8INF846 - Intelligence Artificielle

UQAC
Université du Québec
à Chicoutimi

Travail pratique n°1

Création d'un agent aspirateur

Mathias DURAND (DURM13099909)

Sophie GERMAIN (GERS09559909)

Pierre LACLAVERIE (LACP03119904)

Barnabé PORTIER (PORB26070006)

Remis pour évaluation le 14 février 2022

I. Contextualisation

A. Rappel du problème

Le sujet se concentre sur la modélisation du nettoyage par un robot aspirateur *intelligent* d'une zone de 5x5. Trois éléments vont impacter le cours de la vie du robot: l'apparition d'une poussière, l'apparition d'un diamant et l'action d'aller aspirer le diamant.

L'objectif de ce programme est de modéliser ces trois grandes phases. Il est à noter que les diamants ainsi que les particules de poussière à aspirer apparaissent de manière aléatoire tout au long de l'exécution du programme.

B. Exécution de notre programme

Voici les caractéristiques d'exécution de notre programme :

- Version de Python > 3.8
- Utilisation de la bibliothèque anytree : `pip install anytree`
- Utilisation de la bibliothèque argparse : `pip install argparse` (normalement installée par défaut dans les versions > 3.2 de Python)
- Fichier Python à exécuter : `Main.py`
- Paramétrage supplémentaire optionnel : `--uninformed` pour utiliser l'algorithme non informé UNIQUEMENT (défaut = `False`)
- Paramétrage supplémentaire optionnel : `--size` permet de renseigner la taille de notre manoir (défaut = `5`, correspond à un manoir de 5x5 pièces)
- Paramétrage supplémentaire optionnel : `--time-limit valeur` permet de renseigner une limite de temps à l'exécution (valeur défaut = `120 secondes`)
- Paramétrage supplémentaire optionnel : `--gen-interval valeur` renseigne l'intervalle de temps auquel l'environnement génère des éléments dans le manoir (valeur défaut = `0.05 secondes`). Si cette valeur est trop faible, le robot ne pourra pas suivre la cadence de génération d'objets par l'environnement.
- Paramétrage supplémentaire optionnel : `--verbose` pour afficher la carte du manoir lors de la création de la métrique avec l'algorithme non informé
- Arrêt du programme : après *time-limit* secondes d'exécution ou Ctrl+C (**une fois**) - L'arrêt nécessite parfois quelques secondes
- `Main.py -h` pour revoir ces paramètres optionnels

Il est important de noter que dans le cas d'utilisation de l'algorithme informé, la création de la métrique avec l'algorithme non informé peut prendre quelques minutes (~10 minutes mesurées sur un ordinateur de basse puissance pour une taille 5 et les réglages par défaut).

En effet, dans une matrice 5x5, il s'agit d'enregistrer les plus faibles coûts entre chacune des 25 cases. Utiliser le programme avec une taille de 2 (manoir 2x2) permet de tester plus rapidement l'entièreté du programme.

II. Modélisation de la solution

Nous avons modélisé la solution à l'aide de classes Python :

- Environnement : définit l'environnement (carte des lieux, poids…)
- Agent : définit l'agent intelligent, qui analyse l'environnement avec son Capteur se déplace et peut interagir avec l'environnement avec son Effecteur
- Capteur : permet à l'agent d'analyser l'environnement et ses cases
- Effecteur : permet à l'agent d'agir sur l'environnement : se déplacer, ramasser des bijoux, aspirer de la poussière…

La métrique que nous avons choisi d'utiliser pour le comportement informé de notre agent est basée sur le "coût" de traversée de chaque case. Ainsi, le coût nécessaire pour aller d'une case A à une case B sera la somme des coûts des cases traversées pour ce chemin.

Deux threads sont lancés en parallèle du thread "principal" : un pour l'agent et un pour l'environnement. Ils communiquent par référence par le biais du capteur et de l'effecteur appartenant à l'agent.

Il est à noter qu'une fois lancé, le thread de l'environnement génère aléatoirement des "poussières" (`Element.DIRT`) et des bijoux (`Element.JEWEL`) toutes les X secondes (0.05 secondes par défaut, paramétrable par un argument en ligne de commande), avec des probabilités respectives de 15% et 5%. Fondamentalement, le programme n'a pas vocation à s'arrêter, mais pour des raisons pratiques nous avons mis une limite de temps à l'exécution du programme (2 minutes par défaut, paramétrable par un argument en ligne de commande). Voir la partie Exécution de notre programme pour plus de détails quant à la modification de cette limite.

Note importante : afin de rendre l'affichage plus clair pour l'utilisateur (possibilité de voir le déplacement du robot, puis aspiration de la poussière par exemple), nous avons placé un `time.sleep(0.35)` dans le fichier `Effecteur.py`, fonctions `vaccum()` et `get_jewels()`. Celui-ci ne rentre donc pas dans la logique d'exécution mais n'est présent que pour l'affichage.
L'affichage d'informations lors de la création de la métrique est désactivé par défaut mais il est possible de le retrouver avec l'argument `--verbose`.

III. Code et explications

A. Algorithme d'exploration non-informée

Nous avons choisi l'algorithme de `Breadth-First-Search`, aussi connu sous le nom d'algorithme de recherche en largeur.

Celui-ci permet une recherche sans que l'agent n'ait de connaissance *globale* de l'environnement.

```
def BFS(self, current_agent_position):
    rootNode = AnyNode(id=current_agent_position)
    chemin = []
    lastNode = rootNode
    queue = []
    queue.append(rootNode)
    while len(queue) > 0:
        lastNode = queue.pop(0)
        if self.capteur.isGoal(lastNode.id[0], lastNode.id[1]):
            chemin.append(lastNode.id)
            while lastNode.parent is not None:
                lastNode = lastNode.parent
                chemin.append(lastNode.id)
            return chemin[::-1]
        for neighbor_object in self.capteur.getNeighbors([lastNode.id[0], lastNode.id[1]]):
            neighbor_position = neighbor_object["position"]
            if self.capteur.isInBornes(neighbor_position):
                newNode = AnyNode(id=neighbor_position, parent=lastNode)
                queue.append(newNode)

    return chemin
```

B. Algorithme d'exploration informée

Après avoir exécuté l'algorithme d'exploration non informé, le programme va exécuter notre algorithme d'exploration informée. Celui-ci consiste en l'algorithme `A*` que nous avons adapté à notre contexte.

En prérequis, le `Breadth-First-Search` doit avoir été exécuté suffisamment de fois pour que la matrice d'adjacence des coûts, notre métrique, ait été intégralement remplie (i.e. que le robot ait fait chaque chemin au moins une fois). Cette matrice d'adjacence est utilisée pour calculer les chemins dans `A*`.

L'algorithme `A*` prenant en entrée une source -l'emplacement actuel du robot- et une target -une poussière à aspirer ou un diamant à collecter. En premier lieu il nous faut donc lui trouver une target. Pour cela, on trouve sur la grille toutes les poussières ainsi que les diamants. On cherche, *s'il y en a,* la plus proche poussière à aspirer, selon notre heuristique (cf Heuristique). On obtient alors le nœud de destination, la target.

On applique alors l'algorithme `A*` pour trouver le chemin le plus court selon l'heuristique. Dans le cas où le chemin trouvé est *meilleur* que le chemin trouvé grâce au `Breadth-First-Search` , l'algorithme met alors à jour la matrice représentant les chemins les moins coûteux pour aller d'un point à l'autre.

Pour chaque nœud intermédiaire entre le point de départ et d'arrivée, le robot va aspirer les poussières (ou collecter le diamant) qu'il rencontre. Le robot ne va pas se téléporter comme lors de l'algorithme précédent.

Ci-dessous une capture d'écran du code de l'algorithme A*, cœur de la recherche informée qui est effectuée dans cette partie.

<u>Brève explication de l'algorithme A*.</u>

Cet algorithme permet un parcours d'un graphe en suivant la règle suivante :

	F est le coût global du nœud.
F = G + H avec :	G est la distance entre l' actuel nœud et le point de départ.
	H est l'heuristique utilisée.

L'algorithme va *se diriger* dans la direction du nœud d'arrivée, il va scanner les cases proches du nœud actuel et va appliquer la formule F=G+H. La valeur minimale est prise, puis le parcours va se faire à partir du nœud contenant cette valeur. Si un chemin se trouve être de poids plus important vis-à-vis d'un chemin trouvé précédemment, le chemin actuel est mis de côté, stocké, et l'algorithme va se positionner sur le nœud dont le coût total est le plus faible afin et va continuer à explorer les possibilités qui s'offrent à lui. Quand le nœud d'arrivée, la *target*, est atteint, l'algorithme va renvoyer le chemin le plus court pour aller du nœud dans lequel le robot se trouve jusqu'au nœud d'arrivée.

```
def Astar(self, start_pos, goal_pos):
    if goal_pos is None:
        return []
    openSet = [start_pos]
    cameFrom = {}
    gScore = {}
    gScore[start_pos] = 0
    fScore = {}
    start_mat_metric = (start_pos[0] * self.capteur.environnement.taille)+(start_pos[1])
    goal_mat_metric = (goal_pos[0] * self.capteur.environnement.taille)+(goal_pos[1])

    fScore[start_pos] = self.metric[start_mat_metric][goal_mat_metric]["poids"]

    while len(openSet) > 0:
        current = min(openSet, key=lambda o: fScore[o])
        if current[0] == goal_pos[0] and current[1] == goal_pos[1]:
            return reconstruct_path(cameFrom, current)

        openSet.remove(current)
        for neighbor_obj in self.capteur.getNeighbors(current):
            tentative_gScore = gScore[current] + int(neighbor_obj["weight"])
            neighbor = neighbor_obj["position"]
            if neighbor not in gScore.keys() or tentative_gScore < gScore[neighbor]:
                cameFrom[neighbor] = current
                gScore[neighbor] = tentative_gScore
                neighbor_pos_metric = (neighbor[0] * self.capteur.environnement.taille)
                                      +(neighbor[1])
                goal_pos_metric = (goal_pos[0] * self.capteur.environnement.taille)
                                  +(goal_pos[1])
                fScore[neighbor] = tentative_gScore + self.metric[neighbor_pos_metric]
[goal_pos_metric]["poids"]
                if neighbor not in openSet:
                    openSet.append(neighbor)
    return []
```

C. Test de décision pour le choix entre exploration informée et non informée à partir de l'apprentissage

Durant l'exécution de l'algorithme non informé, la métrique est alimentée petit à petit jusqu'à être complète. On exécute la suite du programme à l'aide de l'algorithme de A*. On utilise un compteur dégressif. L'idée est qu'on veut arrêter l'algorithme quand tous les chemins existent dans notre matrice de métrique. On a besoin de 325 valeurs uniques (les chemins `i->j` sont en miroirs de chemins `j->i`, 300 tels chemins + les 25 qui retournent au point de départ) on commence donc notre compteur à cette valeur. Dès qu'on remplit une case qui était jusque là vide dans notre matrice de métrique (ou plus exactement, à la valeur par défaut de 10000), on décompte.

```
def apprentissage_fini():
            return not self.want_uninformed_only and self.counter_metric == 0
```

D. Collecte et génération de la métrique pour l'apprentissage

On attribue un poids à chaque case, la somme des poids des cases traversées lors d'un déplacement nous donne donc le poids du chemin. Si celui-ci est le meilleur qu'on est trouvé jusque là (donc si c'est le premier ou que son poids est inférieur à celui du chemin précédemment connu) on l'enregistre dans une matrice d'adjacence. Matrice qui représente les liens entre toutes les cases de notre environnement.

```
def execute_action_plan(self, action):
        self.effecteur.update_position(action[-1])
        self.effecteur.move_robot()
        self.effecteur.decide_what_to_do()
        map_env = self.capteur.environnement.map
        poids_chemin = 0
        for coord in action:
            poids_chemin += map_env[coord[0]][coord[1]]["weight"]
        metric = self.metric
        depart = action[0]
        destination = action[-1]
        dep_mat_metric = (depart[0] * self.capteur.environnement.taille)+(depart[1])
        dest_mat_metric = (destination[0] * self.capteur.environnement.taille)+destination[1]
        if poids_chemin < metric[dep_mat_metric][dest_mat_metric]["poids"]:
            if metric[dep_mat_metric][dest_mat_metric]["poids"] == 10000:
                self.counter_metric -= 1
            metric[dep_mat_metric][dest_mat_metric]["poids"] = poids_chemin
            metric[dest_mat_metric][dep_mat_metric]["poids"] = poids_chemin
            metric[dep_mat_metric][dest_mat_metric]["chemin"] = action
            metric[dest_mat_metric][dep_mat_metric]["chemin"] = action[::-1]
        return metric
```

E. Heuristique

Notre environnement est une map de 25 cases. Chacune de ces cases a un contenu `contents` (liste contenant 0 s'il n'y a rien, ou 1 s'il y a un diamant et/ou 2 s'il y a de la poussière et/ou 3 s'il y a le robot) et le coût `weight` relatif à sa traversée. En effet, comme expliqué précédemment, à chaque salle de notre manoir est associé un coût de traversée. Nous avons donc arbitrairement affecté des coûts à chacune des 25 cases de notre environnement.

```
class Environnement:
    def __init__(self, taille, freq):
        weight_matrix = [[2,5,4,1,4],[3,8,6,1,4],[2,4,9,4,8],[12,1,3,6,5],[3,5,9,4,2]]
        self.taille = taille
        self.map = [[{"contents":[], "weight":weight_matrix[i][j]} for i in range(taille)] for j in
range(taille)]
    [...]
```

F. Test (if, for) de l'objectif

Pour des coordonnées particulières, on établit la case comme objectif si et seulement si elle contient un `Element` qui intéresse notre agent, c'est-à-dire de la poussière ou des bijoux.

```
def isGoal(self, i, j):
    if(Element.JEWEL in self.environnement.map[i][j]["contents"]
       or Element.DIRT in self.environnement.map[i][j]["contents"]):
        return True
    return False
```

G. Modélisation de l'état mental BDI dans l'arbre d'exploration

Au cours du développement du programme, nous n'avons pas tout à fait réussi à mettre en valeur la consigne.

```
while am_I_alive: #This is the analysis cycle
environment = observe_environment_with_my_sensors()
state_bdi = update_my_state_bdi(environment,metric)
plan_action = execute_exploration(state_bdi, algo-informed or algo-not-informed)
metric = execute_action_plan(plan_action)
```

Cette étape se trouve disséminée dans plusieurs parties de notre programme. Ainsi, le BDI (Belief, Desire, Intention) est dispersé comme suit :

- Belief est contenu dans la matrice représentant l'heuristique (cf Heuristique)
- Desire est modélisée par l'utilisation de la fonction `isGoal` se trouvant dans le capteur.
- Intention est modélisé par la logique dans `UpdatePosition` : le robot choisit de prendre tel quel le résultat de l'algorithme (informé ou non informé) et de bouger en remontant le chemin qui a été donné. L'affichage dans l'environnement est mis à jour dans `MoveRobot`.

Desire et Intention sont représentés dans la classe `Effecteur` (cf Modélisation de Capteurs et d'actionneurs)

H. Modélisation de capteurs et d'actionneurs.

Le capteur possède une référence vers l'environnement créé dans le Main.py du programme. Il possède différentes fonctions permettant à l'agent qui le possède d'analyser l'environnement et les cases sur lequel il se trouve.

```
class Capteur():
    def __init__(self, environnement):
        self.environnement = environnement

    def isGoal(self, i, j):
        # renvoie True si la position présente de la
        # poussière ou des bijoux, False sinon
        [...]

    def isInBornes(self, position):
        # renvoie True si la position est dans les
        # limites du manoir, False sinon
        [...]

    def observe_environnement(self):
        # renvoie la liste des positions avec un goal
        [...]

    def getNeighbors(self, position):
        # renvoie la liste des positions et poids
        # des voisins du paramètre
        [...]
```

L'effecteur est créé avec une référence vers le même environnement que le capteur, qui va lui permettre d'y appliquer des modifications. L'effecteur permet à l'agent qui le possède de décider que faire à l'instant t : se mouvoir dans l'environnement (notamment important pour l'affichage de la carte des lieux), ramasser des bijoux ou aspirer de la poussière.

```
class Effecteur():
    def __init__(self, environnement):
       [...]

    def move_robot(self):
        self.move_robot_to(self.position)

    def move_robot_to(self, position):
        [...]
        # place le robot dans l'environnement

    def decide_what_to_do(self):
        # analyse le sol pour savoir s'il doit aspirer
        # ou ramasser
        [...]

    def vacuum(self):
        # aspire la poussière et le sol
        [...]

    def get_jewel(self):
        # ramasse les bijoux
        [...]

    def update_position(self, position):
```