A Vertex Incremental Approach for Maintaining Chordality

Anne Berry[*] Pinar Heggernes[†] Yngve Villanger[†]

Abstract

For a chordal graph $G = (V, E)$, we study the problem of whether a new vertex $u \notin V$ and a given set of edges between u and vertices in V can be added to G so that the resulting graph remains chordal. We show how to resolve this efficiently, and at the same time, if the answer is no, specify a maximal subset of the proposed edges that can be added along with u, or conversely, a minimal set of extra edges that can be added in addition to the given set, so that the resulting graph is chordal. In order to do this, we give a new characterization of chordal graphs and, for each potential new edge uv, a characterization of the set of edges incident to u that also must be added to G along with uv. We propose a data structure that can compute and add each such set in $O(n)$ time. Based on these results, we present an algorithm that computes both a minimal triangulation and a maximal chordal subgraph of an arbitrary input graph in $O(nm)$ time, using a totally new vertex incremental approach. In contrast to previous algorithms, our process is on-line in that each new vertex is added without reconsidering any choice made at previous steps, and without requiring any knowledge of the vertices that might be added subsequently.

1 Introduction

Chordal graphs (also called triangulated graphs) are a well-studied class of graphs, with applications in many fields. Some applications require that chordality be maintained incrementally, that is, as edges and/or vertices are added or deleted from the graph, they desire to maintain chordality. Ibarra [28] gives a dynamic algorithm for adding or removing a given edge in $O(n)$ time in a chordal graph if this does not destroy chordality, where n is the number of vertices of the input graph. More recently, a *2-pair* [10] has been defined as a pair of non-adjacent vertices in a chordal graph, such that the graph remains chordal when the edge between these vertices is added to the graph.

A chordal graph can be obtained from any non chordal graph by: adding edges until the graph becomes chordal, a process called *triangulation*, or by removing edges until the graph becomes chordal, thus computing a *chordal subgraph*. Adding or removing a minimum number of edges has been shown to be NP-hard [30, 36]. However, adding or

[*]LIMOS, UMR CNRS 6158, Universite Clermont-Ferrand II, F-63177 Aubiere, France. Email: `berry@isima.fr`

[†]Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Emails: `Pinar.Heggernes@ii.uib.no` and `yngvev@ii.uib.no`

removing an inclusion minimal set of edges can be accomplished in polynomial time. Given an arbitrary chordal subgraph, e.g., an independent set on the vertices of the graph (resp. supergraph, e.g., a complete graph on the same vertex set) of the input graph, edges can be added (resp. removed) one by one after testing that the resulting graph remains chordal, until no further candidate edge can be found. This ensures that maximality (resp. minimality) is achieved, by the results of [32]. The problem of maintaining a chordal graph by edge addition or deletion and the problem of computing a maximal chordal subgraph or a minimal chordal supergraph are thus strongly related.

The problem of adding an inclusion minimal set of *fill* edges, called *minimal triangulation*, has many applications in various fields such as sparse matrix computation [31] and database management [3]. The problem has been well studied since 1976, and several $O(nm)$ time algorithms exist for solving it [4, 5, 8, 16, 32], where m is the number of edges in the input graph. None of these algorithms use an edge incremental approach as described above. However, the algorithm proposed by Blair, Heggernes, and Telle [11], which requires even less time when the fill is small, does use an edge deletion approach.

The reverse problem of computing a maximal chordal subgraph has also been studied, with applications to sparse matrix computation, computing a large clique or a large independent set, and improving phylogenetic data [2, 10, 15, 17, 20, 35]. There exist several algorithms that compute a maximal chordal subgraph in $O(\Delta m)$ time, where Δ is the maximum degree in the graph [2, 17, 35].

In this paper, we present a new process for adding a *vertex* with a given set of incident edges to a chordal graph while maintaining chordality, which we are able to implement more efficiently than if we were to add the corresponding edges one by one. Our process is based on two new characterizations. The first is a characterization of a chordal graph by its edges, which can be regarded as a specialization of the edge characterization for weakly chordal graphs introduced by Berry, Bordat, and Heggernes [7]. The second is a characterization of a unique set of edges $R(G, u, v)$ incident to a vertex u that must be added to a chordal graph G along with edge uv to ensure that chordality is preserved, given that we are only allowed to add edges incident to u. We show that we can compute this set $R(G, u, v)$ of edges in $O(n)$ time, by proposing a data structure that corresponds to a clique tree of the current chordal subgraph. A similar data structure was used by the authors to prove an $O(nm)$ time bound for one of their minimal triangulation algorithms [4, 8, 26]; however, here we present a new implementation of clique trees that allows a more efficient data structure for our purposes.

We use our results to compute both a minimal triangulation and a maximal chordal subgraph of a given arbitrary graph in $O(nm)$ time. This is done by an incremental process that repeatedly adds a new vertex u to the already constructed chordal graph H along with a maximal set of edges between u and H, or a minimal set of extra edges between u and H in addition to the originally specified edges.

Some of the existing algorithms that compute a maximal chordal subgraph or a minimal triangulation also use a vertex incremental process [2, 5, 8, 17, 32, 35], though none of them compute both chordal graphs at the same time. In addition, all these previous algorithms require knowing the whole graph in advance, as either vertices that are not yet processed are marked in some way to define the next vertex in the process, or edges are added between pairs of vertices that are not yet processed. Furthermore, these algorithms require the added vertex to be a simplicial vertex of the transitory chordal graph. One exception from this requirement is the algorithm of [8], but it does add edges between pairs of neighbors of the added vertex that are not yet processed.

Our approach here is completely different from the previous ones, as it is more general: at each vertex addition step, we do not require the added vertex to be or to become simplicial, thereby enabling processing of vertices in any order. Moreover, we add only edges incident to the new vertex, so that we never need to reconsider or change the chordal graph that has been computed thus far.

As a result, our process can add any vertex with any proposed neighborhood, and efficiently give a correction if the resulting graph fails to be chordal, either by computing a maximal subset of the edges to be added, or a minimal set of extra edges along with the proposed ones. In addition, the transitory chordal graph is maintained in a dynamic fashion, as making the desired or necessary additions to the graph does not require a recomputation.

This paper is organized as follows: in the next section we give the necessary graph theoretic background and terminology. Section 3 contains our new characterizations. The algorithms are presented and proved correct in Section 4, whereas the data structure details and time complexity analysis are given in Section 5. We conclude in Section 6.

2 Graph theoretic background and notation

A graph is denoted $G = (V, E)$, with $n = |V|$, and $m = |E|$. A vertex sequence $v_1 - v_2 - ... - v_k$ describes a *path* if $v_i v_{i+1}$ is an edge for $1 \leq i < k$. The *length* of a path is the number of edges in it. A *cycle* is a path that starts and ends with the same vertex, and the length of the cycle is the number of vertices or edges it contains. A *chord* of a cycle (path) is an edge connecting two non-consecutive vertices of the cycle (path). A *clique* is a set of vertices that are pairwise adjacent.

For the following definitions, we will omit subscript G when the graph is clear from the context. The *neighborhood* of a vertex v in G is $N_G(v) = \{u \neq v \mid uv \in E\}$, and for a set of vertices A, $N_G(A) = \cup_{x \in A} N_G(x) \setminus A$. A *simplicial* vertex is one whose neighborhood induces a clique. $G(A)$ is the subgraph induced by a vertex set $A \subseteq V$, but we often denote it simply by A when there is no ambiguity. We would like to stress that we distinguish between subgraphs and induced subgraphs.

For any vertex set $S \subseteq V$ and any vertex $x \in V \setminus S$, C_S^x denotes the connected component of $G(V \setminus S)$ containing x. A subset S of V is called a *separator* if $G(V \setminus S)$ is disconnected. S is a *u, v-separator* if vertices u and v are in different connected components of $G(V \setminus S)$, and a *minimal u, v-separator* if no subset of S is a u, v-separator. S is a *minimal separator* of G if there is some pair $\{u, v\}$ of vertices in G such that S is a minimal u, v-separator. Equivalently, S is a minimal separator if there exist two connected components C_1 and C_2 of $G(V \setminus S)$ such that $N_G(C_1) = N_G(C_2) = S$.

A pair of non-adjacent vertices $\{u, v\}$ is a *2-pair* in G if there is no chordless path of length 3 or more between u and v [24]. If G is not connected, then two vertices that belong to different connected components constitute a 2-pair by definition. If G is connected, it has been shown that $\{u, v\}$ is a 2-pair if and only if $N(u) \cap N(v)$ is a minimal u, v-separator of G [1, 33].

A graph is *chordal* if it contains no chordless cycle of length ≥ 4. Consequently, all induced subgraphs of a chordal graph are also chordal. G is chordal if and only if every minimal separator of G is a clique [19]. Chordal graphs are the intersection graphs of subtrees of a tree [14, 22, 34], and the following result gives a very useful tool which we will use as a data structure in our algorithm.

Theorem 2.1 (Buneman [14], Gavril [22], Walter[34]) *A graph G is chordal if and only if there exists a tree T, whose vertex set is the set of maximal cliques of G, that satisfies the following property: for every vertex v in G, the set of maximal cliques containing v induces a connected subtree of T.*

Such a tree is called a *clique tree* [12], and we will refer to the vertices of T as *tree nodes* to distinguish them from the vertices of G, and sometimes also as *bags* since these contain several graph vertices. Each tree node of T is thus a vertex set of G corresponding to a maximal clique of G. We will not distinguish between maximal cliques of G and their corresponding tree nodes. In addition, it is customary to let each edge K_iK_j of T hold the vertices of $K_i \cap K_j$, where K_i and K_j are maximal cliques of G. Thus, edges of T are also vertex sets. Although a chordal graph can have many different clique trees, all chordal graphs share the following important properties that are related to an efficient implementation of our algorithm.

Theorem 2.2 (Buneman [14], Ho and Lee[27], Lundquist [29]) *Let T be a clique tree of a chordal graph G. A set S is a minimal separator of G if and only if $S = K_i \cap K_j$ for an edge K_iK_j in T, and if $S = K_i \cap K_j$ for an edge K_iK_j in T, then S is a minimal u, v-separator for any $u \in K_i \setminus S$ and $v \in K_j \setminus S$.*

Theorem 2.3 (Blair and Peyton [12]) *T is a clique tree of G if and only if T is a tree whose nodes are the maximal cliques of G, and for every pair of distinct maximal cliques K_i and K_j in G the intersection $K_i \cap K_j$ is contained in every node of T (maximal clique of G) appearing on the path between K_i and K_j in T.*

Note that as a consequence, the intersection $K_i \cap K_j$ is also contained in every edge of T (i.e., in every minimal separator of G) appearing on the path between K_i and K_j in T. A chordal graph has at most n maximal cliques [19] and hence the number of nodes and edges in a clique tree is $O(n)$ [21].

From any given non-chordal graph, one can obtain a chordal graph on the same vertex set by either adding (the added edges are called *fill* edges) or removing edges. $M = (V, F)$ is called a *triangulation* of an arbitrary graph $G = (V, E)$ if $E \subseteq F$ and M is chordal. M is a *minimal triangulation* of G if no proper subgraph of M is a triangulation of G. Similarly, $H = (V, D)$ is called a *chordal subgraph*, or equivalently a *subtriangulation*, of G if $D \subseteq E$ and H is chordal. H is a *maximal chordal subgraph*, or a *maximal subtriangulation*, if (V, D') is non-chordal for every set D' that satisfies

$D \subset D' \subseteq E$. By the results of [32], a given triangulation (subtriangulation) is minimal (maximal) if and only if no single fill edge can be removed (no single removed edge can be added back) without destroying chordality.

3 A new characterization of chordal graphs

In this section we present a new characterization of chordal graphs that will be the basis of our algorithm.

Definition 3.1 *An edge uv is* mono-saturating *in $G = (V, E)$ if $\{u, v\}$ is a 2-pair in $G' = (V,\ E \setminus \{uv\})$.*

Theorem 3.2 *A graph is chordal if and only if every edge is mono-saturating.*

Proof. Let $G = (V, E)$ be chordal, and assume on the contrary that there is an edge $uv \in E$ that is not mono-saturating. Then there is a chordless path P of length more than 2 between u and v in $G' = (V,\ E \setminus \{uv\})$, and thus the following is a chordless cycle of length at least 4 in G: $u - P - v - u$, which contradicts our assumption that G is chordal.

For the other direction, let every edge in G be mono-saturating, and assume on the contrary that G is not chordal. Thus, there exists a chordless cycle C of length at least 4 in G. No edge of C is mono-saturating, a contradiction. ∎

As a corollary of Theorem 3.2, we can deduce the following characterization by 2-pairs by [10], a result that was also observed with a different formulation in [18].

Corollary 3.3 (Berry et al. [10]) *Given a chordal graph $G = (V, E)$, where $uv \notin E$, the graph $H = (V,\ E \cup \{uv\})$ is chordal if and only if $\{u, v\}$ is a 2-pair in G.*

Proof. Let us on the contrary assume that H is not chordal, and that $\{u, v\}$ is a 2-pair in G. The edge uv is mono-saturating in H since $\{u, v\}$ is a 2-pair of G. By Theorem 3.2 there exists an edge xy in G that is not mono-saturating in H, and by Definition 3.1 there exists a chordless path P in $H' = (V,\ (E \setminus \{xy\}) \cup \{uv\})$ preventing xy from being mono-saturated in H. One of the edges in P is uv, since G is chordal and by Theorem 3.2 xy is mono-saturating in G, and by Definition 3.1 P do not exists in $G' = (V,\ E \setminus \{xy\})$. By removing the edge uv from P and inserting xy we obtain a chordless path P' in G, which prevents $\{u, v\}$ from being a 2-pair in G, and thus we have a contradiction.

For the other direction, we know that $\{u, v\}$ is not a 2-pair in G, and thus the edge uv in H is not mono-saturating, and by Theorem 3.2 H is not chordal. ∎

As a consequence, while maintaining a chordal graph by adding edges, we could check every edge of the input graph to see if the endpoints constitute a 2-pair in the transitory chordal subgraph. However, this approach requires that we check every edge several times, as pairs of vertices can become 2-pairs only after the addition of some other edges. Our main result, to be presented as Theorem 3.8, gives a more powerful tool that allows examining each edge of the input graph only *once* during such a process.

Assume the following scenario: given a chordal graph G, we want to add an edge uv to G. Since we want the resulting graph to remain chordal, it may be necessary to add other edges to achieve this. However, we allow addition of edges only incident to u.[1] Naturally, if we add every edge between u and the other vertices of G, the resulting graph is chordal. Our main goal is to add as few edges as possible.

Definition 3.4 *Given a chordal graph $G = (V, E)$ and any pair of non-adjacent vertices u and v in G, $R(G, u, v) = \{ux \mid x$ belongs to a minimal u, v-separator of $G\}$. We will call $R(G, u, v)$* the incident-to-u set of required edges for uv.

Lemma 3.5 *Let $G = (V, E)$ be a chordal graph and let u and v be non-adjacent vertices of G. Then $R(G, u, v) = \{ux \mid x$ is an intermediate vertex of a chordless path in G between u and $v\}$.*

Proof. Let $ux \in R(G, u, v)$, S be a minimal u, v-separator of G containing x, P_1 be a chordless path in G between u and x with all intermediate vertices belonging to C^u_S, and P_2 be a chordless path in G between x and v with all intermediate vertices belonging to C^v_S. The path obtained by concatenating P_1 and P_2 is a chordless path in G between u and v having x as an intermediate vertex.

Conversely, let x be an intermediate vertex of a chordless path P in G between u and v. Vertex set S' obtained from V by removing all vertices of P except x is a u, v-separator of G. Let S be a minimal u, v-separator of G included in S'. Vertex x belongs to S because otherwise P would be a path in $G(V \setminus S)$ between u and v. Therefore $ux \in R(G, u, v)$. ∎

Lemma 3.6 *Let $G = (V, E)$ be a chordal graph, let u and v be non-adjacent vertices of G, let S be a minimal u, v-separator of G, and let graph $M = (V,\ E \cup \{uv\} \cup R(G, u, v))$. Then any chordless cycle in M of length at least 4 containing u contains at most one vertex of S.*

Proof. Suppose on the contrary that some chordless cycle C in M of length at least 4 contains u and distinct vertices x and x' of S. As a minimal separator of a chordal graph, S is a clique in G, so xx' is an edge of M, and by definition of $R(G, u, v)$, ux and ux' are also edges of M. So C has a chord in M, a contradiction. ∎

Lemma 3.7 *Let $G = (V, E)$ be a chordal graph, let u and v be non-adjacent vertices of G, and let S and S' be minimal u, v-separators of G. Then ($S' \subseteq S \cup C^u_S$ and $S \subseteq S' \cup C^v_{S'}$) or ($S' \subseteq S \cup C^v_S$ and $S \subseteq S' \cup C^u_{S'}$).*

Proof. We may assume without loss of generality that S and S' are distinct. As minimal separators of a chordal graph, S and S' are cliques, and since S and S' are distinct minimal u, v-separators of G, $S \setminus S' \neq \emptyset$. Let $x \in S \setminus S'$. Observe that $S' \cap (C^u_S \cup C^v_S) \neq \emptyset$, because otherwise $C^u_S \cup \{x\} \cup C^v_S$ would be a connected subset of

[1]Note that in the incremental approach described in the next section, vertex u is the most recently added vertex.

$V \setminus S'$, which would contradict S' being a u,v-separator of G. Let $x' \in S' \cap (C^u_S \cup C^v_S)$. We first study the case when $x' \in C^u_S$: Since S' is a clique containing x', $S' \subseteq \{x'\} \cup N(x') \subseteq S \cup C^u_S$. It follows that $\{x\} \cup C^v_S$ is a connected subset of $V \setminus S'$, and therefore $x \in C^v_{S'}$. Since S is a clique containing x, $S \subseteq \{x\} \cup N(x) \subseteq S' \cup C^v_{S'}$. For the case when $x' \in C^v_S$, we prove in a similar way that $S' \subseteq S \cup C^v_S$ and $S \subseteq S' \cup C^u_{S'}$. ∎

Theorem 3.8 *Let $G = (V, E)$ be a chordal graph, let u and v be non-adjacent vertices of G, and let graph $M = (V,\ E \cup \{uv\} \cup R(G, u, v))$. Then M is chordal and M is a subgraph of any triangulation of $G' = (V,\ E \cup \{uv\})$ obtained from G' by adding edges incident to u only.*

Proof. Let us first show that M is chordal. Assume on the contrary that M is not chordal, and let C be a chordless cycle of length at least 4 in M. Since G is chordal, C contains an edge $ux \in \{uv\} \cup R(G, u, v)$. Let $C = u - x' - y_1 - y_2 - ... - y_k - x - u$ with $k \geq 1$, and $P_1 = x' - y_1 - y_2 - ... - y_k - x$, which is a chordless path in G. It is sufficient to show that P_1 is a subpath of a chordless path P in G between u and v, since then by Lemma 3.5 uy_1 would belong to $R(G, u, v)$ and therefore would be a chord of C in M, giving a contradiction. In the following $Q_1 \cdot Q_2$ denotes the path obtained by concatenating paths Q_1 and Q_2.

First case : $x = v$ or $x' = v$. Say, $x = v$. If $ux' \in E$ then we are done with $P = u - x' \cdot P_1$. Otherwise $ux' \in R(G, u, v)$, let S be a minimal u,v-separator of G containing x', and let P_0 be a chordless path in G between u and x' with all intermediate vertices belonging to C^u_S. By Lemma 3.6, x' is the only vertex of S in P_1, so all intermediate vertices of P_1 belong to C^v_S. It follows that path $P = P_0 \cdot P_1$ is a chordless path in G between u and v.

Second case : $x \neq v$ and $x' \neq v$. In this case, $ux \in R(G, u, v)$. Let S be a minimal u,v-separator of G containing x and let P_2 be a chordless path in G between x and v with all intermediate vertices belonging to C^v_S. If $ux' \in E$ then by Lemma 3.6, all intermediate vertices of $u - x' \cdot P_1$ belong to C^u_S, so path $P = u - x' \cdot P_1 \cdot P_2$ is a chordless path in G between u and v. Otherwise $ux' \in R(G, u, v)$. Let S' be a minimal u,v-separator of G containing x' and let P_0 be a chordless path in G between u and x' with all intermediate vertices belonging to $C^u_{S'}$. By Lemma 3.7, ($S' \subseteq S \cup C^u_S$ and $S \subseteq S' \cup C^v_{S'}$) or ($S' \subseteq S \cup C^v_S$ and $S \subseteq S' \cup C^u_{S'}$). We may assume without loss of generality that $S' \subseteq S \cup C^u_S$ and $S \subseteq S' \cup C^v_{S'}$. By Lemma 3.6, $x' \notin S$ and $x \notin S'$, so $x' \in C^u_S$ and $x \in C^v_{S'}$. Hence by Lemma 3.6, all intermediate vertices of P_1 belong to $C^v_{S'}$ and C^u_S. Since P_0 and P_1 are chordless and all vertices of P_0 other than x' belong to $C^u_{S'}$, and those of P_1 other than x' belong to $C^v_{S'}$, path $Q = P_0 \cdot P_1$ is chordless. Since $S \subseteq S' \cup C^v_{S'}$, $C^u_{S'} \subseteq C^u_S$, so all vertices of P_0 belong to C^u_S. Since Q and P_2 are chordless and all vertices of Q other than x belong to C^u_S, and those of P_2 other than x belong to C^v_S, path $P = Q \cdot P_2$ is a chordless path in G between u and v, which completes the proof of chordality of M.

Let M' be a triangulation of $G' = (V,\ E \cup \{uv\})$ obtained from G' by adding edges incident to u only. Let us show that M is a subgraph of M', i.e., that every edge of $R(G, u, v)$ is an edge of M'. Suppose on the contrary that there is some edge $ux \in R(G, u, v)$ which is not an edge of M'. By Lemma 3.5, x is an intermediate vertex of a chordless path $u - y_1 - y_2 - ... - y_i = x - ... - y_k = v$ in G. Thus $C = u - y_1 - y_2 - ... - y_k = v - u$ is a cycle in M' such that path $y_1 - y_2 - ... - y_k = v$ is chordless in M'. Let r be the largest integer smaller than i such that y_r is adjacent to u in M' and s be the smallest integer larger than i such that y_r is adjacent to u in M'. Then $u - y_r - y_{r+1} - ... - y_i = x - ... - y_{s-1} - y_s - u$ is a chordless cycle in M' of length at least 4, which contradicts the chordality of M'. ■

The following corollary follows directly from Theorem 3.8.

Corollary 3.9 *Let $G = (V, E)$ be a chordal graph and let u and v be any pair of non-adjacent vertices in G. Then $M = (V,\ E \cup \{uv\} \cup R(G, u, v))$ is the unique minimal triangulation of $G' = (V,\ E \cup \{uv\})$ obtained from G' by adding edges incident to u only.*

A significant consequence of our main theorem is that it is sufficient to determine $R(G, u, v)$ and add it to the current graph G to obtain a new chordal graph. This can be done efficiently, as will be explained in Section 5. This involves maintaining the minimal separator structure of a chordal graph, a problem for which we have a new and efficient data structure associated with a clique tree, which we will describe in Section 5.

4 A vertex incremental algorithm for simultaneous maximal subtriangulation and minimal triangulation

In this section we apply our results of Section 3 to the problem of computing a maximal chordal subgraph $H = (V, D)$ and a minimal triangulation $M = (V, F)$ of an arbitrary graph $G = (V, E)$, where $D \subseteq E \subseteq F$.

Our algorithm is based on the following vertex incremental principle. Start with an empty subset U of V, and a maximal chordal subgraph H of $G(U)$ (respectively a minimal triangulation M of $G(U)$ if we want a minimal triangulation algorithm). The incremental approach is to increase U with a vertex u from $V \setminus U$ at each step. Observe that H (resp. M) is chordal and disconnected after the introduction of u as long as $|U| \geq 1$, since no edges are introduced along with this vertex, and H (resp. M) was chordal before this step. Then for each edge of G incident to u and some vertex v in $U \setminus \{u\}$, we do computations according to Theorem 3.8 and obtain the set $R(H, u, v)$ (resp. $R(M, u, v)$) of edges incident to u that must be added along with uv in order to obtain a chordal supergraph of H (resp. M).

In the case of the maximal subtriangulation algorithm, we will only add uv and $R(H, u, v)$ to $E(H)$ if $R(H, u, v) \subset E(G)$. In the case of the minimal triangulation algorithm, the required edges $R(M, u, v)$ and the edge uv are added to $E(M)$. To prove that this approach actually produces a maximal chordal subgraph (resp. minimal triangulation) we rely on the results in the following two lemmas.

Lemma 4.1 *Given $G = (V, E)$, let $U \subseteq V$, and let H and H' be graphs such that H is a subgraph of H', H' is a chordal subgraph of G, and $H(U)$ is a maximal chordal subgraph of $G(U)$. Then $H(U) = H'(U)$.*

Proof. As an induced subgraph of a chordal graph $H'(U)$ is chordal, and therefore is a chordal subgraph of $G(U)$ having $H(U)$ as a subgraph. So, since $H(U)$ is a maximal chordal subgraph of $G(U)$, $H(U) = H'(U)$. ∎

For the computation of H, assume that H is a maximal chordal subgraph of $G(U)$ on vertex set U. At the first step, U contains a single vertex of G and $H = G(U)$. At each step, a vertex $u \in V \setminus U$ is chosen and added to U and thus to H. Now for each edge uv of G with $v \in U$, we add edge uv to H if and only if every edge of $R(H, u, v)$ is present in G. If uv is added to H, we also add every edge of $R(H, u, v)$ at the same time. After this, none of the edges that are added need to be examined again for possible addition, since they already appear in the transitory chordal subgraph. If some edge of $R(H, u, v)$ is not an edge of G, then we cannot add uv at this step by Theorem 3.8, since we only allow addition of edges incident to u. When we prove the correctness of our algorithm, it will be clear that uv never needs to be examined again for addition. [2] Thus, each edge is examined for addition at most once, and in many cases several edges are added at the same time and disappear from the list of edges that still need to be examined, which is the strength of our algorithm with respect to time complexity. In addition, our algorithm does not touch the unprocessed vertices. Thus, these vertices need not be known in advance, and we can actually take a new vertex u as input in an on-line fashion at each step.

Lemma 4.2 *Given $G = (V, E)$, let $U \subseteq V$, and let M and M' be graphs such that M' is a subgraph of M, M' is a triangulation of G, and $M(U)$ is a minimal triangulation of $G(U)$. Then $M(U) = M'(U)$.*

Proof. As an induced subgraph of a chordal graph, $M'(U)$ is chordal, and therefore is a triangulation of $G(U)$ and a subgraph of $M(U)$. So, since $M(U)$ is a minimal triangulation of $G(U)$, $M(U) = M'(U)$. ∎

For the computation of M, assume that M is a minimal triangulation of $G(U)$ on the vertex set U. The only difference from the discussion above in this case is that, for each edge uv of G with $v \in U$, we add to M edge uv as well as *every edge belonging to* $R(M, u, v)$ regardless of whether or not these edges belong to G. Thus, the difference between the two processes is merely a single **if** statement. Our algorithm can be changed by inserting or deleting this **if** line in order to change between the processes of computing a minimal triangulation and a maximal chordal subgraph, though of course both graphs can be computed by a single algorithm within the same time bound.

With the data structure details given in the next section, we will show that computing and adding the set $R(H, u, v)$ can be done in $O(n)$ time for each examined edge uv. From our algorithm and its proof of correctness, it will be clear that every edge needs to be examined at most once. We are now ready to present our algorithm. We begin with the maximal chordal subgraph version.

Algorithm Incremental Maximal Subtriangulation **(IMS)**

[2] Note that considering $R(H, v, u)$ does not help, since we have already concluded that adding any edge between v and vertices in $U \setminus \{u\}$ will create a chordless cycle.

Input: $G = (V, E)$.
Output: A maximal chordal subgraph $H = (V, D)$ of G.

```
01. Pick a vertex $s$ of $G$;
02. $U = \{s\}$;
03. $D = \emptyset$;
04. for $i = 2$ to $n$ do
05.     Pick a vertex $u \in V \setminus U$;
06.     $U = U \cup \{u\}$;
07.     $N = \emptyset$;
08.     for each vertex $w \in N_G(u)$
09.         if $w \in U$ then
10.             $N = N \cup \{w\}$;
11.         end-if
12.     end-for
13.     while $N$ is not empty do
14.         Pick a vertex $v \in N$;
15.         $N = N \setminus \{v\}$;
16.         $X = \{x \mid x$ belongs to a minimal $u, v$-separator of $H = (U, D)\}$;
17.         $R = \{ux \mid x \in X\}$;
18.         if $R \subseteq E$ then
19.             $D = D \cup \{uv\} \cup R$;
20.             $N = N \setminus X$;
21.         end-if
22.     end-while
23.     $H = (U, D)$;
24. end-for
```

Let us call **IMT** (Incremental Minimal Triangulation) the algorithm that results from removing lines 18 and 21 of Algorithm **IMS**. Thus, in **IMT**, edge set $\{uv\} \cup R$ is always added to the transitory graph for every examined edge uv. In Example 4.3, executions of both of these algorithms are shown on the same input graph. Figure 1 (a) shows **IMS** and (b) shows **IMT**.

Example 4.3 *Consider Figure 1. The vertices of the input graph are processed in the order shown by the numbers on the vertices. At step 1, only vertex* 1 *is added to* H. *At step 2, vertex* 2 *and edge* 21 *are added, and similarly at steps 3 and 4, vertex* 3 *and edge* 32*, and vertex* 4 *and edge* 41 *are added, respectively. The first column of the figure shows graph* H *with thick lines on the input graph after these 4 steps. The chordal graph so far is the same for both the maximal chordal subgraph (a), and the minimal triangulation (b). We will explain the rest of the executions in more detail.*

(a) *At step 5,* $N = \{3, 4\}$*, and edge* 53 *is examined first. In this case, set* X *is empty, and edge* 53 *is thus added. For the addition of edge* 54*,* $X = \{1, 2, 3\}$*, and since required edges* 51 *and* 52 *are not present in* G*, edge* 54 *is not added. At step 6,* $N = \{3, 4, 5\}$*, and edge* 63 *is examined first and added since* X *is empty. For the*

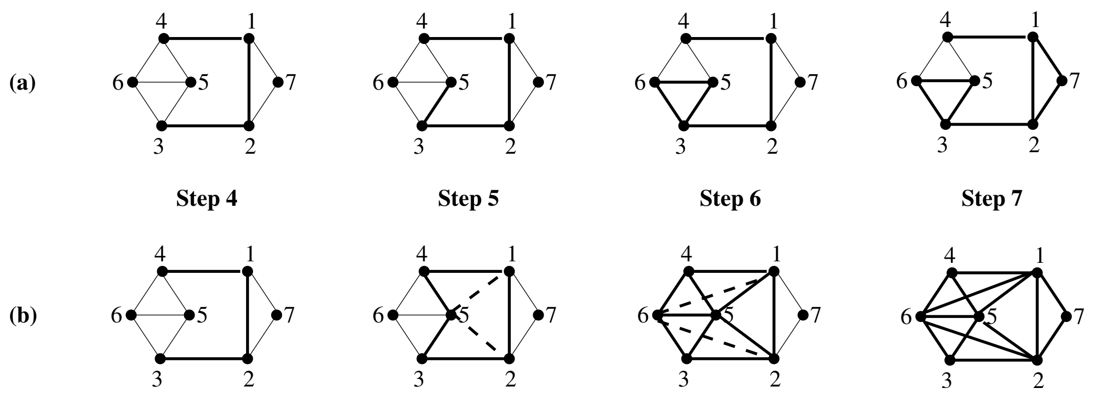

Figure 1: The figure shows graph H in thick lines after steps 4, 5, 6, and 7 of (a) Algorithm **IMS** when computing a maximal chordal subgraph and (b) Algorithm **IMT** when computing a minimal triangulation.

addition of edge 64*,* $X = \{1, 2, 3\}$*, and since required edges* 61 *and* 62 *are not present in* G*, edge* 64 *is not added. For the addition of edge* 65*,* $X = \{3\}$*, and* 65 *is added since edge* 63 *is present in* G *and in* H*.*

(b) *At step 5, edge* 53 *is added as in (a), and in addition, edge* 54 *is added along with the required edges* 51 *and* 52*. At step 6, edge* 63 *is added as in (a). For the addition of edge* 64*,* $X = \{1, 2, 3, 5\}$ *since the minimal* 6, 4*-separators are* $\{1, 5\}, \{2, 5\}$*, and* $\{3\}$*. Thus, edge* 64 *and required edges* 61*,* 62*, and* 65 *are added to* M*.*

Step 7 adds edges 71 *and* 72 *in both (a) and (b) without requiring any additional edges in either case.*

Theorem 4.4 *Algorithm* **IMT** *computes a minimal triangulation, and Algorithm* **IMS** *computes a maximal chordal subgraph, of the input graph.*

Proof. Let $(u_1, u_2, ..., u_n)$ be the sequence of vertices of G successively added to U in an execution of Algorithm **IMT** (resp. **IMS**), and let $U_i = \{u_1, u_2, ..., u_i\}$ for any i from 1 to n.

Algorithm **IMT** : Let $M = (V, F)$ be the output graph. We show by induction that M is a minimal triangulation of G.

Induction hypothesis: $M(U_i)$ is a minimal triangulation of $G(U_i)$, for $1 \leq i \leq n$.

The base case $i = 1$ trivially holds. Assume that $M(U_{i-1})$ is a minimal triangulation of $G(U_{i-1})$ for some i between 2 and n, and we will show that this implies that $M(U_i)$ is chordal and is equal to any triangulation M' of $G(U_i)$ that is a subgraph of $M(U_i)$.

Let M' be a triangulation of $G(U_i)$ that is a subgraph of $M(U_i)$. By Lemma 4.2, $M(U_{i-1})$ is chordal and $M(U_{i-1}) = M'(U_{i-1})$. Let $u = u_i$, $U = U_i$, and $(v_1, v_2, ..., v_k)$ be the sequence of neighbors of u in G successively picked out of N after adding u to U. Let $M_j = (U, F_j)$ be the transitory graph after processing edge uv_j, for $0 \leq j \leq k$ (M_0 is the transitory graph after adding u to U). Let us prove by induction that M_j is chordal and is a subgraph of M' for $0 \leq j \leq k$. The base case $j = 0$ holds since M_0 is obtained from $M(U_{i-1})$ by adding vertex u, $M(U_{i-1})$ is chordal and $M(U_{i-1}) = M'(U_{i-1})$. Assume that M_{j-1} is chordal and is a subgraph of M', for some j between 1 and k. Let us show that this implies that the same is true for M_j, too. $M_j = (U,\ F_{j-1} \cup \{uv_j\} \cup R(M_{j-1}, u, v_j))$. Thus M' is a triangulation of $M'' = (U,\ F_{j-1} \cup \{uv_j\})$ obtained from M'' by adding edges incident to u only, since $M''(U_{i-1}) = M(U_{i-1}) = M'(U_{i-1})$. By Theorem 3.8, M_j is chordal and is a subgraph of M', which completes this part of the proof by induction on j.

As a consequence, $M(U_i) = M_k$ is chordal and is a subgraph of M' and since M' is a subgraph of $M(U_i)$, $M' = M(U_i)$. This completes the proof by induction on i, and thus M is a minimal triangulation of G.

Algorithm **IMS** : Let $H = (V, D)$ be the output graph. We show again by induction that H is a maximal chordal subgraph of G.

Induction hypothesis: $H(U_i)$ is a maximal chordal subgraph of $G(U_i)$, for $1 \leq i \leq n$.

The base case $i = 1$ trivially holds. Assume that $H(U_{i-1})$ is a maximal chordal subgraph of $G(U_{i-1})$, for some i between 2 and n. Let us show that $H(U_i)$ is chordal and is equal to any chordal subgraph H' of $G(U_i)$ having $H(U_i)$ as a subgraph.

Let H' be a chordal subgraph of $G(U_i)$ having $H(U_i)$ as a subgraph. By Lemma 4.1, $H(U_{i-1})$ is chordal and $H(U_{i-1}) = H'(U_{i-1})$. We define u, U, $(v_1, v_2, ..., v_k)$, and $H_j = (U, D_j)$ for $0 \leq j \leq k$ as above. Let us prove by induction on j that H_j is chordal, and if $j \geq 1$ and uv_j is an edge of H', then $uv_j \in D_j$. The base case $j = 0$ holds since H_0 is obtained from chordal graph $H(U_{i-1})$ by adding vertex u. Suppose that H_{j-1} is chordal, and if $j - 1 \geq 1$ and uv_{j-1} is an edge of H', then $uv_{j-1} \in D_{j-1}$, for some j between 1 and k. We show that this implies that the same is true for H_j, v_j and D_j. Let $K = (U,\ D_{j-1} \cup \{uv_j\} \cup R(H_{j-1}, u, v_j))$. H_j is either equal to H_{j-1} or to K, and since H_{j-1} is chordal, by Theorem 3.8, H_j is chordal. Now we assume that uv_j is an edge of H'. Let us show that $uv_j \in D_j$. H' is a triangulation of $H'' = (U,\ F_{j-1} \cup \{uv_j\})$ obtained from H'' by adding edges incident to u only (since $H''(U_{i-1}) = H(U_{i-1}) = H'(U_{i-1})$). By Theorem 3.8, K is a subgraph of H', and therefore of $G(U)$. It follows that $R(H_{j-1}, u, v_j) \subseteq E$, which is the condition for adding edge uv_j, so $uv_j \in D_j$. Thus we have completed the part of the proof by induction on j.

As a consequence, $H(U_i) = H_k$ is chordal and every edge of H' incident to u which has been processed is an edge of $H(U_i)$. Since moreover $H(U_{i-1}) = H'(U_{i-1})$, unprocessed edges of G are edges of H and $H(U_i)$ is a subgraph of H', $H' = H(U_i)$. This completes the proof by induction on i, and thus H is a maximal chordal subgraph of G. ■

5 Data structure details and time complexity

The input graph G is represented by adjacency list data structure, and we use a clique tree T of H as an additional data structure to store and work on the transitory graph H. Thus, after the first step, T has only one tree node, which contains start vertex

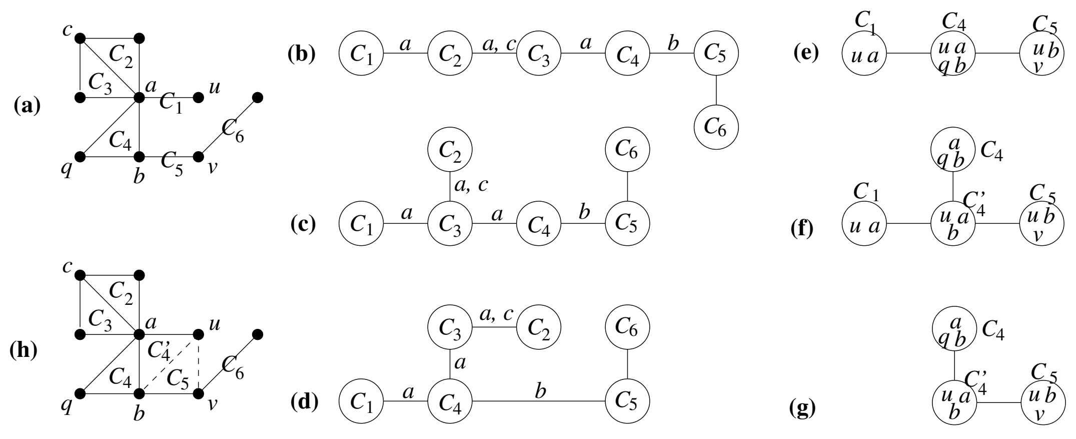

Figure 2: A chordal graph H is given in (a), and (b),(c), and (d) shows a clique tree of H, where $C_u = C_1$, $C_v = C_5$, and $P_{u,v}$ is the path between C_1 and C_5. After steps (c) and (d), path $P_{u,v}$ between C_1 and C_5 is in the desired form, and only this portion of the tree is shown after step (d). In step (e), u is placed in every tree node on $P_{u,v}$, and in step (f) C_4 is separated from the path since edge uq is not intended. C_1 is removed in (g) since it becomes non-maximal. The new corresponding graph H of which the modified tree is a clique tree is shown in (h).

s. As H grows, T will grow maintaining a correct clique tree of H at all steps. Note that T will not always be connected at intermediate steps, as H is not necessarily connected. In this case, each connected component of T will be a correct clique tree of the corresponding connected component of H.[3]

In what follows we describe an implementation of each of the following operations.

1. Compute the union X of all minimal u,v-separators in H, which gives the required edge set $R(H,u,v)$.

2. If $R(H,u,v) \cup \{uv\}$ is to be added to H, update T to reflect this modification of H.

Each of these operations will be shown to require only $O(n)$ time for each examined edge uv of G. We will devote a subsection to each of the above mentioned operations. Subsection 5.1 describes how T is modified to obtain a path $P_{u,v}$ such that every tree edge on $P_{u,v}$ is a distinct minimal u,v-separator (Figure 2(b)-(d)), and how the union X of all these minimal separators is computed from $P_{u,v}$. Subsection 5.2 describes how T is further modified to reflect the addition of new edges to H (Figure 2(e)-(g)), and how to ensure that every tree node in T is a unique maximal clique of H after the modifications.

Since we examine each edge at most once, and there are m edges, the desired time bound will then follow. An illustration of what happens for each examined edge uv is

[3]We could have picked the new vertex u such that $u \in N_G(U)$, but this would result in less general algorithms unnecessarily, and we want our algorithms to have on-line implementations.

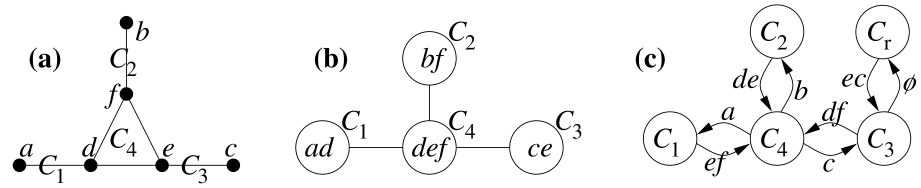

Figure 3: A chordal graph H and its maximal cliques are given in (a). A clique tree of H is given in (b), where each tree node contains the vertices of a unique maximal clique in H. The difference list representation of the same clique tree as in (b) is presented in (c). Tree node C_r is an empty tree node, that is used when we want to compute the set of vertices contained in a tree node. The arrows represent the add and remove lists, and the vertices contained in each list are given by the label attached to the arrow.

summarized in Figure 2; we will refer to parts of this figure as we explain the details in the coming subsections. The main idea is to use a path $P_{u,v}$ of the current clique tree T between a tree node C_u that contains u and a tree node C_v that contains v, and compute the union of tree edges on this path that correspond to minimal u,v-separators. Unfortunately, the sum of the sizes of these edges can be larger than $O(n)$; in fact each edge can be of size $O(n)$. Thus, if the tree nodes and tree edges of T are implemented simply as vertex lists containing vertices of each tree node and edge, then Operation 1 described above cannot be accomplished in $O(n)$ time. For this reason, we present a special kind of implementation of the clique tree, as described below.

Every edge CC' of T is implemented as two lists that we will call *difference lists* (*difflists* for short). One list contains vertices belonging to $C \setminus C'$. This list has two names; it is called both $\mathit{add}(C', C)$ and $\mathit{remove}(C, C')$. The other list contains vertices belonging to $C' \setminus C$. This list is called $\mathit{add}(C, C')$ and also $\mathit{remove}(C', C)$. Now, if every tree node C of T contains pointers to its *add* and *remove* lists, *add* and *remove* lists are in edges of T and not in every clique C of T then C actually does not need to store a list of vertices that it contains. Vertices belonging to C can be computed by using the edges incident to C as follows. For every edge CC', if we know the set of vertices belonging to C', then we add $\mathit{add}(C', C)$ to this set and remove $\mathit{remove}(C', C)$ from this set to get the set of vertices belonging to C. In order to have a starting tree node we need to know the content of one tree node, this might be an empty tree node. Note that the add and remove lists and the structure of the tree are the only stored information in this data structure. In Figure 3 a chordal graph is given in (a), while the regular clique tree of this graph is given in (b), and the clique tree represented by difference lists is given in (c).

5.1 Finding the minimal u, v-separators and computing X

Let K_u be any maximal clique of H that contains u, and let K_v be any maximal clique that contains v. If K_u and K_v are contained in different connected components of T, then $X = \emptyset$ and there is nothing to compute. Let us for the rest of this subsection assume that K_u and K_v are contained in the same connected component of T.

On the path from K_u to K_v in T, do a search from K_u to K_v and let C_u be the tree node closest to K_v containing u. Do a similar search from K_v to K_u and let C_v be the tree node closest to K_u containing v. Let $P_{u,v}$ denote the path of T between C_u and C_v. An example graph H and $P_{u,v}$ are given in Figure 2 (a) and (b), respectively.

Claim 5.1 *Every minimal u, v-separator appears as an edge of $P_{u,v}$, and every edge on $P_{u,v}$ separates u and v.*

Proof. By Theorem 2.1 we know that every internal node of $P_{u,v}$ contains neither u nor v. Theorem 2.3 ensures that every minimal separator S appearing as an edge of $P_{u,v}$ separates u and v, since removing S will separate C_u from C_v in the clique tree, thus also separate u and v in the graph. Conversely, for any minimal u, v-separator S of the graph, S contains as a subset some edge S' of $P_{u,v}$, since removing S will separate u and v in the graph, thus also separate C_u from C_v in the clique tree. Now, since S is a minimal u, v-separator containing S' and by the first part of this proof, the edge S' of $P_{u,v}$ separates u and v, S is equal to S'. Thus S appears as an edge of $P_{u,v}$. ∎

However, some of the tree edges on $P_{u,v}$ might be non-minimal u, v-separators, and some minimal u, v-separators might appear several times as edges of $P_{u,v}$. We will first modify T in $O(n)$ time so that path $P_{u,v}$ between C_u and C_v contains only distinct minimal u, v-separators as its edges.

Observe first that, since every vertex can appear only once in an *add* list and once in a *remove* list on the path $P_{u,v}$, the sum of the lengths of the *add* and *remove* lists on the path $P_{u,v}$ is at most $2n$. We obtain our time bound by reading every *add* and *remove* list in $P_{u,v}$ at most a constant number of times. The maximal cliques (tree nodes) on $P_{u,v}$ are named $C_1, C_2, ..., C_k$, where $C_u = C_1$ is the tree node containing vertex u, and $C_v = C_k$ is the tree node containing v, and edge $S_i = S_{i,i+1} = C_iC_{i+1}$ is an edge of $P_{u,v}$, for $1 \leq i \leq k-1$. We will describe how unnecessary maximal cliques can be removed from $P_{u,v}$, and after each removal, we will assume that the remaining maximal cliques and minimal separators are resorted as $C_1, C_2, ...C_k$, so that before we explain each new modification, we have a consecutive numbering of the maximal cliques on $P_{u,v}$. When we must remove edges of $P_{u,v}$ and insert edges between two non-consecutive tree nodes of $P_{u,v}$, we will need a more general way of naming the intersections between two maximal cliques that are not necessarily adjacent in T. We let $S_{i,j} = C_i \cap C_j$ denote the intersection between C_i and C_j.

Claim 5.2 *Assume that there is a tree edge S_j on $P_{u,v}$, such that either S_j is not a minimal u, v-separator in H or S_j is equal to another minimal separator appearing on $P_{u,v}$. Then there exists a tree edge S_i such that $S_i \subseteq S_{i+1}$ or $S_i \subseteq S_{i-1}$.*

Proof. Observe first that, by Claim 5.1, every minimal u, v-separator appears as an edge of $P_{u,v}$, and every edge on this path separates u from v. Thus, S_j is a (not necessarily minimal) u, v-separator. Consequently, there exists a minimal u, v-separator S_i such that $S_i \subseteq S_j$ and $i \neq j$. Let S_{i+1} or S_{i-1} be the edge adjacent to S_i in $P_{u,v}$ in the direction of S_j. Note that $i+1$ or $i-1$ and j might be equal. It follows from Theorem 2.3 that $S_i \subseteq S_{i+1}$ or $S_i \subseteq S_{i-1}$ since $S_i \subseteq S_j$. ∎

From Claim 5.2, we can conclude that, if no two adjacent tree edges are comparable by subset relation, then $P_{u,v}$ contains only distinct minimal u, v-separators. We will test adjacent tree edges on $P_{u,v}$, and remove the ones that include their neighbors as a subset. In order to obtain our time bound we have to do this test in the difflist data structure.

Claim 5.3 *Let the following path C_i, S_i, C_j, S_j, C_l be a subpath of $P_{u,v}$. Then $S_i \subseteq S_j$ if and only if $remove(C_j, C_l) \subseteq add(C_i, C_j)$.*

Proof. Assume that $remove(C_j, C_l) \subseteq add(C_i, C_j)$. Observe that $S_i \cup add(C_i, C_j) = S_j \cup remove(C_j, C_l) = C_j$, since $S_i = C_i \cap C_j$ and $S_j = C_j \cap C_l$. Remember that *add* and *remove* lists only contain the new vertices, and thus $S_i \cap add(C_i, C_j) = S_j \cap remove(C_j, C_l) = \emptyset$. We can now conclude that $S_i \subseteq S_j$, since $remove(C_j, C_l) \subseteq add(C_i, C_j)$ and $S_i \cup add(C_i, C_j) = S_j \cup remove(C_j, C_l)$ and $S_i \cap add(C_i, C_j) = S_j \cap remove(C_j, C_l) = \emptyset$. For the other direction, assume that $S_i \subseteq S_j$. By the same arguments as in the opposite direction it follows that $remove(C_j, C_l) \subseteq add(C_i, C_j)$, since $S_i \cup add(C_i, C_j) = S_j \cup remove(C_j, C_l)$ and $S_i \cap add(C_i, C_j) = S_j \cap remove(C_j, C_l) = \emptyset$. ■

Claim 5.4 *Let $S_{i,i+1}$ and $S_{j,j+1}$ be tree edges on the path $P_{u,v}$ in the clique tree T, such that $S_{i,i+1} \subseteq S_{j,j+1}$ and $i < j$. Let T' be a clique tree obtained from T, by deleting the tree edge $S_{i,i+1}$ and inserting the tree edge $S_{i,j+1}$. Then T' is a clique tree of the chordal graph H represented by T.*

Proof. The maximal cliques in the clique tree T, are untouched by this operation, so there exists a maximal clique in T' containing the vertex pair u, v if and only if $uv \in E(H)$, and every vertex of H is contained in some maximal clique of T'. Since $S_{i,i+1} \subset C_i$ and $S_{i,i+1} \subseteq S_{j,j+1} \subset C_{j+1}$ then $S_{i,j+1} = C_i \cap C_{j+1} = S_{i,i+1}$, thus it follows that the set of maximal cliques containing a vertex of H induces a connected tree in T' since this is true for T. ■

Given two tree edges $S_{i,i+1}$ and $S_{j,j+1}$ of $P_{u,v}$ then Claim 5.4 can be used to reduce the length of the path $P_{u,v}$ Thus, after this modification, the subpath of $P_{u,v}$ between C_i and C_{j+1} is reduced to $C_i, S_{i,j+1}, C_{j+1}$. This situation corresponds to the change from (b) to (c) in Figure 2. The new *add* and *remove* lists for the tree edge $S_{i,j+1}$ can be computed in the following way:

$$add(C_i, C_{j+1}) = \bigcup_{i \le q < j+1} add(C_q, C_{q+1}) \setminus \bigcup_{i < q < j+1} remove(C_q, C_{q+1}) \tag{1}$$

$$remove(C_i, C_{j+1}) = \bigcup_{i \le q < j+1} remove(C_q, C_{q+1}) \setminus \bigcup_{i < q < j+1} add(C_{q-1}, C_q). \tag{2}$$

The list $add(C_i, C_{j+1})$ can be computed in time $O(\ |\bigcup_{i \le q < j+1} add(C_q, C_{q+1})| \ + |\bigcup_{i < q < j+1} remove(C_q, C_{q+1})|\ )$ in the following way. Let A be a characteristic vector of size n, where every element is 0. For every vertex $u \in \bigcup_{i \le q < j+1} add(C_q, C_{q+1})$, set

$A[u] = 1$, and then for every vertex $u \in \bigcup_{i<q<j+1} \mathit{remove}(C_q, C_{q+1})$, set $A[u] = 0$. Now $\mathit{add}(C_i, C_{j+1})$ can be computed as follows. For every vertex $u \in \bigcup_{i\leq q<j+1} \mathit{add}(C_q, C_{q+1})$ where $A[u] = 1$, add u to $\mathit{add}(C_i, C_{j+1})$. In order to reuse the vector, we clean up: for every vertex $u \in \bigcup_{i\leq q<j+1} \mathit{add}(C_q, C_{q+1})$, set $A[u] = 0$. The list $\mathit{remove}(C_i, C_{j+1})$ is computed in the same way.

Claim 5.5 *Let $S_{i,i+1}$ and $S_{j,j+1}$ be tree edges on the path $P_{u,v}$ in the clique tree T, such that $i < j$. Then $S_{i,i+1} \subseteq S_{j,j+1}$ if and only if $\mathit{remove}(C_q, C_{q+1}) \subseteq \mathit{add}(C_i, C_q)$ for $i < q \leq j$.*

Proof. Let us first show that $S_{i,i+1} \subseteq S_{j,j+1}$ if $\mathit{remove}(C_q, C_{q+1}) \subseteq \mathit{add}(C_i, C_q)$ for $i < q \leq j$. This is proved by induction on j, $i < j$.for any given i, where Claim 5.3 corresponds to $j = i + 1$, which we will use as the base case. Now for the induction hypothesis, let us assume that $S_{i,i+1} \subseteq S_{j,j+1}$ if $\mathit{remove}(C_q, C_{q+1}) \subseteq \mathit{add}(C_i, C_q)$ for $i < q \leq j$, and let us prove that $S_{i,i+1} \subseteq S_{j+1,j+2}$ if moreover $\mathit{remove}(C_{j+1}, C_{j+2}) \subseteq \mathit{add}(C_i, C_{j+1})$. Since $S_{i,i+1} \subseteq S_{j,j+1}$ then by Claim 5.4, the path from C_i to C_{j+1} can be reduced to $C_i, S_{i,j+1}, C_{j+1}$, and from the proof of Claim 5.4 we know that $S_{i,j+1} = S_{i,i+1}$. Finally it follows by Claim 5.3 that $S_{i,i+1} = S_{i,j+1} \subseteq S_{j+1,j+2}$ since $\mathit{remove}(C_{j+1}, C_{j+2}) \subseteq \mathit{add}(C_i, C_{j+1})$ in the new path from C_i to C_{j+2}.

For the other direction, we want to show that $\mathit{remove}(C_q, C_{q+1}) \subseteq \mathit{add}(C_i, C_q)$ for $i < q \leq j$ if $S_{i,i+1} \subseteq S_{j,j+1}$. Let q such that $i < q \leq j$. From Theorem 2.3 it follows that $S_{i,i+1} \subseteq S_{q-1,q}$ and $S_{i,i+1} \subseteq S_{q,q+1}$. Since $S_{i,i+1} \subseteq S_{q-1,q}$, from Claim 5.4 the tree edge $S_{i,i+1}$ can be replaced with $S_{i,q}$, where $S_{i,i+1} = S_{i,q}$. Since $S_{i,q} = S_{i,i+1} \subseteq S_{q,q+1}$, it follows by Claim 5.3 that $\mathit{remove}(C_q, C_{q+1}) \subseteq \mathit{add}(C_i, C_q)$. ∎

If we do this reduction for every pair S_i, S_{i+1} and S_{i+1}, S_i of edges on the path $P_{u,v}$, then it follows from Claim 5.2 that every tree edge on $P_{u,v}$ is a distinct minimal u, v-separator. Thus, we are done with the part that is illustrated in Figure 2(b)-(d). However, it remains to explain how to examine adjacent tree edges in such a way that the total time bound $O(n)$ is maintained.

The idea is to do this in two scans. One from C_u to C_v, and one from C_v to C_u. The same operation is done for both directions, so we will only explain the scan from C_u to C_v. Consider the tree edges in the order given by $P_{u,v}$. For a given tree edge S_i (starting from S_1) we will try to find the largest number t such that the intersection $S_{i,i+t} = C_i \cap C_{i+t}$ is equal to S_i. Replace S_i by $S_{i,i+t}$ and continue by finding the next t using S_{i+t} as the new S_i, and repeat this until $C_{i+t} = C_v$.

A consequence of Theorem 2.3 is that there exists no $q > 0$ such that $S_i = S_{i,i+t} \subseteq S_{i+t+q}$, since $S_{i,i+t} \not\subseteq S_{i+t+1}$, which is the property we want for all tree edges on $P_{u,v}$ in both directions.

Computing the $\mathit{add}(C_i, C_{i+t})$ and $\mathit{remove}(C_i, C_{i+t})$ lists can be done as previously described by only reading the *add* and *remove* lists on the path $P_{u,v}$ between C_i and C_{i+t}. The next search starts from C_{i+t}, and thus the total time used to compute all such lists are $O(n)$, since every *add* and *remove* list on the path $P_{u,v}$ is only used to compute the difflists for one new tree edge, and the new tree edges are never used to create other tree edges.

It remains to efficiently compute the value t, given a path $P_{u,v}$ and a tree edge $S_i = S_{i,i+1}$. The basic idea is as follows. Start with $t = 1$. While $S_{i,i+1} \subseteq S_{i,i+t+1}$, increment t and repeat the test until $S_{i+1} \not\subseteq S_{i,i+t+1}$ or $i + t = k$. From Claim 5.5 we know that this is equivalent to testing whether $\mathit{remove}(C_{i+q}, C_{i+q+1}) \subseteq \mathit{add}(C_i, C_{i+q})$, for $1 \leq q \leq t$. It is important to notice that if $S_{i+1} \subseteq S_{i,i+t}$, then we can verify if $S_{i+1} \subseteq S_{i,i+t+1}$ by testing if $\mathit{remove}(C_{i+t}, C_{i+t+1}) \subseteq \mathit{add}(C_i, C_{i+t})$, since we already know that $\mathit{remove}(C_{i+q}, C_{i+q+1}) \subseteq \mathit{add}(C_i, C_{i+q})$, for $1 \leq q < t$. The $\mathit{remove}(C_{i+t}, C_{i+t+1})$ is only read once for each time we increment t, and notice that $\mathit{remove}(C_i, C_{i+1})$ is not used in this test. Let us now argue that we do not read any of the *remove* lists over again when we continue to find the next t. Every remove list we read is removed from the path $P_{u,v}$, except $\mathit{remove}(C_{i+t}, C_{i+t+1})$ which we used to decide that $S_{i,i+1} \not\subseteq S_{i,i+t+1}$. The $\mathit{remove}(C_{i+t}, C_{i+t+1})$ list becomes the new $\mathit{remove}(C_i, C_{i+1})$ list since we use C_{i+t} as the new C_i when we search for the next t. The newly created $\mathit{remove}(C_i, C_{i+t})$ will not be used to find the next t since we use C_{i+t} as C_i. Thus it follows that every remove list on the path $P_{u,v}$ is only used once for testing.

So it remains to explain how the list $\mathit{add}(C_i, C_{i+t})$ is computed and checked against $\mathit{remove}(C_{i+t}, C_{i+t+1})$ list within the time bound. This is done by using a characteristic vector A of size n as an additional data structure. We will manipulate the vector A, such that $A[u] = 1$ if and only if $u \in \mathit{add}(C_i, C_{i+t})$, and then we check in $O(|\mathit{remove}(C_{i+t}, C_{i+t+1})|)$ time if $\mathit{remove}(C_{i+t}, C_{i+t+1}) \subseteq \mathit{add}(C_i, C_{i+t})$. These checks can be done within the time bound, given that the vector A contains the $\mathit{add}(C_i, C_{i+t-1})$ list and that these add lists are provided in an increasing order for the parameter t.

Equation 1 can be rewritten in the following way: $\mathit{add}(C_i, C_{i+t}) = \mathit{add}(C_i, C_{i+t-1}) \cup \mathit{add}(C_{i+t-1}, C_{i+t}) \setminus \mathit{remove}(C_{i+t-1}, C_{i+t})$. This enables us to obtain $\mathit{add}(C_i, C_{i+t})$ by setting $A[u] = 1$ for every vertex $u \in \mathit{add}(C_{i+t-1}, C_{i+t})$, and setting $A[u] = 0$ for every vertex $u \in \mathit{remove}(C_{i+t-1}, C_{i+t})$, given that A contains $\mathit{add}(C_i, C_{i+t-1})$. Notice that $\mathit{add}(C_{i+t-1}, C_{i+t})$ and $\mathit{remove}(C_{i+t-1}, C_{i+t})$ are not used to compute $\mathit{add}(C_i, C_{i+t-1})$. Therefore, by setting $A[u] = 1$ if $u \in \mathit{add}(C_i, C_{i+1})$ when $t = 1$, then the sequence of lists $\mathit{add}(C_i, C_{i+q})$, for $q \leq k + 1$, can be created in A in increasing order by only reading the *add* and *remove* lists on the path between C_i and C_{i+t+1} once. We have to ensure that every element of A is 0 before we start to compute the next t. This is done within the time bound by reading the *add* lists between C_i and C_{i+t+1} once more, and setting $A[u] = 0$ for every vertex u contained in one of these *add* lists.

We have now argued that every operation required to reduce the path $P_{u,v}$ so that every tree edge in $P_{u,v}$ is a distinct minimal u, v-separator, can be executed by only reading each *add* and *remove* list of $P_{u,v}$ a constant number of times. Since every vertex only can appear once in a *add* list of $P_{u,v}$ and once in a *remove* list, it follows that the reduction of $P_{u,v}$ is an $O(n)$ operation.

Now we will see how to compute X. A pair ux belongs to $R(H, u, v)$ if there exists a minimal u, v-separator containing x. Our goal is to compute the set X of vertices, where $x \in X$ if $ux \in R(H, u, v)$. Observe that $C_1 \subseteq N(u)$, and thus, only the vertices not in S_1 are of interest. The path $P_{u,v}$ is already modified such that every tree edge is a minimal u, v-separator, and every minimal u, v-separator is a tree edge in this path. We can compute X in the following way: start in C_u with an empty vertex set X. Then for $1 \leq i < k-1$ add the vertices contained in $\mathit{add}(C_i, C_{i+1}) \setminus \mathit{remove}(C_{i+1}, C_{i+2})$ to X.

There might be vertices that are only contained in a single maximal clique C_i, and thus not contained in any tree edge. These vertices will be contained in both $add(C_i, C_{i+1})$ and $remove(C_{i+1}, C_{i+2})$. We obtain the desired set X using a characteristic vector A. For each vertex $u \in remove(C_{i+1}, C_{i+2})$ set $A[u] = 1$, then for each vertex $u \in add(C_i, C_{i+1})$ where $A[u] = 0$ add u to X. Finally for the clean up we set $A[u] = 0$ for each vertex $u \in remove(C_{i+1}, C_{i+2})$. Thus, we obtain the desired set X in $O(n)$ time since the *add* and *remove* lists are read a constant number of times and the total sum of these lists on the path $P_{u,v}$ is $O(n)$.

5.2 Modifying T to reflect the addition of uv and $R(H, u, v)$ to H

Let us now discuss how to the clique tree T of H is built and updated as we decide to add edges and vertices to H. When a new vertex u is added to the set U, then H gets a new vertex, and we update T by adding a new maximal clique containing u.

Let H' denote the graph that results from adding uv and $R(H, u, v)$ to H. We will modify T to obtain a clique tree T' of H'. If u and v are not contained in the same connected component of H and T, then we update T in the following way. Find a tree node K_v of T containing v, and a tree node K_u of T containing u. If $|K_v| > 1$ and $|K_u| > 1$ then we create a new tree node K_{uv} containing the vertices $\{u, v\}$, and insert the tree edges K_vK_{uv} and K_uK_{uv}. The add and remove lists for K_vK_{uv} and $K_{uv}K_u$ can be computed straightforwardly in $O(n)$ time. If $|K_v| = 1$ or $|K_u| = 1$, let us say $|K_u| = 1$, then K_u has no neighbor in T. The new tree T' is created by adding vertex v to K_u to obtain tree node K_{uv} and either deleting K_v (if $|K_v| = 1$) or inserting the tree edge K_vK_{uv} (otherwise). Adding vertex v to K_u (resp. deleting K_v) is an $O(n)$ operation since K_u (resp. K_v) has no neighbors, and inserting the tree edge K_vK_{uv} takes $O(n)$ time.

Let us assume that u and v are contained in the same connected component of T and H for the rest of this subsection. In order to update T to reflect that u has now become a neighbor of v and of every vertex in X, we simply place u in every tree edge and every tree node appearing on $P_{u,v}$ in T. This is illustrated in Figure 2(e). However, we must check the resulting tree T' after doing so, because there might be a tree node C on this path containing a vertex q not appearing in any minimal u, v-separator, and in this case u was not supposed to be a neighbor of q. Detecting such a tree node C is easy because then q cannot appear in any other tree node of the path, since X is already computed and $q \notin X$. For any such C, we remove u from C, and we introduce a new tree node C' that contains u and every vertex of C except the vertices that do not appear in any other tree node of $P_{u,v}$. Tree edges incident to C on $P_{u,v}$ are redirected to be incident to C' instead, and tree edge $C'C$ is added to give a clique tree T' that reflects the neighborhood relations of H' correctly. This is illustrated in Figure 2(f), where C_4 corresponds to the mentioned C. If C_u has become a subset of another maximal clique because of this operation, then we must correct T' accordingly. This is shown in Figure 2(g).

Let us now discuss the practical implementation of this $O(n)$ time. First remove the vertex u from the $remove(C_1, C_2)$ list. This ensures that u belongs to every maximal clique (tree node) on $P_{u,v}$. Let us now consider each maximal clique C_i, $2 \le i \le k$, in the order given by $P_{u,v}$. The first step is to decide if C_i contains any vertex q as described above. We know that no such vertex q appears in a tree edge of $P_{u,v}$, and that X is the union of the tree edges in $P_{u,v}$. Thus, $Q = add(C_{i-1}, C_i) \setminus (X \cup \{v\})$ is exactly the set of such vertices q that are only contained in C_i. If $Q = \emptyset$, then we add u to C_i. This is done by adding u to every $add(C_l, C_i)$ list, where $l \notin \{i-1, i+1\}$ and C_l is a neighbor of C_i outside of $P_{u,v}$. The value of i can now be incremented, such that the process can continue from the next tree node. In the case where $Q \neq \emptyset$, we have to create a new tree node $C_{i'} = C_i \cap (X \cup \{v\}) \cup \{u\}$, and a new tree edge $S_{i',i}$ between $C_{i'}$ and C_i. This is done by simply creating the new lists $add(C_i, C_{i'})$ and $remove(C_i, C_{i'})$ as follows: $add(C_i, C_{i'}) = \{u\}$, since $C_{i'} \setminus C_i = \{u\}$, and $remove(C_i, C_{i'}) = Q$. The lists $add(C_{i-1}, C_{i'})$, $remove(C_{i-1}, C_{i'})$, $add(C_{i'}, C_{i+1})$ and $remove(C_{i'}, C_{i+1})$ are not created, but obtained by altering $add(C_{i-1}, C_i)$, $remove(C_{i-1}, C_i)$, $add(C_i, C_{i+1})$ and $remove(C_i, C_{i+1})$. This is done by moving the pointers from C_i to $C_{i'}$, and removing all the vertices in Q from these lists.

Let us show that the $O(n)$ time bound is kept during the modifications explained above. The vertex set Q is computed by storing X in a characteristic vector A of size n, such that $A[u] = 1$ if and only if $u \in X$. A vertex u is contained in Q if $u \in add(C_{i-1}, C_i)$ and $A[u] = 0$, thus it follows that Q can be computed in $O(|add(C_{i-1}, C_i)|)$ time.

Creating each new tree node $C_{i'}$ is a constant time operation. Every time a new $C_{i'}$ is created, we also create a new tree edge $S_{i,i'}$. We first argue that the sum of the sizes of all the $add(C_i, C_{i'})$ and $remove(C_i, C_{i'})$ lists for all such new tree edges is $O(n)$. This immediately follows from the fact that $Q \subseteq add(C_{i-1}, C_i)$, $add(C_{i-1}, C_i) \cap add(C_{j-1}, C_j) = \emptyset$ for $1 \leq i, j < k$ $i \neq j$, and $\sum_{1 \leq i < k} |add(C_{i-1}, C_i)| \leq n$. Thus, the total cost of creating all such new tree edges $S_{i,i'}$ is $O(n)$. In order to move the tree edges $S_{i-1,i}$ and $S_{i,i+1}$ to $S_{i-1,i'}$ and $S_{i',i+1}$ we must change some pointers, and read through the lists to remove vertices in $C_i \setminus C_{i'} = Q$. The total cost of all such operations is less or equal to the sum of all *add* and *remove* lists in $P_{u,v}$, given that Q also is stored in a characteristic vector. It follows that this altogether is an $O(n)$ time operation.

We will now, through the next three claims, prove that tree T' that results from the modifications explained above is a clique tree of $H' = (U, D \cup \{uv\} \cup R(H, u, v))$.

Claim 5.6 *Given a chordal graph $H = (V, D)$, a clique tree T of H, an edge uv, and the required set of edges $R(H, u, v)$, let H' be the graph $(V, D \cup \{uv\} \cup R(H, u, v))$ and let T' be the resulting clique tree after updating T as explained above. Then for each pair of vertices x and y, there is a tree node in T' that contains both x and y if and only if $xy \in D \cup \{uv\} \cup R(H, u, v)$.*

Proof. Before any modifications to T at this step, there is a tree node $C \in T$, that contains both the vertices x and y if and only if $xy \in D$. A tree node C_d is only deleted during the modification process if there exists a remaining tree node C'_d, such that $C_d \subseteq C'_d$. Thus, for every edge $xy \in D$ there exists a tree node in T' that contains both x and y.

Before appropriate tree nodes of T are expanded to contain u, every newly created tree node $C_{i'}$ is a subset of some other tree node C_i. At this point we have the property that the vertex set of every tree node of T is either a maximal clique in H or a subset of a maximal clique in H. Thus T has still the property that there exists a tree node containing x and y if and only if $xy \in D$.

Then u is added to every tree node C of T on the modified path $P_{u,v}$, where $C \subseteq X \cup \{v\}$, and we obtain T'. It follows that for every edge $xy \notin E(H')$, there is no tree node C of T' that contains both x and y, since u is only added to a tree node C if $C \subseteq X \cup \{v\}$.

For the other direction we have to show that for every edge $ux \in R(H, u, v) \cup \{uv\}$ there exists a tree node C in T' containing u and x. By Claim 5.1 every minimal u, v-separator is an edge of $P_{u,v}$, thus there exists a tree node C in T on $P_{u,v}$ containing x for every $ux \in R(H, u, v)$. The tree node C_v in the end of $P_{u,v}$ contains v. If $C \not\subseteq X \cup \{v\}$ for a tree node C in T on $P_{u,v}$, then a new tree node $C' = C \cap (X \cup \{v\})$ is created and used in the path $P_{u,v}$. We can now conclude that for every edge $ux \in R(H, u, v) \cup \{uv\}$ there exists a tree node of T' which contains both u and x. ■

Claim 5.7 *Subtree T'_x induced by the tree nodes in T' that contain vertex x is connected, for every vertex $x \in U$.*

Proof. We assume that all subtrees are connected in T before the last modification. Let us now consider the operations one by one. First operation is when $C_i \notin (X \cup \{v\})$. A new tree node $C_{i'}$ is created, where $C_{i'} \subset C_i$. A tree edge is inserted between C_i and $C_{i'}$, but all subtrees are connected since $C_{i'} \subset C_i$. Next step is to move the edges $S_{i-1,i}$ and $S_{i,i+1}$ to become $S_{i-1,i'}$ and $S_{i',i+1}$. This will not create separated subtrees since $S_{i-1,i} \cup S_{i,i+1} \subseteq C_{i'}$, thus $S_{i-1,i'} = S_{i-1,i}$ and $S_{i',i+1} = S_{i,i+1}$. The second operation is adding the vertex u to $C_{i'}$ in the case where a new tree node $C_{i'}$ is created, and to C_i if no new tree node is created. This changes only the tree induced by the tree nodes containing the vertex u. Since we consider the tree nodes in the order C_2 to C_v, then it follows that the tree T_u is always connected. ■

Claim 5.8 *The tree nodes of T' are exactly the distinct maximal cliques of H', except for C_u in case $C_u \subseteq C_2$.*

Proof. Let us first show that every maximal clique of H' is a tree node of T'. By Claims 5.6 and 5.7, T' defines what is called a tree decomposition of H'. So, by [13] every clique in H' is contained in some node of T'. Since by Claim 5.6 every tree node of T' is a clique in H', a maximal clique in H' cannot be strictly contained in some tree node of T' and therefore is equal to one of them.
Conversely, let us show that tree nodes of T' are distinct maximal cliques of H', except for C_u in case $C_u \subseteq C_2$. Suppose on the contrary that some tree node C of T' is not a maximal clique of H' or is a maximal clique of H' equal to another node of T'. Since every maximal clique of H' is a tree node of T' and since by Claim 5.6, C is a clique of H', there is a tree node C' different from C containing C. Let C'' be the neighbor of C on the path in T' between C and C'. By Claim 5.7, $C = C \cap C' \subseteq C''$. It follows that it is sufficient to show that no tree node of T' is a subset of one of its neighbors, except for C_u in case $C_u \subseteq C_2$.

We will now prove by induction that $C \nsubseteq C''$ unless $C = C_u$ and $C'' = C_2$, where C and C'' are tree nodes of T'. This is clearly true in the base case, where H' consist of only one vertex and T' consist of a single tree node. Let $H = (V, D)$ be the chordal graph such that $H' = (V, D \cup \{uv\} \cup R(H, u, v))$ and let T be the given maximal clique tree of H. There are two cases. The first is when the vertex u is added to a tree node C. The expanded C cannot become a subset of another tree node, but it can become a superset of a tree node C_j, if $C_j \setminus C = \{u\}$. Since C_u is the only tree node in the neighborhood of any tree node different from C_u in $P_{u,v}$ that contains u, then this can only happen to C_u. The second case is when a new tree node $C_{i'}$ is created as a subset of C_i, where $u \notin C_i$ and $C_i \setminus C_{i'} \neq \emptyset$. In this situation $C_{i'}$ is a tree node on the path $P_{u,v}$ and C_i is not, u is added to $C_{i'}$ and not to C_i, thus C_i and $C_{i'}$ are not subsets of each other. From the construction of $C_{i'}$ we know that every neighbor of $C_{i'}$ different from C_i is on the path $P_{u,v}$. Let us now on the contrary assume that $C_{i'} \subseteq C_{i-1}$ or $C_{i'} \subseteq C_{i+1}$. If $C_{i'} \subseteq C_{i+1}$ then $S_{i-1,i'} \subseteq C_{i'} = S_{i',i+1}$, which is a contradiction to the fact that every edge of $P_{u,v}$ is a unique minimal u, v-separator. If $C_{i'} \subseteq C_{i-1}$ then $i = k$ since otherwise $S_{i',i+1} \subseteq C_{i'} = S_{i-1,i'}$ which is a contradiction to the fact that every edge of $P_{u,v}$ is a unique minimal u, v-separator. The only remaining case is that $C_{i'} = C_v$ on the path $P_{u,v}$ with $C_{i'} \subseteq C_{i-1}$. Then $v \in C_{i'}$, so $v \in C_{i-1} \neq C_v$ which is a contradiction to the fact that only C_v in $P_{u,v}$ contains v. ∎

Let us re-sort the tree nodes of the modified path $P_{u,v}$ of T' from $C_u = C_1$ to $C_k = C_v$. With the above three claims, if $C_u \nsubseteq C_2$, then we have proved that T' is a legal clique tree of H'. If $C_u \subseteq C_2$, then we will simply remove C_u, and again we can conclude that T' with this final modification is a legal clique tree of H'.

However, it remains to explain how this final update of removing $C_u = C_1$ can be done in $O(n)$ time, which is challenging. It is easy to check if $C_1 \subseteq C_2$, since $\mathit{remove}(C_1, C_2) = \emptyset$ in this case. Tree node C_1 is deleted in the following way: For every tree edge $S_{1,j}$ where $C_j \neq C_2$, we delete the tree edge $S_{1,j}$ and insert $S_{2,j}$. Afterwards we delete tree edge $S_{1,2}$ and tree node C_1. In order to do this efficiently we actually alter the *add* and *remove* lists and move the tree edges from C_1 to C_2. Let us consider the new tree node C_j, and how to create the *add* and *remove* lists from C_j to C_2. From the previous described technique they can be computed as follows: $\mathit{add}(C_j, C_2) = \mathit{add}(C_j, C_1) \cup \mathit{add}(C_1, C_2) \setminus \mathit{remove}(C_1, C_2)$ and $\mathit{remove}(C_j, C_2) = \mathit{remove}(C_1, C_2) \cup \mathit{remove}(C_j, C_1) \setminus \mathit{add}(C_j, C_1)$. Remember that $\mathit{remove}(C_1, C_2) = \emptyset$, since $C_1 \subseteq C_2$, and that $\mathit{remove}(C_j, C_1) \cap \mathit{add}(C_j, C_1) = \emptyset$. Computing the lists can then be reduced to: $\mathit{add}(C_j, C_2) = \mathit{add}(C_j, C_1) \cup \mathit{add}(C_1, C_2)$ and $\mathit{remove}(C_j, C_2) = \mathit{remove}(C_j, C_1)$. The obstacle regarding the time complexity is that $\mathit{add}(C_1, C_2)$ will be read once for each neighbor of C_1. Thus, we have to ensure that this work does not sum up to more than $O(n)$. Let us count the number of times this can happen.

Claim 5.9 *Let $C_u \subseteq C_2$ in T', and let H'' be the chordal graph right before the first edge uv' incident to u and the set $R(H'', u, v')$ was added to H'', and let T'' be the clique tree of H''. Then $C_u \setminus \{u\}$ is not a tree node of T'' or in other words $C_u \setminus \{u\}$ is not a maximal clique of H''.*

Proof. The tree T' is obtained from T'' by adding new tree nodes which are subsets of tree nodes in T'' and by adding u to tree nodes of this new tree, since only edges incident to u are processed between T'' and T'. Clearly $C_u \setminus \{u\}$ is not a tree node in T'', since $C_u \setminus \{u\} \subseteq C_2 \setminus \{u\}$ and $C_2 \setminus \{u\}$ is contained in some tree node of T'', and every tree node of T'' is a unique maximal clique in H''. ■

Claim 5.10 *Reducing path $P_{u,v}$ such that it contains only distinct minimal u, v-separators, can increase the degree of C_u by at most 1.*

Proof. Two different scans are done on $P_{u,v}$ to reduce the number of tree nodes. The first starts in C_u, and finds the maximal clique furthest from C_u that is a superset of $S_{u,2}$. In this case one tree edge incident to C_u is deleted, and one is created, and the degree of C_u remains the same. In the direction from C_v to C_u, we may find a tree edge that is a subset of $S_{u,2}$. In this case C_u gets a new neighbor and the degree of C_u increases by 1. ■

Observe that the process of reducing the path $P_{u,v}$ can increase the degree of at most one tree node containing the vertex u. This follows from the fact that C_u is the only tree node in $P_{u,v}$ containing the vertex u.

Claim 5.11 *Adding u to every tree node in $P_{u,v}$ does not increase the degree of C_u.*

Proof. One of two things will happen. In one case vertex u is added to C_2, which is the neighbor of C_u in the path $P_{u,v}$. This will not change the degree of C_u in the clique tree T. The second case is if C_2 is not a subset of $X \cup \{v\}$. Then a new tree node C_2' is created, and the tree edge between C_u and C_2 is removed, and inserted between C_u and C_2'. It follows that the degree of C_u is unchanged. ■

Claim 5.12 *The degree of each newly created tree node $C_{i'}$ in T' is at most* 3.

Proof. When a new tree node $C_{i'}$ is created, it is a subset of an existing tree node C_i. Let d be the number of neighbors C_i has in $P_{u,v}$. Thus, d is either 1 or 2. A tree edge is introduced between $C_{i'}$ and C_i, and $C_{i'}$ replaces C_i in the path $P_{u,v}$. The degree of $C_{i'}$ becomes $d + 1$, and thus the degree is at most 3. ■

Remember that the obstacle in obtaining the $O(n)$ time bound was that $add(C_u, C_2)$ is read once for each neighbor of C_u, when $add(C_j, C_2) = add(C_j, C_u) \cup add(C_u, C_2)$ is computed. It remains to show that if $C_u \subseteq C_2$ in T' then $\sum_{C_jC_u \textit{ edge of } T',\ j\neq 2} |add(C_u, C_2)|$ is $O(n)$. We will use an amortized time analysis. Let vertex u be given, and let $d(u)$ denote the degree of u in G. For any neighbor v of u in G such that edge uv is processed in the execution of the algorithm, let $T'(v)$ denote the tree T' when processing edge uv, let $C_u(v)$ denote tree node C_u of this tree, and let $C_j(v)$ denote tree node C_j of $T'(v)$. Let V_1 be the set of neighbors v of u in G such that edge uv is processed and $C_u(v) \subseteq C_2(v)$. Let $S = \sum_{v\in V_1} \sum_{C_j(v)C_u(v) \textit{ edge of } T'(v)} |add(C_u(v), C_2(v))|$. It is sufficient to show that S is $O(n \cdot d(u))$. For any $v \in V_1$, let $E(v)$ be the set of edges incident to $C_u(v)$ in $T'(v)$, and let $E_1(v)$, $E_2(v)$, $E_3(v)$, and $E_4(v)$ be the following subsets of $E(v)$:

- $E_1(v)$ is the set of edges of $E(v)$ created at the same time as $C_u(v)$;
- $E_2(v)$ is the set of edges of $E(v)$ inserted when reducing path $P_{u,v'}$, with $C_u(v') = C_u(v)$, for some edge uv' processed before uv;
- $E_3(v)$ is the set of edges of $E(v)$ inserted when suppressing $C_u(v')$ because $C_u(v') \subseteq C_2(v')$, with $C_2(v') = C_u(v)$, for some edge uv' processed before uv;
- $E_4(v)$ is the set of edges of $E(v)$ inserted as an edge $K_uK_{uv'}$ or $K_uK_{v'}$ with $K_u = C_u(v)$, when previously processing an edge uv' such that u and v' are in different connected components of the current graph.

It follows from Claim 5.9 that $C_u(v)$ is created when inserting some edge uv' previous to uv. The set $E_2(v)$, contains the edge incident to $C_u(v')$ that may be inserted when reducing path $P_{u,v'}$ during the scan from $C_{v'}(v')$ to $C_u(v')$, but not the edge that may be inserted during the scan from $C_u(v')$ to $C_{v'}(v')$ in replacement of edge $C_u(v')C_2(v')$: these two edges (the replaced and replacing ones) are identified in our counting process. In the same way, the edge of $E(v)$ that may be inserted in replacement of $C_u(v')C_2(v')$ when adding u to $C_2(v')$ is identified with the replaced edge. So by Claim 5.11, no edge of $E(v)$ has been inserted when adding u to $C_2(v')$ for any edge uv' processed before uv, and every edge of $E(v)$ belongs to one of the sets $E_1(v)$, $E_2(v)$, $E_3(v)$, and $E_4(v)$.

Let $e = C_j(v)C_u(v)$ be an edge of $E_3(v)$, and let v' be the previously processed vertex such that e was inserted when suppressing $C_u(v')$ because $C_u(v') \subseteq C_2(v')$, with $C_2(v') = C_u(v)$. Then $v' \in V_1$ and e is obtained from some edge $e' = C_{j'}(v')C_u(v')$ belonging to $E_1(v') \cup E_2(v') \cup E_3(v') \cup E_4(v')$. We say that e *derives from* e'. If $e' \in E_3(v')$ then e' derives from some edge e''. It follows that there are sequences $(v_0, v_1, ..., v_p = v)$ of vertices of V_1 and $(e_0, e_1, ..., e_p = e)$ of edges such that $e_0 \in E_1(v_0) \cup E_2(v_0) \cup E_4(v_0)$ and for any i from 1 to p, $e_i \in E_3(v_i)$ and e_i derives from e_{i-1}. Conversely, for any $v \in V_1$ and $e \in E_1(v) \cup E_2(v) \cup E_4(v)$, there are unique such sequences $seq(e) = (v = v_0, v_1, ..., v_p)$ and $(e = e_0, e_1, ..., e_p)$ such that no edge derives from e_p. So S can be rewritten as follows: $S = \sum_{v \in V_1} \sum_{e \in E_1(v) \cup E_2(v) \cup E_4(v)} \sum_{v' \in seq(e)} |add(C_u(v'), C_2(v'))|$.

If $seq(e) = (v = v_0, v_1, ..., v_p)$, then since $C_u(v_i) \subseteq C_2(v_i)$ for any i from 0 to p and $C_2(v_i) = C_u(v_{i+1})$ for any i from 0 to $p-1$, we have that $\sum_{v' \in seq(e)} |add(C_u(v'), C_2(v'))| = \sum_{0 \leq i < p}(|C_u(v_{i+1})| - |C_u(v_i)|) + (|C_2(v_p)| - |C_u(v_p)|) = |C_2(v_p)| - |C_u(v_0)| \leq n$.

Moreover, for any $v \in V_1$, if $C_u(v)$ was created as a node $\{u, v'\}$ (when processing an edge uv' such that u and v' are in different connected components of the current graph) then $|E_1(v)| \leq 2$, and otherwise, by Claims 5.9 and 5.12, $|E_1(v)| \leq 3$, and by Claim 5.10, $\sum_{v \in V_1} |E_2(v)| \leq d(u)$, and finally $\sum_{v \in V_1} |E_4(v)| \leq d(u)$ since only one tree edge is added for each edge uv' such that u and v' are contained in different connected components of T'. Hence $S \leq n(\sum_{v \in V_1}(|E_1(v)| + |E_2(v)| + |E_4(v)|) \leq n(3|V_1| + 2d(u)) \leq n \cdot 5d(u) = O(n \cdot d(u))$.

6 Concluding remarks

In this paper, we contribute new theoretical results on chordality as well as an efficient handling of the corresponding data structures. Not only do we have a new $O(nm)$ time on-line algorithm for minimal triangulation of a graph G, but we are able to compute at the same time a maximal chordal subgraph, thus "minimally sandwiching" the graph between two chordal graphs: $H_1 \subseteq G \subseteq H_2$.

This special feature of our algorithm enables the user, at no extra cost, to choose at each vertex addition step whether he wants to *add* or *delete* edges, or even to do so at each edge addition step. This may be interesting for applications such as updating databases or for sampling techniques in the context of artificial intelligence when maintaining a chordal graph is required or desirable.

Recent work has shown that minimal separation plays an important role in the process of minimal triangulation. Our new characterization of chordal graphs, which uses minimal separation, leads us to believe that there is a corresponding relationship when computing a maximal chordal subgraph.

A continuation of this work would be to compare the running time of our algorithm to other minimal triangulation algorithms with experimental tests. Since often several edges are found and inserted at the time cost of one edge, we conjecture that our algorithm may be very fast in practice. Another important issue to inquire about would be how well our algorithm performs when used as a heuristic for hard problems, such as computing a minimum triangulation or a maximum subtriangulation. Standard ideas from existing heuristics, like picking a vertex of minimum degree at each step, could be integrated into our algorithm and possibly result in higher probability of less fill in minimal triangulations and more edges in maximal subtriangulations.

It appears that chordal graphs are in many ways similar to weakly chordal graphs [23, 7, 6]. It would be interesting to extend our results to define a process which maintains a weakly chordal graph, thus enabling efficient computation of a weak minimal super or maximal sub triangulation, which is an important issue for recent applications to formal concept analysis and data mining [9]. As we pointed out in Section 3, the required set of edges can be seen as a succession of 2-pairs which is computed efficiently. In view of the important role that 2-pairs play in weakly chordal graph recognition [24, 33, 25], our results could possibly be extended to efficiently handle such a succession of 2-pairs in a weakly chordal graph, with the hope of improving the current $O(m^2)$ [25] time complexity for this problem.

Acknowledgments

The authors would like to thank the referees for useful comments, which have improved the readability of the text in general, and of some of the proofs in particular.

References

[1] S. Arikati and P. Rangan. An efficient algorithm for finding a two-pair, and its applications. *Disc. Appl. Math.*, 31:71–74, 1991.

[2] E. Balas. A fast algorithm for finding an edge-maximal subgraph with a TR-formative coloring. *Disc. Appl. Math.*, 15:123–134, 1986.

[3] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database systems. *J. Assoc. Comput. Mach.*, 30:479–513, 1983.

[4] A. Berry. A wide-range efficient algorithm for minimal triangulation. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[5] A. Berry, J. Blair, P. Heggernes, and B Peyton. Maximum cardinality search for computing minimal triangulations of graphs. *Algorithmica*, 39(4):287–298, 2004.

[6] A. Berry and J-P. Bordat. Triangulated and weakly triangulated graphs: Simpliciality in vertices and edges. *6th International Conference on Graph Theory (ICGT 2000)*, 2000. Communication.

[7] A. Berry, J-P. Bordat, and P. Heggernes. Recognizing weakly triangulated graphs by edge separability. *Nordic Journal of Computing*, 7:164–177, 2000.

[8] A. Berry, J-P. Bordat, P. Heggernes, G. Simonet, and Y. Villanger. A wide-range algorithm for minimal triangulation from an arbitrary ordering. Technical Report Reports in Informatics 243, University of Bergen, Norway, 2003. Submitted to *Journal of Algorithms*.

[9] A. Berry and A. Sigayret. Obtaining and maintaining polynomial-size concept lattices. In *Proceedings of FCAKDD, (ECAI 2002)*, pages 3–6, 2002.

[10] A. Berry, A. Sigayret, and C. Sinoquet. Maximal sub-triangulation as improving phylogenetic data. Technical Report RR-02-02, LIMOS, Clermont-Ferrand, France, 2002.

[11] J. R. S. Blair, P. Heggernes, and J. A. Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250:125–141, 2001.

[12] J. R. S. Blair and B. W. Peyton. An introduction to chordal graphs and clique trees. In J. A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 1–30. Springer Verlag, 1993. IMA Volumes in Mathematics and its Applications, Vol. 56.

[13] Hans L. Bodlaender and Rolf H. Möhring. The pathwidth and treewidth of cographs. *SIAM J. Discrete Math.*, 6(2):181–188, 1993.

[14] P. Buneman. A characterization of rigid circuit graphs. *Discrete Math.*, 9:205–212, 1974.

[15] T. F. Coleman. A chordal preconditioner for large-scale optimization. *Applied Math.*, 40:265–287, 1988.

[16] E. Dahlhaus. Minimal elimination ordering inside a given chordal graph. In R. H. Möhring, editor, *Graph Theoretical Concepts in Computer Science - WG '97, LNCS 1335*, pages 132–143. Springer Verlag, 1997.

[17] P. M. Dearing, D. R. Shier, and D. D. Warner. Maximal chordal subgraphs. *Disc. Appl. Math.*, 20:181–190, 1988.

[18] A. Deshpande, M. Garofalakis, and M. I. Jordan. Efficient stepwise selection in decomposable models. In *Proceedings of UAI*, pages 128–135, 2001.

[19] G. A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[20] P. Erdös and R. Laskar. On maximum chordal subgraph. *Cong. Numerantium*, 39:367–373, 1983.

[21] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15:835–855, 1965.

[22] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combin. Theory Ser. B*, 16:47–56, 1974.

[23] R. Hayward. Generating weakly triangulated graphs. *J. Graph Theory*, 21:67–70, 1996.

[24] R. Hayward, C. Hoàng, and F. Maffray. Optimizing weakly triangulated graphs. *Graphs and Combinatorics*, 5:339–349, 1989.

[25] R. Hayward, J. Spinrad, and R. Sritharan. Weakly chordal graph algorithms via handles. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[26] P. Heggernes and Y. Villanger. Efficient implementation of a minimal triangulation algorithm. In R. H. Möhring, editor, *Algorithms - ESA 2002, LNCS 2461*, pages 550–561. Springer Verlag, 2002.

[27] C-W. Ho and R. C. T. Lee. Counting clique trees and computing perfect elimination schemes in parallel. *Inform. Process. Lett.*, 31:61–68, 1989.

[28] L. Ibarra. Fully dynamic algorithms for chordal graphs. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[29] M. Lundquist. *Zero patterns chordal graphs and matrix completions.* PhD thesis, Clemson University, USA, 1990.

[30] A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Disc. Appl. Math.*, 113:109–128, 2001.

[31] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, 1972.

[32] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

[33] J. Spinrad and R. Sritharan. Algorithms for weakly triangulated graphs. *Disc. Appl. Math.*, 59:181–191, 1995.

[34] J. Walter. *Representations of rigid cycle graphs.* PhD thesis, Wayne State University, USA, 1972.

[35] J. Xue. Edge-maximal triangulated subgraphs and heuristics for the maximum clique problem. *Networks*, 24:109–120, 1994.

[36] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.