lirmm-00269440, version 1 - 3 Apr 2008

Identifying Common Connected Components of Graphs

Anh-Tuan Gai* Michel Habib* Christophe Paul* Mathieu Raffinot†

Abstract

The Common Connected Problem (CCP) consists in identifying common connected components of two or more graphs on or reduced to the same vertex set. More formally, let $G_1(V, E_1)$ and $G_2(V, E_2)$ be two such graphs and let $G_1[V']$ and $G_2[V']$ be the two subgraphs induced by a set of vertices $V' \subset V$. If $G_1[V']$ and $G_2[V']$ are both connected, V' is said a common connected component. CCP is the identification of such maximal components (considering the inclusion order), that form a partition of V. Let $n = |V|$ and $m = |E_1|+|E_2|$. We present the first, to our knowledge, non-trivial algorithm solving CCP, running in $O(n \log n + m \log^2 n)$ worst case time. The algorithm combines a dynamical maintenance of spanning forests together with a Hopcroft-like partitioning approach.

1 Introduction

Let $G(V, E)$ be a graph. If X is a subset of vertices of G, then we denote $G[X]$ the subgraph induced by X : the set of vertices of $G[X]$ is X and its edge set is $E \cap \{(u, v) \mid u \in X, v \in X\}$. A connected component $X \subset V$ of G is such that $G[X]$ is connected. A connected component is maximal if it can not be augmented with other vertices.

In this paper, we deal with a new problem that takes as input two graphs $G_1(V, E_1)$ and $G_2(V, E_2)$ on the same vertex set V. We define $n = |V|$, $m_1 = |E_1|$, $m_2 = |E_2|$ and $m = m_1 + m_2$. We consider two graphs on the same vertices for simplicity. However, if they were not to be, we would consider the reduction of the two graphs on the set of common vertices. We now define the main notion of this article, that of common connected component.

Definition 1 *Let $G_1(V, E_1)$ and $G_2(V, E_2)$ be two graphs. A set of vertices $S \subseteq V$ is a* common connected component *of G_1 and G_2 if S is a maximal subset such that both $G_1[S]$ and $G_2[S]$ are connected.*

It is obvious that the set of common connected components of two graphs form a unique partition of V. The Common Connected Problem (CCP) is that of identifying such a partition:

CCP: { Input: two graphs $G_1(V, E_1)$ and $G_2(V, E_2)$.
Output: the partition of V into common connected components.

Practically, CCP arises in many scientific fields, as soon as two graphs have to be compared. We mention below three typical CCP applications only in computational biology. Consider for instance two interval graphs representing two different possible genomes, built on the same sequence database. Comparing the longest "common" contigs, that is, the largest

*LIRMM, 161 rue Ada, 34392 Montpellier Cedex 5, France, {gai,habib,paul}@lirmm.fr

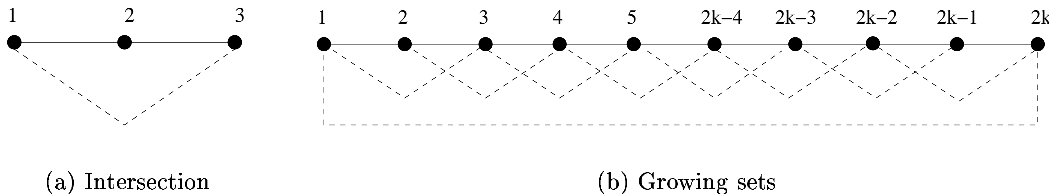
†CNRS - Laboratoire Génome et Informatique, Tour Evry 2, 523, Place des Terrasses de l'Agora, 91034 Evry, France. raffinot@genopole.cnrs.fr

lirmm-00269440, version 1 - 3 Apr 2008

set of sequences that are linked together in the two genomes, can be done solving CCP. Another typical example in this field and also in database management, is when two pairwise relations on a large set of data (for instance, a database of protein sequences linked with two different mesures) has been computed, and that the problem is to extract strong clusters, *i.e.* pairwise related in both graphs. A third example is the closely related problem of gene teams identification [1, 3]. Gene teams are sets of genes closely placed on several genomes. Solving this problem can be done using CCP on ad-hoc graphs. The reader should notice that CCP is the first step in a new area of graph comparison problems and algorithms.

Theoretically, CCP is linked to modular decomposition in a way that is not actually fully understood. Without modifying the resulting partition in common connected components, the two input graphs may be reduced to graphs with disjoint arcs by recursively merging together vertices x and y if $(x, y) \in E_1$ and $(x, y) \in E_2$, and so on until w.l.o.g $E_2 \subset \overline{E_1}$. This reduction is $O(n + m)$ time. If $E_2 = \overline{E_1}$, it corresponds to the prime case of the modular decomposition [14]. Therefore, an $O(n + m)$ modular decomposition algorithm would solve CCP for that particular case. Surprisingly enough, when E_2 is a strict subgraph of $\overline{E_1}$, CCP seems to become more complex.

The Common Connected Problem is difficult to solve efficiently for two main reasons. First, this notion is not the intersection of the connected component of G_1 and G_2. Such an example is shown in figure 1-(a). Intersecting the connected components produces $\{1, 3\}\{2\}$, while the common connected components are $\{1\}\{2\}\{3\}$. Secondly, the common connected components can not easily be grown by adding vertices one by one. Figure 1-(b) shows an example of two graphs on a set $V = \{x_1, \ldots, x_{2k}\}$ such that V is a common connected component. However, no other proper subset $S \subset V$, $|S| > 1$ is connected in both graphs.



(a) Intersection (b) Growing sets

Figure 1: Counter examples for two simple approaches, considering two graphs G_1 (full lines) and G_2 (dashed lines).

A naive approach to solve CCP recursively computes the connected components in the intersection of connected components, and so on, until the partition remains stable. However, figure 2 shows that some graphs need $O(n)$ recursive calls, yielding a worst case complexity in $O(n(n+m))$. In this graph, the vertices are isolated one by one, starting from vertex $2k$ to vertex 1. The common connected components of G_1 and G_2 are $\{1\}\{2\}\{3\} \ldots \{2k-1\}\{2k\}$.

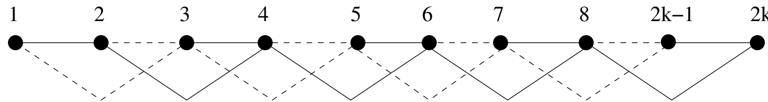


Figure 2: An example illustrating the $O(n(n+m))$ worst case bound of the naive algorithm.

Obtaining faster algorithms for solving CCP is a real challenge, since the natural ap-

lirmm-00269440, version 1 - 3 Apr 2008

proach is unusable on the sizes of the graphs now currently considered. In this paper, we present the first, to the best of our knowledge, non-trivial algorithm solving CCP, that runs in $O(n \log n + m \log^2 n)$ worst case time. Our algorithm combines a partition refinement technique together with a system of spanning forests that are dynamically maintained. Partition refinement techniques have first been introduced by Hopcroft for the automaton minimization problem [12] and then widely extended for a wide bunch of problems [3, 4, 7, 15]. We dynamically maintain spanning forests through the partition refinement process using an algorithm widely inspired by that of Holm *et al.* [11].

Our algorithm is designed for the comparison of two graphs, but it is directly extendable for comparing an arbitrary number of graphs. Because of lack of space, we do not present this extension in this extended abstract. We only cite the result. On an input k graphs $G_1(V, E_1), \ldots G_k(V, E_k)$ on the same vertices, our algorithm identifies the connected components of the k graphs in $O(kn\, log\, n\, +\, m\, log^2\, n)$, where $n = |V|$ and $m = \sum_{i=1}^{k} |E_i|$.

This article is organized as follows. We first present in Section 2 a generic partitioning framework using a naive refinement procedure. Then, we improve this refinement algorithm in Section 3. We merge both in Section 4 and prove the correctness of our complete algorithm and its time and space complexities. We conclude in Section 5.

2 Partition Refinement Paradigm

A partition refinement based algorithm relies on two main processes: (a) the *partition refinement* framework, that is a general framework that can be used for a many problems; and (b) the *refinement* itself, that can be considered as the elementary step of the former algorithm and should be specifically designed for each application. In this section, we focus on the partition refinement paradigm using a naive refinement routine that we define first. This naive refinement procedure presents all the characteristics required to prove the correctness of the partition refinement algorithm that use it, but it is totally inefficient. In Section 3, we improve the refinement technique and build ad-hoc data-structures to achieve the announced complexity.

A naive refinement routine

A partition $\mathcal{P}$ of a set V is a set of disjoint subsets $\{\mathcal{X}_1, \ldots, \mathcal{X}_k\}$, called *parts*, whose union is exactly V. A set S *separates* a part $\mathcal{X}$ if $\mathcal{X} \cap S \neq \emptyset$ and $\mathcal{X} \backslash S \neq \emptyset$. A partition $\mathcal{P}$ is said S*-stable* if no part is separated by the set S.

The pseudo-code of the naive refinement procedure **Refine** is given in figure 3. Refining a partition $\mathcal{P}$ with respect to a *pivot set* S consists in splitting each part $\mathcal{X}$ separated by S (l. 5) into smaller ones (l. 3-4). Such a part $\mathcal{X}$ is first split into two disjoint subparts $\mathcal{X}_S = \mathcal{X} \cap S$ and $\mathcal{X}_{\overline{S}} = \mathcal{X} \backslash S$ (line 3). Since we are interested in computing connected components, $\mathcal{X}_S$ and $\mathcal{X}_{\overline{S}}$ are then respectively partitioned into connected components of $G[\mathcal{X}_S]$ and $G[\mathcal{X}_{\overline{S}}]$ (l. 4). Clearly the resulting partition $\mathcal{P}$ is S-stable. The resulting parts are added to the list returned by the procedure (l. 5-10).

Lemma 1 *Let $\mathcal{P}$ be the connected component partition of the vertex set of graph $G(V, E)$ and S a subset of V. Then Algorithm* **Refine** *(fig. 3) refines $\mathcal{P}$ into the connected component partition $\mathcal{P}'$ of the graph $G'(V, E')$ where*

$$E' = E \backslash E_S \text{ with } E_S = \{(x, y) \in E \mid x \in S,\ y \in V \backslash S\}.$$

Refine($\mathcal{P}$, S)

```
1.   Let L be an empty list                                  (* list of new generated pivot sets *)
2.   For each X ∈ P Do
3.       Let SP[X] be an empty list                           (* list of new generated subparts of X *)
4.       Split X in X_S = X ∩ S and X_S̄ = X \ S
5.       Let X' = {X'_1, ..., X'_k} be the connected components of G_i[X_S] and G_i[X_S̄]
6.       If |X'| ≥ 2 Then
7.           For each X'_i ∈ X' Do
8.               Add X'_i to SP[X]
9.           End of for
10.          Add (X, SP[X]) to L
11.      End of if
12.  End of for
13.  Return L
```

Figure 3: The naive splitting routine for the common connected component problem.

Proof. Since the parts of $\mathcal{P}$ are the connected components of G, any edge of G has its extremities in the same part. Therefore, after the refinement, the extremities of a given edge (x, y) do not belong to the same part iff $x \in S$ and $y \in V \backslash S$. The result follows. □

Refinement process

Algorithm **Refine** (fig. 3) refines an input partition but also returns some new pivot sets. The generic partition refinement algorithm **GenPartRefinement** is shown in figure 4. The refinement process manages the successive calls to **Refine** until no more pivot sets can be generated (l. 5). To solve CCP, we refine two partitions ($\mathcal{P}_1$ for G_1 and $\mathcal{P}_2$ for G_2) that are initialized to the connected component partition of the two graphs (l. 1-3). Some parts of a partition serve as pivot sets to refine the other partition (l. 6-9). The routine **AddPivot** (l. 9) picks in the list L returned by **Refine** (l. 8) the sets that are inserted in the stack of pivots. Let us first assume that any set is included.

GenPartRefinement(V)

```
1.   For each P_i ∈ (P_1, P_2) Do
2.       P_i ← partition of V into the connected component of G_i
3.   End of for
4.   Let pivot ← {X | X ∈ P_1 ∪ P_2} be the stack of pivots
5.   While pivot ≠ ∅ Do
6.       Pick a pivot S in pivot
7.       If S is a part of P_1 (the case is similar for P_2) Then
8.           L ← Refine(P_2, S)
9.           AddPivot(L)
10.      End of for
11.  End of while
```

Figure 4: Generic partition refinement algorithm.

To prove the correctness of the algorithms **Refine** and **PartitionRefinement** we first

lirmm-00269440, version 1 - 3 Apr 2008

prove that the order in which the pivot sets are used does not matter. To simplify, we denote by $\mathcal{P}'$ =**Refine**$(\mathcal{P}, S)$ the partition obtained after the refinement of $\mathcal{P}$ by S.

Lemma 2 *Let $\mathcal{P}$ be the connected component partition of the set of vertices V of graph $G(V, E)$, and let S and S' be two subsets of V. Then*

$$\textbf{\textit{Refine}}(\textbf{\textit{Refine}}(\mathcal{P}, S), S') = \textbf{\textit{Refine}}(\textbf{\textit{Refine}}(\mathcal{P}, S), S').$$

Proof. By lemma 1, **Refine**$(\mathcal{P}, S)$ computes the connected components of $G^S(V, E^S)$ where $E^S = E \backslash E_S$ and **Refine**(**Refine**$(\mathcal{P}, S), S')$ computes the connected components of $\tilde{G}(V, \tilde{E})$ where $\tilde{E} = E^S \backslash (E_{S'} \backslash E_S)$. Consecutively, $\tilde{E} = E \backslash (E_S \cup E_{S'})$. The refinement operation is clearly commutative. □

Lemma 3 *Let $\mathcal{P}$ and $\mathcal{P}'$ be two partitions of the same set of vertices. If $\mathcal{P}'$ is $\mathcal{X}$-stable for any part $\mathcal{X} \in \mathcal{P}$, then any part of $\mathcal{P}$ is the disjoint union of some parts of $\mathcal{P}'$.*

Proof. Assume that there exists a part $\mathcal{X}$ of $\mathcal{P}$ that is not the disjoint union of some parts of $\mathcal{P}'$, then there exists a part $\mathcal{X}'$ of $\mathcal{P}'$ such that $\mathcal{X}' \cap \mathcal{X} \neq \emptyset$ and $\mathcal{X}' \not\subset \mathcal{X}$. By definition $\mathcal{P}'$ is not $\mathcal{X}$-stable. This yields to a contradiction. □

The following three invariants prove the correctness of algorithms **GenPartRefinement** (fig. 4) and **Refine** (fig. 3).

Invariant 1 *Each part $\mathcal{X}$ of $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is connected in G_1 (resp. G_2).*

Invariant 2 *The union of two distinct classes of $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is not connected in both graphs.*

Invariant 3 *If the partition $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is not $\mathcal{X}$-stable for every part $\mathcal{X} \in \mathcal{P}_2$ (resp. $\mathcal{P}_1$), then some pivot, in the stack pivots, will strictly refine this partition.*

Lemma 4 *Algorithms* **GenPartRefinement** *(fig. 4) and* **Refine** *(fig. 3) compute the common connected components of graphs on same vertices.*

Proof. Invariant 1 is a direct consequence of lemma 1.

Let us prove Invariant 2 by induction. It is clearly true at the initialization. So assume it is true before an arbitrary refinement step. Let $\mathcal{X}_2$ be a part of $\mathcal{P}_2$. Let $\mathcal{X}$ and $\mathcal{Y}$ be two parts of $\mathcal{P}_1'$ =**Refine**$(\mathcal{P}_1, \mathcal{X}_2)$. If $\mathcal{X} \cup \mathcal{Y}$ is a part of $\mathcal{P}_1$, this part has been separated by $\mathcal{X}_2$ and therefore is not connected in G_2. Otherwise they are included in two different parts of $\mathcal{P}_1$ and by assumption they are not connected in both graphs.

By definition $\mathcal{P}_1$ is $\mathcal{X}_2$-stable iff any part of $\mathcal{P}_1$ is either included in or disjoint from $\mathcal{X}_2$. Therefore if $\mathcal{P}_1$ is not $\mathcal{X}_2$-stable, $\mathcal{X}_2$ has never been used to refine $\mathcal{P}_1$. Since the routine **AddPivot** inserts any new part in the pivot stack, it follows that the pivot stack is not empty. Invariant 3 is satisfied.

To end the proof, assume the pivot stack is empty. Then by Invariant 3, hypothesis of Lemma 3 holds for both partitions. It follows that the partitions are the same. Moreover since by lemma 2, the pivot sets can be used in an arbitrary order, the result is unique. □

lirmm-00269440, version 1 - 3 Apr 2008

lirmm-00269440, version 1 - 3 Apr 2008

Hopcroft's framework

We improve now the routine **AddPivot**. We first prove that when a part belonging to the pivot stack is split, it can be replaced by all its subparts. To simplify, if $\mathcal{C} = \{S_1, \dots, S_k\}$ is a partition of a set S of vertices, we denote

$$\mathbf{RefineRec}(\mathcal{P}, \mathcal{C}) = \mathbf{Refine}(\dots(\mathbf{Refine}(\mathcal{P}, S_1), \dots, S_k).$$

Lemma 5 *Let $\mathcal{P}$ be a partition and S be a set partitioned into $\mathcal{C} = \{S_1, \dots, S_r\}$. Then*

$$\boldsymbol{RefineRec}(\mathcal{P}, \mathcal{C}) = \boldsymbol{RefineRec}(\boldsymbol{Refine}(\mathcal{P}, S), \mathcal{C})$$

Proof. It is straightforward to see that the set of edges involved in **Refine**$(\mathcal{P}, S)$ is included in the set of edges involved in **RefineRec**$(\mathcal{P}, \mathcal{C})$. The result follows. □

Assume now that a part $\mathcal{X}$ of $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is split into several subparts, and that $\mathcal{P}_2$ (resp. $\mathcal{P}_1$) is $\mathcal{X}$-stable. In this case, we do not need to insert all of these subparts into the pivot stack.

Lemma 6 *Let $\mathcal{P}$ be a S-stable partition of the vertex set of a graph G and let $\mathcal{C} = \{S_1, \dots, S_k\}$ be a partition of S. Then for any i, $1 \leqslant i \leqslant k$*

$$\boldsymbol{RefineRec}(\mathcal{P}, \mathcal{C}) = \boldsymbol{RefineRec}(\mathcal{P}, \mathcal{C} \backslash S_i).$$

Proof. Since $\mathcal{P}$ is S-stable, we can assume $E_S = \emptyset$. Using Lemma 5, we can reduce the proof to the case $k = 2$. Now the result is obvious, since $E_S = \emptyset$ implies $E_{S_1} = E_{S_2}$. Thus for $i = 1$ or 2, **RefineRec**$(\mathcal{P}, \mathcal{C})$ =**Refine**$(\mathcal{P}, S_i)$. □

At this point, we have gathered all the requisites to present the Hopcroft's rule, used for the first time in the automaton minimization algorithm [12].

Hopcroft's rule. *If a partition $\mathcal{P}$ is S-stable and that S is split into $\mathcal{C} = \{S_1, \dots, S_k\}$, then add to the pivot stack all the sets among the S_i with cardinality smaller than or equal to $|S|/2$.*

Using this rule, we design a new **AddPivot** procedure which pseudo-code is given figure 5.

Theorem 1 *Algorithm* **AddPivot** *(fig. 5) used as a subroutine of Algorithm* **GenPartRefinement** *(fig. 4) solve CCP using at most $\mathcal{O}(\log n)$ times each vertex in a pivot set.*

Proof. Lemmas 5 and 6 prove that managing the pivot stack in the manner of **AddPivot** 5 is equivalent to using any part as a pivot set. The correctness of the result is implied by lemma 4. Moreover, the use of Hopcroft's rule guarantees that the sizes of the pivot sets in which any vertex can appear reduces by at least half at each step. This ensures that a vertex can appear at most $\mathcal{O}(\log n)$ time in a pivot set. □

3 An Efficient Refinement Algorithm

The partition refinement paradigm we presented in the previous section is based on a naive refinement routine, that we will now develop. First, we sketch the new refinement process, then we describe in depth the data-structures we use.

```
AddPivot(L)
1.      For each (𝒳, SP[𝒳]) ∈ L  Do
2.          If 𝒳 ∈ pivot Then
3.              For each 𝒳_i ∈ SP[𝒳]  Do
4.                  Add X_i to pivot
5.              End of for
6.              Remove 𝒳 from pivot
7.          Else
8.              For each 𝒳_i ∈ SP[𝒳]  Do
9.                  If |X_i| ≤ |𝒳|/2 Then
10.                     Add X_i to pivot
11.                 End of if
12.             End of for
13.         End of if
14.     End of for
```

Figure 5: Improved AddPivot procedure.

lirmm-00269440, version 1 - 3 Apr 2008

3.1 Sketch of the refinement algorithm

Refining a partition $\mathcal{P}$ of a finite set V with a pivot S consists in maintaining connectivity while the set of edge $E_S = \{(x,y) \in E \,|\, x \in S, y \in V \backslash S\}$ is deleted. Thus, the refinement process can be mainly reduced to a decremental algorithm for graph connectivity.

Many studies have been performed about fully dynamic and decremental algorithms for graph connectivity [5, 9, 10, 11, 16]. We only recall those that are directly of interest in our case. In 1995, Henzinger and King designed a fully-dynamical algorithm running in $O(\log^3 n)$ expected amortized time complexity per update [9]. This bound was further improved to $O(\log^2 n)$ by Henzinger and Thorup in 1996 [10]. In 1997, Thorup presented a decremental randomized algorithm in $O(\log n)$ per edge deletion [16]. In 1998, Holm *et al.* provided a deterministic fully-dynamical algorithm running in amortized $O(\log^2 n)$ per update [11].

Since we do not restrict our input graphs to any probabilistic model, we first consider only deterministic approaches. Our refinement algorithm is based on the fully-dynamical algorithm for graph connectivity from [11], to which the reader may refer to for all details. We only recall below the precise points in this algorithm that are required to explain and prove our refinement algorithm.

3.1.1 Fully-dynamical connectivity algorithm

The fully-dynamical connectivity algorithm maintains a spanning forest F over the graph G. Each tree represents a maximal connected component that allows to efficiently answer connectivity queries. The edges in F are referred as *tree-edges*, and the edges in $G \setminus F$ to as *co-tree edges*. Internally, the algorithm associates to each edge a positive integer weight (or level) $l(e)$. For each i, F_i denotes the sub-forest of F induced by tree-edges of level at least i. Thus, $F = F_0 \supseteq F_1 \supseteq \ldots \supseteq F_L$. The forest F_0 is initialized with a spanning forest of G. The algorithm maintains the two following invariants.

Invariant 4 *With respect to l, F is a maximum spanning forest of G, that is, if (x,y) is a co-tree edge, x and y are connected in $F_{l(x,y)}$.*

lirmm-00269440, version 1 - 3 Apr 2008

Invariant 5 *The maximal number of nodes in a tree in F_i is $n/2^i$.*

The algorithm allows two operations on the edges, **Insert**(e) and **Delete**(e), to respectively add and remove an edge e.

- **Insert**(e) is the simplest. The new edge is given level 0. If the end-points were not connected in $F = F_0$, e is added to F_0.
- **Delete**(e) is trickier, because the spanning forests need to be recovered (if possible) after removing e. This is done using an involved procedure **Replace**$((x, y))$, that recursively searches for a replacement co-tree edge in decreasing weight order when a tree-edge is deleted. This procedure requires sophisticated data structures that are described in the next subsection.

To insure a good worst case complexity, the algorithm amortizes the edge replacement costs over the weights of the edges. These weights never decrease, and the respect of invariant 5 by **Delete**(e) guarantees that the weight of an edge may increase at most $L = \lfloor \log_2 n \rfloor$ times.

3.1.2 Data-structures

This algorithm maintains, for each i, the forest F_i together with all co-tree edges on level i. It answers, for any vertex x, which tree T_x in F_i contains it. It computes the size of T_x. It finds an edge of T_x on level i if one exists. It finds a level i co-tree edge incident to T_x, if any. The trees in F_i may be cut and linked. Finally, any edge on level i may disappear.

All the above operations and queries may be supported in $O(\log n)$ time using the ET-trees from [9]. An ET-tree is a standard balanced binary tree over the Euler tour of a tree. Each node in the ET-tree represents the segment of the Euler tour below it. The point in considering Euler tours is that if trees in a forest are linked or cut, the new Euler tours can be constructed by at most two splits and two concatenations of the original Euler tours. Rebalancing the ET-trees affects only $O(\log n)$ nodes.

There is one ET-tree over each tree in F_i. Each node of the ET-tree contains a number telling the size of the Euler tour segment below it, a bit telling if any tree-edge in the segment has level i, and a bit telling whether there is any level i co-tree edge incident to a vertex in the segment. We denote below ET(x)-tree the ET-tree of F_0 that contains x.

When an edge is inserted on level i, the direct cost is $O(\log n)$. However its level may increase $O(\log n)$ times, so the amortized cost is $O(\log^2 n)$. Deleting a co-tree edge takes time $O(\log n)$. When a tree-edge e is deleted, all the forests F_j, $j \leq l(e)$ are cut, giving an immediate cost of $O(\log^2 n)$. Then **Replace()** is called at most $O(\log n)$ times, each call costs $O(\log n)$ plus the costs amortized over the weights of the edges. Finally, if a replacement edge is found, $O(\log n)$ forests are linked in $O(\log^2 n)$ total time.

Theorem 2 ([11]) *Given a graph G with m edges and n vertices, there exists a deterministic fully dynamic algorithm that maintains a spanning forest of G in $O(\log^2 n)$ amortized time per update.*

3.2 Refinement algorithm, correctness proof

The new refinement process acts as the naive one (section 2), but uses the ET-trees data-structures. The partition $\mathcal{P}$ to refine is represented by a list of ET-trees roots of a spanning

lirmm-00269440, version 1 - 3 Apr 2008

```
FastRefine(𝒫, r the ET-tree root of a pivot set S of type 𝒫')
outputs a list L that represents the new parts of 𝒫 and the parts they are issued from
 1.     Let L and L_used be two empty lists
 2.     Let V_S be the set of vertices in ET(r)-Tree in F_0
 3.     Compute E_{V_S} (cf Lemma 1)
 4.     Let E_c (resp. E_t) be the set of co-tree (resp. tree) edges in E_{V_S}
 5.     For each edge e in E_c Do
 6.         Delete(e) from G
 7.     End of for
 8.     For each edge (x,y) in E_t Do
 9.         Compute t the root of ET(x)-tree in F_0
10.         Let s_t be the size of the Euler tour segment below t in F_0
11.         If t ∉ L_used Then
12.             Let SP[t] be an empty list
13.             Add (t, s_t, SP[t]) to L
14.             NewPart(t, s_t, t)
15.         End of if
16.         Delete((x,y)) from G'
17.         Compute u (resp. v) the root of ET(x)-tree (resp. ET(y)-tree) in F_0
18.         Let s_u (resp. s_v) be the size of the Euler tour segment below u (resp. v) in F_0
19.         Remove (t, s_t) from SP[Origin[t]]
20.         NewPart(u, s_u, Origin[t])
21.         If u ≠ v Then
22.             NewPart(v, s_v, Origin[t])
23.         End of if
24.     End of for
25.     return L

NewPart(x, s_x, o_x)
26.     Origin[x] ← o_x;   add (x, s_x) to SP[o_x]
27.     Add x to L_used
```

Figure 6: Efficient refinement algorithm splitting a partition with a pivot S.

forest, while the pivot set S is represented by a root of a single ET-tree. The algorithm returns a list L containing the new generated subparts. For technical reasons the elements of L are triplets $(t, s_t, SP[t])$ each of those corresponding to a split part and its new subparts. Indeed, (a) t is a root of an original ET-tree in F_0; (b) s_t is the size of the Euler tour below that root in the original tree in F_0; (c) $SP[t]$ is a list of pairs, each of those being a new generated subpart issued from that represented by t together with the size of its Euler tour.

Let us briefly describe the algorithm. First, refer to what is stated in Lemma 1, the parts of the partition correspond to the connected components of a graph. Refining the partition consists in removing from that graph the edge set E_S associated to the pivot set S. In E_S one can distinguish the *tree edges* (that belong to a spanning tree of a connected component corresponding to a part) from the co-tree edges. Clearly the co-tree edges of E_S just have to be removed from G (l. 5-6). Let e be a tree edge. Both of its extremities belong to the same original part, say $\mathcal{X}$. By definition of S it is straightforward that $\mathcal{X}$ will be split (not necessarily by the removal of e). Therefore $\mathcal{X}$ can be added to the list L (l. 11-15). To process e, we remove it and try to find a replacement edge (l. 16). If such an edge exists, the root of the ET-tree may change and some updates in the lists have to be done (l. 19-20). Otherwise the ET-tree associated to $\mathcal{X}$ is split into two new ET-trees. The list of new generated subparts is updated (l. 20-23). The correctness of the algorithm derives from that of the fully-dynamical algorithm of [11].

Theorem 3 *Let $\mathcal{P}$ be the connected component partition of the vertex set of graph $G(V, E)$ whose parts are represented by ET-trees and let S be a subset of V. Then Algorithm* ***FastRefine*** *(fig. 6) refines $\mathcal{P}$ into the connected component of $G'(V, E \setminus E_S)$. It also outputs a list containing the new generated subparts.*

4 Complete algorithm

We now join together the general refinement framework **GenPartRefinement** of section 2 and the specific refinement process **FastRefine** of section 3. The partition refinement process is adapted to the data structures used by the refinement process. The correctness proof and the complexity issues follow.

The pseudo_code of the **PartitionRefinement** procedure is shown in figure 8. The forest maintained for G_1 and G_2 are respectively F^1 and F^2. Partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ are respectively represented by F_0^1 and F_0^2. Each ET-tree in F_0^1 or in F_0^2 represents a part of its corresponding

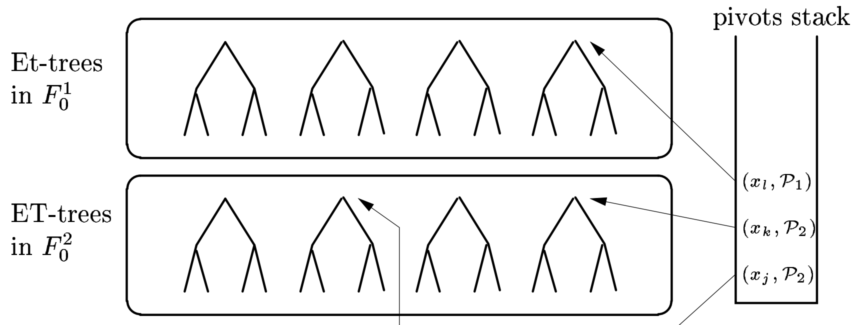


Figure 7: Partitions and data-structures of the pivots.

lirmm-00269440, version 1 - 3 Apr 2008

partition (by the set of distinct vertices it contains). When no confusion is possible, the parts in $\mathcal{P}_1$ and $\mathcal{P}_2$ denote the sets of distinct vertices of the ET-trees in F_0^1 and F_0^2. The stack of pivots contains some of the ET-trees in F_0^1 or in F_0^2, represented by their roots (see subsection 3.1.2 and figure 7). The initializations of lines 2-9 (fig. 8) correspond to the lines 1-4 of the generic partition refinement **GenPartRefinement** (fig. 4). The main while loop (l. 11) corresponds to the while loop (l. 5) of the generic algorithm (fig. 4).

Lines 15-27 correspond to the **AddPivot** procedure (fig. 5). Since a pivot is the root of an ET-tree, a part X_i resulting from Refine$(\mathcal{X}, S)$ may have the same ET-tree root as $\mathcal{X}$. This explains why we delete the old parts (old roots) from the pivots stack (l. 17), before we add the new part (new root) to the pivots stack (For loop line 18-20). Each node of the ET-tree contains a number telling the size of the Euler Tour segment below it (see subsection 3.1.2). The size of the Euler tour of a tree is twice the number of edges. Let s_X be the size of the segment of the Euler tour below the root of a tree in F_0^1 or in F_0^2, containing the vertices of a part $\mathcal{X}$. The number of vertices in $\mathcal{X}$ is equal to $(s_X/2) + 1$. Therefore, the test of line 23 corresponds to that of **AddPivot** line 9.

```
PartitionRefinement(graphs G1(V,E1), G2(V,E2) )
1.    Initializations
2.        Let G'1, G'2 be two empty graphs on V
3.        For each edge e of E1 (resp. E2) Do
4.            insert(e) in G'1 (resp. G'2)
5.        End of for
6.        Let pivots be an empty stack of pivots
7.        For each root r in F0^1 (resp. F0^2) Do
8.            add r of type P1(resp. P2) to pivots
9.        End of for
10.   Refinements
11.       While (pivots is not empty) Do
12.           pick a pivot S in pivots
13.           If S has type P2 (the case type P1 is similar) Then
14.               M ← FastRefine(P1, S)
15.               For each (t, s_t, SP[t]) ∈ M Do
16.                   If t of type P1 ∈ pivots Then
17.                       Delete t from pivots
18.                       For each (x, s_x) in SP[t] Do
19.                           Add x of type P1 to pivots
20.                       End of for
21.                   Else
22.                       For each (x, s_x) in SP[t] Do
23.                           If s_x/2 + 1 ≤ (s_t/2 + 1)/2 Then
24.                               Add x of type P1 to pivots
25.                           End of if
26.                       End of for
27.                   End of if
28.               End of for
29.           End of if
30.       End of while
```

Figure 8: Hopcroft-like algorithm for common connected components.

lirmm-00269440, version 1 - 3 Apr 2008

lirmm-00269440, version 1 - 3 Apr 2008

The following theorem is a directly derives of lemma 1 and theorems 1, 2, and 3.

Theorem 4 *The* **PartitionRefinement** *algorithm (fig. 8) using the procedure* **FastRefine**$(\mathcal{P}, S)$ *(fig. 6) computes the common connected components of two graphs.*

4.1 Complexity issues

A vertex x is said "added" to the pivots stack when an ET-tree (represented by its root) that contains it is added to the stack of pivots.

Theorem 1 ensures that a vertex x is added at most $\log n$ times in the stack of pivots. This is a central complexity argument of both **PartitionRefinement** (fig. 8) and **FastRefine** (fig. 6).

Amortized complexity of the refinement algorithm

We first consider the complexity related to the edges. As a vertex x is added at most $\log n$ times in the stack of pivots (theorem 1), its adjacing edges will be considered at most $O(\log n)$ times when identifying the edges that have to be deleted (l. 3). Moreover, an edge is deleted at most once (l. 7 or l. 18), with an $O(\log^2 n)$ amortized worst case time complexity (theorem 2). The **FastRefine** algorithm is the only part of the complete algorithm that deletes edges. Therefore, the global worst case time complexity with regards to the edges is $O(m \log^2 n)$.

About the complexity related to the vertices of the pivot part, lines 2-3 can be done proportionally to the number of vertices in the ET-tree pivot in F_0. This ensures that the amortized complexity over each vertex for each **FastRefine** call is $O(1)$ time.

Amortized complexity of the complete algorithm

Each line in the initializations of **PartitionRefinement** (fig. 8) costs $O(n)$ time, excepted the For loop line 3 that costs $O(m \log n)$. Therefore, the initialization is $O(n + m \log n)$ time.

Lines 11-30 of **PartitionRefinement** directly correspond to the **Addpivot** procedure (fig. 5). As a vertex x is added at most $\log n$ times in the stack of pivots (theorem 1) and that its amortized participation to the pivot processing by **PartitionRefinement** is $O(1)$, the amortized complexity of each vertex is at most $O(\log n)$. Therefore, the global complexity over all vertices is $\sum_{x \in V} O(\log n) = O(n \log n)$.

Considering space complexity, each vertex and each edge appear twice in at most $O(\log n)$ spanning forests. Therefore, the overall space complexity is $O((n + m) \log n)$.

Theorem 5 *Let $G_1(V, E_1)$ and $G_2(V, E_2)$ be two disjoint graphs on same vertices, with $n = |V|$, $m_1 = |E_1|$, $m_2 = |E_2|$, $m = m_1 + m_2$. The algorithm* ***PartitionRefinement*** *using* **FastRefine** *procedure runs in $O(n\, log\, n + m\, log^2\, n)$ worst case time and $O((n + m) \log n)$ space.*

5 Conclusion

Graph comparisons on the same vertices is still very recent and is rapidly expanding because of the increasing needs in many scientific applications, especially in computational biology. We introduced a new basic problem in this field, called the CCP problem, and we presented an efficient non trivial $O(n\, log\, n + m\, log^2\, n)$ worst case time algorithm that combines a Hopcroft-like partionning algorithm together with a dynamical maintenance of spanning forests.

lirmm-00269440, version 1 - 3 Apr 2008

References

[1] M.-P. Béal, A. Bergeron, and M. Raffinot. Gene Teams and Hopcroft's Partionning Framework. 2003. Submitted.

[2] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Masson, Paris, 1992.

[3] 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, Berlin, 2002.

[4] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19(1):85–98, 1982.

[5] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification — A technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.

[6] A.-T. Gai, M. Habib, C. Paul, and M. Raffinot. Identifying Common Connected Components of Graphs. *Technical report LIRMM-03016*, 2003. `http://www.lirmm.fr/~paul/Biblio/perso.html`.

[7] M. Habib, C. Paul, and L. Viennot. A synthesis on partition refinement: a useful routine for strings, graphs, boolean matrices and automata. In *15th Symposium on Theoretical Aspect of Computer Science (STACS)*, number 1373 in Lecture Notes in Computer Science, pages 25–38. Springer-Verlag, Berlin, 1998.

[8] M. Habib, C. Paul, and L. Viennot. Partition refinement techniques: an interesting algorithmic tool kit. *International Journal of Foundations of Computer Science*, 10(2):147–170, 1999.

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

[10] M.R. Henzinger and M. Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Structures and Algorithms*, 11(4):369–379, 1997.

[11] 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.

[12] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[13] N. Luc, J.-L. Risler, A. Bergeron, and M. Raffinot. Gene Teams: A New Formalization of Gene Clusters For Comparative Genomics. *Computational Biology and Chemistry (ex. Computer and Chemistry)*, 2002. To appear.

[14] R. H. Möhring and F. J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics*, 19:257–356, 1984.

[15] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[16] M. Thorup. Decremental dynamic connectivity. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.