Divide and Conquer Revisited. Application to Graph Algorithms

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

Divide and Conquer Revisited Application to Graph Algorithms

Binh Minh Bui Xuan, Michel Habib, and Christophe Paul

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 $Size(S)$ – for instance, the size $Size(G)$ 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 $S \in Triv^{\mathcal{H}}(\mathcal{S})$, 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 $C^{\mathcal{H}}_{total}(S)$

denotes the total cost for solving $\mathcal{P}$ over S by $\mathcal{H}$, $Rec^{\mathcal{H}}(S)$ time for recursive calls over $\mathbb{S}^{\mathcal{H}}(S)$, and $DC^{\mathcal{H}}(S)$ 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]).

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

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 sum-term, 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-DC* when the divide and combine time for any $\mathbb{S}^{\mathcal{H}}(S) = (S_i)_{i\in[\![1,k]\!]}$ is proportional to $Size(S) - Size(S_{i_0})$, where i_0 is such that $Size(S_{i_0})$ is the largest among $(Size(S_i))_{i\in[\![1,k]\!]}$. In this case, $\Theta(Size(S)\log Size(S))$ 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 $G_1 = (V, E_1)$, $G_2 = (V, E_2)$ on the same vertex set, *CCP* aims to compute a partition of V into subsets $(V_1, \dots V_k)$ such that any V_i is a maximal subset of vertices such that both induced subgraphs $G_1[V_i]$ and $G_2[V_i]$ are connected. *CCP* 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 $\mathcal{O}((n+m)\log^2 n)$ [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 CCP 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 "super-size" the sub-instances, i.e. when axiom $\mathcal{A}_{div}$ below holds.

$$\mathcal{A}_{div}(\mathcal{H}) :\ \forall S \in \mathcal{S}\backslash Triv^{\mathcal{H}}(\mathcal{S}),\ \begin{cases} \mathbb{S}^{\mathcal{H}}(S) = (S_1, S_2, \ldots, S_k) \\ \forall i \in [\![1,k]\!], \quad S_i \in \mathcal{S}\ \wedge\ Size(S_i) \neq 0 \\ Size(S) = Size(S_1) + \ldots + Size(S_k) \end{cases} \tag{2}$$

In this case, a naive bound in $\mathcal{O}(DC^{\mathcal{H}}(S)^2)$ 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 $Rec^{\mathcal{H}}$, which most of the time leads to linear – on the size of $DC^{\mathcal{H}}(S)$ – worst case analysis, leaving no real need for coarser minimising the divide and combine time. Unfortunately, minimising $Rec^{\mathcal{H}}$ 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 $\mathcal{A}_{div}(\mathcal{H})$ with respect to $Size$. We define $\mathcal{H}$ to be *completely-recursive and avoid-the-largest-DC* 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 $\mathbb{S}^{\mathcal{H}}(S) = (S_1, S_2, \ldots, S_k)$, then

$$DC^{\mathcal{H}}(S) \leq \alpha^{\mathcal{H}} \times \left(\sum_{i \in [\![1,k]\!], i \neq i_0} Size(S_i) \right),$$

$$Rec^{\mathcal{H}}(S) = \sum_{i \in [\![1,k]\!]}^{k} C^{\mathcal{H}}_{total}(S_i).$$

Theorem 1 (Avoid-the-Largest-DC Computing Time). *Any* divide-and-conquer *algorithm, satisfying axiom $\mathcal{A}_{div}$, with respect to Size that is completely-recursive and avoid-the-largest-DC runs in $\alpha^{\mathcal{H}} \times Size(S) \log Size(S)$ 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 DaC algorithm $\mathcal{H}$, we prove by induction on $Size(S)$ the following.

$$\forall S \in \mathcal{S},\ C^{\mathcal{H}}_{total}(S) \leq \alpha^{\mathcal{H}} \times Size(S) \log Size(S).$$

Indeed, if S is not trivial, let $\mathbb{S}^{\mathcal{H}}(S) = (S_i)_{i \in [\![1,k]\!]}$, k be such that $Size(S_k)$ is the largest among $(Size(S_i))_{i \in [\![1,k]\!]}$, and $n_i = Size(S_i)$. Then,

$$C^{\mathcal{H}}_{total}(S) = \alpha^{\mathcal{H}} \times \sum_{i=1}^{k-1} n_i\ +\ \sum_{i=1}^{k} C^{\mathcal{H}}_{total}(S_i).$$

Besides, for n_k is largest among $(n_i)_{i\in[\![1,k]\!]}$,

$$\forall 1 \le i \le k-1, \qquad n_i \le \frac{Size(S)}{2}$$
$$\text{Hence, } \forall 1 \le i \le k-1,\ \log n_i \le \log Size(S) - 1$$

Combining these to the inductive hypothesis results in:

$$\begin{aligned}
C^{\mathcal{H}}_{total}(S) &\le \alpha^{\mathcal{H}} \times \left(\sum_{i=1}^{k-1} n_i + \sum_{i=1}^{k} n_i \log n_i\right)\\
&\le \alpha^{\mathcal{H}} \times \left(\sum_{i=1}^{k-1} n_i + \sum_{i=1}^{k-1} n_i \log n_i + n_k \log n_k\right)\\
&\le \alpha^{\mathcal{H}} \times \left(\sum_{i=1}^{k-1} n_i + \sum_{i=1}^{k-1} n_i (\log Size(S) - 1) + n_k \log Size(S)\right)\\
&\le \alpha^{\mathcal{H}} \times Size(S) \log Size(S).
\end{aligned}$$

Finally, let be constructed a sequence $(S_n)_{n\in\mathbb{N}}$ of instances for $\mathcal{P}$ as follows. S_0 is a trivial case for $\mathcal{P}$. For all $i \ge 1$, 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 $(S_n)_{n\in\mathbb{N}}$ is such that $\mathcal{H}$ spends at least $\alpha^{\mathcal{H}} \times Size(S_i) \log Size(S_i)$ 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 $Size(S)$.

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' is the number of edges of the second largest connected component, then only m' 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-largest-DC" 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, Rep) will denotes the competitive graph search.

4 Bringing Competitiveness to the *CCP*

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 CCP 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 $\mathcal{G} = (G_1, G_2)$ 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 $V \setminus X$ – for instance, if X is a connected component in G_i – then:*

$$CCP(\mathcal{G}) = CCP(\mathcal{G}[X]) \cup CCP(\mathcal{G}[V \setminus X]),$$

$G_i[Y]$ is the subgraph of G_i induced by Y, and $\mathcal{G}[Y]$ denotes $(G_1[Y], G_2[Y])$.

Let be given $\mathcal{G} = (G_1, G_2)$ as a non-trivial instance of the CCP 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 $\mathcal{G}[V_1], \ldots, \mathcal{G}[V_k]$. 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

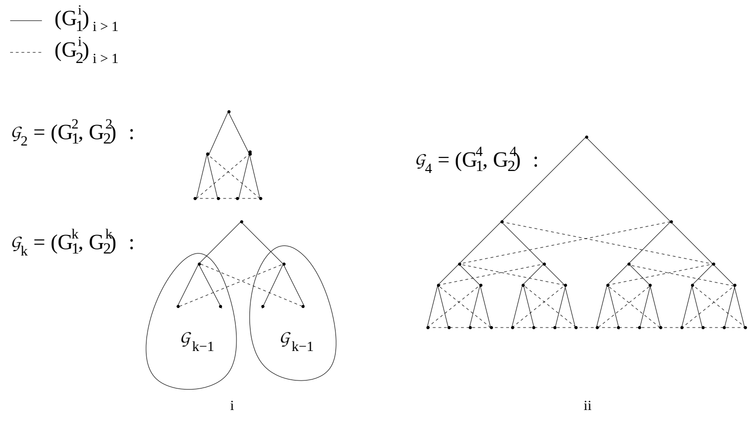
$(G_1^i)_{i>1}$
$(G_2^i)_{i>1}$
$\mathcal{G}_2 = (G_1^2, G_2^2)$:
$\mathcal{G}_k = (G_1^k, G_2^k)$:
$\mathcal{G}_{k-1}$
$\mathcal{G}_{k-1}$
$\mathcal{G}_4 = (G_1^4, G_2^4)$:
i
ii

Fig. 1. A sequence $(\mathcal{G}_i)_{i>2} = ((G_1^i, G_2^i))_{i>2}$ of tricky instances of CCP, yielding the maximal complexity to the *divide-and conquer* scheme for solving CCP. i. Recursive definition for $(\mathcal{G}_i)_{i>2}$. ii. Example of $\mathcal{G}_4$.

as follows: 1) using a competitive search, select the $k-1$ smallest components, say $V_1, \dots V_{k-1}$; 2) using a graph search extract $\mathcal{G}[V_1], \dots, \mathcal{G}[V_{k-1}]$; 3) $\mathcal{G}[V_k]$ can be obtained by removing from $\mathcal{G}$ the vertices of $V \setminus V_k$ together with their incident edges. These three steps can all be done in time linear in the number of edges incident to $\bigcup_{1 \le i \le k-1} V_i$. To prepare the next recursion step, a representative of each connected component of the induced subgraphs $G_1[V_i]$ $(1 \le i \le k)$ has to be collected. For the subgraphs $G_1[V_i]$, for $i < k$, a graph search can be afforded since they are of size small enough. The main trick concerns $G_1[V_k]$.

Let $\overline{V}_k$ denote $V \setminus V_k = \bigcup_{1 \le i \le k-1} V_i$. Since G_1 is connected, the subset of vertices of V_k neighbouring some vertex in $\overline{V}_k$ contains at least one vertex per connected component of $G_1[V_k]$. This set can be computed in time proportional to the number of edges incident to $\overline{V}_k$ 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 data-structures 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 DS representing G.
- `UPDATE`$(DS,$ `DELETION`$, E)$: input a data structure DS and a list E of edges in DS, delete in DS all edges in E, and output the remaining of DS.

– `GET_COMPONENT`(DS, v): input a data structure DS and a vertex v in DS, 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_{extra}. Depending on the inquired data-structure C_{extra} may vary. For instance, the best known $\mathcal{O}((n+m)\log^2 n)$ worst case complexity for solving *CCP* over arbitrary graphs [6] has been reached using *ET-trees* dynamic data structure [9]. Actually, C_{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]. *Edge-ordered dynamic tree* [5] devoted to represent plane graphs also allows to solve *CCP* over planar graphs with a low C_{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 $(V_1, V_2, \ldots, V_p)$ is a collection of disjunctive subsets of $V(G)$, we define the set of “bridges“ in G between $(V_i)_{i\in[\![1,p]\!]}$ as

$$InterEdges(G, (V_i)_{i\in[\![1,p]\!]}) = \{(u, v) \in E(G) \mid \exists i \neq j,\ u \in V_i \wedge v \in V_j\}.$$

– `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 $InterEdges(G, (X, V \setminus X))$.
– `GET_OUTGOING_VERTICES`(G, X): thanks to a classical graph search over $G[X]$, compute all extremities in $InterEdges(G, (X, V \setminus X))$ 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

$$|X| + |E(G[X])| + |InterEdges(G, (X, V \setminus X))|.$$

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 : $(\mathcal{G}, i, Representatives, \mathcal{DS})$: quadruplet, where
       $i$ is a number belonging to $\{1,2\}$ − we define $\overline{i} = 3 - i$;
       $\mathcal{G} = (G_1, G_2)$ a pair of graphs over the same vertices set s.t.
                 $G_{\overline{i}}$ is connected and $E(G_1) \cap E(G_2) = \emptyset$;
       $Representatives$ a list containing exactly one vertex per
                 connected component in $G_i$;
       and $\mathcal{DS} = (DS_1, DS_2)$ 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
 1. If $|Representatives| = 1$ then $Result \leftarrow [V]$;
 2. Else
 3.     $Result \leftarrow []$;
 4.     $(V_1, V_2, \ldots, V_{k-1}) \leftarrow$ GET_SMALL_SUBGRAPHS$(G_i, Representatives)$;
        // Proceeding small subgraphs in time prop. to their size
 5.     Forall $j \in [\![1, k-1]\!]$ do:
 6.         Compute $\mathcal{G}[V_j] = (G_1[V_j], G_2[V_j])$ from the knowledge of $(\mathcal{G}, V_j)$;
 7.         $Representatives \leftarrow$ GET_REPRESENTATIVES$(G_{\overline{i}}[V_j])$;
 8.         $OutgoingEdges \leftarrow$ GET_OUTGOING_EDGES$(G_{\overline{i}}, V_j)$;
 9.         UPDATE$(DS_{\overline{i}},$ DELETION$, OutgoingEdges)$;
10.         Compute $\mathcal{DS}[V_j]$ from the knowledge of $(\mathcal{DS}, V_j)$;
11.         $Result \leftarrow Result$ @ MAIN$(\mathcal{G}[V_j], \overline{i}, Representatives, \mathcal{DS}[V_j])$;
12.     EndFor
        // Coping with the biggest subgraph in time proportional to
        //the total size of the small subgraphs
13      $\overline{V_k} \leftarrow V_1 \cup V_2 \cup \ldots V_{k-1}$;
14.     Compute $\mathcal{G}[V_k]$ from the knowledge of $(\mathcal{G}, \overline{V_k})$;
15.     $OutgoingVertices \leftarrow$ GET_OUTGOING_VERTICES$(G_{\overline{i}}, \overline{V_k})$;
16.     Compute $\mathcal{DS}[V_k] = (DS_1[V_k], DS_2[V_k])$ knowing $(\mathcal{DS}, \overline{V_k})$;
17.     $Representatives \leftarrow$ FILTER$(OutgoingVertices, DS_{\overline{i}}[V_k])$;
18.     $Result \leftarrow Result$ @ MAIN$(\mathcal{G}[V_k], \overline{i}, Representatives, \mathcal{DS}[V_k])$;
19. EndElse
20. Return $Result$.
```

Procedure MAIN.

Complexity Issues To analyse the computing time $C_{total}(\mathcal{G})$ of our algorithm, we need more knowledge on the inquired dynamic data structures. Thus, we define $C_{INIT}(G)$ as the cost of `INIT`(G), $C_{COMP}(DS)$ of `GET_COMPONENT`(DS, v), and $C_{DEL}(DS)$ of `UPDATE`$(DS,$ `DELETION`$, E)$, when E is reduced to a singleton.

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

$$C_{total}(\mathcal{G}) = C_{extra}(\mathcal{G}) \ + \ C_{core}(\mathcal{G}),$$
$$C_{extra}(\mathcal{G}) = C_{Init}(\mathcal{G}) \ + \ C_{Filter}(\mathcal{G}) \ + \ C_{Maintenance}(\mathcal{G}).$$

Lemma 3. *Let be given an instance $\mathcal{G} = (G_1, G_2)$, along with $m = |E(G_1)| + |E(G_2)|$, and $DS_i =$ `INIT`(G_i) for $i \in \{1, 2\}$. Then,*

$$\begin{aligned} C_{Init}(\mathcal{G}) &= C_{INIT}(G_1) + C_{INIT}(G_2) \\ C_{Filter}(\mathcal{G}) &= \mathcal{O}(m \times (C_{COMP}(DS_1) + \log m)) \\ &= \mathcal{O}(m \times (C_{COMP}(DS_2) + \log m)) \\ C_{Maintenance}(\mathcal{G}) &= \mathcal{O}(m \times C_{DEL}(DS_1)) = \mathcal{O}(m \times C_{DEL}(DS_2)) \end{aligned}$$

We now evaluate $C_{core}(\mathcal{G})$. Though routine `MAIN` is an avoid-the-largest-DC *DaC* algorithm, we introduce a function *Size* as follows: if $\mathcal{G} = (G_1, G_2)$ is the instance, then $Size(I) = |E(G_1)| + |E(G_2)|$, where I is the corresponding input given to `MAIN` routine.

Lemma 4. *Let be given $\mathcal{G} = (G_1, G_2)$ as an instance. Then,*

$$C_{core}(\mathcal{G}) = \mathcal{O}(m \log m) \ \textit{where} \ m = |E(G_1)| + |E(G_2)|.$$

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

Indeed let us first prove that `MAIN` is a *DaC* algorithm satisfying axiom $\mathcal{A}_{div}$ with respect to *Size*. Unless its input $I = (\mathcal{G}, i, Representatives, \mathcal{DS})$ is trivial (line 1), the procedure divides V, set of vertices in $\mathcal{G} = (G_1, G_2)$, into a partition $V_1, \ldots, V_{k-1}, V_k$, where $V_k = V \backslash (V_1 \cup \ldots \cup V_{k-1})$ (line 4). This operation implicitly decomposes the input I into $I_1, \ldots, I_k, I_{k+1}$ defined as follows.

$$\begin{aligned} \forall j \in [\![1, k]\!], \qquad I_j &= (\mathcal{G}[V_j], i, Representatives, \mathcal{DS}[V_j]) \\ I_{k+1} &= ((QuotientG_1, QuotientG_2), \_, \_, \_), \end{aligned}$$

where $QuotientG_i = (V, InterEdges(G_i, (V_j)_{j \in [\![1,k]\!]}))$ for $i \in \{1, 2\}$, and the symbol _ stands for dummies. Thus, by constructions of *Size* and $(I_i)_{i \in [\![1,k+1]\!]}$, the non redundant divisions axiom $\mathcal{A}_{div}$ holds for `MAIN`. Furthermore, `MAIN` makes recursive calls (lines 11 and 18) to solve $(I_j)_{j \in [\![1,k]\!]}$. 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 *DaC* 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 $(I_j)_{j\in[\![1,k+1]\!]}$ is I_k. First, in `MAIN`, lines 2-7, 10-14, 16 and 18-19 can be computed in time proportional to $Size(I)-Size(I_k)-Size(I_{k+1}) = Size(I_1)+Size(I_2)+\ldots+Size(I_{k-1})$. Now, ignoring lines 9 and 17, there still are two lines left: 8 and 15. Fortunately, their computing time is proportional to

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

Since obviously I_k is the largest instance, we can afford the former computing time and `MAIN` is an avoid-the-largest-DC *DaC* algorithm.

The lemma then follows from Theorem 1. □

	best so far	our algorithm	conjecture
forests of disjunctive paths	$\mathcal{O}(n + \text{“}K\text{“})$ [13]	$\mathcal{O}(n\log n)$	$\mathcal{O}(n)$?
arbitrary graphs	$\mathcal{O}(n\log n + m\log^2 n)$ [6]	$\mathcal{O}(n\log n + m\log^2 n)$	?
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]	$\mathcal{O}((n+m)\log n) = \mathcal{O}(\Delta n\log n)$	$\mathcal{O}(n+m) = \mathcal{O}(\Delta n)$?
planar graphs	$\mathcal{O}(n\log^2 n)$ [6]	$\mathcal{O}(n\log n)$	?
forests of trees	$\mathcal{O}(n\log^2 n)$ [6]	$\mathcal{O}(n\log n)$	?
permutation graphs	$\mathcal{O}(n\log n + m\log^2 n)$ [6]	$\mathcal{O}(n\log n + m\log^2 n)$	$\mathcal{O}((n+m)\log n)$?

Fig. 2. Time complexity solving problem *CCP* over two graphs with disjoint edge sets.

Theorem 2. *Let be given $\mathcal{G} = (G_1, G_2)$ as an instance of CCP, along with $m = |E(\mathcal{G})| = |E(G_1)| + |E(G_2)|$ and $n = |V(G_1)| = |V(G_2)|$. Our CCP algorithm runs in $\mathcal{O}((n+m)\log n)$ worst case time if $\mathcal{G}$ is a pair of planar graphs, in $\mathcal{O}((n+m)\log^2 n)$ otherwise.*

Proof. Both *ET-trees* [9] and *edge-ordered dynamic tree* [5] data structures yield $C_{INIT}(G) = \mathcal{O}((n_G+m_G)\log n_G)$ and $C_{COMP}(DS) = \mathcal{O}(\log n_{DS})$, where $n_G = |V(G)|$, $m_G = |E(G)|$, and n_{DS} is the size of DS, which can be the number of vertices or the number of edges for both data structures are trees. Besides, *ET-trees* maintain $C_{DEL}(DS) = \mathcal{O}(\log^2 n)$ amortised cost per deletion, and *edge-ordered dynamic tree* maintain an amortised $C_{DEL}(DS) = \mathcal{O}(\log n)$.

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 $G[V \setminus X]$ has at least one vertex that belongs to the list*

$$OutgoingVertices = \texttt{GET\_OUTGOING\_VERTICES}(G, X).$$

According to Lemma 5, whenever `FILTER` is inquired at line 17 in `MAIN`, its input $(OutgoingVertices, DS_{\overline{i}}[V_k])$ is such that $DS_{\overline{i}}[V_k]$ encodes $G_{\overline{i}}[V_k]$ and each connected component in $G_{\overline{i}}[V_k]$ 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 DS that represents some graph, along with a list X containing at least one representative vertex per connected component in the graph. Then,* `FILTER`(X, DS) *selects from X exactly one representative vertex per connected component in the graph that DS 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 $(G^i)_{i \in [\![1,k]\!]}$ has exactly one vertex some list Rep, then* `GET_SMALL_SUBGRAPHS`(G, Rep) *outputs $(V(G^1), V(G^2), \ldots, V(G^{k-1}))$.*

We now is ready for proving `MAIN`.

Theorem 3.
1. *Every possible input $(\mathcal{G}, i, Representatives, \mathcal{DS})$ to* `MAIN` *holds the following.*

   i. *$\mathcal{G} = (G_1, G_2)$ is a pair of planar graphs, where $V(G_1) = V(G_2) = V$ and $E(G_1) \cap E(G_2) = \emptyset$.*
   ii. *$i \in \{1, 2\}$ is such that $G_{\overline{i}}$ 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. *$\mathcal{DS} = (DS_1, DS_2)$ is a pair of data structures that respectively represent G_1 and G_2.*

2. *If the above properties hold,* `MAIN`$(\mathcal{G}, i, Representatives, \mathcal{DS})$ *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 $|Representatives| = 1$ 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, $|Representatives| \geq 2$ and line 1 is skipped. Moreover, Lemma 7 states that, after line 4, $(V_1, V_2, \ldots, V_{k-1})$ 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.

$$CCP(\mathcal{G}) = CCP(\mathcal{G}[V_1]) \cup CCP(\mathcal{G}[V_2]) \cup \ldots \cup CCP(\mathcal{G}[V_k]).$$

Since $|V_i| \leq p$ for all $1 \leq i \leq k$, we deduce from the inductive hypothesis that `MAIN` correctly computes all the sets $CCP(\mathcal{G}[V_i])$ at lines 11 and 18. Therefore, lines 3-19 correctly compute $CCP(\mathcal{G})$, which is union of all the sets $CCP(\mathcal{G}[V_i])$. 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. $\square$

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 CCP 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 187–198. 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 fully-dynamic 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.