Heuristic for the pathwidth on Chordal Graphs

Juan Antonio Doldan Bi Li Fatima Zahra Moataz Nicolas Nisse

August 29, 2014

1 Introduction

A *tree-decomposition* of a graph G [19] is a way to represent G by a family of subsets of its vertex-set organized in a tree-like manner and satisfying some connectivity property. The *treewidth* of G measures the proximity of G to a tree. More formally, a tree decomposition of $G = (V, E)$ is a pair $(T, \mathcal{X})$ where $\mathcal{X} = \{X_t | t \in V(T)\}$ is a family of subsets, called *bags*, of V, and T is a tree, such that:

- $\bigcup_{t \in V(T)} X_t = V$;
- for any edge $uv \in E$, there is a bag X_t (for some node $t \in V(T)$) containing both u and v;
- for any vertex $v \in V$, the set $\{t \in V(T) | v \in X_t\}$ induces a subtree of T.

The *width* of a tree-decomposition $(T, \mathcal{X})$ is $max_{t \in V(T)} |X_t| - 1$ and its *size* is order $|V(T)|$ of T. The treewidth of G, denoted by $tw(G)$, is the minimum width over all possible tree-decompositions of G.

If T is constrained to be a path, $(T, \mathcal{X})$ is called a *path-decomposition* of G. The pathwidth of G, denoted by $pw(G)$, is the minimum width over all possible path-decompositions of G.

Tree/path-decompositions are a fundamental algorithmic tool used by many dynamic programming algorithms on graphs (See [6]). It has been shown that difficult problems, such as some NP-complete ones, can be solved efficiently over classes of graphs of bounded treewidth (See [11, 4, 2] among others).

Assuming $P \neq NP$, many natural problems require superpolynomial running time when complexity is measured in terms of the input size, but can be computable in a time that is polynomial in the input size and exponential or worse in a parameter k. A parameterized problem is called *fixed parameter tractable* if it admits a solving algorithm whose running time on input instance (I, k) is $f(k) \times |I|^{\alpha}$, where f is an arbitrary function depending only on k [11]. This way of measuring the complexity becomes an important tool to consideraly improve the complexity bound of problems that are traditionally considered *intractable*, thus transforming them into *tractable* ones. For instance, the dominating set problem of a graph G can be solved with running time $O(4^k n)$, where n is the number of nodes of a tree decomposition of G and k is the treewidth. [1].

Though both the problem of calculating the treewidth and the pathwidth are fixed parameter tractable in the general case [5], their complexities have such a bad bound that it rapidly becomes impractical (or even impossible) to calculate the exact tree/pathwidth. For some classes of graphs, polynomial algorithms can be found for each of the problems [7, 14], while for other classes good approximation algorithms have been developed [9]. Knowing that there are good results on determined classes is a good motivation to look forward and try to get new approximation algorithms on other families of graphs.

A chordal graph is a finite undirected graph in which any induced cycle has three nodes.

The pathwidth problem for chordal graphs is NP-hard [14], but tree decompositions are obtained polynomially on them. Gustedt also shows that the pathwidth of a $k-$starlike[1] graph can be calculated in $O(|V(G)|^{2k+1})$ time and space [14].

[1] $k-$starlike graphs are starlike graphs where the size of each clique minus the central clique is bounded by a constant k.

This impulsed us to try to obtain good path decompositions in the general case of chordal graphs by creating an approximation algorithm.

In this work, we propose an heuristic to obtain the pathwidth of chordal graphs and show the problems we encountered at the moment of bounding the approximation factor of the algorithm.

Related works

There are some techniques that help us solve the pathwidth problem, with a certain degree of certainty. For example, some classes of graph searchings can return us an approximation of twice the pathwidth. But which classes? And moreover, what are graph searchings?

In node graph searching, some searchers try to capture an invisible fugitive in a connected graph G [16]. The searchers and the fugitive occupy the vertices in G. The fugitive is arbitrarily fast and can go through the paths of G as long as it does not meet a searcher. He is captured at vertex v if at least one searcher occupies v at the same time with him and he cannot escape, i.e., all neighbours of v are also occupied by some searchers. Searchers can be placed at or removed from the vertices of G. A strategy for the searchers is a sequence of steps (placement or removal) that results in capturing the fugitive whatever he does.

The number of searchers used by a strategy is the maximum number of searchers simultaneously occupying vertices of the graph. It is interesting to find a strategy with the minimum number of searchers to capture a fugitive in a given graph. For any graph, there is a *monotone* strategy, i.e. each vertex is occupied at most once by a searcher, capturing the fugitive using minimum number of searchers [3, 17, 20]. This shows that the problems of computing the number of searchers are in NP.

It is shown that vertex separation is identical to pathwidth [15]. While vertex separation is equal to node search number of the graph minus one [16]. The edge search number is between the vertex separation and vertex separation plus 2 [12].

A search strategy $\mathcal{S}$ is *connected* if, at any time of $\mathcal{S}$, the vertices into which the fugitive cannot be form a connected component in the graph.

It is proven that search number equals the monotone search number for any graph G in [3, 17, 20]. Yang et al. constructed a graph G such that the connected search uses less searchers than the monotone connected search in [21]. It is still unknown whether the problem of computing the connected search number is in NP. However, it is proven in [10] that the monotone connected search number is bounded by twice the search number plus one; which is equal to twice the pathwidth plus three.

This means that if we manage to find the monotone connected search number, we almost have a $2-$approximation of the pathwidth.

The pathwidth problem is $NP-$hard for weighted trees [18]. For a weighted tree, Dereniowski gives a generic algorithm for finding an optimal connected search strategy and a polynomial time $3-$approximation algorithm [9].

In the next pages we will show our attempts to construct an approximation algorithm to get the pathwidth of a chordal graph by modifying Dereniowski's algorithm. We will also show which were the problems we faced, how we solved some of them and our intuition on how to solve the ones that are left.

2 Applying Dereniowski's Algorithm to a Chordal Graph

In this section we will firstly explain the basics of Dereniowski's algorithm (as seen in [9]).

Afterwards, we will give the pseudocode to our modification of Dereniowski's algorithm with some notes on how it works.

Since the algorithm works with weigthed trees, we need to get one weighted tree out of a chordal graph. We show an heuristic to get weighted tree decompositions from chordal graphs followed by the problems that arise in the approximation factor.

Lastly, we explain the computational complexity of all the process.

2.1 Notation and definitions

Given a graph $G = (V, E)$, a *(node) search strategy* $\mathcal{S}$ for G is a sequence $\mathcal{S}_1, \ldots, \mathcal{S}_l$ of sets of nodes of $V(G)$. $\mathcal{S}$ is a k-search strategy for G for a given integer k if $\mathcal{S}$ is a search strategy for G and uses at most k searchers; i.e., none of the sets of $\mathcal{S}$ has more than k elements.

Let $|\mathcal{S}|$ be the number of moves (sets) in $\mathcal{S}$. For each set S_t, $t \in 1 \ldots |\mathcal{S}|$, each node $v \in S_t$ represents that there is a searcher in v at step t. Denote by $s(\mathcal{S}_t)$, $t \in \{1, \ldots, |\mathcal{S}|\}$, the number of searchers used in step $\mathcal{S}_t$. Denote $s(\mathcal{S})$ the maximum number of searchers used by a step of strategy $\mathcal{S}$.

A vertex $v \in V$ is cleared by a strategy $\mathcal{S}$ if the fugitive cannot go to v at the end of $\mathcal{S}$; an edge uv is cleared if both u and v are cleared. A vertex $v \in V$ is *guarded* at $\mathcal{S}_t$ if $v \in \mathcal{S}_t$.

A search strategy $\mathcal{S}$ is partial if a subset of the vertices is cleared at the end of $\mathcal{S}$. We say that $\mathcal{S}$ clears a subgraph G' of G if all vertices in G' are cleared at the end of $\mathcal{S}$. Let $(\{X_i | i \in I\}, T = (I, F))$ be the tree decomposition of G. For brevity we say that $\mathcal{S}$ clears a node $i \in I$ if all vertices in the bag X_i are cleared at the end of $\mathcal{S}$.

Given a rooted tree T and a node $x \in V(T)$; let T_x be the subtree of T rooted at x. Respectively, V_x is the set of vertices that are direct descendents of x and E_x is the set of edges that connect x with V_x.

Let $C_V(\mathcal{S})$ (resp. $C_I(\mathcal{S})$) be the set of vertices (resp. nodes) cleared by $\mathcal{S}$ in G (resp. T). Define $\mathcal{S}_{\preceq t}$, $t \in \{1, \ldots, |\mathcal{S}|\}$, to be the partial search strategy obtained by performing the first t moves of $\mathcal{S}$.

Let $\delta_V(\mathcal{S})$ (resp. $\delta_I(\mathcal{S})$) be the set of vertices (resp. nodes) guarded at the end of $\mathcal{S}$. If $\mathcal{S}$ clears G, then obviously $\delta_V(\mathcal{S}) = \delta_I(\mathcal{S}) = \emptyset$.

A partial search strategy $\mathcal{S}$ can be *k-extended* to a partial strategy $\mathcal{S}'$ for G if $|\mathcal{S}'| \geq |\mathcal{S}|$, $\mathcal{S}_t = \mathcal{S}'_t$ for each $t = 1, \ldots, |\mathcal{S}|$ and $s(\mathcal{S}') \leq k$. In such case, we also say that $\mathcal{S}'$ is a *k-extension*, or an *extension* for short of $\mathcal{S}$.

2.2 Dereniowski's Original Algorithm

Dereniowski's algorithm is based in the idea that there are some searches that can be considered *greedy*.

We are going to consider *greedy* searches as described by Dereniowski [9]. A partial search $\mathcal{S}$ of T is greedy if:

1. $\mathcal{S}$ places some searchers on the root of the tree and clears at least one edge.
2. $w(u) \geq w(\delta(\mathcal{S}) \bigcap V(T_u))$ for each $u \in C_V(\mathcal{S})$.
3. $E_u \bigcap C_E(\mathcal{S}) = \emptyset$ or $E_u \subseteq C_E(\mathcal{S})$ for each $u \in C_V(\mathcal{S})$.

The algorithm is divided into two procedures.

The main procedure, called *GWTAS* (*Generic Weighted Tree Approximate Searching*), finds a connected search strategy for a given tree T and each possible root. The input also includes a function Γ that assigns a set of permutations to each V_v, $v \in V(T)$. Once the root is decided, the process acts in a bottom up way. For each T_v, such that $v \in V$ and $E_v \neq \emptyset$ the algorithm gets a partial search strategy $\mathcal{S}(T_v)$ by invoking the second subroutine *GSS* (*Generic Subtree Searching*). $\mathcal{S}(T_v)$ is greedy and such that there is not another k-greedy search for $k < s(\mathcal{S}(T_v))$. Once $\mathcal{S}(T_v)$ is performed for each $v \in V(T)$, the algorithm finds a connected search strategy $\mathcal{S}$ for T (based on T's root) iteratively doing performing this step:

If $\mathcal{S}$ is the current partial search strategy, then a vertex v of $\delta(\mathcal{S})$ and the next iteration is $\mathcal{S} \oplus \mathcal{S}(T_v)$. The new strategy is still connected and based in T's root and E_v is cleared according to Γ whenever $V_v \neq \emptyset$.

GSS receives as input a tree T rooted at a vertex v, a permutation π of V_v and a set $\mathcal{A}$ that contanis a partial search strategy $\mathcal{S}(T_u)$ for each $u \in V(T_v) \setminus (\{v\} \cup L_t)$, where L_T are the leaves of T. *GSS* returns a partial search strategy $\mathcal{S}_\pi$ for T starting from v that clears E_v according to π. Moreover, each subtree T_u, $u \in V_v$, is cleared by a concatenation of strategies from $\mathcal{A}$.

In each iteration of the main loop *GSS* tries to compute a partial search strategy $\mathcal{S}_\pi$ for T_v. k_π is the number of searchers available for $\mathcal{S}_\pi$. All k_π searchers are placed at v and the computation starts.

There is an inner loop which is the one responsible for computing $\mathcal{S}_\pi$. If it is possible, in the $i-$th iteration, $w(\pi_i)$ searchers go from v to π_i. If there are not enough searchers, k_π increases accordingly and a new iteration of the main

loop begins. If we were able to put searchers in π_i, the algorithm will check if there is a vertex u in $\delta(\mathcal{S}_\pi)$ such that $\mathcal{S}_\pi \oplus \mathcal{S}(T_u)$. If we find such u, we add $\mathcal{S}(T_u)$ to $\mathcal{S}_\pi$.

Then, the algorithm checks if $w(\delta(\mathcal{S}_\pi)) \leq w(v)$ and if we cleared all the edges in E_v. In the affirmative case, the computation stops. Elsewise, k_π increases and the computation of a new search strategy S_π begins.

It is proven that this algorithm always returns a *greedy* partial search strategy [9].

GWTAS is much simpler, but needs a new definition.

We say that Γ is an ordering of T if, for the set X of neighbours of every vertex that is not a leaf, it assigns a non empty set $\Gamma(X)$ of permutations of X.

Then, *GWTAS* receives as input an unrooted tree T, and an ordering Γ of T. It proceeds to consider every possible root r of T, and then, generates a post-ordering of the nodes of T (i.e. each node precedes its parent). Then, for each node of the order, it calculates every possible search strategy using *GSS* and the ordering $\pi \in \Gamma(V_{v_i}, i \in 1 \dots |V(T)|$. Afterwards, it stores the best $\mathcal{S}(T_{v_i})$ in a set, and uses this information to create the best greedy search strategy S_r that starts from r and clears the entire tree. In the end, it chooses the root that produced the better search (i.e. the one that used less searchers).

2.3 Dereniowski's algorithm modification

Now we will focus on the modification of Dereniowski's algorithm. Having explained it before, we will mostly focus on the details that were changed for it to work with these new data structures.

The original algorithm uses the weighted tree and considers that once all the children of a node are cleared, its weight becomes 0 and stops being taken into account when adding up the number of searchers used. In our modification, nodes will represent maximal cliques, which may have had vertices in common in the original graph. This has to be taken into account. When calculating the number of searches used by a step of the search, we need not only to calculate the weight of those nodes of T that are part of the frontier but also not to count twice the vertices of G that are part of many bags of T that are part of the frontier. That's why one of the slightest changes that we produce in the algorithm is the way the weight of the frontier is calculated. We use a list of pairs of integers in which position i represents the number n_i of maximal cliques in G to which i belongs, and the maximal clique c_i topologically closest to the root in T to which i belongs. i.e. Given different tree decompositions, n_i will never vary, but c_i will do.

In the following, we consider that $G = (V, E)$ is a chordal graph and $(\{X_i | i \in I\}, T = (I, F))$ is a tree decomposition of G, but to simplify the notation, we will refer to the tree decomposition as T. We use the same notation (and simplification) for any graph decomposition $(\{X_i | i \in I\}, H = (I, F))$ of G. Unless specifically stated, we are going to work with *weighted labelled maximal cliques tree decompositions*; which are tree decompositions in which each node corresponds to a maximal clique of G, has a distinctive name and there is a function w, such that $w(i), i \in I(T)$ returns $|X_i|$.

Given a chordal graph G, its *weighted edges maximal cliques graph decomposition* H is the graph constructed by creating a node for each maximal clique and creating edges between the nodes corresponding to cliques that have vertices in common. The weight $w(e)$ of any edge e is the number of vertices shared by the cliques corresponding to the endpoints of e in G. Unless specifically stated, *weighted edges maximal cliques graph decompositions* are going to be named *graph decompositions* for short.

Let us define a *clique-by-clique* search. Given a graph G and a search $\mathcal{S}$, $\mathcal{S}$ is a *clique-by-clique* search (shortened *cbc*) if it alternates between clearing one maximal clique of G and, if unguarding every possible vertex without allowing recontamination.

Formally, a *cbc* search $\mathcal{S}$ is a sequence of vertex sets such that:

- $\mathcal{S}_0 = \emptyset$.
- For all set $\mathcal{S}_{2\times k+1}$, $k = 0 \dots \frac{|\mathcal{S}|}{2}$, the vertices of one uncleared maximal clique are added to $\mathcal{S}_{2\times k}$.
- For all set $\mathcal{S}_{2\times k}$, $k = 1 \dots \frac{|\mathcal{S}|}{2}$, all the vertices that can be removed from set $\mathcal{S}_{2\times k-1}$ without allowing recontamination, are removed.

To simplify, we call *clearing steps* of $\mathcal{S}$ the odd steps, since we use them to clear a new clique; and *bordering steps* of $\mathcal{S}$ the even steps, since we take the border of what we have already cleaned.

Given a chordal graph G and its tree $(\{X_i | i \in I\}, T = (I, F))$ and T rooted at $r \in I$, let π be a permutation of the d children of node i in T. We say that a search strategy $\mathcal{S}$ clears the d children of i *according to* π if the $\pi(q)$-th child is cleared prior to the $\pi(q+1)$-th child by $\mathcal{S}$ for each $q = 1, \ldots, d-1$.

Given a graph G, a tree decomposition T and a subtree T_i of T; we consider $G[T_i]$ to be the subgraph of G that T_i has as one of its tree decompositions.

The algorithm is divided in three to facilitate its comprehension by the reader.

We first describe an algorithm GSS (Generic Subtree Searching) finding a greedy strategy for $G[T_i]$ with the minimum number of searchers which clears the all the children of i in T according to a given order. Then we generate another subroutine GTDS (Generic Tree Decomposition Searching) that given a tree decomposition of a chordal graph, uses GSS to get the optimal search starting on each possible root of that tree decomposition. Then we use GTDS as a subroutine to design a generic algorithm find the optimal *cbc* monotone connected search strategy for G with minimum number of searchers clearing the nodes of T according to some given order.

Algorithm 1: Find a greedy strategy for $G[T_i]$ with the minimum number of searchers which clears the all the children of r in T according to a given order.

Input: the subgraph $G[T_r]$ and T_r rooted at node r and a set $\mathcal{A}$ containing $\mathcal{S}(G[T_j])$ for each $j \in V(T_r) \setminus (\{r\} \cup L_T)$, an order π of the children of r in T_r, an ordered list L stating, for every vertex of $G[T_r]$, the node in T where it first appears and the number of cliques that contain it.

Output: a partial search strategy for $G[T_r]$

1 $k_\pi \leftarrow w(r)$;
2 Set $d \leftarrow$ the number of children of i;
3 **repeat**
4 Set $s_q \leftarrow +\infty$ for each $q = 1, \ldots, d$;
5 $\mathcal{S}_\pi \leftarrow \langle r \rangle$ (i.e. put all k_π searchers at node r);
6 **for** $q \leftarrow 1$ *to* d **do**
7 **if** $s(\mathcal{S}_\pi \oplus \pi(q)) > k_\pi$ **then**
8 $s_q \leftarrow s(\mathcal{S}_\pi \oplus \pi(q))$
9 $x_q \leftarrow r$
10 go to line 18;
11 $\mathcal{S}_\pi \leftarrow \mathcal{S}_\pi \oplus \pi(q)$
12 **while** $\exists j \in \delta(\mathcal{S}_\pi) \setminus \{r\}$ *such that* $s(\mathcal{S}(G[T_j])) \leq k_\pi - |\delta_V(\mathcal{S}_\pi) \setminus X_j|$ **do**
13 $\mathcal{S}_\pi \leftarrow \mathcal{S}_\pi \oplus \mathcal{S}(G[T_j])$
14 **if** $\delta(\mathcal{S}_\pi) \setminus \{r\} \neq \emptyset$ **then**
15 find $j \in \delta(\mathcal{S}_\pi) \setminus \{r\}$ with the minimum $k' = |\delta_V(\mathcal{S}_\pi) \setminus X_j| + s(\mathcal{S}(G[T_j]))$
16 set $s_q \leftarrow k'$, $x_q \leftarrow j$;
17 **if** $\delta(\mathcal{S}_\pi) \neq \emptyset$ **then**
18 find $p \in \{1, \ldots, d\}$ such that $x_p \in \delta_I(\mathcal{S}_\pi)$ $s_p \leq s_q$ for each $q = 1, \ldots, d$ such that $x_q \in \delta_I(\mathcal{S}_\pi)$, and set $k_\pi \leftarrow s_p$
19 **until** $w(\delta(s_\pi)) \leq w(r)$ *and all the children of* r *are cleared* ;
20 **return** $\mathcal{S}_\pi$.

To calculate the number of searchers that are left after one step, $s(\mathcal{S}_\pi \oplus \pi(q))$ has to take into account all those vertices that were already covered by an ascendant, and also all those vertices that will not form part of the frontier anymore. That is where the list of pairs L is used. When analising the complexity of the algorithm we will explain how this calculations are processed.

Note that for each $r \in I$, let $N_T(r)$ be the set of neighbours of r in T. Define an *ordering* Γ for T as follows: for each node $r \in T$, for each set $D \in \zeta_i \equiv \{D : D = N_T(r) \setminus \{j\} \neq \emptyset$ for $j \in N_T(r)$ or $D = N_T(r)\}$, Γ assigns a non empty set $\Gamma(D)$ of permutations of D.

Let us now analise the second pseudocode.

Given a chordal graph G, $max\_cliques(G)$ is the set of all maximal cliques that can be found in G.

Algorithm 2: Find a monotone connected search strategy for a chordal graph with minimum number of searchers clearing the nodes in T.

Input: a chordal graph G, a rooted tree decomposition (X, T) of G, an ordered list L stating, for every vertex of $G[T]$, the node in T where it first appears and the number of cliques that contain it.

Output: a *cbc* monotone connected search strategy for G.

$\Xi' \leftarrow \emptyset$;
for *each* $r \in I(T)$ **do**
  Let $(i_1, \ldots, i_{|I|})$ be any post-ordering (each child precedes its parent) of vertices of T;
  $\mathcal{A} \leftarrow \emptyset$;
  for $t \leftarrow 1$ *to* $|I|$ **do**
    if i_t *has children* **then**
      $\Xi \leftarrow \emptyset$;
      for *each order* π *in* Γ *for the children of* i_t **do**
        Add to Ξ the search strategy returned by Algorithm 1 for $T_{i_t}, G[T_{i_t}], \pi, \mathcal{A}, L[T_{i_t}]$;
      $\mathcal{S}(G[T_{i_t}]) \leftarrow \mathcal{S}$, where $\mathcal{S} \in \Xi$ and $s(\mathcal{S}) = \min\{s(\mathcal{S}') : \mathcal{S}' \in \Xi\}$;
      $\mathcal{A} \leftarrow \mathcal{A} \cup \{\mathcal{S}(G[T_{i_t}])\}$;
  Let $\mathcal{S}_r \leftarrow \mathcal{S}(G[T_r])$;
  while $C_I(\mathcal{S}_r) \neq I$ **do**
    Find $i \in \delta_I(\mathcal{S}_r)$ with the minimum $|\delta_V(\mathcal{S}_r) \setminus X_i| + s(\mathcal{S}(G[T_i]))$;
    $\mathcal{S}_r \leftarrow \mathcal{S}_r \cup \mathcal{S}(G[T_i])$;
  $\Xi' \leftarrow \Xi' \cup \{\mathcal{S}_r\}$;
return $\mathcal{S}$ such that $\mathcal{S} \in \Xi'$ and $s(\mathcal{S}) = \min\{s(\mathcal{S}') : \mathcal{S}' \in \Xi'\}$.

The last piece of pseudo code is really simple, and shows how we apply the different subroutines given a chordal graph G.

Algorithm 3: Find a *cbc* monotone connected search strategy for a chordal graph with minimum number of searchers.

Input: a chordal graph G, a rooted tree decomposition (X, T) of G, an ordered list L stating, for every vertex of $G[T]$, the node in T where it first appears and the number of cliques that contain it.

Output: an optimal *cbc* monotone connected search strategy for G.

$\mathcal{T} \leftarrow$ a list of pairs with every weighted tree decomposition (X, T) of G and an ordered list L stating, for every vertex of $G[T]$, the node in T where it first appears and the number of cliques that contain it.
$\mathcal{S} \leftarrow$ Algorithm 2 for $T[1]$
for *every* $\mathbf{t} \in \mathcal{T}$ **do**
  if $s($*Algorithm 2 for* $\mathbf{t}) < s(\mathcal{S})$ **then**
    $\mathcal{S} \leftarrow$ Algorithm 2 for $\mathbf{t}$
return $\mathcal{S}$.

For now, we do not care about how we get every tree decomposition, this will be explained later, once we take a

look into the problems that arise from bounding the complexity of the algorithm and the approximation factor of an heuristic.

First, let us see that the algorithm returns what we expect.

Claim 0.1. *Given a chordal graph G and T its maximal clique tree decomposition. Applying the modification of Dereniowski's algorithm on T returns an optimal* cbc *monotone connected search $\mathcal{S}$ of G.*

Proof. Let us first prove, that the result is a *cbc* monotone connected search. It is clique by clique because every vertex in T represents a maximal clique of G, so whenever we are clearing one vertex, we are actually representing the clearing of a maximal clique. Since the original algorithm generated a monotone connected search on T, and we are not changing the way this search is performed, our modification returns a monotone connected search. Therefore, the result is a *cbc* monotone connected search.

Since the algorithm exhaustively generates all possible weighted tree decompositions of G, roots each of them in every possible node, and then tries each possible monotone connected search; all the *clique connected* monotone connected searchs are tried, in particular, any optimal monotone connected search will be given as the output of the algorithm. □

2.4 Problems when bounding the complexity

We know the algorithm returns an optimal *clique connected* monotone connected search for a given chordal graph G, but if we analise the complexity, we realise that it is not polynomial.

First of all, Dereniowski in his paper [9] generates a 3-approximability heuristic because the algorithm is not polynomial. But even if we use this heuristic, the algorithm is not polynomial. This is because we have an exponential number of tree decompositions for any given tree (See Claim 0.2). Let us now introduce a simple heuristic to generate tree decompositions of chordal graphs.

2.5 Heuristic for Transforming a Chordal Graph into a weighted tree

The first part of the algorithm consists in creating a graph decomposition H in which each maximal clique of G becomes a node, and is connected to another node in H whenever the maximal cliques to which they are associated have vertices in common. The number of vertices shared is going to be weight of the edge.

The second part, a little bit more confusing, consists in generating a weighted tree decomposition T of this graph decomposition H, in which the tree generated is a maximum spanning tree of the graph from H [13]. We also create a second structure which will be used by the modification of Dereniowski's algorithm. This is a list of all the vertices of G, with information of on how many maximal cliques they appear and which one in the first in topological order, once we decide a root for the tree.

The pseudocodes for these two algorithms follow:

3 Problems when bounding the Approximability Factor

So far we have showed that if we try to reach the solution by exhaustive search, we will not have a polynomial algorithm. Also, we know that the generic algorithm we provide will generate a path decomposition with a certain number of searchers. But we still have not talked about how good this solution is.

3.1 Clique connected searchs problem

The first problem we encounter is that, since we are using an algorithm that works in a connected way on trees, and we are associating each maximal clique of G with a node on a tree, when we talk about cleaning one node, it's impossible for us to clear the nodes of G without using a clique connected search.

As we can see in Figure 1, there are graphs for which *clique connected* searchs are not the optimal searchs. So we need to bound the way this affects the behaviour of our algorithm.

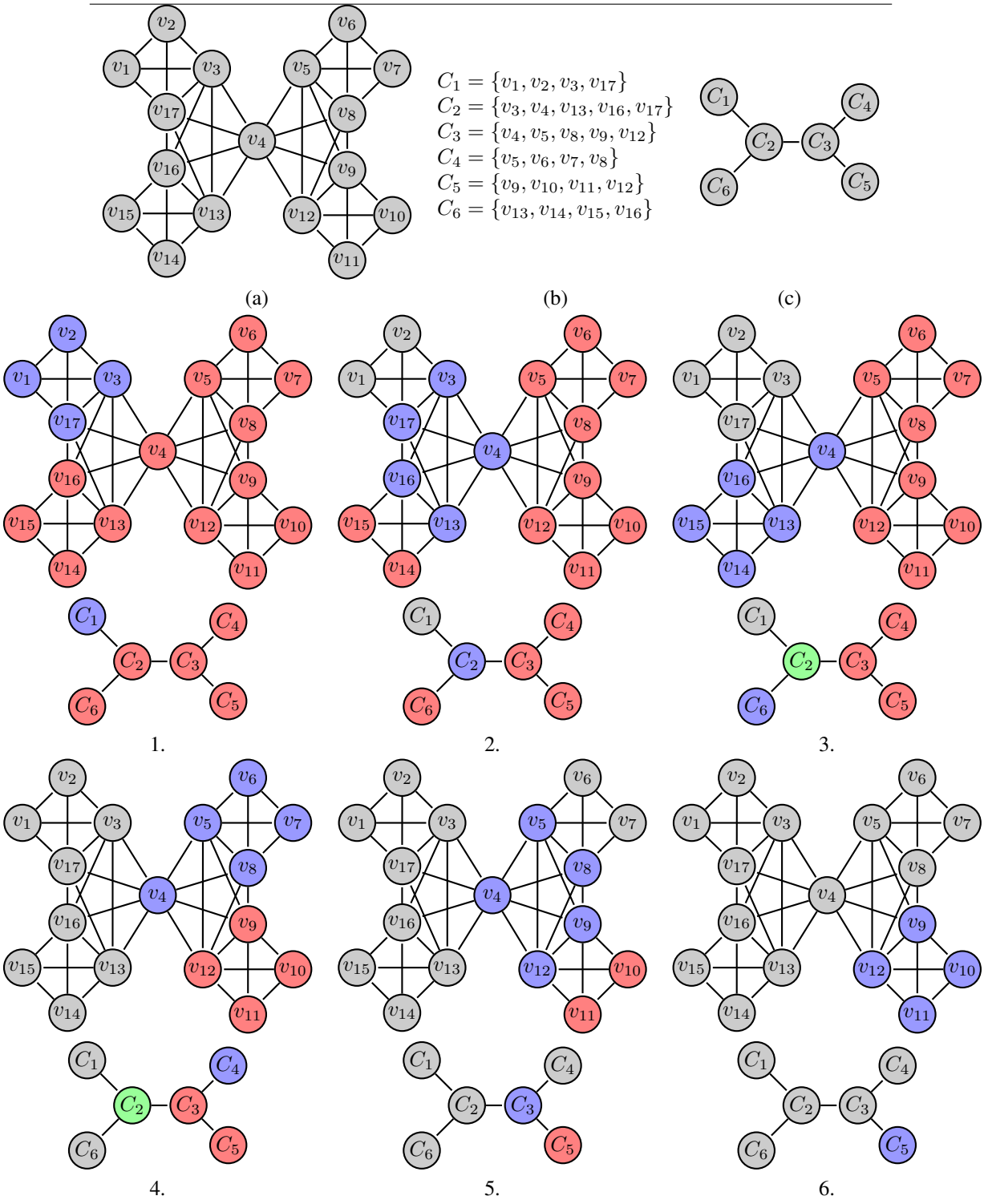

Figure 1: **(a)** Original graph G. **(b)** Nodes of each maximal clique of G. **(c)** Maximal Cliques Graph Decomposition H of G. $1 \ldots 6$ Steps of an Optimal Monotone Connected Search of G, and the correspondent clearing in H.**Red:** Uncleared. **Blue:** Guarded. **Green:** Guarded Frontier. **Gray:** Cleared.

Algorithm 4: Obtain a weighted-edges maximal cliques graph decomposition H from a chordal graph G

Input: A chordal grap G
Output: A weighted-edges graph decomposition $H = (I, F, w)$ and a set of sets X

$X \leftarrow \emptyset$
$I(H) \leftarrow \emptyset$
$F(H) \leftarrow \emptyset$
for *each maximal clique* C_i *of* G **do**
  $X_i \leftarrow V(C_i)$
  $I(H) \leftarrow I(H) \cup \{i\}$
  for *every set* $X_j \in X$ **do**
    if $X_j \cap X_i \neq \emptyset$ **then**
      $F(H) \leftarrow F(H) \cup \{ij\}$
      $w(H, i, j) \leftarrow |X_j \cap X_i|$
  $X \leftarrow X \cup \{X_i\}$
return H and X.

Algorithm 5: Obtain a modified rooted weighted tree T from a weighted-edges graph H

Input: A weighted-edges graph H, a set of sets X, and a node ρ of $I(H)$
Output: A weighted rooted tree $T = (I, F, w)$ with an array L of pairs of integers

$I(T) \leftarrow I(H)$
for *each* $i \in I(T)$ **do**
  $w(i) \leftarrow |X_i|$
for *each maximal clique* C_i *of* H **do**
  if $(\forall e, e' \in E(C_i))w(e) = w(e'))$ **then**
    $r \leftarrow$ closest node to ρ $E(T) \leftarrow E(T) \cup \{ir | i \in V(C_i), i \neq e\}$
  else
    $T' \leftarrow$ a maximum spanning tree of C_i $E(T) \leftarrow E(T) \cup E(T')$
$O \leftarrow$ list with ρ
while $|O| \neq |I(T)|$ **do**
  for *every* $i \in O$ **do**
    if $ij \in F(T) \wedge j \notin O$ **then**
      $O.push\_back(j)$
$L \leftarrow$ array of $|V(G)|$ times $< 0, 0 >$
for $i = 1 \ldots |O|$ **do**
  for *every* $j \in X_{O_i}$ **do**
    if $L_{j_0} = 0$ **then**
      $L_j \leftarrow < i, L_{j_1} + 1 >$
    else
      $L_j \leftarrow < L_{j_0}, L_{j_1} + 1 >$

3.2 Tree Decomposition problem

Given a chordal graph G and H its weighted edges maximal cliques graph decomposition, any clique tree decompostion T of G is a maximum spanning tree of H. (As seen in [13].)
We know graphs have many tree decompositions. If we analize Figure 2, we can see that the monotone connected

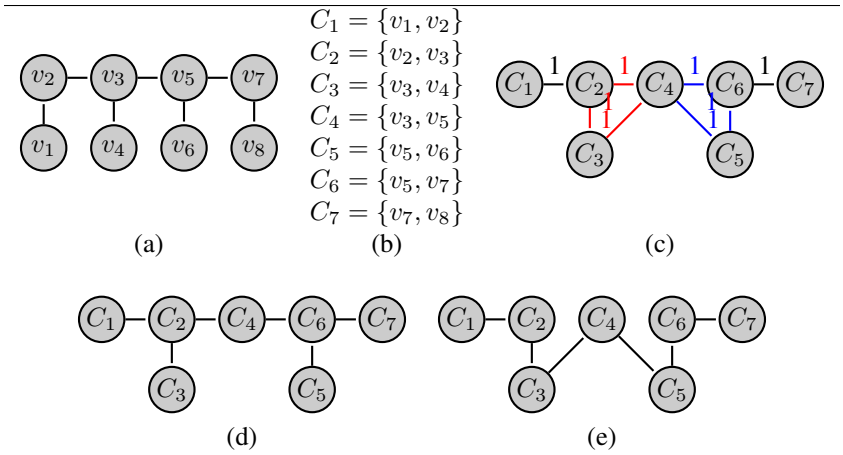

Figure 2: **(a)** Original graph G. **(b)** Nodes of each maximal clique of G. **(c)** Weighted-Edges Maximal Cliques Graph Decomposition H of G. **(d)** T Maximum spanning tree of H. **(e)** T' Maximum spanning tree of H.

search number for T is greater than the one for T' (2 and 3 respectively). This means that we have added one new difficulty to our algorithm. We want to have the best tree decomposition, but we also want the algorithm to be polynomial. Therefore, we have two options, either construct a good tree decomposition, or bound the maximum error we can produce in the result when producing a bad tree decomposition.

Our first approach was to isolate the different problems that came from different tree decompositions to solve them. So we focused on the structures of the original graph that gave possibility to different tree decompositions.

Given a chordal graph G, its *weighted edges maximal cliques graph decomposition* H, and a path p in H; the weight of p, $w(p) \triangleq min\{w(e)|e \in E(p)\}$.

Given a chordal graph G with n maximal cliques, then G has a *weak structure* if and only if its *graph decomposition* H, is a clique in which all the edges have the same weight.

Given a chordal graph G and, G has a *strong structure* if and only if there is only one labelled clique tree decomposition of G.

Given a chordal graph G and G' induced subgraph of G, we say G' is a *strong component* of G[2], if for every *tree decomposition* T of G, there exists a *tree decomposition* T' of G' such that T' is a subtree of T.

A *shared set* of vertices between two cliques C_1 and C_2, is the maximal set of vertices $V = v_1 \dots v_n$ such that $(\forall v_i \in V) v_i \in V(C_1) \cap V(C_2)$.

Claim 0.2. *Given a chordal graph G, the following statements are equivalent:*

1. *G has a weak structure.*
2. *All the maximal cliques in G share the exact same set of k vertices ($k \in \mathbb{N}$).*
3. *G has n^{n-1} different labelled rooted maximal clique tree decompositions.*

Proof. $2 \Rightarrow 1$) If all the maximal cliques in G share the same set of vertices V, such that $|V| = k$. For every pair of cliques C_i, C_j; we have, in H, a pair of nodes n_i, n_j, such that they are connected by the edge $n_i n_j$ and the edge weighs k. Therefore, H is a clique with all its edges of weight k, and G has a weak structure.

$3 \Rightarrow 2$) By absurd, let's assume G has n^{n-1} tree decompositions, but not all maximal cliques share only a set V of vertices, such that $|V| = k$.Therefore, there are two possibilities.

[2]Equivalently, G' is stronger than G, or has a stronger structure.

1. The intersections between different cliques have different cardinalities. But this means there are less possibilities when creating the maximum spanning tree because some edges weight more than others and have to be put always. If all the edges had the same weight, we could use Cayley's Formula [8], which states that there are n^{n-2} different trees, but we know we have $k < n^{n-2}$. Rooting each tree decomposition, we end up having $k \times n$ rooted maximal cliques tree decompositions; which is strictly less than n^{n-1}.

2. The intersections between different cliques have all the same cardinality k, but are not the same set V of vertices. Then, we have at least three cliques C_1, C_2, C_3 such that they all share k vertices, but they don't share the same k vertices. Therefore, there have to be three sets $S_1 = V(C_1) \cap V(C_2), S_2 = V(C_1) \cap V(C_3), S_3 = V(C_2) \cap V(C_3)$ and $\not S_1 = S_2 = S_3$. We don't want all the sets to be the same, because if not, this set would be V.

   But then, each set has at least one element that distinguish it from the rest of the intersections. Properly, $(\exists v_1 \in S_1) v_1 \notin (S_2 \cup S_3)$. But S_1 is the intersection between C_1 and C_2, while $S_2 = V(C_1) \cup V(C_3)$ and $S_3 = V(C_2) \cup V(C_3)$, which means that v_1 is connected to every vertex of S_2 through C_1, and to every vertex of S_3 through C_2. Let's call S' the subgraph induced by the vertices in $\{v\} \cup V(S_1) \cup V(S_2)$. From what we said, it follows that S' is a clique, and therefor $V(S') \cap V(C_1)$ is $S_1 \cup \{v_1\}$; but $|S_1 \cup \{v_1\}| > |S_1| = k$, when one of the assumptions was that every intersection between maximal cliques had k elements. S' may not be maximal, but it is a subclique of a maximal clique that shares at least k vertices with C_1, contradiction that comes from assuming the set V shared is not the same between all cliques.

$1 \Rightarrow 3$) Since G has a weak structure, the maximal clique graph decomposition H of G is a clique of n nodes, being n the number of maximal cliques of G. Knowing the tree decompositions are maximum spanning trees ([13]), and knowing that Cayley's formula [8] also counts the number of spanning trees of a clique of size n, we get that the clique H has n^{n-2} different spanning trees. These trees are maximum, because by definition, every edge of H has the same weigth. To each different spanning tree we can assign n possible different roots, so the total number of labelled maximal cliques rooted tree decompositions is n^{n-1}. □

Given a graph G, we call spanning star S of G, to a spanning tree of G such that $(\exists v \in V(S))(\forall v' \in V(S) \setminus \{v\}) degree(v') = 1$. We call $centre(S)$ such vertex.

Given a weighted-edges graph G, we say G is *uniform* if every edge weighs the same.

Claim 0.3. *Given a chordal graph G, its graph decomposition H, and a rooted tree decomposition T of G:*

If every maximal clique C_i in H is uniform *and T has a spanning star S_i on C_i in which $centre(S_i)$ is the node topologically closest (in H) to the root r of T; our modification to Dereniowski's algorithm returns an optimal* clique ordered *monotone connected search rooted at r.*

Proof. Let us assume there is a *clique ordered* monotone connected search $\mathcal{S}$ that cannot be obtained by applying the algorithm. $\mathcal{S}$ clears every node in T starting from r. For every star S_i in T, let us say $V(S_i) = \{S_{i_1} \dots S_{i_{|S_i|}}\}$, such that $S_{i_1} = centre(S_i)$, and $(\forall j, k \in \{1 \dots |S_i|\}) j < k \Rightarrow S_{i_j}$ is cleared before S_{i_k} by $\mathcal{S}$.

Then, since $centre(S_i)$ is the node topologically closest to r, it is going to be cleared before any other node in $\mathcal{S}$. If $s' \in S_i, s' \neq centre(S_i)$ is cleared before, it means that there is a path from r to s' that does not go through $centre(S_i)$, but that would mean T is not a tree, which is absurd.

But once we have cleared $centre(S_i)$, since S_i is a star, the algorithm can clear whichever other node it needs. For example, given an order $S_{i_2} \dots S_{i_{|V(S_i)|}}$, this can be followed; thus allowing us to return any *clique ordered* monotone connected search rooted at r. In particular, $\mathcal{S}$. □

Corollary 1. *Given a chordal graph G, its associated graph decomposition H such that every maximal clique C_i in H is* uniform*, and a node $r \in I(H)$; if we can make a tree decomposition T such that T has a spanning star S_i on C_i in which $centre(S_i)$ is the node topologically closest to r; then our modification of Dereniowski's algorithm returns an opimal clique connected monotone connected search.*

Claim 1.1. *Given G chordal graph and C_1, C_2, and C_3 maximal cliques in G, such that they are pairwise intersecting. If there is a labelled clique tree decomposition $(X_i | i \in I, T(I, F))$ of G in which $c_2c_3 \in F(T)$ then $V(C_1) \cap V(C_2) \subseteq V(C_3) \vee V(C_1) \cap V)C_3) \subseteq V(C_3)$.*

Proof. Let's assume the premise is true. Since we are in a tree, we have two possibilities:

1. To go from c_1 to c_2 the path goes through c_3: Since we have that $V(C_1) \cap V(C_2) \neq \emptyset \wedge V(C_1) \cap V(C_3) \neq \emptyset \wedge V(C_2) \cap V(C_3) \neq \emptyset$; in the maximal clique graph decomposition H of G, c_1, c_2, c_3 form a clique, from which we chose the edge c_2c_3 to be in F. From the tree decomposition property of the induced subtrees, the path between c_1 and c_3 has to contain $V(C_1) \cap V(C_3)$ in every node. In particular, the bag associate to node c_2 has them. Therefore, $V(C_1) \cap V(C_3) \subseteq V(C_2)$.

2. To go from c_1 to c_3 the path goes through c_2: The same as above but changing the subindices.

□

Given a rooted tree T, and a node $\rho \in V(T)$ we call T_ρ the subtree of T rooted at ρ composed by all its descendents.

Claim 1.2. *Given a graph G and a tree decomposition $(X = \{X_i | i \in I\}, T = (I, F))$. For every pair of nodes $v_1, v_2 \in I$, every node v_{12_i} in the path between them in T has a corresponding bag $X_{12_i} \in X$ such that $X_{v_1} \cap X_{v_2} \subseteq X_{12_i}$.*

Proof. By contradiction, let's suppose there is a node v in I, such that v is in the path between v_1 and v_2 in T, but $X_{v_1} \cap X_{v_2} \nsubseteq X_v$. Then, $(\exists x \in X_{v_1} \cap X_{v_2}) x \notin X_v$. But this means that x does not induce a subtree in T, which means T is not a tree decomposition. This contradiction comes from assuming $X_{v_1} \cap X_{v_2} \subseteq X_v$. □

Claim 1.3. *Given a chordal graph G, a tree decomposition T of G and ρ, α, β, γ, maximal cliques of G such that $\{\alpha\rho, \beta\rho, \alpha\gamma\} \subseteq V(T_\rho)$. Then,*
there exists a tree decomposition T' of G such that $\{\alpha\rho, \beta\rho, \beta\gamma\} \subseteq V(T_\rho)$ if and only if $\alpha \cap \gamma \subseteq \alpha \cap \beta$.

Proof. $\Rightarrow$) By absurd. Let's suppose $\alpha \cap \gamma \nsubseteq \alpha \cap \beta$; therefore, exists $x \in V(\alpha \cap \gamma)$ such that $x \notin V(\beta)$. This implies that if we take the subgraph induced by x in T', it's not a tree, and then T' is not a tree decomposition of G. We reach this contradiction by assuming $\alpha \cap \gamma \nsubseteq \alpha \cap \beta$.

$\Leftarrow$) Since every vertex in $\gamma \cap \alpha$ is contained in $\alpha \cap \beta$, $\gamma \cap \beta = \gamma \cap \alpha = \gamma \cap \rho$. If not, since $\gamma \in V(T_\alpha)$, if there exists $v \in V(\gamma \cap \beta)$ and $v \notin V(\rho)$, the subgraph induced by v in T is not a tree, which is absurd because T is a tree decomposition. This makes $\gamma \cap \beta \subseteq \gamma \cap \rho$.

Through claim 1.2, $\{\gamma\alpha, \rho\alpha\} \subseteq V(T_\rho) \Rightarrow \rho \cap \gamma \subseteq \alpha \cap \gamma$.

Again through claim 1.2, whe have that $\beta \cap \gamma \subseteq \rho \cap \gamma \subseteq \alpha \cap \gamma$. We know by the premise that $\alpha \cap \gamma \nsubseteq \alpha \cap \beta$, which means $\alpha \cap \gamma \subseteq \beta \cap \gamma$. Therefore, $\alpha \cap \gamma = \beta \cap \gamma$.

We have that, $\rho \cap \gamma \subseteq \alpha \cap \gamma$, $\beta \cap \gamma \subseteq \rho \cap \gamma$, and $\alpha \cap \gamma = \beta \cap \gamma$. Which proves what we wanted, $\gamma \cap \beta = \gamma \cap \alpha = \gamma \cap \rho$.

Therefore, if we disconnect γ from α and reconnect it in β we have a candidate tree decomposition T' that fulfils $\{\alpha\rho, \beta\rho, \beta\gamma\} \subseteq V(T_\rho)$. Let us see it is a tree decomposition:

- Since all the vertices of G were contained in bags of T and we have not changed them, all the vertices are contained in bags of T'.
- Since all the edges of G were contained in bags of T and we have not changed them, all the edges are contained in bags of T'.
- For every vertex in $V(G) \setminus V(\beta \cap \gamma)$, the subtrees induced in T' are the same that in T. For the ones in $V(\beta \cap \gamma)$, we take the induced subtree in T, add an edge $\beta\gamma$ and erase $\alpha\gamma$. When we add $\beta\gamma$ we create a cycle, which is broken deleting $\alpha\gamma$. Therefore, every vertex of G induces a subtree in T'.

Then, T' is a tree decomposition of G. □

Given a search strategy $\mathcal{S}$; $C_V(\mathcal{S})$ and $C_E(\mathcal{S})$ are the set of vertices and edges (respectively) cleared at the end of $\mathcal{S}$.

Given a tree T and a node $u \in V(T)$. E_u is the set of edges that are incident to u except for the one that connects u with its parent, ie the ones that connect u with its children (if any). V_u is the set of children of u.

Claim 1.4. *Given a chordal graph G and a* cbc *monotone connected search $\mathcal{S}$.*

$s[\mathcal{S}_{i-1}] < s[\mathcal{S}_{i+1}] \leq s[\mathcal{S}_i] \Rightarrow \mathcal{S}_i$ clears a maximal clique c in G such that $fr(c) \nsubseteq \delta(\mathcal{S}_{i-1})$, where $fr(c)$ is the set of vertices of c that are contained by cliques not cleared in $\mathcal{S}_i$.

Proof. By absurd. Let's suppose $fr(c) \subseteq \delta(\mathcal{S}_{i-1})$. Since $s[\mathcal{S}_{i-1}] < s[\mathcal{S}_i]$ and $\mathcal{S}$ is *cbc*, in $\mathcal{S}_i$ we are clearing a new maximal clique. Therefore, in $\mathcal{S}_{i+1}$, we need to maintain only the frontier of $\mathcal{S}_i$. $\delta(\mathcal{S}_{i+1}) \subseteq \delta(\mathcal{S}_{i-1}) \bigcup fr(c)$; but since $fr(c) \subseteq \delta(\mathcal{S}_{i-1})$, then $\delta(\mathcal{S}_{i+1}) \subseteq \delta(\mathcal{S}_{i-1})$ (because we may not need some vertices from $\delta(\mathcal{S}_{i-1})$ anymore).

Recalling that $\mathcal{S}$ is a *cbc* and that we are clearing a clique in $\mathcal{S}_i$, every node guarded by searchers in $\mathcal{S}_{i-1}$ and in $\mathcal{S}_{i+1}$ are part of their respective frontiers. Therefore, $s[\mathcal{S}_{i+1}] = |\delta(\mathcal{S}_{i+1})| \leq |\delta(\mathcal{S}_{i-1})| = s[\mathcal{S}_{i-1}]$. But $s[\mathcal{S}_{i-1}] < s[\mathcal{S}_{i+1}]$. Contradiction reached by assuming $fr(c) \subseteq \delta(\mathcal{S}_{i-1})$. □

Claim 1.5. *Given a chordal graph G, H its graph decomposition, and T a tree decomposition of G; the nodes of every maximal clique in H induce a subtree in T.*

Proof. Let's suppose they don't necessarily induce a subtree. Let's take a maximal clique C of H such that $V(C)$ does not induce a subtree in T. Then, there is a vertex $v \notin V(C)$ such that $(\exists u \in V_v)(\exists w, w' \in V(C))w \in V(T_u) \vee w' \notin V(T_u)$. Plainly, it means that there is at least one vertex that's not part of C but is in the middle of the path between two vertices of C.

But then, given $u \in V_v$ such that $(\exists w \in V(C))w \in V(T_u)$; $(\forall c_i \in V(C))c_i \in V(T_u)$ and $(\forall c_j \in V(C))c_j \notin V(T_u)$, we have that for every path $P_{ij} = \{c_i \dots c_j\}, v \in P_{ij}$. This means that v has at least one vertex in common with every $c_i, c_j \in V(C)$, because if not, those vertices wouldn't induce a connected subtree in T, and we know that they all share vertices because they are part of a clique in H. But if v has one vertex in common with every vertex of C, this means that v is connected to every vertex of C in H, and the clique C is not maximal. This is a contradiction that comes from supposing that there is a maximal clique C of H such that $V(C)$ does not induce a subtree in T. □

Corollary 2. *Given G chordal and H its weighted egdes maximal clique graph decomposition, G has more than one labelled clique tree decomposition if and only if there exist $u, v \in V(H)$, such that there are two paths p_1, p_2 between them with $w(p_1) = w(p_2) = w$ and every other path between u and v weights at most w.*

References

[1] ALBER, J., BODLAENDER, H. L., FERNAU, H., KLOKS, T., AND NIEDERMEIER, R. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica 33*, 4 (2002), 461–493.

[2] ARNBORG, S., LAGERGREN, J., AND SEESE, D. Easy problems for tree-decomposable graphs. *Journal of Algorithms 12*, 2 (1991), 308–340.

[3] BIENSTOCK, D., AND SEYMOUR, P. D. Monotonicity in graph searching. *Journal of Algorithms 12*, 2 (1991), 239–245.

[4] BJESSE, P., KUKULA, J., DAMIANO, R., STANION, T., AND ZHU, Y. Guiding sat diagnosis with tree decompositions. In *Theory and Applications of Satisfiability Testing* (2004), Springer, pp. 315–329.

[5] BODLAENDER, H., GILBERT, J., HAFSTEINSSON, H., AND KLOKS, T. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms 18*, 2 (1995), 238 – 255.

[6] BODLAENDER, H. L. *Dynamic programming on graphs with bounded treewidth*. Springer, 1988.

[7] BODLAENDER, H. L. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing* (1993), ACM, pp. 226–234.

[8] CAYLEY, A. A theorem on trees. *Quart. J. Math 23* (1889), 376–378.

[9] DERENIOWSKI, D. Approximate search strategies for weighted trees. *Theoretical Computer Science 463* (2012), 96–113.

[10] DERENIOWSKI, D. From pathwidth to connected pathwidth. *SIAM Journal on Discrete Mathematics 26*, 4 (2012), 1709–1732.

[11] DOWNEY, R. G., AND FELLOWS, M. R. *Parameterized complexity*, vol. 3. springer Heidelberg, 1999.

[12] ELLIS, J. A., SUDBOROUGH, I. H., AND TURNER, J. S. The vertex separation and search number of a graph. *Information and Computation 113*, 1 (1994), 50–79.

[13] GALINIER, P., HABIB, M., AND PAUL, C. Chordal graphs and their clique graphs. In *Graph-Theoretic Concepts in Computer Science* (1995), Springer, pp. 358–371.

[14] GUSTED, J. On the pathwidth of chordal graphs. *Discrete Applied Mathematics 45*, 3 (1993), 233 – 248.

[15] KINNERSLEY, N. G. The vertex separation number of a graph equals its path-width. *Information Processing Letters 42*, 6 (1992), 345–350.

[16] KIROUSIS, L. M., AND PAPADIMITRIOU, C. H. Searching and pebbling. *Theoretical Computer Science 47*, 3 (1986), 205–218.

[17] LAPAUGH, A. S. Recontamination does not help to search a graph. *Journal of the ACM 40*, 2 (1993), 224–245.

[18] MIHAI, R., AND TODINCA, I. Pathwidth is np-hard for weighted trees. In *Frontiers in Algorithmics*, X. Deng, J. Hopcroft, and J. Xue, Eds., vol. 5598 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 181–195.

[19] ROBERTSON, N., AND SEYMOUR, P. D. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms 7*, 3 (1986), 309–322.

[20] SEYMOUR, P. D., AND THOMAS, R. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B 58*, 1 (1993), 22–33.

[21] YANG, B., DYER, D., AND ALSPACH, B. Sweeping graphs with large clique number. *Discrete Mathematics 309*, 18 (2009), 5770 – 5780.