TP D'OPTION INFORMATIQUE 2
Graphes

1 Algorithme de Kruskal

Du point de vue des graphes non orientés, un **arbre** est un graphe connexe acyclique. Un arbre est **couvrant** pour un graphe G si c'est un sous-graphe de G qui en contient tous les sommets.
Un graphe non connexe n'admet clairement pas d'arbre couvrant. Un graphe connexe admet toujours un arbre couvrant (il suffit de supprimer une arête d'un cycle tant qu'il y en a un).
Lorsque le graphe est pondéré, on peut attribuer à un arbre couvrant un poids qui est la somme des poids de ses arêtes. L'algorithme de Kruskal permet de calculer un arbre couvrant de poids minimal d'un graphe pondéré connexe :

```
kruskal(G)
    l'arbre A est initialement vide
    Chaque sommet est initialement isolé dans A
    soit L la liste des arêtes de G
    on trie L par poids croissants
    pour chaque arête (u,v) de L
        si find(u) et find(v) sont différents
            on ajoute (u,v) à A
            union(u,v)
    on renvoie A
```

`find(u)` calcule la composante connexe de `u` dans `A`, et `union(u,v)` fusionne les composantes connexes de `u` et `v` dans `A`. On parle de structure de données **union-find**.

1. Appliquer à la main cet algorithme sur le graphe

```
g = [|  [(1,1.) ; (2,0.) ; (4,5.)] ;
    [(0,1.) ; (2,2.) ; (3,3.) ; (4,6.)]  ;
    [(0,0.) ; (1,2.) ; (3,10.)] ;
    [(2,10.) ; (1,3.) ; (4,0.)] ;
    [(3,0.) ; (1,6.) ; (0,5.)] |]
```

2. Implémenter un algorithme de tri efficace.
3. Nous allons représenter les composantes connexes par un tableau, similairement à l'arborescence construite lors d'un parcours de graphe. Le **représentant** d'une composante connexe sera le sommet à la racine de l'arbre associé à cette composante connexe.
   (a) Écrire une fonction `find` prenant en argument un tel tableau et un sommet, et renvoyant le représentant de sa composante connexe.
   (b) Écrire une fonction `union` prenant en argument un tel tableau et deux sommets u et v, et fusionnant les composantes connexes de u et v.
4. Implémenter l'algorithme de Kruskal.
5. On peut optimiser la structure union-find en faisant en sorte que chaque sommet parcouru lors d'un appel à `find` prenne pour père le représentant de sa composante connexe. Implémenter cette optimisation.

2 Algorithme de Dijkstra

1. Quelle structure de donnée est utilisée dans l'algorithme de Dijkstra ?
2. Récupérer l'implémentation de cette structure de donnée dans le fichier TP graphe.ml disponible sur cahier-de-prepa.

3. Écrire une fonction `dijkstra` prenant en argument un graphe pondéré `g` sous forme de listes d'adjacences et un sommet `s` et renvoyant le tableau des distances à `s` ainsi que le tableau des prédécesseurs dans les plus courts chemins depuis `s`.
4. Écrire une fonction `chemin` prenant en argument un tel tableau des prédécesseurs et un sommet `u`, et renvoyant le chemin de `s` (la racine de l'arbre dans lequel apparaît `u`) à `u`, sous forme de liste.

   Par exemple, `chemin [|0; 4; 1; 4; 0|] 2` doit renvoyer `[0; 4; 1; 2]`.
5. Rappeler un exemple de graphe pondéré sur lequel l'algorithme de Dijkstra ne donne pas le résultat correct.

3 Algorithme A*

L'algorithme A^* est une variante de l'algorithme de Dijkstgra considérant un sommet cible c fixé et utilisant une heuristique h estimant la distance entre un sommet u et ce sommet cible c. Les différences entre les deux algorithmes sont :

- seul le sommet source est ajouté à la file de priorité initialement. Lorsque l'algorithme de Dijkstra modifie la priorité d'un sommet, l'algorithme A* ajoute le sommet dans la file s'il n'y est pas déjà présent (on pourra utiliser un tableau `marque` pour garder en mémoire les sommets présents dans la file) ;
- la priorité d'un sommet u dans la file n'est pas $d.(u)$ (la distance de s à u actuellement calculée) mais $d.(u) + h\,u\,c$ (on ajoute l'estimation selon l'heuristique h de la distance de u à c) ;
- si le sommet défilé est c, l'algorithme s'arrête en renvoyant $d.(c)$ ainsi que le chemin de s à c encodé dans le tableau des prédécesseurs ;
- si la file devient vide, l'algorithme s'arrête par une erreur signalant que c ne peut pas être atteint depuis s.

Selon l'heuristique utilisée, l'algorithme A^* ne renvoie pas toujours un chemin minimal de s à c. En contrepartie, l'heuristique peut permettre d'orienter la recherche dans la bonne direction au sein du graphe, et ainsi de réduire significativement le temps de calcul.

1. Écrire une fon ction `a_star` implémentant cet algorithme. On prendra en argument le graphe `g`, l'heuristique `h` (de type `int -> int -> int * float`), le sommet source `s` et le sommet cible `c`.
2. Tester la fonction précédente avec l'heuristique nulle. Vérifier que le résultat coïncide alors avec celui de l'algorithme de Dijkstra.
3. Créer un tableau associant à chaque sommet les coordonnées cartésiennes d'un point de $\mathbb{R}^2$, et tester l'algorithme A^* avec comme heuristique la distance usuelle dans le plan.