma 6.15 (Phase 3 node-joins). *Let x be the last vertex of an LBFS ordering of the connected graph $G+x$. Assume that all the phase 1 and phase 2 node-joins have been performed on $cl(ST(G))$ and that the resulting GLT satisfies Invariant 6.11. If the node-join(u, u') is performed, then the resulting GLT satisfies Invariant 6.11.*

Proof. As discussed in the proof of Lemma 6.13, if u or u' is a degenerate node (it must be a clique in this case), then the cost of creating the corresponding label-edges can be charged to the list of some degenerate stamps while preserving Invariant 6.11.

Let $q \in V(u)$ and $r \in V(u')$ be the extremities of the edge uu' . Since all phase 1 and phase 2 joins have been performed, we can assume that $d(q) > 1$ and $d(r) > 1$. Before assigning charge to account for the $d(q) \cdot d(r)$ new label-edges that are created, we will want to redistribute any charge on $Charge(q)$ and $Charge(r)$.

First consider the redistribution of the charge on $Charge(q)$. Let t be a neighbour of q in $G(u)$. If t is the root marker vertex, then t 's group in $Charge(q)$ is free of charge by condition 2 of Invariant 6.11, and so no charge in this group needs to be redistributed. So let t be a non-root marker vertex that is a neighbour of q . Then by condition 3 of Invariant 6.11, we know that at most one vertex in t 's group in $Charge(q)$ has been assigned charge. If such a vertex exists, then call it λ . Notice that during the u, u' node-join, $Charge(t)$ will lose q 's group but will gain the $d(r) > 1$ groups in $Charge(r)$. By condition 3 of Invariant 6.11, at least one of these groups (say γ) will be free of charge. The charge on λ is reassigned to one of the vertices in γ in this case. Continuing this for all such t removes all charge on $Charge(q)$, and so it can be deleted along with q and no charge is lost.

We now turn to the redistribution of the charge on $Charge(r)$. Let t' be a neighbour of r in $G(u')$, and notice that by condition 3 of Invariant 6.11, at most one vertex in t' 's group in $Charge(r)$ has been assigned charge. If such a vertex exists, call it λ' . By condition 4 of Invariant 6.11, we know that λ' has been assigned no more than one unit of charge. Now, once more by condition 2 of Invariant 6.11, we know that r 's group in $Charge(t')$ is free of charge. Furthermore, during the

join, $Charge(t')$ will lose r 's group but will gain the $d(q) > 1$ groups in $Charge(q)$. Let γ' be one of these new groups. In this case we reassign λ' 's one unit of charge (if it exists) to a label in γ' . Continuing this for all such t' removes all charge on $Charge(r)$, and so it can be deleted along with r and no charge is lost.

We finally assign $d(q) \cdot d(r)$ new units of charge. Let t' and γ' be as above. So r 's former group in $Charge(t')$ is replaced by $d(q) > 1$ new groups, one of them called γ' . Only γ' (possibly) has any charge assigned to it, and only one unit at that; the other $d(q) - 1$ groups are free of charge. To $d(q) - 2$ of the groups free of charge we assign one unit of charge, and to the remaining group free of charge we assign two units of charge. Lastly, if one of the groups corresponds to the root marker vertex of u , then the charge just assigned to that group is shifted to another, which must exist. The result is that only one vertex in each group contains charge, none having been assigned more than three units, and the root group becomes free of charge.

It is easy to verify that Invariant 6.11 continues to hold under this (re)assignment of charge. \square

6.3 Bounding the total charge, and the running time of our algorithm

Lemmas 6.12, 6.13, 6.14 and 6.15 guarantee that Invariant 6.11 holds during the LBFS incremental construction of the split-tree of G . Consequently, the total number of label-edges created all along the construction is bounded by the total charge residing on all $Charge$ lists.

Lemma 6.16. *Let G be a connected graph. The total number of label-edges created during our LBFS incremental construction of $ST(G)$ is $O(n + m)$, where n is the number of vertices in G and m is the number of its edges.*

Proof. Assign primary stamps to the marker vertices in $ST(G)$ as described earlier. By condition 1(a) of Invariant 6.11, we can focus on the $Charge$ lists residing on marker vertices in prime nodes and on leaves of $ST(G)$.

By Invariant 6.11, the elements of the lists $Charge(x)$ received at most 3 units of charge. As these lists contain exactly $\sum_{x \in V(G)} d_G(x) = \sum_{x \in V(G)} |A(x)|$ elements, the total charge on these lists is bounded by $O(m)$.

Let u be a prime node, and let $r \in V(u)$ be u 's root marker vertex, and let q be one of u 's non-root marker vertices. If q and r are adjacent, then download all the charge from q 's group in $Charge(r)$ to r 's group in $Charge(q)$. By condition 4 of Invariant 6.11, the total charge in r 's group in $Charge(q)$ is not more than four units. By the same condition 4, the total charge in the other groups in $Charge(q)$ is no more than three units.

Now, by choice of q and Lemma 6.7, we can assume that q has been assigned a primary stamp. Moreover, the vertex $s_1(q)$ acting as primary stamp is in $A(q)$, by Lemma 6.8. So by definition of $Charge(q)$, we know $s_1(q)$ is adjacent to every label in $Charge(q)$. The total charge on all such $Charge$ lists is therefore $O(n + m)$, by Lemma 6.9 and our discussion above. \square

More important than the bound above is what it implies:

Lemma 6.17. *Let G be a connected graph. The total number of node-joins and `union()` operations performed by our LBFS incremental construction of $ST(G)$ is $O(n + m)$, where n is the number of vertices in G and m is the number of its edges.*

Proof. Notice that during every node-join at least one new label-edge is created (Lemma 5.14). The bound on the number of node-joins now follows from Lemma 6.16: it is $O(n + m)$.

There are exactly three ways our algorithm applies a `union()` operation: once after each `initalization()` in a node-join, once for finalizing every node-join, and once when a new prime node is formed. The number of such applications of the first way is $O(n)$ by Lemma 6.2, the number of the second way is $O(n + m)$ as said above, and the number of the third way is $O(n)$, since at most one new prime node is formed for each vertex inserted. \square

Lemma 6.18. *Let G_n be a connected graph with n vertices and m edges whose split-tree is incrementally constructed by repeated application of Algorithm 3; that is $ST(G_{i+1}) = ST(G_i + x_{i+1})$ is built from $ST(G_i) = (T_i, \mathcal{F}_i)$ for $1 \leq i \leq n - 1$. Then the sum of $|T_i(N_{G_{i+1}}(x_{i+1}))|$ over all $1 \leq i \leq n - 1$ is $O(n + m)$.*

Proof. For a fixed i , let G denote G_i , T denote T_i , x denote x_{i+1} , and $N(x)$ denote $N_{G_{i+1}}(x)$. We can divide the nodes of $T(N(x))$ into two groups: those that remain in $ST(G + x)$, and those that do not. The number of those that remain is $O(|N(x)|)$, by Lemma 2.20. So the total number of nodes in the first group over the entire execution of our algorithm is $O(n + m)$. Every node in the second group participates in at least one node-join. So the total number of nodes in the second group over the entire execution of our algorithm is also $O(n + m)$, by Lemma 6.17. \square

Lemma 6.19. *Let G be a connected graph. The total number of `find()` operations performed by our LBFS incremental construction of $ST(G)$ is $O(n + m)$, where n is the number of vertices in $G + x$ and m is the number of its edges.*

Proof. Our algorithm uses `find()` operations to traverse the split-tree. These traversals can take place during case identification and state assignment, cleaning, and contraction. Case identification and state assignment take place according to Algorithm 5; cleaning takes place according to Algorithm 7; and contraction takes place according to Algorithm 9. So by Lemmas 5.10, 5.13, and 5.15, the total number of `find` operations required for the insertion of x is $O(|T(N(x))|)$. So the total number for the whole algorithm is $O(n + m)$ by Lemma 6.18. \square

Lemma 6.20. *Let G be a connected graph. The total number of nodes (including fake nodes) created in the rooted GLT data-structure used by our algorithm, and the total number of elements in the union of children-sets, is $O(n + m)$.*

Proof. A node is created either when the new vertex x is inserted, or when a node-split is performed. At most one node-split is performed in the case where there is no fully-mixed subtree. Therefore, the total number of nodes created in this case and by the insertion of x is $O(n)$ over the course of the LBFS construction of $ST(G)$. If there is a fully-mixed subtree, then node-splits are performed during the cleaning step and they involve nodes in $T(N(x))$. At most two node-splits are performed at each such node. So the total number of created nodes for the whole algorithm is $O(n + m)$ by Lemma 6.18. Elements of children-sets in the data-structure are either leaves or some nodes created at some step of the algorithm, hence their total number is also $O(n + m)$. \square

The previous lemmas culminate in the theorem below, which is the main result of our paper:

Theorem 6.21. *The split-tree $ST(G)$ of a graph $G = (V, E)$ with n vertices and m edges can be built incrementally according to an LBFS ordering in time $O(n + m)\alpha(n + m)$, where α is the inverse Ackermann function.*

Proof. Lemma 6.1 establishes an $O(n)$ bound on the number of non-root degenerate marker vertices. As every node has degree at least 3, the total number of degenerate marker vertices created during the LBFS incremental construction is $O(n)$. The total number of label-edges created during the LBFS incremental construction is $O(n + m)$, by Lemma 6.16. Therefore our algorithm generates an $O(n + m)$ size data-structure.

In addition to the cost of computing an LBFS, which takes time $O(n + m)$ [26, 27], we have to bound the cost of the tree traversals, based on `find()` operations, plus the total cost of the `initialization()`, `union()` operations involved in the contraction steps. The sum, over all the algorithm, of the term $|T(N(x))|$ in Lemmas 5.10, 5.13, and 5.15, is $O(n + m)$, by Lemma 6.18. The total cost of `initialization()` operations required by the algorithm is $O(n)$, by Lemma 6.2. The total number of *union* and *find* operations is $O(n + m)$, by Lemmas 6.17 and 6.19, respectively.

Finally, the cost of the union-find requests amounts to $O(\alpha(N)(n + m) + N)$, where N is the total number of elements in the union of children-sets. This number N is $O(n + m)$ by Lemma 6.20. It is easy to prove that $\alpha(C \cdot (n + m)) \sim \alpha(n + m)$ for any fixed constant C . Indeed, because of the way the Ackermann function increases, for any n , $\alpha(n) = k - 1$ implies $\alpha(C.n) \leq k$ for every k large enough. Hence $\alpha(C.n) - \alpha(n) \leq 1$ for n large enough, which implies $\alpha(C.n) \sim \alpha(n)$. In particular, we have that $\alpha(O(n + m)) = O(\alpha(n + m))$.

So to conclude, the total cost of our algorithm is $O((n + m)\alpha(n + m))$, which can also be written $O(n + m)\alpha(n + m)$. \square

To conclude, let us mention that we are prevented from achieving linear time only by the node-join. Let u and u' be two adjacent nodes, with u the parent of u' . To effect their node-join, the children of u' must be made children of u . That is the bottleneck. Our implementation does its best to avoid it by using union-find, but the optimal complexity for union-find involves the inverse Ackermann function. It seems to us that our charging argument can not be extended to cover the cost of reassigning u' 's children, thereby eliminating union-find and achieving linear time. However, it is worth emphasizing that, from the practical viewpoint, the inverse Ackermann function can be thought of as a constant, and that every other aspect of our algorithm is consistent with linear time.

This paper's companion [25] extends our split decomposition algorithm to recognize circle graphs in same time. It is the first sub-quadratic circle recognition algorithm, and the first progress on the problem in fifteen years.

References

- [1] A. Bouchet. Reducing prime graphs and recognizing circle graphs. *Combinatorica*, 7:243–254, 1987.
- [2] A. Bouchet. Circle graph obstructions. *Journal of Combinatorial Theory Series B*, 60:107–144, 1994.
- [3] M. Buriel and J.P. Uhry. Parity graphs. *Annals of Discrete Mathematics*, 21:253–277, 1984.
- [4] P. Charbit, F. de Montgolfier, and M. Raffinot. A simple linear time split decomposition algorithm of undirected graphs. *CoRR*, abs/0902.1700, 2009. <http://arxiv.org/abs/0902.1700>.
- [5] M. Chudnovsky, G. Cornuejols, X. Liu, P. Seymour, and K. Vuskovic. Recognizing berge graphs. *Combinatorica*, 25:143–186, 2005.

- [6] M. Chudnovsky, N. Robertson, P. Seymour, and R. Thomas. The strong perfect graph theorem. *Annals of Mathematics*, 164(1):51–229, 1993.
- [7] S. Cicerone and G. Di Stefano. Graph classes between parity and distance-hereditary graphs. *Discrete Applied Mathematics*, 95:197–216, 1999.
- [8] S. Cicerone and G. Di Stefano. On the extension of bipartite to parity graphs. *Discrete Applied Mathematics*, 95(1-3):181–195, 1999.
- [9] T. H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
- [10] D.G. Corneil. Lexicographic breadth first search - a survey. In *International Workshop on Graph Theoretical Concepts in Computer Science (WG)*, *Lecture Notes in Computer Science*, 3353:1–19, 2004.
- [11] D. Corneil, M. Habib, J.-M. Lanlignel, B. Reed, and U. Rotics. Polynomial time recognition of clique-width ≤ 3 graphs. *Discrete Applied Mathematics*, 160(6):834–865, 2012.
- [12] B. Courcelle. The monadic second-order logic of graphs XVI: canonical graph decomposition. *Logical Methods in Computer Science*, 2(2):1–46, 2006.
- [13] B. Courcelle, J. Engelfriet, and G. Rozenberg. Handle rewriting graph grammars. *Journal of Computer and System Science*, 46:218–270, 1993.
- [14] W.H. Cunningham. Decomposition of directed graphs. *SIAM Journal on Algebraic Discrete Methods*, 3:214–228, 1982.
- [15] W.H. Cunningham and J. Edmonds. A combinatorial decomposition theory. *Canadian Journal of Mathematics*, 32(3):734–765, 1980.
- [16] E. Dahlhaus. Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition. *Journal of Algorithms*, 36(2):205–240, 2000.
- [17] F. Dragan, F. Nicolai, and A. Brandstädt. LexBFS-orderings and powers of graphs. In *International Workshop on Graph Theoretical Concepts in Computer Science (WG)*, *Lecture Notes in Computer Science*, 1197:166–180, 1996.
- [18] J. Engelfriet and V. van Oostrom. Logical description of context-free graph languages. *Journal of Computer and System Science*, 55:489–503, 1997.
- [19] D. Eppstein, M.T. Goodrich, and J. Yu Meng. Delta-confluent drawings. In *International Symposium on Graph Drawing (GD)*, *Lecture Notes in Computer Science*, 3843:165–176, 2005.
- [20] C.P. Gabor, W.L. Hsu, and K.J. Suppovit. Recognizing circle graphs in polynomial time. *Journal of ACM*, 36:435–473, 1989.
- [21] T. Gallai. Transitiv orientierbare graphen. *Acta Math. Acad. Sci. Hungar.*, 18:25–66, 1967.
- [22] C. Gavaille and C. Paul. Distance labelling scheme and split decomposition. *Discrete Mathematics*, 273(1-3):115–130, 2003.

- [23] E. Gioan and C. Paul. Dynamic distance hereditary graphs using split decomposition. In *International Symposium on Algorithms and Computation (ISAAC), Lecture Notes in Computer Science*, 4835:41–51, 2007.
- [24] E. Gioan and C. Paul. Split decomposition and graph-labelled trees: characterizations and fully-dynamic algorithms for totally decomposable graphs. *Discrete Applied Mathematics*, 160(6):708–733, 2012.
- [25] E. Gioan, C. Paul, M. Tedder, and D. Corneil. Practical and efficient circle graph recognition. arXiv:1104.3284, 2011.
- [26] M.C. Golumbic. *Algorithmic graph theory and perfect graphs, 2nd Edition*. Elsevier, 2004.
- [27] M. Habib, R.M. McConnell, C. Paul, and L. Viennot. Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.
- [28] M. Habib and C. Paul. A survey of the algorithmic aspects of modular decomposition. *Computer Science Review*, 4(1):41–59, 2010.
- [29] P. Hammer and F. Maffray. Completely separable graphs. *Discrete Applied Mathematics*, 27:85–99, 1990.
- [30] E. Howorka. A characterization of distance hereditary graphs. *Quart. Journal Math. Oxford Series*, 2(112):417–420, 1977.
- [31] W.-L. Hsu. $O(n.m)$ algorithms for the recognition and isomorphism problems on circular-arc graphs. *SIAM Journal on Computing*, 24(3):411–439, 1995.
- [32] T.-H. Ma and J. Spinrad. An $O(n^2)$ algorithm for undirected split decomposition. *Journal of Algorithms*, 16:145–160, 1994.
- [33] R.H. Möhring and F.J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics*, 19:257–356, 1984.
- [34] S.-I. Oum. Rank-width and vertex minors. *Journal of Combinatorial Theory, Series B*, 95(1):79–100, 2005.
- [35] D.J. Rose, R.E. Tarjan, and G.S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.
- [36] J. Spinrad. Recognition of circle graphs. *Journal of Algorithms*, 16:264–282, 1994.
- [37] J. Spinrad. *Efficient Graph Representation*, volume 19 of *Fields Institute Monographs*. American Mathematical Society, 2003.
- [38] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):146–160, 1975.

- [39] M. Tedder. *Applications of lexicographic breadth-first search to modular decomposition, split decomposition and circle graph recognition*. PhD thesis, Department of Computer Science, University of Toronto, 2011. <http://www.cs.toronto.edu/~mtedder/>.
- [40] N. Trotignon and K. Vuskovic. A structure theorem for graphs with no cycle with a unique chord and its consequences. *Journal of Graph Theory*, 63(1):31–67, 2010.
- [41] W. Tutte. *Connectivity in Graphs*. Toronto University Press, 1966.