TP 3 : Graphes et Dijkstra

L'objectif de ce TP est d'approfondir la définition et l'utilisation de types récursifs en fonctionnel. Pour cela, on définit deux nouveaux types, *graphe* et *tas*, et on les utilise dans le contexte de l'algorithme de recherche de plus court chemin dans un graphe de Dijsktra. La dernière partie du TP introduit l'utilisation d'interfaces graphiques en OCaml.

Type graphe

Un graphe G est un objet mathématique défini au minimum par un couple $G = (V, E)$, avec V un ensemble de sommets et E un ensemble d'arêtes. Dans la suite du TP, nous utiliserons le type *graphe* suivant :

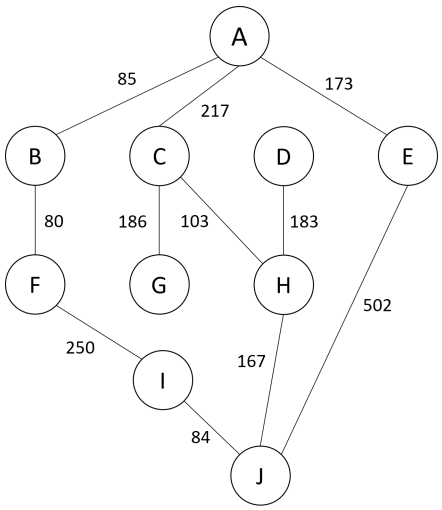
```
type 'a graphe = {
    sommets: 'a array;
    aretes: ((int * float) list) array
} ;;
```

NB : Nous utilisons ici la première manière de définir un type en OCaml, en décrivant sa structure. Nous avions utilisé cette même stratégie pour le type matrice du TP2. Ce type de définition est équivalent aux struct de C/C++.

Question 1. Décrivez rapidement la structure de données utilisée pour stocker l'ensemble V des sommets ? Même question pour l'ensemble E des arêtes.

Question 2. Le type *graphe* permet-il de représenter des graphes dirigés ?

Le graphe suivant est utilisé comme exemple pour expliquer le fonctionnement de l'algorithme de Dijkstra sur la page française de Wikipédia dédiée à ce même algorithme :



Question 3. En OCaml, créez une valeur de type graphe qui correspond au graphe précédent.

Algorithme de Dijkstra

L'algorithme de Dijkstra permet de découvrir les plus courts chemins qui permettent de naviguer d'un sommet de départ à tous les autres dans un graphe. Il consiste à maintenir à jour simultanément deux tableaux, le tableau des distances et celui des prédécesseurs. Les deux contiennent autant d'éléments qu'il y a de sommets dans le graphe. La valeur enregistrée à l'indice i dans le tableau des distances indique la distance du plus court chemin entre le sommet de départ et celui associé à i. Celle enregistrée à l'indice i dans le tableau des prédécesseurs indique l'indice du sommet à suivre depuis le sommet associé à i pour retourner au sommet de départ par le plus court chemin.

Pour un graphe et un sommet de départ donnés, le principe de l'algorithme de Dijkstra est le suivant :

1) Créer le tableau des distances et initialiser ses éléments avec la valeur infinie (*infinity* en OCaml) ;
2) Créer le tableau des prédécesseurs et initialiser ses éléments avec la valeur -1 ;
3) À l'indice du sommet de départ, mettre à jour le tableau des distances et celui des prédécesseurs respectivement avec la valeur 0 et avec l'indice du sommet de départ ;
4) Dresser la liste des sommets :
   - Si tous les sommets ont été traités (prédécesseur différent de -1), retourner les tableaux des distances et des prédécesseurs ;
   - Sinon :
     (a) Sélectionner le sommet qui a la plus petite valeur dans le tableau des distances (*sommet_dist_min*) ;
     (b) Aux indices des sommets voisins de *sommet_dist_min* (accessibles par une arête), si la distance enregistrée à l'indice de *sommet_dist_min* additionnée avec le poids de l'arête est inférieure à la distance enregistrée à l'indice du voisin, mettre à jour le tableau des distances et enregistrer *sommet_dist_min* comme nouveau prédécesseur du voisin ;
     (c) Reprendre à l'étape 4 en retirant *sommet_dist_min* de la liste des sommets.

Question 4. Implémentez en OCaml une fonction *dijkstra* qui prend en entrée une valeur de type *graphe* et l'indice du sommet de départ. L'étape 4 doit être implémentée sous la forme d'une fonction en récursion terminale. Pour les autres processus itératifs décrits, vous pouvez utilisez les fonctions du module *List*.

Question 5. Vérifiez que votre algorithme fonctionne, vous devriez obtenir les résultats suivants avec 0 comme sommet de départ :

indice	0	1	2	3	4	5	6	7	8	9
distance	0	85.	217.	503.	173.	165.	403.	320.	415.	487.
prédécesseur	0	0	0	7	0	1	2	2	5	7

Dijkstra avec tas

Un tas est généralement implémenté sous la forme d'un arbre binaire dont la structure respecte une seule règle : la valeur associée à un noeud parent doit toujours être plus prioritaire que celles de ses enfants. Ce que signifie "être prioritaire" dépend des utilisations. Il peut par exemple s'agir d'être inférieur ou supérieur dans les cas les plus simples. On utilise ce type de structure pour définir l'ordre de traitement d'un ensemble d'éléments. Pour cela, il suffit de créer le tas en respectant la règle puis de le vider en retirant systématiquement l'élément situé à la racine de l'arbre, qui est donc toujours le plus prioritaire.

Le type *tas* peut être défini de la manière suivante en OCaml :

```
type 't tas = Vide | Noeud of 't tas * 't * 't tas ;;
```

NB: en fonctionnel, on appelle ce genre de définition de type un ***variant*** *: les expressions séparées par les pipes (|) représentent toutes des variants valides de constructeurs pour le type. Chaque constructeur peut avoir un nom différent, commençant obligatoirement par une majuscule. Quand un constructeur possède des paramètres, leur types sont renseignés à la suite du mot-clé of, sous la forme d'un tuple quand il y en a plusieurs (en OCaml, chaque élément du tuple est séparé par *). Un type peut être récursif et défini de manière polymorphique, c'est-à-dire à l'aide d'un type générique qui sera inféré lors de la construction d'une expression. Cela permet de déclarer une seule fois la forme de la structure de donnée, qui sera ensuite utilisable avec n'importe quel autre type (entiers, chaînes de caractères, ...).*

Question 6. Récupérez les fonctions *ajouter*, *ajouter_plusieurs*, *supprimer_premier_noeud*, *vider* et *creer_tas* dans le fichier *tp3_code.ml* fourni avec ce sujet.

Question 7. Créez une nouvelle version de la fonction *dijkstra* qui utilise un tas plutôt qu'une liste dans l'étape 4 de l'algorithme.

Affichage graphique

Pour cette dernière partie du TP, on va appliquer la fonction *dijkstra* sur un grand graphe construit à partir d'un nuage de points aléatoire généré sur une interface graphique.

Pour charger le module graphique d'OCaml, vous devrez exécuter les lignes suivantes dans votre session interactive :

```
#load "graphics.cma";;
open Graphics ;;
```

Question 8. Dans le fichier *tp3_code.ml* fourni avec ce sujet, récupérez les fonctions suivantes :

- *genere_sommets* : génère un tableau de sommets dont les éléments ont pour type une paire d'entiers correspondant aux coordonnées x et y du point sur l'interface graphique. Ces coordonnées ne peuvent pas se situer dans les zones de l'interface définies par le paramètre *trous* ;
- *affichage_sommets* : ouvre l'interface graphique et affiche le tableau de sommets.

Question 9. Appelez correctement les fonctions précédentes pour générer une liste de sommets et l'afficher. Via le paramètre *trous*, définissez une zone rectangulaire et une zone circulaire dans lesquelles les sommets ne peuvent pas être générés.

Question 10. Récupérez les fonctions *genere_aretes* et *affichage_aretes* dans le fichier *tp3_code.ml*. À quelle condition une arête est-elle générée entre deux sommets ?

Question 11. Générez la liste d'arêtes du graphe et affichez-la avec les fonctions précédentes.

Question 12. Pour finir, récupérez les fonctions *dessine_chemin* et *affichage_chemins* dans le fichier *tp3_code.ml*. Utilisez-les pour afficher sur l'interface graphique le plus court chemin qui passe par un sommet choisi aléatoirement entre le premier et le dernier sommet du graphe.