Joris Monnet

HE-ARC 2020

INF3 dlm-b

Intelligence Artificielle

TP2 : A*

Sommaire :

Heuristiques :

Supposons que l'on veuille se rendre à la ville B. Pour tout nœud n, on va s'intéresser aux heuristiques suivantes
— h0(n) = 0
— h1(n) = "la distance entre n et B sur l'axe des x"
— h2(n) = "la distance entre n et B sur l'axe des y"
— h3(n) = "la distance à vol d'oiseau entre n et B"
— h4(n) = "la distance de Manhattan entre n et B"
Parmi ces heuristiques, lesquelles sont admissibles ?

Rappel (source : cours de M. Carrino & M. Ghorbel) :
Théorème : Une fonction heuristique est admissible si ∀ n, 0 ≤ h(n) ≤ h*(n) avec h*(n) = coût optimal réel de n au but

H_0(n) :
$0 \leq 0 \leq h^*(n)$: 0 est égal à la borne inférieure du théorème donc inférieur à h*(n). De ce fait cette heuristique est **admissible**

H_1(n) :
Cette heuristique étant une distance, celle-ci sera toujours positive ou nulle.
De plus, le projeté orthogonal sur l'axe des x de la distance Bn est compris entre 0(si B et n sont alignées suivant l'axe des y) et la distance Bn (si B et n sont alignés sur l'axe des x) or la distance Bn correspond au cout optimal réel de n vers B car dans notre repère cartésien et réel, la distance la plus courte entre deux points est le segment droit les reliant.
De ce fait : $0 \leq h1(n) \leq h^*(n)$, cette heuristique est donc **admissible**.

H_2(n) :
Raisonnement identique au précèdent :
Cette heuristique étant une distance, celle-ci sera toujours positive ou nulle.
De plus, le projeté orthogonal sur l'axe des y de la distance Bn est compris entre 0(si B et n sont alignées suivant l'axe des x) et la distance Bn (si B et n sont alignés sur l'axe des y) or la distance Bn correspond au cout optimal réel de n vers B car dans notre repère cartésien et réel, la distance la plus courte entre deux points est le segment droit les reliant.
De ce fait : $0 \leq h2(n) \leq h^*(n)$, cette heuristique est donc **admissible**.

H_3n) :
Comme dit auparavant, la distance à vol d'oiseau est dans notre repère le cout optimal pour aller de n vers B et inversement, en effet cela correspond au segment droit reliant ces deux points. De plus, étant une distance, elle est positive ou nulle.
$0 \leq h3(n)\ et\ h3(n) = h^*(n)$ Cette heuristique est donc **admissible.**

H_4n) :

La distance de Manhattan correspond à la somme des deux projetés orthogonaux de la distance optimale entre deux points. Or on remarque par les propriétés du triangle rectangle, que son hypoténuse est toujours plus petite que la somme de ces deux autres cotés où ici la distance à vol d'oiseau est l'hypoténuse et les autres cotés les projetés.

Exemple :

n(1,1) et B(0,0)

Distance de Manahattan = 1+1=2

Coût optimal = $\sqrt{2}$ < 2

Nous avons donc : $0 \leq h^*(n) \leq h4(n)$ dans cet exemple.

Pour aller plus loin : La distance de Manhattan est admissible uniquement lorsqu'il n'y a qu'un seul projeté, donc que B et n sont alignés sur un axe comme pour h1 ou h2.

Cette heuristique n'est donc **pas admissible**.

Implémentation A*

Mon projet contient :

- Les fichiers de données : connections.txt & positions.txt
- Main.py qui est le fichier à lancer pour démarrer l'application
- aStar.py qui implémente l'algorithme et les heuristiques
- City_Links.py qui implémente les classes City, Links et les fonctions qui lisent les fichiers de données

aStar.py :

Ce fichier contient les 5 heuristiques demandées. De plus elles sont placées dans un dictionnaire dicHeuristics avec comme clé leur numéro comme chaine de caractères afin de pouvoir les choisir dans l'interface graphique.

Il contient de même l'algorithme en lui-même. Celui-ci prend en paramètres une ville de départ, une d'arrivée, une heuristique et le dictionnaire contenant toutes les villes. J'utilise la classe heapq afin de donner des priorités aux nœuds à visiter, de cette façon la classe trie par elle-même les nœuds par priorité. Cette priorité est donnée à chaque nœud suivant l'heuristique choisie. Dans le cas où l'algorithme ne trouve pas de solutions, une exception est levée pour en avertir l'utilisateur.

City_Links.py :

Ce fichier contient la classe City, une ville à comme attribut sa position en x, en y, un parent (utilisé pour retrouver le chemin du A* dans le main), un nom et une liste de toutes les connections qu'elle possède avec les autres villes.

J'ai redéfini la fonction __str__ pour l'affichage, et __lt__ pour éviter un bug rencontré. La fonction getNeighbourLeafs permet d'avoir la liste de toutes les villes qui sont connectées à la ville concernée. La fonction getWeightOf() permet quant à elle de récupérer le poids de la connexion entre la ville est une ville de destination qui sont connectées entre elles. Enfin createLinks créer les connexions lors de la lecture des fichiers de données en remplissant la liste des connexions appartenant à cette ville.
La classe Links ne comporte qu'une ville de destination est le poids de la connexion.

Enfin, les trois dernières fonctions de ce fichier sont celles qui permettent de lire les fichiers de données, elles placent toutes les villes dans un dictionnaire, et créer les liens entre les villes.

Main.py :

Ce fichier contient l'interface graphique du programme. D'abord le programme affiche toutes les villes disponibles et en demande deux à l'utilisateur, une de départ et une d'arrivée. Ensuite il affiche toutes le heuristiques disponibles et l'utilisateur fait son choix. Puis l'algorithme est lancé et il affiche les résultats avec le nombre de nœuds visités et le chemin emprunté.

Expérimentation :

L'utilisation des différentes heuristiques a-t-elle une influence sur l'efficacité de la recherche ? (En termes du nombre de nœuds visités)

Oui : Exemple avec le trajet Berlin-Lisbonne, le nombre de nœuds visitées corresponds à « steps » :

```
Path : Berlin -> Lisbon found in 22 steps
----------------------------------------------------------------
START
----------------------------------------------------------------
- Berlin
- Hamburg
- Amsterdam
- Brussels
- Paris
- Madrid
- Lisbon
----------------------------------------------------------------
END
----------------------------------------------------------------
```

22 avec h0

```
Path : Berlin -> Lisbon found in 19 steps
----------------------------------------------------------------
START
----------------------------------------------------------------
- Berlin
- Hamburg
- Amsterdam
- Brussels
- Paris
- Madrid
- Lisbon
----------------------------------------------------------------
END
----------------------------------------------------------------
```

19 avec h1

```
Path : Berlin -> Lisbon found in 25 steps
----------------------------------------------------------
START
----------------------------------------------------------
- Berlin
- Hamburg
- Amsterdam
- Brussels
- Paris
- Madrid
- Lisbon
```

25 avec h2

```
Path : Berlin -> Lisbon found in 24 steps
-----------------------------------------
START
-----------------------------------------
- Berlin
- Hamburg
- Amsterdam
- Brussels
- Paris
- Madrid
- Lisbon
```

24 avec h3

```
Path : Berlin -> Lisbon found in 10 steps
-----------------------------------------
START
-----------------------------------------
- Berlin
- Hamburg
- Amsterdam
- Brussels
```

10 avec h4

Pouvez-vous trouver des exemples où l'utilisation de différentes heuristiques donne des résultats différents en termes de chemin trouvé ?

Oui, exemple avec Paris-Prague :

H0 :

```
Path : Paris -> Prague found in 13 steps
-----------------------------------------
START
-----------------------------------------
- Paris
- Brussels
- Amsterdam
- Munich
- Prague
-----------------------------------------
END
```

H4 :

```
Path : Paris -> Prague found in 6 steps
---------------------------------------
START
---------------------------------------
- Paris
- Brussels
- Amsterdam
- Hamburg
- Berlin
- Prague
---------------------------------------
END
---------------------------------------
```

Dans un cas réel, quelle heuristique utiliseriez-vous ? Pourquoi ?

J'utiliserais h3, la distance à vol d'oiseau dans la plupart de mes tests est plus performante que les 3 premières heuristiques. H4 n'est pas admissible et donc peux ne pas trouver de solution, ce n'est donc pas une solution envisageable.

Aller plus loin : chercher la définition d'heuristique "consistente" ou "monotone" :

Formally, for every node *N* and each successor *P* of *N*, the estimated cost of reaching the goal from *N* is no greater than the step cost of getting to *P* plus the estimated cost of reaching the goal from *P*. That is:

$h(N) \leq c(N, P) + h(P)$ and
$h(G) = 0.$

where

- *h* is the consistent heuristic function
- *N* is any node in the graph
- *P* is any descendant of *N*
- *G* is any goal node
- c(N,P) is the cost of reaching node P from N

Source : https://en.wikipedia.org/wiki/Consistent_heuristic

Quel est son impact sur les performances de l'algorithme A ?*

L'algorithme A* sera plus performant en utilisant de telles heuristiques. En effet cela permettra de toujours avoir le plus petit cout possible pour arriver à un nœud donné par lequel passe l'algorithme, comme dans une best-first search sur le graphe en utilisant l'algorithme de Dijkstra.

Si vous assurez à votre algorithme une heuristique monotone, comment pourriez-vous améliorer votre de code ?

On passerais de cela :

```
if neighbourLeaf not in totalWeight or tempWeight < totalWeight[neighbourLeaf]:
    totalWeight[neighbourLeaf] = tempWeight
    neighbourLeaf.parent = current

    priority = h(neighbourLeaf, cityB) + tempWeight #use heuristic to change priority in heapq
    heapq.heappush(frontier,(priority,neighbourLeaf))
```

A ça :

```
for neighbourLeaf in current.getNeighbourLeafs(dicCity):
    tempWeight = totalWeight[current] + current.getWeightOf(neighbourLeaf)

    #check for each neighbour Leaf if it come closer to destination or not by
    #or if the algorithm find a smaller path from source to this leaf
    if neighbourLeaf not in totalWeight:
        totalWeight[neighbourLeaf] = tempWeight
        neighbourLeaf.parent = current

        priority = h(neighbourLeaf, cityB) + tempWeight #use heuristic to chan
        heapq.heappush(frontier,(priority,neighbourLeaf))
```

On n'aurait plus besoin de vérifier si le chemin qu'on vient d'emprunter est le plus cours pour atteindre ce nœud.

Parmi les 5 heuristiques ci-dessus, il y en a des monotones ? Si non, proposer une heuristique monotone pour notre problème du voyageur (pas besoin de l'implémenter)

Oui, les 4 premières heuristiques sont monotones, seul la distance de Manhattan ne l'est pas (une heuristique consistante est forcément admissible). En effet, le cout pour aller de nœud en nœud est toujours positif donc $h_{0->3}(n) \leq h^*(n+1) + c(n,n+1)$.