An $O(n \cdot m)$ algorithm for calculating the closure of *lca*-type operators

Vincent Ranwez[1], Stefan Janaqi[2], Sylvie Ranwez[2]

[1] Institut des Sciences de l'Evolution de Montpellier (ISE-M), UMR 5554 CNRS,Université Montpellier II, place E. Bataillon, CC 064, 34 095 Montpellier cedex 05, France.
vincent.ranwez@univ-montp2.fr

[2] LGI2P/EMA Research Centre, Site EERIE, Parc scientifique G. Besse, 30 035 Nîmes cedex 1, France.
{sylvie.ranwez, stefan.janaqi}@mines-ales.fr

Abstract

The least common ancestor on two vertices, denoted $lca(x,y)$, is a well defined operation in a directed acyclic graph (*dag*) G. We introduce $U_{lca}(S)$, a natural extension of $lca(x,y)$ for any set S of vertices. Given such a set S_0, one can iterate $S_{k+1} = U_{lca}(S_k)$ in order to obtain an increasing set sequence. G being finite, this sequence has always a limit which defines a closure operator. Two equivalent definitions of this operator are given and their relationships with abstract convexity are shown. The good properties of this operator permit to conceive an $O(n \cdot m)$ time complexity algorithm to calculate its closure. This performance is crucial in applications where *dags* of thousands of vertices are employed. Two examples are given in the domain of life-science: the first one concerns genes annotations' understanding by restricting Gene Ontology, the second one deals with identifying taxonomic group of environmental DNA sequences.

1 Introduction

In this paper we address the problem of efficiently computing the closure of *lca*-type operators in directed acyclic graphs (*dag*) $G = (V, E)$. Such graphs appear in numerous applications such as: ontologies (semantic representation), phylogeny networks (speciation histories) or inheritance graphs (object programming languages). A *least common ancestor* of

ARS COMBINATORIA 104(2012), pp. 107-128

two vertices x and y, denoted $lca(x,y)$, is an ancestor of both vertices, that has no proper descendant that is also an ancestor of x and y. Now, let S be a set of vertices of interest of G. On one side, the set V of all vertices contains all $lca(x,y)$ for any couple $x,y \in S$ but, this does not help to focus on relevant parts of G containing S. On the other side, filtering G to keep only vertices of S gives few insight about the relationships among vertices of S. In order to preserve those relationships, one can consider the *least* overset $\overline{S}$ of S, that contains all $lca(x,y)$ for any $x,y \in \overline{S}$. For people used to convexity concepts, this sounds as: "... for any two points in $\overline{S}$, the segment relying them lies in $\overline{S}$". Actually, we show that it does not just "sounds as" and we define a set $\overline{S}$ that really satisfies this property. First we define the operator:

$$U_{lca}(S) = \cup_{x,y \in S}\, lca(x,y).$$

Then, we show that the closure of this operator, denoted by $\overline{S}$ verifies the four axioms of convex hull (see [1]):

(U.1) $\overline{\emptyset} = \emptyset, S \subseteq \overline{S}$;

(U.2) (monotonicity) $S_1 \subseteq S_2 \Rightarrow \overline{S_1} \subseteq \overline{S_2}$;

(U.3) (idempotence) $\overline{(\overline{S})} = \overline{S}$;

(U.4) (finiteness) if $x \in \overline{S}$, then there is a finite set $F \subseteq S$ such that $x \in U_{lca}(F)$.

The sets S that are equal to their closure ($S = \overline{S}$) are called convex and form a convex space verifying the following properties (this is a classical result of convexity theory):

(C.1) $\emptyset, V$ are convex;

(C.2) if A, B are convex, then $A \cap B$ is convex;

(C.3) if A_i, are convex and $A_i \subseteq A_{i+1}$ for $i = 1, 2, ...$ then $\cup_{i \geq 1} A_i$ is convex;

Thus, $U_{lca}(S)$ and its closure have nice mathematical structures that are exploited in our greedy algorithm to reach a low time complexity of order $O(n \cdot m)$. Starting with a set S_0 of vertices and the topological order of V, the algorithm decides once for all if a new vertex v is in the U_{lca}-*closure* of S_0 or not. It is easy to define and calculate similarly the U_{gcd}-*closure* (*greatest common descendant*). One has just to reverse all

the edges of G and apply the U_{lca} results and algorithm.

Of course, the operator $lca(x,y)$ is not new and there are several good algorithms to calculate it. To the best of our knowledge, all the efficient approaches to calculate $lca(x,y)$ pass through the computation of shortest paths. The main result of [2] is that, for any couple of vertices x,y, a *representative* lca can be computed with time complexity $O(|V|^{\omega})$ with $\omega = 2.688$ (see [2, 3]). We stressed the word 'representative' because a couple of vertices can have more than one lca and, as shown further, one of the main characteristics of our algorithm is that it identifies *all* of them. Yet, the construction of [2] is interesting on its own, since it transforms the lca searching to a shortest path problem. An improved version of the "*one representative lca*" algorithm is given with $\omega = 2.575$ (see [4]). This optimization relies on a novel reduction of all-pairs lca problem to the problem of finding maximum witnesses for Boolean matrix product. As the domain is active, actually [5] have given an algorithm that calculates *all* $lca(x,y)$ for *all* pairs of vertices with a mean time complexity $O(|V|^3 loglog(|V|))$ and worst time complexity $O(|V|^{3.3399})$.

Remind that we need the U_{lca}*-closure* of a set S (denoted by $\bar{S}$) and a straightforward way to use the above subroutines would have a complexity of at least $O(|V|^{3.3399})$ for computation time and $O(|V|^2)$ for memory space (in order to store pre-computed lca). Our algorithm does not need to calculate $lca(x,y)$ for any couple of vertices x,y. It constructs $\bar{S}$ by using the closure and convexity axioms above, in particular (U.2), (U.3) and (C.2), (C.3). This solution has lower time – $O(|\bar{S}||E|)$ – and space – $O(|\bar{S}||V|)$ – complexity. The advantage of this solution is even more relevant in practice, since for most real cases $|\bar{S}| \ll |V|$ and for most *dag* $|E| \ll |V^2|$.

The paper is organized as follows: section 2 gives the definitions and properties of U_{lca}*-closure*; section 3 gives an efficient algorithm to compute this closure and its proof of correctness; applications of our results are provided in section 4; concluding remarks are given in section 5.

2 Least common ancestor operator and its closure

2.1 Preliminary definitions

In this section, we give the definitions of two generalized common operators: *least common ancestor* (*lca*) and *greatest common descendant* (*gcd*) for a *direct acyclic graph* (*dag*). The following definitions are provided to make the paper self-contained. For further definitions on graphs see [6]. Given a *dag* $G = (V, E)$ and an edge (x, y) we say that x is the *child* and y is the *parent*. The *indegree* $d_G^-(v)$ (*outdegree* $d_G^+(v)$) of a vertex v is the number of edges with head v (tail v). When G contains a directed (v, u)-path, the vertex u is said to be an *ancestor* of v and the vertex v is a *descendant* of u. For a non-empty subset W of V, the subgraph of G whose vertex set is W and whose edge set is the set of edges of G that have both ends in W is called the subgraph of G *induced* by W and is denoted $G[W]$.

Given a vertex v of the *dag* $G = (V, E)$, the set $A_G(v)$ denotes the subset of *ancestors* of v in G. The generalization of this definition to a set $S \subseteq V$ of vertices is straightforward, i.e. $A_G(S) = \bigcap_{v \in S} A_G(v)$. For simplicity, we will omit index G from the notations whenever there is no ambiguity.

Definition 2.1.1. [2] The *least common ancestors* $lca(S)$ of a vertex subset $S \subseteq V$ with respect to a *dag* $G = (V, E)$ are the vertices $u \in A(S)$, such that $d_H^-(u) = 0$ in the graph $H = G[A(S)]$ induced by $A(S)$.

This definition generalizes the widely known concept of least common ancestor (see [2, 3]) for a couple of vertices, i.e. $lca(\{x, y\}) = lca(x, y)$. It follows immediately from Definition 2.1.1 that $x = lca(x, y)$ if there is a directed (y, x)-path. By extension, we define $lca(x, x) = x$ for all x.

Note that, unless the *dag* G is a tree, $lca(x, y)$ may contain several vertices and the existence of a pair of vertices $x, y \in S$ such that $lca(x, y) = lca(S)$ is not guaranteed. For instance, in the example presented in Fig. 1, $\text{lca}(\{C1, C2, C3\}) = \{A1, A2\}$ while $\text{lca}(C1, C2) = \{B1, A2\}$, $\text{lca}(C1, C3) = \{B2\}$ and $\text{lca}(C2, C3) = \{A1, B3\}$. The notion of *least common semi-strict ancestors* has been introduced in order to characterize a unique ancestor of a set of vertices for both tree and dag [7]. An alternative definition of the *lca* in terms of partially ordered sets, has been proposed by [8].

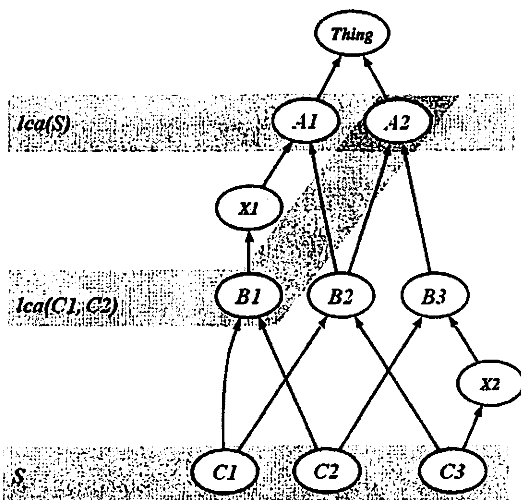

Fig. 1. Illustration[1] of the *lca* operator

2.2 Two equivalent definitions of U_{lca}-*closure* in a *dag*

Closure operators are widely used in mathematics, especially in geometry. The best known example comes from convexity in a Euclidian space. A lot of properties follow from the fact that a convex set, for example a convex polygon, can be obtained by giving a *finite* set of points and the segment operator $s(x,y)$. A natural question is whether these geometric properties are preserved when extending convexity definition to graphs. It turns out that with a little set of axioms, a lot of good properties of convexity ([1, 9, 10]) can be transposed in discrete structures like graphs. The richest transposition is obtained for graphs endowed with interval convexities where the notion of segment $s(x,y)$ is replaced by that of interval $I(x,y)$ that is the bunch of the shortest paths between x and y in the given graph (see [10]). It is out of the scope of this paper to explore these properties. Nevertheless, the definition of closure and its properties show clearly that these objects are convex.

In the previous section, we provided an intuitive and natural definition of the least common ancestors of a set of vertices denoted as $lca(S)$. Yet, in applications, when considering a set of vertices S, every least common ancestor of a pair of vertices of S is a key vertex to gain insight

[1] Figures were created using GraphViz, version 2.26 http://www.graphviz.org/

into relationships among vertices of S. We thus now introduce a new operator that makes use of the lca operator and generalizes it to a well defined closure operator.

Definition 2.2.1. Let S be a subset of vertices of G. The U_{lca}*-operator* on S is defined as:

$$U_{lca}(S) = \cup_{x,y \in S} lca(x, y) \tag{1}$$

It follows from this definition that $S \subseteq U_{lca}(S)$ and that the U_{lca}*-operator* is monotonous, i.e.:

$$A \subseteq B \Rightarrow U_{lca}(A) \subseteq U_{lca}(B).$$

The Fig. 2 below illustrates the definition of the U_{lca}*-operator* and emphasizes its difference with the standard *lca* operator. In this example, given the set $S = \{C1, C2, C3\}$, $U_{lca}(S) = \{B1, B2, B3, C1, C2, C3\}$, while $lca(S) = \{A1, A2\}$.

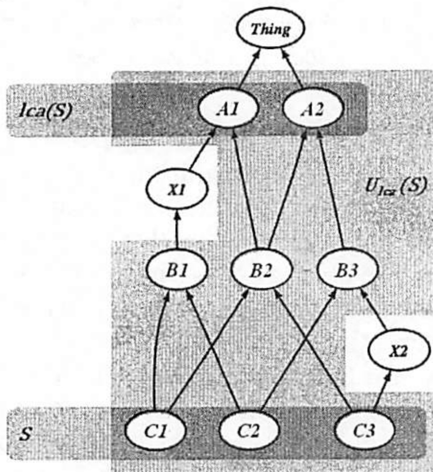

Fig. 2. U_{lca} and lca are two different operators.

In some cases, having only $U_{lca}(S)$ and/or $lca(S)$ is not enough to understand all relationships among vertices of S. Such a case is depicted in Fig. 3, where the vertex A is helpful for understanding $C1$ and $C2$ relationships but is neither included in $U_{lca}(S)$ nor in $lca(S)$. Vertex A is of interest since A is the *lca* of $B1$ and $B2$, which in turn are *lca* of two vertices of S. This leads us to the following definition of the U_{lca}*-closure* of S.

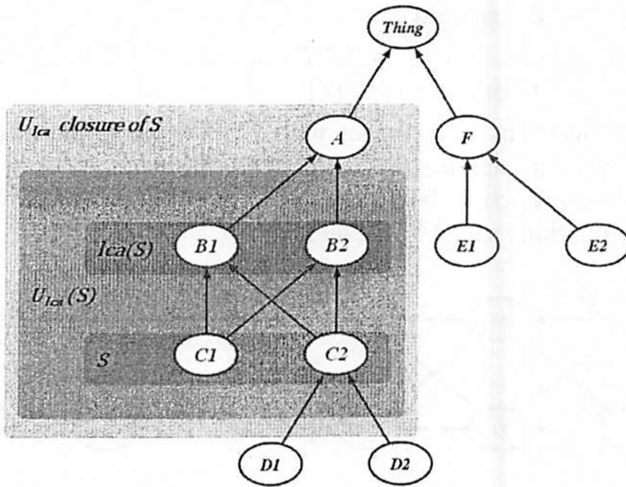

Fig. 3. Illustration of U_{lca}-*closure*

Let S be a subset of vertices of G and let consider the increasing set sequence defined by:

$$S_0 = S \text{ and } S_{i+1} = U_{lca}(S_i), i = 0, 1, ...$$

Since G is finite and $\forall i\ S_i \subseteq S_{i+1}$, there is a number $c, 0 \leq c \leq |V|$ such that $\forall k \geq c, S_k = U_{lca}(S_k)$. This fixpoint (or fixset) is reached because of the monotonicity of U_{lca}-*operator*. In fact, once this relation holds for a given k it holds for all greater values. So, c and S_c are well defined.

Definition 2.2.2. The number c is the *closure index* and the set S_c is called the U_{lca}-*closure* of S and is denoted by $\bar{S}$.

It is clear from the Definition 2.2.1 that $U_{lca}(\emptyset) = \emptyset$ and $S \subseteq U_{lca}(S)$. Also, U_{lca} is monotonous and it verifies the finiteness property because G is finite. Since U.1, U.2 and U.4 hold for the U_{lca}-*operator*, they also hold for its closure. Moreover, by definition of the closure, $\overline{S} = S_c = U_{lca}(S_c) = U_{lca}(\overline{S})$, therefore the idempotence axiom U.3 is also satisfied.

This definition provides a simple (and inefficient!) iterative algorithm to compute U_{lca}-*closure*. The time complexity of this algorithm is related to the closure index. In the simple case when G is a tree, the closure index cannot be greater than 1. The following lemma shows that this is

no longer true in the general case.

Lemma 2.2.1. For a *dag* $G = (V, E)$ and a set $S \subseteq V$, the number of iterations needed to obtain $\bar{S}$ is $O(|V|)$.

Proof. It is clear that S_k increases with at least one vertex at each iteration, hence proving that $c \leq (|V| - |S|)$. On the other hand, as shown by the example below, c can be as large as $(|V| - |S|)/2$. It follows that the number of iterations needed to obtain $\bar{S}$ is $O(|V|)$. □

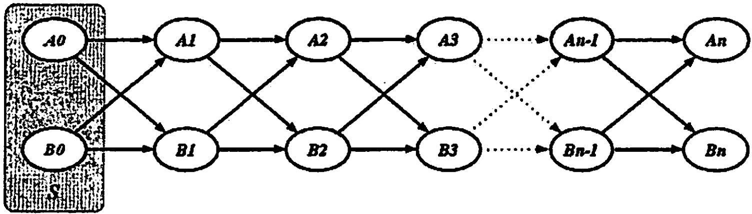

Fig. 4. The closure index can be proportional to $|V|$. $S_0 = \{A_0, B_0\}$. At iteration k two vertices $\{A_k, B_k\}$ are added so that $S_k = \{A_0, \ldots, A_k, B_0, \ldots B_k\}$.

There is an alternative (descending) way to define U_{lca}*-closure*. For this, let the family of U_{lca}*-closed* sets containing S be denoted by:

$$\mathcal{L}(S) = \{L \subseteq V \mid S \subseteq L \text{ and } L = U_{lca}(L)\} \quad (2)$$

Now, we show that the family $\mathcal{L}(S)$ verifies the axioms C.1, C.2 and C.3. The axiom C.1 is obvious from the definition. The axiom C.3 is conceived originally in order to treat the case of infinite increasing set sequences in continuous spaces. As we are dealing with finite graphs, this axiom is obviously verified. Nevertheless, it is the basis of our greedy algorithm because, when treating the increasing set sequence of U_{lca}-*closed* sets, only the last one has to be stored for further treatment. The following lemma gives the proof of axiom C.2.

Lemma 2.2.2. $\mathcal{L}(S)$ is non-empty and closed for the intersection, i.e. $M, N \in \mathcal{L}(S) \Rightarrow M \cap N \in \mathcal{L}(S)$.

Proof. $\mathcal{L}(S)$ is non-empty since V obviously belongs to $\mathcal{L}(S)$. Let us now prove that $M, N \in \mathcal{L}(S) \Rightarrow M \cap N \in \mathcal{L}(S)$.

- $S \subseteq M \cap N$. As $S \subseteq M$ and $S \subseteq N$, $S \subseteq M \cap N$.
- $M \cap N = U_{lca}(M \cap N)$.
    - o $M \cap N \subseteq U_{lca}(M \cap N)$, by the U_{lca} definition.

- $U_{lca}(M \cap N) \subseteq M \cap N$. This comes from the fact that the U_{lca} preserves monotonicity:
  $M \cap N \subseteq M \Rightarrow U_{lca}(M \cap N) \subseteq U_{lca}(M) = M$
  $M \cap N \subseteq N \Rightarrow U_{lca}(M \cap N) \subseteq U_{lca}(N) = N$
  $U_{lca}(M \cap N) \subseteq M, U_{lca}(M \cap N) \subseteq N \Rightarrow U_{lca}(M \cap N) \subseteq M \cap N$. □

Definition 2.2.3. The U_{lca}*-closure* of S is the set $\bar{\bar{S}} = \bigcap_{M \in \mathcal{L}(S)} M$.

Lemma 2.2.3. The two definitions of U_{lca}*-closure* are equivalent.

Proof. One need to show that $\bar{\bar{S}} = \bar{S}$.

- $\bar{\bar{S}} \subseteq \bar{S}$. By definition, $\bar{S} = U_{lca}(\bar{S})$, furthermore the monotonicity of the set sequence $S_0, S_1 \ldots, \bar{S}$ ensures that $S \subseteq \bar{S}$. Therefore $\bar{S} \in \mathcal{L}(S)$, thus proving that $\bar{\bar{S}} \subseteq \bar{S}$.
- $\bar{S} \subseteq \bar{\bar{S}}$. By definition, S is included in every set of $\mathcal{L}(S)$ and thus in their intersection. It follows that $S_0 \subseteq \bar{\bar{S}}$, and therefore $U_{lca}(S_0) \subseteq U_{lca}(\bar{\bar{S}})$, which can be rewritten $S_1 \subseteq \bar{\bar{S}}$. Applying the U_{lca} operator to both terms leads to $S_2 \subseteq \bar{\bar{S}}$, $S_3 \subseteq \bar{\bar{S}}$ and so on until $S_c \subseteq \bar{\bar{S}}$, thus proving that $\bar{S} \subseteq \bar{\bar{S}}$. □

These definitions and properties provide the framework for our algorithm.

3 An Efficient algorithm to compute U_{lca}*-closure*

In our applications, we have encountered *dags* that may contain several thousand of vertices. Thus, efficient algorithms are needed to compute the U_{lca}*-closure*. As mentioned in introduction, several good algorithms exist to retrieve all the vertices $v \in lca(x,y)$. When calculating the U_{lca}*-closure*, one could use one of these algorithms as a subroutine. This approach, detailed below, provides a straightforward solution to calculate the U_{lca}*-closure* and a (high) upper bound on time complexity. Then, we introduce an optimized solution that takes advantages of convexity properties and topological vertex order. The main result is an algorithm with a worst time complexity of order $O(|\bar{S}||E|)$.

3.1 Straightforward algorithm to compute U_{lca}*-closure*

In Algorithm 1 the *lca* of each couple of vertices of $\bar{S}$ is computed once leading to $O(|V|^2)$ calls of the $lca(x,y)$ subroutine. It follows that the time complexity of this algorithm is bounded by the preprocessing of the

dag that allows obtaining the *lca* of two vertices in constant time. As mentioned in the introduction, the best known solution to this pre-process problem has a $O(|V|^{3.3399})$ worst time complexity and requires $O(|V|^2)$ memory space.

```
Name: Straightforward_Ulca_closure
Input: a dag G and a set of vertices S of G
Result: S̄.
    S_k ← S; S_new ← S; S_tmp ← S_k;
    do
        for each (x, y) ∈ S_k × S_new
            S_tmp ← S_tmp ∪ lca(x, y);
        end
        S_new ← S_tmp − S_k;
        S_k ← S_tmp;
    while S_new ≠ ∅
    return S_k
```

Algorithm 1. A straightforward U_{lca}-*closure* algorithm

3.2 Optimized algorithm to compute U_{lca}-closure in $O(|\bar{S}||E|)$

This subsection details an optimized algorithm that determines $\bar{S}$ for a *dag* $G = (V, E)$ in $O(|\bar{S}||E|)$ time complexity. The key idea of this greedy algorithm is that, though there are $O(|V|^2)$ couples of vertices, at most $O(|V|)$ vertices can be added to $\bar{S}$. Rather than computing *lca* for each pair of vertices, our greedy algorithm considers each vertex and decides whether or not it must be added to $\bar{S}$. This can be done efficiently by taking a topological vertex order induced by the *dag*.

Our U_{lca}-*closure* algorithm considers vertices in *post order*, i.e. a vertex is never considered before considering all of its descendants. Indeed, vertices of a *dag* can be ordered along a horizontal line such that all descendants of a vertex are placed to its left. We call this a *post order* since, as shown in [11], one can be efficiently obtained using the post-order indices of a depth-first search. We give this classical ordering algorithm (Algorithm 2) to make the paper self-contained.

Name: postOrder **Input**: a *dag* G **Result**: the list of vertices of G in postOrder	**Name**: postOrderRec **Input**: a *dag* G, a vertex n of G **Result**: add the list of desc(n), in post order, to the postOrder list of G

G.postOrder ← empty list
for *root* **in** G.vertices()
    if *root* has no parent
        postOrderRec(*root*)
end
return G.postOrder
	mark n as visited
	for s **in** children(n)
	    if s has not been visited
	        postOrderRec(s)
	end
	append n to the postOrder list of G

Algorithm 2. Post order implementation.

Name: U_{lca}*-closure*
Input: a *dag* G, a set S of vertices
Result: $S^{AL} = \overline{S}$

$S^{AL} \leftarrow \emptyset$
P = postOrder(G)
for n **in** P
    $S^D(n) \leftarrow \emptyset$ // $S^D(n)$ is the set of descendants of n present in $\bar{S}$
    $maxS^D(n) \leftarrow 0$ // $maxS^D(n)$ is the maximal value $|S^D(s)|$ with s a child of n
    for s **in** children(n)
        $S^D(n) \leftarrow S^D(n) \cup S^D(s)$ (*)
        $maxS^D(n) \leftarrow$ max($|S^D(s)|$, $maxS^D(n)$)
    end
    if $n \in S$ OR $|S^D(n)| > maxS^D(n)$ (**)
        $S^D(n) \leftarrow S^D(n) \cup \{n\}$
        $S^{AL} \leftarrow S^{AL} \cup \{n\}$
    end
end
return S^{AL}

Algorithm 3. Computation of U_{lca}*-closure*

Proposition 3.2.1. (*Proof of correctness*). Given the inputs $G = (V, E)$ and S, the set S^{AL} returned by Algorithm 3 is the closure of S with respect to G, that is $S^{AL} = \bar{\bar{S}} = \bar{S}$.

Proof. Let P denote the array of vertices of G sorted by the postOrder function. The U_{lca}*-closure* algorithm goes through $P[1], \ldots, P[k], \ldots, P[|V|]$ gathering, for each k, a subset of S^{AL} denoted $S^{AL}(k)$. It is clear that $S^{AL} = S^{AL}(|V|)$. We show by induction that:

$$S^{AL}(k) = \bar{\bar{S}} \cap P[1..k] \; and \\ S^D(P[k]) = \bar{\bar{S}} \cap desc(P[k]), k = 1, 2, \ldots, |V| \quad (3)$$

In other words we want to show that $\bar{\bar{S}}$ is constructed as an increasing sequence. Each term $S^{AL}(k)$ of this sequence is closed in the subgraph of G induced by the vertices $\{P[1], \ldots, P[k]\}$. In fact, it cannot contain vertices that are in $\{P[k+1], \ldots, P[|V|]\}$. The axiom C.3 says that when taking the union of these terms, the result is the last one.

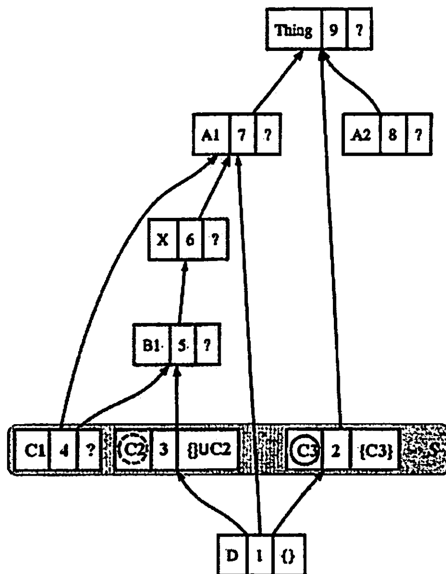

Fig. 5. U_{lca}*-closure* algorithm: considering a vertex of S.
For each vertex n, the 3 following characteristics are displayed: its label, its rank in the postOrder vector and its current set $S^D(n)$. This figure shows the information available at the point (**) when processing the vertex $C2$ (dotted circle). At this step the two sets $S^D(D)$ and $S^D(C3)$ have already been computed. The other S^D sets are not yet initialized (marked with '?'). $C3$ has been identified as part of $\bar{\bar{S}}$ (represented by a circle around it) and the algorithm is considering whether or not $C2$ belongs to $\bar{\bar{S}}$. Since $C2 \in S$ the test line (**) returns true and $C2$ will be included into $\bar{\bar{S}}$.

<u>**The statement (3) holds for $k = 1$.**</u>
Vertex $P[1]$ has no descendant. It is kept in S^{AL} if and only if $P[1] \in S \subseteq \bar{\bar{S}}$ (the line (**) of the algorithm.) Thus, $S^{AL}(1) = \bar{\bar{S}} \cap P[1..1]$ and $S^{D}(P[1]) = \bar{\bar{S}} \cap desc(P[1])$.
<u>**Assuming that (3) holds for $1, ..., (k-1)$, then it also holds for k.**</u>
Let $n = P[k]$ be the current vertex and $\{s_1, ..., s_p\}$ be the set of immediate descendants (children) of n. Since vertices are considered in post order, all children of n are at the left of n in array P. When the algorithm is considering n, all of its children $s_1, ..., s_p$ have already been treated and set $S^D(s_i)$ has been recorded for each $s_i \in \{s_1, ..., s_p\}$. At the point (**), the current recorded set for vertex n (see point (*)) is:

$$S^D(n) = S^D(s_1) \cup ... \cup S^D(s_p) \quad (4)$$

The test at the point (**) of the algorithm is used to decide whether or not n is in the closure and should be added to $S^D(n)$ and to S^{AL}.

- <u>Case $n \in S$</u>. n is added to $S^{AL}(k)$ as well as to $S^D(n)$ and evidently $n \in \bar{\bar{S}} \cap P[1..k]$ and $S^D(P[k]) = \bar{\bar{S}} \cap desc(P[k])$ (see Fig. 5 for an example).
- <u>Case $n \notin S$</u>. In this case, assuming the induction hypothesis, at the point (**), all the vertices of $S^D = S^D(s_1) \cup ... \cup S^D(s_p)$ are in $\bar{\bar{S}}$ and $S^D = S^D(n) - \{n\} = \bar{\bar{S}} \cap desc(n)$. The only thing remaining to prove is that n will be included in $S^{AL}(k)$ and in $S^D(n)$ if, and only if, it is the least common ancestor of two vertices of S^D.
  - If $|S^D(n)| > maxS^D(n)$ then there are at least two vertices z, t of S^D, such that $n = lca(z, t)$ and n should be added to $S^{AL}(k)$ as well as to $S^D(n)$. (see Fig. 6 for an example.)
    As $|S^D(n)| > maxS^D(n)$, there are at least two vertices $z, t \in S^D$ such that $\{z, t\} \nsubseteq S^D(s_i), i = 1, ..., p$. It follows that there are two distinct children s_i, s_j of n such that $z \in S^D(s_i), t \in S^D(s_j)$. By definition of the lca, $n \in lca(z, t)$ if, and only if, $n \in A(z, t)$ and n has no descendant in the ancestors set $A(z) \cap A(t)$. The former assertion is obvious, let us prove the latter by supposing that this is not the case (*reductio ad absurdum*). So, there is a vertex $n' \in A(z) \cap A(t)$ and a (n, n') directed path in G. This path necessarily goes through a child s_m of n and, according to the induction hypothesis, $S^D(s_m) = \bar{\bar{S}} \cap desc(s_m)$. It follows that $\{z, t\} \subseteq S^D(s_m)$, which is impossible.

- If the test (**) is not true, then n is not in the closure and is added neither to S^D nor to S^{AL}. (see Fig. 7 for an example.)
  The main thing to prove is that when $|S^D(n)| = maxS^D(n)$, there are not two vertices z, t of $S^D(n)$ such that $n = lca(z, t)$. As $|S^D(n)| = maxS^D(n)$ then there is some $i \in \{1, \dots, p\}$ such that $S^D(n) = S^D(s_i)$. In this case, n cannot be the lca of a couple of vertices (z, t) because the vertex s_i is (by construction) an ancestor of z, t and a descendant of n. It follows that $n \notin \bar{\bar{S}}$ and the proof is complete. □

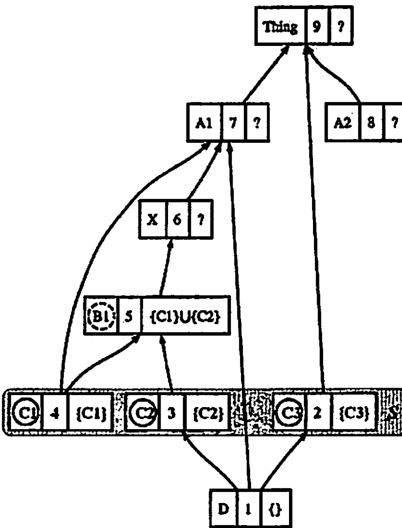

Fig. 6. U_{lca}-*closure* algorithm: considering a vertex of $\bar{S}$.
This figure displays information available at the point (**) while processing the vertex $B1$ (see Fig. 5 for legend). $C1, C2$ and $C3$ have been identified as part of the U_{lca}-*closure* of S (encircled). The current set $S^D(B1)$ is the union of $S^D(C1)$ and $S^D(C2)$. This union being larger than the two sets used to deduce it, $B1$ is identified as part of the U_{lca}-*closure* of S and $S^D(B1)$ will be updated accordingly.

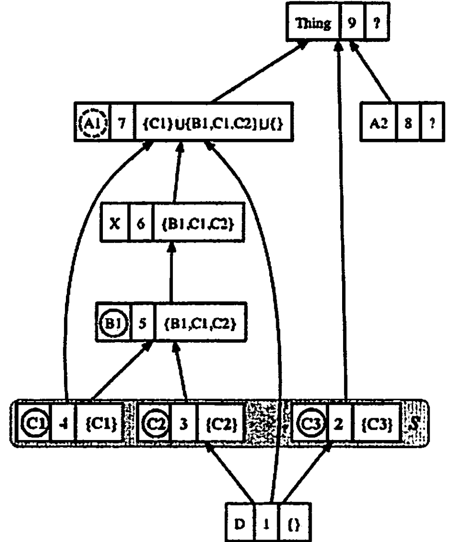

Fig. 7. U_{lca}-*closure* algorithm: considering a vertex that is not in $\bar{S}$.

The algorithm is considering $A1$. At the point (**), the set $S^D(A1)$ combines the three sets $S^D(C1)$, $S^D(X)$ and $S^D(D)$. With the resulting set being equal to $S^D(X)$ and $A1 \notin S$, $A1$ will not be added to the U_{lca}-*closure* of S.

Proposition 3.2.2. (*Time complexity of U_{lca}-closure algorithm*) For a vertex set S in a *dag* $G = (V, E)$, the U_{lca}-*closure* algorithm runs in $O(|\bar{S}||E|) = O(|V||E|)$.

Proof. Obtaining the postorder vector of vertices is done through a classical depth first search traversal of the graph in $O(|E|)$. The complexity of the remaining part of the algorithm, made of two nested for loops, is obviously determined by the number of executions of line (*).This line computes, in linear time $O(|V|)$, the union of two sets of at most $|\bar{S}|$ elements and is executed for every child of every vertex, i.e. $O(|E|)$ times. It follows that the overall complexity of this algorithm is $O(|\bar{S}||E|)$. □

Proposition 3.2.3. (*Space complexity of U_{lca}-closure algorithm*) For a vertex set S in a *dag* $G = (V, E)$, the U_{lca}-*closure* algorithm requires $O(|\bar{S}||V|) = O(|V|^2)$ memory space.

Proof. For each vertex, a subset of $\bar{S}$ is stored. In the worst case, $|\bar{S}| = |V|$ leading to a complexity of $O(|V|^2)$. □

Note that in most real cases $|\bar{S}| \ll |V|$. Moreover, when all the parents of a vertex n have been treated, the subset of $\bar{S}$ attached to n becomes useless. As a consequence, some memory space can be freed. This can be easily done by maintaining a counter for each vertex initialized to its number of parents. When treating a vertex, the counters of all of its children are decreased by one, and when a child reaches a zero value its memory is freed. This does not reduce the worst case complexity, since this optimization is useless when the *dag* is made of one vertex that has $|V| - 1$ children, but it significantly reduces the memory space needed in real applications.

3.3 Building a relevant excerpt of a *dag* from a subset of its vertices

When searching for the least relevant overset $\bar{S}$, the U_{lca}-*closure* algorithm described above provides an efficient solution to identify them. Our set of relevant vertices is $S_r = \bar{S}$.

Then, one may need to extract the corresponding excerpt of the *dag*. This reduced *dag* can be seen as a dedicated "view" of the largest *dag* and can be used to speed up further analysis or to allow end user interaction/visualization related to the task. This "relevant *dag* excerpt" must preserve the partial order among vertices of S_r that is induced by the original *dag* even though some intermediary vertices have not been kept

in S_r. More formally, given the *dag* $G = (V, E)$ and a subset S_r of V, we define the relevant *sub-dag* $G_r = (V_r, E_r)$ as:

- $V_r = S_r$
- $(u, v) \in E_r$ iff there is a directed path in G going from u to v without crossing any vertices of V_r

The set E_r of edges can be efficiently computed thanks to the topological order induced by the *dag*. This time we will consider a vertex u only after having considered all of its ascendants. Such an order can be obtained by considering the post order vector from tail to head (*reverse post-order*).

```
Name:    relevantDagExcerpt
Input:   a dag G = (V, E) a set S_r of relevant vertices
Result:  G_r = (V_r, E_r) the relevant dag excerpt.

    V_r ← S_r; G_r ← (V_r, ∅)
    for each u in reverse(postOrder(G))
        V_RRA(u) ← ∅
        for each f in parents(u)
            if f ∉ V_r
                V_RRA(u) ← V_RRA(u) ∪ V_RRA(f) (*)
            else
                V_RRA(u) ← V_RRA(u) ∪ f
        if u ∈ V_r
            for each v in V_RRA(u)
                E_r ← E_r ∪ (u, v)
    return G_r
```

Algorithm 4. Relevant *dag* excerpt algorithm

Let $V_{RRA}(u)$ be the set of *Relevant Reachable Ancestors* of u containing vertices that are present in V_r and can be reached from u through a path crossing no other vertices of V_r. When considering vertices in *reverse post-order*, the set $V_{RRA}(u)$ of the current vertex u is the union of the sets $V_{RRA}(f)$ of all its parent vertices that are not in V_r, plus all its parent vertices that are in V_r. The set E_r is then constructed by adding, for each vertex u of V_r, the edges (u, v) between u and any vertex v of $V_{RRA}(u)$ as detailed in Algorithm 4.

The complexity of this algorithm is similar to that of the U_{lca}-*closure* (Algorithm 3). As for this latter algorithm, the key instruction, line (*), computes the union of two sets of at most $|V_T|$ elements and is executed for every parent of every vertex of the initial *dag* G i.e. $O(|E|)$ times.

4 Building relevant sub-dag views based on closure: two case studies

This section illustrates the usefulness of our approach for two biological applications. The first one is related to ontology based annotation while the second one is related to species identification for metagenomic analysis.

4.1 Building sub-ontology to apprehend gene annotations

Ontologies are successfully used as semantic guides when navigating through the huge and ever increasing quantity of digital documents [12]. They are a graph based representation of domain semantics where vertices represent concepts of the fields and labeled edges represent concept relationships. The *is-a* relationship is central in ontology for it is the sole one that appears in formal ontology definition [13]; it is the sole that is present in all ontologies; and it is by far the most widely used relationship to link concepts. When restricted to *is-a* edges the ontology graph is a *dag* that is often referred to as the backbone of the ontology [13, 14].

The need for sub-ontology extraction is clearly exposed in [15], where authors point out the fact that an application focuses only on particular aspects of the whole ontology. Having concise and meaningful sub-ontology is also crucial in any computer assisted ontology operation needing a human expert, such as ontology design and evolution or visual filtering within conceptual maps. When focusing on a subset of concepts (e.g. those indexing a given document or those over represented in the index of a set of documents) a graphical representation of their *is-a* relationships is very helpful. The most widespread solution is to display those concepts of interest with *all* their ancestors. This rough solution is (manually) used in many publications (e.g. [16, 17]) as well as within Web based tools (http://www.informatics.jax.org/GOgraphs/OrthoDisease). The U_{lca}-*closure* provides a more concise excerpt of the *is-a dag* by keeping only ancestors that highlight relation among the concept of interests. Indeed this ontology problematic was at the origin of our work on

U_{lca}*-closure* and we have developed a dedicated tool called OntoFocus. To illustrate the relevance and scalability of this approach, OntoFocus has been used to restrict the Gene Ontology (containing about 30,000 terms) to the 50 concepts of the BRCA1 gene associated with BReast CAncer susceptibility according to the European Bioinformatics Institute (http://www.ebi.ac.uk/GOA/.) The corresponding sub-ontology inferred by OntoFocus in about one minute contains 92 relevant concepts.

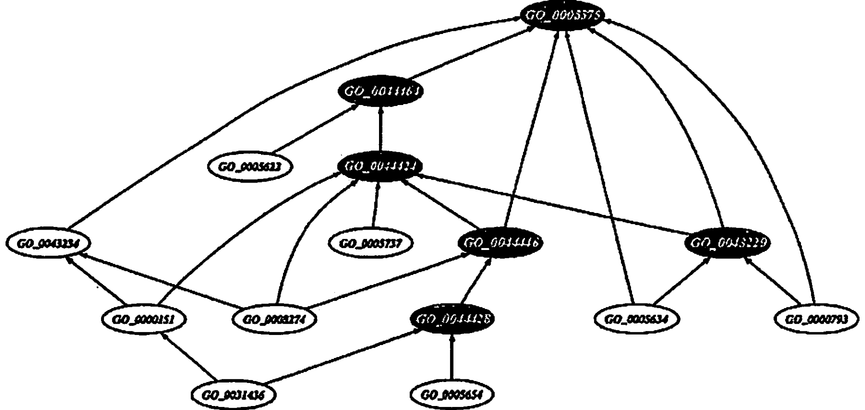

Fig. 8. Visualization of cellular component excerpt (GO_0005575).
GO-sub-ontologies constructed by OntoFocus using BRCA1 annotation (white colored concepts). Blue colored concepts were added by OntoFocus to explicit semantic relationships among white ones.

In Fig. 8 one of the three connected parts of the sub-ontology is presented, that corresponds to *cellular components*. As one may see, the visualization is very comfortable and within human cognitive and perceptive limits. The two other parts, not shown here, contain 15 and 63 terms, which also allow a comfortable visualization.

Several applications may be underlined. First of all, the user-centered sub-ontology may be useful for biological users in exploiting annotations. This highlights, for example, that several annotations are refinements of the *intracellular part* (GO_0044424.) The same approach may be used to simultaneously consider the annotation of several genes that share some biological characteristics (e.g. genes having similar expression profiles in microarray experiments.)

4.2 Identifying taxonomic group of environmental DNA sequences

New high-throughput sequencing techniques allow to obtain millions of

short portions of DNA genomes (or transcriptome) called reads. These techniques can be used to sequence DNA of a single species. In that case the quantity of obtained information allows assembling almost the whole genome of this species. Alternatively, one can also choose to sequence the whole set of genomes available in a given environment (e.g. human guts, ocean or soil sample). This is particularly useful to study the evolution of the biodiversity of the sampled environment in response to some changes (e.g. illness, climate change). In this latter case a key task is to assign the sequenced reads to a given species or taxonomic group. This is generally done based on a phylogenetic tree whose leaves represent today species and internal vertices speciation events that define taxonomic groups. To assign a taxonomic identity to a given read, the unknown DNA sequence is compared to those of the phylogeny tree leaves that are: colored in blue when similar to the read and in red otherwise. In the easiest case there is a single blue leaf and the read will be annotated with the corresponding species. For ambiguous reads, there are several blue leaves and the read is traditionally annotated based on their *lca*.

A recent paper described an original approach that performed better on simulated and real datasets [18]. The idea is to identify the internal vertex n that best annotates a read based on the number of its blue descendants (true positives), red descendants (false positives), as well as the number of blue and red leaves that are not descendant of n (true and false negatives). For doing so, it suffices to test what they also called "relevant vertices" that are the least common ancestors of two or more blue leaves. This can be seen as a particular case of our U_{lca}-*closure* when the *dag* is a tree. Moreover they provide an algorithm to restrict the taxonomic tree to this relevant set of vertices which is also a particular case of our more general *dag* excerpt algorithm. Our work provides theoretical results and an algorithm that extend their read annotation approach to the case where the taxonomy is depicted by a phylogenetic network instead of a phylogenetic tree. Phylogenetic networks are *dags* whose leaves also represent extant species that received more and more attention in evolutionary biology (a recent book is entirely dedicated to them [19]). Indeed, by authorizing a vertex to have several parents they allow to represent phylogenetic uncertainty (it is not clear which parent is the real one) and complex biological events (such as species hybridization or lateral gene transfers).

5 Concluding remarks

This paper introduced the concept of U_{lca}-*closure* of a set S of vertices in a *dag* and an optimized algorithm to identify it. This algorithm has the best known time complexity $O(|\bar{S}||E|)$ while using only $O(|V|^2)$ memory space. This low complexity comes from the convexity properties of the closure of U_{lca}-*operator* that allow to obtain a greedy algorithm.

Many applications may benefit from such an algorithm. Two of them, developed in this paper, concern the life sciences domain. One is related to the widespread Gene Ontology while the other is related to environmental metagenomic analysis.

Future directions of our work include further study of the relationship between the closure concept and convexity and related algorithm's optimization.

Acknowledgments

This publication is the result of a collaboration between the "Institut des Sciences de l'Evolution de Montpellier" (UMR 5554 - CNRS) and the "LGI2P Research Centre from Ecole des Mines d'Alès". This work was supported by the French Agence Nationale de la Recherche '*Domaines Emergents*' [ANR-08-EMER-011 '*PhylAriane*']. This publication is contribution ISEM 2011-094 of the Institut des Sciences de l'Evolution de Montpellier (UMR 5554 - CNRS), France.

References

[1] P. H. Edelman and R. E. Jamison, "The theory of convex geometries," in *Geometriae Dedicata*, vol. 19, *Mathematics and Statistics*: Springer Netherlands, 1985, pp. 247-270.

[2] M. A. Bender, G. Pemmasani, S. Skiena, and P. Sumazin, "Finding least common ancestors in directed acyclic graphs," *Proceedings of the Twelfth Annual Acm-Siam Symposium on Discrete Algorithms*, pp. 845-854, 2001.

[3] R. Yuster, "All-pairs disjoint paths from a common ancestor in O (n(omega)) time," *Theoretical Computer Science*, vol. 396, pp. 145-150, 2008.

[4] A. Czumaj, M. Kowaluk, and A. Lingas, "Faster algorithms for finding lowest common ancestors in directed acyclic graphs," *Theory of Computer Science* vol. 380, pp. 37-46, 2007.

[5] S. Eckhardt, A. M. Mühling, and J. Nowak, "Fast lowest common ancestor computations in dags," in *Proceedings of the 15th annual European conference on Algorithms*. Eilat, Israel: Springer-Verlag, 2007.
[6] J. A. Bondy and U. S. R. Murty, *Graph Theory with Appplications*. New York: American Elsevier Co., 1976.
[7] G. Cardona, M. Llabrés, F. Rossello, and G. Valiente, "Metrics for Phylogenetic Networks I: Generalizations of the Robinson-Foulds Metric," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 6, pp. 46-61, 2009.
[8] H. Aitkaci, R. Boyer, P. Lincoln, and R. Nasr, "Efficient Implementation of Lattice Operations," *Acm Transactions on Programming Languages and Systems*, vol. 11, pp. 115-146, 1989.
[9] S. Janaqi and C. Payan, "Une Caractérisation des Produits d'Arbres et des Grilles," *Discrete Mathematics*, vol. 163, pp. 201-208, 1997.
[10] H. M. Mulder and L. Nebesky, "Axiomatic characterization of the interval function of a graph," *European Journal of Combinatorics*, vol. 30, pp. 1172-1185, 2009.
[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, "Graph Algorithms," in *Introduction to Algorithms*, 6 ed. Cambridge: The MIT Press, 1992.
[12] A. Abecker and L. van Elst, "Ontologies for Knowledge Management," in *Handbook on Ontologies second edition, International handbooks on information systems*, S. S. a. R. S. (Eds.), Ed. Heidelberg: Springer, 2009, pp. 713-734.
[13] N. Guarino, D. Oberle, and S. Staab, "What is an Ontology?," in *Handbook on ontologies, International handbooks on information systems*, S. a. S. Staab, Rudi, Ed. Heidelberg: Springer, 2009, pp. 1-17.
[14] C. Ben Necib and J. C. Freytag, "Ontology based query processing in database management systems," *On the Move to Meaningful Internet Systems 2003: Coopis, Doa, and Odbase*, vol. 2888, pp. 839-857, 2003.
[15] M. Bhatt, A. Flahive, C. Wouters, W. Rahayu, D. Taniar, and T. Dillon, "A distributed approach to sub-ontology extraction," *18th International Conference on Advanced Information Networking and Applications, Vol 1 (Long Papers), Proceedings*, pp. 636-641, 2004.
[16] M. E. Dolan and J. A. Blake, "Using ontology visualization to understand annotations and reason about them," presented at KR-MED 2006 "Biomedical Ontology in Action", Baltimore, Maryland, USA, 2006.

[17] M. E. Dolan and J. A. Blake, "Using Ontology Visualization to Coordinate Cross-species Functional Annotation for Human Disease Genes," in *Computer-Based Medical Systems, IEEE Symposium on*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 583-587.

[18] J. C. Clemente, J. Jansson, and G. Valiente, "Flexible taxonomic assignment of ambiguous sequencing reads," *BMC Bioinformatics*, vol. 12, pp. 8, 2011.

[19] D. H. Huson, R. Rupp, and C. Scornavacca, *Phylogenetic Networks: Concepts, Algorithms and Applications*: Cambridge University Press, 2011.