M1 IRSI

4AI04 Intelligence Artificelle

Compte-Rendu TP : Algorithmes de Recherche

Auteur :
Stéphane Sobucki
3300032

Auteur :
Jieyeon Woo
3521100

2018-2019

Table des matières

0 Introduction

Nous allons, dans ce TP, implémenter différents algorithmes de recherche de chemin dans un graphe pour permettre à un robot d'atteindre un but final depuis sa position initiale en évitant les obstacles au long de sa trajectoire, dans un espace 2D discret. Nous allons discuter de la performance des différents algorithmes mis en place (complexité en espace, complexité en temps et chemin le plus court).

1 Préparation Théorique

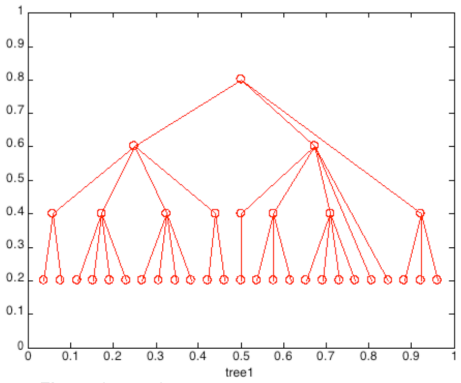

Figure 1 : tree1

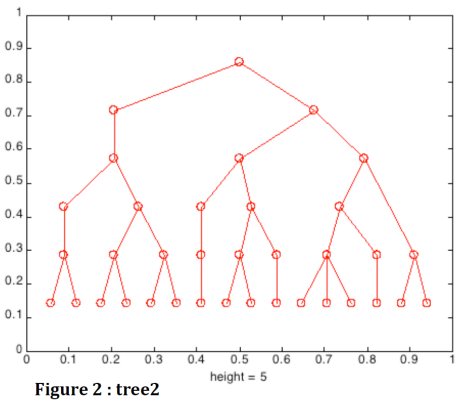

Figure 2 : tree2

Figure 3 :

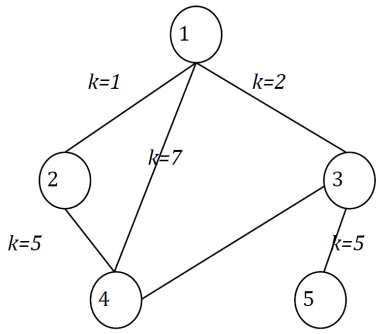

FIGURE 1 – Graphes

1.1 Complexité en Espace

Q1) Selon le critère de complexité en espace, on peut déduire l'algorithme de recherche non-informé qui est le plus adéquat pour chaque graphe.

On sait que la complexité en espace de l'algorithme en largeur et de l'alogrithme en profondeur. L'algorithme en largeur : $O(b^{d+1})$ L'algorithme en profondeur : $O(bm)$ avec b : le nombre de branches, d : la profondeur et m : la longueur maximale.

Pour la figure 1 :tree1, on a $b = 5$, $d = 2$ et $m = 3$, qui nous donne : $O(b^{d+1}) = O(5^{2+1}) = O(5^3) = O(125)$ en largeur et $O(bm) = O(5 * 3) = O(15)$ en profondeur. Donc, l'algorithme du profondeur est plus adéquat pour la figure 1.

Pour la figure 2 :tree2, on a $b = 3$, $d = 4$ et $m = 5$, qui nous donne : $O(b^{d+1}) = O(3^{4+1}) = O(3^5) = O(248)$ en largeur et $O(bm) = O(3 * 5) = O(15)$ en profondeur. Donc, l'algorithme de profondeur est plus adéquat pour la figure 2.

1.2 Représentations des données

Pour stocker les informations concernant le graphe étudié de manière adéquate, pour qu'on puisse appliquer des algorithmes de recherche, on peut utiliser une matrice d'incidence. La matrice d'incidence décrit le graphe en indiquant quelles branches arrivent sur quels noeuds.

$$M_{i,a} = \begin{cases} 1 & si\ i\ est\ l'origine\ de\ a \\ -1 & si\ i\ est\ l'extrémité\ de\ a \\ 0 & sinon \end{cases}$$

Q2) La matrice d'incidence associée au graphe de la figure 3 est :

$$I = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Q3) On peut également utiliser une matrice d'adjacence, une liste de successeurs ou une liste de cocycles pour stocker ces informations.

La matrice d'adjacence contient le nombre d'arrêtes liant le noeud i au noeud j.

$$M_{i,j} = \begin{cases} \text{O} & si\ \text{(i,j)} \notin E \\ 1 & si\ \text{(i,j)} \in E \end{cases}$$

La matrice d'adjacence associée au graphe de la figure 3 est :

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

On obtient une matrice d'adjacence qui est symétrique et de dimension 5X5 comme le graphe est non-orienté.

La liste de successeurs crée un tableau d'indirections sur les successeurs de chaque noeud.

La liste des cocycle est la liste de successeurs avec les arcs.

2 Cas Pratique

Les algorithmes de recherche commence par le noeud initial et il liste ses voisins pour les explorer un par un. On marque les noeuds déjà développés pour éviter d'explorer un même noeud plusieurs fois.

2.1 Algorithmes Non-Informés

L'algorithme non-informé est un algorithme qui n'a aucune information, tels que le coût et la récompense ne sont pas disponibles. On étudie deux algorithmes de recherche non-informés : la recherche en largeur(algorithme "Breadth First search" BFS) et la recherche en profondeur(algorithme "Depth First search" DFS).

Q4) L'algorithme de recherche en largeur permet d'atteindre le but à partir de la position initial en utilisant une file d'attente FIFO(First In First Out).

La fonction "RechercheEnLargeur.m" effectuant la recherche en largeur reçoit comme entrées le graphe étudié sous la forme d'une liste d'incidence(NodeList), le noeud associé à la position initiale(RootNode) et le noeud associé au but(TargetNode), et elle renvoit le nombre de noeuds parcourus(noeudsparcourus) et le chemin trouvé(chemin).

Q5) L'algorithme de recherche en profondeur permet d'atteindre le but à partir de la position initial en utilisant une file d'attente LIFO(Last In First Out).

Comme la fonction précédente, la fonction "RechercheEnProfondeur.m" effectuant la recherche en profondeur reçoit comme entrées le graphe étudié sous la forme d'une liste d'incidence(NodeList), le noeud associé à la position initiale(RootNode) et le noeud associé au but(TargetNode), et elle renvoit le nombre de noeuds parcourus(noeudsparcourus) et le chemin trouvé(chemin).

2.2 Algorithmes Informés

L'algorithme informé est un algorithme qui a l'information sur le coût de développment (du chemin) et sur une heuristique (défini par la distance du noeud considéré au noeud "but"). On étudie l'algorithme A*.

On suppose qu'on peut estimer la distance entre une case quelconque et la case but par une ligne droite dans l'espace 2D.

Q6) L'algorithme A* cherche le meilleur chemin entre la position initial et le but en faisant une évaluation heuristique sur chaque noeud et en visitant les noeuds par ordre de cette évaluation heuristique. Ici, l'heuristique est calculée par la distance euclidienne.

La fonction "Astar.m" effectuant la recherche selon l'algorithme A* reçoit comme entrées le graphe étudié(NodeList), le noeud associé à la position initiale(RootNode), le noeud associé au but(TargetNode) et la matrice pour trouver les coordonnées dans le plan des noeuds (Map_plan2node), et elle renvoit le nombre de noeuds parcourus(noeudsparcourus) et le chemin trouvé(chemin).

On visualise séquentiellement le parcours découvert entre la case initiale et le but grâce à la fonction "AnimatePath.m". On applique cette fonction pour visualiser les chemins déterminés avec les différents algorithmes.

Q7) On explore l'influence de l'heuristique en utilisant la distance de Manhattan pour calculer l'heuristique.

La fonction "AstarMan.m" est la même fonction que "Astar.m" avec la nouvelle heuristique calculé par la distance de Manhattan. Nous allons comparer cette heuristique avec la distance euclidienne utilisée précédemment.

La distance de Manhattan et la distance euclidienne sont équivalentes pour des chemins unidirectionnels. La distance euclidienne permet un "balayage" plus fin puisque pour des chemins en diagonal, la différence entre l'heuristique des noeuds voisins pour les deux meilleures possibilitées est plus faible qu'avec la distance de Manhattan. Ainsi, on peut dire que la distance euclidienne donne une heuristique minorante qui fournira le résultat optimal. Néanmoins, avec la distance Manhattan on voit que l'on stocke moins de noeuds en mémoire.

Q8) On voit l'effet de coûts variables sur l'algorithme A*. Ce problème est intéressant pour un cas d'un robot qui se déplace sur un terrain accidenté. Ici, le variations de relief et d'adhérence peut être représentées par des coûts de déplacements différenciés.

On crée de nouveaux coûts non-uniformes dans les variables Nodes et NodeList(n).K en exécutant le script "CoutVariable.m".

On applique ces coûts de déplacement et on va étudier l'influence de ces coûts pour l'algorithme A*.

En visualisant les chemins des trois algorithme A* précédents, on peut voir que l'optimalité du chemin est en ordre de : A* avec la distance euclidienne, A* avec la distance de Manhattan et A* avec la distance euclidienne et les coûts variables. Même si, le chemin pour l'algorithme A* avec les coûts variables ne semble pas le meilleur chemin en 2D, il peut être le chemin qui nous donne le plus de recompense. On remarque également que la complexité en espace est en ordre de : A* avec la distance de Manhattan, A* avec la distance Euclidienne et A* avec la distance euclidienne et les coût variables.

Q9) Complexité en temps :

On détermine empiriquement l'algorithme le plus rapide. On a créé 5 scénarios variés pour essayer de généraliser nos résultats. On va stocker le temps de résolution de chaque algorithme pour les différents scénarios et faire la moyenne. Idéalement, il faudrait créer une fonction qui génère des scénarios alétoires pour avoir plus de mesures.

	DFS	BFS	$A*$	$A*Manhattan$	$A*variablecost$
$temps(ms)$	3.537	3.061	56.098	17.553	37.201

On constate que les algorithmes de recherches non informés (largeur et profondeur) sont en moyenne, sur notre plage de test, 10 fois plus rapide en temps que les algorithmes informés (A*).

Q10) Complexité en espace :

On utilise les matrices d'incidence associées aux arbres des figures 1 et 2 stockées dans "tree1.mat" et "tree2.mat". Pour récupérer les listes correspondantes, on utilise également la fonction "Matrix2List.m".

On estime empiriquement l'algorithme de moindre complexité en espace pour chacun des deux graphes en utilisant les fonctions "RechercheEnLargeur.m" et "RechercheEnProfondeur.m".

Pour cela, on fait des initialisations aléatoires du noeud de départ et du noeud but. On stocke la complexité en espace pour les deux algorithmes et les deux arbres. On calcule ensuite la moyenne pour

déterminer la complexité pour chacun des algorithmes.

Pour 10000 itérations, on a les valeurs de complexité en espace moyenne suivante :

	$Arbre1$	$Arbre2$
$Largeur$	21.58	21.22
$Profondeur$	19.41	19.08

On constate que l'algorithme de moindre complexité en espace pour les deux graphes est l'algorithme de recherche en profondeur.