TD Info 2 : Graphes

Michael Monerau

11 octobre 2010

Introduction

Quelques rappels :
- Retrouvez les TDs et corrigés sur http ://www.eleves.ens.fr/home/monerau
- Indentez votre code même si ça vous paraît inutile. Ca vous évite beaucoup d'erreurs de syntaxe et ça rend votre code plus clair.

1 Définitions

Les graphes sont des objets essentiels en informatique. Ils permettent de modéliser beaucoup de problèmes sous une forme commune. Ainsi, avoir de bons algorithmes pour les étudier est essentiel.

Nous allons voir la base de l'algorithmique des graphes dans ce TD.

Un graphe est une paire (V, E) où V est un ensemble (les *sommets*) et $E \subseteq V \times V$ est un ensemble de paires d'éléments de V (les *arêtes*).

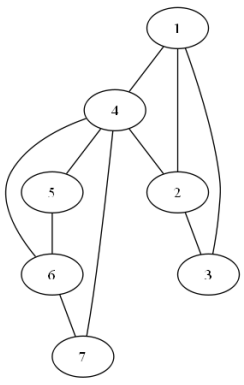

FIGURE 1 – Un exemple de graphe

Parfois on peut vouloir mettre un coût sur chaque arête, et donc on dispose également d'une fonction de poids $w : E \longrightarrow \mathbb{R}^+$.

2 Représentation interne

Il y a essentiellement deux représentations algorithmiques d'un graphe. Elles ont chacun leurs avantages et inconvénients, et le choix de l'une ou l'autre dépend des graphes qu'on considère.

2.1 Matrice d'adjacence

La matrice d'adjacence d'un graphe $G = (V, E)$ est la matrice $M_G \in \mathcal{M}_{|V|}(\mathbb{R})$ définie par :

$$\forall i, j \in [1, |V|], \quad M_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{sinon} \end{cases}$$

Clairement, une telle matrice définit uniquement un graphe et réciproquement.

La matrice d'adjacence du graphe de la figure 1 est :

$$M_G = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Dans le cas d'un graphe pondéré, on définit plutôt $M_{i,j} = w(i, j)$ quand $(i, j) \in E$.

Question 1. Définition

Définir cette matrice en Caml et la conserver dans une variable `grtest` pour tester les algorithmes que l'on va écrire. Le type du graphe doit être `int vect vect`.

Remarque : Attention, bien utiliser `make_matrix` et non `make_vect`.

Question 2. Degré

Le degré d'un sommet est le nombre d'arêtes qui y sont reliées.

Écrire une fonction `degre : int -> int vect vect -> int` telle que `degre i G` calcule le degré du sommet `i` dans `G`.

2.2 Listes d'adjacence

Ici on décrit un graphe $G = (V, E)$ par $|V|$ listes de sommets $(l_1, \ldots, l_{|V|})$. Chaque liste l_i est la liste des voisins de i, ie. la liste des $j \in V$ tels que $(i, j) \in E$.

Dans l'exemple 1, cette représentation serait :

$$L_G = \begin{cases} 1 : & [2, 3, 4] \\ 2 : & [1, 3, 4] \\ 3 : & [1, 2] \\ 4 : & [1, 2, 5, 6, 7] \\ 5 : & [4, 6] \\ 6 : & [4, 5, 7] \\ 7 : & [4, 6] \end{cases}$$

Question 3. **Définition**

Définir le graphe de l'exemple 1 sous cette forme en le stockant dans `grtest_listes`. Le type doit être `int list vect`.

Question 4. **Degré**

Réécrire la fonction `degre` mais cette fois pour traiter un graphe sous cette forme. Cela doit donner `degre_liste : int -> int list vect -> int`

2.3 Comparaison

En quoi ces deux représentations diffèrent ?

Question 5. **Comparaison**

Lorsqu'on dispose d'un graphe dense (beaucoup d'arêtes), quelle représentation est la meilleure d'un point de vue algorithmique ? Lorsqu'il est clairsemé (peu d'arêtes) ?

Question 6. **Complexités**

Quelle est la complexité de l'algorithme `degre` selon la représentation du graphe ?

Question 7. **Nombre d'arêtes**

Supposons qu'on sache que les graphes dont on dispose ont peu d'arêtes. Écrire une fonction `nombre_aretes : graphe -> int` qui compte le nombre d'arêtes du graphe (choisir la représentation appropriée).

3 Parcours de graphe

Lorsqu'on dispose d'un graphe, on veut pouvoir le parcourir pour visiter tous les sommets. Beaucoup d'algorithmes sur les graphes sont à base de parcours, et il est donc important de maîtriser ces briques de base.

L'objectif d'un parcours est d'appliquer une fonction (passée en argument) à chaque sommet. L'ordre de visite des sommets dépend de l'algorithme utilisé comme nous allons le voir.

3.1 Parcours en profondeur

3.1.1 L'algorithme

On commence d'un sommet de départ `depart` passé en argument. On prend un de ses voisins `i` et on le visite. On choisit ensuite un voisin `j` de `i` (différent de `depart`) et on le visite. Ainsi de suite, jusqu'à être bloqué et sans jamais revisiter un sommet déjà visité. Alors, on revient au dernier embranchement où on avait un choix, et on fait un autre choix. On fait cela jusqu'à ne plus avoir de choix.

Lorsqu'on n'a plus de choix, soit on a visité tous les sommets, soit non (il peut y avoir plusieurs composantes connexes). Dans ce cas, on recommence à partir de n'importe quel sommet non encore visité.

Une formulation plus consise et plus proche de l'algorithme : on visite un sommet, on le note comme visité, puis on visite récursivement tous les sommets adjacents non encore visités.

Voilà par exemple un parcours en profondeur sur le graphe de la figure 1 : `1 -> 2 -> 3, 4 -> 5 -> 6 -> 7`. Mais il n'y a pas unicité selon le choix lorsqu'on parcourt les voisins, voici un autre : `1 -> 4 -> 7 -> 6 -> 5, 3 -> 2`.

Question 1. **DFS (Depth First Search)**

Implémenter l'algorithme de manière récursive, sur un graphe sous forme de matrice d'adjacence.

Quelle en est la complexité ? Quelle serait la complexité pour un graphe sous forme de listes d'adjacence ?

3.1.2 Exemples d'algorithmes qui l'utilisent

Sortir d'un labyrinthe, détermination des composantes connexes, détection de cycle, arbre couvrant, 2-colorabilité, graphe bipartite, ...

3.2 Parcours en largeur

Ici l'ordre de visite des sommets n'est plus le même. Chaque fois que l'on visite un sommet, on visite ensuite *chacun de ses voisins* avant de passer aux voisins de voisins.

C'est-à-dire que plutôt que de suivre un chemin jusqu'au bout puis de revenir en arrière, on explore par étage depuis le sommet de départ : d'abord ceux qui sont à distance 1, puis 2, etc.

3.2.1 L'algorithme

Il s'agit donc ici de simplement visiter les sommets en se souvenant de leurs voisins. Puis après avoir fait tous les sommets d'un étage, on passe aux voisins qu'on a retenus et qui sont en attente.

Bien sûr, on ne revisite toujours pas des sommets déjà visités.

Pour l'implémentation, nous allons utiliser une pile FIFO (First-in First-Out). En Caml, on gère une telle pile de la manière suivante :

D'abord on importe le module `queue` par `#open "queue";;`.
Ensuite, là où on veut faire une pile, on écrit `let fifo = new () in ....`
Pour ajouter un élément à la file, on fait `add v fifo` où `v` est l'élément à ajouter.

Enfin, pour prendre l'élément en tête de la file (et l'enlever de la file), on fait `take fifo`.

Question 2. BFS (Breadth First Search)
Implémenter l'algorithme pour une représentation matricielle.

Quelle en est la complexité ? Quelle serait la complexité pour un graphe sous forme de listes d'adjacence ?

3.2.2 Exemples d'algorithmes qui l'utilisent

Plus court chemin entre deux sommets, beaucoup d'algorithmes utilisant une DFS mais ne reposant pas sur l'ordre de visite.

3.3 Parcours généralisé

À bien y regarder, ces deux parcours ne sont pas si différents l'un de l'autre. En effet, ils rentrent tout deux dans le cadre de l'algorithme de visite de graphe dit *généralisé*.

Remarquons que si on remplace la pile FIFO du BFS par une pile LIFO, on obtient alors la DFS.

3.3.1 L'algorithme

Plus généralement, la file de priorité qu'on utilise pour sélectionner le prochain voisin à visiter peut être choisie comme bon nous semble, et on accède alors à de nouveaux algorithmes sans effort.

3.3.2 Exemples d'algorithmes qui l'utilisent

Arbre couvrant de poids minimal sur un graphe pondéré (Algorithme de Prim), chemins les plus courts dans un graphe pondéré (algorithme de Dijkstra), ...

4 Clôture transitive : algorithme de Warshall

4.1 L'algorithme

Voyons maintenant un algorithme qui n'est pas à base de parcours.

Question 1. Théorique
Soit un graphe $G = (V, E)$ de matrice d'adjacence M_G. Que représente la matrice M_G^2 ? M_G^k ? $M_G^{|V|}$?

Question 2. Algorithme de Warshall
Utiliser les remarques précédentes pour calculer la clôture transitive d'un graphe G sous forme de matrice d'adjacence. On pourra donner la réponse sous la forme d'une matrice valant 1 en (i, j) si et seulement s'il existe un chemin dans G de i à j.

4.2 Commentaires

Si on regarde la complexité, on peut passer ici grâce à une astuce en $O(V^3)$. Ce n'est pas mieux que V DFS différentes en partant de chaque point. Cependant, l'algorithme est ici succint, pas besoin de machinerie DFS. De plus, l'idée de l'algorithme peut être réutilisée et a donné par exemple naissance à l'algorithme de Floyd-Warshall (calcul des chemins les plus courts entre paires de points d'un graphe pondéré).