Graphes

Étienne MIQUEY
etienne.miquey@ens-lyon.fr

L'objectif de ce TP est de voir différentes représentations d'un graphe en CamL, afin de pouvoir être à l'aise par la suite avec cet outil précieux. En guise de rappel, formellement un *graphe* est un couple $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, où $\mathcal{V}$ est l'ensemble des *sommets*, et $\mathcal{E}$ celui des *arêtes*. Un graphe peut-être *orienté*, ou non. On peut éventuellement rajouter une fonction de poids sur les arêtes, pour représenter une distance, un coût, un flot, ou je ne sais trop quoi d'autre. De temps à autre, on attribue aussi à un sommet une couleur, ou un potentiel. Bref, c'est un outil pratique et modulable, dont on se sert beaucoup en programmation.

1 Adjacence & plus court chemin

Un graphe sera ici représenté par sa liste d'adjacence [1], c'est-à-dire la donnée, pour chacun des sommets, des sommets adjacents. Afin de pouvoir s'amuser un peu par la suite, nous attribuerons à chaque arête un poids, strictement positif.

```
type node == int;;
type weight == int;;
type graph == (node*weight) list vect;;
```

Vous ne manquerez pas de remarquer que dans cette représentation, le graphe est orienté. Comme dans la plupart des cas, si l'on veut travailler avec des graphes non-orientés, il faut bien penser à compter les arêtes dans un sens et dans l'autre.

Question 1. Comme d'habitude, écrivez (rapidement) une fonction d'impression d'un graphe, qui vous simplifiera bien la vie.

```
print_graph : graph -> unit
```

Question 2. Afin de vous familiariser avec cette représentation, écrivez les primitives suivantes, qui ne devraient guère vous prendre de temps :

- `new_graph` qui prend en argument un entier `n`, et retourne un graphe de taille `n` le plus simple possible
  ```
  new_graph : int -> graph
  ```
- `set_edge` qui prend en argument un graphe, un couple de noeuds `(a,b)`, un poids, et crée l'arête de `a` vers `b` si elle n'existe pas déjà.
  ```
  set_edge : graph -> node -> node -> weight -> unit
  ```
- `new_cycle` qui prend en argument un entier `n`, et renvoie un cycle de taille `n`
  ```
  new_cycle : int -> graph
  ```

Nous allons maintenant implémenter l'algorithme de Dijkstra, qui permet de calculer le plus court chemin allant d'un sommet (que nous nommerons source) à un autre (que nous nommerons cible). Le principe de l'algorithme est relativement simple. On prend un ensemble $\mathcal{S}$ contenant au départ tous les sommets, et un tableau qui renseignera la meilleure distance à la source. Au départ, ce tableau contient $+\infty$ pour tous les sommets, sauf la source qui est à distance 0.

Question 3. Écrire une fonction `init` qui renvoie $\mathcal{S}$ et un tableau `distance` initialisés. Cette fonction pourra être du type suivant :

```
init : graph -> node -> int vect * bool vect
```

À un instant donné, on suppose avoir déjà calculé la distance à la source pour les sommets de $\mathcal{V}\backslash\mathcal{S}$. On cherche alors dans $\mathcal{S}$ le sommet le moins loin de la source parmi les voisins des sommets de $\mathcal{V}\backslash\mathcal{S}$. On met ensuite à jour la distance pour les voisins de ce sommet, et on le retire de $\mathcal{S}$. Puis on recommence l'opération jusqu'à ce que $\mathcal{S}$ soit vide.

Question 4. Écrire une fonction `find_min` qui prendra en argument le graphe, le tableau de distance, l'ensemble $\mathcal{S}$, et qui retourne le sommet ayant un voisin dans $\mathcal{S}$ qui soit le plus proche de la source.

```
find_min : graph -> int vect -> bool vect -> int
```

Question 5. Écrire une fonction `update` qui mettra à jour la distance pour les voisins d'un sommet.

```
update : graph -> int vect -> node -> unit
```

1. *On peut aussi utiliser la matrice d'ajacence, c'est juste un peu plus moche en espace*

Question 6. Écrire la fonction `dijkstra` qui prendra en entrée un graphe, un sommet source et un sommet cible, et renvoie la distance entre ces deux sommets. Quelle est la complexité totale de cet algorithme ?

```
dijkstra : graph -> node -> node -> int
```

Question 7 (Bonus)**.** Sauriez-vous prouver la validité de cet algorithme ? Avez-vous une idée qui permettrait en plus de connaître le plus court chemin ?

2 Enregistrement & coloriage

Nous allons désormais représenter un sommet par un enregistrement qui pointera vers ses voisins. Par la suite, nous nous intéresserons au coloriage du graphe, de fait, chaque sommet aura un champ couleur, qui pourra être vierge, blanc ou noir.

```
type color = none | white | black;;
type node = { id: int; mutable color : color; mutable neighbours : node list};;
type graph2 == node vect;;
```

Vous noterez que l'identifiant peut être redondant, notamment si la case `i` du graphe contient le noeud `i`. En effet, on aurait pu choisir aussi d'utiliser des listes de sommets au lieu d'un tableau, ce qui serait plus général, mais un peu plus pénible à manipuler par la suite. Mais vous avez tout à fait le droit de faire ce choix si vous le préférez.

Question 8. Reprendre les questions 1 et 2 avec les nouveaux types de données.

On dit qu'un graphe est *n-coloriable* lorsque l'on peut colorier l'ensemble des sommets du graphe avec n couleurs de telle sorte que chaque sommet ait ses voisins d'une couleur différente. Ce problème est, dans le cas général, NP-complet (*i.e.* il n'existe pas d'algorithme polynomial le résolvant). En revanche, il est facile pour $n=2$, ce à quoi nous allons nous intéresser.

Question 9. Écrire une fonction `inverse_color` qui renvera la couleur inverse de celle donnée en entrée, et lèvera une exception pour la couleur vierge.

```
inverse_color : color -> color
```

On appelle *coloriage épidemique* le coloriage qui consiste à donner à un sommet une couleur, et à tous ses voisins la couleur inverse, et ainsi de suite récursivement.

Question 10. Écrire une fonction `colour_node` qui prendra en argument une couleur `c` et un sommet `n`, coloriera ce sommet de cette couleur, et tous les sommets accessibles depuis celui-ci de manière épidémique, et renvera si le coloriage a réussi.

```
colour_node : color -> node -> bool
```

Question 11. Écrire une fonction `colour_graph` qui prendra en argument un graphe et testera s'il est 2-coloriable.

```
colour_graph : graph2 -> bool
```

Question 12. Testez votre fonction sur divers exemples. Avez-vous une idée de caractérisation simple des graphes 2-coloriables ?

3 Liberté & arbre couvrant

Il existe encore d'autres manières de représenter un graphe (notamment en le décrivant par la liste de ses arêtes, au lieu de celle de ses sommets), le tout est en général de savoir de quoi l'on a besoin pour notre problème en particulier. Si vous en êtes arrivé là, c'est que les graphes sont vos amis, et qu'on peut aller plus loin. En choisissant comme des grands la structure de données qui vous semble la mieux adaptée, je vous propose de coder l'algorithme de Prim, qui permet de trouver un arbre couvrant de poids minimal pour un graphe non-orienté.

Le principe est le suivant : on choisit un sommet au hasard pour former la base de l'arbre, que l'on va construire récursivement. À la fin de l'étape n, on dispose d'un arbre contenant n sommets et $n-1$ arêtes. On regarde alors toutes les arêtes reliant un sommet de l'arbre à un sommet qui n'est pas sur l'arbre, et l'on en choisit une de poids minimal, que l'on ajoute à l'arbre. L'algorithme se termine lorsque tous les sommets sont contenus dans l'arbre.

Question 13. Écrire une fonction `prim` qui prend en argument un graphe et renvoie un arbre couvrant de poids minimal.