

Epita:Algo:Cours:Info-Spe:les connexités

De EPITACoursAlgo.

Rappels :

- Un graphe non orienté est connexe si pour toute paire de sommets disjoints u et v , il existe une chaîne ($u \rightsquigarrow v$) reliant ces deux sommets.
- Un graphe orienté est fortement connexe si pour toute **paire ordonnée** de sommets disjoints u et v , il existe un chemin de u vers v ($u \rightsquigarrow v$).

Dans ce chapitre, nous allons présenter des algorithmes permettant de déterminer les composantes connexes de graphes non orientés. Puis, nous nous placerons dans le cadre de graphes orientés et donc de l'extraction des composantes fortement connexes.

On peut bien sur vouloir être plus exigeant et désirer qu'un graphe reste connexe, même après la suppression d'un ou plusieurs de ses sommets quelconques. Cela s'appelle la **p-connexité**, p étant un entier représentant le nombre maximum de sommets pouvant être retirés tout en conservant la propriété de connexité du graphe. Nous nous limiterons à p=2.

Sommaire

- 1 Composantes connexes
 - 1.1 Cas statique
 - 1.1.1 Parcours en profondeur
 - 1.1.2 Parcours en largeur
 - 1.1.3 Fermeture transitive
 - 1.1.3.1 Spécification formelle
 - 1.1.3.2 Algorithme de WARSHALL
 - 1.1.4 Cas évolutif
 - 1.1.4.1 Spécification formelle
 - 1.1.4.2 Algorithmes de base
 - 1.1.4.3 L'union pondérée
 - 1.1.4.4 La compression de données
 - 2 Composantes fortement connexes
 - 2.1 Algorithme de Tarjan (double parcours et inversion de graphe)
 - 2.2 Algorithme de Tarjan (parcours simple et pile)
 - 3 Biconnexité

Composantes connexes

Cette section traitera de l'extraction des composantes connexes d'un graphe non orienté G défini par un couple $\langle S, A \rangle$. L'analyse du résultat est toujours la même; Si en sortie nous n'avons qu'une seule composante, le graphe est connexe, sinon nous pouvons déterminer les sommets membres de chaque composante et donc ceux pour lesquels il existe une relation chaîne (\rightsquigarrow).

Remarque : Nous partirons du principe que l'ensemble de sommets S est statique (fixé) et que l'ensemble des arêtes A peut être statique ou évolutif.

Cas statique

Les trois moyens de déterminer les composantes connexes d'un graphe non orienté statique (S et A fixés) sont le parcours en profondeur, le parcours en largeur et le calcul de **la fermeture transitive**. Nous connaissons les deux premiers, nous allons découvrir le troisième.

Parcours en profondeur

A l'issu du parcours en profondeur, nous examinons la forêt couvrante associée à celui-ci. S'il n'y a qu'une arborescence, le graphe est connexe, sinon il y a autant de composantes connexes qu'il y a d'arborescences et les sommets de chacune d'elles sont les noeuds qui les composent.

Parcours en largeur

Le principe est globalement le même que pour le parcours en profondeur, seul l'ordre de rencontre des sommets varie (ordre hiérarchique).

Fermeture transitive

Calculer la fermeture transitive d'un graphe sert à déterminer si ce graphe est connexe.

Définition: On appelle **fermeture transitive** d'un graphe G défini par le couple $\langle S, A \rangle$, le graphe G^* défini par le couple $\langle S, A^* \rangle$ tel que pour toutes paires de sommets $x, y \in S$, il existe une arête $\{x, y\}$ dans G^* si et seulement si il existe une chaîne entre x et y dans G .

Remarques :

- La fermeture transitive G^* d'un graphe connexe G est un graphe complet
- Un graphe G est connexe si sa fermeture transitive G^* est un graphe complet
- Deux sommets x, y sont de la même composante connexe de G si et seulement si il existe une arête $\{x, y\}$ dans la fermeture transitive G^* de G .

Spécification formelle

Nous allons réaliser la construction de la fermeture transitive par itérations successives. C'est à dire que pour tester l'existence d'une chaîne entre deux sommets x et y , nous allons vérifier s'il en existe une qui passe par le sommet s_1 , puis s'il en existe une qui passe par les sommets s_1 et s_2 , etc. Finalement s'il en existe une qui passe par tous les sommets du graphe.

Remarque : Si les sommets sont des entiers (1, 2, 3, ...) l'opération de calcul de la fermeture transitive se traduit par une simple itération.

On définit alors les opérations **itérer** et **fermeturetransitive** :

```

opérations
  fermeturetransitive : graphe → graphe
  itérer : entier × graphe → graphe

axiomes
  /* Un graphe a le même ensemble de sommets que sa fermeture transitive */
  s estun sommet de itérer(i,g) = s estun sommet de itérer(i-1,g)

  <s,s'> estun arête de itérer(i,g) =
    <s,s'> estun arête de itérer(i-1,g) |
    (<s,i> estun arête de itérer(i-1,g) & <i,s'> estun arête de itérer(i-1,g))
  <s,s'> estun arête de fermeturetransitive(g) =
    <s,s'> estun arête de itérer(nbsommets(g),g)

avec
  graphe g
  entier i          /* i > 0 */
  sommet s, s'      /* s et s' sommets de g */

```

Algorithme de WARSHALL

Principe :

Comme le montrent les axiomes précédents, il existe une chaîne allant d'un sommet s à un sommet s' en passant par des sommets inférieurs ou égaux à i , soit :

- s'il existe une chaîne allant d'un sommet s à un sommet s' en passant par des sommets inférieurs ou égaux à $i-1$,
- s'il existe une chaîne allant d'un sommet s à un sommet i en passant par des sommets inférieurs ou égaux à $i-1$ et s'il existe une chaîne allant d'un sommet i à un sommet s' en passant par des sommets inférieurs ou égaux à $i-1$.

Soit c la matrice d'adjacence de G , en suivant la spécification, nous pouvons déterminer la matrice c^* de G^* . Supposons que $c_{i[s,s']}$ représente l'existence d'une chaîne allant de s à s' passant par des sommets inférieurs ou égaux à i , on a alors :

$c_{i[s,s']} = c_{i-1[s,s']} \text{ ou } (c_{i-1[s,i]} \text{ et } c_{i-1[i,s']})$

Pour l'algorithme qui suit, le type de données employé est le suivant :

```
Constantes
Nbs = ...                                /* Nombre de sommets du graphe */

Types
t_matNbsNbsbool = Nbs x Nbs booléen    /* Matrice booléenne carrée de Nbs sommets */
```

Ce qui donne :

```
Algorithme procédure Marshall
Paramètres locaux
  t_matNbsNbsbool c                      /* Matrice d'adjacence du graphe g */
Paramètres locaux
  t_matNbsNbsbool c2                     /* Matrice d'adjacence du graphe g* */
Variables
  entier i, s, s2                         /* i, s et s2 sont des sommets */
Début
  /* Initialisation de C2 */
  pour s ← 1 jusqu'à Nbs faire
    pour s2 ← 1 jusqu'à Nbs faire
      C2[s,s2] ← C[s,s2]
    fin pour
  fin pour

  /* Calcul de C2 (fermeture-transitive) */
  pour i ← 1 jusqu'à Nbs faire
    pour s ← 1 jusqu'à Nbs faire
      pour s2 ← 1 jusqu'à Nbs faire
        C2[s,s2] ← C2[s,s2] ou (C2[s,i] et C2[i,s2])
      fin pour
    fin pour
  fin pour
Fin Algorithme procédure Marshall
```

Remarque : Dans l'algorithme, nous n'utilisons qu'une seule matrice \mathbf{C}_2 pour calculer les valeurs successives de \mathbf{C}_i . En effet, $\mathbf{C}_i[s,s]$ vaut déjà $\mathbf{C}_{i-1}[s,s]$ avant chaque nouvelle évaluation. L'utilisation de l'opérateur **ou** nous garantit la conservation des valeurs **vrai** qui auraient été obtenues lors des itérations précédentes.

Cas évolutif

Un graphe non orienté de n sommets peut être connu par la donnée successive de ses arêtes. Les composantes connexes du graphe peuvent alors être obtenues par regroupement successifs des composantes connexes du graphe en évolution.

L'ajout d'une arête $x --- y$ peut, en effet, modifier l'ensemble des composantes connexes d'un graphe en réunissant celle de x avec celle de y si elles sont distinctes.

On doit alors être capable :

- de dire à quelle composante connexe appartient un sommet donné,
- de savoir si deux sommets appartiennent ou non à une même composante,
- de réunir deux composantes connexes en une seule.

Remarque : Déterminer les composantes connexes d'un graphe G , c'est déterminer l'ensemble des classes d'équivalence pour la relation d'équivalence **chaîne**, ou x **chaîne** y est vrai si-et-seulement-si il existe une chaîne entre les deux sommets x et y .

De manière plus générale, on utilise les fonctions **trouver** et **réunir** pour construire la partition associée à une relation d'équivalence évolutive (*relation dont les instances sont connues progressivement*).

Ces deux fonctions sont utilisées de la manière suivante :

- **trouver** associe à tout élément de l'ensemble sur lequel est défini la relation d'équivalence un élément privilégié de sa classe d'équivalence (*le représentant de la classe*)
- **réunir** rassemble deux classes d'équivalence distinctes en une seule.

Spécification formelle

Un ensemble muni d'une relation d'équivalence peut être considéré comme un graphe non orienté qui vérifie certaines propriétés. Les éléments de l'ensemble sont alors représentés par les sommets et la relation d'équivalence par la relation arête.

Remarque : S'il s'agit de graphes non orientés, la propriété de symétrie résulte des axiomes.

En fait, on ne connaît pas toute la relation d'équivalence, mais seulement certaines occurrences données progressivement.

Comme on sait que c'est une relation d'équivalence, l'ajout explicite d'une seule occurrence (*celui d'une arête dans le graphe G*) s'accompagne de l'ajout implicite d'un certain nombres d'autres occurrences (*d'autres arêtes dans le graphe G**).

Cela revient à construire *la fermeture réflexive et transitive du nouveau graphe*.

Appelons **fermeture** l'opération qui effectue la construction de la fermeture réflexive et transitive d'un graphe. Sa spécification utilise l'opération **fermeturetransitive** spécifiée précédemment.

On a donc, pour **fermeture**, **trouver** et **réunir** les définitions suivantes :

```

opérations

fermeture : graphe → graphe
/* définie partout */
trouver : Sommet x Graphe → Sommet
/* Définie uniquement sur des graphes représentant des relations d'équivalence */
réunir : Sommet x Sommet x Graphe → Graphe
/* Définie uniquement sur des graphes représentant des relations d'équivalence */

préconditions
trouver(s,g) est définissi s est un sommet de g = Vrai & g = fermeture(g)
réunir(s,s',g) est définissi s est un sommet de g = Vrai &
                           s' est un sommet de g = Vrai &
                           g = fermeture(g)

axiomes

(* Un graphe a le même ensemble de sommets que sa fermeture *)
s est un sommet de fermeture(g) = s est un sommet de g
(* Réflexivité *)
s est un sommet de g = Vrai ⇒ <s,s> est une arête de fermeture(g) = Vrai
(* Transitivité *)
s ≠ s' ⇒ <s,s'> est une arête de fermeture(g) = <s,s'> est une arête de fermeturetransitive(g)
(* Deux éléments d'une même classe d'équivalence ont le même représentant *)
<s,s'> est une arête de g = Vrai ⇒ trouver(s,g) = trouver(s',g)
(* Le représentant se représente lui-même *)
trouver(trouver(s,g),g) = trouver(s',g)
(* Quand on ajoute une occurrence à la relation (arête), *)
(* on construit sa fermeture réflexive et transitive *)
réunir(s,s',g) = fermeture(ajouter l'arête <s,s'> à g)

avec

graphe g
sommet s, s'      (* s et s' sommets de g *)

```

Algorithmes de base

Principe :

Nous allons représenter les classes d'équivalence sous la forme d'une forêt. Chaque arbre de cette forêt a pour racine le représentant de la classe qu'il représente (l'arbre). Dans ce cas **trouver(x,G)** revient à chercher l'arbre qui contient x et à retourner la racine de cet arbre. Si on connaît le père de chaque élément, cela devient enfantin (*c'est vrai :)*). De plus, pour réunir deux classes d'équivalence, il suffit d'accrocher un des arbres à la racine de l'autre (*c'est ce que je disais, un enfant de 5 ans y arriverait*).

Algorithme :

Pour représenter la forêt, nous allons utiliser un tableau de **n** entiers, appelé **pere**, représentants les descendants directs des noeuds dans la forêt. Si un noeud **i** ∈ **[1,n]** a pour père un noeud **j** ∈ **[1,n]**, on aura **pere[i] = j**. Si un noeud **i** ∈ **[1,n]** n'a pas de père, à ce moment là, on aura **pere[i] = -1**.

Remarques :

- Cette valeur servira de sentinelle lors de la recherche de la racine d'une arborescence. En effet, la racine d'un arbre est le seul noeud qui n'a pas de père.

- Si au départ, toutes les classes d'équivalence sont réduites à des singletons, notre tableau **pere** verra tous ses éléments initialisés à -1.

Pour les algorithmes qui suivent, le type de donnée employé est le suivant :

```
Constantes
Nbs = ...                                /* Nombre de sommets du graphe */

Types
t_vectNbsEnt = Nbs entier    /* Vecteur d'entiers mémorisant l'ascendant direct */
```

Ce qui donne pour **trouver** :

```
/* trouver(x, pere) retourne la racine de l'arbre contenant x */
Algorithme fonction trouver : entier
Paramètres locaux
  entier      x
  t_vectNbsEnt pere                      /* Vecteur d'ascendance */
Début
  tant que pere[x] > 0 faire
    x ← pere[x]
  fin tant que
  retourne(x)
Fin Algorithme fonction trouver
```

Et pour **réunir** :

```
/* réunir(x, y, pere) accroche la racine de l'arbre contenant y */
/* à la racine de celui contenant x, s'ils sont distincts */
Algorithme procédure réunir
Paramètres locaux
  entier      x,y
Paramètres globaux
  t_vectNbsEnt pere                      /* Vecteur d'ascendance */
Variables
  entier rx, ry
Début
  rx ← trouver(x,pere)
  ry ← trouver(y,pere)
  si rx <> ry alors
    pere[ry] ← rx
  fin si
Fin Algorithme procédure réunir
```

Nous allons présenter un exemple de construction de graphe par ajouts successifs d'arêtes. Ce graphe possède **18** sommets et ses arêtes sont connues progressivement dans l'ordre suivant:

1 --- 6, 3 --- 2, 8 --- 7, 12 --- 15, 17 --- 16, 11 --- 10, 9 --- 8, 17 --- 18, 11 --- 7, 1 --- 2, 4 --- 5, 13 --- 15, 10 --- 9, 15 --- 14, 4 --- 6, 13 --- 14, 16 --- 15, 3 --- 4, 5 --- 6, 18 --- 12

Nous découperons la construction de ce graphe en 4 phases, ce qui nous permettra de mieux comprendre le fonctionnement de ces algorithmes. Nous présenterons pour chaque phase d'ajouts (cf. Tableaux 1, 2, 3 et 4), l'état du tableau **pere** ainsi que la forêt correspondante.

Tableau 1. Phase N°1.

Arêtes	Forêt obtenue
1 --- 6, 3 --- 2, 8 --- 7, 12 --- 15, 17 --- 16, 11 --- 10, 9 --- 8, 17 --- 18,	
Vecteur pere	

sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Père	-1	3	-1	-1	-1	1	8	9	-1	11	-1	-1	-1	-1	12	17	-1	17

Tableau 2. Phase N°2.

Arêtes	Forêt obtenue
11 --- 7, 1 --- 2, 4 --- 5, 13 --- 15	<pre> graph TD 1 --- 6 1 --- 4 4 --- 5 11 --- 10 11 --- 9 13 --- 12 17 --- 16 17 --- 18 2 --- 6 8 --- 10 15 --- 12 7 --- 15 </pre>
Vecteur pere	
sommet	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Père	-1 3 1 -1 4 1 8 9 11 11 -1 13 -1 -1 12 17 -1 17

Tableau 3. Phase N°3.

Arêtes	Forêt obtenue
10 --- 9, 15 --- 14, 4 --- 6, 13 --- 14	<pre> graph TD 4 --- 1 4 --- 5 11 --- 10 11 --- 9 13 --- 12 13 --- 14 17 --- 16 17 --- 18 1 --- 6 1 --- 3 6 --- 2 8 --- 10 15 --- 12 7 --- 15 </pre>
Vecteur pere	
sommet	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Père	4 3 1 -1 4 1 8 9 11 11 -1 13 -1 13 12 17 -1 17

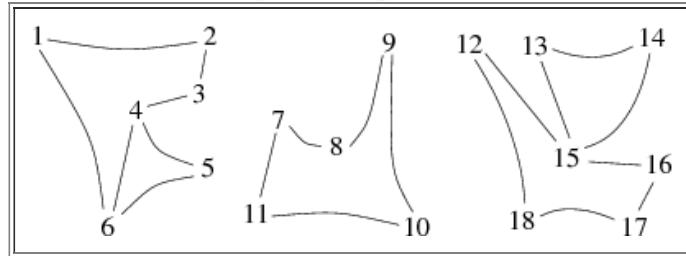
Tableau 4. Phase N°4.

Arêtes	Forêt obtenue
16 --- 15, 3 --- 4, 5 --- 6, 18 --- 12	

Vecteur <i>pere</i>	
sommet	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Père	4 3 1 -1 4 1 8 9 11 11 -1 13 17 13 12 17 -1 17

Les arbres de la forêt finale correspondent aux composantes connexes du graphe. Les liens ne sont pas forcément ceux du graphe. Sur cet exemple, le graphe réellement obtenu est celui représenté par la figure 1 (*il est à noter que weJ signifie trois en Klingon, délire non ?*).

Figure 1. Graphe weJ.



On peut voir que la forêt obtenue à une profondeur maximale de 3 pour 18 sommets. La forme obtenue est assez compacte. Malheureusement, ce n'est pas toujours le cas. Par exemple, si l'on prend un graphe de n sommets dont les $n-1$ arêtes sont données dans l'ordre suivant :

2 --- 1, 3 --- 1, ..., (i-1) --- 1, i --- 1, ..., n --- 1

La forêt obtenue par appels successifs de la procédure **réunir** est constituée d'un unique arbre de hauteur ***n-1*** et de racine ***n*** (*arbre dégénéré*). Dans ce cas, la complexité au pire de **trouver** est $\Theta(n)$. Cela revient à dire que tester l'appartenance d'un sommet à une composante connexe (*classe d'équivalence*) peut prendre au pire ***n*** comparaisons.

Pour éviter cela, on propose deux améliorations qui combinées donnent quelque chose de très performant. La première est :

L'union pondérée

L'idée est la suivante: Au cours de l'ajout d'une arête $x --- y$, au lieu d'accrocher systématiquement l'arbre contenant y à la racine de celui contenant x , nous allons accrocher celui de plus petite taille (contenant le moins d'éléments) à la racine de celui de plus grande.

On appelle cela **l'union pondérée** de deux arbres.

Algorithm :

La difficulté est de trouver un moyen rapide (voire instantané) de connaître la taille d'un arbre à n'importe quel moment. L'astuce est la suivante : Pour représenter la forêt, nous allons, là aussi, utiliser le tableau ***pere***. La différence est que si un noeud $i \in [1, n]$ n'a pas de père, à ce moment là, on aura ***pere[i] = -taille*** avec ***taille*** le nombre d'éléments contenus dans l'arbre dont ***pere[i]*** est racine (*représentant*).

*Remarque : De la même manière, au départ toutes les classes d'équivalence étant réduites à des singletons, notre tableau ***pere*** verra tous ses éléments initialisés à -1.*

Ce qui donne pour **réunir** :

```
/* réunir(x, y, pere) accroche la racine de l'arbre de plus petite taille */
/* à la racine de celui de plus grande taille s'ils sont distincts */
Algorithme procédure réunir
Paramètres locaux
    entier      x, y
Paramètres globaux
```

```

    t_vectNbsEnt pere
Variables
    entier rx, ry
Début
    rx ← trouver(x, pere)
    ry ← trouver(y, pere)
    si rx <> ry alors
        si pere[rx] > pere[ry] alors
            pere[ry] ← pere[ry]+pere[rx]
            pere[rx] ← ry
        sinon
            pere[rx] ← pere[rx]+pere[ry]
            pere[ry] ← rx
        fin Si
    fin Si
Fin Algorithme procédure réunir

```

Remarque : Si les deux arbres font la même taille, on retombe sur le système de base, à savoir accroche du deuxième arbre à la racine du premier.

Reprendons l'exemple précédent de construction de graphe par ajouts successifs d'arêtes. Conservons la découpe en 4 phases et observons les résultats de l'application de la nouvelle procédure **réunir** (cf. Tableaux 5, 6, 7 et 8).

Tableau 5. Phase N°1.

Arêtes		Forêt obtenue																																																					
1 --- 6, 3 --- 2, 8 --- 7, 12 --- 15, 17 --- 16, 11 --- 10, 9 --- 8, 17 --- 18																																																							
Vecteur pere																																																							
<table border="1"> <thead> <tr> <th>sommet</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th></tr> </thead> <tbody> <tr> <th>Père</th><td>-2</td><td>3</td><td>-2</td><td>-1</td><td>-1</td><td>1</td><td>8</td><td>-3</td><td>8</td><td>11</td><td>-2</td><td>-2</td><td>-1</td><td>-1</td><td>12</td><td>17</td><td>-3</td><td>17</td></tr> </tbody> </table>																		sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Père	-2	3	-2	-1	-1	1	8	-3	8	11	-2	-2	-1	-1	12	17	-3	17
sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18																																					
Père	-2	3	-2	-1	-1	1	8	-3	8	11	-2	-2	-1	-1	12	17	-3	17																																					

Tableau 6. Phase N°2.

Arêtes		Forêt obtenue																																																					
11 --- 7, 1 --- 2, 4 --- 5, 13 --- 15																																																							
Vecteur pere																																																							
<table border="1"> <thead> <tr> <th>sommet</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th></tr> </thead> <tbody> <tr> <th>Père</th><td>-4</td><td>3</td><td>1</td><td>-2</td><td>4</td><td>1</td><td>8</td><td>-5</td><td>8</td><td>11</td><td>8</td><td>-3</td><td>12</td><td>-1</td><td>12</td><td>17</td><td>-3</td><td>17</td></tr> </tbody> </table>																		sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Père	-4	3	1	-2	4	1	8	-5	8	11	8	-3	12	-1	12	17	-3	17
sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18																																					
Père	-4	3	1	-2	4	1	8	-5	8	11	8	-3	12	-1	12	17	-3	17																																					

Tableau 7. Phase N°3.

Arêtes	Forêt obtenue																																						
10 --- 9, 15 --- 14, 4 --- 6, 13 --- 14																																							
Vecteur <i>pere</i>																																							
<table border="1"> <tr> <td>sommet</td> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td> </tr> <tr> <td>Père</td> <td>-6</td><td>3</td><td>1</td><td>1</td><td>4</td><td>1</td><td>8</td><td>-5</td><td>8</td><td>11</td><td>8</td><td>-4</td><td>12</td><td>12</td><td>12</td><td>17</td><td>-3</td><td>17</td> </tr> </table>		sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Père	-6	3	1	1	4	1	8	-5	8	11	8	-4	12	12	12	17	-3	17
sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18																					
Père	-6	3	1	1	4	1	8	-5	8	11	8	-4	12	12	12	17	-3	17																					

Tableau 8. Phase N°4.

Arêtes	Forêt obtenue																																						
16 --- 15, 3 --- 4, 5 --- 6, 18 --- 12																																							
Vecteur <i>pere</i>																																							
<table border="1"> <tr> <td>sommet</td> <td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td> </tr> <tr> <td>Père</td> <td>-6</td><td>3</td><td>1</td><td>1</td><td>4</td><td>1</td><td>8</td><td>-5</td><td>8</td><td>11</td><td>8</td><td>-7</td><td>12</td><td>12</td><td>12</td><td>17</td><td>12</td><td>17</td> </tr> </table>		sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Père	-6	3	1	1	4	1	8	-5	8	11	8	-7	12	12	12	17	12	17
sommet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18																					
Père	-6	3	1	1	4	1	8	-5	8	11	8	-7	12	12	12	17	12	17																					

La forêt obtenue maintenant à une profondeur maximale de 2, ce qui est compte tenu de la faible taille du graphe une amélioration notable.

Cela dit, nous pouvons rendre la méthode encore plus efficace en compactant encore plus les formes d'arbres obtenues. C'est la deuxième amélioration, on l'appelle :

La compression de données

L'idée est la suivante: A chaque appel de la fonction **trouver** pour un sommet *x*, on accroche directement à la racine de l'arbre contenant *x* tous les sommets (*x* compris) se trouvant sur le chemin allant de *x* à cette racine.

Pour cela nous allons utiliser une nouvelle fonction **trouver**, ce qui donne :

```
/* trouver(x, pere) retourne la racine rx de l'arbre contenant x */
/* et effectue la compression du chemin de x vers rx */ 
Algorithme fonction trouver : entier
Paramètres locaux
    entier x
Paramètres globaux
    t_vectNbsEnt pere
    entier rx          /* Vecteur d'ascendance */
Variables
    entier y
Début
    rx ← x
    tant que pere[rx] > 0 faire      /* recherche de rx */
        rx ← pere[rx]
    fin tant que
    tant que pere[x] <> rx faire      /* compression du chemin de x vers rx */
```

```

y ← x
x ← pere[x]
pere[y] ← rx
fin tant que
retourne(rx)
fin pour
Fin Algorithme fonction trouver

```

Si l'on combine ces deux améliorations on obtient ce que l'on appelle : **L'union compressée et pondérée**.

En conservant l'exemple précédent de construction de graphe par ajouts successifs d'arêtes, nous observons les résultats suivants : Les deux premières phases sont identiques à la précédente méthode, la différence intervient à partir de la troisième phase (cf. Tableaux 9 et 10).

Tableau 9. Phase N°3.

Arêtes	Forêt obtenue			
10 --- 9, 15 --- 14, 4 --- 6, 13 --- 14	1 6 3 2 4 5 8 7 9 11 10 12 15 13 14 17 16 18			
Vecteur pere				
sommet	1	2	3	4
Père	-6	3	1	1
	4	1	8	-5
	1	8	8	8
				-4
				12
				12
				12
				17
				-3
				17

Tableau 10. Phase N°4.

Arêtes	Forêt obtenue			
16 --- 15, 3 --- 4, 5 --- 6, 18 --- 12	1 6 3 4 5 2 8 7 9 11 10 12 15 13 14 17 18 16			
Vecteur pere				
sommet	1	2	3	4
Père	-6	3	1	1
	1	1	1	1
	8	-5	8	8
			8	8
				-7
				12
				12
				12
				17
				12
				12

On constate quelques variations comme le fait que le sommet 10 est accroché directement au 8. En effet, l'ajout de l'arête **10 --- 9** ne modifie pas l'ensemble des classes d'équivalence (composantes connexes). Il ne se passait donc rien avec la première version de **trouver**. En revanche, la deuxième version (qui compresse) en profite quand même pour modifier la forêt. La hauteur de cette dernière à toujours un maximum à 2, mais sa profondeur moyenne a baissée.

Sur cet exemple, ce n'est pas forcément très probant, mais si maintenant arrivé à la phase 4, nous ajoutons successivement les arêtes **2 --- 16** et **10 --- 18**, nous obtiendrions, suivant les trois méthodes, les forêts représentées figure 2.

Figure 2. Résultats des 3 méthodes après ajouts des arêtes 2 --- 16 et 10 --- 18.

Méthode de base	Union pondérée	Union pondérée et compression

Lorsque l'on construit les classes d'équivalence d'un ensemble de n éléments par *unions pondérées et compressions de données* successives, trouver la classe d'équivalence d'un élément, ou tester si deux éléments appartiennent à la même classe, demande une complexité quasi-constante au pire.

Pour toute suite d'opérations qui sont des *unions pondérées* et des *compressions de données*, si l'on a m opérations **trouver**, la complexité en temps est au pire $\Theta(n + m \cdot \alpha(m + n, n))$, avec $\alpha(u, v)$ une fonction qui croît très lentement, beaucoup plus lentement qu'une fonction logarithmique (elle est définie comme l'inverse de la fonction d'Ackermann). Dans la pratique, on peut considérer $\alpha(u, v) \leq 4$.

Pour conclure, on peut tester une fois que la forêt est construite si le graphe est connexe au pire en $\Theta(n)$, avec n le nombre de sommets (**trouver** est appelée pour chaque sommet).

Remarque : C'est un peu plus couteux que le parcours en profondeur, mais il n'y a pas besoin de stocker les arêtes du graphe.

Composantes fortement connexes

Nous allons maintenant envisager deux méthodes permettant de déterminer les composantes fortement connexes (*cfc*) d'un graphe orienté, et donc de déterminer s'il est fortement connexe. C'est *l'algorithme de Tarjan*.

- La première méthode est basée sur un double parcours profondeur avec inversion de graphe,
- la deuxième utilise un seul parcours profondeur en s'aidant d'une pile.

Algorithme de Tarjan (double parcours et inversion de graphe)

Principe :

Supposons un graphe G orienté. Notre algorithme **composante fortement connexe** va s'appuyer sur deux procédures de parcours : **parcprofsoff** et **parcprofinv** qui vont respectivement réaliser les choses suivantes :

parcprofsoff : effectue le parcours en profondeur du graphe G tout en construisant une liste L des sommets rencontrés en ordre suffixe. Comme de coutume, On commence arbitrairement sur le sommet 1 et on traite les autres en ordre croissant de succession.

parcprofinv : effectue le parcours en profondeur du graphe inverse G^{-1} et construit le vecteur *cfc* associant à chaque sommet son numéro de composante fortement connexe. Pour cela, on utilise la liste L de sommets précédemment créée en ordre décroissant. C'est à dire que l'on commence par le dernier élément de L (le sommet de plus grande valeur suffixe dans le parcours profondeur de G) et l'on remonte sur les précédents non marqués (lors des retours d'appels).

Remarques :

- Les numéros suffixes (ordre pour G) des racines des arborescences de la forêt couvrante de G^{-1} sont donc en ordre décroissant.
- Les sous-graphes engendrés par les sommets des arborescences de la forêt couvrante de G^{-1} ainsi obtenue, forment les composantes fortement connexes de G .

Algorithme :

Pour les algorithmes qui suivent, les types de données employés sont les suivants :

Constantes

```
Nbs = ... /* Nombre de sommets du graphe */
```

Types

```
t_vectNbsEnt = Nbs entier /* Vecteur d'entiers */
t_vectNbsBool = Nbs booléen /* Vecteur de booléens */
```

Ce qui donne pour l'algorithme principal **composante fortement connexe** :

```
/* Après l'exécution cfc[i] contiendra le numéro de la composante fortement connexe */
/* à laquelle appartient le sommet i */
Algorithme procédure composantefortementconnexe
Paramètres locaux
    graphe g
Paramètres globaux
    t_vectNbsEnt cfc /* Vecteur de composantes fortement connexes */
Variables
    t_vectNbsEnt lsom /* Liste des sommets */
    t_vectNbsBool marque /* Vecteur de marquage de sommets */
    entier i,k
Début
    /* Initialisation */
    pour i ← 1 jusqu'à Nbs faire
        cfc[i] ← 0
        marque[i] ← Faux
        lsom[i] ← 0
    fin pour
    k ← 0
    /* K utilisé comme compteur suffixe */
    /* Construction de la liste lsom des sommets de g rencontrés en ordre suffixe */
    pour i ← 1 jusqu'à Nbs faire
        si non(marque[i]) alors
            parcprofsuff(i,g,marque,lsom,k)
        fin si
    fin pour
    /* Ici devrait avoir lieu l'inversion du graphe en vue de son parcours */
    /* Remise à Faux de Marque */
    pour i ← 1 jusqu'à Nbs faire
        marque[i] ← Faux
    fin pour
    /* K utilisé comme compteur de cfc détectées */
    k ← 0
    pour i ← Nbs jusqu'à 1 décroissant faire
        si non(marque[ième(l,i)]) alors
            k ← k + 1
            parcprofinv(ième(l,i),g,marque,cfc,k)
        fin si
    fin pour
Fin Algorithme procédure composantefortementconnexe
```

Pour la procédure **parcprofsuff** :

```
/* parcprofsuff(x,g,m,l,k) où x est le sommet racine, k est le compteur de sommets */
/* m le tableau de marque et l la liste de sommets rencontrés en ordre préfixe */
Algorithme procédure Parcprofsuff
Paramètres locaux
    entier x
    graphe g
Paramètres globaux
    t_vectNbsBool m /* Vecteur de marquage de sommets */
    t_vectNbsEnt l /* Liste des sommets */
    entier k
Variables
    entier i,y /* y est un sommet */
Début
    m[x] ← True
    pour i ← 1 jusqu'à d0+(x,g) faire
        y ← ième-succ(i,x,g)
        si non(m[y]) alors
            parcprofsuff(y,g,m,l,k)
        fin si
    fin pour
    /* Traitement suffixe du sommet */
    k ← k + 1
    l ← insérer(l,k,x)
Fin Algorithme procédure parcprofsuff
```

Et pour la procédure **parcprofinv**:

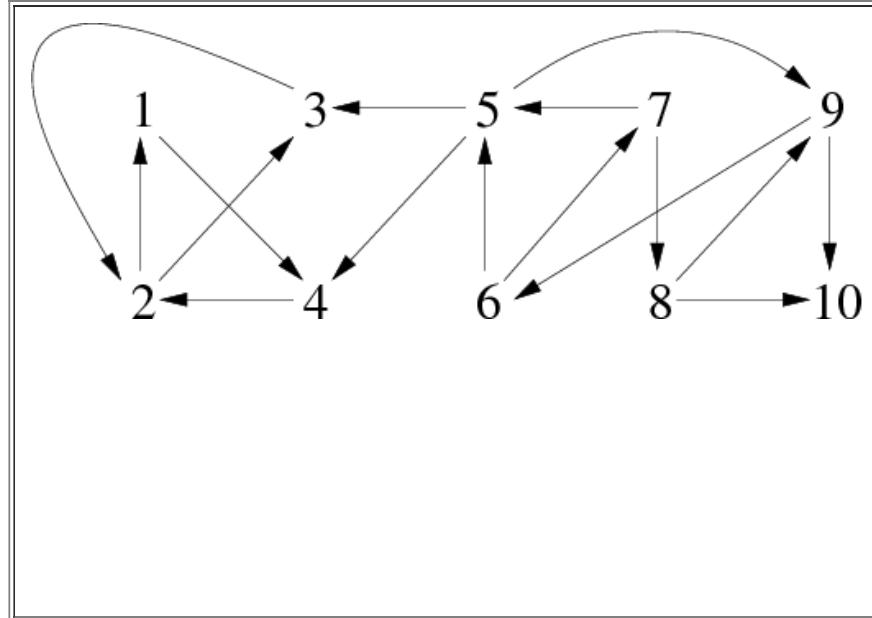
```

/* parcprofinv(x,g,m,cfc,k) parcours en profondeur le graphe inverse */
/* et remplit le tableau cfc par la valeur k; x est un sommet */
Algorithme procédure parcprofinv
Paramètres locaux
    entier x
    graphe g
Paramètres globaux
    t_vectNbsBool m           /* Vecteur de marquage de sommets */
    t_vectNbsEnt cfc          /* vecteur de composantes fortement connexes */
    entier k
Variables
    entier i,y                /* y est un sommet */
Début
    m[x] ← True
    cfc[x] ← k
    /* L'astuce consiste à ne pas inverser le graphe, mais à le parcourir via les prédecesseurs */
    pour i ← 1 jusqu'à d°-(x,g) faire
        y ← ième-pred(i,x,g)
        si non(m[y]) alors
            parcprofinv(y,g,m,cfc,k)
        fin si
    fin pour
Fin Algorithme procédure parcprofinv

```

Prenons pour exemple le graphe G de la figure 3:

Figure 3. Graphe orienté G .



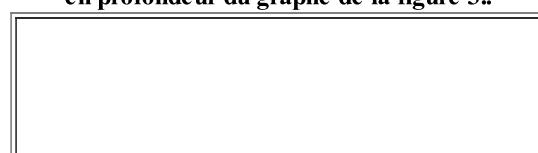
Après exécution de l'algorithme **parcprofssuff**, nous obtenons la liste ***Isom*** suivante (représentée sous la forme d'un vecteur pour en faciliter la lecture) :

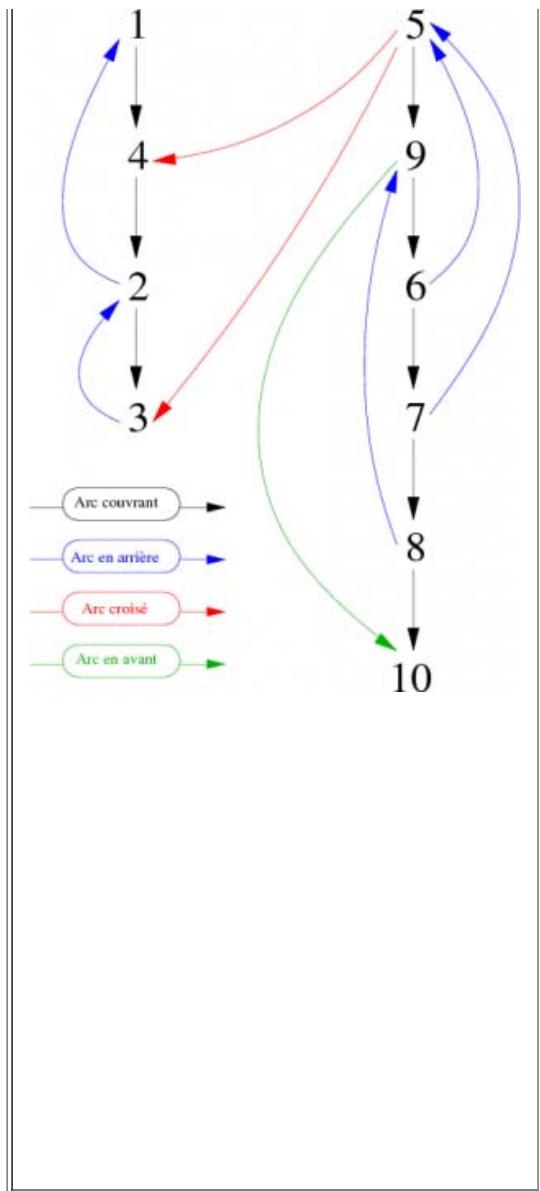
Tableau 11. *Isom* – Liste des sommets rencontrés en ordre suffixe.

Ordre suffixe (indice)	1	2	3	4	5	6	7	8	9	10
Sommets	3	2	4	1	10	8	7	6	9	5

L'exécution de ce parcours aurait pu être représenté graphiquement comme sur la figure 4. Si l'on conserve uniquement les arcs couvrants (en noir), nous visualisons la forêt couvrante associée au parcours en profondeur du graphe G , ce qui permet de facilement connaître l'ordre suffixe de rencontre des sommets du graphe. *Là encore, le parcours est effectué à partir du sommet 1 et en prenant les successeurs en ordre croissant.*

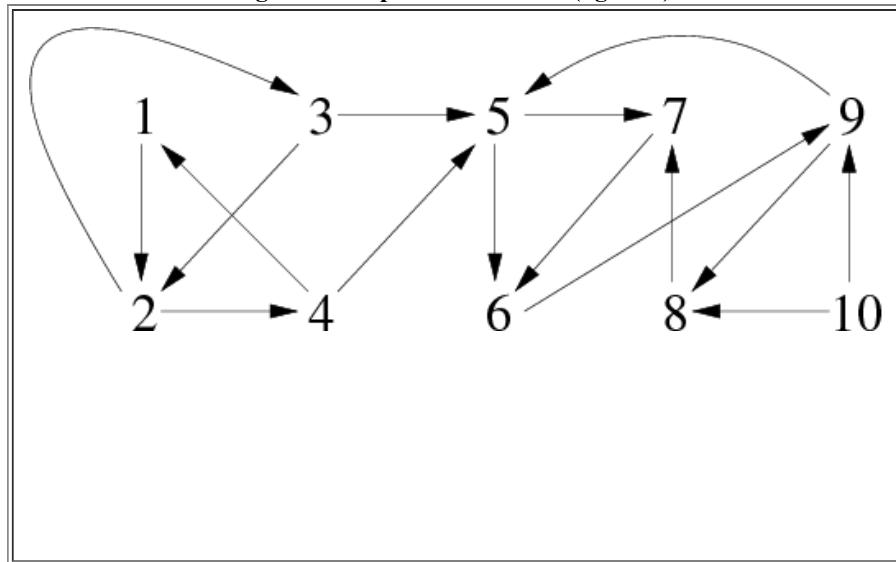
Figure 4. Forêt couvrante associée au parcours en profondeur du graphe de la figure 3..





Prenons maintenant le graphe G^{-1} (inverse de G) présenté figure 5:

Figure 5. Graphe inverse de G (figure 3).



Après exécution de l'algorithme **Parcprofinv**, nous obtenons le vecteur cfc suivant (tableau 12) :

Tableau 12. cfc -- Composantes fortement connexes de chaque

Sommets (indice)	1	2	3	4	5	6	7	8	9	10
N° de composante	3	3	3	3	1	1	1	1	1	2

Remarques :

- L'ordre croissant des successeurs est conservé, en revanche le parcours a commencé à partir du sommet 5 puisqu'il est celui de plus grand N° suffixe dans lsom.
- A la fin de l'exécution, le compteur k de composante vaut 3 (nombre de composantes fortement connexes du graphe).

Là aussi, l'exécution de ce parcours aurait pu être représentée graphiquement comme sur la figure 6. Si l'on conserve uniquement les arcs couvrants, il devient alors aisé de repérer les arborescences correspondant aux trois composantes fortement connexes de ce graphe que l'on peut voir représentées sur la figure 7.

Figure 6. Forêt couvrante associée au parcours en profondeur du graphe inverse de G (figure 5) en utilisant lsom en ordre inverse.

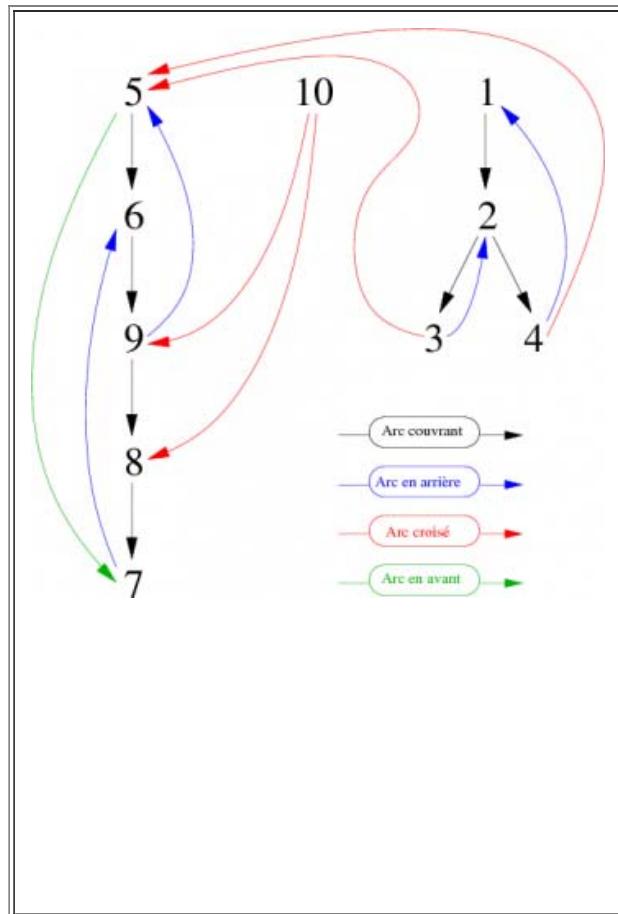
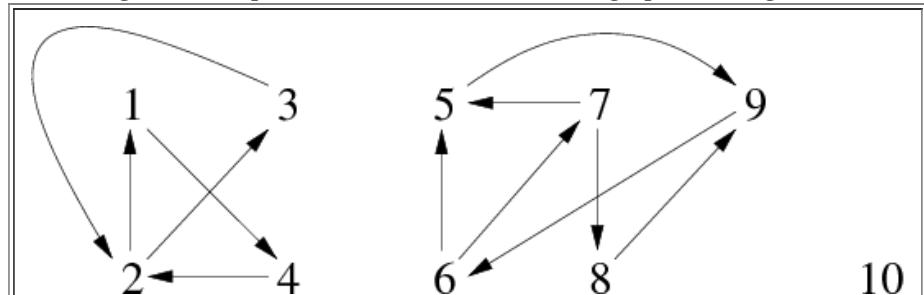


Figure 7. Composantes fortement connexes du graphe de la figure 3.



Algorithme de Tarjan (parcours simple et pile)

Comme nous l'avons dit précédemment, **l'algorithme de Tarjan** peut être optimisé pour déterminer les composantes fortement connexes d'un graphe en effectuant un seul parcours en profondeur (*Il en a d'ailleurs la complexité.*)

Cet algorithme exploite les propriétés des **Forêts couvrantes** associées au parcours en profondeur de graphes suivantes :

- Propriétés élémentaires:

1. Soient x et y deux sommets d'une même composante fortement connexe, z un sommet d'un chemin de x vers y , alors z appartient à la même composante fortement connexe que x et y .
2. Soient x et y deux sommets tels qu'il y ait un chemin de x vers y . Si dans un parcours en profondeur, on marque x alors que y n'est pas encore marqué, y sera dans la même arborescence que x .

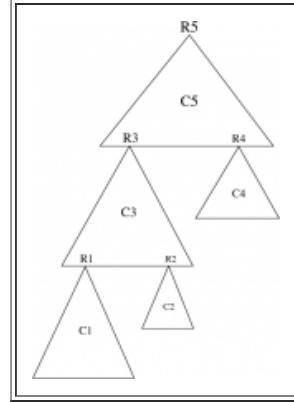
- Propriétés:

1. Soient $G = \langle S, A \rangle$ un graphe orienté et $C_i = \langle S_i, A_i \rangle$ une composante fortement connexe de G avec $S_i \subseteq S$ et $A_i \subseteq A$. Soit B_i l'ensemble des arcs couvrants de A_i dans la forêt couvrante associée au parcours en profondeur de G . Alors $G_i = \langle S_i, B_i \rangle$ est une arborescence.

Remarque: On appelle racine de la composante fortement connexe C_i , la racine r_i de l'arborescence G_i associée à C_i . C'est le 1^{er} sommet de G_i rencontré en ordre préfixe.

2. Quel que soit i , les sommets de la composante fortement connexe C_i sont les descendants de r_i dans la forêt couvrante associée au parcours en profondeur de G qui ne sont pas dans les composantes fortement connexes C_1, \dots, C_{i-1} (cf figure 8).

Figure 8. Composantes fortement connexes et leur racine.



Principe de l'algorithme:

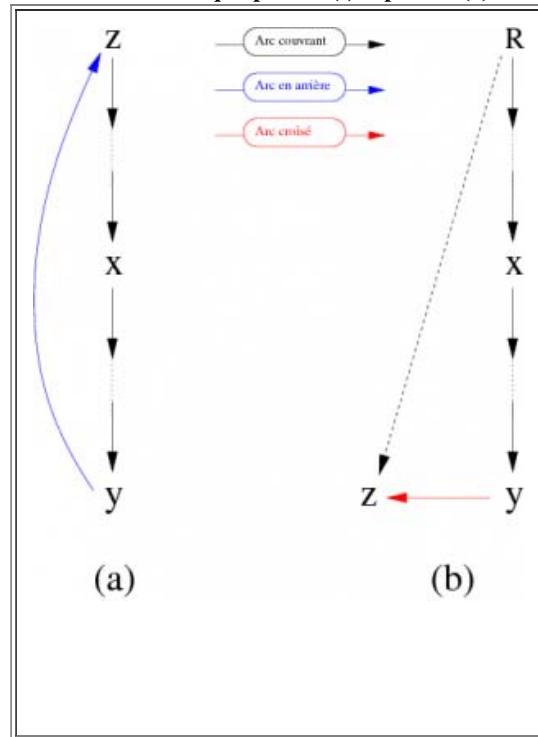
Les racines r_i étant considérées en ordre suffixe de rencontre, on obtient les sommets de la composante fortement connexe C_i en ne prenant que les sommets de l'arborescence de racine r_i qui n'ont pas été déjà pris en compte dans les précédentes Composantes.

On introduit alors deux fonctions qui vont nous permettre, à l'aide des propriétés suivantes, d'exploiter cette remarque:

- **prefixe(x):** qui retourne pour chaque sommet x du graphe G son numéro d'ordre préfixe dans le parcours profondeur de G .
- **retour(x):** qui retourne pour chaque sommet x la valeur préfixe d'un sommet de la composante fortement connexe de x rencontré avant lui lors du parcours profondeur s'il existe ou alors la valeur préfixe de x lui-même.

Appelons ce sommet z , pour qu'il existe (*membre de la même cfc que x et tel que $\text{prefixe}(z) < \text{prefixe}(x)$*) un tel sommet doit être (graphiquement) placé au dessus et/ou à gauche du sommet x . Cela sous-entend qu'en partant de x il faut suivre un chemin commençant par une suite (éventuellement vide) d'arcs courants continuée par un arc en arrière ou un arc croisé. Graphiquement: peut-on trouver un des deux cas présentés figure 9?

Figure 9.
Sommet z tel que $\text{prefixe}(z) < \text{prefixe}(x)$.



- Dans le cas (a), il est clair que, lors du parcours profondeur de la forêt couvrante, le sommet z sera rencontré avant le sommet x et que sa valeur préfixe sera donc inférieure à celle du sommet x .
- Dans le cas (b), c'est un petit peu plus compliqué, il faut partir du principe qu'il est vrai que $\text{prefixe}(z) < \text{prefixe}(x)$. A partir de ce moment-là:
 - si les deux sommets sont dans une même cfc, x n'est pas la racine de cette composante,
 - sinon c'est que z appartient à une autre cfc (que celle de x) rencontrée avant. Dans ce cas, la racine Rz de la cfc à laquelle appartient z possède une numérotation suffixe inférieure à la racine Rx de la cfc à laquelle appartient x . Le sommet z bien que successeur de x dans le parcours profondeur de la forêt a déjà été affecté à une précédente cfc.

On peut alors donner une définition de la fonction **retour**:

```
retour(x) = min { prefixe(x), prefixe(z), retour(y) }
```

Où le retour de x est calculé lors de sa dernière rencontre (ordre suffixe) sur tousles sommets z et y tels que $x \rightarrow z$ est un arc en arrière (figure 9. (a)) ou un arc croisé (figure 9.(b)) et $x \rightarrow y$ est un arc couvant.

La valeur minimale rentronnée par *retour(x)* pour chaque sommet x sera:

- $\text{prefixe}(x)$ s'il n'existe pas de chemin de x à z avec un sommet z tel que $\text{prefixe}(z) < \text{prefixe}(x)$.
- $\text{prefixe}(z)$ si le chemin de x à z ne comprend pas d'arcs courvants.
- $\text{retour}(y)$ s'il y a au moins un arc couvant $x \rightarrow y$ dans le chemin. A ce moment là, le sommet x peut atteindre ce que peut atteindre le sommet y .

Algorithme:

L'évaluation de la valeur de retour pour chaque sommet x se fera récursivement à l'aide d'un parcours profondeur. Le sommet est empilé et sa valeur de retour est initialisé à sa propre valeur préfixe. Cette valeur est éventuellement mise à jour lors des rencontres des successeurs de x (par le biais de leur valeur préfixe ou de leur valeur de retour). A la dernière rencontre du sommet x (*ordre suffixe*), sa valeur de retour est comparée

à sa valeur préfixe. En cas d'égalité, nous sommes en présence d'une nouvelle racine de cfc. En effet ce sommet ne peut atteindre aucun autre sommet du graphe qui se trouverait avant lui lors du parcours en profondeur de ce graphe. Il suffit alors de dépiler tous les sommets encore dans la pile (x compris) pour obtenir les sommets de cette nouvelle cfc.

L'algorithme utilise un certains nombre de tableaux permettant entre autres de mémoriser les valeurs préfixes, les valeurs de retour ou l'appartenance à une cfc de chaque sommet du graphe parcouru.

Pour les algorithmes qui suivent, les types de données employés sont les suivants :

```
Constantes
Nbs = ... /* Nombre de sommets du graphe */

Types
t_vectNbsEnt = Nbs entier /* Vecteur d'entiers */
```

Ce qui donne pour l'algorithme principal **composante fortement connexe** :

```
/* Après l'exécution cfc[i] contiendra le numéro de la composante fortement connexe */
/* à laquelle appartient le sommet i, et cptcfc le nombre de cfc détectées */
Algorithme procédure composantefortementconnexe
Paramètres locaux
graphe g
Paramètres globaux
t_vectNbsEnt cfc /* Vecteur de composantes fortement connexes */
entier cptcfc /* Compteur de composantes fortement connexes */
Variables
t_vectNbsEnt pref, ret /* Vecteurs de valeur préfixe et retour des sommets */
entier x, cpt /* x:sommet, cpt:compteur préfixe */
pile p /* Pile de sommets */
Début
cpt ← 0
p ← pilevide
/* Initialisation */
pour x ← 1 jusqu'à Nbs faire
    pref[x] ← 0
fin pour
/* Détermination des cfc */
cptcfc ← 0
pour x ← 1 jusqu'à Nbs faire
    si pref[x]=0 alors
        parcfc(x, g, cpt, cptcfc, p, pref, ret, cfc)
    fin si
fin pour
Fin Algorithme procédure composantefortementconnexe
```

Pour la procédure **parcfc** :

```
/* parcfc(x, g, cpt, cptcfc, p, pref, ret, cfc) où x est le sommet racine, cpt est le compteur préfixe de sommets */
/* cptcfc le compteur de cfc, p la pile de sommets, pref le tableau de valeur préfixe(marque) des sommets, */
/* ret le tableau de valeur de retour et cfc' le tableau tel que de cfc[i] donne la cfc de i */
Algorithme procédure Parcfc
Paramètres locaux
entier x
graphe g
Paramètres globaux
entier cpt, cptcfc
pile p
t_vectNbsEnt pref, ret, cfc
Variables
entier i, y, min /* y est un sommet, min le retour temporaire de x */
Début
/* Initialisation et marquage */
cpt ← cpt + 1
pref[x] ← cpt
min ← cpt
p ← empiler(x,p)
pour i ← 1 jusqu'à d0+(x,g) faire
    y ← ième-succ(i,x,g)
    si pref[y]=0 alors /* y non marqué, arc couvrant */
        parcfc(x, g, cpt, cptcfc, p, pref, ret, cfc)
        si ret[y]<min alors
            min ← ret[y]
        fin si
    sinon /* y déjà marqué, arc en arrière ou croisé */
        si pref[y]<min alors
            min ← pref[y]
        fin si
    fin si
fin pour
```

```

ret[x] ← min
si ret[x]=pref[x] alors      /* nouvelle cfc dont x est racine */
    cptcfc ← cptcfc + 1
    tant que sommet(p) <> x faire
        y ← sommet(p)
        cfc[y] ← cptcfc
        pref[y] ← N + 1
        p ← depiler(p)
    fin tant que
    cfc[x] ← cptcfc
    pref[x] ← N + 1
    p ← depiler(p)
fin si
Fin Algorithme procédure parcfc

```

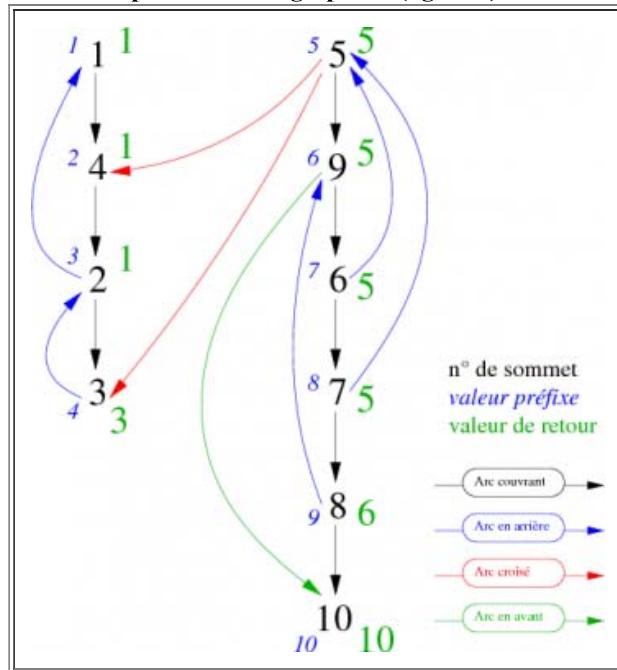
Après exécution de l'algorithme **composante fortement connexe**, nous obtenons le vecteur *cfc* suivant (tableau 13) :

Tableau 13. *cfc* -- Composantes fortement connexes de chaque

sommets.		1	2	3	4	5	6	7	8	9	10
Sommets (indice)											
N° de composante	3	3	3	3	1	1	1	1	1	2	

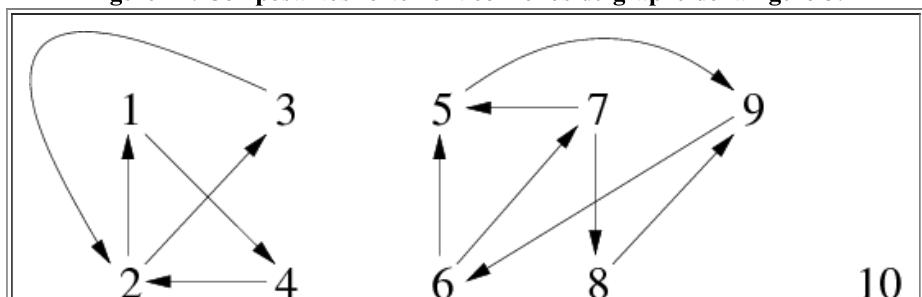
Nous pouvons visualiser le parcours et le mode de calcul des valeurs de retour des sommets du graphe à l'aide de la forêt couvrante représentée en figure 10. On constate trois composantes fortement connexes déterminées par leur racine respective dont les valeurs de préfixe et de retour sont identiques (sommets 1, 5 et 10).

Figure 10. Forêt couvrante associée au parcours en profondeur du graphe *G* (figure 3).



Les composantes fortement connexes sont alors les suivantes (cf. figure 11):

Figure 11. Composantes fortement connexes du graphe de la figure 3.



Biconnexité

La biconnexité (2-connexe) est une propriété des réseaux connexes. Cette propriété vérifie le fait que si un sommet du graphe (n'importe lequel) disparaît, tous les autres restent en connexion.

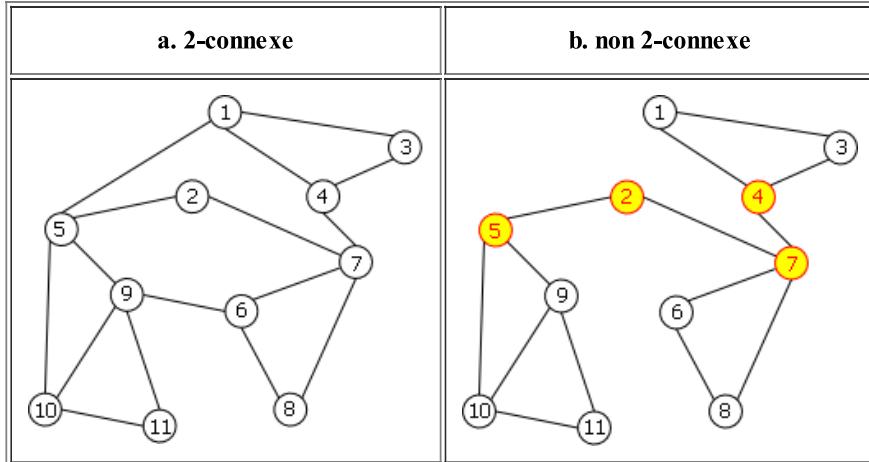
Définitions:

- Soit G un graphe non orienté connexe, on dit que G est biconnexe s'il possède au moins 3 sommets et si quel que soit s un sommet de G , le graphe $G - \{s\}$ est toujours connexe.
- Soit s un sommet de G , on dit que s est un **point d'articulation** de G (*cut-point, articulation point*) si le graphe $G - \{s\}$ n'est plus connexe.
- Soit a une arête de G , on dit que a est un **isthme** de G (*cut-edge*) si le graphe $G - \{a\}$ n'est plus connexe.

Remarque: Un graphe biconnexe ne possède pas de point d'articulation.

On peut voir sur l'exemple de la figure 12.b un graphe non orienté connexe qui n'est pas biconnexe. Il possède 4 points d'articulation: 2, 4, 5 et 7. En effet, si vous enlevez un de ces sommets, vous séparez les sommets restants en deux groupes distincts n'étant plus connectés l'un à l'autre.

Figure 12. Graphes connexes.



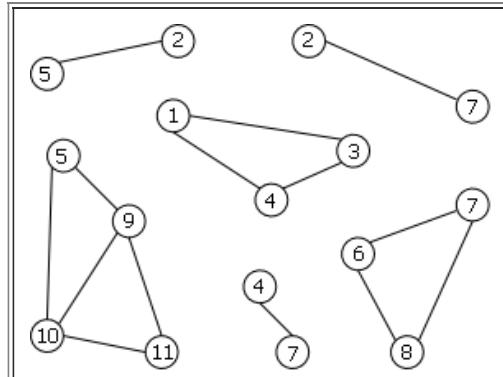
La 2-connexité est utilisé, entre autres choses, pour tester la vulnérabilité des réseaux. En cas de conflit il suffit de détruire les points d'articulation pour séparer/fragiliser tout un système de communication.

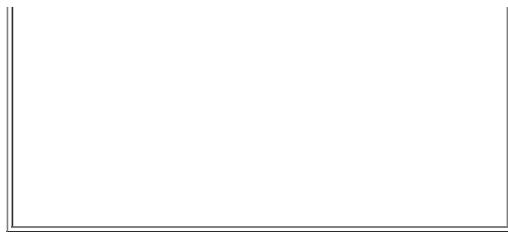
Malheureusement ou heureusement (tout dépend ;)), tous les graphes connexes ne sont pas biconnexes. on s'intéresse alors aux composantes biconnexes du graphe. pour définir les composantes biconnexes, on introduit la notion de **bloc**.

- un **bloc** est soit une arête, soit un graphe 2-connexe.
- On appelle **C2C (composante 2-connexe)** d'un graphe G tout bloc maximal de G . C'est à dire tout bloc de G qui n'est pas strictement inclus dans un autre bloc de G .

La figure 13 montre l'ensemble des composantes 2-connexes du graphe de la figure 12b.

Figure 13. C2C du graphe de la figure 12b.





On peut constater que:

- les C2C forment une partition de l'ensemble des arêtes du graphe,
- les C2C sont soit disjointes, soit ont en commun un point d'articulation.

Algorithm:

On se propose de réaliser un algorithme d'extraction des points d'articulations d'un graphe non orienté connexe. Il pourra être modifié pour déterminer les composantes 2-connexes d'un graphe. Ce dernier effectue un seul parcours en profondeur et examine les propriétés des sommets de la **forêt couvrante** associée au parcours en profondeur du graphe. Cette forêt est réduite à une arborescence, dans la mesure où le graphe est connexe. Elle ne comporte d'ailleurs que deux types d'arcs : couvrant et en arrière le graphe étant non orienté.

L'idée est la suivante : Lorsque l'on supprime un sommet x du graphe G , on coupe la chaîne qui existait entre ses descendants et ses ascendants. La question est : A-t-on perdu pour autant la connexité entre ces derniers ?

Autrement dit : Existe-t-il une chaîne entre les descendants et les ancêtres de x dans le graphe $G - \{x\}$?

Si une telle chaîne existe, elle sera formée d'arcs couvrants en dessous de x et d'un arc en arrière remontant au dessus de x . Nous allons tester cette existence à l'aide d'une fonction **plus haut(x)** qui donnera pour chaque sommet x le numéro d'ordre préfixe du sommet le plus haut de l'arbre couvrant. Comment : En suivant un chemin, de longueur éventuellement nulle (si le sommet ne peut pas atteindre de sommet placé plus haut que lui), composé d'arcs couvrants suivi éventuellement d'un arc en arrière.

La numérotation préfixe d'un sommet x est obtenu par la fonction **prefixe(x)**.

La fonction **plus haut(x)** est définie récursivement de la manière suivante :

```
plus haut(x) = min { prefixe(x), prefixe(z), plus haut(y) }
```

Où le minimum (**plus haut(x)**) est calculé lors de sa dernière rencontre (ordre suffixe) sur tous les sommets z tels que $x \rightarrow z$ est un arc en arrière et sur tous les sommets y tel que $x \rightarrow y$ est un arc couvrant.

Cette définition rappelle celle de la fonction **retour(x)** de l'algorithme de Tarjan. La différence réside essentiellement dans le fait qu'il n'existe pas d'arcs croisés (le graphe étant non orienté).

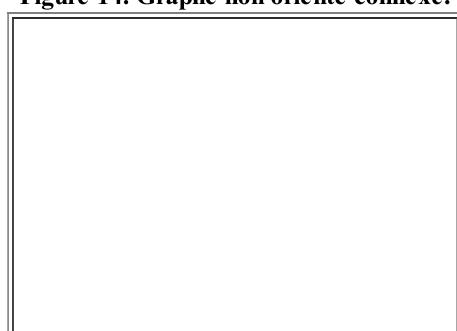
Principe de l'algorithme:

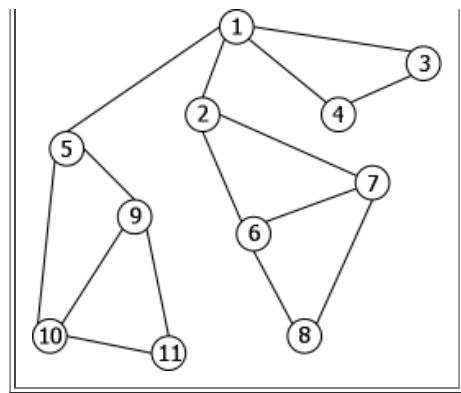
L'évaluation de la valeur **plus haut** pour chaque sommet x se fera récursivement à l'aide d'un parcours profondeur:

- Les valeurs **préfixe(x)** sont obtenues lors de la première rencontre des sommets x lors d'un parcours profondeur.
- les valeurs **plus haut(x)** ne sont obtenues que lors de la dernière rencontre du sommet x (en ordre suffixe) puisqu'elles sont déterminées à l'aide des valeurs **plus haut** et **préfixe** des successeurs de x .

Prenons le graphe non orienté connexe de la figure 14:

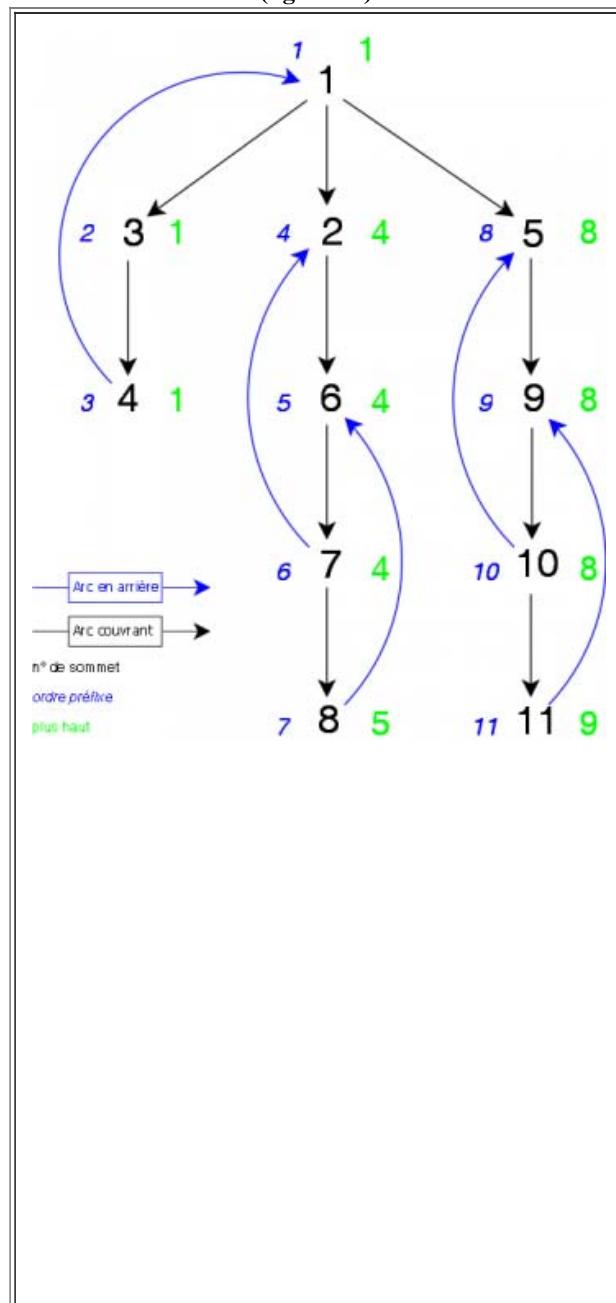
Figure 14. Graphe non orienté connexe.





L'arbre couvrant associé au parcours profond de ce graphe est celui de la figure 15:

Figure 15. Arbre couvrant associé au parcours du graphe (figure 14).



En analysant l'arbre obtenu ainsi que les valeurs de **préfixe** et **plus haut** de chaque sommet, on en déduit qu'un sommet est un point d'articulation s'il satisfait une des deux conditions suivantes :

- x est le sommet racine de l'arbre. S'il a plus d'un fils alors c'est un point d'articulation. En effet, si vous le supprimez, il n'y aura plus de chaîne entre les sommets des différentes branches descendantes de x .
- x n'est pas le sommet racine de l'arbre. S'il existe un sommet y tel que $x \rightarrow y$ est un arc couvrant et que **plus haut(y)** est supérieur ou égal à **préfixe(x)** alors x est un point d'articulation. En effet, cela veut dire que l'on ne peut pas atteindre d'ascendants de x depuis y .

L'algorithme utilise des tableaux permettant entre autres de mémoriser les valeurs préfixes et les valeurs de plus haut. Le tableau de préfixe servira à marquer les sommets déjà rencontré et un vecteur père permettra de savoir si le lien entre deux sommets est un arc retour ou simplement la bidirection père-fils.

Pour les algorithmes qui suivent, les types de données employés sont les suivants :

```
Constantes
Nbs = ... /* Nombre de sommets du graphe */

Types
t_vectNbsEnt = Nbs entier /* Vecteur d'entiers */
```

La particularité de ce parcours est de ne pas déclencher du sommet **1** au sommet **Nbs**, mais du premier au dernier successeur de la racine du graphe (sommel de départ) pour pouvoir déterminer si celle-ci est un point d'articulation ou non. Ce qui donne pour l'algorithme principal **pointdarticulation** :

```
/* Construit la liste l des points d'articulation du graphe non orienté g */
/* de Nbs sommets supposé connexe , avec Nbs >= 3 */
Algorithme procédure pointdarticulation
Paramètres locaux
graphe g
Paramètres globaux
liste l /* Liste des points d'articulation */
entier nbpa /* Nombre de points d'articulation */
Variables
t_vectNbsEnt pref, ph, pere /* Vecteurs de valeur préfixe et plus haut des sommets ainsi que le père */
entier r, x, appel, cpt, i /* r et x:sommets, appel:nombre de fils de r, cpt:compteur préfixe, i:indice */
Début
/* initialisation */
pour i ← 1 jusqu'à Nbs faire
  pref[i] ← 0
  plus haut[i] ← 0
  pere[i] ← 0
fin pour
cpt ← 1, l ← listevide
nbpa ← 0, r ← 1
appel ← 0
pref[r] ← cpt
plus haut[r] ← cpt
/* parcours des fils de la racine r */
cpt ← cpt + 1
pour i ← 1 jusqu'à d°(r,g) faire
  x ← ième-succ(i,r,g)
  si pref[x]=0 alors
    pere[x] ← r
    parc2c(x, g, pref, plus haut, pere, cpt, l, nbpa)
    appel ← appel + 1
  fin si
fin pour
/* Détermination de la racine comme point d'articulation */
si appel > 1 alors
  nbpa ← nbpa + 1
  l ← insérer(l, nbpa, r)
fin si
Fin Algorithme procédure pointdarticulation
```

Et pour la procédure **parc2c** :

```
Algorithme procédure Parc2c
Paramètres locaux
entier x
graphe g
Paramètres globaux
t_vectNbsEnt pref, plus haut, pere
entier cpt
liste l
entier nbpa
Variables
entier i, y, ph /* y est un sommet, ph le plus haut temporaire de x */
booléen detecte /* permet de ne pas traiter plusieurs fois un même PA */
Début
```

```

/* Initialisation et marquage */
pref[x] ← cpt
ph ← cpt
cpt ← cpt + 1
detecte ← faux
pour i ← 1 jusqu'à d°(x,g) faire
    y ← ième-succ(i,x,g)
    si pref[y]=0 alors /* y non marqué, arc couvrant */
        pere[y] ← x
        parc2c(y, g, pref, plushaut, pere, cpt, l, nbpa)
        si plushaut[y]<ph alors
            ph ← plushaut[y]
            /* Détermination de x comme point d'articulation */
            si non(detecte) et plushaut[y]>=pref[x] alors
                detecte ← vrai
                nbpa ← nbpa + 1
                l ← insérer(l, nbpa, x)
            fin si
        fin si
    sinon /* y déjà marqué, arc en arrière */
        si y<>pere[x] et pref[y]<ph alors
            ph ← pref[y]
        fin si
    fin si
fin pour
plushaut[x] ← ph
Fin Algorithmme procédure parc2c

```

Après exécution de l'algorithme **pointdarticulation**, si la liste *l* est vide, cela signifie que le graphe *g'* est 2-connexe. A l'inverse, *nbpa* nous permet avant de traiter les points d'articulations contenus dans *l* d'en connaître le nombre.

La complexité de cet algorithme est celle d'un parcours en profondeur.

extrapolation à la k-connexité

(Christophe "krisboul" Boullay)

Récupérée de « http://algo.infoprepa.epita.fr/index.php?title=Epita:Algo:Cours:Info-Spe:les_connexit%C3%A9s&oldid=2739 »

- Dernière modification de cette page le 29 octobre 2013 à 20:51.