**Mathématiques pour l'informatique** **Licence 3 d'informatique**

Travaux pratiques n° 3 — Parcours en largeur et quelques applications

## *I. Introduction et rappels*

L'objectif de ce TP est d'implémenter l'algorithme de parcours en largeur avec comme application la recherche de plus courts chemins combinatoires et l'élaboration d'un calcul de composantes connexes pour les graphes.

Les sommets d'un graphe $G$ sur $n$ sommets sont numérotés par les entiers de 1 à $n$ (c'est notre convention depuis la dernière série d'exercices) ; les graphes sont représentés par des tableaux de listes d'adjacence, par conséquent, on utilise la classe *GraphSimple* élaborée dans la seconde série d'exercices.

## *II. Description de l'algorithme du parcours en largeur*

Pour le parcours en largeur, on fixe un sommet origine $r$ (la racine du parcours) à partir duquel on va explorer (parcourir) le graphe en largeur.

Pour gérer le parcours (et, en particulier, éviter de tomber dans des boucles infinies en suivant les arêtes du graphe), on utilise pendant l'exécution de l'algorithme, un code-couleur qui indique le statut d'un sommet vis-à-vis du parcours (dans l'algorithme, il est représenté par un attribut *color* sur les sommets) :

— les sommets **verts** sont les sommets qui ne sont pas encore explorés (à l'origine, tous les sommets sont verts) ;

— les sommets **orange** sont les sommets que l'on a « vus », mais dont on n'a pas exploré les voisins (comme on le verra, lorsqu'un sommet est orange, il est stocké dans la file d'attente utilisée par l'algorithme) ;

— les sommets **rouges** sont les sommets explorés dont on a également exploré les voisins (un sommet devient rouge au moment où il sort de la file d'attente).

Pendant le parcours, on établit deux attributs sur les sommets, l'un nommé *parent* définit une arborescence enracinée au sommet $r$ (l'origine, ou la racine, du parcours en largeur), l'autre est un entier qui donne la distance, en nombre d'arêtes ou d'arcs entre la racine du parcours et un sommet (cet attribut de distance n'a de sens que lorsque le sommet est rouge, sinon, il est indéfini). Voici quelques détails sur la gestion de ces deux attributs et la manière dont ils sont définis :

— l'attribut ***parent*** sur chaque sommet permet de « remonter » dans l'arborescence définie par le parcours en largeur ; le parent d'un sommet $x$ est un autre sommet, $y$, tel que $y$ est le prédécesseur de $x$ dans le chemin qui mène de $r$ à $x$ decouvert par le parcours en largeur ; évidemment, la racine du parcours, $r$, n'a pas de prédécesseur (car l'unique chemin de $r$ à $r$ découvert par le parcours est le chemin vide d'arêtes) et, dans ce cas, on donnera à l'attribut la valeur sentinelle 0 (qui n'est pas le numéro d'un sommet) ;

— l'attribut ***distance*** (en nombre d'arêtes) permet, lorsque l'algorithme a terminé son exécution, de connaître la distance en nombre d'arêtes, du plus court chemin allant de $r$ à tout autre sommet (accessible depuis $r$) : cette distance est à valeurs entières positives ou nulles, et on a évidemment $distance(r) = 0$.

Patrice.Naudin@math.univ-poitiers.fr

Le pseudo-code du parcours en largeur à partir d'un sommet $r$ est le suivant :

« initialisation » : **pour tout** $x$ sommet du graphe $G$ **faire**
  color($x$) ⟵ *Green* ;
**fait**
Créer une file d'attente $F$, initialement vide
Placer $r$ dans $F$ ;
distance($r$) ⟵ 0 ; color($r$) ⟵ Orange ; parent($r$) ⟵ 0 ;
« itération » : **tant que** $F$ n'est pas vide **faire**
  Retirer un sommet, $x$, de $F$
  **pour tout** $y$ adjacent à $x$ **faire**
    **si** color($y$) = *Green* **alors**
      color($y$) ⟵ *Orange* ; distance($y$) ⟵ distance($x$) + 1 ; parent($y$) ⟵ $x$ ;
      Placer $y$ dans $F$
    **fin si**
  **fait**
  color($x$) ⟵ *Red*
**fait**

### *II. Éléments Java pour traduire l'algorithmique utilisée*

Voyons tout d'abord comment réaliser les itérations sur un graphe que l'on utilise dans l'algorithme précédent. Tout d'abord, l'ensemble des sommets du graphe étant constitué des entiers de 1 à $n$ (qui est l'ordre du graphe, donné par la fonction *order*), on peut itérer dans les sommets du graphe nommé *graph*, c'est-à-dire exécuter la boucle

- « **pour tout** $x$ **sommet du graphe faire** ..... **fait** »

en utilisant la construction Java suivante

- « **for** (**int** $x = 1$ ; $x$ <= *graph.order*() ; $x$++) {.....}.

De manière sensiblement analogue, la boucle qui permet d'itérer sur les voisins d'un sommet $x$ donné de *graph*, à savoir la boucle (utilisée dans l'algorithme précédent)

- « **pour tout** $y$ **adjacent au sommet** $x$ **faire** ..... **fait** »

pourra être réalisée en itérant dans la liste d'adjacence du sommet $x$ en utilisant la construction *getAdjacencyList*

- « **for** (**int** $y$ : *graph.getAdjacencyList*($x$)) {.....}.

Et voilà, ce n'est pas plus difficile que cela.

Comme on l'a vu, on a besoin d'une file d'attente dans l'algorithme de parcours en largeur ; cette file d'attente est de taille bornée par l'ordre du graphe, on peut donc profiter de ce fait pour implémenter la file dans un tableau ; sinon, il est possible d'utiliser la classe prédéfinie *LinkedList* qui contient tout ce qu'il faut pour réaliser une file d'attente, une pile, etc.

**Exercice 1. Fonctions et types auxiliaires**

Pour implémenter le parcours en largeur, il va falloir implémenter les trois attributs sur les sommets : la couleur, le parent, et la distance. Ces attributs pourront être représentés par des tableaux, indexés par les sommets (attention au fait que les sommets sont les entiers de 1 à $n$), et dont les valeurs sont respectivement de type couleur, entier et entier. Toute la structure qui est décrite ici peut composer une classe qui sera utilisée pour faire le parcours en largeur d'un graphe.

**a.** Il faut tout d'abord définir un type énuméré *color* à trois valeurs *Green*, *Orange*, *Red*.

**b.** Puis il faut définir les trois tableaux, *color*, *distance* et *parent*, ainsi que les méthodes permettant d'affecter ces attributs ou de récupérer leur valeur (ces méthodes ont toujours le sommet considéré comme unique argument [pour le getter] ou comme l'un de leurs argument [pour le setter]).

**c.** Le tableau de couleur devra être initialisé avant de commencer le parcours en largeur.

**d.** Implémenter alors l'algorithme de parcours en largeur à partir d'un sommet $r$. *Nota bene* : lorsque l'on fait un parcours en largeur, la racine du parcours est généralement n'importe quel sommet.

**e.** Tester l'algorithme sur quelques exemples de graphes, par exemple, le graphe de Petersen, et contrôler les valeurs des attributs *parent* et *distance* en fin d'algorithme.

**f.** Tester en particulier l'algorithme sur le graphe donné dans les fichiers `graph-003`. Qu'observe-t-on ?

**g.** Évidemment, le graphe n'étant pas connexe, à la fin du parcours en largeur, il reste des sommets verts, ce qui signifie que certains sommets (et donc aussi certaines arêtes) n'ont pas été examinés. Quand on veut faire un parcours complet du graphe, on doit donc relancer des phases de parcours en largeur à partir de racines restées vertes jusqu'à ce que tous les sommets soient rouges. Cela se fait en exécutant cet algorithme :

**pour tout** $r$ sommet du graphe $G$ **faire**
  **si** color($r$) = *Green* **alors**
    Parcours en largeur du graphe à partir du sommet $r$
  **fin si**
**fait**

Implémenter ce complément d'algorithme.

### *III. Implémentation de l'algorithme et applications*

#### Exercice 2. Test de connexité

On a donc remarqué qu'un graphe est connexe si et seulement si tout sommet peut être atteint par un chemin à partir du sommet 1 (ou de n'importe quel autre sommet, d'ailleurs). Cela se voit sur le tableau final du parcours en largeur : tous les sommets doivent avoir été visités : autrement dit, il n'y a pas de sommet vert.

Sur ce principe, écrire une fonction qui associe à un graphe $G$ une valeur booléenne qui vaut *true* si, et seulement si, le graphe $G$ est connexe. La tester sur quelques graphes non orientés.

#### Exercice 3. Détermination des composantes connexes d'un graphe

L'algorithme permettant de calculer les composantes connexes d'un graphe est assez simple à réaliser à partir d'un algorithme de parcours de graphe, quel qu'il soit : au démarrage de chaque parcours, on identifie la racine du parcours que l'on décide de choisir comme représentant de la composante connexe dans laquelle elle se trouve.

En faisant le parcours de graphe à partir de la racine $r$ du parcours, on marque tous les sommets rencontrés en leur associant cette racine ; lorsque l'on a terminé le parcours à partir de cette racine, tous les sommets rencontrés auront été marqués par $r$. Si le graphe n'est pas connexe, alors il reste des sommets qui n'ont pas été visités par l'algorithme de parcours, et il faut alors repartir de l'un de ces sommets, $r'$, comme racine d'un nouveau parcours et marquer par $r'$ tous les sommets rencontrés. À la fin de ce second parcours, s'il reste des sommets non encore visités, on recommencera. Etc.

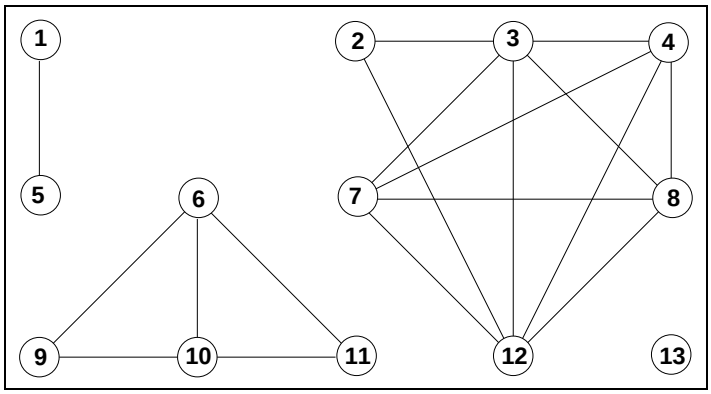


**Figure 1.** Un graphe non connexe

Pour un graphe d'ordre $n$, l'algorithme demandé construit donc une structure, typiquement, un tableau indexé par les sommets du graphe, noté $cc$, dont les valeurs seront également des sommets et tel que $cc(i)$ soit le représentant de la composante connexe du graphe dans laquelle se trouve le sommet $i$.

Par exemple, le graphe de la figure 1 n'est pas connexe, il comporte quatre composantes connexes, respectivement :

$$\{1,5\}, \quad \{2,3,4,7,8,12\}, \quad \{6,9,10,11\}, \quad \{13\}$$

Le tableau calculé par l'algorithme pourrait être le tableau suivant :

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $cc(x)$ | 1 | 2 | 2 | 2 | 1 | 6 | 2 | 2 | 6 | 6 | 6 | 2 | 13 |