TP 2
DFS, BFS et Composantes Fortement Connexes

Gilles Simonin

Résumé

Votre travail lors de ce TP est de mettre en place la version itérative du BFS (Parcours en largeur), la version récursive du BFS (parcours en profondeur), puis d'utiliser le DFS pour calculer les composantes fortement connexes.

1 Parcours d'un graphe en largeur

Vous pourrez coder vos algorithmes dans la classe *GraphToolsList* dans les méthodes de la classe. Cette classe permet d'utiliser la méthode principale "*main*" pour appliquer des algos à un graphe utilisant une représentation par listes d'adjacence.

Question 1

Vous allez coder votre premier algorithme de graphe. En reprenant l'algorithme itératif du parcours en largeur (BFS) vu dans le cours 2-théorie des graphes. Il s'agit du Parcours en Largeur qui consiste à partir d'un sommet aléatoire entre 0 et $n-1$ (vous pouvez choisir 0 pour vous simplifier la vie), puis de visiter d'abord les voisins direct à ce sommet, puis les voisins non encore visités de ces voisins, et ainsi de suite tant qu'il y des voisins non visités.

Cette méthode emploie une structure de list géré en FIFO (Queue) pour vous simplifier la vie. C'est comme une structure en tableau mais la taille de la liste est géré dynamiquement. il suffit d'ajouter et d'enlever les éléments.
Dans le code, ajoutez une variable noté "fifo" qui utilise une list de type "queue".
Pour ajouter un élément i, il suffit de d'appeler la méthode "fifo.add(i) ;".
Pour enlever le plus vieux élément ajouté, il suffit de faire l'appel de méthode "fifo.poll() ;" .

Question 2

Complétez votre code sur la méthode BFS() pour afficher chaque sommet visité lors de l'appel de la méthode "poll()". Vous ajouterez des tests dans votre méthode *main* pour appeler la méthode sur le graphe généré et vérifier si les sommets sont bien visités dans l'ordre d'un BFS.

Après avoir fait "Run", vous devriez avoir une liste dans la console indiquant l'ordre de visite des sommets (penser à espacer les entiers par un espace " " dans votre affichage). Puis dessinez le graphe sur une feuille et vérifiez votre résultat.

2 Parcours d'un graphe en profondeur

Question 3

Vous allez coder le parcours en profondeur (DFS) de manière récursive. Vous vous appuierez sur l'algorithme récursif proposé dans le cours sur la théorie des graphes.
Il s'agit du Parcours en Profondeur qui consiste à partir de sommets non visités, puis de parcourir le voisinage en prenant à chaque fois le premier voisin atteignable et en relançant une recherche de voisins.
Vous définirez deux méthodes "*explorerGraphe*" et "*explorerSommet*".

La première méthode lance l'exploration des sommets depuis un sommet de départ (cette fonction sera appelé tant qu'il y aura des sommets non visités. La deuxième méthode permet de parcourir en profondeur et de manière récursive le voisinage. Ajoutez un affichage dans la deuxième méthode chaque fois qu'un sommet est marqué comme visité.

Question 4

Complétez la méthode *main* en ajoutant des tests pour vérifier si l'appel de votre méthode "*explorerGraphe*" sur le graphe généré affichait bien chaque sommet visité.

3 Calcul des composantes fortement connexes

Nous nous intéressons maintenant à un algorithme qui consiste à parcourir le graphe avec un DFS et à trouver des composantes fortement connexes.
Cette approche vient de Kosaraju qui a prouvé que pour trouver des composantes fortement connexes, il suffisait de faire un parcours en profondeur en notant l'ordre de fin d'exploration des sommets visités, puis de refaire un DFS en suivant cet ordre de manière inversé sur le graphe inversé. C'est ce que vous allez faire en trois étapes.

Un graphe orienté $G = (V, A)$ est dit *fortement connexe* si, étant donné deux sommets x et y de V, il existe un chemin de x vers y ET il existe un chemin de y vers x dans G. La relation ainsi définie entre x et y forme une relation d'équivalence. Les classes d'équivalences induites par cette relation sur V forment une partition de V appelée *composantes fortement connexes*.

Question 5

Observez que le graphe partiel induit par les arcs parcourus lors d'un appel à l'algorithme `explorerGraphe`() est une forêt couvrante.

Nous souhaitons modifier l'algorithme afin de pouvoir distinguer trois classes de sommets : les non visités, les sommets en cours de visite et les sommets totalement visités. Un sommet s est *non visité* s'il n'a pas encore été examiné par la procédure `explorerSommet`(s) ; un sommet s est dit *en cours de visite* s'il a déjà fait l'objet d'un appel à la procédure `explorerSommet`(s) mais que cet appel ne s'est pas encore terminé ; enfin, un sommet s est dit *totalement visité* lorsque l'appel à `explorerSommet`(s) est terminé (c'est à dire que tous les successeurs de s ont été visités).

Question 6

Proposez une modification de l'algorithme intégrant cette classification dynamique à partir d'un tableau d'entier `visite`[], pour lequel $\forall x \in V$, `visite`$[x] \in \{0, 1, 2\}$ suivant l'état de x.

Maintenant, nous souhaitons être en mesure de *dater* les évènements subis par chaque sommet pour cet algorithme. Cela va permettre de créer un lien de *filiation* entre les sommets du graphe. En pratique, pour sommet $x \in V$, nous voulons un connaitre le moment où le sommet a été rencontré pour la première fois (noté `debut`$[x]$), et le moment où tous les successeurs de x ont été visités (noté `fin`$[x]$). Ainsi, on dira que y est un descendant de x dans notre parcours si et seulement si on rencontre y, et termine sa visite, avant d'avoir terminé de visiter x.

Question 7

À partir d'une variable globale incrémentée judicieusement et deux tableaux d'entiers `debut`[] et `fin`[], proposez une modification de l'algorithme. Quelle est la complexité de l'algorithme ainsi modifié ?

Question 9

Que peut-on dire de `debut`$[x]$, `fin`$[x]$, `debut`$[y]$ et `fin`$[y]$ pour deux sommets x et y du graphe G ? Vous supposerez dans votre observation que `debut`$[x] <$ `debut`$[y]$ (sinon inversez les notations), remarquez que deux cas peuvent se produire : y est un descendant de x ou non.

Question 10

Étant donné deux sommets x et y d'un graphe orienté $G = (V, A)$, observez que y est un descendant de x (il existe un chemin de x à y dans le parcours) si et seulement si, à l'instant `debut`$[x]$, il existe un chemin de x à y n'empruntant que des sommets z tels que `visite`$[z] = 0$.

Question 11

Complétez la méthode *explorerSommet* pour stocker dans une **variable de classe** notée "fin", de type List, l'ordre des sommets complètement explorés. Ici pour simplifier les choses, nous ajoutons juste dans la liste les éléments selon la fin de l'exploration. Utilisez l'appel de méthode 'fin.add(s)'.

Question 12

Maintenant que vous avez la liste 'fin' bien affichée dans la console après avoir lancé le "Run", vous allez l'utiliser pour calculer les composantes fortement connexes.

Question 13

Vous avez normalement codé la méthode "inverserGraphe" pour obtenir une nouvelle matrice.

Vous n'avez plus qu'à créer deux méthodes 'explorerGrapheBis' et 'explorerSommetBis' pour parcourir les sommets du graphe (qui a déjà été inversé) selon l'ordre donné par le tableau inversé 'fin'. En gros il faut copier et modifier ce que vous aviez fait dans la méthode 'explorerGraphe' mais cette fois ci en prenant les sommets de départ selon l'ordre dans 'fin'.

Vous chercherez à modifier les méthodes 'explorerGrapheBis' et 'explorerSommetBis' afin d'afficher les sommets atteint depuis chaque sommet choisi dans 'explorerGrapheBis'.