Fast algorithms for identifying maximal common connected sets of interval graphs

Fabien Coulona, Mathieu Raffinotb,*

aLaboratoire d’Informatique Fondamentale et Appliquée de Rouen (LIFAR), Faculté des Sciences, place Emile Blondel, 76821 Mont-Saint-Aignan, France
bCNRS-Poncelet Laboratory, Independent University of Moscow, 11 street, Bolchoï Vlassievski, 119 002 Moscow, Russia

Received 19 May 2004; received in revised form 23 December 2005; accepted 15 February 2006

Available online 19 April 2006

Abstract

Given a family of interval graphs \( F = \{G_1 = (V, E_1), \ldots, G_k = (V, E_k)\} \) on the same vertices \( V \), a set \( S \subseteq V \) is a maximal common connected set of \( F \) if the subgraphs of \( G_i, 1 \leq i \leq k, \) induced by \( S \) are connected in all \( G_i \) and \( S \) is maximal for the inclusion order. The maximal general common connected set for interval graphs problem (gen-CCPI) consists in efficiently computing the partition of \( V \) in maximal common connected sets of \( F \). This problem has many practical applications, notably in computational biology. Let \( n = |V| \) and \( m = \sum_{i=1}^{k} |E_i| \). For \( k \geq 2 \), an algorithm in \( O((kn + m) \log n) \) time is presented in Habib et al. [Maximal common connected sets of interval graphs, in: Combinatorial Pattern Matching (CPM), Lecture Notes in Computer Science, vol. 3109, Springer, Berlin, 2004, pp. 359–372]. In this paper, we improve this bound to \( O(kn \log n + m) \). Moreover, if the interval graphs are given as \( k \) sets of \( n \) intervals, which is often the case in bioinformatics, we present a simple \( O(kn \log^2 n) \) time algorithm.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Comparative genomics; Interval graph; Common connected set; Umbrella-free ordering

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 \( E_X = E \cap \{(u, v) \mid u \in X, v \in X\} \).

A connected component of \( G = (V, E) \) is a set \( S \subseteq V \) that is connected and that cannot be augmented with other vertices.

Let \( F \) be a family of graphs on (or restricted to) the same vertices, say \( F = \{G_1 = (V, E_1), \ldots, G_k = (V, E_k)\} \). We denote \( n = |V| \) and \( m = \sum_{i=1}^{k} |E_i| \). A connected set \( X \subseteq V \) of \( F \) is such that each \( G_i[X] \) is connected.

Definition 1. A set \( S \subseteq V \) of vertices is a maximal common connected set of a family \( F = \{G_1 = (V, E_1), \ldots, G_k = (V, E_k)\} \) of graphs if \( S \) is a maximal, with respect to the inclusion order, connected set of \( F \).

* Corresponding author. Fax: +7 095 2916501.
E-mail address: mathieu@raffinot.net (M. Raffinot).
The maximal common connected sets of $V$ obviously form a partition of $V$ and the general common connected problem (gen-CCP), defined in [4], is to efficiently compute this partition. This problem arises in comparative genomics for the identification of clusters of genes/proteins/domains that are closely placed on chromosomes of several species, considering specific distances that take into account biological properties. The notions of common intervals [14,8] and gene teams [1] are two specific solutions designed for simple distances induced by positions on a linear model of chromosomes. The resulting clusters can be identified, respectively, in $O(kn)$ and $O(kn \log^2 n)$ worst case time. Gen-CPP solves the problem for three-dimensional distances in $O(n \log n + m \log^2 n)$ time [4]. However, many of the distances appearing in computational biology are in fact given by interval graphs. This is mainly due to the fact that large biological contigs are built through interval graphs of smaller sequences (cDNA, ESTs, etc.). Solving gen-CCP efficiently on interval graphs is a real challenge, that has already been addressed in [7]. This restriction is called general common connected problem on intervals (gen-CCPI).

Formally, 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 [10]. Fig. 1 gives an example of such a graph. Let $F = \{G_1 = (V, E_1), \ldots, G_k = (V, E_k)\}$ be a family of interval graphs. Fig. 2 shows an example of such a family and the partition of its vertices into maximal common connected sets. If the family $F$ only contains a single graph, the problem is reduced to searching for connected components and is efficiently solved in $O(n + m)$ time. Otherwise, gen-CCPI is solved in [7] in $O((kn + m) \log n)$ time. The algorithm maintains a dynamic representation of connected components for all the graphs of the family using forests of maximal clique paths. This representation is then combined with an Hopcroft-like partitioning framework [9,11,2,5,12].

In this paper we propose a new $O(kn \log n + m)$ worst case time algorithm which is, in any case, faster than that of [7]. The algorithm uses a new dynamic representation of the connected components of an interval graph that is combined with a highly simplified Hopcroft-like partitioning framework. The representation is based on a specific vertex ordering that verifies the umbrella-free property [13,3]. Compared to the algorithm of [7], the partitioning framework is very similar but simpler. It resembles the original gene teams identification algorithm [1]. However, our new representation is more difficult to manage and the partitioning operation is more delicate.

Moreover, as an interval graph represents a set of intervals on the real line, it may therefore be given as a set of $n$ intervals instead of as a full interval graph (see Fig. 1). This is often the case in computational biology. Building the corresponding graph is $O(n + m)$, and gen-CCPI may also be solved for a family of $k$ sets of $n$ intervals with our new algorithm in $O(kn \log n + m)$ time. However, in this specific case, we exhibit a simple algorithm solving gen-CCPI in $kn \log^2 n$, independently of $m$. As $m$ may be counted (without building the real graph) on $n$ intervals in $O(n)$, the algorithm is of use as soon as $m = \Omega(n \log^2 n)$. 

Fig. 1. A set of intervals and its corresponding interval graph.

Fig. 2. Two interval graph families on the same vertices and the corresponding maximal common connected sets.
This paper is thus organized as follows. In the next Section 2, we set forth the general framework of CCPI that is very similar to that of [7]. We then present the simple $k n \log^2 n$ algorithm if the family is given as $k$ sets of $n$ intervals. Finally, the following Section 4 is devoted to our more involved $O(kn \log n + m)$ time algorithm.

2. Recursive partitioning framework

We now present a general framework for our two algorithms solving gen-CCPI. This framework is directly adapted from that of [7], to which the reader is referred for details. The framework is first presented for a family of two graphs only, solving the problem CCPI (non-generic). It is then directly extended for solving gen-CCPI.

**Lemma 1** (Habib et al. [7]). Let $G_1$ and $G_2$ be graphs on the same vertex set $V$ and let $C$ be a connected component of $G_1$ or $G_2$ distinct from $V$. Then $\text{CCPI}(G_1, G_2) = \text{CCPI}(G_1[C], G_2[C]) \cup \text{CCPI}(G_1[V \setminus C], G_2[V \setminus C])$.

**Proof.** Let $S$ be a maximal common connected set. By definition $S$ is connected in $G_1$. Since $C$ is a connected component, $S$ is either included in $C$ or in $V \setminus C$. 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 $G_2[C]$, either a maximal common connected set of $G_1[V \setminus C]$ and $G_2[V \setminus C]$. \qed

Throughout this paper, graph $G$ is accessible through a representation generically denoted by $R(G)$.

A simple paradigm for a recursive algorithm derives from Lemma 1. The inputs are two representations $R(G_1)$ and $R(G_2)$ of $G_1$ and $G_2$ that are interval graphs on the same vertex set $V$. Then the algorithm searches for a connected component of $G_1$ or $G_2$ distinct from $V$. This is performed by a procedure named small connected component (SCC) applied on the two representations. If such a component $L \subset V$ of graph $i$ exists, according to Lemma 1, two recursive calls are launched on (a) the representations $R'_i = R(G_i[L])$ and $R''_i = R(G_i[V \setminus L])$ of the subgraphs induced by $L$ and (b) the representations $R''_i = R(G_i[V \setminus L])$ and $R''_j = R(G_j[V \setminus L])$ of the subgraphs induced by $V \setminus L$. These subgraphs are obtained through a procedure EXTRACT applied to $(L, i)$. The complete generic pseudo-code CCPI-ALGORITHM is shown in Fig. 3.

**Lemma 2** (Habib et al. [7]). CCPI-ALGORITHM computes the partition of the vertices of a pair of interval graphs into maximal common connected sets.

**Proof.** CCPI-ALGORITHM ends since the recursive calls are launched on strict subsets and that the recursive calls stop when both graphs are connected (even reduced to a single element). The correctness of the algorithm directly derives from Lemma 1. \qed

This approach extends directly to identify maximal common connected sets in a family of more than two interval graphs, solving the complete gen-CCPI.
Lemma 3 (Habib et al. [7]). Let $G_1, \ldots, G_k$ be interval graphs on the same vertex set $V$ and let $C \neq V$ be a connected component of some $G_i$. Then gen-CCPI($G_1, \ldots, G_k$) = gen-CCPI($G_1[C], \ldots, G_k[C]$) U gen-CCPI($G_1[V\setminus C], \ldots, G_k[V\setminus C]$).

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, $S$ is either included in $C$ or in $V\setminus C$. It follows that any maximal common connected set of $G_1, \ldots, G_k$ is a maximal common connected set, either of $G_1[C], \ldots, G_k[C]$, or of $G_1[V\setminus C], \ldots, G_k[V\setminus C]$. \hfill $\Box$

The modifications of the algorithm are straightforward. Procedure SCC now identifies a connected component $L$ of one of the graphs $G_i$, $1 \leq i \leq k$. If this component is not $V$ itself, it is then extracted from all the $G_i$ and the algorithm recurses on the representations $R(G_i[V\setminus L]), 1 \leq i \leq k$ on one side and on $R(G_i[L]), 1 \leq i \leq k$ on the other side.

In the remainder of the paper, we present our algorithms for two interval graphs only. Their extension to $k > 2$ interval graphs is direct.

Complexity. The general framework CCPI-Algorithm (Fig. 3) may lead to many different algorithms, which are not always efficient. Without ad hoc procedures SCC and EXTRACT, this recursive paradigm yields to a $\Theta(n(n + m))$ worst case time algorithm.

To accelerate the algorithms, we impose the procedure SCC($R_1, R_2$) to identify a small connected component in one of the two graphs represented by $R_1$ and $R_2$. Small means that the number of vertices in the resulting connected component has to be less than or equal to half of the number of vertices in the original graph. If such a connected component exists, SCC returns a list $L$ of its vertices together with the identifier of the graph in which the component has been found. Otherwise, it returns an empty list.

(Note that the aim of SCC is different from the aim of SIS algorithm of [7]. Procedure SIS searches for a small-induced subgraph, taking into account vertices and edges, while SCC only considers the number of vertices.)

The purpose of procedure EXTRACT($L, i$) is to split each graph $G_1$ and $G_2$ in two separate graphs with respect to the list of vertices $L$ and the number $i$ of the graph of which $L$ is a connected component. It then returns the quadruple of induced graph representations ($R(G_1[L]), R(G_2[L]), R(G_1[V\setminus L], R(G_2[V\setminus L]))$. To obtain fast algorithms, the time complexity of EXTRACT($L, i$) should be proportional to $|L|$.

We now present two distinct representations of interval graphs. The first one is based on sets of intervals and is associated to procedures SCCs and EXTRACTs. The second is a specific interval graph representation and is associated to SCCg and EXTRACTg.

3. A simple $n \log^2 n$ time algorithm for sets of intervals

In this section, we consider the case where the family of interval graphs is given as $k$ sets of $n$ intervals. We exhibit a simple representation of each graph that allows us to solve CCPI ($k = 2$) in $O(n \log^2 n)$, and therefore (see Lemma 3), gen-CCPI in $O(kn \log^2 n)$.

Let $I = \{i_1 = [x_1, y_1], i_2 = [x_2, y_2], \ldots, i_n = [x_n, y_n]\}$ be a set of $n$ intervals on the real line. The representation of the interval set is a sequence of interval (say $i_j$) left ends (written $i_j$) and right ends (written $i_j$). This sequence represents the interval extremities encountered when reading the interval set on horizontal axis. If several intervals have the same extremities, the order is chosen arbitrarily. More formally, the sequence is written as a doubly linked list $A_I$. At the beginning of SCCs, $A_I$ can be considered as a table $A_I[1, \ldots, 2n]$, where $A_I[j] = i_z$ (resp., $A_I[j] = i_z$) if $j$ is the position of the left (resp., right) end of the interval $i_z$. Fig. 4 gives an example of such a representation of the interval set of Fig. 1. The doubly linked list structure is of use only in procedure EXTRACT. Therefore, for simplicity, we explain procedure SCCs considering $A_I$ as a table.

The table $A_I$ may be computed from the set of interval in $O(n \log n)$ time by sorting the extremities of the intervals in $I$. 

---

Fig. 4. The doubly linked list representation $A_I$ of the set of intervals of Fig. 1.
3.1. Small connected component identification

Let us consider first a single set of intervals $I$, accessible through its representation $A_I$. In order to identify a small connected component in $I$, we parse the table $A_I$ using simultaneously two indices $pb$ (resp., $pe$) starting from the beginning to the center of $A_I$ (resp., from the end down to the center).

At the same time we maintain two counters $cb$ and $ce$, respectively, associated with $pb$ and $pe$. They store the number of unclosed intervals encountered, respectively, in $A_I[1,..., pb]$ and $A_I[pe, ..., 2n]$. At the beginning, $pb = 1$, $cb = 1$, $pe = 2n$, $ce = 1$.

It is obvious to see that $A_I[1,..., pb]$ is a set of connected components iff $cb = 0$. Symmetrically, $A_I[pe, ..., 2n]$ is a set of connected components iff $ce = 0$.

Maintaining the two counters when moving the indices is easy. If $A_I[pb] = i$, $cb$ increases. Otherwise, $A_I[pb] = \tilde{i}$ and $cb$ decreases. Symmetrically, if $A_I[pe] = i$, $ce$ decreases. Otherwise, $A_I[pe] = \tilde{i}$ and $ce$ increases.

As soon as $cb = 0$ or $ce = 0$, a connected component is found. As the two indices run alternately position by position, this component is assured to be represented by a table of half the size of the original table, that is of size less than or equal to $n$.

If the two indices cross without having identified such a component, then the whole set of intervals is a single connected component.

Now, to identify a small class over the two sets of intervals $I_1$ and $I_2$, we maintain 2 indices $pb$ and $pe$ moving alternatively, and four counters $cb_1$, $cb_2$, $ce_1$, $ce_2$ associated in the same way. If the process stops before the pointers cross, that is if one of the counters is equal to 0, a connected component is found. Its representation size is less than or equal to $n$. Otherwise, the two graphs are connected and the vertices form a maximal common connected set. The pseudo-code of SCCs is given in Fig. 5.

3.2. Extraction

Once a small connected component $L$ (of size $p = |L|$) of $I$ (say w.l.o.g. $I_1$) has been returned by SCCs, it is extracted by EXTRACTs out of the doubly linked lists $A_{I_1}$ and $A_{I_2}$. This is simply done by thus:

(a) cutting the doubly linked lists $A_{I_1}$ into: (i) $A'_{I_1}$ representing the intervals of $L$ and; (ii) $A''_{I_1}$ representing the remaining intervals. This can be done in $O(1)$ time if the position of the cut in $A_{I_1}$ is saved after SCCs;
Fig. 6. An umbrella-free representation of our example graph. The dash arrows represent the doubly linked list. Note that the right neighbors of each vertex are sorted from nearest to furthest (not shown in the figure).

(b) extracting one by one the left and right ends of each interval \( i \) of \( L \) out of \( A_I \). As \( A_I \) is a doubly linked list, this can be done in \( O(p) \) time touching only the intervals of \( L \) and their direct neighbors. Then \( A'_I \) is the remaining doubly linked list;
(c) sorting the left and right ends of the intervals of \( L \) according to the order of \( I_2 \) and building the new corresponding doubly linked list \( A'_I \). This can be done in \( O(p \log p) \) time.

These observations prove the following Property 1.

**Proposition 1.** Algorithm \( \text{EXTRACT}(L, i) \) has a worst case time complexity of \( O(p \log p) \), where \( p = |L| \).

### 3.3. Complexity

The complexity is calculated through an amortized approach that is usually of use when dealing with a Hopcroft partitioning framework [9,2,6,5]. It is based on the following observation.

Let \( i \) be a left (resp., right) end of an interval \( i \). If \( i \) is used in a list \( L_0 \) to perform an extraction through \( \text{EXTRACT} \) and later re-used in another list \( L_1 \) in \( \text{EXTRACT} \), then \( |L_1| \leq |L_0|/2 \). This is obviously true since SCCs ensures that the returned connected component on a set of \( n \) intervals (if it exists) has a size less than or equal to \( n/2 \). Consequently, a specific end \( i \) can be used in an extraction only \( \log n \) times.

Each extraction of a list \( L \) of size \( p = |L| \) can be performed in \( O(p \log p) \) (Proposition 1). This is equivalent to the fact that each element \( i \) participates at maximum for \( O(\log p) \). The contribution of an element \( i \) to the final complexity is therefore at maximum \( \sum_{i=1}^{\log(n)} \log(n/2^i) \) which is \( O(\log^2 n) \). As there are \( 2n \) left and right ends, the global complexity is \( O(n \log^2 n) \) worst case time.

### 4. A more involved \( m + n \log n \) algorithm

In this section, we make use of the particular characterization of interval graphs that was mentioned in [13,3]. Given an interval graph \( G = (V, E) \), there is a linear order \( < \) on the set of vertices \( V \) such that, for any choice of three vertices \( u, v, w \in V \) with \( u < v \) and \( v < w \),

\[
(u, w) \in E \quad \text{implies} \quad (u, v) \in E.
\]  

Such an ordering of vertices is said to be *umbrella-free*. In particular, if the graph is given as a collection of intervals, then the ordering of interval left end positions satisfies this property.

A graph \( G \) is now stored as the doubly linked list \( UF_G \) of its vertices. The list is ordered with respect to \( < \). Each vertex \( v \) holds a doubly linked list \( \overrightarrow{v} \) of its right neighbors and a doubly linked list \( \overleftarrow{v} \) of its left neighbors. The list \( \overrightarrow{v} \) is sorted from the nearest to the furthest right neighbor. The sorting is initially performed in linear time since the right neighbors of a vertex form an interval in \( UF_G \) according to relation (1). Fig. 6 shows a schematic representation of \( UF_G \) for our example graph. The linear order \( < \) can be computed in \( O(m + n) \) time using the recognition algorithm of [3]. Therefore, the representation \( UF_G \) can be obtained in \( O(m + n) \) time.
Proposition 2. An umbrella-free representation of a graph G is a concatenation (in any order) of all its connected component umbrella-free representations.

Proof. Let UF be an umbrella-free representation of G, and let us prove that vertices of two distinct connected components are not interleaved in UF. Indeed, let be three distinct vertices \( v_1 \prec v_2 \prec v_3 \) such that \( v_1 \) and \( v_3 \) are in a same connected component \( CC \). Since they are connected, there exist \( v'_1 \) and \( v'_3 \) in \( CC \), such that \( v'_1 \prec v_2 \prec v'_3 \) and \( v'_3 \in \rightarrow v'_1 \).

Then the umbrella-free property gives \( v_2 \in \rightarrow v'_1 \) and \( v_2 \in CC \).

This proves that UF is a concatenation of the umbrella-free representations of its connected components. Trivially, the ordering of these representations has no impact on the umbrella-free property. □

The general framework is slightly modified. Each vertex in a graph G holds an identifier \( \text{ident}(v) \) of the current sub-problem. That is, before G is split into \( G[L] \) and \( G[V \setminus L] \), vertices in L receive a new arbitrary identifier that has not already been assigned during the recursive algorithm. Edges that link a vertex in L to a vertex in \( V \setminus L \) are not physically deleted during the extraction. They are simply detected later as non-existing edges because their identifiers are different.

Definition 2. An edge \((v_1, v_2) \in E\) is said to be a phantom edge if \( \text{ident}(v_1) \neq \text{ident}(v_2) \).

The two algorithms SCCg and EXTRACTg manipulate doubly linked lists through the following classical operators. Let \( l \) be such a list, \( \text{head}(l) \) (resp., \( \text{tail}(l) \)) is the first (resp., last) element of \( l \) or NIL if \( l \) is empty. Let \( v \) be an element of \( l \), \( \text{next}_l(v) \) (resp., \( \text{prev}_l(v) \)) points to the successor (resp., predecessor) of \( v \) in \( l \), or NIL if \( v \) is the last (resp., first) element of \( l \).

4.1. Small connected component identification

The principle of SCCg is rather similar to SCCx. Given the two graphs \( G_1 \) and \( G_2 \) we consider four parsers that explore simultaneously the two graph representations \( UF_1 = UF_{G_1} \) and \( UF_2 = UF_{G_2} \), each one starting at one of their four extremities, searching for a small connected component.

The first parser that parses a small connected component, stops the other parsers and returns this connected component.

We have to define two types of parsers: a left (resp., right) parser is one that can start from the left (resp., right) end of a representation. Figs. 8 and 9 give their pseudo-code. When arriving at a command synchronize, a parser waits until the three other parsers arrive at a command synchronize.

Left-to-right parser: The LEFT-PARSER performs a left-to-right parsing of the graph. Fig. 7 illustrates its principle. The arrow represents vertex v. Marked vertices are black. At each step we parse all right neighbors of v from the furthest to the nearest, but we stop as soon as we meet a marked vertex, so that vertices are visited at most twice.

Without considering any synchronization, we prove the following Lemma 4.

Lemma 4. Algorithm LEFT-PARSER returns the small component L of G containing the first vertex of UFg if such a component exists. Otherwise, it returns \( \emptyset \) (Fig. 8).
**Proof.** Let CC be the connected component that contains the leftmost element of UF\(_G\). We prove below that all marked vertices belong to CC and that all elements of CC are eventually marked by the algorithm if the algorithm stops at line 15. As each marked element is stored in list L (line 2 and line 12), L contains all elements of CC at the end of algorithm.

We first prove by induction that all marked vertices are in CC. The leftmost vertex of UF\(_G\) is marked at the beginning, and so does v\(_2\). Each time a new vertex v\(_2\) is marked (line 11), it is connected through a non-phantom edge to a vertex v (line 5) that is marked (line 4). By induction, v belongs to CC, and so does v\(_2\).

It remains to prove that all vertices of CC are eventually marked. To this end we establish the following invariant (I) for the loop 4–14: all elements of CC positioned to the left of a marked vertex are marked.

Invariant (I) is trivially true at line 3. Assume it is true at line 5. Let v\(_3\) be the rightmost marked vertex of \(\overrightarrow{v}\). The loop 6–13 marks all elements of \(\overrightarrow{v}\) positioned between v\(_3\) and v\(_2\), respectively, outer and inner ends. The umbrella-free property (Eq. (1)) ensures that all vertices between v\(_3\) and v\(_2\) are contained in \(\overrightarrow{v}\). Therefore, all elements between v\(_3\) and v\(_2\) are marked. By the induction hypothesis, all elements to the left of v\(_3\) are marked, and consequently invariant (I) is kept.

We complete the correctness proof by supposing there remains unmarked vertices in CC at line 15. Let z be the leftmost such vertex. Since the leftmost vertex of UF\(_G\) is marked at the beginning, z is not the left end of its connected component, so that Lemma 6 states the existence of a vertex w in CC positioned at the left of z, with z \(\in\) \(\overrightarrow{w}\), and w is marked. When the algorithm gets out of loop 4–14, vertex v is unmarked so that v is positioned at the right of w (by invariant (I)). Consequently, an iteration of the algorithm has been executed with v = w, and necessarily, z has been marked during this iteration. By contradiction, all vertices of CC are marked. □

**Lemma 5.** Algorithm LEFT-PARSER runs in O(l + k) time, where l is the number of distinct vertices in the connected component containing the leftmost element of UF\(_G\) and where k is the number of deleted phantom edges.

**Proof.** The number of vertices that are marked by the algorithm is l (see the proof of Lemma 4). For each marked vertex v, the loop 6–13 runs in time proportional to the number of unmarked vertices in \(\overrightarrow{v}\) plus a single test to stop the loop if a marked vertex is encountered (line 6). Each unmarked vertex is then marked (line 11). At the same time, each encountered phantom edge is deleted (lines 7–9). The total number of touched vertices for the loop 6–13 over all iterations of loop 4–14 is thus the total number of marked vertices plus the number of deleted phantom edges, say k.
The loop 4–14 considers all marked vertices once, and consequently the total number of operations on vertices is twice the number of marked vertices plus $k$. □

**Right-to-left parser.** Procedure RIGHT-PARSER proceeds on a right-to-left parse of the umbrella-free representation of the graph $G$, using the following lemma.

**Lemma 6.** Let $G$ be an interval graph and $UF_G$ its umbrella-free representation. A vertex $v$ is the left end of a connected component if and only if $\leftarrow v$ is empty.

**Proof.** If $\leftarrow v$ is not empty, it is obvious that $v$ is not a left end of a connected component, since it is connected to vertices at its left. Suppose now that $v$ is not at the left end of a connected component. There exist two vertices $z$ and $z'$ such that $z \prec v \prec z'$, $z \neq v$, and $z' \in \leftarrow z$. The umbrella-free property (1) directly implies that $v \in \leftarrow z$, and thus $\leftarrow v \neq \emptyset$. □

Note that the corresponding assertion is false for right ends of connected components. Thus, for the right-to-left parse we simply follow the doubly linked list of vertices until a vertex with no left neighbor is found. The pseudo-code of RIGHT-PARSER is given in Fig. 9.

**Lemma 7.** Algorithm RIGHT-PARSER returns the small connected component of $G$ containing the last vertex of $UF_G$ if such a component exists. Otherwise, it returns $\emptyset$.

**Proof.** Algorithm RIGHT-PARSER searches for the left end of a connected component using Lemma 6. At the same time, it deletes the encountered phantom edges (line 5). Let $CC$ be the connected component that contains the last vertex of $UF_G$.

Elements of $CC$ and elements out of $CC$ are not interleaved in $UF_G$ (Property 2). Hence it is sufficient to prove that the algorithm stops as soon as $v$ reaches the left end of $CC$. This is ensured by Lemma 6 and by the condition of loop 3–8. □

**Lemma 8.** Algorithm RIGHT-PARSER runs in $O(l + k)$ time, where $l$ is the number of distinct vertices touched and $k$ the number of deleted phantom edges.

**Proof.** A new vertex $v$ is touched only $O(1)$ time except in the loop 4 and 5. However, in this last loop, each time $v$ is touched, a phantom edge is deleted. Therefore, RIGHT-PARSER runs in $O(l + k)$ time, where $l$ is the number of distinct vertices touched and $k$ the total number of deleted phantom edges over all iterations of loop 4 and 5. □

**Procedure SCCg:** Let $G_1$ and $G_2$ be two graphs on a same set of $n$ vertices, given by their umbrella-free representations $UF_1$ and $UF_2$. The algorithm $SCCg$ applied on $(UF_1, UF_2)$ consists of the simultaneous execution of
GetOrder($UF_G, L$)
1. While L is non empty Do
2. Let $v$ be a vertex of L
3. While $\overrightarrow{v}$ is not empty Do
4. $v$ ← an element of $\overrightarrow{v}$
5. End of while
6. $L_{cc} ← [v]$
7. Mark $v$; Remove $\overrightarrow{v}$ from L
8. While $v \neq$ NIL AND $\overrightarrow{v} \neq \emptyset$ Do
9. $L_{temp} ← \emptyset$
10. $v_2 ← tail(\overrightarrow{v})$
11. While $v_2$ unmarked Do
12. $L_{temp} ← v_2, L_{temp}$
13. Mark $v_2$; Remove $v_2$ from L
14. $v_2 ← prev_{\overrightarrow{v}}(v_2)$
15. End of while
16. $L_{cc} ← L_{cc}, L_{temp}$
17. $v ← next_{L_{cc}}(v)$
18. End of while
19. Store $L_{cc}$ as a sorted connected component
20. End of while
21. Return The list of connected components.

Fig. 10. Sorting each connected component induced by $L$ in $G$ represented by its umbrella-free representation $UF_G$.

RIGHT-PARSER($UF_1$), RIGHT-PARSER($UF_2$), LEFT-PARSER($UF_1$), LEFT-PARSER($UF_2$). It returns the set $L$ given by the first ending parser, together with the identifier of its graph. Note that if $L$ is empty, the synchronization mechanism guarantees that all parsers have already touched more than $n/2$ distinct vertices. In this case, no small class is found, which means that $G_1$ and $G_2$ are both connected.

Theorem 1. Algorithm SCCg is correct and has an $O(l + k)$ complexity, where $l$ is the size of the returned connected component, and $k$ is the number of phantom edges that have been deleted in the graph during its execution.

Proof. Lemmas 7, 8, 4 and 5 ensure that SCC returns either a small connected component including an extremity of $UF_1$ or $UF_2$ or returns $\emptyset$ if all of these connected components are formed of strictly more than $n/2$ vertices.

The connected components are not interleaved in the umbrella-free representations (Property 2), so that a connected component with more than $n/2$ vertices cannot include the two extremities of the representation unless it covers the whole representation. Hence, if the right and the left parser of a given representation $UF_G$ both return $\emptyset$, then necessarily the whole representation is connected. Hence SCC returns $\emptyset$ if and only if $G_1$ and $G_2$ are connected. □

4.2. Extraction

We now investigate how to split a graph $G$ into $G[L]$ and $G[V \setminus L]$, where $L$ is a given list of vertices.

The list $L = \{l_1, l_2, \ldots, l_h\}$ is not initially ordered with respect to the order of vertices in $UF_G$, which forbids a priori to cut the doubly linked list $UF$ in two doubly linked lists representing $G[L]$ and $G[V \setminus L]$ in time $O(h)$. However, the structure of interval graph enables us to sort each connected component induced by $L$ without involving an $O(h \log h)$ complexity. This is done by algorithm GETORDER the pseudo-code for which is given in Fig. 10. Using Property 2, this is enough to rebuild an umbrella-free representation of $G[L]$.

We assume that any edge linking a vertex in $L$ to a vertex of $V \setminus L$ is a phantom edge. Each time a phantom edge is encountered, the algorithm deletes it and continues as if the edge did not exist. For simplicity, we do not detail the management of phantom edges in the pseudo-code. For instance, $tail(\overrightarrow{v})$ returns the rightmost vertex connected to $v$ by a non-phantom edge, and NIL if no such vertex exists. The assertion $\overrightarrow{v} \neq \emptyset$ is true if there exists at least one right non-phantom edge outgoing from $v$. 
Thus, the overall worst case complexity is $O(\cdot)$.

**Proof.** At line 2, a vertex $v$ is picked. Let $CC$ be the connected component to which it belongs. We prove below that at line 16 the list $L_{cc}$ contains all vertices of $CC$ sorted according to the ordering of $UF_G$. As those vertices are removed from $L$, the iteration loop of line 1 surely outputs the list of all sorted connected components induced by $L$.

We now focus on the correctness of lines 2–16. At line 5, $v$ is the leftmost element of component $CC$ because of Lemma 6: loop 3 and 4 cannot stop unless $v$ is the leftmost element of its connected component.

We prove two invariants for loop 7–15: first (I) all elements of $CC$ positioned to the left of a marked vertex are marked, and (II) marked vertices are stored in $L_{cc}$ and sorted.

Invariant (I) is trivially true at line 6. Assume it is true at line 9. Let $v_3$ be the rightmost marked vertex of $\overrightarrow{v}$. The loop 10–13 fills $L_{temp}$ with elements of $\overrightarrow{v}$ positioned between $v_3$ and $v_2$, respectively, outer and inner ends. The umbrella-free property (Eq. 1) ensures that all vertices between $v_3$ and $v_2$ are contained in $\overrightarrow{v}$. Therefore, all elements between $v_3$ and $v_2$ are marked. By the induction hypothesis, all elements to the left of $v_3$ are marked, and consequently invariant (I) is kept.

Invariant (II) is deduced by observing that vertices contained in $L_{temp}$ are sorted accordingly to the order of $UF_G$. Before execution of line 14, elements of $L_{temp}$ are all positioned at the right of elements of $L_{cc}$ since they were not marked before entering the loop 10–13 and we have (I). Hence, when appending $L_{temp}$ to $L_{cc}$, invariant (II) is kept.

It remains to prove that all vertices of $CC$ are eventually marked. During loop 7–15, $v$ parses the whole list $L_{cc}$, so that for each vertex $v \in L_{cc}$, all vertices in $\overrightarrow{v}$ are marked. Now suppose that there exists an unmarked vertex in $CC$ at the end of the algorithm, and let $v'$ be the leftmost unmarked vertex of $CC$. Since $v'$ is not the leftmost element of $CC$ and because of Lemma 6, $v'$ has a left neighbor $v''$ in $CC$. Then $v'$ is in $\overrightarrow{v''}$ and this contradicts the assertion $v'$ is unmarked.

Together with invariants (I) and (II), this completed the correctness proof.

Let us prove the announced complexity. The loop 3 and 4 ends trivially within $|L|$ iterations. Each time the loop 10–13 is entered, an unmarked vertex of $L$ is marked, so that the loop is entered at most $|L|$ times. The loop 7–15 ends within $|L|$ iterations because $v$ parses $L_{cc}$. Without considering phantom edges, the algorithm has an $O(\mid L \mid)$ complexity. Each time a phantom edge is encountered, it is deleted. Let $k$ be the number of deleted phantom edges. Thus, the overall worst case complexity is $O(|L| + k)$ time. □
The procedure \textsc{Extract}\textsubscript{g}(L, i) splits UF\textsubscript{G\textsubscript{1}} and UF\textsubscript{G\textsubscript{2}} with respect to L. Assume w.l.o.g. that \( i = 1 \). The set L is known to be a connected component of \( G\textsubscript{1} \), and the ends of this component are known, so that the splitting of UF\textsubscript{G\textsubscript{1}} is done in constant time. The splitting of UF\textsubscript{G\textsubscript{2}} is carried out the following way:

(a) \( UF\textsubscript{G\textsubscript{2}} \leftarrow \emptyset \).
(b) \( CC \leftarrow \textsc{GetOrder}(UF\textsubscript{G\textsubscript{2}}, L) \).
(c) For each component \( CC\textsubscript{i} \) in CC, extract from UF\textsubscript{G\textsubscript{2}} the doubly linked list formed of elements of \( CC\textsubscript{i} \), and append the extracted list to \( UF\textsubscript{G\textsubscript{2}}' \).
(d) Return \( UF\textsubscript{G\textsubscript{2}} \) and \( UF\textsubscript{G\textsubscript{2}}' \).

Note that the resulting \( UF\textsubscript{G\textsubscript{2}}' \) is a correct umbrella-free representation of \( G\textsubscript{2}[L] \), though the order of connected components may have changed.

**Theorem 2.** Algorithm \textsc{Extract}\textsubscript{g}(L, i) outputs the umbrella-free representations of \((G\textsubscript{1}[L], G\textsubscript{1}[V \setminus L]), (G\textsubscript{2}[L], G\textsubscript{2}[V \setminus L])\). It runs in time \( O(|L| + k) \) where \( k \) is the number of phantom edges that have been deleted in the graph during the execution.

**Proof.** This is straightforward from Property 2 and from the complexity of algorithm \textsc{GetOrder}. \(\square\)

4.3. Complexity

We have seen that procedures \textsc{Extract}\textsubscript{g} and \textsc{SCCg} meet all specifications required by \textsc{CCPI-Algorithm}. Theorem 3 states the complexity of the whole algorithm.

**Theorem 3.** The time complexity of algorithm \textsc{CCPI-Algorithm} using procedures \textsc{Extract}\textsubscript{g} and \textsc{SCCg} is \( O(m + n \log(n)) \).

**Proof.** During the whole \textsc{CCPI-Algorithm}, the cost of the deletion of phantom edges is contained in an overall \( O(m) \) complexity. Let us do not consider these costs in the complexity of \textsc{Extract}\textsubscript{g} and \textsc{SCCg}. Using the very same proof as for Section 3.3, we obtain an \( O(n \log(n)) \) overall complexity for \textsc{CCPI-Algorithm}. Adding the cost of phantom edge deletions, we finally get \( O(m + n \log(n)) \). \(\square\)

5. Conclusion

We presented two efficient algorithms to compute the partition of vertices in maximal common connected sets of two or more interval graphs. The first solution is an efficient and simple algorithm in the case that the interval graphs are given as sets of intervals. It then runs in \( O(kn \log^2 n) \) worst case time. The second solution is a more involved \( O(m + kn \log n) \) worst case time. It takes as entry a family of \( k \) interval graphs of \( n \) vertices and \( m \) edges in total.

Note that this second algorithm may also be applied to sets of intervals, by first building the corresponding interval graphs. In this case, it is faster than the first algorithm if \( m \leq kn \log^2 n \).

Our second algorithm is, in any case, faster than the best previous algorithm by a \( \log(n) \) factor on the edges. However, obtaining a more efficient algorithm or a lower bound remains an unresolved problem.

**References**