Université de Montpellier / Faculté Des Sciences

Algorithmes et structures des graphe orienté

Structures de données — HMIN 215

8 avril 2020

1 Notion de graphe orienté

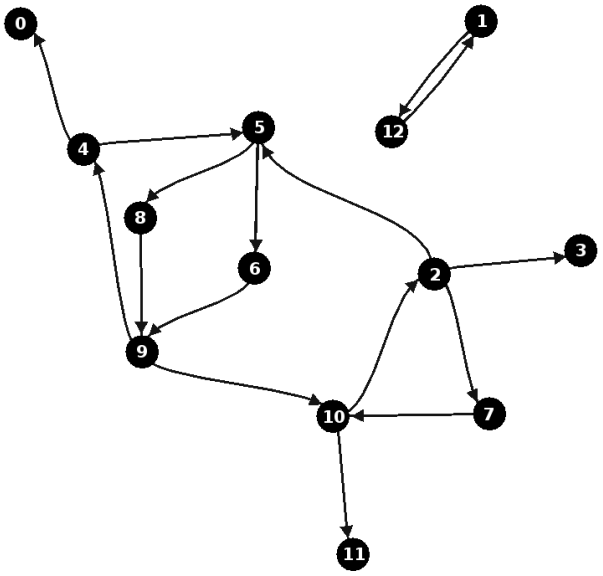


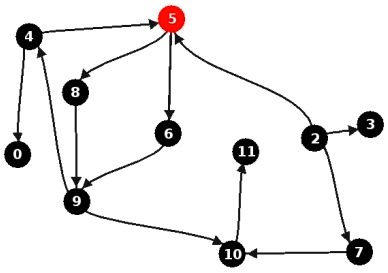
FIGURE 1 – Graphe dessiné par GraphStream

Un graphe orienté G = (V, E) est un couple composé d'un ensemble de noeuds / sommets V et d'un ensemble d'arcs E $\subseteq$ V $\times$ V. Cet ensemble d'arcs représente donc une relation entre les éléments de V. Un arc est un couple (s, t).

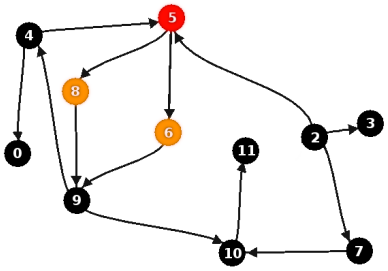
1.1 Parcours en largeur

Un parcours en largeur explore le graphe à partir d'un sommet donné (sommet de départ ou sommet source) en visitant d'abord tous les voisins de ce sommet, puis les voisins des voisins ... jusqu'à avoir parcourus tous les sommets qu'il est possible d'atteindre, sans visiter deux fois le même sommet.

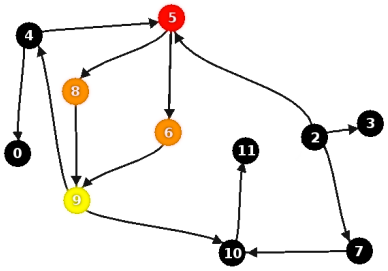
Un exemple valant mieux qu'un long discours :



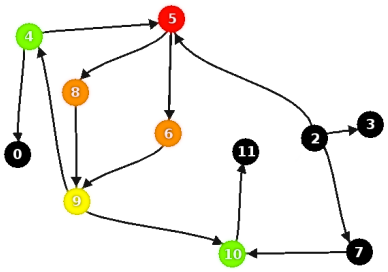
On choisit comme sommet de départ le sommet 5
état du parcours : [5]
sommets à parcourir : [8,6]



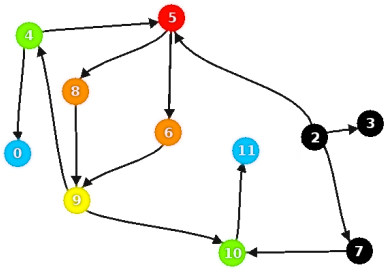
On ajoute les voisins de 5 : 8 et 6
état du parcours : [5,8,6]
sommets à parcourir : [9,9]



On ajoute les voisins de 8 et 6 : 9 Attention a ne pas ajouter 9 deux fois ! (il faut marquer que 9 a déjà été visité comme voisin de 8)
état du parcours : [5,8,6,9]
sommets à parcourir : [4,10]



On ajoute les voisins de 9 : 4 et 10
état du parcours : [5,8,6,9,4,10]
sommets à parcourir : [0,5,11]



On ajoute les voisins de 4 et 10 : 0 et 11. Attention a ne pas ajouter 5 où l'on va causer un parcours circulaire (il faut marquer que 5 a déjà été visité).
état du parcours : [5,8,6,9,4,10,0,11]
sommets à parcourir : []

Le parcours est fini, les sommets 2,3 et 7 ne sont pas atteignables car il n'y a pas d'arcs orientés vers eux. Vous pouvez choisir votre façon de gérer la liste des sommets à parcourir, par exemple vérifier que le sommet n'a pas été visité avant de de l'ajouter, ou qu'il n'est pas déjà présent dans la liste ...

2 Environnement de travail

Vous répondrez aux questions dans les classes du package *tp*, **GrapheBaseAlgo** pour l'étape 1 et **GrapheImplementationAlgo** pour l'étape 2. Ces classes dépendent des interfaces *IGraphe* et *IGrapheAlgo* qui garantissent respectivement la possibilité de construire et afficher un graphe ; et la présence de toutes les fonctions correspondantes aux algorithmes a développer.

Le Main du programme est contenu dans la classe Main, que vous pouvez modifier pour effectuer vos tests. Voici le contenu du Main en exemple :

```
1    IGraphe g = new GrapheBaseAlgo(13);
2    remplirGrapheTest1(g);
3    AfficheurGraphe.afficherGraphe(g);
4    testerTousLesAlgos((IGrapheAlgo)g);
```

La ligne 1 déclare votre nouveau **GrapheBaseAlgo** en tant qu'interface IGraphe. Ce graphe est ensuite passé a une fonction qui construit les arcs (Ligne 2). Nous vous recommandons de créer vos propres graphes, en définissant d'autres fonctions de remplissage, afin de tester vos algorithmes sur toutes sortes de cas. La ligne 3 fait appel à la classe **AfficheurGraphe** pour afficher votre graphe dans une fenêtre (voir Fig. 1). La fonction d'affichage prend en paramètre le type *IGraphe*. Enfin la ligne 4 fait appel à une fonction de test pour tester tous les algorithmes. Nous vous recommandons de créer vos propres test afin d'être sûr de tester tous les cas possibles.

2.1 GrapheBaseAlgo

la classe **GrapheBaseAlgo** hérite de la classe graphe base et vous fournit les fonctions suivantes :

- void ajouterArc(int numeroSortant, int numeroEntrant) : ajoute un arc depuis numeroSortant vers numeroEntrant
- ArrayList<Integer> arcsSortants(int numeroSommet) : retourne le numéro des sommets de tous les arcs sortants du sommet passé en paramètre
- int nombreDeSommets() : retourne le nombre total de sommets du graphe
- void marquerSommet(int numeroSommet) : place une marque sur le sommet
- boolean testSommetMarque(int numeroSommet) : teste si vous avez placé une marque sur le sommet
- void effacerMarques() : efface toutes les marques sur tous les sommets du graphe

3 Question

3.1 Étape 1

Complétez la classe **GrapheBaseAlgo** en utilisant les fonctions décrites ci-dessus, vous implanterez **OBLIGATOIREMENT** les deux fonctions suivantes :

- double calculerDensite() : calcule la densité du graphe, c'est à dire le nombre d'arcs sur le nombre maximum d'arcs possibles (pour connaître le nombre maximum d'arcs possible il faut se rappeler que tous les sommets peuvent être connectés à tous les sommets y compris soi-même, et que dans un graphe orienté l'arc a->b est différent de l'arc b->a)

- ArrayList<Integer> parcourEnLargeur(int numeroSommet) : effectue le parcours en largeur à partir d'un sommet, et renvoie la liste des numéros de sommets dans l'ordre parcouru.

Vous implanterez ensuite **AU MOINS DEUX** des fonctions suivantes :

- double calculerNombreMoyenVoisins() : calcule le nombre moyen de voisins
- ArrayList<Integer> predecesseursDuNoeud(int numeroSommet) : retourne l'ensemble des prédécesseurs d'un noeud (les origines des arcs entrants dans le noeud)
- boolean existeChemin(int sommetDepart, int sommetArrive) : retourne vrai si deux sommets sont reliés par un chemin orienté d'arcs
- IGraphe plusPetitGrapheSymetrique() : construit le plus petit graphe symétrique contenant ce graphe
- IGraphe grapheComplementaire() : construit le graphe complémentaire de ce graphe

Enfin, pour ceux qui ont du courage (**ENTIÈREMENT FACULTATIF**), essayez de faire la fonction :
<Integer> plusCourtChemin(int sommetDepart, int sommetArrive)
qui retourne le plus court chemin entre deux sommets.

3.2 Etape 2

Complétez maintenant la classe **GrapheImplementationAlgo**. Cette classe n'hérite plus de la classe GrapheBase, vous devez donc implémenter votre propre structure de graphe. Choisissez **une seule** des possibilités suivantes. Implantez ensuite le constructeur, les fonctions de IGraphe et si besoin les fonctions de marquage des sommets) :

- public GrapheImplementationAlgo(int nombreDeSommets)
- public void ajouterArc(int numeroSortant, int numeroEntrant)
- public ArrayList<Integer> arcsSortants(int numeroSommet)
- public int nombreDeSommets()

3.2.1 Possibilité 1 : Implémentation par une matrice d'adjacence

La matrice d'adjacence est une matrice de booléens M, carrée et indexée par les indices des sommets. $M[i][j]$ contient la valeur vrai si et seulement s'il y a un arc entre i et j. C'est la représentation à préférer lorsque le nombre de sommets est connu à l'avance et lorsque le graphe est dense (il contient beaucoup d'arcs).

3.2.2 Possibilité 2 : Implémentation par des listes de successeurs

Dans cette implémentation, on associe à chaque sommet la liste des sommets qui sont ses voisins par un arc sortant. On utilise préférentiellement cette représentation lorsque le graphe n'est pas très dense.

3.2.3 Possibilité 3 : Implémentation par représentation nodale

Cette représentation est inspirée du principe des listes chaînées. Dans cette implémentation, chaque sommet est représenté par une instance de classe, on associe à chaque sommet la liste des sommets qui sont ses voisins par un arc sortant, cette liste contient les instances (ou "pointeurs") représentant chaque sommet. Cette classe sommet peut aussi contenir

d'autres informations comme une marque, une valeur, ou encore la liste des prédécesseur. Cette représentation est proche d'une implémentation par liste de successeurs, et possède certains avantages quand on souhaite par exemple représenter un environnement.

3.2.4 Optimisation

Selon votre choix de structure, il est possible de rendre vos algorithmes plus performants.