Algorithmique
TP 8 : Algorithmes gloutons

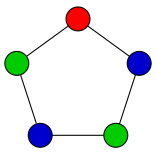
Le premier exercice de ce TP doit être déposé sur l'espace "Rendu TP8".

1 Coloration gloutonne d'un graphe ✍

Implémentation de la classe `Graph` Dans ce TP, nous vous fournissons une implémentation de la classe `Graph` qui est très proche de celle la semaine précédente. Les principales différences sont que notre graphe n'est pas orienté et que l'on ne met pas d'information (label) sur les arêtes. Du coup, la classe `Edge`, qui représentait une arête, n'est plus nécessaire : la liste d'adjacence contient désormais, pour chaque sommet, la liste de ses voisins sous forme d'entiers. Le fichier `Graph.java` contient les fonctions permettant de créer et manipuler des graphes. Vous ne devez pas changer ce fichier pour le TP, mais il est utile de le parcourir, car les méthodes qu'il définit vous seront nécessaires pour la suite. En voici notamment deux :

— `int getVertexSize()` renvoie le nombre de sommets du graphe.

— `List<Integer> getNeighbors(int i)` renvoie la liste des voisins du sommet `i`.



Le problème de la k-coloration La k-coloration d'un graphe consiste à attribuer une couleur à chacun de ses sommets, sans que deux sommets *voisins* n'aient la même couleur. On dispose de k couleurs différentes, codées par les entiers allant de 0 à $k-1$. Étant donné un graphe quelconque, déterminer le plus petit nombre de couleurs nécessaire pour le colorier est un problème difficile ; nous allons donc dans un premier temps nous intéresser à un problème plus simple : la coloration gloutonne d'un graphe.
Le degré d'un sommet (noté `degree`) est égal à son nombre de voisins au sein du graphe. Le degré maximal d'un graphe, que l'on notera Δ, est le plus grand degré parmi tous les sommets de ce graphe. Il est toujours possible de $(\Delta+1)$-colorier un graphe : en effet, supposons que l'on souhaite colorier les sommets les uns après les autres. Au moment de colorier le sommet `i`, on a déjà colorié une partie de ses voisins. Dans son voisinage, il y aura au plus `degree(i)` couleurs différentes. Or, `degree(i)`$\leq \Delta$, par définition de Δ. Du coup il existe une couleur, parmi les $\Delta+1$ couleurs disponible, que le sommet `i` peut prendre. En conclusion, si l'on dispose de $\Delta+1$ couleurs, il est possible de colorier "de manière gloutonne" les sommets les uns après les autres.

Programmer cet algorithme Pour programmer un algorithme de coloration gloutonne, le fichier `GraphColor.java` fournit une classe qui étend la classe `Graph`.

1. Pour encoder la coloration des sommets, on ajoutera comme attribut privé un tableau `Integer[] colors` qui contiendra les couleurs des sommets. Avant de les avoir attribuées, les couleurs seront initialisées à `null`.

2. Coder les nouveaux constructeurs de la classe, qui gèrent `colors`, ainsi que la mise à jour de la fonction `addVertex` pour que cela crée et mette à jour ce tableau.
3. Coder les méthodes d'accès et d'affectation pour le nouvel attribut. Coder également la fonction `Getcolor(int src)` qui renvoie la couleur du sommet `src`.
4. Implémenter les méthodes `degree` et `maxDegree`. La première calcule le degré d'un sommet, la seconde renvoie de degré maximal du graphe Δ (indication : utiliser la fonction `getNeighbors`).
5. Coder la fonction `goodColor` qui vérifie si la coloration est bonne. Elle doit vérifier que chaque sommet possède bien une couleur non `null`, que les couleurs sont toutes comprises entre 0 et Δ (inclus), et que pour chaque arête du graphe, les deux voisins reliés ont des couleurs différentes.
6. Coder la fonction `colorNeighbors` qui renvoie un tableau qui dit, à partir d'un noeud `src`, quelles sont les couleurs présentes dans son voisinage. Plus précisément, le tableau sera de taille $\Delta + 1$, et la case i aura la valeur `true` si et seulement si la couleur i apparaît dans le voisinage de `src`.
7. Coder la fonction `fillColors` qui $(\Delta + 1)$-colorie le graphe de manière gloutonne.

2 Arbre couvrant minimal : algorithme de Prim

Arbre couvrant minimal Pour un graphe non orienté pondéré G, un arbre couvrant minimal est un arbre A (c'est à dire un graphe sans cycle) qui couvre G (c'est-à-dire : A est un sous-graphe de G et tous les sommets de G sont adjacents à une des arêtes de A) et dont le poids total des arêtes est minimal par rapport à tous les arbres couvrants de G.

Algorithme de Prim L'algorithme de Prim, ci-dessous, permet de calculer un tel arbre, de façon gloutonne, en partant d'un sommet s choisi :

```
fonction prim(G, s)
    pour tout sommet t
           cout[t] := +infini
           pred[t] := null
    cout[s] := 0
    F := file de priorité contenant les sommets de G avec cout[.] comme priorité
    tant que F != vide
         t := F.defiler
         pour toute arête t--u avec u appartenant à F
             si cout[u] >= poids de l'arête entre les sommets t et u
                    pred[u] := t
                    cout[u] := poids de l'arête entre les sommets t et u
                    F.notifierDiminution(u)

    retourner pred
```

À faire Cette rédaction de l'algorithme de Prim, issue de Wikipédia, est différente de celle donnée en cours (elle est un peu plus « bas niveau »).
Donnez vous le temps de comprendre qu'il s'agit bien du même algorithme :

1. Par exemple que le couple de tableaux (`cout`, `pred`) est équivalent à un tableau de couples $(poids, sommet)$, et si on supprime les `null` et qu'on ajoute à chaque couple son indice dans le tableau, c'est aussi équivalent à une collection de triplets $(sommet, poids, sommet)$.

   À quoi correspond cette collection dans l'algorithme du cours ?
2. À quoi correspond alors la file `F` ?

Éléments techniques pour l'implémentation en Java Pour cet exercice, une classe `WeightedGraph` vous est fournie, pour représenter les graphes non orientés pondérés.
Par ailleurs, l'algorithme de Prim fait intervenir une file de priorité de sommets dont la politique de priorité est le coût actuellement connu pour les sommets.
En Java, on peut utiliser la classe `PriorityQueue`. On vous donne quelques indications :

— L'ajout d'un élément se fait via la méthode `offer`, la suppression de l'élément le plus prioritaire se fait par la méthode `poll` (qui retourne cet élément) et le test de file vide se fait par la méthode booléenne `isEmpty`.

— Pour créer une telle file de priorité, avec la bonne politique (en supposant que le tableau des coûts s'appelle `cost`), on peut utiliser l'expression suivante :

```
new PriorityQueue<Integer>(Comparator.comparing(x -> cost[x]))
```

Explication : le paramètre passé au constructeur est une fonction de comparaison qui lit le tableau des coûts.

— Pour « notifier » la file d'un changement de priorité, comme indiqué dans l'algorithme, il n'y a pas de méthode prévue dans la classe `PriorityQueue`. Mais, en réalité, il suffit de supprimer l'élément modifié de la file (méthode `remove`) et de l'ajouter à nouveau (méthode `offer`).

À faire Dans la classe `Prim` fournie, codez la méthode `spanningTree` qui implémente l'algorithme de Prim appliqué au graphe (à recouvrir) et au sommet (racine du futur arbre) passés en paramètre.
Le résultat est rendu sous la forme d'un tableau donnant le « père » de chaque sommet du graphe dans l'arbre ainsi construit. Pour la racine, on doit avoir la valeur `null` (pas de père).