Maximal Common Connected Sets of Interval Graphs

Michel Habib, Christophe Paul, Mathieu Raffinot

HAL Id: lirmm-00108787
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00108787

Submitted on 23 Oct 2006

Maximal Common Connected Sets
of Interval Graphs[⋆]

Michel Habib[1], Christophe Paul[1], and Mathieu Raffinot[2]

[1] CNRS - Université de Montpellier 2, LIRMM
161 rue Ada, 34392 Montpellier Cedex 5, France
{habib,paul}@lirmm.fr
[2] CNRS - Laboratoire Génome et Informatique, Evry
and Ecole Normale Supérieure, Paris, France.
raffinot@genopole.cnrs.fr

Abstract. Given a pair of graph $G_1 = (V, E_1)$, $G_2 = (V, E_2)$ on the same vertex set, a set $S \subseteq V$ is a maximal common connected set of G_1 and G_2 if the subgraphs of G_1 and G_2 induced by S are both connected and S is maximal the inclusion order. The maximal Common Connected sets Problem (CCP for short) consists in identifying the partition of V into maximal common connected sets of G_1 and G_2. This problem has many practical applications, notably in computational biology.
Let $n = |V|$ and $m = |E_1| + |E_2|$. We present an $\mathcal{O}((n + m) \log n)$ worst case time algorithm solving CCP when G_1 and G_2 are two interval graphs. The algorithm combines maximal clique path decompositions of the two input graphs together with an Hopcroft-like partitioning approach.

1 Introduction

Let $G = (V, E)$ be a loopless undirected graph. The degree of a vertex $x \in V$ in the graph G is denoted by $d_G(x)$. Let X be a subset of vertices of G, we denote $G[X]$ the subgraph induced by X: the set of vertices of $G[X]$ is X and its edge set is $E_X = E \cap \{(u, v) \mid u \in X, v \in X\}$. We denote by $m_X = |E_X|$ the number of edges in $G[X]$ and by $|G[X]| = |X| + m_X$ the size of the induced subgraph. A set X of vertices is *connected in G* if $G[X]$ is a connected graph.

Let $\mathcal{F}$ be a non empty family of graphs on (or restricted to) the same vertex set[3], say $\mathcal{F} = \{G_1 = (V, E_1), \dots G_k = (V, E_k)\}$. A set S of vertices ($S \subseteq V$) is said *connected* in $\mathcal{F}$ if X is connected in all $G_i \in \mathcal{F}$.

Definition 1. *A set* $S \subseteq V$ *of vertices is a* maximal common connected set *of a family* $\mathcal{F} = \{G_1 = (V, E_1), \dots G_k = (V, E_k)\}$ *of graphs if* S *is a connected set in* $\mathcal{F}$ *and no other set* $X \supset S$ *is connected in* $\mathcal{F}$.

---

[3] Or equivalently the vertices could be considered uniquely labeled.

Trivially, the maximal common connected sets of $\mathcal{F}$ form a partition of the vertex set. If $\mathcal{F}$ only contains a single graph, a maximal common connected set reduces to the classical notion of *connected component*. Well-known linear time algorithms that identify the connected components partition of a single graph exist. However, when $|\mathcal{F}| \geqslant 2$, the problem becomes much harder. In [8], the problem of finding maximal common connected sets of two graphs, namely the *CC-Problem* (CCP for short), was addressed. If $|\mathcal{F}| > 2$, the problem is named *gen-CCP* (see Section 2).

CC-Problem:
> *Input:*   two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$.
> *Output:*   the partition of V into the maximal common connected sets
>   of $\{G_1, G_2\}$.

Before we continue with the discussion, let us define some notations: the number of vertices will be denoted by $n = |V|$, while m will design the total number of edges in the two graphs G_1 and G_2, i.e. $m = m_1 = m_2$ with $m_1 = |E_1|$, $m_2 = |E_2|$.

A natural approach to solve CCP is to first search the maximal connected components of G_1. Then, in each of these components, search the connected components of G_2. In each such new connected component of G_2, search the maximal connected components of G_1, and repeat this process until the two sets of components on G_1 and G_2 are similar. A simple example on which this approach yields to $\Omega(n)$ steps is given in [8], where the two graphs are in fact interval graphs. Since each step consists in a search on a subgraph whose size may decrease one by one, the complexity of this method is $\Omega(n(n + m))$ worst case time. The algorithm proposed in [8] runs in $\mathcal{O}(n \log n + m \log^2 n)$ for general graphs. Their algorithm mixes dynamical connectivity maintenance with a partitioning approach.

However, obtaining faster algorithms for solving CCP is a real challenge, since the graphs currently considered in many applications, like computational biology, are huge: comparing graphs with more than $250\,000$ vertices becomes frequent (see for instance the TERAPROT project [14]).

This paper improves the practical and theoretical complexity of CCP for a restricted graph family, that of *interval* graphs. A graph is an *interval graph* iff there is a one-to-one mapping between its vertices and a set of intervals on the real line such that two vertices are adjacent iff their corresponding intervals intersect [13]. This family of graphs represents a large part of the graphs involved in applications of CCP in computational biology, because a chromosome is naturally represented by construction as the interval graph of smaller sequences (cDNA, ESTs, etc). For instance, comparing the longest "common" contigs of two chromosomes built on the same cDNA database requires solving CCP on interval graphs.

We present an algorithm for solving CCP on interval graph in $\mathcal{O}((n+m) \log n)$ worst case time. The algorithm is both faster and simpler than the algorithm solving CCP on general graphs. It combines an Hopcroft-like partitioning frame-

work together with a kind of dynamical maintenance of a spanning separator forest. Interval graphs are represented through a forest of clique paths that roughly captures all the possible separators of the graphs. This forest is "dynamically" maintained, in the sense that we are able to quickly compute the new clique representation after extracting a set of vertices. Sets of vertices are extracted following an Hopcroft-like partitioning framework, inspired by the gene teams identification algorithm [1] that has later been proved to resemble a simplified Hopcroft partitioning approach [4].

This article is organized as follows. Section 2 explains the whole framework of a recursive partitioning algorithm to solve CCP. Section 3 presents data-structures and algorithms that allow us to improve the time complexity for interval graphs. Finally the whole algorithm and its complexity are explained and proved.

2   A Recursive Partitioning Algorithm

Solving CCP on two graphs G_1 and G_2 on the same vertex set V consists in computing a partition of V whose parts are the maximal common connected sets. A *partition* $\mathcal{P}$ of a set V is a set of disjoint subsets $\{\mathcal{X}_1, \ldots \mathcal{X}_k\}$, whose union is exactly V. Our partitioning algorithm is based on the following simple lemma.

Lemma 1. *Let G_1 and G_2 be graphs on the same vertex set V and let C be a maximal connected component of G_1 distinct from V. Then*

$$CCP(G_1, G_2) = CCP(G_1[C], G_2[C]) \cup CCP(G_1[V \backslash C], G_2[V \backslash C])$$

Proof. Let S be a maximal common connected set of the pair G_1 and G_2. By definition S is connected in G_1. Since C is a maximal connected component, S is either included in C or in $V \backslash C$. It follows that any maximal common connected set of G_1 and G_2 is either a maximal common connected set of $G_1[C]$ and $G_2[C]$ or of $G_1[V \backslash C]$ and $G_2[V \backslash C]$.   □

A simple paradigm for a recursive algorithm derives from Lemma 1. The inputs are two graphs G_1 and G_2 on the same vertex set V and a partition $\mathcal{P}$ of V. Initially, $\mathcal{P}$ is set to the trivial partition $\{V\}$. Then, it first searches for a connected component of G_1 or G_2 distinct from V. If such a component C exists, according to Lemma 1, two recursive calls are launched on the subgraphs induced respectively by C and $V \backslash C$. A sketch of this algorithm, named **CCP-Algorithm**, is given in Figure 1.

Lemma 2. *CCP-Algorithm* *computes the maximal common connected set partition of a pair of graphs.*

Proof. The algorithm ends since (a) the recursive calls are launched on strict subgraphs and (b) it stops the recursive calls when both graphs are connected. The correctness of the algorithm directly derives from Lemma 1.   □

CCP-Algorithm$(G_1 = (V, E_1), G_2 = (V, E_2))$
1.   **If** G_1 and G_2 are both connected **Then**
2.   **Return** $\mathcal{P} = \{V\}$
3.   **Else**
4.   **If** G_1 is not connected **Then**
5.   Let C be a connected component of G_1
6.   **Else**
7.   Let C be a connected component of G_2
8.   **End of if**
9.   Let $\mathcal{P}' =$**CCP-Algorithm**$(G_1[C], G_2[C])$
10.   Let $\mathcal{P}'' =$**CCP-Algorithm**$(G_1[V \backslash C], G_2[V \backslash C])$
11.   **Return** $\mathcal{P} = \mathcal{P}' \cup \mathcal{P}''$
12.   **End of if**

Fig. 1. Recursive algorithm computing the maximal common connected sets partition of the vertex set V of a pair of graphs G_1 and G_2

Generalization to an arbitrary number of graphs One can also consider the *gen-CC Problem* of computing the maximal common connected sets of an arbitrary number of graphs (i.e. CCP applied to a family of $k \geq 2$ graphs). Lemma 1 can be generalized and the algorithm modified.

Lemma 3. *Let* $\mathcal{F} = \{G_1, \ldots, G_k\}$ *be a family of graphs on the same vertex set* V *and let* $C \neq V$ *be a connected component of* G_1*. Then* $\textbf{gen-CCP}(G_1 \ldots G_k) = \textbf{gen-CCP}(G_1[C] \ldots G_k[C]) \cup \textbf{gen-CCP}(G_1[V \backslash C] \ldots G_k[V \backslash C])$

Proof. Let S be a maximal common connected set of the k graphs. By definition S is connected in G_1. Since C is a connected component of G_1, S is either included in C or in $V \backslash C$. It follows that any maximal common connected set of $\mathcal{F}$ is either a maximal common connected set of the family $\{G_1[C], \ldots, G_k[C]\}$ or of the family $\{G_1[V \backslash C], \ldots G_k[V \backslash C]\}$.   □

It is straightforward to modify the algorithm in order to handle an arbibrary number of graphs. The connected component C has just to be a connected component of an arbitrary graph among $G_1, \ldots G_k$. The generalized algorithm is depicted in Figure 2.

Observations on the complexity Notice that the main difficulties of the above algorithms are first to compute a connected component C of one input graph if it exists and then to extract the subgraphs induced by C and $V \backslash C$. However, without *ad-hoc* data-structures, such a recursive approach may yield to a $\Theta(n(n + m))$ worst case time algorithm. In [8], using a sophisticated data-structure to maintain dynamic connectivity [11], an $\mathcal{O}(n \log n + m \log^2 n)$ algorithm for CCP on general graphs was proposed.

For simplicity, in the rest of the paper, we restrict the study to the case of 2 graphs. The main difference between CPP and gen-CCP consists in managing

gen-CCP-Algorithm$(G_1 = (V, E_1), \ldots G_k = (V, E_k))$
1.   **If** all G_i $(1 \leqslant k)$ is connected **Then**
2.   **Return** $\mathcal{P} = \{V\}$
3.   **Else**
4.   Let G_i be a non connected graph among $G_1, \ldots G_k$
5.   Let C be a connected component of G_i
6.   Let $\mathcal{P}' =$**gen-CCP-Algorithm**$(G_1[C], \ldots G_k[C])$
7.   Let $\mathcal{P}'' =$**gen-CCP-Algorithm**$(G_1[V \backslash C], \ldots G_k[V \backslash C])$
8.   **Return** $\mathcal{P} = \mathcal{P}' \cup \mathcal{P}''$
9.   **End of if**

Fig. 2. Recursive algorithm to compute the maximal common connected sets partition of the vertex set V of a family $\mathcal{F} = \{G_1, \ldots, G_k\}$ of k graphs

the connected graphs and the non-connected graphs. It will clearly appear below that the data-structures we use, enable us to run the test of line 1 of algorithm **CCP-Algorithm** (Figure 1) in $\mathcal{O}(1)$ time. It follows that the complexity of gen-CCP differs from the complexity of CCP only by a factor k. The remainder of the article focuses on improving the worst case complexity of **CCP-Algorithm** on two interval graphs.

Section 3 presents two algorithms that retrieve the connected components of the subgraphs $G_1[C], G_2[C], G_1[V \backslash C], G_2[V \backslash C]$ after having extracted C. These algorithms strongly rely on interval graph structural properties. Their complexities are both $\mathcal{O}(|C| + d(C))$ where $d(C) = \sum_{x \in C} d(x)$, which is proportional, not exactly to the size of their induced subgraph, but close to.

However, even if an arbitrary connected component C can be extracted in $\mathcal{O}(|C| + d(C))$, it would not be enough to reach the announced $\mathcal{O}((n+m)\log(n))$ complexity. It could still lead to $\Theta(n(n+m))$ operations. To lower the whole complexity, we combine the extraction scheme with an Hopcroft's partitioning approach [12]. Only *small* connected components have to be considered. *Small* means that the size of the connected component considered has to be less than or equal to the half of the size of the original graph. Such a connected component always exists if the graph is not connected, but it is quite complicated to retrieve it efficiently. This is the purpose of the SIS algorithm of subsection 3.3. The whole partionning approach is presented in section 4.

3   Clique Path Representation of Interval Graphs

This section presents the material for managing the interval graphs. We first introduce some well-known properties and the data-structures used in the algorithms. Then two algorithms that update the data-structures for induced subgraphs are developed. These algorithms allow efficient recursive calls. Finally, the last SIS algorithm searches for a small connected component of a given interval graph.

3.1   Preliminaries and Data-structures

Let $G = (V, E)$ be a graph and $G[X]$ be the subgraph induced by X. We set $d(X) = \sum_{x \in X} d_G(x)$. Notice that the definition of interval graphs (see the introduction) directly implies that this family of graphs is *hereditary*: any induced subgraph of an interval graph is an interval graph. A *clique* is a complete induced subgraph (not necessarily *maximal* for the inclusion).

Definition 2. *Let $G = (V, E)$ be a connected interval graph. A* clique decomposition path *of G is a path $P = (\mathcal{C}, F)$ such that:*

1. *any set $C \in \mathcal{C}$ is a set of vertices and $\bigcup_{C \in \mathcal{C}} C = V$;*
2. *any $(u, v) \in E$ is contained in some $C \in \mathcal{C}$;*
3. *the set $\mathcal{C}_u = \{C \in \mathcal{C} \mid u \in C\}$ induces a subpath P_u of P*
4. *any $C \in \mathcal{C}$ is a clique;*

A clique decomposition path will be denoted hereafter by CDP. Notice that a CDP gives an interval intersection model of the interval graph: the underlying path P and the family of subpaths P_u that contains the vertex u. If the condition 4 is not required, a decomposition path can be defined for arbitrary graphs and this is the basis of the pathwidth theory (see [2]).

Dealing with interval graphs, we usually define the *Maximal Clique decomposition Path* (shorten as MCP) where any clique $C \in \mathcal{C}$ has to be a maximal clique.

A *separator* is a set S of vertices whose removal disconnects the graph in two or more connected components. A separator S is *minimal* if there exists a pair of vertices u, v, which are separated by S and such that no subset of S separates u and v in different connected components. Since interval graphs are chordal (graphs with no induced cycle of length larger than 3), any minimal separator is a clique [7]. The following lemma gives some hints on the structure of the set of minimal separators of an interval graph.

Lemma 4 (*e.g.* [9]).
Let P be a MCP of an interval graph G. A set of vertices S is a minimal separator iff it is the intersection S of some consecutive cliques C_1 and C_2 in P.

For our purposes, we label the edges of a CDP by the intersection of the corresponding cliques. A non-connected interval graph clearly enjoys a CDP: the edges between two cliques of different paths are labelled by the empty set since these cliques belong to different connected components and are disjoint. The number of cliques in a CDP P is denoted $|P|$. We say that the set of paths defines a *linear forest* denoted CPF for Clique Path Forest. When all the paths are maximal, the forest is denoted MCPF.

Lemma 5 (*e.g.* [9]). *Let G be an interval graph with n vertices and m edges. Any MCP is of size $\mathcal{O}(n + m)$.*
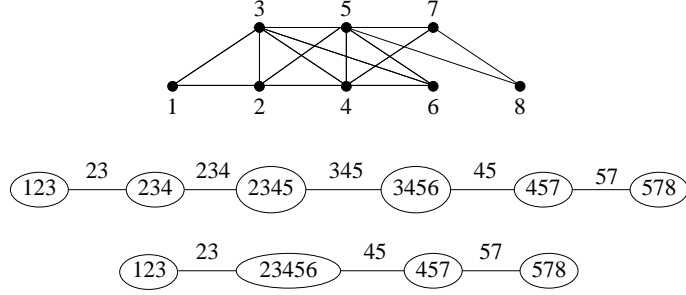
Fig. 3. An interval graph with two CDPs. The second is maximal.

Many linear-time interval graphs recognition algorithms exist. The first is due to Booth and Lueker in 1976 [3]. Most recent ones [10, 6] are much simpler than the original. All these algorithms can be easily modified to output in $O(n + m)$ a maximal clique path decomposition.

For algorithmic settings, in the case of non-connected interval graphs, the set of paths of a CPF is stored in a list $\mathcal{F}$. The cliques of a CDP are stored in a doubly linked list and the CDP are rooted at one of their extremities. A given clique C stores a pointer to its *father* $f(C)$ and to its *child* $s(C)$; its set of vertices is stored in a doubly linked list and its size is denoted n_C. Moreover, each edge is assigned to a record containing: (a) its two extremities; (b) the label of its minimal separator (see lemma 4) whose vertices are stored in a doubly linked list; (c) the size n_S of this separator. In addition, two lists, namely L_S and L_C, are associated to any vertex x. The list L_S (resp. L_C) contains pointers to the copies of x in each separator (resp. clique) containing x.

3.2   Dynamic Clique Decomposition Path

Lemma 6. *Let $P = (\mathcal{C}, F)$ be a CDP of $G = (V, E)$ and $X \subseteq V$. Then $P' = (\mathcal{C}', F')$ defined as follows is a CDP of $G[V \backslash X]$.*

 – $\mathcal{C}' = \{f(C) \mid C \in \mathcal{C}\}$, *where $f(C) = C \backslash X$;*
 – $(f(C_1), f(C_2)) \in F'$ *iff $(C_1, C_2) \in F$;*
 – $(f(C_1), f(C_2))$ *is labelled by $f(C_1) \cap f(C_2) = (C_1 \cap C_2) \backslash X$.*

Proof. Let us consider two vertices u and v belonging to $V \backslash X$. The proof directly derives from the definition. First any $f(C)$ is a clique and $\bigcup_{C \in \mathcal{C}} f(C) = V \backslash X$. If u and v are adjacent, there exists a clique $C \in \mathcal{C}$ containing both u and v. Clearly $f(C)$ also contains both u and v. Since the set $\mathcal{C}_u \subset \mathcal{C}$ of cliques containing $u \in V \backslash X$ occurs consecutively in P, the set $\mathcal{C}'_u \subset \mathcal{C}'$ also occurs consecutively in P'.   □

Notice that some separators may be empty after the extraction of X, in which case the resulting CDP is in fact a CPF.

For complexity issues, the above operation is implemented by two different algorithms. Given a CDP, the first one removes the vertices of a given set from each clique: it is called REMOVE. The second, in contrast, computes the intersection of any clique with a given set: it is called EXTRACT.

REMOVE(P, X)

1.   Let $\mathcal{F}$ be a linear forest containing P
2.   **For** any $x \in X$ **Do**
3.   **For** any clique C st $x \in C$ **Do**
4.   Remove x from C
5.   Decrease n_C by 1
6.   **End of for**
7.   **For** any separator S between cliques C and C' st $x \in S$ **Do**
8.   Remove x from S
9.   Decrease n_S by 1
10.   **If** $n_S = 0$ **Then**
11.   Let (P, C_r) be the CDP containing the edge (C, C') labeled by S
12.   Remove the edge (C, C') from P (C is the father of C')
13.   Create in $\mathcal{F}$ the new CDP (P', C')
14.   **Else**
15.   **If** $n_S = n_C$ **Then**
16.   Remove C from P
17.   Connect $s(C)$ and $f(C)$ with the edge labelled by $s(C) \cap f(C)$
18.   **End of if**
19.   **If** $n_S = n_{C'}$ **Then**
20.   Remove C' from P
21.   Connect $s(C')$ and $f(C')$ with the edge labelled by $s(C') \cap f(C')$
22.   **End of if**
23.   **End of if**
24.   **End of for**
25.   **End of for**
26.   **Return** $\mathcal{F}$

Fig. 4. Maintaining a MCPF of a graph after removing a set of vertices X from an interval graph represented by a MCP P.

The pseudo-code of algorithm REMOVE(P, X) is given in Figure 4. Lemma 7 states its validity and time complexity.

Lemma 7. *Let P be a MCP of the connected interval graph $G = (V, E)$. The algorithm REMOVE(P, X) (Figure 4) computes in $O(|X| + d(X))$ time a linear forest of $G[V \setminus X]$ where each path is a MCP of the corresponding connected component.*

Proof. First, notice that by lemma 6, when the vertices of X have been removed (lines 4 and 8), $\mathcal{F}$ is a CDP of $G[V \backslash X]$ (but no longer a maximal one). Let S be the intersection between two consecutive cliques C and C' of a given path $P \in \mathcal{F}$ (w.l.o.g. $C = f(C')$).

- $S = \emptyset$ (S is no longer a separator since it is empty): Since P is a CDP C and C' belongs to different connected components, the path P can be split into two CDPs. The first one contains the clique from the root to C while the second one is rooted at C' and contains the clique derivating from C' (lines 10-13).
- $n_S = n_C$ (the case $n_S = n_{C'}$ is similar): C is no longer a maximal clique (it is included in C'). Therefore we can remove C from P (lines 15-22).

It follows that when $\mathcal{F}$ has been cleaned up, any clique is a maximal clique and each new CDP is therefore a MCP. For complexity issue, since the number of copies of elements of X is $\mathcal{O}(|X| + d(X))$ and since each copy is touched once, removing X cost $\mathcal{O}(|X| + d(X))$. The cleaning can be done within the same complexity since (a) removing a separator or a clique costs $\mathcal{O}(1)$; (b) the number of removing operations is bounded by the number of copies of elements of X.   □

We now consider the maximal clique path decomposition of the induced subgraph $G[X]$. The pseudo-code of algorithm EXTRACT(P, X) is given in Figure 5. The next lemma 8 states its validity and time complexity.

Lemma 8. *Let F be a MCP of the connected interval graph $G = (V, E)$. The algorithm EXTRACT(F, X) (Figure 5) computes in $\mathcal{O}(|X| + d(X))$ time a MCPF of $G[X]$.*

Proof. A similar proof than that of lemma 7 shows that a linear forest of $G[X]$ can be computed in $\mathcal{O}(|X| + d(X))$.   □

3.3   Smaller Induced Subgraph (SIS) Algorithm

The SIS algorithm on two MCPs P_1 and P_2 allows us to find the smallest of the two induced subgraphs in time proportional to the size of this smallest subgraph. The difficulty comes from that the sizes of the two paths are not necessarily representative of the sizes of their induced subgraphs. It may happen that $|P_1| < |P_2|$, but that $|G[V_1]| > |G[V_2]|$, where V_1 (resp. V_2) is the set of vertices contained in the cliques of P_1 (resp. P_2).

To overcome this obstacle, we use a trick. We perform simultaneously a Depth First Search (DFS) on the two paths. We read a new clique (or path node) of each MCP alternatively, until we reach the end of one of the paths. During this search, we compute for each path the sums S_1 and S_2 of the sizes of the cliques we encountered.

At the end of these simultaneous DFS, the smallest MCP, say P_1, has been totally covered, and S_1 is the size of its induced subgraph. If $S_1 \leq S_2$, the simplest case (a), the subgraph induced by P_1 is smaller than that induced by

---

EXTRACT(P, X)
1.   Let $\mathcal{F}$ be an empty linear forest
2.   **For** any $x \in X$ **Do**
3.   **For** any clique C containing x **Do**
4.   **If** C has not been already duplicated **Then**
5.   Create a copy $C' = \{x\}$ in $\mathcal{F}$ in a new singleton CDP
6. $n_{C'} \leftarrow 1$
7.   **Else**
8.   Let C' the existing copy of C
9. $C' \leftarrow C' \cup \{x\}$
10. $n_{C'} \leftarrow n_{C'} + 1$
11.   **End of if**
12.   **End of for**
13.   **For** any separator S containing x **Do**
14.   Let (C_1, C_2) be the edge labeled by S in P (wlog $C_1 = f(C_2)$)
15.   **If** (C_1, C_2) has not been duplicated **Then**
16.   Create a new edge (C_1', C_2') labelled by x
17. $n_{S'} \leftarrow 1$
18.   **Else**
19.   Let S' the label of the edge (C_1', C_2')
20. $S' \leftarrow S' \cup \{x\}$
21. $n_{S'} \leftarrow n_{S'} + 1$
22.   **End of if**
23.   **End of for**
24.   **End of for**
25.   Remove from $\mathcal{F}$ any non maximal clique as in lines 15-22 of REMOVE (see Figure 4)
26.   **Return** $\mathcal{F}$

Fig. 5. Maintaining a MCPF of the induced subgraph of $G[X]$ when extracting X from G.

P_2, and SIS returns P_1. Otherwise, if $S_1 > S_2$ we continue the DFS of the second path P_2, computing the new sum S_2' for each new clique encountered. Figure 6 illustrates this search. The process goes on, until, case (b), either the whole MCP P_2 has been visited, in which case $S_2' \leq S_1$ and SIS returns P_2, either, case (c), S_2' becomes greater than S_1 and SIS returns P_1. Figure 7 illustrates these two last cases.

It is obvious that SIS returns the MCP which represents the smallest induced graph. We prove in Lemma 9 that its complexity only depends on the size of this smallest induced subgraph.

Lemma 9. *Given two MCP's, Algorithm SIS returns the one that represents the smallest induced subgraph, say $G[X]$ in time $O(|X| + m_X)$.*

Proof. Let c_1 (resp. c_2) be the number of cliques visited in P_1 (resp c_2) at the end of SIS algorithm. The total number of cliques visited is $c_1 + c_2$.
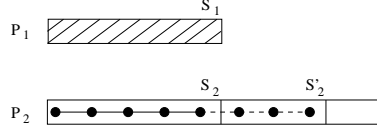
Fig. 6. Continuing the Depth First Search in the longest MCP P_2 if $S_1 > S_2'$ while until either $S_2' \geq S_1$ or P_2 is completely covered.

In case (a), P_1 represents the smallest subgraph $G[X]$ of size $S_1 = |X| + m_X$. The number $c_1 + c_2$ is in this case $2c_1$. As $c_1 \leq S_1$ (lemma 5), the complexity of SIS is $O(|X| + m_X)$.

In case (b), $S_2' \leq S_1$. The path P_1 represents the smallest subgraph $G[X]$ of size $S_2' = |X| + m_X$. Therefore SIS returns P_2. As the first DFS stopped first w.l.o.g on P_1, $c_2 > c_1$, and, as $c_2 \leq S_2'$ (lemma 5), $c_1 + c_2 < 2S_2'$ and the complexity of SIS is $O(|X| + m_X)$.

In case (c), the path P_1 represents the smallest subgraph $G[X]$ of size $S_1 = |X| + m_X$. As $c_2 \leq S_1 + 1$, $c_1 + c_2 \leq 2S_1 + 1$ and the complexity of SIS is $O(|X| + m_X)$. □

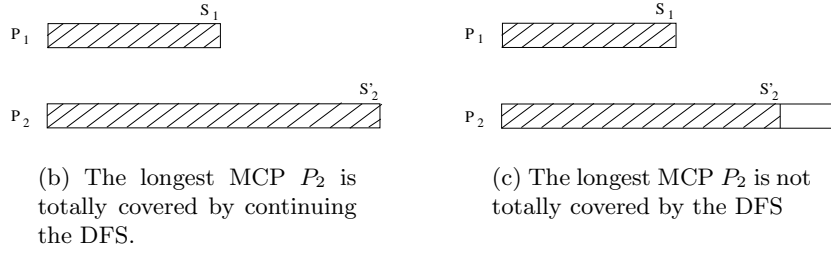(b) The longest MCP P_2 is totally covered by continuing the DFS.

(c) The longest MCP P_2 is not totally covered by the DFS

Fig. 7. Two ending cases when continuing the DFS on the longest MCP P_2. In the first case (a), the DFS covers all the vertices of P_2. Then as $S_2' \leq S_1$, SIS returns P_2. In the second case, at most S_1 vertices of P_2 have been visited by the DFS without exploring all the tree. Then SIS returns P_1.

4 The Whole CCP Algorithm For Interval Graphs

The whole CCP algorithm for interval graphs (CCPI-Algorithm) is shown on Figure 8. The algorithm takes as input two lists L_1 and L_2 that are respectively the clique forest decompositions of the two graphs G_1 and G_2. It outputs a partition of the vertex set. At lines 5 and 7, it searches for a connected component

C whose size is at most half of the size of the corresponding graph. By lemma 9, it can be done in time proportional to that connected component. Let P be the MCP of C. W.l.o.g. we assume that C is a connected component of G_1 and P is the first MCP of L_1. Lines 11 and 12 compute the four subgraphs on which the recursive calls are launched. Using EXTRACT(L_2,P) and REMOVE(L_2,P) we compute the subgraphs of G_2 induced respectively by the vertices V_P belonging to the cliques of P and $V \setminus V_P$. As seen in Lemmas 8 and 7, it can be done in $\mathcal{O}(|V_P| + d(V_P))$.

CCPI-Algorithm(L_1, L_2)
1.   **If** $|L_1| = 1$ and $|L_2| = 1$ **Then**
2.   **Return** $\mathcal{P} = \{V\}$ /* G_1 and G_2 are both connected */
3.   **Else**
4.   **If** $|L_1| \geq 2$ **Then**
5. $P \leftarrow \text{SIS}(L_1[1], L_1[2])$ /* G_1 is not connected */
6.   **Else**
7. $P \leftarrow \text{SIS}(L_2[1], L_2[2])$ /* G_2 is not connected */
8.   **End of if**
9.   /* **we assume below w.l.o.g that** $P = L_1[1]$ */
10. $L'_1 \leftarrow P$; $L''_1 \leftarrow L_1 \backslash P$
11. $L'_2 \leftarrow \text{EXTRACT}(L_2, V[P])$; $L''_2 \leftarrow \text{REMOVE}(L_2, V[P])$
12.   Let $\mathcal{P}' = $**CCPI-Algorithm**$(L'_1, L'_2)$
13.   Let $\mathcal{P}'' = $**CCPI-Algorithm**$(L''_1, L''_2)$
14.   **Return** $\mathcal{P} = \mathcal{P}' \cup \mathcal{P}''$
15.   **End of if**

Fig. 8. Recursive algorithm to compute the partition of the vertex set into maximal common connected sets of two interval graphs G_1 and G_2 represented respectively by the MCPF $L1$ and L_2.

Theorem 1. *The CCPI-algorithm applied on $MCPF(G_1)$ and $MCPF(G_2)$ correctly identifies the common connected components of G_1 and G_2.*

Proof. The CCPI-Algorithm fully respects the general algorithm framework described in section 2. Indeed lemmas 8 and 7 ensures that the recursive calls are done on the right subgraphs. The only difference is that we now choose which maximal component we extract first.   □

To analyse its complexity, we use an amortized argument that is common to many Hopcroft-like approaches, but did not appear in the original paper [12]. To our knowledge, it is due to [5].

Theorem 2. *The CCPI-Algorithm is worst case $\mathcal{O}((n + m) \log n)$ time .*

Proof. We first focus on the number of times a vertex x and a transition (y, z) may participate to EXTRACT and REMOVE. W.l.o.g., let S_1 be the size of the

subgraph of G_1 at the beginning of a recursive call of CCCIA. If a connected component of G_1 is extracted through its MCP P, then the size of the induced subgraph of P is less than or equal to $S_1/2$. This is straightforward since P has been isolated through SIS as the smallest of the two induced subgraphs. By induction, if x and (y, z) participate to many EXTRACT and REMOVE, they are contained in subgraphs whose sizes are divided by at least two at each recursive call. Therefore, they may only participate to $\log(n + m)$ EXTRACT and REMOVE calls.

Secondly, we amortized the cost of each EXTRACT and REMOVE of a path P on all the vertices and edges of the induced subgraph of P. The complexity of EXTRACT and REMOVE (lemmas 8 and 7)) for extracting a set X out of a graph G is $|X| + d(X)$. We amortize the cost $|X|$ over each vertex of X, and therefore a vertex $x \in X$ participates for a constant amount of time. The term $d(X)$ is amortized over the edges. As an edge (x, y) may be visited when considering x and when considering y, an edge can be visited only twice and therefore participates for a constant amount of time.

In consequence, each vertex or each edge costs at most $\log(n + m)$. This leads to an overall complexity of $\mathcal{O}((n + m) \log(n + m))$. As in the worst case, $m = \mathcal{O}(n^2)$, the final complexity is $\mathcal{O}((n + m) \log n)$ worst case time. $\square$

The space complexity is $O(n + m)$, since the two MCPFs are space linear in $n + m$ and that the recursive call of CCCIA algorithm can be managed with a list of at least $O(\log(n + m))$ pointers on the MCPFs.

5   Conclusion

We presented an $\mathcal{O}((n+m) \log n)$ worst case time and $\mathcal{O}(n+m)$ space algorithm for solving CCP on interval graphs. For this kind of graph, our algorithm is both faster and simpler than the actual best algorithm for general graphs, running in $\mathcal{O}(n \log n + m \log^2 n)$ [8]. Our algorithm combines an Hopcroft partitioning approach with a maintenance of a spanning clique forest decomposition of the two graphs. Designing faster algorithms or proving a lower bound for CCP remains open, on interval and general graphs. It is also worthwhile to notice that even on chordal graphs the general upper bound can still not be improved.

References

1. M.-P. Béal, A. Bergeron, and M. Raffinot. Gene Teams and Hopcroft's Partionning Framework. 2003. Submitted.
2. A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. In *Workshop on Algorithms in Bioinformatics (WABI)*, number 2452 in Lecture Notes in Computer Science, pages 464–476. Springer-Verlag, Berlin, 2002.
3. H. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2), 1993.
4. K.S. Booth and G.S. Lueker. Testing for the consecutive ones properties, interval graphs and graph planarity using pq-tree algorithm. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

5. A. Cardon and M. Crochemore. Partitioning a graph in $O(|A|\log_2|V|)$. *Theoretical Computer Science*, 19(1):85–98, 1982.
6. D.G. Corneil, S. Olariu, and L. Stewart. The ultimate interval graph recognition algorithm? In *Proceedings of the ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 175–180, 1998.
7. G.A. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Uni. Hamburg*, 25, 1961.
8. A.-T. Gai, M. Habib, C. Paul, and M. Raffinot. Identifying Common Connected Components of Graphs. *Technical report*, (RR-LIRMM-03016), 2003. `http://www.lirmm.fr/~paul`.
9. P. Galinier, M. Habib, and C. Paul. Chordal graphs and their clique graph. In M. Nagl (Ed.), editor, *Graph-Theoretic Concepts in Computer Science, WG'95*, volume 1017 of *Lecture Notes in Computer Science*, pages 358–371, Aachen, Germany, June 1995. 21st Internationnal Workshop WG'95, Springer.
10. M. Habib, R. McConnell, C. Paul, and L. Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.
11. J. Holm, K. De Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.
12. J. E. Hopcroft. An $n\log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
13. C.G. Lekkerkerker and J.C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.
14. TERAPROT project. `http://www.infobiogen.fr/services/Teraprot/`.