▶ **MP & MP* – Option Informatique**
Neuxième TP Caml
Mercredis 27 février et 6 mars 2007

Graphes et algorithmes de graphes

Les graphes constituent une structure de données extrêmement générale, ce qui lui permet d'être utilisée dans de nombreux domaines de l'informatique. Une représentation classique des graphes serait semblable à une carte routière, les villes étant les sommets et les routes les arêtes joignant les sommets. Naturellement, il existe d'autres représentations des graphes, nettement utilisables pour les calculs. Nous allons durant ces deux heures étudier les deux méthodes les plus classiques pour représenter les graphes, ainsi que différents algorithmes de graphes extrêmement classiques.

1 Représentation des graphes

1.1 Définitions

Formellement, on dit que $G = (S, A)$ est un graphe si S est un ensemble quelconque (dont les éléments sont appelés « sommets »), et si A est constitué de couples de sommets (on a $A \subseteq S \times S$). On appelle ces couples « arêtes ». Habituellement, on note $n = |S|$ le nombre de sommets, et $m = |A|$ le nombre d'arêtes.
Si s est un sommet quelconque de S, le voisinage $\Gamma(s)$ de s est l'ensemble des sommets $s' \in S$ tels que l'arête (s, s') existe ($\Gamma(s) = \{s' \in S, (s, s') \in A\}$). Le degré de s est le nombre de sommets $s' \in S$ tels que l'arête (s, s') existe ($d(s) = |\Gamma(s)|$).
On peut classer les graphes en 2 grandes catégories :

(1) les graphes non-orientés, pour lesquels l'arête $a = (x, y)$ est la même que l'arête $a' = (y, x)$,

(2) les graphes orientés, pour lesquels l'arête $a = (x, y)$ est distincte de l'arête $a' = (y, x)$ (qui n'est pas toujours présente). Pour les graphes orientés, on parle d'« arcs » plutôt que d'arêtes.

On étend facilement les notions de degré et de voisinage aux graphes orientés en parlant de voisinages entrants et sortants, et de degrés entrants et sortants, notés respectivement $\Gamma^-(s), \Gamma^+(s), d^-(s), d^+(s)$.
Dans la suite de ce TP, nous nous attacherons essentiellement aux graphes orientés, et les sommets seront toujours numérotés $0, \ldots, n-1$. On remarquera qu'il est facile d'adapter un algorithme pour les graphes orientés aux graphes non-orientés en ajoutant pour chaque arc (x, y) un arc (y, x).
D'abord, nous allons étudier les deux principales façons de représenter un graphe en mémoire.

1.2 Listes d'adjacences

La représentation par listes d'adjacences d'un graphe $G = (S, A)$ consiste en un tableau Adj de $|S|$ listes, une pour chaque sommet u de G. Pour chaque $u \in S$, la liste Adj[u] est l'ensemble des sommets v tels que l'arc (u, v) existe.
Pour représenter les graphes sous forme de listes d'adjacences, nous utiliserons le type Caml suivant :

```
type 'a graph_list = {sommets : 'a vect; arcs : (int list) vect} ;;
```

▶ **Question 1** *Donnez la représentation Caml des différents exemples de graphes donnés ci-dessus.*

▶ **Question 2** *Ecrivez une fonction Caml* inDegree_list *qui calcule le degré sortant d'un graphe G donné en argument.*
De la même façon, écrivez une fonction Caml outDegree_list *, qui calcule le degré entrant d'un graphe G donné en argument.*

1.3 Matrice d'adjacences

La représentation par une matrice d'adjacences du graphe $G = (S, A)$ est donnée par une matrice $M = (m_{i,j})_{0 \leq i \leq n-1, 0 \leq i \leq n-1}$, telle que

$$m_{i,j} = \begin{cases} 1 & \text{si } (i, j) \in A, \\ 0 & \text{sinon.} \end{cases}$$

Pour représenter les graphes sous forme de matrices d'adjacences, nous utiliserons le type Caml suivant :

```
type 'a graph_mat = {sommets: 'a vect; arcs: int vect vect};;
```

▶ **Question 3** *Donnez la représentation Caml des différents exemples de graphes donnés ci-dessus.*

▶ **Question 4** *Ecrivez une fonction Caml* inDegree_mat *qui calcule le degré sortant d'un graphe G donné en argument.*
De la même façon, écrivez une fonction Caml outDegree_mat*, qui calcule le degré entrant d'un graphe G donné en argument.*

▶ **Question 5** *Comparez les deux représentations classiques de graphes (taille, vitesse, ...).*

2 Différentes méthodes parcours

2.1 Parcours en largeur

Le parcours en profondeur d'un graphe est l'un des algorithmes de parcours les plus simples. De plus, comme il utilisé dans de nombreux autres algorithmes (algorithme de Djisktra pour les plus courts chemins, de Prim pour l'arbre couvrant de poids minimum, ...), il reste intéressant à étudier.
Considérons un graphe $G = (S, A)$, et un sommet « origine » s_0, à partir duquel nous voulons parcourir le graphe, c'est-à-dire atteindre ses différents sommets (le plus rapidement possible, évidemment).
Au début de l'algorithme, on commence par visiter tous les sommets directement accessibles depuis s_0 ; ils forment un ensemble S_1. Ensuite, on parcourt tous les sommets (non déjà visités) directement accessibles depuis un sommet quelconque de S_1 ; ils forment l'ensemble S_2. De façon récursive, on visite tous les sommets directement accessibles (et non déjà visités) depuis un sommet quelconque du niveau S_k, et ils forment le niveau S_{k+1}.
Il faut pouvoir établir une distinction entre les sommets non encore découverts, les sommets en cours de visite et les sommets déjà visités. Pour cela, nous allons utiliser des étiquettes, et nous dirons que les sommets non découverts sont blancs, les sommets en cours de visite sont gris et les sommets déjà visités sont noirs. Quand un sommet non découvert est rencontré au cours de la recherche, il perd sa belle couleur blanche pour devenir gris.

```
type etiquette = blanc | gris | noir;;
```

Pour l'algorithme lui-même,

(1) on construit une liste L de sommets à visiter qui contient initialement que s_0.
(2) Ensuite, tant que cette liste L n'est pas vide et on prend son élément de tête u (en le supprimant de la liste).
(3) On regarde tous les sommets v adjacents à u et encore blancs,
(4) on les marque comme étant gris,
(5) on les rajoute à la fin de la liste L,
(6) on peut alors marquer u comme étant noir,
(7) on passe à l'élément suivant de la liste L.

▶ **Question 6** *Choisissez une représentation (matricielle ou listes d'adjacence) et implémentez en Caml une fonction* largeur g f *qui parcourt en largeur un graphe* g *donné en argument et qui à chaque sommet applique la fonction* f.

```
largeur : 'a graph -> ('a -> unit) -> unit <fun>
```

2.2 Parcours en profondeur

Comme son nom l'indique, le but d'un parcours en profondeur est d'abord de descendre le plus profondément possible dans le graphe.
La fonction profondeur sera une fonction récursive, qui part d'un sommet u, le marque en gris (ce qui signifiera « en cours de visite »), s'appelle récursivement sur tous ses descendants v qui sont encore blancs, et enfin marque en noir u.
▶ **Question 7** *Choisissez une représentation (matricielle ou listes d'adjacence) et implémentez en Caml une fonction* profondeur g f *qui parcourt en profondeur (qui l'eût cru ?) un graphe* g *donné en argument et qui à chaque sommet applique la fonction* f.

```
profondeur : 'a graph -> ('a -> unit) -> unit <fun>
```

3 Plus courts chemins

Une application classique (néanmoins réaliste) des graphes est la recherche de plus courts chemins. Si on représente une carte routière par un graphe (les sommets sont les villes et les arcs les routes), on peut se demander de quelle façon aller de Paris à Rome. Comme tous les chemins mènent à Rome, la difficulté n'est pas d'arriver à bon port, mais d'y arriver le plus rapidement.
Dans cette partie, nous utiliserons une représentation matricielle, en affectant des poids aux arcs. Ces poids correspondent à la durée du trajet, toujours positive*. Au lieu de mettre des* 0 *et des* 1 *dans la matrice, nous mettrons la durée du trajet (égale à $+\infty$ si la route - l'arc - n'existe pas).*

▶ **Question 8** *Quel peut-on avoir comme problème, si l'on autorise les poids des arcs à être négatifs ?*

Il existe différents algorithmes pour chercher (et même trouver) des plus courts chemins, la principale différence entre eux étant la gestion des poids négatifs. Nous allons regarder l'algorithme de Dijkstra.

(1) L'algorithme de Dijkstra tient à jour un ensemble E de sommets dont les poids finaux de plus court chemin à partir de l'origine s ont déjà été calculés.
(2) On prend le sommet u de $S - E$ le plus proche de s,
(3) on rajoute u à E,
(4) on met à jour sa distance à s et son père pour aller à s,
(5) on met à jour les distances des sommets adjacents à u,

(6) et on recommence l'étape 2 tant que $S - E$ n'est pas vide.

Nous avons donc besoin de
- un tableau `d` qui contient pour chaque sommet u la distance de u à s (initialisé à $+\infty$),
- un tableau `p` qui contient, pour chaque sommet u, son père pour aller à s,
- une liste E,
- une liste F qui contient tous les sommets de $S - E$.

▶ **Question 9** *Écrivez en Caml une fonction* `dijkstra g s` *qui calcule les plus courts chemins de* s *aux autres sommets. Cette fonction devra renvoyer deux tableaux, un contenant les prédécesseurs de chaque sommet, l'autre contenant la distance de chaque sommet à* s*.*

4 Tri topologique

▶ **MP & MP* – Option Informatique**
Neuxième TP Caml
Mercredis 27 février et 6 mars 2007

Graphes et algorithmes de graphes

Un corrigé

▶ **Question 5** La représentation par listes d'adjacences a l'avantage de consommer relativement peu de mémoire, surtout quand le graphe représenté est « creux » (quand il n'a que peu d'arcs). La quantité de mémoire demandée est en $O(max(|S|, |A|)) = O(|S|+|A|)$. En revanche, la recherche de l'existence d'un arc (u, v) donné n'est pas en temps constant, vu que l'on peut avoir à parcourir toute une liste de taille $|S|$.
La taille de la représentation matricielle est en $O(|S|^2)$ (donc souvent plus importante que la représentation par listes d'adjacences), mais l'existence d'un arc (u, v) donné se fait en temps constant.
Si le graphe considéré est non-orienté, nous avons $m_{i,j} = 1$ si, et seulement si, $m_{j,i} = 1$; la matrice est donc symétrique et il peut être intéressant de ne stocker que la moitié de la matrice (asymptotiquement, la complexité en place est identique, mais en pratique cela peut-être intéressant).

▶ **Question 8** S'il existe des poids négatifs sur les arcs, alors il peut exister un circuit de poids strictement négatif.
S'il existe un circuit de poids strictement négatif, alors il n'existe pas de court chemin, vu qu'il faudrait passer une infinité de fois par ce circuit pour avoir un chemin de poids $-\infty$.