Identification of Maximal Common Connected Sets of Interval Graphs and Tree Forests

Fabien Coulon[1] and Mathieu Raffinot[2]

[1] Laboratoire d'Informatique Fondamentale et Appliquée de Rouen (LIFAR), Faculté des Sciences, place Emile Blondel, 76821 Mont-Saint-Aignan. France.
Fabien.Coulon@univ-rouen.fr

[2] CNRS - Laboratoire Génome et Informatique, Evry, and Laboratoire Dynamique et Organisation des Génomes, Ecole Normale Supérieure, Paris. France.
raffinot@genopole.cnrs.fr

Abstract. Given a family of graphs $F = \{G_1 = (V, E_1), \dots, G_k = (V, E_k)\}$ on the same vertices V, a set $S \subset V$ is a maximal common connected set of F if for each $1 \leq i \leq k$, the subgraph of G_i induced by S is connected, and S is maximal for the inclusion order. The General Common Connected Set Problem (Gen-CCP) is to efficiently compute the partition of V in maximal common connected sets of F. In this paper, we consider solving Gen-CCP for two specific families of graphs. When all G_i are interval graphs, the problem (Gen-CCPI) 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.*, 2004. In this paper, we propose a simple $O(m + kn \log^2 n)$ algorithm, that actually runs in $O(kn \log^2 n)$ if the graph is given as k sets of n intervals, which is often the case in bioinformatics. Secondly, we consider Gen-CCP over tree forests (Gen-CCPT). Most efficient algorithms for solving Gen-CCP are structured over a (sometimes hidden) Hopcroft's partioning framework. In the case all G_i are tree forests, we propose a new competitive approach based on range searching.

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 $S \subset V$ that is connected and that can not be augmented with other vertices.

Let F be a family of graphs on (or restricted to) the same vertices, say $F = \{G_1 = (V, E_1), \dots, G_k = (V, E_k)\}$. We denote $n = |V|$ and $m = \sum_{i=1}^{k} |E_i|$. A connected set $X \subset V$ 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 $F = \{G_1 = (V, E_1), \dots, G_k = (V, E_k)\}$ of graphs if S is a maximal, with respect to the inclusion order, connected set of F.*

The *maximal common connected sets* of V obviously form a partition of V and the *General Common Connected Problem* (Gen-CCP), defined in [3], 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 3-dimensional distances in $O(n \log n + m \log^2 n)$ time [3]. 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 intervals graphs is a real challenge, that has already been adressed in [6]. This restriction is called General Common Connected Problem on Intervals (gen-CCPI).

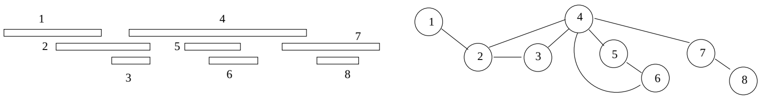

Fig. 1. A set of intervals and its corresponding interval graph.

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]. Figure 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. 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 [6] 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 (see Figure 2). This representation is then combined with an Hopcroft-like partitioning framework [9, 13, 2, 5].

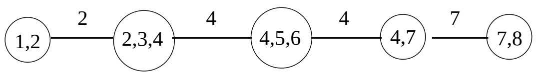

Fig. 2. Representation of our interval graph example using maximal clique paths.

In this paper, we first propose a new $O(kn \log^2 n + m)$ worst case time algorithm. Our approach is to first "realize" the interval graphs as sets of intervals. This can be done $O(kn+m)$ with many algorithms [4]. We then exhibit a simple algorithm solving gen-CCPI in $kn \log^2 n$. This algorithm uses a simple dynamic representation of the connected components of an interval graph that is combined with a highly simplified Hopcroft like partitioning framework. In a sense, it resembles the original gene teams identification algorithm [1] that is the origin of CCP. Compared to the algorithm of [6], our algorithm is faster as soon as $m = \Theta(n \log n)$, which is quite always true in practice. Moreover, notice that if the interval graphs are given as sets of n intervals instead of a full interval graph (see Figure 1), our algorithm runs in $kn \log^2 n$, independently of m.

We secondly consider Gen-CCP on tree forests instead of interval graphs. This case can be solved in $O(kn \log n)$ time using an easy simplification of the general approach of [6]. The $\log n$ factor directly derives from the Hopcroft's partitioning framework and is a bottleneck for all CCP like algorithms. In order to achieve linear complexity (if possible), we need to overcome this obstacle by another approach. We therefore present a new algorithm solving CCP on tree forests that is not based on a partitioning framework, but still remains in $O(kn \log n)$. This new algorithm is based on integer range searching [11, 12].

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 [6]. We then present our new simple $O(m + kn \log^2 n)$ algorithm for interval graphs. Finally, the following Section 4 is devoted to our new approach solving Gen-CCP on tree forests.

2 Recursive partitioning framework

We now present a general framework for our algorithm solving gen-CCPI. This framework is directly adapted from that of [6], 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 ([6]). *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 $CCPI(G_1, G_2) = CCPI(G_1[C], G_2[C]) \cup CCPI(G_1[V \backslash C], G_2[V \backslash C])$.*

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 $V \backslash C$. 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 $G_2[C]$, either a maximal common connected set of $G_1[V \backslash C]$ and $G_2[V \backslash C]$. □

Throughout this paper, a graph G is accessible through its representation $R(G)$ as a list of intervals.

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 $R'_1 = R(G_1[L])$ and $R'_2 = R(G_2[L])$ of the subgraphs induced by L and (b) the representations $R''_1 = R(G_1[V \backslash L])$ and $R''_2 = R(G_2[V \backslash L])$ of the subgraphs induced by $V \backslash L$. These subgraphs are obtained through a procedure Extract applied to (L, i). The complete generic pseudo-code CCPI-Algorithm is shown in Figure 3.

CCPI-Algorithm$(R_1 = R(G_1), R_2 = R(G_2))$
$(L, i) \leftarrow \mathrm{SCC}(R_1, R_2)$
if $L \neq \emptyset$ **then**
  $[(R'_1, R'_2), (R''_1, R''_2)] \leftarrow$ Extract$(L, i))$
  CCPI-Algorithm(R'_1, R'_2)
  CCPI-Algorithm(R''_1, R''_2)
else The vertices of G_1 form a maximal connected common set

Fig. 3. Generic framework CCPI-Algorithm for identifying maximal connected sets of two interval graphs.

Lemma 2 ([6]). CCPI-Algorithm *computes the maximal common connected sets partition of the vertices of a pair of interval graphs.*

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. □

This approach extends directly to identify maximal common connected sets in a family of more than two intervals graphs, solving the complete gen-CCPI.

Lemma 3 ([6]). *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$(G_1[C] \ldots G_k[C]) \cup$ gen-CCPI$(G_1[V \backslash C] \ldots G_k[V \backslash C])$*

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

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 $R(G_i[V \backslash L]), 1 \leq i \leq k$ 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 (Figure 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 [6]. 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 separated 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 a representations of interval graphs, based on sets of intervals and is associated to procedures SCCs and EXTRACTs.

3 An $n \log^2 n$ time algorithm for sets of intervals

In this section, we consider a family of interval graphs 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_1) left end (written i_1) and right end (written $\overline{i}_1$). This sequence roughly 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 $A_I[1 \cdots 2n]$, where $A_I[j] = i_\alpha$ (resp. $A_I[j] = \overline{i}_\alpha$) if j is the position of the left (resp. right) end of the interval i_α. Figure 4 gives an example of such a representation of the interval set of Figure 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 intervals in $O(n \log n)$ time by sorting the extremities of the intervals in I.

$$A_I \,\big|\, 1\ 2\ \overline{1}\ 3\ 4\ \overline{2}\ \overline{3}\ 5\ 6\ \overline{5}\ \overline{6}\ 7\ \overline{4}\ 8\ \overline{8}\ \overline{7}$$

Fig. 4. The doubly linked list representation A_I of the set of intervals of Figure 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, ce respectively associated with pb and pe. They store the number of unclosed intervals encountered respectively in $A_I[1 \cdots pb]$ and $A_I[pe \cdots 2n]$. At the beginning, $pb = 1$, $cb = 1$, $pe = 2n$, $ce = 1$.

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

Maintaining the two counters when moving the indices is easy. If $A_I[pb] = i_\alpha$, cb increases. Otherwise, $A_I[pb] = \overline{i}_\alpha$ and cb decreases. Symmetrically, if $A_I[pe] = i_\alpha$, ce decreases. Otherwise, $A_I[pe] = \overline{i}_\alpha$ 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 4 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 SCC*s* is given in Figure 5.

3.2 Extraction

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

(b) extracting one by one the left and right ends of each interval i of L out of A_{I_2}. As A_{I_2} is a doubly linked list, this can be done in $O(p)$ time touching only the intervals of L and their direct neighbours. Then A''_{I_2} is the remaining doubly linked list.

SCC$s(A_{I_1}[1 \ldots 2n]), A_{I_2}[1 \ldots 2n])$

$pb \leftarrow 2; pe \leftarrow 2n - 1;$
for $j \in \{1, 2\}$ **do** $cb_j \leftarrow 1;\ ce_j \leftarrow 1$
$(L, i) \leftarrow (\emptyset, 0)$
$ConCompFound \leftarrow$ False
while $ConCompFound =$ False AND $pb < pe$ **do**
    for $j \in \{1, 2\}$ **do**
        if $A_{I_j}[pb] = i_\alpha$ **then** $cb_j \leftarrow cb_j + 1$ **else** $cb_j \leftarrow cb_j - 1$
        if $A_{I_j}[pe] = \overline{i}_\alpha$ **then** $ce_j \leftarrow ce_j + 1$ **else** $ce_j \leftarrow ce_j - 1$
    if $cb_1 = 0$ OR $cb_2 = 0$ OR $ce_1 = 0$ OR $ce_2 = 0$ **then**
        $(L, i) \leftarrow$ (list of vertices of the con. comp., index of the zero counter found)
        $ConCompFound \leftarrow$ True
    $pb \leftarrow pb + 1; pe \leftarrow pe - 1$
return (L, i)

Fig. 5. Procedure SCCs where the graphs are given as sets of intervals and represented through A_{I_1} and A_{I_2}.

(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_2}. This can be done in $O(p \log p)$ time.

These observations prove the following proposition 1.

Property 1. Algorithm EXTRACTs(L,i) has a worst case time complexity in $O(p \log p)$, 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, 7, 5]. It is based on the following observation.

Let i be a left (resp. right) bound of an interval i. If i is used in a list L_0 to perform an extraction through EXTRACTs and later re-used in another list L_1 in EXTRACTs, 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 bound 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 $\sum_{i=1}^{\log n} log(\frac{n}{2^i})$ which is $O(\log^2 n)$. As there are $2n$ left and right bounds, the global complexity is $O(n \log^2 n)$ worst case time.

Notice that if the graphs are explicitely given as sets of edges and vertices, the previous $n \log^2 n$ algorithm is of use after a "realisation" phase of $O(kn+m)$ [4]. Thus, the resulting algorithm in this case is $O(m + n \log^2 n)$ time.

4 CCP on tree forests

Gen-CCP on tree forests can be solved in $O(kn \log n)$ time using an easy simplification of the general approach of [6]. However, the remaining $\log n$ factor directly derives from the Hopcroft's partitioning framework and is a bottleneck for all Gen-CCP like algorithms (including our previous one). To achieve a better complexity (if possible), we need to overcome this obstacle by another approach. This is the reason why we aim to explore alternative methods solving Gen-CCP on tree forests, even if the overall complexity is not improved. We expect that one of these new approaches will lead to a better algorithm, or to a lower bound. In the following, we prensent the algorithm for two forets only. However, the results are directly extensible to $k > 2$ forests. An instance of CCP on two tree forests is provided in Figure 6.

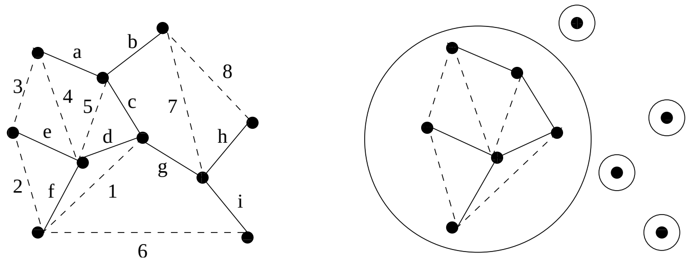

Fig. 6. CCP on tree forests. The left figure represents two tree forests on the same set of vertices and the right figure shows the partition of those vertices in maximal connected sets.

We present a new approach solving CCP on tree forests that is not based on a partitioning framework, but still remains in $O(n \log n)$. This new algorithm is based on decremental integer range searching [12, 15].

4.1 Euler tour and interval representation

Let $T = (V, E)$ be a tree over a set of V vertices. We first define the oriented graph $G_T(V, E_T)$ on the same vertices than T by letting:

$$(x, y) \in E_T \text{ and } (y, x) \in E_T \iff (x, y) \in E.$$

Definition 2. *An Euler Tour of T, named Et_T, is a circuit that traverses every edge of G_T exactly twice. The circuit is described by the sequence Et_T of traversed vertices.*

Notice that an Euler tour can begin at any node of T. We arbitrarily fix one. For a tree forest, we build an Euler tour for each tree. By extension, for a tree forest TF, we denote Et_{TF} the set of euler tour of each tree in TF.

Let FT_1 and FT_2 be two tree forests over a same set of vertices V. Let $Et_1 = Et_{TF_1}$. Each position in Et_1 corresponds to a vertex in V. An *interval representation* of T_2 over Et_1 is a set $I_{Et_1}(T_2)$ of intervals over positions of Et_1 where each interval represents an edge of T_2. Let $e = (v_1, v_2)$ be an edge of T_2, an interval I over positions of Et_1 represents e as soon as the extremities of I correspond respectively to v_1 and v_2. A vertex may appear several times in Et_1, so that the choice of an interval that represent e is not unique. Figure 7 gives an example of an Euler tour on the tree given in plain edges at Figure 6, together with an interval representation of the tree forest given in dashed edges.

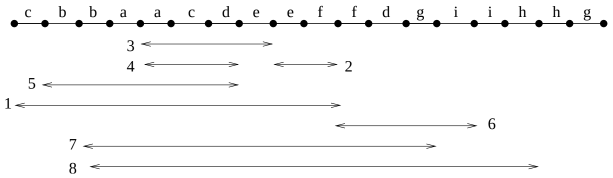

Fig. 7. Euler tour and interval representation.

The Euler tour Et_T of a tree T is easily updated when an edge $e = (v_1, v_2)$ of T is deleted. We define the *cutting interval* I of edge e over positions of Et_T. The left bound of I is the right position of the first appearance of e in Et_T. The right bound of I is the left position of the last appearance of e in Et_T. These are inner bounds. An example is shown at Figure 8.

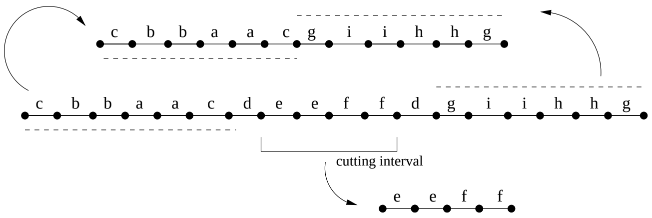

Fig. 8. Deletion of edge d: the Euler tour is cut w.r.t. the cutting interval of d.

Once e is deleted, a forest of two trees is obtained. The Euler tours of the forest are obtained as (1) the sequence of Et_T inside the cutting interval and (2) the sequence outside the cutting interval.

4.2 CCP as cutting edges on interval representations

Let FT_1 and FT_2 be two tree forests over a same set of vertices V. Let F be a file containing all edges of T_2 (resp. T_1) whose representing interval joins two vertices contained in different Euler tours of Et_1 (resp. Et_2). The common connected problem is solved for the couple (FT_1,FT_2) by the following DoubleCut algorithm.

While there is an edge e in F,

(a) Drop e from F (suppose w.l.o.g. that e is an edge of T_1).
Let $T_1(e)$ be the tree in FT_1 that contains e, and $Et_{T1(e)}$ be the Euler tour representation of $T_1(e)$.
Let $[e_a, e_b]$ be the cutting interval of e over $Et_1(e)$.
(b) Append to F all edges of T_2 whose representing interval has one extremity inside $[e_a, e_b]$ and another extremity outside of $[e_a, e_b]$,
(c) Cut $Et_1(e)$ in two Euler representations with respect to the cutting interval $[e_a, e_b]$

Proposition 1. *Algorithm DoubleCut applied on forests FT_1 and FT_2 correctly computes the Euler tours of tree forests whose connected components are the maximal common connected components of FT_1 and FT_2.*

Proof. The algorithm maintains the Euler tours of FT_1 and FT_2 while deleting edges that have been added to F. We prove three fondamentals points.

First, the edge added F coming from FT_1 (resp. FT_2) join to (distinct) connected component of FT_2 (resp. FT_1). Moreover, all such edges are added to F. This is true by definition at the beginning. Then, each step (b) adds all edges in F that joins two vertices that are separated by step (c).

Secondly, we prove that the maximal common connected sets are conserved through points (a,b,c). Step (c) removes edge e that was in F, thus that links two distinct component in the other graph. It can therefore be cut without affecting the maximal common connected sets.

Suppose now that there exists an edge e' at the end of the algorithm that links two connected components of the other graph. This is incompatible with the fact that F is empty (Step (a)). Thus, the remaining subset partition in connected component is the same in the two resulting graphs. The second point we proved insures that this partition is also the partition in maximal connected sets.

4.3 Decremental half infinite range searching over a grid

Two steps are critical in algorithm DoubleCut. Step (b) requires to efficiently detect edges of T_2 whose representing intervals $[i_a, i_b]$ have an extremity in $[e_a, e_b]$ and another outside. Step (c) imposes to be able to remove edges out of an interval representation to add it in F.

These two points may be solve using decremental half infinite range searching over a grid. Given an integer grid $U \times U$ and a set of points P over the grid, the *half infinite range searching problem* is a well known problem of computational geometry that consists in finding all points of P ranging in an area of the form $[A, B] \times [C, D]$ where at least one bound among A, B, C, D is infinite. It is decremental if the data structure can be updated to efficiently remove some points. In Step (b), the condition over $[i_a, i_b]$ is equivalent to:

$$(e_a \leq i_a \leq e_b \ \wedge \ e_b < i_b) \ \vee \ (e_a \leq i_b \leq e_b \ \wedge \ i_a < e_a) \tag{1}$$

Indeed, finding all couples (i_a, i_b) that satisfy equation 1 consists in two half infinite range searching queries. In practical terms, consider a grid $2n \times 2n$ and let P be the set of points (i_a, i_b) in the grid where $[i_a, i_b]$ is a representing interval of an edge in T_2. The set of intervals $[i_a, i_b]$ that satisfy 1 is the union of answers of the two half range searching queries over area $[e_a, e_b] \times [e_b + 1, +\infty]$ and over area $[-\infty, e_1 - 1] \times [e_a, e_b]$. Step (c) requires to solve Step (b) using a dynamical data structure.

A data structure for managing a set of intervals over a grid of size U is presented in [11]. It has the following properties: (a) construction in $O(U \log U)$; (b) update (deletion of an interval) in $O(\log n)$; (c) half range queries in $O(n \log n + u)$, where u is the number of resulting intervals satisfying the query.

Proposition 2. *Algorithm DoubleCut is $O(n \log n)$ worst case time.*

Proof. In our case, $U = O(n)$. In consequence, the preprocessing phase is in $O(n \log n)$ time. A specific edge can be removed from an interval representation only once. Thus, the overall cost of all update operations is bound by $O(n \log n)$. Each time an interval is found through a half range query, the corresponding edge is removed from the interval representation and added to F. Thus, the total number of intervals found by all the queries is bound by $O(n)$ and the complexity of all the half range queries by $O(n \log n)$.

References

1. M.-P. Béal, A. Bergeron, S. Corteel, and M. Raffinot. An algorithmic view of gene teams. *Theoretical Computer Science*, 320(2-3):395–418, 2004.
2. A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19(1):85–98, 1982.
3. A.-T. Gai, M. Habib, C. Paul, and M. Raffinot. Identifying Common Connected Components of Graphs. *Technical report LIRMM-03016*, 2003. Submitted.
4. M. C. Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, 1980.
5. 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.
6. M. Habib, C. Paul, and M. Raffinot. Maximal common connected sets of interval graphs. In *Combinatorial Pattern Matching (CPM)*, number 3109 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2004.

7. M. Habib, C. Paul, and L. Viennot. A synthesis on partition refinement: a useful routine for strings, graphs, boolean matrices and automata. In *15th Symposium on Theoretical Aspect of Computer Science (STACS)*, number 1373 in Lecture Notes in Computer Science, pages 25–38. Springer-Verlag, Berlin, 1998.
8. S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Combinatorial Pattern Matching (CPM)*, number 2089 in Lecture Notes in Computer Science, pages 207–218. Springer-Verlag, Berlin, 2001.
9. 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.
10. 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.
11. E. M. McCreight. Priority search trees. *Siam J. on Computing*, 14:257–276, 1985.
12. M. H. Overmars. Efficient data structures for range searching on a grid. *J. Algorithms*, 9(2):254–275, 1988.
13. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
14. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.
15. D.E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394, 1984.