
INTELLIGENCE
ARTIFICIELLE

Table des matières

Introduction

L’intelligence artificielle est une science qui fait appel à plusieurs autres disciplines et à des concepts à comprendre. L’heuristique est l’un d’eux, et intervient en ce sens pour proposer des solutions acceptables. En effet, une heuristique, en optimisation combinatoire, théorie des graphes et théorie de la complexité, est un algorithme qui fournit rapidement une solution réalisable, mais pas nécessairement optimale, pour un problème d'optimisation complexe.

La recherche heuristique est basée sur des algorithmes de construction. On cite :

- Greedy Best-First Search
- Recherche glouton
- Algorithme A*

La méthode de recherche heuristique, contrairement à la méthode non informée, utilise l’information disponible pour rendre le processus vers la solution efficace.
Dans le cadre d’un projet d’implémentation d’un algorithme de recherche heuristique, nous avons choisi l’algorithme A* pour résoudre un problème de chemin le plus court.

I. Présentation du projet

Dans le cadre d'un projet d'IA, ce rapport présente des clarifications à propos du travail acharné. Le projet consiste à implémenter un algorithme A* pour trouver le chemin le plus court pour aller d'un point vers un autre.

1- Principe de l'algorithme A*

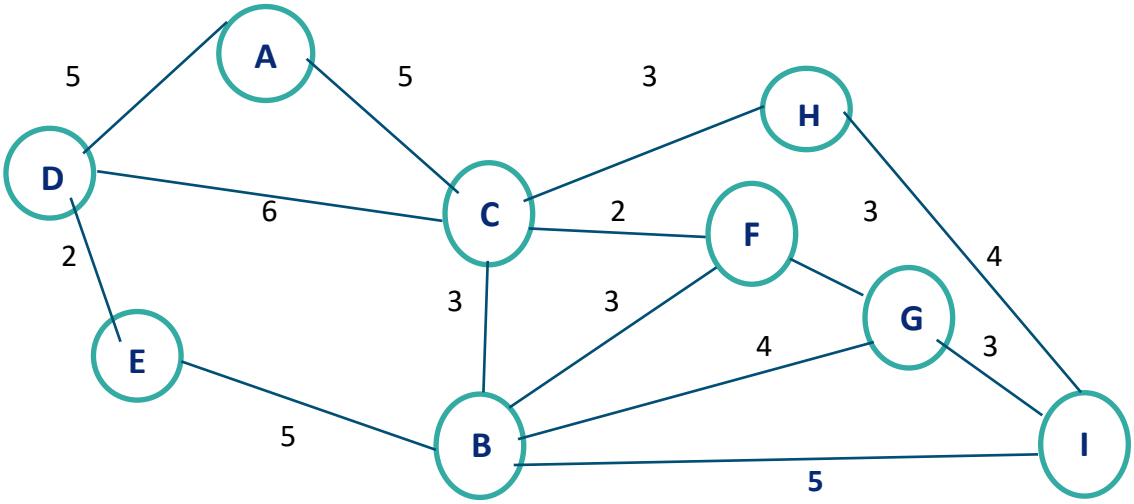
L'algorithme de recherche A* est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final. Il a été créé pour que la première solution trouvée soit l'une des meilleures, c'est pourquoi il est célèbre dans des applications comme les jeux vidéo privilégiant la vitesse de calcul sur l'exactitude des résultats. Il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. C'est un algorithme simple, ne nécessitant pas de prétraitement, et ne consommant que peu de mémoire.
On fixe en préalable ces paramètres nécessaires pour l'algorithme A*.

> START : nœud de départ
> GOAL : nœud d'arrivée
> h(i) : distance estimée d'un nœud i au nœud d'arrivée (i ∈ {1, 2, ..., N})
> g(i) : distance réelle d'un nœud i au nœud de départ (i ∈ {1, 2, ..., N})
> f(i) : somme des distances h(i) et g(i)

2- Application de l'algorithme

Ce projet sera appliqué sur un exemple de carte, on cherche à trouver le chemin le plus court de A vers I. Chaque connexion avec un nœud a un coût g(n)
En utilisant la méthode heuristique, ce tableau contient les données nécessaires pour trouver la solution à l'aide des deux heuristiques h1 et h2 :

Noeud	A	B	C	D	E	F	G	H	I
H1	10	5	5	10	10	3	3	3	0
H2	10	2	8	11	9	6	3	4	0
H3	10	5	8	11	10	6	3	4	0



II. Réalisation du projet

1- Implémentation de l'algorithme

```
#START my program

def algorithme_a_star(start, goal, graph,h):
    #Le chemin qu'on cherche pour aller de start à goal
    chemin=[]
    #les noeuds possibles (clès du graphe)
    pos=[*graph]
    #Ajouter le point de départ à notre chemin
    chemin.append(start)
    #La somme du cout de notre chemin choisit (uniform coast search (g))
    s=0
```

- Les inputs sont : point de départ (A dans notre cas) / point d'arrivée (I) / graph / la fonction heuristique h
- Output S va être le chemin qu'on cherche.
- La variable pos c'est une liste qui va contenir les clés du graphe. Ce dernier est sous forme de dictionnaire.

```
for i in range(len(pos)):
    #Commencer par notre  noeud de départ
    if pos[i]==start:
        #les cas possibles pour se déplacer à partir de notre point de départ avec leur cout
        test = graph[pos[i]]
        #trier seulement les noeuds proches du noeud courant
        state=[*test]
        compar={}
        for k in range(len(state)):
            compar[state[k]]=s+test[state[k]]+h[state[k]]
            node=[*compar]
            min=node[0]
            #Si on a plus d'un cas on doit choisir min(f) parmi eux
            if (len(node))>1:
                for m in range(1,len(node)):
                    if compar[min]> compar[node[m]]:
                        min = node[m]
            #Sinon le seul cas possible pour se dèplacer c'est node[0]
            else:
                min=node[0]
        #Ajouter à la fin le  noeud suivant à notre chemin
        chemin.append(min)
```

Pour aller du noeud A vers le noeud I on doit parcourir les valeurs de notre liste pos et choisir le bon chemin :

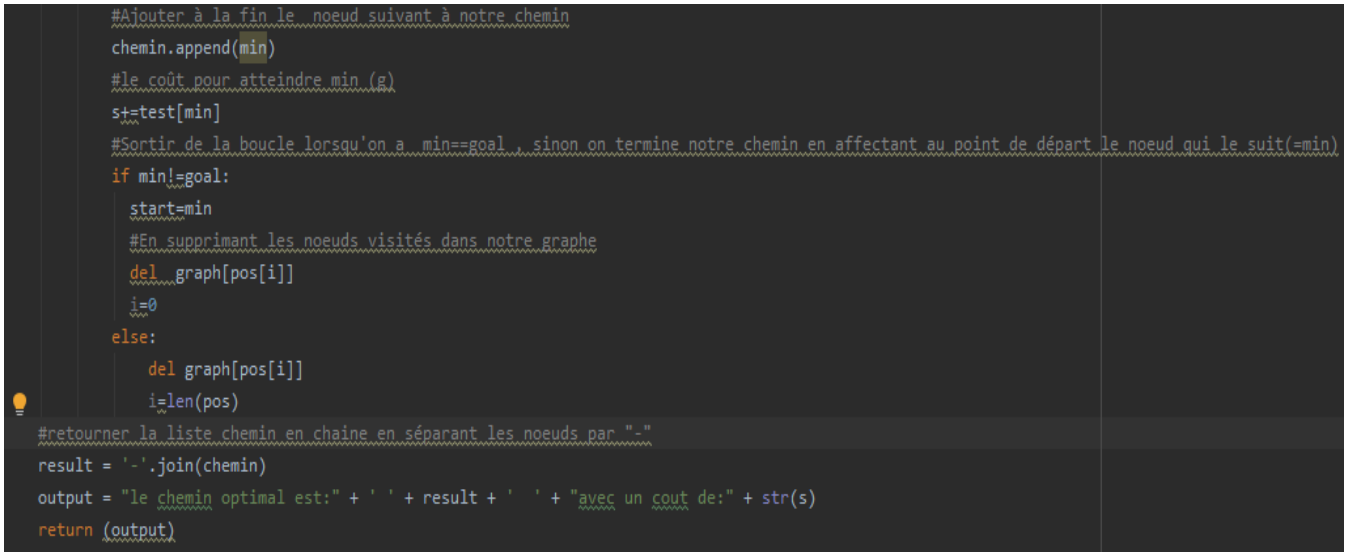
On établit une condition if pour qu'on puisse commencer par notre premier élément (qui est A dans notre cas)

- test : c'est un dictionnaire des nœuds proches (avec leurs valeurs de g) du nœud courant traité, **par exemple** si pos[i] va être A donc test[pos[i]] va être {'d':5,'c':5}
- state stockera les nœuds proches (clés de test) pour le cas du dernier **exemple** - state =['d','c']
- compar : c'est un dictionnaire qui va avoir comme clé les nœuds possibles et chaque clé a comme valeur sa fonction A* 'f' . **Exemple :** pour le cas de h=h1=> compar={'d' :15,'c' :10}

```
            node=[*compar]
            min=node[0]
            #Si on a plus d'un cas on doit choisir min(f) parmi eux
            if (len(node))>1:
                for m in range(1,len(node)):
                    if compar[min]> compar[node[m]]:
                        min = node[m]
            #Sinon le seul cas possible pour se dèplacer c'est node[0]
            else:
                min=node[0]
        #Ajouter à la fin le  noeud suivant à notre chemin
        chemin.append(min)
```

- Pour cette partie : on a node qui va contenir les clés de dictionnaire compar pour qu'on puisse après chercher le meilleur nœud qui a le min(f) . Si la taille de la liste node > 1 ; c'est à dire qu'on a deux cas possibles de chemin donc on doit par la suite choisir le nœud avec f minime **exemple** : node=['d','c'] on a len(node)=2 donc on va considérer que min ='d' et de comparer compar['d']=15 avec compar['c'] =10 donc min='c'

  Sinon, si on a seulement un élément dans la liste node ,il va être stocké dans la première position de notre liste donc on a un seul chemin possible et que min=node[0].



```
        #Ajouter à la fin le  noeud suivant à notre chemin
        chemin.append(min)
        #le coût pour atteindre min (g)
        s+=test[min]
        #Sortir de la boucle lorsqu'on a  min==goal , sinon on termine notre chemin en affectant au point de départ le noeud qui le suit(=min)
        if min!=goal:
          start=min
          #En supprimant les noeuds visités dans notre graphe
          del  graph[pos[i]]
          i=0
        else:
            del graph[pos[i]]
            i=len(pos)
    #retourner la liste chemin en chaine en séparant les noeuds par "-"
    result = '-'.join(chemin)
    output = "le chemin optimal est:" + ' ' + result + '   ' + "avec un cout de:" + str(s)
    return (output)
```

- Après qu'on a choisi le bon nœud (stocker dans la variable min) on va l'ajouter à notre liste chemin , en ajoutant aussi la distance entre nœud trouver et le nœud qui le précède à s pour trouver finalement le cout réel de notre chemin.

- Si le nœud trouvé est différé de notre objectif (nœud final) donc on doit terminer notre recherche en affectant au variable start notre nœud trouvé et supprimer le nœud déjà visité du graphe (donc au final(Quand on sort de notre boucle) la variable graphe va contenir seulement les nœuds non visitées.

  Sinon si le nœud trouvé(min) égal à notre objectif (nœud final) on doit supprimer le nœud visiter du graphe et affecter à la variable i la valeur len(pos) pour qu'on puisse sortir de la boucle.

- Finalement, on convertit la liste chemin en chaine de caractères result pour retourner le chemin le plus court traversé de A à I.
- Output va contenir : le chemin sous forme de chaine de caractère et son cout réel (g(I))

NOTRE CODE :

```
def algorithme_a_star(start, goal, graph,h):
    #Le chemin qu'on cherche pour aller de start à goal
    chemin=[]
    #les noeuds possibles (clès du graphe)
    pos=[*graph]
    #Ajouter le point de départ à notre chemin
    chemin.append(start)
    #la somme du cout de notre chemin choisit (uniform coast search (g))
    s=0
    for i in range(len(pos)):
        #Commencer par notre  noeud de départ
        if pos[i]==start:
            #les cas possibles pour se déplacer à partir de notre point de départ avec leur cout
            test = graph[pos[i]]
            #trier seulement les noeuds proches du noeud courant
            state=[*test]
            compar={}
            for k in range(len(state)):
                compar[state[k]]=s+test[state[k]]+h[state[k]]
                node=[*compar]
                min=node[0]
                #Si on a plus d'un cas on doit choisir min(f) parmi eux
                if (len(node))>1:
                    for m in range(1,len(node)):
                        if compar[min]> compar[node[m]]:
                            min = node[m]
                #Sinon le seul cas possible pour se dèplacer c'est node[0]
                else:
                    min=node[0]
            #Ajouter à la fin le  noeud suivant à notre chemin
            chemin.append(min)
            #le coût pour atteindre min (g)
            s+=test[min]
            #Sortir de la boucle lorsqu'on a  min==goal , sinon on termine notre chemin en affectant au point de départ le noeud qui le suit(=min)
            if min!=goal:
              start=min
              #En supprimant les noeuds visités dans notre graphe
              del  graph[pos[i]]
              i=0
            else:
                del graph[pos[i]]
                i=len(pos)
    #retourner la liste chemin en chaine en séparant les noeuds par "-"
    result = '-'.join(chemin)
    output = "le chemin optimal est:" + ' ' + result + '  ' + "avec un cout de:" + str(s)
    return (output)
```

2- Exécution

```
graph={
    'a':{'d':5,'c':5},
    'd': {'c': 6, 'e': 2},
    'c': {'b': 3, 'f': 2, 'h': 3},
    'e': {'b': 5},
    'f': {'b': 3, 'g': 3},
    'g': {'i': 3, 'b': 4},
    'b': {'c': 3, 'f': 3, 'g': 4, 'i': 5},
    'h': {'i': 4}
}

h1={
    'a':10, 'd':10,'c':5,'e':10,'h':3,'f':3,'b':5,'g':3,'i':0
}
h2={
    'a':10, 'd':11,'c':8,'e':9,'h':4,'f':6,'b':2,'g':3,'i':0
}

h3={
    'a':10, 'd':10,'c':8,'e':10,'h':4,'f':6,'b':5,'g':3,'i':0
}
#on applique à h1
print('A* avec h1:')
print(algorithme_a_star('a', 'i', graph,h1))
#on applique à h2
print('A* avec h3:')
print(algorithme_a_star('a', 'i', graph,h2))
#on applique à h3
print('A* avec h3:')
print(algorithme_a_star('a', 'i', graph,h3))
```

```
Python Console
A* avec h1:
le chemin optimal est: a-c-f-g-i  avec un cout de:13

>>>
```

```
Python Console
A* avec h2:
le chemin optimal est: a-c-b-i  avec un cout de:13
```

```
A* avec h3:
le chemin optimal est: a-c-h-i  avec un cout de:12

>>>
```

Conclusion :

A la fin, à rappeler que l'intelligence artificielle est « l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de simuler l'intelligence humaine. » L'IA contribue à résoudre des problématiques telles que la recherche du chemin. Ce travail présente une solution d'un chemin plus court d'un point de départ vers un point d'arrivée et cela à l'aide des algorithmes de recherche heuristiques. Plus particulièrement, l'algorithme A* qui est une méthode optimale et mémorisable. Et donc, l'implémentation du code reste la dernière étape après avoir choisi le langage.