Competitive Graph Searches

Binh-Minh Bui-Xuan, Michel Habib, Christophe Paul

Competitive Graph Searches

Binh-Minh Bui Xuan [a,*] Michel Habib [b] Christophe Paul [a]

[a] *CNRS - LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5, France.*
{buixuan,paul}@lirmm.fr

[b] *CNRS - LIAFA, 2 place Jussieu, Case 7014, 75251 Paris Cedex 05, France.*
habib@liafa.jussieu.fr

Abstract

We exemplify an optimisation criteria for divide-and-conquer algorithms with a technique called generic competitive graph search. The technique is then applied to solve two problems arising from biocomputing, so-called *Common Connected Components* and *Cograph Sandwich*. The first problem can be defined as follows: given two graphs on the same set of n vertices, find the coarsest partition of the vertex set into subsets which induce connected subgraphs in both input graphs. The second problem is an instance of sandwich problems: given a partial subgraph G_1 of G_2, find a partial subgraph G of G_2 that is partial supergraph of G_1 (sandwich), and that is a cograph. For the former problem our generic algorithm not only achieves the current best known performance on arbitrary graphs and forests, but also improves by a $\log n$ factor when the input is made of planar graphs. However, our complexity for intervals graphs is slightly lower than a recent result. For the latter problem, we first study the relationship between the common connected components problem and the cograph sandwich problem, then, using our competitive graph search paradigm, we improve the calculation of cograph sandwiches from $O(n(n+m))$ downto $O(n + m\log^2 n)$, where n is the number of vertices and m of total edges.

Key words: graph search, divide-and-conquer algorithm, common connected graph component, sandwich graph problem

1 Introduction

The classical divide-and-conquer algorithmic framework (see e.g. [10,21]) can be summarised as *dividing* the input problem into some sub-problems; then conquering the sub-problems by *making recursive calls*; and *combining* the

[*] Corresponding author. Tel: +33 4 67 41 86 05. Fax: +33 4 67 41 85 00.

sub-solutions into a global solution. The best known examples probably are standard sorting algorithms and dynamic programming algorithms. Without specific assumptions, the method helps with designing algorithms running in quadratic worst case time. Classical optimisation techniques to improve this bound mostly consist of holding some condition on the *recursive computation*, e.g. with merge-sort, median computation [4], and algorithms derived from the planar separator theorem [20].

Actually, even when no condition is placed on the recursive computation, it is acquired that cutting down the *divide* and *combine* part also improves the global computing time [1,19]. However, applied examples of this paradigm are scant up to our knowledge. This paper gives a series of such examples. To this aim we consider the problem of, given a graph and a list of one representative vertex per connected component, visiting all connected components but the largest. We depict how a so-called *competitive graph search* can solve the problem in linear time on the size of the visited vertices and edges. Notice that the size of the largest component might be very close to that of the initial graph. In this case the competitive graph search records a small time complexity.

Using the competitive graph search, we first give a solution to a problem arisen from computational biology: given two graphs G_1 and G_2 on the same vertex set V, find the coarsest partition of V into $V_1, \ldots, V_k$ such that, for all $1 \leq i \leq k$, both induced subgraphs $G_1[V_i]$ and $G_2[V_i]$ are connected [2,16]. Depending on the data structure, our solution can be used for different graph classes. Its performance equals the best known so far for arbitrary graphs [14] and forests [11]. For planar graphs, we improve the performance by a $\log n$ factor, namely with an $O(n \log n)$ computing time. Our complexity for interval graphs is in $O(n + m \log n)$, while a recent result improved this to $O(m + n \log n)$ [12]. Finally, we study the relationship between the common connected components problem and another class of problems issued from biocomputing, namely sandwich graph problems [15], and improve the computation of cograph sandwiches from $O(n(n+m))$ [15] to $O(n + m \log^2 n)$ as a corollary of competitive graph searching.

2 Algorithmic Aspects

2.1 Divide and Conquer Paradigm

This paper addresses the following formalism. Let $\mathcal{P}$ be a problem on a set $\mathcal{S}$ of data structures, and *Size* a function from $\mathcal{S}$ to $\mathbb{R}^+$. $\mathcal{H}$ is a *divide-and-conquer algorithm with respect to Size solving* $\mathcal{P}$ if:

- there exists a set $\mathcal{T} \subseteq \mathcal{S}$ of trivial inputs on which $\mathcal{H}$ solves $\mathcal{P}$ in $O(1)$ time;
- any $S \in \mathcal{S}$ with $Size(S) \leq 1$ is a trivial input, namely $S \in \mathcal{T}$;
- for all $S \notin \mathcal{T}$, $\mathcal{H}(S)$
  - first divides S into some sub-instances $S_1, \ldots, S_k$ holding $Size(S_i) > 0$ for all i and holding $Size(S_1) + \ldots + Size(S_k) \leq Size(S)$,
  - then recurses with $\mathcal{H}(S_1), \ldots, \mathcal{H}(S_k)$,
  - and finally combines the results in order to provide the output of $\mathcal{H}(S)$.

Let $C(S)$ be the total computing time of $\mathcal{H}(S)$, $Div(S)$ be the time for finding $S_1, \ldots, S_k$, and $Com(S)$ for combining the sub-solutions into the output of $\mathcal{H}(S)$. Then, for all $S \notin \mathcal{T}$, $C(S) = Div(S) + \sum_{i=1}^{k} C(S_i) + Com(S)$ straight from definition. Let $n = Size(S)$. If $Div(S) + Com(S) = O(n)$, then there is a naive bound $C(S) = O(n^2)$ (see e.g. [10,21]). Well-known optimisation techniques divide S into two subproblems S_1 and S_2 of equal size. This yields $O(n \log n)$ time algorithms such as Merge sort (see e.g. [10,21]).

Besides, the naive quadratic bound is known to improve as recursive calls decrease. For instance, most famous algorithms such as the median computation [4] or algorithms deriving from the planar separator theorem [20] reach linear worst case time bound by avoiding a fraction of S on recursive calls, namely by granting $\frac{Size(S_1)+\ldots+Size(S_k)}{Size(S)} < 1$. The success of such examples might explain why minimising the divide and combine time $Div(S)+Com(S)$ usually is disregarded in standard optimisation approaches. In this paper, we address the case when recursive calls have to be applied on all parts, namely when $\frac{Size(S_1)+\ldots+Size(S_k)}{Size(S)} \leq 1$ with the bound reached. As a result of a larger theorem in [1], minimising $Div(S) + Com(S)$ here becomes fruitful according to an "*avoid the largest*" idea. Within our terminology, it could be stated as follows.

Proposition 1 *[1] Let $\mathcal{H}$ be a divide-and-conquer algorithm, and α be such that, for all $S \in \mathcal{S} \setminus \mathcal{T}$, $Div(S)+Com(S) \leq \alpha \times (Size(S) - \max_{i=1}^{k} Size(S_i))$, where $S_1, \ldots, S_k$ is the partition of S given by $\mathcal{H}(S)$. Then, for all input $S \in \mathcal{S}$, $\mathcal{H}(S)$ runs at most in $\alpha \times Size(S) \log Size(S)$ time. This bound is best possible.*

Proof: by induction on $s = Size(S)$. If S is not trivial and $S_1, \ldots, S_k$ are such that $s_k = Size(S_k)$ is greater than any $s_i = Size(S_i)$, then

$$\begin{aligned}
Div(S) + Com(S) + \sum_{i=1}^{k} C(S_i) &\leq \alpha \times \left(\sum_{i=1}^{k-1} s_i + \sum_{i=1}^{k} s_i \log s_i \right) \\
&\leq \alpha \times \left(\sum_{i=1}^{k-1} s_i + \sum_{i=1}^{k-1} s_i \log \frac{s}{2} + s_k \log s \right) \\
&\leq \alpha \times s \log s.
\end{aligned}$$

Now, let $\mathcal{P}$ and $\mathcal{H}$ be such that there exist $S_0 \in \mathcal{T}$ and S_q $(q \geq 1)$ where

$\mathcal{H}$ divides S_q into two sub-instances that are both identical to S_{q-1}. Then, $\mathcal{H}$ computes at least in $\alpha \times Size(S_q) \log Size(S_q)$ time on S_q. □

Remark: The standard optimisation technique used in Merge sort results in the same bound. However, the size of the input given to Merge sort is granted to geometrically decrease (by half) as inductive levels grow, implying that the induction depth is lesser than $\log Size(S)$. On the other hand, our result still holds even when the induction depth is linear on $Size(S)$.

Though it may be straightforward to avoid the largest part for linear data structures such as ordered arrays, it is less easy in other cases, for instance when dealing with graphs. Indeed, the challenge is to avoid some "largest" graph component without exploring the whole graph. We exemplify the practical potential of Proposition 1 on graphs with a so-called *competitive graph search* technique.

2.2 Competitive Graph Search

Let $G = (V, E)$ be a graph. We define the size of G as its number of vertices and edges: $Size(G) = |V| + |E|$. An edge is within a vertex subset if both extremities of the edge belong to the subset; and a vertex subset has the size of the subgraph it induces. This section addresses two problems.

Exploring Connected Components: Let *Rep* be a list of pointers to one representative vertex per connected component of G. The first problem consists of, given G and *Rep*, visiting all connected components of G but the largest. To this aim, a *competitive graph search* proceeds as follows. At the beginning, all components are competitors via their corresponding representative vertex in *Rep*. Then, each step of the search visits one new element – vertex or edge (the "or" is exclusive) – of each competitor. If no new element is found for some, these components no more compete. This process continues as long as there are at least two remaining competitors. Obviously the last competitor C is the largest and has not been entirely visited. Indeed, if s' is the size of the second largest competitor C', then only s' elements of C have been visited, which leads to the following result.

Proposition 2 *Given a graph G and a list of pointers to one representative vertex per connected component of G, a* competitive graph search *visits all connected components of G but the largest component C in time bounded by $2 \times (s_G - s_C)$ with s_G the number of vertices and edges of G, and s_C the number of vertices and edges of $G[C]$.*

Proof: the exact visiting time is $(s_G - s_C) + s'$ with s' the size of the second largest component. □

Exploring Induced Subgraphs: Let $\{V_1, \ldots, V_k\}$ be a vertex partition of G, *Rep* be a list of one pointer to each V_i, and $\texttt{oracle}(v, w)$ be true if and only if the vertices v and w belong to the same V_i. The second problem consists of, given G, *Rep* and `oracle`, visiting all subgraphs of G induced by $V_1, \ldots, V_k$ but the largest subgraph. Here, we still start with the members of *Rep* representing the competitors $V_1, \ldots, V_k$. Then, each step still tries to visit one new element (vertex or edge) of each competitor. The hitch is that inter-edges, namely those in $IE = \{vw \in E \mid \exists i \neq j \text{ s.t. } v \in V_i \text{ and } w \in V_j\}$, belong to none of the competitors. However, thanks to `oracle`, the search can check at any moment whether an edge is inter-edge. Thus, for each competitor, each step of the graph search either discovers a new vertex, or checks the outgoing edges until one edge belonging to that competitor is found. The remaining of the search behaves like before.

Proposition 3 *Given a graph $G = (V, E)$, a partition $\{V_1, \ldots, V_k\}$ of V, a list of pointers to one representative vertex per V_i $(1 \leq i \leq k)$, and a function* `oracle` *testing whether two vertices belong to the same V_i, a* competitive graph search *visits the subgraphs $G[V_1], \ldots, G[V_k]$ but the largest in time bounded by $2 \times (s_G - s_C) + M$ with M the number of inter-edges between the subgraphs, s_G the number of vertices and edges of G, and s_C the number of vertices and edges of the largest subgraph.*

Proof: the exact visiting time is $(s_G - s_C) + s' + M'$ with s' the size of the second largest subgraph, and M' the number of visited inter-edges. $\square$

To conclude, the main technical difficulty of a competitive graph search is to manage an entry to each competitor before starting and to maintain this as an invariant during the recursive process. Notice that this generic competitive search can be applied to other discrete structures such as directed graphs, hypergraphs or matroids. Let us examine the paradigm on two graph problems.

3 Common Connected Component Computation

Given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, a common connected component C of (G_1, G_2) is a maximal vertex subset such that both $G_1[C]$ and $G_2[C]$ are connected. For example, the only such components presented in Fig. 2 are singletons. This problem was introduced in [6] for the study of the genes structure. One graph is obtained by the distance between genes in the sequence with respect to a given threshold, the other graph can be any graph on the same set of genes generated by some chemical reaction. The problem also arises from comparative genomics, e.g. in the search of *gene-teams* where G_1 and G_2 are two graphs defined by two genomic sequences on the same set of genes [2]. Adjacency between genes is given by their distance in the

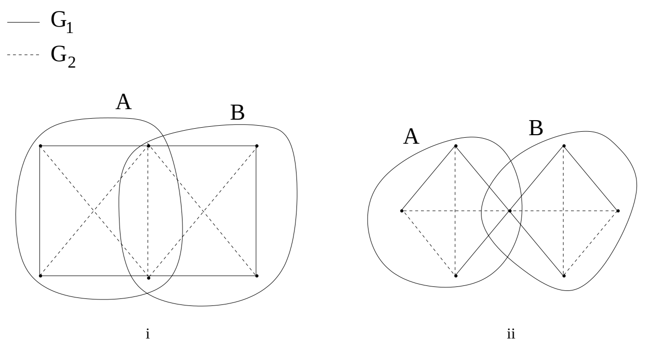


Fig. 1. i. G_1 not cycle free and (b) violated. ii. G_1 not a path and (c) violated.

sequence with respect to a given threshold. Let F denotes the family of common connected components. As usual when dealing with families of subsets, it is interesting to check under which conditions the family is equipped with a lattice structure. Let us first consider the basic properties:

(a) Let $A, B \in F$ and $A \cap B \neq \emptyset$ then $A \cup B \in F$.

(b) Let $A, B \in F$ and $A \cap B \neq \emptyset$ then $A \cap B \in F$.

In fact (a) is obviously true, but (b) does not necessarily hold (see Figure 1).

Lemma 1 *If G_1 and G_2 are forests then F satisfies (b).*

Proof: Let us consider two vertices $a, b \in A \cap B$. By definition there exist a chain from a to b in $G_1[A]$, and a chain from a to b in $G_1[B]$. Since G_1 is a forest this chain is necessarily unique and therefore included in $A \cap B$. $\square$

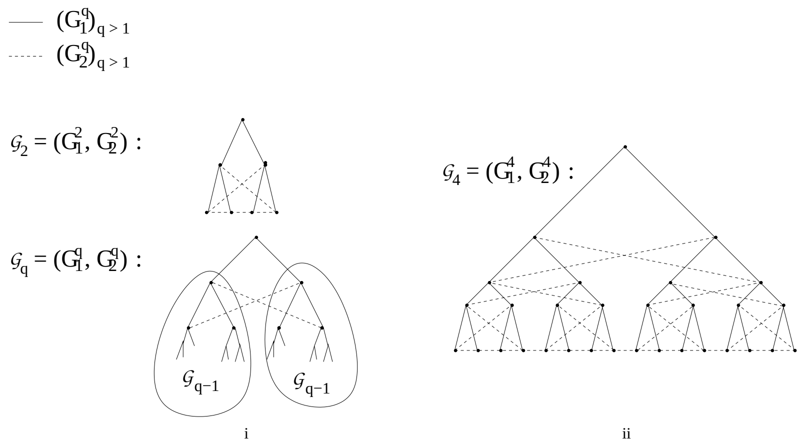
As defined in [9], weak partitive families satisfy (a), (b) and the following:

(c) Let $A, B \in F$ that overlap, then $A \setminus B, B \setminus A \in F$.

Lemma 2 *If G_1, G_2 are forests of chains, then F satisfies (c).*

Proof: Let us consider $x, y \in A \setminus B$. If x, y are not connected in $G_1[A \setminus B]$, then the unique chain joining x and y goes trough a vertex z in $G_1[A \cap B]$. Since it exists at least a vertex $t \in B \setminus A$, then the connected component of G_1 containing x, y, z, t is not chain, a contradiction. $\square$

Noticed that if G_1 and G_2 only are supposed to be forests, then (c) might be violated (e.g. in Figure 1). However, in case of forests of chains, F is a weak partitive family. Then, a theorem in [9] implies the existence of a unique decomposition tree for the family F. Using this tree, recursiveness can easily be conducted on each common connected set. This corresponds to a well-studied case and the computation of the decomposition tree can be done in $O(|V|)$ by adding some slight modifications to the computation of common intervals of



$(G_1^q)_{q>1}$
$(G_2^q)_{q>1}$
$\mathcal{G}_2 = (G_1^2, G_2^2)$:
$\mathcal{G}_q = (G_1^q, G_2^q)$:
$\mathcal{G}_{q-1}$
$\mathcal{G}_{q-1}$
$\mathcal{G}_4 = (G_1^4, G_2^4)$:
i
ii

Fig. 2. i. A sequence $(\mathcal{G}_q)_{q>1} = ((G_1^q, G_2^q))_{q>1}$ of instances for which our common connected component computation runs in $\Theta(n \log n)$. ii. Details of $\mathcal{G}_4$.

two permutations [23] (here the two chains can be seen as two permutations). Another equivalent version of this problem is the computation of the modular decomposition tree of a permutation graph and we can use algorithms from [3,7] which also run in $O(|V|)$. Let us now address the general case and consider an algorithm scheme based on the following simple partitioning lemma.

Lemma 3 *[14] Let us suppose that there is no edge between X and $V \setminus X$ in one graph among $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$. Then, the common connected components of (G_1, G_2) are those of $(G_1[X], G_2[X])$ plus those of $(G_1[V \setminus X], G_2[V \setminus X])$.*

Firstly, one can suppose w.l.o.g. that $E_1 \cap E_2 = \emptyset$ by recursively merging together vertices x and y if $(x, y) \in E_1 \cap E_2$ [14]. Besides, if none of G_1 and G_2 is connected, a preliminary standard graph search can build the sub-instances $(G_1[X], G_2[X])$ for all connected components X of G_1. Lemma 3 states that computing the common connected components of (G_1, G_2) results in those of the latter sub-instances. Hence, we suppose w.l.o.g. G_1 connected. Finally, another preliminary graph search can compute a list of one representative vertex per connected component of G_2 before launching our main recursive algorithm.

Concisely, the main algorithm addresses the problem of finding the common connected components of two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, given along with a list *Rep* such that $E_1 \cap E_2 = \emptyset$, G_1 is connected, and *Rep* has exactly one representative vertex per connected component of G_2. It proceeds as follows.

- If $k = |Rep| = 1$ then return V.
- Otherwise, let $V_1, \ldots, V_k$ be the connected components of G_2. Then, for all $1 \leq i \leq k$, we compute $G_1[V_i]$, $G_2[V_i]$, and a list Rep_i containing one representative vertex per connected component of $G_1[V_i]$.

- By inverting G_1 and G_2, we make recursive calls on $(G_2[V_i], G_1[V_i], Rep_i)$ and return all results.

The correctness follows from Lemma 3. Obviously, the above operations can be done using standard graph searches, which would yield a naive $O(n(n+m))$ solution. However, we can benefit from competitive graph searches to improve the bound. Let $s(G) = Size(G) = |V| + |E|$ for any graph $G = (V, E)$, $s_i^1 = s(G_1[V_i])$, $s_i^2 = s(G_2[V_i])$, and $s_i = s_i^1 + s_i^2$. Let the "sum of all but the max" $\text{sam}_{i\in I} s_i$ be a shortcut for $\sum_{i\in I} s_i - \max_{i\in I} s_i$.

Lemma 4 *If s_i^1, s_i^2 are positive and $s_i = s_i^1 + s_i^2$ for all $i \in I$, then:*

$$\text{sam}_{i\in I}\ s_i^p \quad \leq \quad \text{sam}_{i\in I}\ s_i, \qquad\qquad \textit{with } p \in \{1, 2\}.$$

Proof: Let i_0 and i_1 be such that $s_{i_0} = \max_{i\in I} s_i$ and $s_{i_1} = \max_{i\in I} s_i^1$. Obviously, $s_{i_0}^1 \leq s_{i_1}^1 \leq s_{i_1}$. Besides, $\sum_{i\in I\setminus\{i_0,i_1\}} s_i^1 \leq \sum_{i\in I\setminus\{i_0,i_1\}} s_i$. Adding the two inequalities allows to conclude. □

As already mentioned a competitive graph search *on the connected components* of G_2 computes all $G_2[V_i]$ except for $G_2[V_{i_2}]$ with $s_{i_2}^2 = \max_{1\leq i\leq k} s_i^2$. During the search, we label the vertices in V_i $(i \neq i_2)$ so that they can be distinguished afterwards. Those in V_{i_2} keep their old label so that they also come as a distinct k^{th} class. We define `oracle` which tests whether two vertices have same labels. By removing from G_2 vertices and edges of the $k-1$ computed graphs, we compute $G_2[V_{i_2}]$. The operations so far run in $O(\text{sam}_{1\leq i\leq k} s_i^2)$ time. Using the function `oracle`, a competitive graph search *on the induced subgraphs* of G_1 computes all $G_1[V_i]$ except for $G_1[V_{i_1}]$ with $s_{i_1}^1 = \max_{1\leq i\leq k} s_i^1$. Let IE contain all inter-edges in G_1 between $G_1[V_1], \ldots, G_1[V_k]$. By removing from G_1 vertices and edges of $G_1[V_i]$ $(i \neq i_1)$, plus the inter-edges in IE, we compute $G_1[V_{i_1}]$. This step takes $O(|IE| + \text{sam}_{1\leq i\leq k} s_i^1)$ time. As $G_1[V_i]$ $(i \neq i_1)$ are of size small enough, we simply compute Rep_i $(i \neq i_1)$ thanks to standard searches (such as the breath-first graph search) on those graphs. This latter step takes $O(\text{sam}_{1\leq i\leq k} s_i^1)$ time. From Lemma 4 all operations so far run in $O(|IE| + \text{sam}_{1\leq i\leq k} s_i)$. Finally, we assume that Rep_{i_1} is computed by some routine $\mathcal{R}$, and result in the following main theorem.

Routine $\mathcal{R}$: *Given a connected graph $G_1 = (V, E_1)$ and a vertex partition $V_1, \ldots, V_k$, the routine $\mathcal{R}$ computes a list Rep_{i_1} containing one representative vertex per connected component of $G_1[V_{i_1}]$, where V_{i_1} is the largest among $V_1, \ldots, V_k$.*

Main theorem *Given a routine $\mathcal{R}$ as defined above, the common connected components of two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ can be computed in $O(n + m \log n + t_{\mathcal{R}})$ time, where $n = |V|$, $m = |E_1| + |E_2|$, and $t_{\mathcal{R}}$ stands for the global computing time (through recursions) of the routine $\mathcal{R}$.*

Proof: The preliminary operations for computing *Rep* and for rendering $E_1 \cap E_2 \neq \emptyset$ and G_1 connected run in $O(n+m)$. Now, our main algorithm follows the divide-and-conquer paradigm. Therein, the combine time of each step is $O(1)$. Moreover, except for the "$|IE|$" terms due to inter-edges and the cost of calls to the routine $\mathcal{R}$, the divide time fulfills requirements of Proposition 1. According to this, we split the global complexity analysis of the main algorithm into three parts. The first counts the "$|IE|$" terms, the second the total cost of $\mathcal{R}$, and the third the remaining. Let $G_1' = (V', E_1')$ and $G_2' = (V', E_2')$ be the input graphs given to the main algorithm. Let $n' = |V'|$ and $m' = |E_1'| + |E_2'|$. Then, the "$|IE|$" part is in $O(m') = O(m)$ since an edge can be "inter-edge" only once throughout the running of the main algorithm. The second part was denoted by $t_{\mathcal{R}}$. From Proposition 1, the third complexity part is in $O((n'+m')\log(n'+m')) = O(m \log n)$ because $m' \leq m \leq n^2$, and G_1' connected implies $n' = O(m')$. Whence, the whole running is in $O(n + m\log n + t_{\mathcal{R}})$. $\Box$

Implementation of routine $\mathcal{R}$: The idea of computing Rep_{i_1} is the following. Let OG be the outgoing vertices in G_1 from $G_1[\cup_{i \neq i_1} V_i]$ to $G_1[V_{i_1}]$, namely $OG = \{y \in V_{i_1} \mid \exists x \notin V_{i_1} \text{ s.t. } (x,y) \in E_1\}$. Since G_1 is connected, $Rep_{i_1} \subseteq OG$. Computing OG only takes $O(|IE| + \text{sam}_{1 \leq i \leq k} s_i^1)$ time. Our idea is to filter OG efficiently until we obtain Rep_{i_1}.

Tool boxes B_1 and B_2: To this aim, we will use two tool boxes. The first tool box B_1 computes a spanning-forest of a given graph G, and for each vertex in G a pointer to the identifier of the spanning tree it belongs to. Given a graph G and one such spanning-forest representation of G, plus an edge e in G, the second tool box B_2 computes the spanning-forest representation of $G \setminus \{e\}$, and update the pointers to spanning tree identifiers.

Thanks to B_1, right before launching the main recursive algorithm of the common connected component problem, we compute the spanning-forest representations of the two input graphs. At each recursive step we use the `oracle` function to compute the inter-edge set IE in $O(|IE| + \text{sam}_{1 \leq i \leq k} s_i^1)$ time. Using B_2 we delete all edges of IE of the corresponding spanning-forest representation. Then, each vertex in OG has a pointer to the identifier of its spanning tree in the current G_1. We then sort those identifiers using standard sorting. Finally we scan the sorted identifiers and only keep one vertex of OG per identifier, which will form the list Rep_{i_1}.

Sorting the identifiers would take $O(|OG| \log |OG|) = O(|IE| \log |IE|) = O(|IE| \log m)$ time. The sum of all the $|IE|$ terms throughout the computation is bounded by m. Hence, except for the cost of calls to B_1 and B_2, the complexity still is in $O(n + m \log n)$.

Forests: If the input graphs given to the main algorithm are forests, they form their own spanning forests. The only thing to be cared off is keeping a pointer

	best so far	this paper	conjecture
forests of trees	$O(n \log n)$ [11]	$O(n \log n)$	$O(n)$
interval graphs	$O(m + n \log n)$ [12]	$O(n + m \log n)$	$O(n + m)$
unit interval graphs	$O(n \log \Delta \log n)$ [2]	$O(n\Delta \log n)$	$O(n + m) = O(n\Delta)$
planar graphs	$O(n \log^2 n)$ [14]	$O(n \log n)$	$O(n)$
permutation graphs	$O(n \log n + m \log^2 n)$ [14]	$O(n + m \log^2 n)$	$O(n + m)$
arbitrary graphs	$O(n \log n + m \log^2 n)$ [14]	$O(n + m \log^2 n)$	$O(n + m \log n)$

Fig. 3. Common connected component computation time, with n the number of vertices, m the total number of edges, and Δ the maximum vertex degree.

for each vertex to the identifier of the spanning tree it belongs to. This, for B_1 can be done easily in $O(n+m)$ time. For B_2, let the edge to be deleted be $e = (x, y)$. We only need to update the identifiers of the spanning tree that has contained e before the deletion. The deletion of the edge e will split the old spanning tree into two parts, x and y could be seen as representatives for each of both part. Then, a competitive graph search will update the identifier of the smaller part in time proportional to the size of the smaller one, and the task of B_2 is complete. The complexity of B_2 thus is $O(m \log m)$ from Proposition 1. Finally, $m = O(n)$ in case of forests. We conclude that $t_{\mathcal{R}} = O(n \log n)$.

Corollary 1 *The common connected components of forests can be computed in $O(n \log n)$ time.*

Non-forest cases: For arbitrary graphs, we benefit from results of [18] on the so-called *ET-tree* data structure [17]. Let m' be the number of edges in the two graphs given as input to the main algorithm, and n' be the number of their vertices. Then, the cost for B_1 in this case is in $O((n' + m') \log^2 n')$ [18], or $O(m' \log^2 n')$ as m' is higher than n' (one of the two graphs is connected). Finally, the cost for B_2 in this case is in $O(\log^2 n')$ per operation [18]. As before, we note that an edge can be "inter-edge" only once during the whole computation, thus the total cost for calls to B_2 is in $O(m' \log^2 n')$. Hence, $t_{\mathcal{R}} = O(n + m \log^2 n)$. Likewise, we use results on *edge-ordered dynamic tree* [13] for planar graphs. The corresponding B_1 and B_2 respectively run in $O(m' \log n')$ and $O(\log n')$, and the total running time of both tool boxes is $O(m' \log n')$. Notice that the number of edges in a planar graph is bounded by three times the number of vertices, and $t_{\mathcal{R}} = O(n \log n)$. For interval graphs, the same idea can be done using *clique-path representation* [16] for an $O(m')$ B_1, an $O(\log n')$ B_2, and a total $t_{\mathcal{R}} = O(n + m \log n)$ running time.

Corollary 2 *One can compute the common connected components of arbitrary graphs in $O(n + m \log^2 n)$ time; of planar graphs in $O(n \log n)$ time; and of interval graphs in $O(n + m \log n)$ time.*

Our algorithm turns out to be a generic algorithm for all the related graph classes. As a consequence, mixing different classes is allowed, and yields the computing time equals to the upper one. For instance, the common connected computing time for a planar graph G_1 and an interval graph G_2 is in $O(n + m \log n)$.

4 Application to sandwich cographs

We now address the graph sandwich problems defined by Golumbic, Kaplan, and Shamir (1995) [15]:

Input: $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ two undirected graphs such that $E_1 \subseteq E_2$ and Π be a graph property.

Results: a sandwich graph $G = (V, E)$ satisfying property Π and such that $E_1 \subseteq E \subseteq E_2$.

Edges of E_1 are forced, those of E_2 optional, and those of $E_3 = \overline{E_2}$ forbidden. Unfortunately most cases are NP-complete, e.g. with Π: G being comparability, chordal, strongly chordal, etc. Only few polynomial cases are known, among which cographs [15], and sandwich homogeneous set (i.e. module) [5,8]. Therefore it is a natural question to ask for efficient algorithms for these polynomial cases.

The following result exhibits a strong relationship between the cograph sandwich problem and common connected components. Let us recall that the class of cographs is the smallest class of graphs containing the one vertex graph and closed under series and parallel composition. Therefore any cograph can be seen as a modular decomposition tree without prime nodes. Equivalently cographs can be defined as the class of graphs excluding P_4 as induced subgraph [22], where P_4 denotes the path of length 4.

Theorem 1 *Let $G_1 = (V, E_1)$ be a graph of required edges, and $G_2 = (V, E_2)$ with $E_1 \subseteq E_2$ be a graph of possible edges. The dual graph $G_3 = \overline{G_2}$ of G_2 is defined as the graph of forbidden edges. Then, there exists a sandwich $G = (V, E)$ between G_1 and G_2 (meaning $E_1 \subseteq E \subseteq E_2$) which is a cograph if and only if the common connected components of G_1 and G_3 are singletons.*

Proof: Suppose there is a cograph $G = (V, E)$ with $E_1 \subseteq E \subseteq E_2$. G is not co-connected or the root of the modular decomposition tree of G would be a prime node. Let $V_1, \ldots, V_k$ be the partition of V into connected components of: G if it is not connected, $\overline{G}$ otherwise. That $E_1 \subseteq E \subseteq E_2$ implies there is no inter-edges between the vertex subsets V_i in one graph among G_1 and G_3. Then, using the partitioning Lemma 3, the common connected components of G_1 and G_3 exactly are the union of those of $G_1[V_i]$ and $G_3[V_i]$ for all i.

Obviously, $G[V_i]$ is a sandwich of $G_1[V_i]$ and $G_2[V_i]$. Furthermore, $G[V_i]$ is a cograph, otherwise it would contain an induced P_4, and so would G. Hence, an inductive argument on the vertex subsets V_i will allow to conclude that all common connected components of G_1 and G_3 are singletons.
Conversely, suppose that all common connected components of G_1 and G_3 are singletons. We build a graph $G = (V, E)$ as follows. If $|V| = 1$, $E = \emptyset$. Otherwise, the instance can be divided into two cases. If G_1 is not connected, let $V_1, \ldots, V_k$ be its connected components. We define E such that any pair (x, y) satisfying $x \in V_i$, $y \in V_j$, and $i \neq j$ implies $(x, y) \notin E$. If G_1 is connected, then necessarily G_3 is not connected (otherwise V is a common connected component). Let $V_1, \ldots, V_k$ be the connected components of G_3. We define E such that any pair (x, y) satisfying $x \in V_i$, $y \in V_j$, and $i \neq j$ implies $(x, y) \in E$. In both cases (G_1 not connected or G_3 not connected), the definition of E within each V_i follows inductively on $V_1, \ldots, V_k$. The fact that all common connected components of G_1 and G_3 are singletons guarantees that, for all pairs $(x, y) \in V^2$ with $x \neq y$, we have chosen whether (x, y) belongs to E without contradictory definitions. Hence, G is well-defined. Then, using standard cograph characterisations, G can be proved to be a cograph. (We actually have built the decomposition tree of the cograph.) One can also verify that G is a sandwich between G_1 and G_3 by its construction. □

The above proof is constructive: if all common connected components of G_1 and G_3 are singletons, an algorithm is depicted to compute a cograph that is sandwich of G_1 and G_2. Therein, each step divides the graph into subgraphs induced by some $V_1, \ldots, V_k$, then decides whether edges between the V_i exist, and finally recurses in the subgraphs. This actually follows a divide-and-conquer scheme, with a $O(1)$ combining time. Moreover, deciding the adjacency between the V_i results in labelling the corresponding node in the modular decomposition tree with series or parallel, which can be done in $O(1)$ time. Finally, identifying the subgraphs induced by the V_i can be cared off by a competitive graph searching. Hence, when a sandwich cograph exists, we can build one such in $O(n + m \log n)$ time, where n denotes the number of vertices, and m the number of edges of G_1 and G_3.

Corollary 3 *The sandwich cograph problem can be solved by a robust – in the sense of certifying – algorithm in $O(n+m\log^2 n)$ time, where n is the number of involved vertices, and m the number of forced edges and forbidden edges.*

Proof: We first compute in $O(n+m\log^2 n)$ time the common connected components of the graph G_1 of forced edges and the graph G_3 of forbidden edges. Suppose that all common connected components of G_1 and G_3 are singletons. Then, a sandwich cograph can be build in $O(n + m \log n)$ as depicted in the proof of Theorem 1. We now suppose that there is some common connected component C that is not a singleton. Then, any sandwich G of G_1 and G_2 verify that $G[C]$ is co-connected ($G[C]$ is partial supergraph of $G_1[C]$ and $\overline{G}[C]$

is partial supergraph of G_3). We deduce $G[C]$ not a cograph and must contain and P_4, and so must G. Thus, C is our certificate to state that no sandwich of G_1 and G_2 can be a cograph. In this case, one can verify in linear time that both $G_1[C]$ and $G_3[C]$ are connected and deduce that every sandwich of G_1 and G_2 must contain a P_4. $\square$

The above result improves the $O(n(n+m))$ complexity of the algorithm proposed in [15]. However, we think that:

Conjecture: There exists a linear time algorithm to solve the sandwich cograph problem.

Such an algorithm would imply a linear characterisation of the totally degenerate case of the common connected component problem, when all components are singletons. Similarly the P_4-structure of common connected components is worth being further studied as shows the following proposition, which is highly related to Theorem 1, and states that common connected components must contain many P_4's.

Proposition 4 *Let C be a common connected component of $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ with $E_1 \cap E_2 = \emptyset$, then both $G_1[C]$ and $G_2[C]$ contain a P_4.*

Proof: Consider the root of the modular decomposition tree of $G_1[C]$. It cannot be a parallel node since $G_1[C]$ is connected, nor it can be a series node since $E_1 \cap E_2 = \emptyset$ and $G_2[C]$ is connected. Therefore it is a prime node. Hence $G_1[C]$ is not a cograph and must contain a P_4. Similar argument holds for $G_2[C]$. $\square$

5 Conclusion and Perspectives

This paper gives a generic common connected components computation, which also exemplifies an infrequent divide-and-conquer optimisation scheme. Since divide-and-conquer is a very basic method, our algorithm is simply structured while holding some efficient performances (Fig. 3). We also improve the computation of cograph sandwiches as a corollary of this algorithm.

In general, as soon as some dynamic data structure satisfying our requirements on the tool boxes B_1 and B_2 (see page 9) is provided, our general algorithmic scheme will apply. We hope that this technique could be helpful to solve other problems, e.g. with common strongly connected components, and be extended to probabilistic algorithms on problems of very large size.

References

[1] L. Alonso, E. Reingold, and R. Schott. Multidimensional divide-and-conquer maximin recurrences. *SIAM Journal on Discrete Mathematics*, 8(3):428–447, 1995.

[2] 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.

[3] A. Bergeron, C. Chauve, F. de Montgolfier, and M. Raffinot. Computing common intervals of k permutations, with applications to modular decomposition of graphs. In *13th Annual European Symposium on Algorithms (ESA05)*, volume 3669 of *LNCS*, pages 779–790, 2005.

[4] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Science*, 7(2):36–44, 1973.

[5] C. Bornstein, C.M.H. de Figueiredo, and V.G.P. de Sá. The pair completion algorithm for the homogeneous set sandwich problem. *Information Processing Letters*, pages 87–91, 2006.

[6] F. Boyer, A. Morgat, L. Labarre, J. Pothier, and A. Viari. Syntons, metabolons and interactons: an exact graph-theoretical approach for exploring neighbourhood between genomic and functional data. *Bioinformatics*, 21(23):4209–4215, 2005.

[7] B.-M. Bui Xuan, M. Habib, and C. Paul. Revisiting T. Uno and M. Yagiura's Algorithm. In *16th International Symposium of Algorithms and Computation (ISAAC05)*, volume 3827 of *LNCS*, pages 146–155, 2005.

[8] M.R. Cerioli, H. Everett, C.M.H. de Figueiredo, and S. Klein. The homogeneous set sandwich problem. *Information Processing letters*, 67:31–35, 1998.

[9] M. Chein, M. Habib, and M.C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37(1):35–50, 1981.

[10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[11] F. Coulon and M. Raffinot. Identification of maximal common connected sets of interval graphs and tree forests. In *1st International Conference on Algorithms and Computational Methods for Biochemical and Evolutionary Networks (CompBioNets'04)*, 2004. *to appear.*

[12] F. Coulon and M. Raffinot. Fast algorithms for identifying maximal common connected sets of interval graphs. *Discrete Applied Mathematics*, 154(12):1709–1721, 2006.

[13] D. Eppstein, G. Italiano, R. Tamassia, R. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *Journal of Algorithms*, 13:33–54, 1992.

[14] A.-T. Gai, M. Habib, C. Paul, and M. Raffinot. Identifying common connected components of graphs. Technical Report RR-LIRMM-03016, LIRMM, Université de Montpellier II, 2003.

[15] M.C. Golumbic, H. Kaplan, and R. Shamir. Graph sandwich problems. *Journal of Algorithms*, 19:449–473, 1995.

[16] M. Habib, C. Paul, and M. Raffinot. Common connected components of interval graphs. In *15th Annual Symposium on Combinatorial Pattern Matching (CPM04)*, volume 3109 of *LNCS*, pages 347–358, 2004.

[17] M. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *27th Symposium on Theory of Computing*, volume 787, pages 519–527, 1995.

[18] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, $2-$edge, and biconnectivity. In *30th Annual ACM Symposium on Theory of Computation*, pages 79–89, 1998.

[19] Z. Li and E. Reingold. Solution of a divide-and-conquer maximin recurrence. *SIAM Journal on Computing*, 18(6):1188–1200, 1989.

[20] R. Lipton and R. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9(3):615–627, 1980.

[21] K. Mehlhorn. *Data Structures and Efficient Algorithms.* Springer Verlag, EATCS Monographs, 1984.

[22] D. Seinsche. On a property of the class of n-colorable graphs. *Journal of Combinatorial Theory Series B*, pages 191–193, 1974.

[23] T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.