Parcours de graphes

On révise ce qu'on a vu en première année en informatique de tronc commun : parcours de graphes et Dijkstra.

I Généralités sur les graphes

I.1 Sommets et arêtes/arcs

Définition 1

1. On appelle graphe non orienté un couple $G = (S, A)$ où
   - S est un ensemble fini appelé ensemble des sommets du graphe ;
   - A est un ensemble de paires de la forme $\{s_i, s_j\}$ avec $s_i \neq s_j \in S$ appelé ensemble des arêtes du graphe.
2. On appelle graphe orienté un couple $G = (S, A)$ où
   - S est un ensemble fini appelé ensemble des sommets du graphe ;
   - A est un ensemble de couples de la forme (s_i, s_j) avec $s_i \neq s_j \in S$ appelé ensemble des arcs du graphe.

Remarque 1 : On peut voir les graphes non orientés comme des cas particuliers des graphes orientés pour lesquels un arc de la forme (u, v) est présent si et seulement si l'arc (v, u) l'est aussi.

Remarque 2 : Comme S est fini, on peut à renommage près supposer avoir $S = \{0, 1, \dots, n-1\}$, où $|S| = n$. On le fait dans toute la suite.

Définition 2 : Voisins

Soit $G = (S, A)$ un graphe et $i \in S$. On appelle voisin de i un sommet j tel que $\{i, j\} \in A$ (pour un graphe non orienté) ou $(i, j) \in A$ (pour un graphe orienté).

Définition 3 : Graphe pondéré

Un graphe pondéré est un couple (G, p) où $G = (S, A)$ est un graphe (orienté ou pas) et $p : A \to \mathbb{R}$ est une application appelée pondération des arcs ou des arêtes.

Implémentation : on se concentrera sur l'implémentation par listes d'adjacences.

```
type 'a graphe = int list array;;
```

La longueur du tableau donnant le nombre de sommets, la $i^{\text{ième}}$ case du tableau donnant la liste des voisins de i. Le graphe est alors non orienté si et seulement si le tableau `g` précédent vérifie $\forall i, j \in S$, `List.mem i g.(j)` $\Leftrightarrow$ `List.mem j g.(i)`.

Dans le cas d'un graphe pondéré, on peut opter pour l'implémentation suivante :

```
type 'a graphe = (int*float) list array;;
```

La $i^{\text{ième}}$ case du tableau donne alors la liste des couples (j, ρ) pour lesquels j est un voisin de i et ρ la pondération de l'arc (i, j) ou de l'arête $\{i, j\}$.

Définition 4 : k-colorabilité

Soit $G = (S, A)$ un graphe à n sommets et $k \leqslant n$. On dit que G est k-coloriable lorsqu'on peut colorier les sommets de G à l'aide de k couleurs de sorte que G ne présente aucun arc ou aucune arête entre deux sommets de même couleur.

Le programme se concentre sur la notion de bicolorabilité, c'est-à-dire la 2-colorabilité.

Exemple 1 : Un graphe est dit complet lorsque tous ses sommets sont voisins les uns des autres. Un graphe complet à n sommets n'est k-coloriable pour aucun entier $k < n$.

I.2 Degré d'un sommet dans un graphe

Définition 5

Soit (G, A) un graphe et $i \in S$ un sommet de G.

- On appelle degré sortant de i et on note $\deg_s(i)$ le nombre d'arcs de la forme $(i, \_)$.
- On appelle degré entrant de i et on note $\deg_e(i)$ le nombre d'arcs de la forme $(\_, i)$.
- Pour un graphe non orienté, les deux notions coïncident. On parle alors de degré de i et on le note $\deg(i)$.

Lemme 1 Soit $G = (S, A)$ un graphe.

1. Si G est orienté on a $\sum_{i \in S} \deg_s(i) = \sum_{j \in S} \deg_e(j) = |A|$.
2. Si G est non orienté on a $\sum_{i \in S} \deg(i) = 2|A|$.

I.3 Chemins dans un graphe

Définition 6

Soit $G = (S, A)$ un graphe.

- On appelle chemin de longueur p dans le graphe G un $(p+1)$-uplet de sommets $(s_0, s_1, \dots, s_p)$ tel que, pour tout $i \in \{0, \dots, p-1\}$, s_{i+1} soit un voisin de s_i.
- Dans le cas où on a $s_0 = s_p$, le chemin est appelé un cycle.

Théorème-définition 1

On définit une relation $\mathcal{R}$ sur S par $i\mathcal{R}j \Leftrightarrow$ il existe un chemin de i vers j et il existe un chemin de j vers i.

La relation $\mathcal{R}$ est une relation d'équivalence.
Ses classes d'équivalence sont appelées les composantes fortement connexes de G.

Pour G non orienté, on parle simplement de composantes connexes.
Dans ce cas, la relation $\mathcal{R}$ est simplement $i\mathcal{R}j \Leftrightarrow$ il existe un chemin de i vers j.

Exemples 2 :

I.4 Arbres couvrants

Définition 7

On appelle arbre enraciné un couple (r, G), où G est un graphe non orienté connexe acyclique (sans cycle), et r un sommet de G.

On peut voir un arbre enraciné comme un arbre général au sens de la définition donnée en première année. La racine de cet arbre est r, les nœuds de profondeur 1 sont les voisins de r, les nœuds de profondeur 2 les voisins des voisins de r (qui ne peuvent pas être des voisins de r car le graphe est acyclique), etc. Tous les sommets apparaissent bien dans cet arbre par connexité.

Définition 8

Soit $G = (S, A)$ un graphe non orienté connexe.
On appelle arbre couvrant de G un graphe de la forme $G' = (S, A')$ avec $A' \subset A$ tel que G' soit acyclique.

Autrement dit, un arbre couvrant de G s'obtient à partir de G en retirant des arêtes jusqu'à obtenir un arbre, mais en retirant suffisamment peu d'arêtes pour que cet arbre passe par tous les sommets de G.

II Algorithmes de parcours

II.1 Parcours de graphe

Parcourir un graphe, c'est obtenir une liste d'une partie de ses sommets (si possible tous) en partant d'un sommet initial et en découvrant itérativement de nouveaux sommets en considérant des voisins de sommets déjà visités. Suivant la stratégie employée pour découvrir de nouveaux voisins, on peut explorer les sommets du graphe dans différents ordre. On appelle arborescence du parcours les listes de sommets découverts à chaque itération du parcours.

II.2 Parcours en profondeur (Depth First Search) et applications

Dans un parcours en prodondeur, on explore récursivement tous les sommets de chaque voisin avant de passer au voisin suivant (s'il n'a pas déjà été visité).

Exemple 3 :

Un parcours en profondeur est en fait **un parcours par pile** (ou par liste chaînée) : les sommets à voir sont stockées dans une pile, et chaque découverte d'un nouveau sommet se traduit par l'empilement de ses voisins sur la pile de sommets à voir.

Implémentation : on a besoin d'une information en plus par rapport au parcours en profondeur d'un arbre, qui est pour chaque sommet de savoir s'il a déjà été exploré à tout moment de l'exécution de l'algorithme. Pour que ceci puisse être obtenu rapidement, il convient d'utiliser .....

Remarque 3 : Sauf à modifier légèrement l'algorithme, le parcours en profondeur parcours seulement la **composante connexe** du sommet de départ du parcours. On peut d'ailleurs l'utiliser pour calculer une composante connexe.

Une application du parcours en profondeur est le **tri topologique** d'un graphe orienté acyclique.

Définition 9

On appelle tri topologique d'un graphe orienté acyclique $G = (S, A)$ une relation d'ordre total $\leqslant$ sur l'ensemble des sommets du graphe tel que, pour $u, v \in S$, s'il existe un arc de u vers v alors $u \leqslant v$.

En général, un tri topologique n'est pas unique, et l'existence n'est assurée que par le caractère acyclique orienté du graphe.

Remarque 4 : La relation "il existe un chemin de u vers v" vérifie la condition mais n'est pas un ordre total (c'est seulement un ordre partiel).

Pour décrire un ordre total sur l'ensemble des sommets de G, il suffit de donner la liste de ces sommets dans l'ordre total en question. Trier topologiquement les graphes acycliques orientés, c'est donc écrire une fonction `tri_topologique : graphe → int list` telle que, pour tout graphe orienté acyclique `g`, `tri_topologique g` est une liste de sommets de `g` telle que pour tous sommets u et v de `g`, u précède v dans la liste pour tout arc du sommet u au sommet v.

Exemple 4 :

On obtient un tri topologique en adaptant l'algorithme du parcours en profondeur : en effet, l'énuméreration des sommets dans l'ordre inverse de la numérotation postfixe d'un parcours en profondeur est un tri topologique. Plus clairement : pour chaque sommet pas encore vu, on explore en profondeur ses voisins pas encore vus **puis** on rajoute le sommet en tête de l'ordre. Allons-y :

II.3 Parcours en largeur et applications

Dans un parcours en largeur, on explore d'abord tous les voisins d'un sommet avant d'explorer les voisins des voisins (s'ils n'ont pas déjà été visités).

Exemple 5 :

Un parcours en largeur est en fait **un parcours par file** : les sommets à voir sont stockées dans une file, et chaque découverte d'un nouveau sommet se traduit par l'enfilement de ses voisins à la fin de la file de sommets à voir.

Implémentation : on peut reprendre l'implémentation du parcours en profondeur mais en utilisant une file à la place d'une pile ou d'une liste.

Une application du parcours en largeur est la recherche d'un **chemin de longueur minimale** dans un graphe non pondéré : étant donné un sommet source et un sommet cible, on effectue un parcours en largeur d'origine le sommet source mais on interrompt le parcours dès qu'on atteint le sommet but.

II.4 Implémentation efficace de l'algorithme de Dijkstra

Définition 10 Soit (G, p) un graphe orienté pondéré.
On appelle poids d'un chemin la somme des poids des arcs empruntés par le chemin.

L'algorithme de Dijkstra permet de déterminer les chemins de poids minimal à depuis un sommet source donné vers les autres sommets d'un graphe orienté pondéré. On reprend simplement l'idée du parcours en largeur pour déterminer un chemin de longueur minimale, mais puisqu'ici ce n'est plus la **longueur** du chemin que l'on souhaite minimiser mais son **poids**, on n'explore plus les sommets dans l'ordre du parcours en largeur en utilisant une file, mais dans l'ordre des poids des chemins en utilisant une file de priorité (avec pour priorités les distances minimales au sommet source en passant par des sommets déjà examinés, c'est donc une file de priorité pour laquelle prioritaire signifie minimal).

Exemple 6 :

Remarque 5 :
L'algorithme de Dijkstra est un algorithme glouton, il n'est correct que pour un graphe orienté pondéré **à poids positifs**.

Notons $n = |S|$ le nombre de sommets du graphe et $a = |A|$ son nombre d'arcs.

Pour l'impémentation, on peut procéder naïvement comme en tronc commun (à moins que vous n'ayez pas fait comme ça) : on stocke les distances à l'origine (et les prédécesseurs) de chaque sommet à l'aide d'un tableau de taille n^2, que l'on remplit ligne à ligne en effectuant n itérations dont le traitement nécessite dans le pire des cas n calculs de minimums, eux-mêmes de coût linéaire. La complexité est alors en $O(n^3)$. Faisons mieux.

En utilisant une file de priorité, chaque traitement de sommet demande au plus une insertion et une suppression dans une file de priorité de taille $\leqslant n$, ce qui coûte donc $O\big(\lg(n)\big)$, et une éventuelle mise à jour du poids courant, ce qui coûte $O(1)$. Les sommets sont traités à chaque apparition comme voisin d'un sommet précédent du parcours, donc $\sum_{s\in S} \deg(s) = 2|A|$ fois. On a donc une complexité en $O(a\lg(n))$.

Implémentation : on peut décrire l'algorithme comme suit.

```
Initialiser une file de priorité f avec (s, 0).
Initialiser un tableau des distances d avec 0 pour s et +∞ pour les autres sommets.
Initialiser un tableau des chemins avec [s] pour chaque sommet.
Tant que la file est pas vide, faire :
       Extraire le minimum de la file.
       Pour tous les sommets voisins du minimum :
              Mettre à jour les distances des voisins si nécessaire.
              Mettre à jour les chemins des voisins si nécessaire.
              Enfiler les voisins avec leurs distances comme priorités si nécessaire.
       Fin du pour.
Fin du tant que.
```

On voit que le "si nécessaire" revient à chaque examen d'un voisin : il s'agit de comparer l'actuelle distance associée à un sommet avec la distance obtenue en passant par le sommet en cours de traitement (le minimum de la file). On n'hésitera pas à écrire une fonction auxiliaire qui compare les distances et effectue les mises à jour si nécessaire (dans la littérature, cette opération est appelée *relâchement*).