Master : Science des Données, Génie Logiciel et Réseaux (SDGL&R)

Année universitaire : 2023-2024

Module : Structures de données avancées(S1)

Pr. S. EL KASMI

TP : L'algorithme A*

L'algorithme A* est utilisé pour trouver le plus court chemin entre un point de départ et un point d'arrivée dans un graphe pondéré. Il est largement utilisé en intelligence artificielle et en résolution de problèmes de recherche de chemins.

L'idée de base de l'algorithme A* est d'estimer le coût total d'un chemin en combinant la fonction d'évaluation heuristique (estimation du coût restant pour atteindre l'objectif) et le coût déjà accumulé depuis le point de départ. L'algorithme utilise une file de priorité pour explorer les nœuds du graphe de manière sélective, en choisissant toujours le nœud avec le coût total le plus faible comme prochain nœud à explorer.

a. Etapes principales de l'algorithme

Voici les étapes principales de l'algorithme A* :

1. Initialisation : Définir le point de départ, le point d'arrivée et initialiser la file de priorité contenant uniquement le point de départ avec un coût total initial de 0.

2. Boucle principale : Tant que la file de priorité n'est pas vide, répéter :

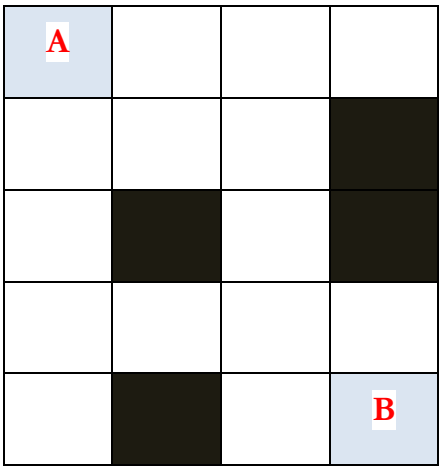
   - ✓ Extraire le nœud de coût total minimum de la file de priorité.
   - ✓ Vérifier si le nœud extrait est le point d'arrivée. Si oui, le chemin le plus court a été trouvé.
   - ✓ Sinon, générer les voisins du nœud extrait et mettre à jour leur coût total en utilisant la fonction d'évaluation heuristique et le coût accumulé actuel.

- ✓ Pour chaque voisin, si le voisin n'est pas déjà dans la file de priorité ou si le nouveau coût total est inférieur à l'ancien, mettre à jour le coût total du voisin et l'ajouter à la file de priorité.

b. Description de l'algorithme

Illustrons le fonctionnement de cet algorithme par un exemple :

Un individu cherche à se déplacer d'un point A à un point B sur un chemin avec des obstacles. La situation est représentée par les schémas ci-dessous :



Les cases bleues avec les étiquettes A et B représentent respectivement le point de départ (A) et l'arrivée(B). Les cases foncées représentent les obstacles, et les transparentes celles où il est possible de se déplacer.

L'enjeu ici est de trouver le plus court chemin menant à B à partir de A, l'individu ne pout pas emprunter les cases noires(obstacles).

Le caractère optimal de ce chemin dépendra du paramètre d'heuristique choisi. Dans notre cas, l'heuristique sert à fournir une estimation de la distance de restante à chaque étape de l'algorithme. Cette estimation dépend de la métrique choisie dans notre problème d'optimisation, et n'a pas à rendre une distance exacte.

Dans notre cas, on pourrait choisir comme heuristique la distance de Manhattan séparant A et B, ou la distance en ligne droite entre ces deux points, calculée à l'aide du théorème de Pythagore (voir définition de la fonction heuristique).

L'algorithme A* essaye par conséquent de minimiser la somme F = G+H, où G est la distance parcourue depuis le point de départ et H l'estimation de la distance restante à parcourir jusqu'à l'arrivée, en fonction de l'heuristique définie au départ de l'algorithme.

Ainsi, l'algorithme choisit à chaque étape la case lui permettant de se rapprocher de l'arrivée, et stocke dans une autre liste d'autres options moins optimales aux premiers abords. Cette liste secondaire permet de choisir une autre option au cas où l'algorithme rencontre un obstacle, ce qui garantit la justesse de l'algorithme.

Remarque : Le choix de l'heuristique d'estimation est crucial pour l'optimalité du chemin renvoyé. Si l'on surestime le chemin restant à parcourir à chaque étape, l'algorithme risque de ne pas renvoyer le chemin le plus court. En revanche, si l'on sous-estime l'heuristique, la complexité de l'algorithme augmentera, ce qui lui fera perdre son atout principal qu'est sa vitesse (l'heuristique nulle représente l'algorithme de Dijsktra). Si l'estimation est juste, l'algorithme n'explorera pas toutes les options possibles et renverra une solution optimale. On dit dans ce cas que l'heuristique fournie est admissible.

c. Programmation

Les paramètres d'entrée de la fonction « AEtoile » sont :

- ✓ Le dictionnaire Graphe qui associe à chaque sommet défini par ses coordonnées (x, y) la liste des cases adjacentes accessibles.

  Par exemple, le graphe précédent sera représenté par :

  Graphe={
  (0,0):[(0,1),(1,0)], (0,1):[(0,0),(0,2),(1,1)],
  (0,2):[(0,1),(0,3),(1,2)], (0,3):[(0,2)], (1,0):[(0,0),(2,0),(1,1)],
  (1,1):[(1,0),(0,1),(1,2)], (1,2):[(0,2),(1,1),(2,2)], (2,0):[(1,0),(3,0)],
  (2,2):[(1,2),(3,2)], (3,0):[(2,0),(4,0),(3,1)], (3,1):[(3,0),(3,2)],
  (3,2):[(3,1),(3,3),(2,2),(4,2)], (3,3):[(3,2),(4,3)], (4,0):[(3,0)],
  (4,2):[(3,2),(4,3)], (4,3):[(4,2),(3,3)]
  }

- ✓ Les coordonnées du sommet de départ
- ✓ Les coordonnées du sommet d'arrivée

On considère les fonctions suivantes :

- ✓ reconstruire_chemin(pred, dep, arr) : qui prend en paramètre le dictionnaire des prédécesseurs et les sommets de départ et d'arrivée. La fonction retourne une liste contenant le plus court chemin conçu par la fonction « AEtoile ».
- ✓ distance(a, b) : Qui calcule et retourne la distance euclidienne entre deux sommets.
- ✓ heuristic(a, b) : Qui calcule et retourne la distance de Manhattan entre deux sommets

Travail à faire :

Proposer une définition de la fonction AEtoile(Graphe, dep, arr) qui prend en paramètre un dictionnaire représentant le graphe étudié et deux tuples représentent respectivement les sommets de départ et d'arrivée.

```
from numpy import *
def heuristic(a, b):#fonction heuristique, distance de Manhattan
        return abs(a[0] - b[0]) + abs(a[1] - b[1])
def distance(a, b):#distance euclidienne entre deux points
        return ((a[0] - b[0])**2 + (a[1] - b[1])**2)**0.5
def reconstruire_chemin(pred, dep, arr):#fct pour reconstituer le
chemin entre les sommets "dep" et "arr"
        actuel = arr
        chemin = [actuel]
        while actuel != dep:
                actuel = pred[actuel]
                chemin=[actuel]+chemin
         return chemin
def AEtoile(Graphe,dep,arr):
        #Affectation des coûts inf à tous les sommets à
l'exception de "dep"
         G, pred = {},{}
        for noeud in Graphe:
```

```
                G[noeud]=inf#dictionnaire de stockage des coûts
d'accès aux nœuds(distance déjà parcourue)
                pred[noeud]=-1# dictionnaire de stockage des
prédécesseurs de chaque noeud
        G[dep] = 0
        #A compléter
        #
        #
        #



        #
        return G[arr], reconstruire_chemin(pred, dep, arr)
```

Des exemples d'utilisation de la fonction « AEtoile »

```
Graphe={
(0,0):[(0,1),(1,0)], (0,1):[(0,0),(0,2),(1,1)],
(0,2):[(0,1),(0,3),(1,2)], (0,3):[(0,2)], (1,0):[(0,0),(2,0),(1,1)],
(1,1):[(1,0),(0,1),(1,2)], (1,2):[(0,2),(1,1),(2,2)], (2,0):[(1,0),(3,0)],
(2,2):[(1,2),(3,2)], (3,0):[(2,0),(4,0),(3,1)], (3,1):[(3,0),(3,2)],
(3,2): [(3,1),(3,3),(2,2),(4,2)], (3,3):[(3,2),(4,3)], (4,0):[(3,0)],
(4,2): [(3,2),(4,3)], (4,3):[(4,2),(3,3)]
}

dep = (0, 0)
arr = (3, 1)
print(AEtoile(Graphe, dep, arr))
```

Résultat

```
(4.0, [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1)])
```

```
dep = (0, 0)
arr = (4,3)
print(AEtoile(G, dep, arr))
```

Résultat

```
(7.0, [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3), (4, 3)])
```