KHODJA Méziane
LE BALANGER Alexandre
MORAND Martin
TRAN-CONNAN Thiên Nhân

01/10/2021
Maîtrise en informatique
Parcours professionnel

Étudiants à l'Université du Québec à Chicoutimi

IA
COMPTE RENDU DE TP 1

Création d'un agent aspirateur

« Nous attestons que ce travail est original et qu'il ait référencé de façon appropriée chaque source utilisée »

UQAC
Université du Québec
à Chicoutimi

Introduction

Le premier travail pratique d'Intelligence Artificielle va nous permettre de mieux assimiler les connaissances théoriques vues en cours, comme par exemple le fonctionnement des algorithmes de recherches, ou encore la modélisation de l'heuristique de l'agent. Nous allons effectuer ce premier travail en groupe de 4, en utilisant l'éditeur de code Visual Studio Code et le langage de programmation Python. Afin d'organiser notre travail collaboratif, nous allons utiliser le service de gestion de développement logiciel Github, le lien de celui-ci est le suivant : https://github.com/ImArxus/IAAspiMaison.

But du projet

Le but de ce travail pratique va être de modéliser le fonctionnement d'un agent aspirateur intelligent qui aura pour but de garder un manoir propre, ce dernier sera représenté par une grille de 5x5 cases. Afin de complexifier l'environnement, ce dernier créera sporadiquement de la saleté ou des bijoux dans certaines pièces, le rôle de notre aspirateur sera ainsi d'aspirer la saleté tout en ramassant les bijoux. Afin de modéliser le comportement du robot, nous exploiterons deux algorithmes vus en cours, un informé et l'autre non informé, qui seront respectivement A* et BFS dans notre cas. Une mesure des performances sera également réalisée afin de comparer ces deux exécutions. Par exemple, le robot perdra des points à chaque déplacement et à chaque mauvaise action réalisée et au contraire en gagnera lorsqu'il effectuera une action correcte. Dans la suite de ce rapport, nous expliquerons notre réalisation détaillée des différentes parties que sont la modélisation de l'environnement, de l'heuristique et des algorithmes de recherche informée et non-informée.

I - Modélisation de l'environnement

Pour la modélisation de l'environnement, nous avons créé deux classes. La première classe s'appelle "Cell.py", elle va constituer les pièces de l'environnement. Elle est constituée des attributs comme "dust" qui matérialise la poussière et "jewel" les bijoux. Elle comporte aussi des coordonnées afin qu'elle puisse être localisée au sein de l'environnement. Ensuite il y a la deuxième classe "Grid.py" qui constitue le cadre global de l'environnement. Elle a comme attributs "rows" et "cols" qui permettent de donner un périmètre à l'environnement. Et bien sûr, il y a aussi un dernier attribut qui est une liste de Cell. Enfin en les implémentant ensemble, nous obtenons un environnement de taille 5 x 5 ce qui donne 25 pièces comme demandé dans le sujet. L'environnement ressemble alors à un tableau de deux dimensions. On a aussi créé une méthode *__str__* pour chacune des classes afin de pouvoir afficher en console. Nous avons ensuite implémenté une librairie native de Python qui est Tkinter afin d'obtenir un affichage graphique afin

de mieux visualiser l'environnement. Nous avons choisi d'utiliser des couleurs afin de décrire l'état d'une pièce dans l'environnement : rouge pour la poussière, bleu pour le bijou, jaune quand la pièce contient les deux éléments, et blanc quand la pièce ne contient rien. Nous avons aussi implémenté du multithreading avec la librairie threading de python. Il y a alors une classe qui sera le thread de l'environnement et une autre qui sera le thread de l'agent.

II - Modélisation de l'heuristique

Pour notre heuristique, nous avons réalisé une fonction *distance(node_posX: int, node_posY: int) -> int* dans la classe *Node* de notre agent. Cette fonction permet de calculer le nombre de déplacements nécessaires pour atteindre le nœud donnée en paramètre de la fonction. Pour cela, cette fonction additionne simplement l'écart entre les coordonnées du nœud actuel et du nœud cible en abscisse et en ordonnée. Le résultat de cette fonction, additionnée à la valeur de l'attribut de *Node* nommé *energy_cost* et implémenté à la création de chaque nouveau nœud fils, permet ainsi d'avoir une valeur quantifiable quant à l'optimalité des actions de notre agent. Ceci sera particulièrement utile plus loin lorsque nous verrons l'algorithme de recherche informé.

III - Algorithme de recherche non informée

L'expression "non informée" de ce type algorithme signifie que ce dernier ne connaît qu'uniquement l'information disponible dans la définition du problème, l'agent va devoir ainsi chercher dans les cases juxtaposées celle où il devra se déplacer pour effectuer une action. Il existe plusieurs algorithmes rentrant dans cette catégorie, à savoir BFS (stratégie de recherche en largeur, UCS (stratégie de recherche à coût uniforme), DFS (stratégie de recherche en profondeur), … Nous avons choisi de mettre en place la technique BFS (Breadth First Search), car celle-ci est complète et optimale dans notre cas, étant donné que les coûts de déplacement entre pièces juxtaposées sont toujours égaux à 1. Les défauts de cette méthode vont être que le temps et l'espace nécessaire vont avoir une croissance exponentielle en fonction de la profondeur. Or le fait que nous possédons un environnement restreint ne contenant que 25 cases va nous permettre d'effectuer cette technique sans qu'elle soit trop compromettante à mettre en place.

Afin de concevoir cet algorithme, j'ai créé une classe *AlgoNI.py* contenant quelques paramètres disponibles dans le tableau suivant :

Paramètre	Rôle
grid	Permet de représenter la grille sur laquelle notre algorithme va agir.
robot	Permet de représenter le robot sur lequel notre algorithme va agir.
nodeStudied	Permet de représenter les nœuds déjà étudiés. Nous y stockons uniquement des 'str'

	contenant la position des cellules visitées, celles-ci seront de la forme ‘00’.
nodeToVisit	Permet de représenter les nœuds à étudier. Nous y stockons uniquement des ‘str’ contenant la position des cellules à visiter, celles-ci seront de la forme ‘00’.
cellToVisit	Permet de représenter les cellules à visiter. Nous y stockons uniquement des objets de type ‘Cell’. Il est important que nous conservions à la fois les positions et les cellules car les premiers vont nous permettre de comparer aisément les éléments entre eux et les seconds vont nous permettre d’agir sur les cellules.

Tableau 1 - Explication des paramètres de notre classe AlgoNI.py

Afin d’utiliser cet algorithme, nous n’avons qu’à spécifier la grille et le robot à utiliser, et d’y inscrire trois tables vides. Nous ajoutons ensuite la position actuelle dans la liste de celles à étudier, puis nous créons une boucle ‘while’. Tant que nous n’avons pas trouvé une cellule à atteindre pour y effectuer des actions, et que notre liste de cellules à visiter n’est pas vide; nous appelons la fonction récursive *analyseGrid()*. Cette dernière permet d’analyser la cellule actuelle et de la retourner si elle possède des bijoux ou de la saleté. Dans le cas contraire, nous allons étudier si elle possède des cases voisines qui n’ont pas encore prévu d’être testé ou déjà testé, si oui nous les ajoutons à notre liste de cellules à analyser. Bien évidemment, la cellule actuelle est supprimée de celles à visiter et est ajoutée à celles étudiées. Lorsqu’une cellule contenant de la poussière ou des bijoux est obtenue, nous la renvoyons à l’aspirateur qui va alors calculer les actions nécessaires pour s’y déplacer avec la fonction *calcul_destination_to_cell(),* puis les effectuer avec la fonction *action_robot()*. Afin de pouvoir effectuer cette méthodologie de manière continue, nous créons une boucle ‘while’.

IV - Algorithme de recherche informée

Un algorithme informé fonctionne sur le principe que l’agent connaît l’environnement dans lequel il évolue. Ainsi, il est capable de déterminer si il est plus ou moins proche de son objectif en utilisant l’heuristique présentée plus tôt.

La structure de cet algorithme est principalement située dans la classe *Sensors.py* du dossier *Agent*. C’est dans cette dernière que vont être calculées toutes les actions à effectuer par le robot. Pour chercher le chemin à parcourir pour atteindre la case poussiéreuse la plus proche, nous avons choisit d’utiliser l’algorithme de recherche A* vu en cours, pour son optimalité en termes de coût de déplacement. Celui-ci fonctionne de la manière suivante :

- On crée une liste de nœuds (arbre d’exploration) où chaque nœud correspond à une cellule de la grille avec comme parent un autre nœud à partir duquel accéder à celui-ci.
- On ajoute à la liste de nœud le nœud de la cellule actuelle où se tient le robot.
- Puis, on effectue une boucle qui ne s’achève qu’une fois que nous avons la suite d’instructions optimale pour atteindre la case poussiéreuse.

Pour vérifier la condition d’arrêt de cette boucle, nous avons créé une fonction nommée *goal_reached(node: Node) -> bool* dont le paramètre correspond au noeud actuel où devrait être rendu le robot à la fin de l’exécution des actions qui lui sont transmises. Cette fonction a pour but

d'estimer la performance globale de l'agent et de retourner True uniquement si cette estimation est supérieure à la performance actuelle de l'agent, autrement dit, si le chemin déterminé par l'algorithme A* est optimal. Pour ce faire, cette fonction calcule pour chaque action à effectuer par le robot la performance qui en résulterait. Chaque action est récupérée en inspectant le champ "action" de chaque objet *Node*, obtenu un à un par la méthode *get_parent()* à l'intérieur d'une boucle *while()*. En ce qui concerne la performance par action, celle-ci est calculée dans la méthode *perfomance_after_action(node: Node, action: str) -> int* :

- Pour chaque action effectuée par le robot, la mesure de performance baisse de 1.
- Si le robot aspire un bijou, la performance baisse de 50 points, puisque celui-ci est censé les attraper plutôt que de les aspirer. Cette action peut survenir si un bijou se trouve sur la même case qu'une poussière.
- Si le robot aspire une poussière, la performance augmente de 25.
- Si le robot attrape un bijou, la performance augmente de 10.

À l'intérieur de cette boucle, c'est ici que va se passer notre exploration. On prend tout d'abord le premier nœud de la liste que l'on inscrit dans une variable temporaire pour ensuite le supprimer de la liste : cela a pour but de ne pas traiter plusieurs fois le même nœud. Ensuite, on cherche tous les nœuds voisins de ce nœud avec les actions correspondantes pour les atteindre ("left", "right", "up" et "down") ou les actions de nettoyage/ramassage ("clean" et "grab"), à l'aide de la fonction *expand(self, grid: Grid, robot) -> list[Node]*. Pour faire cela, cette fonction liste toutes les actions possibles à partir du nœud courant puis crée un nouveau nœud pour chacune de ces actions, en augmentant la profondeur des nœuds fils, ainsi que leur coût en énergie et en calculant la nouvelle distance du nœud contenant la cellule poussiéreuse. Puis on ajoute tous ces nouveaux nœuds à une liste renvoyée par la fonction. Enfin, on trie la liste de nœud par ordre croissant en fonction de l'énergie dépensée pour atteindre ces nœuds et de l'heuristique explicitée plus tôt.

Une fois tous nos nœuds récoltés et la liste de ceux-ci triée, on récupère les actions contenues dans ces nœuds sous forme de liste, dont on inverse ensuite l'ordre. En effet, la première action en sortie de l'algorithme A* est en réalité la dernière, soit "clean" puisque les nœuds renvoyés conduisent à la cellule poussiéreuse. Finalement, on ajoute toutes ces actions dans le bon ordre à la liste des actions à effectuer par le robot. Puis, comme pour l'algorithme précédent, on les exécute à l'aide de la fonction *action_robot(self, fenetre, dessin, c) -> None* située dans la classe *Effectors.py*.

Conclusion

En conclusion ce TP nous a permis de tester par la pratique les notions vues en cours, telles que la création d'algorithmes de recherches ou encore la modélisation de l'heuristique d'un agent. Il nous a également permis de manipuler des outils informatiques, comme par exemple le maniement de Python sur lequel nous n'avions pas trop eu l'habitude de programmer. Finalement, nous avons aussi pu mesurer les performances de nos deux types d'algorithme, cependant nous trouvons que notre algorithme de recherche non informé est plus performant que celui informé, ce qui semble étonnant. L'utilisation d'une grille plus importante pourrait mettre en évidence le meilleur fonctionnement de l'algorithme A*.

Références

https://www.tutorialspoint.com/python/python_multithreading.htm : Pour le multi-threading

http://math.univ-lyon1.fr/irem/Formation_ISN/formation_interfaces_graphiques/module_tkinter/exo_canevas.html : Utilisation de Tkinter.

Instructions

Pour lancer notre programme, il faut que vous utilisiez la version 3.9 de Python pour qu'il puisse s'exécuter correctement. Ensuite, ce projet est à exécuter directement à partir de la racine du dépôt cloné en lançant la commande "*python3 main.py*".

Concernant la vidéo, on peut distinguer sur la fenêtre graphique de gauche l'agent fonctionnant avec un algorithme informé, et non-informé pour celui de droite. Le choix de l'algorithme se fait en changeant la valeur du booléen à la ligne 28 de la classe *Thread_Robot.py*, plus précisément le paramètre de la fonction *self_action(informed: bool)* : *True* pour un agent informé et False pour un agent non-informé.