Parcours de graphe

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/01/

1 Notre amie, la classe des listes

```
class List {
  int val ;  List next ;

  List (int val, List next) {
    this.val = val ; this.next = next ;
  }
}
```

1. Comment représenter la liste vide ? Si `p` est de type `List`, peut on toujours faire `p.val` ?
2. Comment définir et initialiser une variable `p` à la liste $1, 2, 3$?
3. Comment comprendre l'affectation `this.next = next` ? Proposer un codage qui évite **this**.

2 Une structure de donnée très abstraite

```
interface Bag {
  boolean isEmpty () ;
  void add(int s) ; // Ajoute un élément au sac.
  int get() ;       // Renvoie (et enlève) un élement du sac.
}
```

1. Comment comprendre cette construction bizarre de Java ? De quoi nos sacs sont-ils des sacs ? Les sacs sont-ils des ensembles ?
2. À priori, que fait ce petit code :

```
// b est un Bag
   for (int i = 0 ; i < 16 ; i++)
     b.add(i) ;
   while (!b.isEmpty())
     System.out.println(b.get()) ;
```

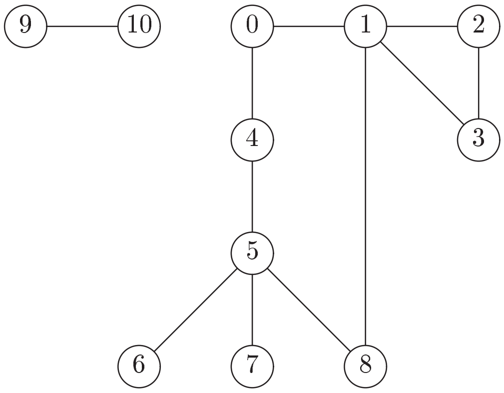
   Qu'obtient-on si le sac est une pile, une file ?
3. Écrire une classe `Lifo` des piles implémentant l'interface `Bag`.

3 Réalisation des graphes

```
class Graph { // Graphes non-orientés
  static List [] succ ;

  static void empty(int size) { succ  = new List[size] ;}

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

1. Allocations : Que font `succ[src] = new List (dst, succ[src])` et `succ = new List [size]` ?

2. Où sont les sommets et les arcs dans cette représentation des graphes ?
3. Justifier le commentaire. Comment faire une classe des graphes orientés ?
4. Construire ce graphe :



5. Une particularité de l'allocation des tableaux (d'objets) garantit la correction de notre représentation, laquelle ? Qu'en est-il des tableaux de scalaires ?

4 Réalisation du parcours avec un sac

On choisit un parcours du graphe particulièrement abstrait :

```
static void xfs(int s, Bag b) {
  boolean [] vu = new boolean[succ.length] ;

  b.add(s) ; vu[s] = true ;
  do {
    int n = b.get() ;
    System.out.print("␣" + n) ;
    for (List p = succ[n] ; p != null ; p = p.next)
      if (!vu[p.val]) {
        b.add(p.val) ; vu[p.val] = true ;
      }
  } while (!b.isEmpty()) ;
}
```

1. Peut-on se passer de la variable `n` ?
2. Commençons par prouver la terminaison. On pourra chercher à partitionner l'ensemble des sommets en trois.
3. Poursuivons par la correction ainsi exprimée : notre méthode affiche la composante connexe de s.
4. Enfin terminons par le coût, montrer que le parcours est en $O(V + E)$, où V est le nombre de sommets et E le nombre d'arcs (de la composante connexe de s).
5. Comment procéder pour afficher toutes les composantes connexes du graphe ?

5 Arbre de recouvrement

Voici une variation sur parcours avec un sac.

```
static int [] findSpanningTree(int s, Bag b) {
  int [] pere = new int[succ.length] ;

  for (int i = pere.length-1 ; i >= 0 ; i--)
    pere[i] = -1 ;

  b.add(s) ; pere[s] = s ; // par convention
  do {
    int n = b.get() ;
    for (List p = succ[n] ; p != null ; p = p.next)
      if (pere[p.val] < 0) {
        b.add(p.val) ; pere[p.val] = n;
      }
  } while (!b.isEmpty()) ;
  return pere ;
}
```

1. Quelle est l'objet renvoyé ? Pourquoi le tableau `pere` est-il initialisé explicitement ?
2. Je prétends que cette méthode renvoie un arbre. Comment est-il représenté ? Donner des arbres possibles dans le cas où le sac est une file, une pile (départ en zéro).
3. Je prétend encore que cet arbre est un arbre de recouvrement de la composante connexe de s. Ai-je raison ?
4. Notre programme peut-il produire n'importe quel arbre de recouvrement ?
5. Comment faire pour, étant donnés deux sommets s et s', afficher un chemin de s à s' si il en existe ?
6. Comment faire pour, étant donnés deux sommets s et s', afficher un plus court chemin de s à s' si il en existe ?

6 Parcours en profondeur d'abord

Dans la question précédente, nous avons vu que le codage en pratique du parcours en largeur d'abord peut s'éloigner un peu de du codage avec un sac. Voici le parcours en profondeur programmé récursivement. (l'arbre de recouvrement est affiché à la volée).

```
static private void doDfs (int s, int pere,boolean [] vu) {

  if (vu[s]) return ;
  vu[s] = true ;
  System.out.print ("␣" + s + "<" + pere + ">") ;
  for (List p = succ[s] ; p != null ; p = p.next) {
    doDfs(p.val, s, vu) ;
  }
}

static void dfs(int s) {
  boolean [] vu = new boolean [succ.length] ;

  doDfs(s,s,vu) ;
}
```

1. Modifier le programme pour qu'il numérote les sommets dans l'ordre du parcours (numérotation préfixe de l'arbre de recouvrement).

2. Obtenir une numérotation suffixe de l'arbre de recouvrement.
3. Programmer un véritable parcours en profondeur d'abord avec une pile explicite.

Solutions des exercices

Exercice 1

1. La liste vide est représentée par la constante **null**. Si `p` est **null**, `p.next` échoue en levant une exception.
2. Au plus simple :

   ```
   List p = new List (1, new List (2, new List (3, null))) ;
   ```

3. Le mot-clé **this** représente l'objet lui-même (ici en cours de création, puisque nous somme dans un constructeur). Ici **this**.`val` désigne le champ `val` de l'objet **this**.

   On a tendance à ne pas expliciter **this**, mais ici, si on écrit `val` tout court, on fait référence à l'argument homonyme. Pour éviter l'emploi explicite de **this**, il suffit ici de renommer les arguments du constructeur.

   ```
    List (int v, List n) {
       val = v ; next = n ;
     }
   }
   ```

Exercice 2

1. La construction est une interface, c'est à dire qu'elle définit un certain nombre de méthodes (y compris leurs types ou signatures) que toute classe prétendant implémenter devra au moins posséder, avec des types compatibles (égaux ?). Le compilateur vérifiera. Par ailleurs les commentaires donnent une propriété-clé des sacs Attention, rien dans le compilateur ne garantira cette propriété ou contrat, le programmeur devra faire attention.

   Les sacs sont des sacs d'entiers (type **int**), comme montré par les signatures des méthodes `add` et `get`. Nos sacs ne sont pas des ensembles, car il peuvent contenir des doublons.
2. Le code va imprimer les entiers de 0 à 15 dans un ordre non-spécifié. La propriété des sacs garantit la terminaison de la boucle **while**. Pour une pile (*Last in, first out*), les entiers sont imprimés selon l'ordre décroissant. Pour les files (*First in, first out*) c'est l'ordre croissant.
3. Et voilà :

   ```
   class Lifo implements Bag {
     private List me ;

     Lifo () { me = null ; }

     public boolean isEmpty() { return me == null ; }

     public int get () {
       if (me == null) throw new Error ("Pile␣vide") ;

       int r = me.val ;
       me = me.next ;
       return r ;
     }

     public void add (int x) { me = new List (x,me) ; }
   }
   ```

   Bizarrement, les méthodes déclarées dans l'interface sont implicitement publiques.

Exercice 3

1. Une allocation de cellule de liste (un objet de classe `List`) et une allocation d'un tableau de size `size`. Attention dans le dernier cas, aucune cellule de liste n'est allouée.
2. Les graphes sont représentés par listes de voisins. Les sommets sont des entiers de 0 à un entier donné égal à la taille du tableau `succ` moins 1.
3. La représentation par listes de voisins décrit plutôt un graphe orienté. Toutefois, ici la méthode `add` garantit que si s' est dans la liste des voisins de s, alors s est dans la liste des voisins de s'.
4. Facile, commencer par les 11 sommets (`empty(11)`) poursuivre en ajoutant les arcs un par un.
5. Les cases des tableaux d'objets valent initialement **null**, ce qui correspond bien à une liste vide de voisins. Dans le cas des scalaires, on a une valeur particulière qui peut passer pour le zéro de son type (et **false** pour les booléens). En fait, la même règle s'applique en Java à toutes les variables non initialisées. Attention, dans certains langages (C, Pascal) certaines variables ont un contenu indéfini à la création.

Exercice 4

1. Pas vraiment, si on remplace ses deux occurrences par `b.get` on retire deux éléments du sac à chaque itération.
2. Divisons l'ensemble d'abord en deux, sommets vus et non-vus (selon la valeur de `vu[s]`). Puis divisons vraiment en trois, non-vus N, dans le sac (et donc forcément vus) S, et visités V.

   Chaque itération de la boucle **while** termine (car la boucle **for** termine) et se solde par l'ajout d'exactement un élément à V. Le programme termine donc, car V ne peut croître indéfiniment.
3. À un instant donné, les sommets affichés sont ceux de V. Dès qu'un sommet passe de S à V tous ses voisins non encore vus passent de N à S et donc seront affichés une fois le programme terminé (S est alors vide).

   On procède par induction sur la longueur du chemin de s à s' (car on se souvient de la définition de la composante connexe).
   – Si $s = s'$, alors s' est affiché à la première itération.
   – Écrivons le chemin comme $[s = s_1, s_2, \ldots, s_k, s_{k+1} = s']$. Par induction, s_k est affiché lors d'une itération donnée. À la sortie de cette même itération tous les voisins de s_k sont nécessairement vus, et sont donc soit déjà affichés, soit dans le sac. Ces derniers seront affichés avant la fin du la boucle **while**, car le sac est alors vide.

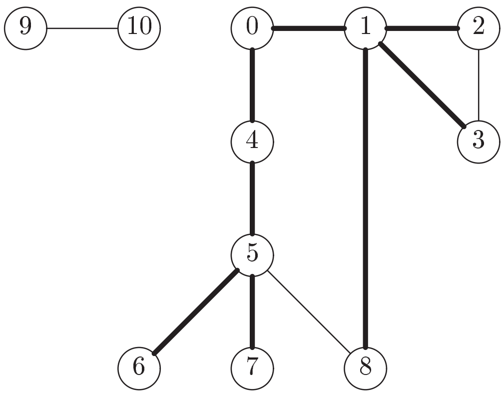
   En fait les sommets affichés sont exactement ceux que l'on peut atteindre à partir de s, puisqu'aucun sommet n'est affiché deux fois.
4. On voit facilement que la boucle **while** s'exécute V fois (c'est la preuve de terminaison). Reste à majorer le nombre d'itérations de la boucle **for**. On procède globalement. Pour un sommet donné (une itération donnée de la boucle **while**), tous les voisins sont examinés une fois. Chaque arc est donc examiné deux fois.
5. Il faut parcourir tous les sommets du graphe. Une technique possible est de faire de `vu` un tableau d'entiers destiné à enregistrer les numéros de composante connexe.

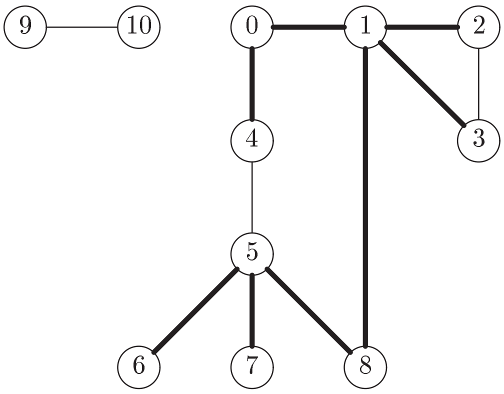
```java
private static void xcc(int s, Bag b,int [] cc, int ncc) {

  b.add(s) ; cc[s] = ncc ;
  while (!b.isEmpty()) {
    int n = b.get() ;
    System.out.print(" " + n) ;
    for (List p = succ[n] ; p != null ; p = p.next)
```

```
        if (cc[p.val] == 0) {
          b.add(p.val) ; cc[p.val] = ncc ;
        }
    }
    System.out.println() ;
  }

  static void findCC(Bag b) {
    int [] cc = new int[succ.length] ;
    int ncc = 1 ;
    for (int s = 0 ; s < succ.length ; s++)
      if (cc[s] == 0) {
        xcc(s,b,cc,ncc) ;
        ncc++ ;
      }
  }
```

Exercice 5

1. Comme indiqué par son type, la méthode renvoie un tableau d'entiers de taille le nombre de sommet du graphe. Les éléments du tableau sont des sommets du graphe. Le tableau `pere` joue (entre autres) le rôle du tableau `vu`. Ses cases étant remplies avec des numéros de sommets au cours du parcours, il faut une valeur distincte de tous les numéros de sommets possibles pour signifier « non-vu ».
2. Le tableau `pere` encode un graphe, `pere[i] == j` signifiant un arc entre les sommets i et j. C'est initialement l'arbre formé de l'unique sommet s. Chaque itération de boucle interne introduit un nouveau sommet et un arc de ce nouveau sommet à un sommet de l'arbre. On ne peut ainsi créer ni boucle ni partage et tous les sommets sont reliés à la racine. (Ou mieux, tous les sommets sont reliés à la racine par un unique chemin sans répétition d'arcs.)





3. Il faut noter que l'arbre obtenu avec la pile n'est pas tout à fait celui que donnerait un parcours en profondeur d'abord. Avec notre méthode il est impossible de produire un arbre filiforme, puisque tous les arcs issus de la racine sont ajoutés en même temps.
4. Un arbre de recouvrement est simplement un arbre dont les sommets sont la composante connexe de s et les arcs des arcs du graphe.
5. Et voilà :

```
static void findPath(int s, int ss) {
  int [] pere = findSpanningTree(ss,new Lifo ()) ;

  if (pere[s] < 0) {
    System.out.println("Pas␣de␣chemin␣de␣" + s + "␣à␣" + ss) ;
  } else {
    int j = s ;
    while (j != ss) {
      System.out.print(j + "␣->␣") ;
      j = pere[j] ;
    }
    System.out.println(ss) ;
  }
}
```

6. Pour obtenir le (un) plus court chemin, il faut adopter une file. Le principe est que les sommets sont alors visités des plus proches du point de départ aux plus lointains.

   Essayons nous à le prouver, en montrant que l'arbre de recouvrement est, dans ce cas, une arborescence des plus courts chemins.

   Il suffit de montrer que si le tableau `pere` est bien une arborescence des plus courts chemins au début d'une itération, il en va de même à la sortie.

   Une première remarque simple (valant d'ailleurs pour tous les sacs) est que si un sommet appartient à V, alors tous ses voisins sont vus.

   Appelons distance de s à s', notée $d(s')$ la longueur d'un plus court chemin de s à s'. Par ailleurs, définissons d comme la plus grande des distances des sommets de V (ou -1 si V est vide). Montrons que du début à la fin d'une itération, la partition des sommets en V, file (non-vide) et N s'organise selon l'un ou l'autre des schémas suivants :
   - La file contient tous les sommets de distance $d+1$.
   - L'ensemble N regroupe tous les sommets de distance supérieure ou égale à $d+2$.

   Ou alors,
   - Le contenu de la file est de la forme $[s_1; s_2; \ldots s_n; t_1; t_2; \ldots; t_m]$ ou les s_i sont à la distance d de la racine et les t_i à la distance $d+1$. Par ailleurs il y a au moins un s_i.
   - Les sommets de N sont à une distance supérieure ou égale à $d+1$.

Notons que le premier schéma s'applique initialement.

Donc on extrait s_1 de la file, il y a deux selon l'organisation présente. Commençons par la seconde organisation :

(a) $d(s_1) = d$, alors, par définition du plus court chemin les voisins de s_1 sont de distance inférieure ou égale à $d+1$. Mais ceux des voisins qui sont ajoutés à la fin de la file sont nécessairement de distance supérieure ou égale à $d+1$ (car ce sont les voisins non-vus). Les arcs ajoutés à l'arborescence sont donc bien des plus court chemins. Par ailleurs s_1 passe dans V ce qui ne change pas d. Si s_2 existe on est donc encore dans le cadre de la seconde organisation. Sinon, il suffit de montrer que tous les sommets de distance $d+1$ sont dans la file. Or c'est bien le cas car les sommets de distance d sont tous dans V et donc tous leurs voisins sont vus, ceux qui sont de distance $d+1$ ne peuvent se trouver que dans la file,

(b) Dans la première organisation, le raisonnement est à peu près identique. Dans tous les cas d s'incrémente en d'. Si la file ne contenait qu'un élément on retrouve ensuite la première organisation (ou une file vide), sinon on retrouve la seconde organisation.

Le raisonnement peut d'ailleurs se transformer en un programme, qui explicite les s_i et et t_i à l'aide de deux listes `cur` et `next`.

```
static void bfs(int s) {
  List cur, next=null ;
  boolean [] vu = new boolean [succ.length] ;
  int d = -1 ;

  cur = new List(s,null) ; vu[s] = true ;
  while (cur != null) {
    d++ ; System.out.print("␣[" + d + "]") ;
    for (List p = cur ; p != null ; p = p.next) {
      System.out.print("␣" + p.val) ;
      for (List q = succ[p.val] ; q != null ; q = q.next) {
        if (!vu[q.val]) {
          vu[q.val] = true ;
          next = new List(q.val,next) ;
        }
      }
    }
    cur = next ; next = null ; d++ ;
  }
}
```

Notons qu'en entrant dans la boucle **for** `(List p =`..., la liste `cur` contient tous les sommets de distance d, tandis qu'en sortant de cette même boucle la liste `next` contient tous les sommets de distance $d+1$.

On notera aussi que le point important est que les sommets sont introduits dans l'arborescence selon les distances croissantes. En effet la méthode `bfs` n'a pas tout à fait le même comportement que le parcours à l'aide d'une file, (il y a en quelque sorte inversion des listes de voisins).

Exercice 6

1. Et voilà, en utilisant, `num` passé en argument et rendu en résultat. Une variable globale (statique) fait tout aussi bien l'affaire.

```
static private int doPrefix (int s, int num, boolean [] vu) {
```

```
    if (vu[s]) return num ;
    vu[s] = true ;
    System.out.print("␣" + s + ":" + num) ; num++ ;
    for (List p = succ[s] ; p != null ; p = p.next) {
      num = doPrefix(p.val, num, vu) ;
    }
    return num ;
  }

  static void prefix(int s) {
    boolean [] vu = new boolean [succ.length] ;

    doPrefix(s,0,vu) ; System.out.println() ;
  }
```

2. Et voilà :

```
  static private int doSuffix (int s, int num, boolean [] vu) {

    if (vu[s]) return num ;
    vu[s] = true ;
    for (List p = succ[s] ; p != null ; p = p.next) {
      num = doSuffix(p.val, num, vu) ;
    }
    System.out.print("␣" + s + ":" + num) ; num++ ;
    return num ;
  }

  static void suffix(int s) {
    boolean [] vu = new boolean [succ.length] ;

    doSuffix(s,0,vu) ; System.out.println() ;
  }
```

La seule différence est que les numéros sont attribués après le parcours récursif des voisins.

3. C'est un peu gratuit. Une solution possible est de marquer les sommets comme vus à la sortie de la pile.

```
  private static void xbfs(int s) {
    Lifo stack = new Lifo () ;
    boolean [] vu = new boolean[succ.length] ;

    stack.add(s) ; stack.add(s) ;
    do {
      int n = stack.get () ;
      int pere = stack.get () ;
      if (!vu[n]) {
        vu[n] = true ;
        System.out.print("␣" + n + "<" + pere + ">") ;
        for (List p = succ[n] ; p != null ; p = p.next) {
          stack.add(n) ; stack.add(p.val) ;
        }
      }
    } while (!stack.isEmpty()) ;
  }
```

On notera que la pile est utilisée pour se souvenir du père de chaque sommet empilé.

Par ailleurs, la terminaison devient un peu plus subtile. Il suffit de considérer la couple d'entiers $(V - n_v, n_s)$ où V est le nombre de sommets du graphe, n_v est le nombre de sommets vus et n_s la taille de la pile. Alors, du début à la fin d'une itération, ce couple décroît strictement pour l'ordre lexicographique sur les paires d'entiers naturels. Si cette explication choque un peu, considérer l'entier $2V(V - n_v) + n_s$, strictement décroissant.