# Divide and Conquer Revisited. Application to Graph Algorithms 

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

## To cite this version:

Binh-Minh Bui-Xuan, Michel Habib, Christophe Paul. Divide and Conquer Revisited. Application to Graph Algorithms. 04027, 2004, pp.14. lirmm-00109200

## HAL Id: lirmm-00109200

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00109200
Submitted on 24 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Divide and Conquer Revisited Application to Graph Algorithms 

Binh Minh Bui Xuan, Michel Habib, and Christophe Paul<br>CNRS - LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5, France, \{buixuan, habib, paul\}@lirmm.fr


#### Abstract

Divide-and-conquer is a seminal paradigm of computer science that can be summarised as divide the problem into subproblems, conquer (solve) the subproblem and combine the partial solutions. Without any specific assumptions on the size of the subproblems, it enables to design quadratic time worst case bound algorithms. Well-known algorithms (e.g. median search [3]) propose to minimise the recursive computation in order to yield linear time. Up to our knowledge, no known method proposes to cut down the divide-and-combine part. This paper show that doing so quadratic time can also be improved. As an example of application, the Common Connected Problem is considered (a problem arising from computational biology [2]). Given a pair of graphs $G_{1}$ and $G_{2}$ on the same vertex set $V$, it consists of finding the coarsest partition of $V$ such that each part induces a connected subgraph of both $G_{1}$ and $G_{2}$. Using a divide-and-conquer approach, we propose a generic algorithm that, depending on the data-structure, can be used as well for arbitrary graphs, interval graphs and planar graphs. This algorithm equals the best known complexity bounds for the two former cases $[6,8]$ and improves the planar case by a $\log n$ factor.


## 1 Introduction

The political proverb divide ut imperes is now a fundamental strategy in computer science; the best known examples probably are standard sorting algorithms such as Quicksort or Mergesort, dynamic programming algorithms...Divide-and-conquer paradigms are recursive and involve three steps level of the recursion: it divides the problem into subproblems; conquers the subproblems (solves them); and combines the solutions to subproblems into a global solution.

Formally, let be given a problem $\mathcal{P}$ over some set $\mathcal{S}$ of data structures, along with a function Size that maps a data structure $S \in \mathcal{S}$ to some arbitrary number $\operatorname{Size}(S)$ - for instance, the size $\operatorname{Size}(G)$ of a graph may ignore the number of vertices and only count the number of edges. An algorithm $\mathcal{H}$ is defined as a divide-and-conquer algorithm for $\mathcal{P}$ with respect to Size, or DaC algorithm for short when no confusion occurs, if $\mathcal{H}$ takes $S \in \mathcal{S}$ in the input; and, unless $S$ is a trivial case for $\mathcal{P}$, namely unless $S \in \operatorname{Triv}^{\mathcal{H}}(\mathcal{S})$, divides it into a set $\mathbb{S}^{\mathcal{H}}(S)$ of subsets of $S$, then conquers, i.e. possibly performs some recursive calls over a subset of $\mathbb{S}^{\mathcal{H}}(S)$; and finally combines the results to the output. If $C_{\text {total }}^{\mathcal{H}}(S)$
denotes the total cost for solving $\mathcal{P}$ over $S$ by $\mathcal{H}, \operatorname{Rec}^{\mathcal{H}}(S)$ time for recursive calls over $\mathbb{S}^{\mathcal{H}}(S)$, and $D C^{\mathcal{H}}(S)$ the divide and combine time, then the total cost for solving $\mathcal{P}$ over $S$ with $\mathcal{H}$ can be formalised by induction as follows (see [4]).

$$
\begin{cases}C_{\text {total }}^{\mathcal{H}}(S)=1 & \text { if } S \text { is a trivial case for } \mathcal{P}  \tag{1}\\ C_{\text {total }}^{\mathcal{H}}(S)=D C^{\mathcal{H}}(S)+\operatorname{Rec}^{\mathcal{H}}(S) & \text { otherwise }\end{cases}
$$

The last inductive relation contains two sum-terms, implying that the total computing time for divide-and-conquer algorithms can be lightened by at least two distinct approaches. Since the costs of divide-and-conquer methods are mainly though as dependant on the cost for a data of lesser size, the second sumterm, related to the number of recursive calls, is usually optimised to decrease the complexity. In this framework, linearity can be achieved when one can avoid the recursive calls on a fraction of $S$ (see e.g. the median finding algorithm [3] or algorithms deriving from the planar separator theorem [10]) Such examples mostly do not care about the first sum-term, since minimising the recursive part turns out to be an efficient way to lessen the total cost for divide-and-conquer algorithms.

For a survey on divide-and-conquer methods see standard textbooks on algorithms $[4,12]$. Our works mainly lay on the alternative of saving the divide and combine time in a given divide-and-conquer algorithm, when no minimising on recursive calls might easily occur. More specifically, we investigate an "avoid the largest" idea under the formal framework of divide-and-conquer algorithms. A divide-and-conquer algorithm $\mathcal{H}$ is avoid-the-largest- $D C$ when the divide and combine time for any $\mathbb{S}^{\mathcal{H}}(S)=\left(S_{i}\right)_{i \in \llbracket 1, k \rrbracket}$ is proportional to $\operatorname{Size}(S)-\operatorname{Size}\left(S_{i_{0}}\right)$, where $i_{0}$ is such that $\operatorname{Size}\left(S_{i_{0}}\right)$ is the largest among $\left(\operatorname{Size}\left(S_{i}\right)\right)_{i \in \llbracket 1, k \rrbracket}$. In this case, $\Theta(\operatorname{Size}(S) \log \operatorname{Size}(S))$ computing time is obtained even if recursive steps have to be applied on all parts. It could seem hard and tricky at first sight to design decomposition algorithms that avoid at each recursive step to consider the largest part (and therefore that do not scan the entire data), but an algorithm achieving this requirement will afterwards be presented.

Indeed, as an application this paper presents a method, called competitive search, which enables us to solve the Common Connected Problem (CCP for short). Given a pair of graph $G_{1}=\left(V, E_{1}\right), G_{2}=\left(V, E_{2}\right)$ on the same vertex set, $C C P$ aims to compute a partition of $V$ into subsets $\left(V_{1}, \ldots V_{k}\right)$ such that any $V_{i}$ is a maximal subset of vertices such that both induced subgraphs $G_{1}\left[V_{i}\right]$ and $G_{2}\left[V_{i}\right]$ are connected. $C C P$ arises in theoretical computer science since it is strongly related to the modular decomposition problem [11, 7] but also in applied fields, namely computational biology (interested reader should refer to [2, 8] for this aspect). The algorithm we present achieves the best known complexity in case of arbitrary graphs (namely $\mathcal{O}\left((n+m) \log ^{2} n\right)$ [6]) and interval graphs $(\mathcal{O}((n+m) \log n)[8])$ and improves the complexity in case of planar graphs to $\mathcal{O}((n+m) \log n)$.

The paper is organised as follows. Section 2 proposes the analysis of the "avoid-the-largest-DC" divide-and-conquer algorithm. Section 3 depicts the example of competitive search to illustrate the "avoid-the-largest-DC" paradigm.

The competitive search is then used in Section 4, which is dedicated to $C C P$ resolution.

## 2 Minimising the Divide-and-Combine Time

Divide-and-conquer techniques have numerous variants. The aim of this paper is not to review them. Indeed it focuses in the case when divisions do not "supersize" the sub-instances, i.e. when axiom $\mathcal{A}_{\text {div }}$ below holds.

$$
\mathcal{A}_{d i v}(\mathcal{H}): \forall S \in \mathcal{S} \backslash \operatorname{Triv}^{\mathcal{H}}(\mathcal{S}),\left\{\begin{array}{l}
\mathbb{S}^{\mathcal{H}}(S)=\left(S_{1}, S_{2}, \ldots, S_{k}\right)  \tag{2}\\
\forall i \in \llbracket 1, k \rrbracket, \quad S_{i} \in \mathcal{S} \wedge \operatorname{Size}\left(S_{i}\right) \neq 0 \\
\operatorname{Size}(S)=\operatorname{Size}\left(S_{1}\right)+\ldots+\operatorname{Size}\left(S_{k}\right)
\end{array}\right.
$$

In this case, a naive bound in $\mathcal{O}\left(D C^{\mathcal{H}}(S)^{2}\right)$ is easily obtained when analysing $\mathcal{H}(S)$ worst case computing time. To improve the quadratic bound, common techniques consist of minimising the recursive part $R e c^{\mathcal{H}}$, which most of the time leads to linear - on the size of $D C^{\mathcal{H}}(S)$ - worst case analysis, leaving no real need for coarser minimising the divide and combine time. Unfortunately, minimising $R e c^{\mathcal{H}}$ sometimes might be intricate and difficult to realise. In this case, it somehow becomes crucial to investigate the "incomplete divide and combine time" paradigm.

Let be given a divide-and-conquer algorithm $\mathcal{H}$ holding $\mathcal{A}_{\text {div }}(\mathcal{H})$ with respect to Size. We define $\mathcal{H}$ to be completely-recursive and avoid-the-largest- $D C$ when there exists a constant $\alpha^{\mathcal{H}}$ such that, for all non-trivial input $S$, if $S_{i_{0}}$ is the largest with respect to Size among $\mathbb{S}^{\mathcal{H}}(S)=\left(S_{1}, S_{2}, \ldots, S_{k}\right)$, then

$$
\begin{aligned}
& D C^{\mathcal{H}}(S) \leq \alpha^{\mathcal{H}} \times\left(\sum_{i \in \llbracket 1, k \rrbracket, i \neq i_{0}} \operatorname{Size}\left(S_{i}\right)\right), \\
& \operatorname{Rec}^{\mathcal{H}}(S)=\sum_{i \in \llbracket 1, k \rrbracket}^{k} C_{\text {total }}^{\mathcal{H}}\left(S_{i}\right) .
\end{aligned}
$$

Theorem 1 (Avoid-the-Largest-DC Computing Time). Any divide-andconquer algorithm, satisfying axiom $\mathcal{A}_{\text {div }}$, with respect to Size that is completelyrecursive and avoid-the-largest-DC runs in $\alpha^{\mathcal{H}} \times \operatorname{Size}(S) \log \operatorname{Size}(S)$ worst case time where $S$ is the given input. The upper bound is reached.
Proof. Let be given a completely-recursive and avoid-the-largest-DC $D a C$ algorithm $\mathcal{H}$, we prove by induction on $\operatorname{Size}(S)$ the following.

$$
\forall S \in \mathcal{S}, C_{\text {total }}^{\mathcal{H}}(S) \leq \alpha^{\mathcal{H}} \times \operatorname{Size}(S) \log \operatorname{Size}(S)
$$

Indeed, if $S$ is not trivial, let $\mathbb{S}^{\mathcal{H}}(S)=\left(S_{i}\right)_{i \in \llbracket 1, k \rrbracket}, k$ be such that $\operatorname{Size}\left(S_{k}\right)$ is the largest among $\left(\operatorname{Size}\left(S_{i}\right)\right)_{i \in \llbracket 1, k \rrbracket}$, and $n_{i}=\operatorname{Size}\left(S_{i}\right)$. Then,

$$
C_{\text {total }}^{\mathcal{H}}(S)=\alpha^{\mathcal{H}} \times \sum_{i=1}^{k-1} n_{i}+\sum_{i=1}^{k} C_{\text {total }}^{\mathcal{H}}\left(S_{i}\right)
$$

Besides, for $n_{k}$ is largest among $\left(n_{i}\right)_{i \in \llbracket 1, k \rrbracket}$,

$$
\begin{aligned}
& \forall 1 \leq i \leq k-1, \quad n_{i} \leq \frac{\operatorname{Size}(S)}{2} \\
& \text { Hence, } \forall 1 \leq i \leq k-1, \log n_{i} \leq \log \operatorname{Size}(S)-1
\end{aligned}
$$

Combining these to the inductive hypothesis results in:

$$
\begin{aligned}
C_{\text {total }}^{\mathcal{H}}(S) & \leq \alpha^{\mathcal{H}} \times\left(\sum_{i=1}^{k-1} n_{i}+\sum_{i=1}^{k} n_{i} \log n_{i}\right) \\
& \leq \alpha^{\mathcal{H}} \times\left(\sum_{i=1}^{k-1} n_{i}+\sum_{i=1}^{k-1} n_{i} \log n_{i}+n_{k} \log n_{k}\right) \\
& \leq \alpha^{\mathcal{H}} \times\left(\sum_{i=1}^{k-1} n_{i}+\sum_{i=1}^{k-1} n_{i}(\log \operatorname{Size}(S)-1)+n_{k} \log \operatorname{Size}(S)\right) \\
& \leq \alpha^{\mathcal{H}} \times \operatorname{Size}(S) \log \operatorname{Size}(S) .
\end{aligned}
$$

Finally, let be constructed a sequence $\left(S_{n}\right)_{n \in \mathbb{N}}$ of instances for $\mathcal{P}$ as follows. $S_{0}$ is a trivial case for $\mathcal{P}$. For all $i \geq 1, S_{i}$ is such that $\mathcal{H}$ divides $S_{i}$ into two sub-instances that are both identical to $S_{i-1}$. Thus, every element $S_{i}$ in $\left(S_{n}\right)_{n \in \mathbb{N}}$ is such that $\mathcal{H}$ spends at least $\alpha^{\mathcal{H}} \times \operatorname{Size}\left(S_{i}\right) \log \operatorname{Size}\left(S_{i}\right)$ solving $\mathcal{P}$ on $S_{i}$.

Remark 1. As opposed to median search [3] or planar separator techniques [10], in which the size of the input along recursive calls diminish geometrically, yielding a logarithmic bound on the recursion depth, it is noteworthy that the above complexity is reached even if the recursion depth of such a method can be linear in $\operatorname{Size}(S)$.

## 3 Competitive Graph Searches do Avoid the Biggest Divide and Combine Time

This section focuses on a particular graph search algorithm, namely the competitive graph search, that identifies any connected component but the largest without exploring entirely the graph. That algorithm ables us to answer the "avoid the largest" in the divide and combine step and therefore follows the settings of Theorem 1. First of all, notice that the size of a connected component is best represented by the number of its edges.

Roughly, a classical search on a given graph begins by selecting some arbitrary vertex, then stays within the connected component that contains the selected vertex until the whole component is identified. Therefore, applying a classical graph search for identifying the small connected components might require investigating the whole initial graph, for vertex selections are haphazard.

A competitive search can be launched if a pointer to each connected component is available. Dealing with a given connected component, whenever one
of its unvisited edge is identified, the search on that component is postponed until one new edge is identified for each of the remaining components. Once all the edges of a component have been visited, that component is removed from the competitive search. The algorithm continues as long as there are at least two remaining components. Obviously the last connected component $C$ is the largest one and has not been entirely visited by the search. Indeed, if $m^{\prime}$ is the number of edges of the second largest connected component, then only $m^{\prime}$ edges of $C$ have been discovered.

Lemma 1. Given a pointer per connected component of a graph $G$, a competitive search identifies the connected components but the largest, say $C$, in time 2. ( $m_{G}-$ $m_{C}$ ) where $m_{G}$ and $m_{C}$ are respectively the number of edges of $G$ and $C$.

Remark 2. Using such a competitive search to answer the "avoid-the-largestDC" aspect of Theorem 1 leads to constant $\alpha^{\mathcal{H}}=2$.

In the following, the list of pointers required by the competitive search will be a list, namely Rep, of vertices in $G$ such that every connected component in $G$ has exactly one representative vertex in Rep. And GET_SMALL_SUBGRAPHS ( $G, R e p$ ) will denotes the competitive graph search.

## 4 Bringing Competitiveness to the $\boldsymbol{C C P}$

Let us first recall that the Common Connected Problem, given two graphs $G_{1}$ and $G_{2}$ on the same vertex set $V$, aims to identify maximal subsets of vertices that induce connected subgraphs of both $G_{1}$ and $G_{2}$. As explained in [6], solving $C C P$ may be restricted to pair of graphs with disjoint edge sets. Moreover let us assume w.l.o.g. that one of the two graphs, say $G_{1}$, is connected. If $G_{2}$ is also connected, then such an instance is trivial. Lemma $2[6]$ is the basis of the algorithm and allows us to use the competitive search.

Lemma 2. [6] Let $\mathcal{G}=\left(G_{1}, G_{2}\right)$ be given as a pair of graphs over the same vertex set $V$. If $X \subset V$ is such that there exists one graph $G_{i} \in \mathcal{G}$ where no edge can be found between $X$ and $V \backslash X$ - for instance, if $X$ is a connected component in $G_{i}$ - then:

$$
C C P(\mathcal{G})=C C P(\mathcal{G}[X]) \cup C C P(\mathcal{G}[V \backslash X]),
$$

$G_{i}[Y]$ is the subgraph of $G_{i}$ induced by $Y$, and $\mathcal{G}[Y]$ denotes $\left(G_{1}[Y], G_{2}[Y]\right)$.
Let be given $\mathcal{G}=\left(G_{1}, G_{2}\right)$ as a non-trivial instance of the $C C P$ and satisfying the above hypothesis. Let $V_{1}, V_{2}, \ldots, V_{k}$ stand for the connected components of $G_{2}$. Lemma 2 motivates dividing $\mathcal{G}$ into $\mathcal{G}\left[V_{1}\right], \ldots, \mathcal{G}\left[V_{k}\right]$. Assuming a graph search has already collected a list containing one representative per $V_{i}$, such division step, which consists of removing in $G_{1}$ the edges linking vertices of different $V_{i}$ 's (henceforth a dynamic data-structure for maintaining connectivity is required, see next subsection), can be done in an "avoid-the-largest-DC" manner

$$
\begin{array}{r}
-\left(\mathrm{G}_{\mathrm{i}}^{\mathrm{i}}\right)_{\mathrm{i}>1} \\
\cdots \quad\left(\mathrm{G}_{2}^{\mathrm{i}}\right)_{\mathrm{i} i>1}
\end{array}
$$

$$
G_{2}=\left(\mathrm{G}_{1}^{2}, \mathrm{G}_{2}^{2}\right):
$$



Fig. 1. A sequence $\left(\mathcal{G}_{i}\right)_{i>2}=\left(\left(G_{1}^{i}, G_{2}^{i}\right)\right)_{i>2}$ of tricky instances of $C C P$, yielding the maximal complexity to the divide-and conquer scheme for solving $C C P$. i. Recursive definition for $\left(\mathcal{G}_{i}\right)_{i>2}$. ii. Example of $\mathcal{G}_{4}$.
as follows: 1) using a competitive search, select the $k-1$ smallest components, say $\left.V_{1}, \ldots V_{k-1} ; 2\right)$ using a graph search extract $\left.\mathcal{G}\left[V_{1}\right], \ldots, \mathcal{G}\left[V_{k-1}\right] ; 3\right) \mathcal{G}\left[V_{k}\right]$ can be obtained by removing from $\mathcal{G}$ the vertices of $V \backslash V_{k}$ together with their incident edges. These three steps can all be done in time linear in the number of edges incident to $\bigcup_{1 \leq i \leq k-1} V_{i}$. To prepare the next recursion step, a representative of each connected component of the induced subgraphs $G_{1}\left[V_{i}\right](1 \leq i \leq k)$ has to be collected. For the subgraphs $G_{1}\left[V_{i}\right]$, for $i<k$, a graph search can be afforded since they are of size small enough. The main trick concerns $G_{1}\left[V_{k}\right]$.

Let $\bar{V}_{k}$ denote $V \backslash V_{k}=\bigcup_{1 \leq i \leq k-1} V_{i}$. Since $G_{1}$ is connected, the subset of vertices of $V_{k}$ neighbouring some vertex in $\bar{V}_{k}$ contains at least one vertex per connected component of $G_{1}\left[V_{k}\right]$. This set can be computed in time proportional to the number of edges incident to $\bar{V}_{k}$ by routine GET_OUTGOING_VERTICES defined afterwards. Then, using the dynamic data-structure for graph connectivity, a cleaning will be performed by routine FILTER to save only one representative per connected component. Finally the algorithm go ahead recursively by alternating the respective roles of $G_{1}$ and $G_{2}$ depicted above.

Before presenting the whole algorithm, let us first describe the dynamic datastructures and the various routines involved in its pseudo-code.

Dynamic Data Structure for Connectivity Maintenance We bid our reader to notice that, with a certain degree of abusiveness, a dynamic data structure sometimes is blended together with its represented graph. The dynamic data structure should allow three following queries.

- INIT $(G)$ : input a graph $G$, output a data structure $D S$ representing $G$.
- UPDATE $(D S$, DELETION, $E)$ : input a data structure $D S$ and a list $E$ of edges in $D S$, delete in $D S$ all edges in $E$, and output the remaining of $D S$.
- GET_COMPONENT $(D S, v)$ : input a data structure $D S$ and a vertex $v$ in $D S$, output the connected component that contains $v$.

The use of the dynamic data-structure (in the initialisation step, in the FILTER routine and along the updates) implies an extra cost $C_{\text {extra }}$. Depending on the inquired data-structure $C_{\text {extra }}$ may vary. For instance, the best known $\mathcal{O}\left((n+m) \log ^{2} n\right)$ worst case complexity for solving $C C P$ over arbitrary graphs [6] has been reached using ET-trees dynamic data structure [9]. Actually, $C_{\text {extra }}$ is the only responsible for the second logarithmic factor in the global computing time. This highly motivates looking for dynamic data structures with lower maintaining cost as clique-path representation of interval graph does [8]. Edgeordered dynamic tree [5] devoted to represent plane graphs also allows to solve $C C P$ over planar graphs with a low $C_{\text {extra }}$. It turns out that our algorithm can be seen as a generic algorithm for all these three different cases. Moreover the mixing different dynamic data-structures is allowed (e.g. for $G_{1}$ being an interval graph and $G_{2}$ a planar graph).

Basic routines Let us briefly sketch the primary routines. For convenience, if $\left(V_{1}, V_{2}, \ldots, V_{p}\right)$ is a collection of disjunctive subsets of $V(G)$, we define the set of "bridges" in $G$ between $\left(V_{i}\right)_{i \in \llbracket 1, p \rrbracket}$ as

$$
\text { InterEdges }\left(G,\left(V_{i}\right)_{i \in \llbracket 1, p \rrbracket}\right)=\left\{(u, v) \in E(G) \mid \exists i \neq j, u \in V_{i} \wedge v \in V_{j}\right\}
$$

- GET_REPRESENTATIVES $(G)$ : thanks to a classical graph search over $G$, compute a list with exactly one vertex per connected component in $G$.
- GET_OUTGOING_EDGES $(G, X)$ : thanks to a classical graph search over $G[X]$, compute Inter Edges $(G,(X, V \backslash X))$.
- GET_OUTGOING_VERTICES $(G, X)$ : thanks to a classical graph search over $G[X]$, compute all extremities in $\operatorname{Inter\operatorname {Edges}}(G,(X, V \backslash X))$ that are exterior to X.

The first procedure runs in linear time on the size of the input graph. The two last procedures compute in time proportional to

$$
|X|+|E(G[X])|+\mid \text { InterEdges }(G,(X, V \backslash X)) \mid
$$

Given a list of vertices containing at least one vertex per connected component of some graph, FILTER routine works as follows: 1) it queries the dynamic data-structure to get an identifier (e.g. an integer) of the connected component of each element of the list; 2) the elements of the list are sorted with respect to the identifier of its connected component; 3) by a scan, the list is finally filtered to save only one element per connected component.

We are now able to depict the recursive core of our algorithm (see procedure MAIN).

INPUT : ( $\mathcal{G}, i$, Representatives, $\mathcal{D S})$ : quadruplet, where
$i$ is a number belonging to $\{1,2\}$ - we define $\bar{i}=3-i$;
$\mathcal{G}=\left(G_{1}, G_{2}\right)$ a pair of graphs over the same vertices set s.t.
$G_{\bar{i}}$ is connected and $E\left(G_{1}\right) \cap E\left(G_{2}\right)=\emptyset$;
Representatives a list containing exactly one vertex per connected component in $G_{i}$;
and $\mathcal{D S}=\left(D S_{1}, D S_{2}\right)$ data structures representing $\mathcal{G}$.
OUTPUT: Result: list of maximal common connected components in $\mathcal{G}$.
// The trivial and terminating case: $G_{i}$ is also connected
If $\mid$ Representatives $\mid=1$ then Result $\leftarrow[V]$;
Else
Result $\leftarrow[]$;
$\left(V_{1}, V_{2}, \ldots, V_{k-1}\right) \leftarrow$ GET_SMALL_SUBGRAPHS $\left(G_{i}\right.$, Representatives $)$; // Proceeding small subgraphs in time prop. to their size Forall $j \in \llbracket 1, k-1 \rrbracket$ do:

Compute $\mathcal{G}\left[V_{j}\right]=\left(G_{1}\left[V_{j}\right], G_{2}\left[V_{j}\right]\right)$ from the knowledge of $\left(\mathcal{G}, V_{j}\right)$;
Representatives $\leftarrow$ GET_REPRESENTATIVES $\left(G_{\bar{i}}\left[V_{j}\right]\right)$;
OutgoingEdges $\leftarrow$ GET_OUTGOING_EDGES $\left(G_{\bar{i}}, V_{j}\right)$;
UPDATE ( $D S_{\bar{i}}$, DELETION, OutgoingEdges);
Compute $\mathcal{D S}\left[V_{j}\right]$ from the knowledge of ( $\mathcal{D S}, V_{j}$ );
Result $\leftarrow$ Result @ $\operatorname{main}\left(\mathcal{G}\left[V_{j}\right], \bar{i}\right.$, Representatives, $\left.\mathcal{D} \mathcal{S}\left[V_{j}\right]\right)$; EndFor
// Coping with the biggest subgraph in time proportional to //the total size of the small subgraphs
$\overline{V_{k}} \leftarrow V_{1} \cup V_{2} \cup \ldots V_{k-1} ;$
Compute $\mathcal{G}\left[V_{k}\right]$ from the knowledge of ( $\mathcal{G}, \overline{V_{k}}$ );
OutgoingVertices $\leftarrow$ GET_OUTGOING_VERTICES $\left(G_{\bar{i}}, \overline{V_{k}}\right)$;
Compute $\mathcal{D S}\left[V_{k}\right]=\left(D S_{1}\left[V_{k}\right], D S_{2}\left[V_{k}\right]\right)$ knowing ( $\left.\mathcal{D S}, \overline{V_{k}}\right)$;
Representatives $\leftarrow \operatorname{FILTER}\left(\right.$ OutgoingVertices, $\left.D S_{\bar{i}}\left[V_{k}\right]\right)$;
Result $\leftarrow$ Result @ $\operatorname{maIN}\left(\mathcal{G}\left[V_{k}\right], \bar{i}\right.$, Representatives, $\left.\mathcal{D S}\left[V_{k}\right]\right)$;
EndElse
20. Return Result.

Complexity Issues To analyse the computing time $C_{\text {total }}(\mathcal{G})$ of our algorithm, we need more knowledge on the inquired dynamic data structures. Thus, we define $C_{I N I T}(G)$ as the cost of $\operatorname{INIT}(G), C_{C O M P}(D S)$ of $\operatorname{GET}$ COMPONENT $(D S, v)$, and $C_{D E L}(D S)$ of UPDATE $(D S$, DELETION, $E)$, when $E$ is reduced to a singleton.

As before, we differentiate in $C_{\text {total }}(\mathcal{G})$ a variable $\operatorname{cost} C_{\text {extra }}(\mathcal{G})$ due to queries to the dynamic data structures from the remaining $C_{\text {core }}(\mathcal{G})$. Three terms are contained within $C_{\text {extra }}(\mathcal{G})$ : the cost $C_{\text {Init }}(\mathcal{G})$ for initialising the dynamic data structures, the total cost $C_{\text {Filter }}(\mathcal{G})$ for calls to FILTER, and the total cost $C_{\text {Maintenance }}(\mathcal{G})$ due to edge-deletions. Concisely:

$$
\begin{aligned}
C_{\text {total }}(\mathcal{G}) & =C_{\text {extra }}(\mathcal{G})+C_{\text {core }}(\mathcal{G}) \\
C_{\text {extra }}(\mathcal{G}) & =C_{\text {Init }}(\mathcal{G})+C_{\text {Filter }}(\mathcal{G})+C_{\text {Maintenance }}(\mathcal{G})
\end{aligned}
$$

Lemma 3. Let be given an instance $\mathcal{G}=\left(G_{1}, G_{2}\right)$, along with $m=\left|E\left(G_{1}\right)\right|+$ $\left|E\left(G_{2}\right)\right|$, and $D S_{i}=\operatorname{INIT}\left(G_{i}\right)$ for $i \in\{1,2\}$. Then,

$$
\begin{aligned}
C_{\text {Init }}(\mathcal{G}) & =C_{I N I T}\left(G_{1}\right)+C_{I N I T}\left(G_{2}\right) \\
C_{\text {Filter }}(\mathcal{G}) & =\mathcal{O}\left(m \times\left(C_{C O M P}\left(D S_{1}\right)+\log m\right)\right) \\
& =\mathcal{O}\left(m \times\left(C_{C O M P}\left(D S_{2}\right)+\log m\right)\right) \\
C_{\text {Maintenance }}(\mathcal{G}) & =\mathcal{O}\left(m \times C_{D E L}\left(D S_{1}\right)\right)=\mathcal{O}\left(m \times C_{D E L}\left(D S_{2}\right)\right)
\end{aligned}
$$

We now evaluate $C_{\text {core }}(\mathcal{G})$. Though routine MAIN is an avoid-the-largest-DC $D a C$ algorithm, we introduce a function Size as follows: if $\mathcal{G}=\left(G_{1}, G_{2}\right)$ is the instance, then $\operatorname{Size}(I)=\left|E\left(G_{1}\right)\right|+\left|E\left(G_{2}\right)\right|$, where $I$ is the corresponding input given to MAIN routine.
Lemma 4. Let be given $\mathcal{G}=\left(G_{1}, G_{2}\right)$ as an instance. Then,

$$
C_{\text {core }}(\mathcal{G})=\mathcal{O}(m \log m) \text { where } m=\left|E\left(G_{1}\right)\right|+\left|E\left(G_{2}\right)\right|
$$

Proof. When ignoring the processing time of lines 9 and 17, which already is taken into account within $C_{\text {extra }}$, MAIN meets the definition of an avoid-the-largest-DC $D a C$ algorithm satisfying axiom $\mathcal{A}_{\text {div }}$ with respect to Size.

Indeed let us first prove that MAIN is a $D a C$ algorithm satisfying axiom $\mathcal{A}_{\text {div }}$ with respect to Size. Unless its input $I=(\mathcal{G}, i$, Representatives, $\mathcal{D S})$ is trivial (line 1), the procedure divides $V$, set of vertices in $\mathcal{G}=\left(G_{1}, G_{2}\right)$, into a partition $V_{1}, \ldots, V_{k-1}, V_{k}$, where $V_{k}=V \backslash\left(V_{1} \cup \ldots \cup V_{k-1}\right)$ (line 4). This operation implicitly decomposes the input $I$ into $I_{1}, \ldots, I_{k}, I_{k+1}$ defined as follows.

$$
\begin{aligned}
\forall j \in \llbracket 1, k \rrbracket, \quad I_{j} & =\left(\mathcal{G}\left[V_{j}\right], i, \text { Representatives, } \mathcal{D} \mathcal{S}\left[V_{j}\right]\right) \\
I_{k+1} & =\left(\left(\text { Quotient } G_{1}, \text { Quotient }_{2}\right),_{-},{ }_{-}\right),
\end{aligned}
$$

where Quotient $G_{i}=\left(V\right.$, InterEdges $\left.\left(G_{i},\left(V_{j}\right)_{j \in \llbracket 1, k \rrbracket}\right)\right)$ for $i \in\{1,2\}$, and the symbol_ stands for dummies. Thus, by constructions of Size and $\left(I_{i}\right)_{i \in \llbracket 1, k+1 \rrbracket}$, the non redundant divisions axiom $\mathcal{A}_{\text {div }}$ holds for MAIN. Furthermore, MAIN makes recursive calls (lines 11 and 18) to solve $\left(I_{j}\right)_{j \in \llbracket 1, k \rrbracket}$. As for $I_{k+1}$, which is a trivial non-productive case, its resolution is done by being skipped. Finally, MAIN
merges, in insignificant time, the corresponding results (lines 11 and 18) to its output (line 20). Hence, MAIN is a $D a C$ algorithm with respect to Size with an insignificant merging time.

We now compute the cost of dividing $I$, when $I$ is not trivial, and achieve proving that MAIN is avoid-the-largest-DC. Notice that the largest element among $\left(I_{j}\right)_{j \in \llbracket 1, k+1 \rrbracket}$ is $I_{k}$. First, in MAIN, lines 2-7, 10-14, 16 and 18-19 can be computed in time proportional to $\operatorname{Size}(I)-\operatorname{Size}\left(I_{k}\right)-\operatorname{Size}\left(I_{k+1}\right)=\operatorname{Size}\left(I_{1}\right)+\operatorname{Size}\left(I_{2}\right)+$ $\ldots+\operatorname{Size}\left(I_{k-1}\right)$. Now, ignoring lines 9 and 17, there still are two lines left: 8 and 15 . Fortunately, their computing time is proportional to

$$
\sum_{i=1}^{k-1} \operatorname{Size}\left(S_{i}\right)+\operatorname{Size}\left(S_{k+1}\right) .
$$

Since obviously $I_{k}$ is the largest instance, we can afford the former computing time and MAIN is an avoid-the-largest-DC $D a C$ algorithm.

The lemma then follows from Theorem 1.

|  | best so far | our algorithm | conjecture |
| :--- | :---: | :---: | :---: |
| forests of <br> disjunctive paths | $\mathcal{O}\left(n+\right.$ " $\left.K^{\prime \prime}\right)[13]$ | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n)$ ? |
| arbitrary graphs | $\mathcal{O}\left(n \log n+m \log ^{2} n\right)[6]$ | $\mathcal{O}\left(n \log n+m \log ^{2} n\right)$ | $?$ |
| interval graphs | $\mathcal{O}((n+m) \log n)[8]$ | $\mathcal{O}((n+m) \log n)$ | $?$ |
| unit interval graphs | $\mathcal{O}(n \log \Delta \log n)[1]$ | $\mathcal{O}((n+m) \log n)=$ | $\mathcal{O}(n+m)=$ |
|  |  | $\mathcal{O}(\Delta n \log n)$ | $\mathcal{O}(\Delta n)$ ? |
| planar graphs | $\mathcal{O}\left(n \log ^{2} n\right)[6]$ | $\mathcal{O}(n \log n)$ | $?$ |
| forests of trees | $\mathcal{O}\left(n \log ^{2} n\right)[6]$ | $\mathcal{O}(n \log n)$ | $?$ |
| permutation graphs | $\mathcal{O}\left(n \log n+m \log ^{2} n\right)[6]$ | $\mathcal{O}\left(n \log n+m \log ^{2} n\right)$ | $\mathcal{O}((n+m) \log n) ?$ |

Fig. 2. Time complexity solving problem $C C P$ over two graphs with disjoint edge sets.

Theorem 2. Let be given $\mathcal{G}=\left(G_{1}, G_{2}\right)$ as an instance of $C C P$, along with $m=|E(\mathcal{G})|=\left|E\left(G_{1}\right)\right|+\left|E\left(G_{2}\right)\right|$ and $n=\left|V\left(G_{1}\right)\right|=\left|V\left(G_{2}\right)\right|$. Our CCP algorithm runs in $\mathcal{O}((n+m) \log n)$ worst case time if $\mathcal{G}$ is a pair of planar graphs, in $\mathcal{O}\left((n+m) \log ^{2} n\right)$ otherwise.

Proof. Both ET-trees [9] and edge-ordered dynamic tree [5] data structures yield $C_{I N I T}(G)=\mathcal{O}\left(\left(n_{G}+m_{G}\right) \log n_{G}\right)$ and $C_{C O M P}(D S)=\mathcal{O}\left(\log n_{D S}\right)$, where $n_{G}=$ $|V(G)|, m_{G}=|E(G)|$, and $n_{D S}$ is the size of $D S$, which can be the number
of vertices or the number of edges for both data structures are trees. Besides, $E T$-trees maintain $C_{D E L}(D S)=\mathcal{O}\left(\log ^{2} n\right)$ amortised cost per deletion, and edge-ordered dynamic tree maintain an amortised $C_{D E L}(D S)=\mathcal{O}(\log n)$.

Then, the theorem follows from Lemma 3 and 4 . Notice that $G_{1}$ is supposed to be connected, therefore $\mathcal{O}(n+m)=\mathcal{O}(m)$.

Correctness The correctness of the global algorithm follows from the one of MAIN, which will be the issue of Theorem 3. However, we need some preliminaries before stating the theorem.

Lemma 5. Let be given a connected graph $G=(V, E)$, along with a subset $X \subset V$ of its vertices. Then, each connected component in $G[V \backslash X]$ has at least one vertex that belongs to the list

$$
\text { OutgoingVertices = GET_OUTGOING_VERTICES }(G, X) \text {. }
$$

According to Lemma 5, whenever FILTER is inquired at line 17 in MAIN, its input (OutgoingVertices, $D S_{\bar{i}}\left[V_{k}\right]$ ) is such that $D S_{\bar{i}}\left[V_{k}\right]$ encodes $G_{\bar{i}}\left[V_{k}\right]$ and each connected component in $G_{\bar{i}}\left[V_{k}\right]$ has at least one representative vertex in list OutgoingVertices. Then, FILTER is to get rid of the extra-representatives from OutgoingVertices so that a competitive graph search can further be conducted in GET_SMALL_SUBGRAPHS. This is to be formalised in Lemmas 6 and 7 below.

Lemma 6. Let be given a dynamic data structure $D S$ that represents some graph, along with a list $X$ containing at least one representative vertex per connected component in the graph. Then, $\operatorname{FILTER}(X, D S)$ selects from $X$ exactly one representative vertex per connected component in the graph that $D S$ represents.

Lemma 7. Let be given a graph $G$; let $G^{1}, G^{2}, \ldots, G^{k}$ stand for the partitioning of $G$ into maximal connected subgraphs of $G$. If each graph among $\left(G^{i}\right)_{i \in \llbracket 1, k \rrbracket}$ has exactly one vertex some list Rep, then GET_SMALL_SUBGRAPHS(G, Rep) outputs $\left(V\left(G^{1}\right), V\left(G^{2}\right), \ldots, V\left(G^{k-1}\right)\right)$.

We now is ready for proving MAIN.

## Theorem 3.

1. Every possible input ( $\mathcal{G}, i$, Representatives, $\mathcal{D S}$ ) to MAIN holds the following.
i. $\mathcal{G}=\left(G_{1}, G_{2}\right)$ is a pair of planar graphs, where $V\left(G_{1}\right)=V\left(G_{2}\right)=V$ and $E\left(G_{1}\right) \cap E\left(G_{2}\right)=\emptyset$.
ii. $i \in\{1,2\}$ is such that $G_{\bar{i}}$ is connected.
iii. Representatives is a list of vertices such that every connected components in $G_{i}$ has exactly one representative among its elements.
iv. $\mathcal{D S}=\left(D S_{1}, D S_{2}\right)$ is a pair of data structures that respectively represent $G_{1}$ and $G_{2}$.
2. If the above properties hold, $\operatorname{MAIN}(\mathcal{G}, i$, Representatives, $\mathcal{D S})$ computes the maximal common connected components in $\mathcal{G}$.

Proof.

1. Mostly follows from Lemma 6.
2. By induction on the size $n$ of the vertex set.

If $n=1$, or if $G_{i}$ is connected, then $\mid$ Representatives $\mid=1$ and MAIN will returns $[V]$ (line 1). The lemma holds.

Let now be given a number $p$ such that the lemma holds for any possible input to MAIN where the two graphs share the same vertex set of size lesser than or equals to $p$. We suppose that $n=p+1$. As before, if $G_{i}$ is connected, the lemma holds. If $G_{i}$ is not connected, $\mid$ Representatives $\mid \geq 2$ and line 1 is skipped. Moreover, Lemma 7 states that, after line $4,\left(V_{1}, V_{2}, \ldots, V_{k-1}\right)$ are the corresponding non-empty vertex sets to the maximal connected subgraphs of $G_{i}$ but the biggest. For convenience, let $V_{k}$ stand for the remaining vertex set. Lemma 2 can be applied successively on $V_{1}, V_{2}, \ldots, V_{k-1}$ and results in the following.

$$
C C P(\mathcal{G})=C C P\left(\mathcal{G}\left[V_{1}\right]\right) \cup C C P\left(\mathcal{G}\left[V_{2}\right]\right) \cup \ldots \cup C C P\left(\mathcal{G}\left[V_{k}\right]\right)
$$

Since $\left|V_{i}\right| \leq p$ for all $1 \leq i \leq k$, we deduce from the inductive hypothesis that MAIN correctly computes all the sets $C C P\left(\mathcal{G}\left[V_{i}\right]\right)$ at lines 11 and 18. Therefore, lines 3-19 correctly compute $C C P(\mathcal{G})$, which is union of all the sets $C C P\left(\mathcal{G}\left[V_{i}\right]\right)$. The lemma holds.

Finally, the inductive reasoning over $V$ is exhibited in order to achieve the proof: the lemma holds when $|V|=1$; besides, for all $p \in \mathbb{N}^{+}$, if the lemma holds when $|V| \leq p$, the lemma holds when $|V|=p+1$; therefore, the lemma holds for all size $p \in \mathbb{N}^{+}$of the vertex set $V$.

## 5 Conclusion

We have presented here a generic algorithm which enlightens a new divide and conquer scheme. As a direct byproduct this scheme can be used to solve variations of the $C C P$ problem, such as Common Strongly Connected Components as soon as a dynamic data structure satisfying our requirements is provided.

We hope that it could be helpful to solve other problems and be extended to probabilistic algorithms on problems of very large size.

## References

1. Marie-Pierre Béal, Anne Bergeron, Sylvie Corteel, and Mathieu Raffinot. An algorithmic view of gene teams. Theoretical Computer Science, February 2004.
2. A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. In Workshop on Algorithms in Bioinformatics (WABI), number 2452 in Lecture Notes in Computer Science, pages 464-476. Springer Verlag, 2002.
3. M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. Journal of Computer and System Science, 7(2):36-44, 1973.
4. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms. Second Edition. The MIT Press, 2001.
5. David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert E. Tarjan, Jeffery Westbrook, and Moti Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. J. Algorithms, 13:33-54, 1992.
6. Anh-Tuan Gai, Michel Habib, Christophe Paul, and Mathieu Raffinot. Identifying common connected components of graphs. Technical Report RR-LIRMM-03016, LIRMM, Université de Montpellier II, July 2003.
7. M. Habib, F. de Montgolfier, and C. Paul. A simple linear-time modular decomposition algorithm. In T. hagerup, editor, Scandinavian Workshop on Algorithm Theory-SWAT04, number 3111 in Lecture Notes in Computer Science, pages 187198. 9th Scandinavian Workshop on Algorithm Theory, 2004.
8. M. Habib, C. Paul, and M. Raffinot. Common connected components of interval graphs. In S. Cenk Sahinalp and S. Muthukrishnan, editors, Combinatorial Pattern Matching - CPM04, number 3109 in Lecture Notes in Computer Science, pages 347-358. 15th Annual Combinatorial Pattern Matching Symposium, 2004.
9. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fullydynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In 30th annu ACM Sympos. Theory Comput., pages 79-89, 1998.
10. R.J. Lipton and R.E. Tarjan. Applications of a planar separator theorem. SIAM Journal on Computing, 9(3):615-627, 1980.
11. R. M. McConnell and J. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 536-545, 1994.
12. K. Mehlhorn. Data Structures and Efficient Algorithms. Springer Verlag, EATCS Monographs, 1984.
13. Takeaki Uno and Mutsunori Yaguira. Fast algorithms to enumerate all common intervals of two permutations. Algorithmica, 26, Number 2:290-309, 2000.
