Biconnexité

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/02/

1 Qu'est qu'un graphe, exactement ?

Dans cette séance un graphe est défini comme une paire (X, A) où X est un ensemble (fini) de sommets s_i et A un ensemble d'arcs notés $s_i \to s_j$ dans le cas des graphes orientés et $s_i \leftrightarrow s_j$ dans le cas des graphes non-orientés. Dans ce dernier cas $s_i \leftrightarrow s_j$ et $s_j \leftrightarrow s_i$ désignent le même arc.

1. Un chemin, est une suite de sommets et d'arcs $s_1 \to s_2 \to \ldots \to s_k$. Un chemin est simple quand tous ses sommets sont deux à deux distincts.

   Monter que si il existe un chemin de s à s' (on suppose $s \neq s'$), alors il existe un chemin simple de s à s'.

2. Un cycle est un chemin de s à s dont tous les arcs sont deux à deux distincts et dont tous les sommets sont distincts, sauf pour s qui apparaît exactement deux fois.

   Montrer, dans un graphe orienté, que si il existe un chemin de s à lui même, alors il existe un cycle. Qu'en est-il pour les graphes non-orientés ?

3. Un arbre T est défini rapidement inductivement, comme soit un sommet unique et aucun arc, ou un sommet-racine s, n sommets fils (c'est-à dire n arcs $s \to s_1$, $\ldots s \to s_n$) et n sous arbres $T_1, \ldots T_n$ de racines respectives $s_1, \ldots s_n$. En tentant de montrer qu'un arbre ne contient pas de cycle, préciser la définition.

2 Profondeur d'abord (Tarjan)

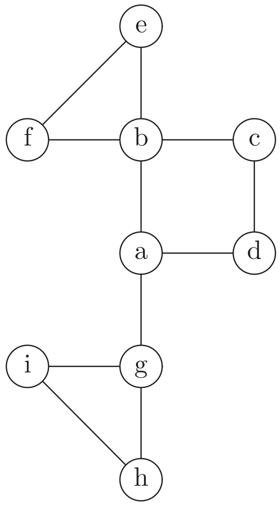
Voici un parcours en profondeur d'abord, tel tel que présenté par R. Tarjan.

```
static private int numOrder ;
static private int [] num ;
static private void doTarjan (int s, int pere) {
  numOrder++ ; num[s] = numOrder ;

  for (List p = succ[s] ; p != null ; p = p.next) {
    int n = p.val ;
    if (num[n] == 0) {
      System.out.println("Arbre : " + s + " -> " + n) ;
      doTarjan(n, s) ;
    } else if (num[n] < num[s] && n != pere) {
      System.out.println("Retour : " + s + " -> " + n) ;
    }
  }
}

static void tarjan(int s) {
  numOrder = 0 ; num = new int[succ.length] ;
  doTarjan(s,s) ;
}
```

1. R. Tarjan prétend que, appliqué à un graphe non-orienté, le parcours en profondeur d'abord revient à orienter le graphe. Justifier, puis nuancer.
2. Parcourir le graphe suivant.



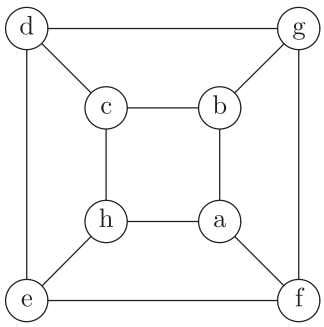
3. Un palmier est un graphe orienté (X, A), dont les arcs se partionnent en T et R tels que : (X, T) est un arbre, les arcs de R relient un sommet à un de ses ancêtres dans (X, T). Justifier que le parcours construit un palmier.
4. Discuter la présence de cycles dans le graphe d'origine selon que le parcours affiche « `Retour` » ou pas. Puis considérer le cas d'un graphe orienté.

3 Points d'articulation

On considère désormais un graphe connexe et non-orienté (X, A), sans arcs d'un sommet à lui même. Un point d'articulation a est un sommet tel qu'il existe deux autres sommets distincts s et t connectés et que a appartient à tous les chemins de s à t. (Autrement dit la suppression du sommet a coupe à coup sûr le lien entre s et t). Un graphe qui ne contient pas de point d'articulation est biconnexe.

Par ailleurs on peut définir une relation (d'équivalence) sur les *arcs*, deux arcs sont équivalents quand il sont égaux ou si existe un cycle les contenant tous deux. Un graphe est biconnexe quand tous ses arcs sont en relation.

1. Les graphes suivants sont-ils biconnexes ?



2. Montrer l'équivalence des deux définitions de graphe biconnexe.
3. Toujours selon Tarjan, le programme suivant identifie les points d'articulation.

```
// renvoie le numéro du point d'attache
private static int doFindArticulation (int s, int pere) {
```

```java
    int lowpt ; boolean found = false ;
    ++numOrder ; lowpt = num[s] = numOrder ;

    for (List p = succ[s] ; p != null ; p = p.next) {
      int n = p.val ;
      if (num[n] == 0) {
        int lowN = doFindArticulation(n, s) ;
        lowpt = Math.min(lowN,lowpt) ;
        if (lowN >= num[s]) {
          System.out.println("Articulation␣:␣" + s) ;
        }
      } else if (num[n] < num[s] && n != pere) {
        lowpt = Math.min(num[n],lowpt) ;
      }
    }
    return lowpt ;
  }

  static void findArticulations(int s) {
    numOrder = 0 ; num = new int[succ.length] ;
    int nfils = 0 ;

    numOrder++ ; num[s] = numOrder ;
    for (List p = succ[s] ; p != null ; p = p.next) {
      int n = p.val ;
      if (num[n] == 0) {
        doFindArticulation(n, s) ;
        nfils++ ;
      }
    }
    if (nfils > 1)
      System.out.println("La␣racine␣" + s + "␣est␣un␣point␣d'articulation") ;
  }
```

Appliquer le programme au graphe de l'exercice précédent.

4. Un sommet peut-il être affiché deux fois ? Si oui donner un exemple.
5. Justifier la correction du programme. Le comparer à celui du poly.

4 Composantes biconnexes

Les composantes biconnexes sont intuitivement les composantes connexes du graphe une fois supprimés les point d'articulation. Plus précisément,la relation d'équivalence sur les arcs définit les classes d'équivalence A_1, A_2, ..., A_k des arcs du graphe (X, A). On peut alors définir les ensembles X_1, X_2, ..., X_k de sommets participant respectivement aux arcs de A_1, A_2, ..., A_k.

Il est évident que les graphes (X_i, A_i) sont biconnectés (et en un sens maximaux). Il semble logique que les points d'articulations sont les éventuels points communs des X_i (on ne tentera aucune preuve).

1. Trouver les composantes biconnexes des graphes non-biconnexes déjà vus.
2. Concevoir, en utilisant une pile d'arcs, un programme qui affiche les (arcs des) composantes biconnexes, à raison d'une ligne par composante biconnexe.

5 Calcul naïf des composantes fortement connexes

1. Rappeler la définition des composantes fortement connexes dans un graphe orienté.
2. Caractériser la composante fortement connexe d'un sommet à l'aide du graphe des prédécesseurs (ou graphe inverse).
3. Concevoir un algorithme simple pour afficher les composantes fortement connexes. Indication : on utilisera le graphe des des prédécesseurs, et deux méthodes d'exploration partageant un tableau de marques.
4. Trouver un exemple d'exécution qui montre que l'algorithme est au moins quadratique (en le nombre de sommet).
5. Programmer.

Solutions des exercices

Exercice 1

1. Si le chemin n'est pas simple, il contient un sommet répété. C'est-à-dire qu'il existe un sous-chemin $t = s_i \rightarrow s_{i+1} \rightarrow \ldots \rightarrow s_{i+k} = t$ $(k > 1)$. Le nouveau chemin $s = s_1 \rightarrow \ldots \rightarrow s_i \rightarrow s_{i+k+1} \rightarrow \ldots \rightarrow s'$ est de longueur strictement inférieure au chemin initial. En itérant le procédé on obtient un chemin simple (c'est une preuve par récurrence un peu rapide).
2. Le procédé itératif précédent s'applique encore, il produira un chemin de s à s dont tous les sommets « intérieurs » sont deux à deux distincts et distincts de s.

   Dans un graphe orienté la condition « sommets deux à deux distincts » entraîne que tous les arcs du chemin sont deux à deux distincts. On a donc produit un cycle.

   Ce n'est plus vrai dans un graphe non-orienté. Considérer le graphe s_1, s_2 avec $s_1 \leftrightarrow s_2$. Le chemin $s_1 \leftrightarrow s_2 \leftrightarrow s_1$ contient l'arc $s_1 \leftrightarrow s_2$ deux fois et n'est donc pas un cycle.
3. Par induction évidemment. Il faut préciser un peu la définition en ajoutant que les sommets des sous-arbres sont tous deux à deux distincts et distincts de la racine. Un sommet solitaire ne permet aucun chemin et donc aucun cycle. Si l'arbre T contient un cycle, ce cycle contient des sommets de deux sous-arbres distincts, disons T_1 et T_2 (sinon un sous-arbre contiendrait un cycle). Un tel cycle contient nécessairement un « passage » de T_1 vers T_2 (et ceci même si le cycle passe par la racine, ce qui est à priori possible dans le cas non-orienté). Plus précisément, il existe deux sommets u_1 et u_2, reliés par un arc, et tel que $u_1 \in T_1$ et $u_2 \in T_2$. Or, l'arc $u_1 \rightarrow u_2$ ne peut appartenir ni à T_1, ni à T_2 (ni à aucun autre T_i, ni relier s à un de ses fils), ce n'est donc pas un arc de T, contradiction.

Exercice 2

1. Distinguons comme le programme deux catégories d'arcs produits. Les arcs de l'arbre de recouvrement, notés $\rightarrow$ et les arcs de retour notés $- \rightarrow$. Il est clair que si les sommets s et t sont reliés par un de ces arcs alors, il existe un arc $s \leftrightarrow t$ dans le graphe.

   Par ailleurs, le parcours visite tous les sommets (démontrable par un argument sur la distance au sommet de départ) et donc tous les arcs du graphe d'origine *entre sommets distincts* sont visités deux fois. Ces arcs sont orientés lors de la première visite. Les arcs de la forme $s \leftrightarrow s$ sont visités une fois et ne sont pas orientés.

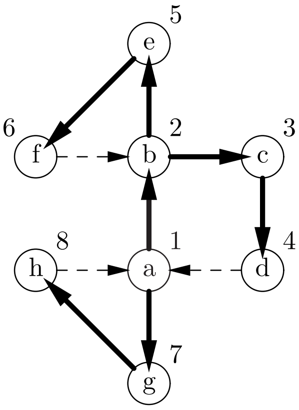
   Si un arc de l'arbre $s \rightarrow t$ est construit, alors un arc $t - \rightarrow s$ ne peut pas être construit (à cause du test `n != pere`).

   Si l'arc $s - \rightarrow n$ est construit, alors l'arc $n - \rightarrow s$ ne peut pas être construit (à cause de la condition `num[n] < num[s]`). L'arc $s \rightarrow n$ n'est évidemment pas construit, et l'arc $n \rightarrow s$ ne sera pas construit plus tard (car s est numéroté).

   Il y a deux nuances. D'une part il faut supposer le graphe d'origine connexe, et d'autre part les arcs $s \leftrightarrow s$ disparaissent.
2. Voir la figure 1. Les numéros sont indiqués.
3. Par construction même, les arcs `arbre` définissent bien un arbre, au sens de la définition inductive amendée. Soit ensuite un arc de retour $s - \rightarrow t$. Il faut montrer l'existence d'un chemin $t \rightarrow^* s$. Soit il faut montrer que s est numéroté lors de l'appel `doTarjan(t,_)`. Par construction, t porte un numéro plus petit que s, c'est-à-dire que t est numéroté avant s. Le sommet t ne peut donc pas être un descendant de s.

   Par ailleurs, l'arc $t \rightarrow s$ n'est pas construit. Or si, s n'était pas numéroté lors de l'appel `doTarjan(t,_)`, alors il existerait un sommet u ancêtre strict à la fois de s et t, et deux arcs $u \rightarrow u_1$ et $u \rightarrow u_2$ distincts. tels que u_1 est un ancêtre de t et u_2 est un ancêtre de s. Mais alors l'arc $t \leftrightarrow s$ serait examiné et l'arc $t \rightarrow s$ construit.

Fig. 1 – Un palmier

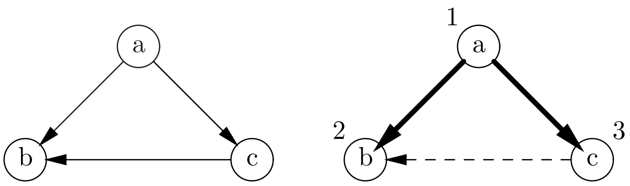


4. Si aucun arc de retour n'est construit, alors en supprimant l'orientation des arcs de (X, T) on retrouve le graphe d'origine, qui est donc un arbre.

   Le raisonnement vaut encore pour un graphe d'origine orienté. Par conséquent si aucun arc de retour n'est construit le graphe est un arbre et est donc acyclique. À la nuance près que les cycles d'un sommet vers lui-même ne sont jamais détectés.

   Si un arc de retour est construit. Dans le cas non-orienté, le palmier contient un cycle, et donc, à plus forte raison le graphe d'origine.

   Ce n'est plus vrai dans le cas d'un graphe orienté. Considérer le parcours suivant.



   Notons bien que l'origine de l'arc de retour porte bien un numéro supérieur à sa cible. Un tel arc est plus exactement dénommé arc transverse (voir le poly)

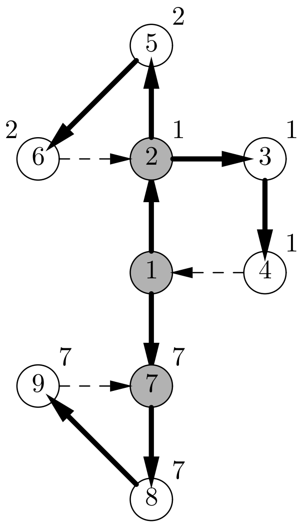
Exercice 3

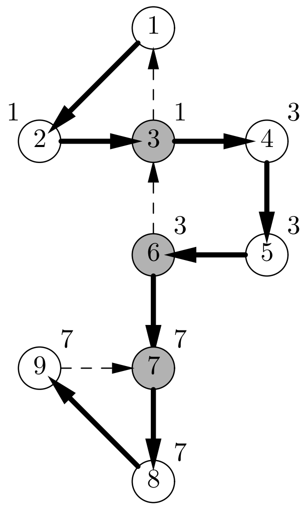
1. Oui, oui (il faut au moins trois sommets pour qu'existe un point d'articulation) non et oui.
2. Soit donc un graphe sans point d'articulation. Soient encore deux arcs distincts $s \leftrightarrow s'$ et $t \leftrightarrow t'$. On prouve l'existence d'un cycle contenant les deux arcs, par induction sur la longueur d'un chemin liant s à t (ou s à t', ou s' à t, ou s' à t').
   - Si $s = t$ (par exemple), alors s n'étant pas un point d'articulation on peut construire un chemin de s' à t' ne passant pas par s' et donc un cycle.
   - Sinon, le graphe étant connexe, il existe un chemin de s à t que l'on peut supposer simple. Prenons un arc de ce chemin $u \leftrightarrow u'$ distinct des deux chemins des extrémités. Par induction il existe deux cycles contenant l'un les arcs $s \leftrightarrow s'$ et $u \leftrightarrow u'$, l'autre contenant les arcs $t \leftrightarrow t'$ et $u \leftrightarrow u'$. En orientant les notations on peut faire que C_s et C_t sont écrits comme $s \leftrightarrow \ldots \leftrightarrow u \leftrightarrow u' \leftrightarrow \ldots \leftrightarrow s$ et $t \leftrightarrow \ldots \leftrightarrow u \leftrightarrow u' \leftrightarrow \ldots \leftrightarrow t$. Soit w le premier sommet commun à C_s et C_t et w' le dernier sommet commun. Les sommets w et w' existent (car les deux cycles commencent et finissent par des sommets distincts, et contiennent des sommets communs) et sont distincts (w doit apparaître avant u ou lui être égal, w' doit apparaître après u' ou lui être égal, et u distinct de u'). On peut

ensuite en « enlevant » les parties des cycles comprises entre w et w', produire un cycle $s \leftrightarrow \ldots \leftrightarrow w \leftrightarrow \ldots \leftrightarrow t \leftrightarrow \ldots \leftrightarrow w' \leftrightarrow \ldots \leftrightarrow s$, qui contient les arcs $s \leftrightarrow s'$ (au début ou à la fin) et $t \leftrightarrow t'$ (avant ou après t).

Réciproquement (si j'ose dire). Soit un graphe contenant un point d'articulation a. Il existe donc deux sommets distincts s et t, reliés par un chemin simple $s \leftrightarrow s' \leftrightarrow \ldots \leftrightarrow a \leftrightarrow \ldots \leftrightarrow t' \leftrightarrow t$. Il ne peut pas exister de cycle contenant à la fois les arcs, $s \leftrightarrow s'$ et $t \leftrightarrow t'$, car il n'existe pas de cycle contenant à la fois s et t. En effet il existerait alors un chemin de s à t ne passant pas par a.

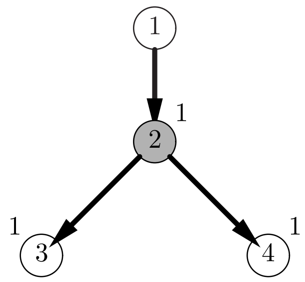
3. Et voilà, en partant de deux sommets différents.





Dans les deux cas les résultats entiers du parcours sont indiqués et les points d'articulation sont grisés.

4. Il suffit que le test `lowN >= num[s]` soit vrai deux fois. C'est le cas avec le graphe suivant pour le sommet numéroté 2.



5. Vu du palmier, l'appel `doFind(s,_)` calcule $\mathsf{lowpt}[s]$, qui est le plus petit numéro de sommet dans l'ensemble $\{ s \} \cup \{ t, s \rightarrow^{*} - \rightarrow t \}$.

   Or, soit a distinct de la racine, n un fils de a dans (X, T) (il existe $a \rightarrow n$). Si $\mathsf{lowpt}[n] \leq \mathsf{num}[a]$ alors a est un point d'articulation, car il n'existe aucun arc du graphe d'origine entre un ancêtre strict de a et n qui ne passe pas par a. En effet, tous les chemins (du graphe d'origine) ne passant pas par a restent confinés au sous-arbre de racine n, puisque les éventuels arcs de retour issus d'un sommet de T_n pointent vers des ancêtres de leur origine qui sont nécessairement des sommets de T_n.

   Si $\mathsf{lowpt}[n] > \mathsf{num}[a]$ alors ont peut construire un chemin ne passant pas par a à tout autre sommet du graphe, en employant au besoin un arc de retour dans le bon sens et des arcs de l'arbre dans un sens indifférent.

   Le cas de la racine est encore plus simple. Elle est un point d'articulation si et seulement si elle possède deux fils ou plus dans (X, T).

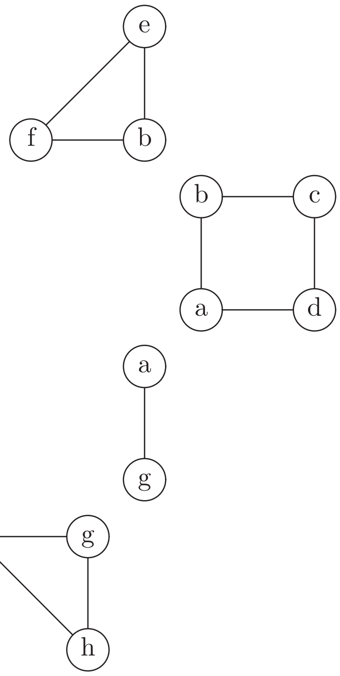
Enfin les feuilles (de l'arbre de recouvrement) ne peuvent pas être détectées comme point d'articulation. Or, elles ne sont jamais des points d'articulation du graphe d'origine, n'étant pas même des points d'articulation de l'arbre (non-orienté).

Le programme du poly s'abstient du test `num[n] < num[s] && n != pere` ce qui n'est pas gênant, car alors la valeur de `lowpt` n'est pas changée. En effet dans ces deux cas on a $\mathsf{num}[n] \geq \mathsf{num}[s]$, et on sait par ailleurs $\mathsf{lowpt}[s] \leq \mathsf{num}[s]$.

Exercice 4

– Pour le graphe à trois sommets.

Et pour l'autre.



– Voici le programme.

```
private static Lifo stack ;

private static int doFindBiconnect (int s, int pere) {
  int lowpt ;
  ++numOrder ; lowpt = num[s] = numOrder ;

  for (List p = succ[s] ; p != null ; p = p.next) {
    int n = p.val ;
    if (num[n] == 0) {
      stack.push(new Edge(s,n)) ;
      int lowN = doFindBiconnect(n, s) ;
      lowpt = Math.min(lowN,lowpt) ;
      if (lowN >= num[s]) {
        System.out.print("Composante␣␣:␣" + s ) ;
        Edge e = stack.pop () ;
        while (num[e.src] >= num[n]) {
          System.out.print("␣" + e) ;
```

```
          e = stack.pop() ;
        }
        System.out.println("␣" + e) ;
      }
    } else if (num[n] < num[s] && n != pere) {
      stack.push(new Edge(s,n)) ;
      lowpt = Math.min(num[n],lowpt) ;
    }
  }
  return lowpt ;
}

static void findBiconnect(int s) {
  numOrder = 0 ; num = new int[succ.length] ;
  stack = new Lifo () ;
  doFindBiconnect(s,s) ;
}
```

Le code est compliqué par la boucle d'affichage des arcs.
L'idée est d'abord d'empiler les arcs du palmier. L'affichage des composantes est en fait assez simple : juste après un appel récursif, tous les arcs visités durant l'appel ont été empilés et plus aucun arc concernant les sommets visités ne sera empilé.
Prenons le cas de `s` point d'articulation, possédant un seul fils, et dont aucun descendant dans l'arbre n'est un point d'articulation, les arcs de la composante sont alors sur la pile, du sommet au dernier arc empilé avant l'appel récursif. Ce dernier arc est le premier à partir du haut dont le numéro de la source est inférieur au numéro de n (de fait c'est s).
Dans le cas de `s` point d'articulation, possédant deux sous-arbres dont l'un est une composante, on voit que la méthode `doFind` rendra une pile contenant les arcs du sous-arbre qui n'est pas la composante et ceci que la composante soit parcourue en premier ou en second.
R. Tarjan propose une preuve utilisant à proprement parler l'induction sur le nombre de composantes biconnexes, et reposant au final sur les remarques ci-dessus.

Exercice 5

Je n'ai pas le temps de détailler pour le moment, voici un code qui répond à la question.

```
class OGraph { // Graphes orientés
  private List [] succ ;

  OGraph (int size) { succ  = new List[size] ;}

  void arc(int src, int dst) {
    succ[src] = new List (dst,succ[src]) ;
  }

  OGraph inverse () {
    OGraph r = new OGraph(succ.length) ;

    for (int src = 0 ; src < succ.length ; src++)
      for (List p = succ[src] ; p != null ; p = p.next)
        r.arc(p.val,src) ;
    return r ;
  }

  void descendre(int s, int orderSC, int [] numSC) {
```

```
    numSC[s] = -orderSC ;
    for (List p = succ[s] ; p != null ; p = p.next)
      if (numSC[p.val] <= 0 && numSC[p.val] != -orderSC)
        descendre(p.val, orderSC, numSC) ;
  }

  void remonter(int s, int orderSC, int [] numSC) {
    numSC[s] = orderSC ;
    System.out.print("␣" + s) ;
    for (List p = succ[s] ; p != null ; p = p.next)
      if (numSC[p.val] == -orderSC)
        remonter(p.val, orderSC, numSC) ;
  }


  void findStrongConnect () {
    OGraph ig = inverse () ;
    int  orderSC = 0 ;
    int [] numSC = new int[succ.length] ;

    for (int s = 0 ; s < succ.length ; s++) {
      if (numSC[s] <= 0) {
        ++orderSC ;
        System.out.print("SCC␣" + orderSC + ":") ;
        descendre(s, orderSC, numSC)  ;
        ig.remonter(s, orderSC, numSC) ;
        System.out.println() ;
      }
    }
  }


  public static void main (String [] argv) {
    OGraph g = new OGraph(11) ;
    g.arc(0,1) ; g.arc(0,4) ;
    g.arc(1,2) ; g.arc(1,3) ; g.arc(1,8) ;
    g.arc(2,3) ;
    g.arc(4,5) ; g.arc(4,0) ; g.arc(4,5) ;
    g.arc(5,6) ; g.arc(5,7) ; g.arc(5,8) ;
    g.arc(9,10) ;
    g.arc(0,10) ; g.arc(10,4) ;
    g.print() ;
    g.findStrongConnect() ;
  }
}
```