TP D'INFORMATIQUE 14
Recherche de plus court chemin

1 Algorithme de Dijkstra

On rappelle que l'algorithme de Dijkstra travaille sur un graphe (orienté ou non) *pondéré* avec des poids positifs, représenté par listes d'adjacences, et calcule les distances et plus courts chemins de tous les sommets accessibles depuis un sommet source donné. Son pseudo-code est :

```
Dijkstra(G,s):
    Les sommets sont initialement de distance infinie et n'ont pas de père
    s a pour distance 0
    on crée une liste L de tous les sommets
    tant que L est non vide:
        on sort de L le sommet u de distance minimale
        pour chaque successeur v de u:
            on relâche v depuis u, autrement dit :
            soit d' la distance de u + le poids w de l'arc (u,v)
            si d' est strictement inférieure à la distance de v:
                la distance de v devient d'
                le père de v devient u
    on renvoie les distances et les pères
```

1. Représenter sur papier le graphe orienté pondéré

   ```
   G1 = [[(1,10),(4,5)], [(2,1),(4,2)], [(3,4)], [(2,6),(0,7)], [(1,3),(2,9),(3,2)]]
   ```

2. Appliquer à la main l'algorithme de Dijkstra sur `G1` depuis le sommet source 0.
3. Implémenter cet algorithme en Python. Les distances et pères seront stockés en mémoire sous forme de listes dont les indices sont les numéros des sommets. On peut définir une valeur infinie en Python avec `float('inf')`.
4. Écrire une fonction `chemin` prenant en argument le tableau des pères et un sommet `u`, et renvoyant le chemin du sommet source à `u`, sous forme de liste.
5. Déterminer la complexité de cette implémentation de l'algorithme de Dijkstra, en fonction du nombre de sommets n et du nombre d'arcs m.
6. Proposer un exemple de graphe contenant des poids négatifs et pour lequel les distances et plus courts chemins renvoyés par l'algorithme de Dijkstra ne sont pas corrects.

2 Algorithme A*

L'algorithme A^* est une variante de l'algorithme de Dijkstra se concentrant sur un sommet destination `d` particulier, en se basant sur une heuristique permettant d'estimer la distance entre un sommet `u` et ce sommet `d`.
Un exemple d'heuristique est de calculer la distance euclidienne entre deux sommets, en supposant qu'ils ont une position dans le plan. Dans cette partie, on supposera qu'on a un tableau `pos` associant à chaque sommet un couple de coordonnées cartésiennes.

```
def h(u,d):
    x1,y1 = pos[u]
    x2,y2 = pos[d]
    return ((x2 - x1)**2 +  (y2 - y1)**2)**0.5
```

Le pseudo-code de A^* est :

```
A_etoile(G,s,d,h):
    Les sommets sont initialement de distance infinie et n'ont pas de père
    s a pour distance 0
    on crée une liste L contenant seulement s
    tant que L est non vide:
        on sort de L le sommet u dont la quantité distance + h(u,d) est minimale
        si u = d, on s'arrête en renvoyant la distance de d et le chemin de s à d
        sinon, pour chaque successeur v de u:
            on relâche v depuis u
            si on a diminué la distance de v et que v n'est pas dans L, on l'y ajoute
    on renvoie "d n'est pas accessible depuis s"
```

1. Implémenter cet algorithme en Python
2. Tester avec le graphe `G1`, les positions `pos = [(0, 0), (1, 2), (-3, -1), (4, -1), (2,2)]` et l'heuristique de distance euclidienne. Retrouve-t-on les mêmes chemins qu'avec Dijkstra ?
3. Si l'heuristique surestime la distance réelle de `u` à `d`, l'algorithme A^* peut renvoyer un chemin non optimal. Observer ce phénomène en modifiant les positions des sommets.

3 Application au jeu vidéo

Dans beaucoup de jeux vidéos, il est nécessaire de calculer rapidement la trajectoire que doit choisir une entité dans un environnement (*pathfinding*). L'objectif de cette partie est d'observer que l'heuristique de A^* permet de jouer sur ce compromis entre vitesse de calcul et qualité du chemin renvoyé.
Nous allons représenter l'environnement par une matrice dont les cases peuvent prendre les valeurs 0 (case libre) ou 1 (obstacle). Chaque case libre est reliée aux cases libres voisines (directes, pas en diagonale) par une arête de poids 1 (cette matrice représente donc un graphe, mais on notera bien que ce n'est **pas** une matrice d'adjacence). Un sommet sera identifié à son couple de coordonnées dans la matrice.

1. Écrire une variante de A^* `A_etoile_grille` travaillant sur une telle matrice. Par soucis d'efficacité, on pourra stocker les distances et pères calculés dans des dictionnaires, et stocker les sommets à traiter dans un `set` plutôt qu'une liste. Afin d'estimer le temps de calcul, la fonction renverra le nombre de passages dans le `while` en plus de la distance et du chemin pour aller à `d`.
2. Pour tester la fonction précédente, nous allons générer des environnements aléatoires. Écrire une fonction `carte` prenant en argument deux entiers `n`, `p` et un flottant $0 \leq$ `q` ≤ 1 et renvoyant une matrice de dimension $n \times p$ dont la case i, j vaut 1 avec probabilité $q\left(1 - \dfrac{4((i-(n-1)/2)^2 + (j-(p-1)/2)^2)}{(n-1)^2+(p-1)^2}\right)$ et 0 sinon (la probabilité d'obstacle est maximale au centre, où elle vaut q). On pourra utiliser la fonction `random.binomial` de `numpy`.
3. Tester A^* pour aller de $(0, 0)$ à $(99, 99)$ sur une carte 100×100 avec $q = 0.5$, pour les heuristiques suivantes :
   - heuristique nulle (ramène A^* à Dijkstra)
   - distance euclidienne
   - distance Manhattan
   - distance Manhattan multipliée par 1.1
   - distance Manhattan doublée

   Lesquelles renvoient le chemin optimal ? Comment se comparent-elles concernant le nombre d'itérations ?
4. Écrire une variante `tracer_progressif` de A^* réaffichant à chaque étape la matrice après une courte pause. On mettra des -1 dans les cases déjà visités, et des -2 dans les cases du chemin de `s` à `u`. On pourra utiliser `pause`, `imshow`, `clf` de `matplotlib.pyplot`.
5. Ajouter un système de tour par tour, des portes et des clés, des monstres et des trésors.