TP D'INFORMATIQUE 15
Pathfinding dans une grille

1 Algorithme A*

L'algorithme A^* est une variante de l'algorithme de Dijkstra se concentrant sur un sommet destination `dest` particulier, en se basant sur une heuristique permettant d'estimer la distance entre un sommet `u` et ce sommet `dest`.
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 h2(u,dest):
    x1,y1 = pos[u]
    x2,y2 = pos[dest]
    return ((x2 - x1)**2 +  (y2 - y1)**2)**0.5
```

Un autre exemple est l'heuristique nulle, qui renvoie toujours 0, et avec laquelle l'algorithme A^* se comporte comme Dijkstra :

```
def h0(u,dest):
    return 0
```

On rappelle le pseudo-code de A^* :

```
Fonction A_etoile(G,s,dest,h):
    D[s] vaut 0
    D[u] est infinie pour les autres sommets u
    chaque sommet u n'a initialement pas de père P[u]
    seul le sommet s est non marqué
    tant qu'il reste un sommet non marqué :
        soit u non marqué de (D[u] + h(u,dest)) minimal
        on marque u
        si u = dest :
            on s'arrête en renvoyant D[dest] et chemin(P, dest)
        sinon, pour chaque successeur v de u:
            soit d' = D[u] + le poids w de l'arc (u,v)
            si d' est strictement inférieur à D[v]:
                D[v] = d'
                P[v] = u
                v devient non marqué
    si on sort de la boucle principale, dest n'est pas accessible depuis s
```

1. Écrire une fonction `chemin` prenant en argument le tableau des prédécesseurs `P` et un sommet `u` accessible depuis le sommet source `s`, et renvoyant le chemin de `s` à `u`, sous forme de liste. Par exemple, `chemin([None,4,1,4,0], 2)` doit renvoyer `[0,4,1,2]`.
2. Implémenter A^* en Python.
3. Tester avec le graphe du TP précédent :
   `G1 = [[(1,10), (4,5)], [(2,1),(4,2)], [(3,4)], [(2,6),(0,7)],[(1,3),(2,9),(3,2)]]`,
   muni des positions spatiales `pos = [(0, 0), (0, 1), (0, 2), (1, 2), (1, 1)]`, et vérifier sur différents sommets destination que l'heuristique nulle et l'heuristique euclidienne donnent la même distance.
4. Si l'heuristique surestime la distance réelle de `u` à `dest`, l'algorithme A^* peut renvoyer un chemin non optimal. Modifier le tableau `pos` pour augmenter certaines distances euclidiennes entre sommets, de sorte que A* renvoie un chemin non optimal.

2 Application au pathfinding

Dans beaucoup de jeux vidéo, 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. On souhaite créer une telle grille de manière aléatoire. Écrire une fonction `carte` prenant en argument deux entiers `n`, `p` (≥ 2) 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
$$q\left(1 - \frac{4((i-(n-1)/2)^2 + (j-(p-1)/2)^2)}{(n-1)^2+(p-1)^2}\right)$$
et 0 sinon (la probabilité d'obstacle est maximale au centre, où elle vaut q). On pourra utiliser la fonction `random.binomial(n,p)` de `numpy` pour simuler une variable suivant une loi binomiale de paramètres n, p.
2. Afficher une matrice `M` générée par la fonction précédente en utilisant les instructions :
```
import matplotlib.pyplot as plt
plt.imshow(M)
plt.show()
```
3. Écrire une fonction `voisins` prenant en argument une telle matrice, et une case supposée libre (x, y), et renvoyant la liste des cases libres voisines de (x, y) (on prendra garde de ne pas provoquer d'erreur si (x, y) est sur un bord de la grille).
4. Écrire une variante de A^* `A_etoile_grille` travaillant sur une matrice. Comme les sommets ne sont plus des entiers, on va stocker les distances `D` et les prédécesseurs `P` dans des dictionnaires, dont les clés seront des sommets sous forme de couples, plutôt que des listes. Par souci d'efficacité, on stockera les sommets **non marqués** dans un `set`, qui implémente un ensemble et qui fonctionne comme un dictionnaire où les clés n'ont pas de valeurs associées. On donne les commandes utiles pour manipuler un `set` :
```
S = set()     #crée un set vide S
S.add(u)      #ajoute l'élément u au set S (ne change pas S s'il contient déjà u)
S.remove(u)   #supprime l'élément u du set S
len(S)        #nombre d'éléments dans le set S
for u in S:   #itère sur chaque élément u du set S
```
Afin d'estimer le temps de calcul, la fonction renverra également le nombre de passages dans la boucle principale en plus de la distance et du chemin pour aller à `dest`.
5. 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 en abscisse + distance en ordonnée)
   - distance Manhattan multipliée par 1.1
   - distance Manhattan doublée

   Lesquelles renvoient le chemin optimal ? Comment se comparent-elles en nombre d'itérations ?
6. 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 visitées, et des -2 dans les cases du chemin de `s` à `u`. On pourra utiliser `pause`, `imshow`, `clf` de `matplotlib.pyplot`.
7. Ajouter un système de tour par tour, des portes et des clés, des monstres et des trésors.