UQAC

Cours 8INF847 - Intelligence artificielle

Kévin Bouchard, Ph.D.

TP 1 - Agent intelligent et exploration

Claire Mielcarek

Esther Prudhon-Delagrange

Orann Weber

8 octobre 2018

Table des matières

Table des figures

Introduction

Ce projet a pour objectif de modéliser un agent aspirateur intelligent ayant pour but d'aspirer la poussière et de ramasser les bijoux dans l'environnement dans lequel il évolue.

1 Modélisation

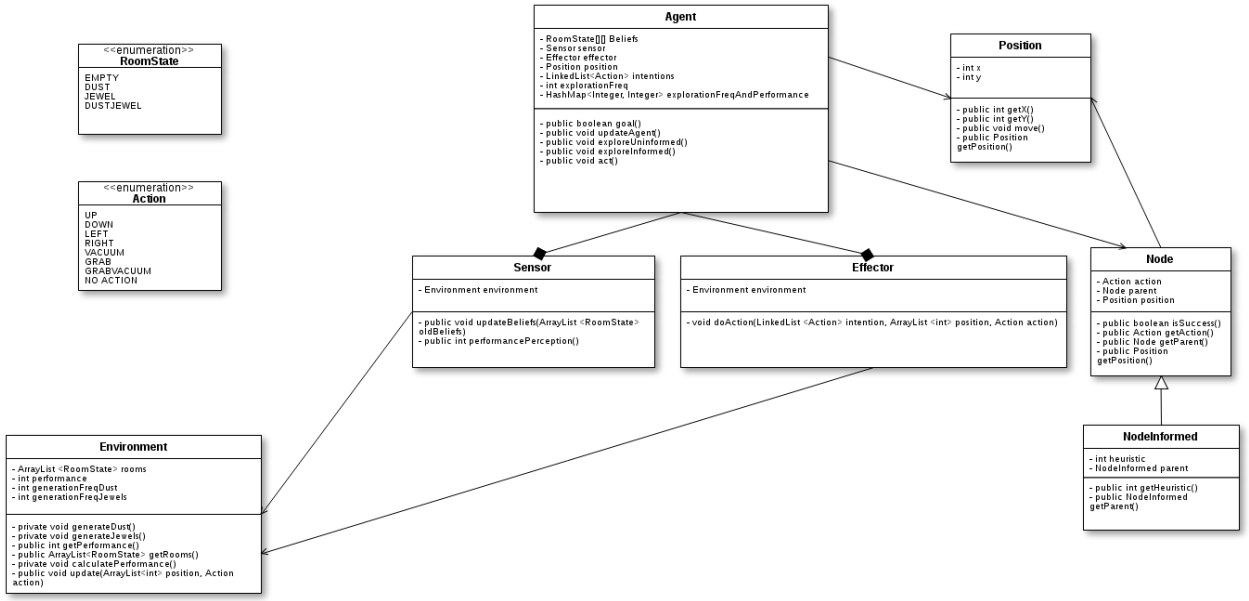

FIGURE 1 – Diagramme de classe

1.1 Modélisation et propriétés de l'environnement

Notre environnement à modéliser est constitué de 100 pièces, que nous représentons à l'aide d'un tableau. Chaque pièce étant caractérisée par son état, les valeurs contenues dans le tableau sont de type `RoomState`, un type énuméré représentants les différents états possibles : `EMPTY` (vide), `DUST` (présence de poussière), `JEWEL` (présence d'un bijou), `DUSTJEWEL` (présence de poussière et d'un bijou). L'environnement évalue la performance de l'agent et dispose pour cela d'un attribut performance. On lui donne également deux attributs de fréquence qui représentent les délais que l'on fixe à l'environnement pour la génération aléatoire de poussière et de bijoux.

Notre environnement dispose des propriétés suivantes :

— Complètement observable : Bien que l'on ne reçoive des perceptions de l'environnement que lors de l'observation et la mise à jour de l'état interne de l'agent, l'agent

peut en théorie faire appel à ses capteurs afin d'avoir accès à l'état de l'environnement en tout temps.
— Stochastique : On génère aléatoirement des poussières et des bijoux dans l'environnement, il ne dépend donc pas seulement de son état courant et de l'action de l'agent.
— Episodique : L'agent agit selon des séquences perception/action qui sont définies uniquement par les perceptions car il ne garde pas en mémoire ses actions précédentes.
— Dynamique : L'environnement évolue en parallèle du déroulement du programme de l'agent.
— Discret : L'état de l'environnement est représenté par des variables de type enum, il est donc discret. Il en est de même pour les perceptions et actions de l'agent. On utilise des sauts de temps, le temps est modélisé de manière discrète.
— Mono-agent.

1.2 Modélisation de l'agent

L'agent que l'on utilise est un agent basé sur les buts. On choisit de lui donner un seul but dans un premier temps, qui est de vider l'environnement. On note que lorsque son but est accompli il ne "meurt" pas, puisque l'on sait que notre environnement pourra continuer à évoluer, mais prend cependant une pause de 30 secondes et affiche sa réussite.

Comme tout agent, il est composé de deux éléments, un capteur et un effecteur, qui lui permettent d'interagir avec l'environnement et qui constituent ses attributs. Le capteur interviendra dans sa phase d'observation de l'environnement et l'effecteur dans sa phase d'action.
Comme tout agent, il dispose d'un état mental représenté par les éléments suivants :
— Ses croyances (beliefs), qui vont correspondre à ce qu'il croit quant à l'environnement et qui vont donc être modélisées par un tableau de `RoomState`.
— Ses désirs/buts, qui sont modélisés par une fonction `goal()` testant la réussite de ses buts.
— Ses intentions, c'est à dire son plan d'action afin de réaliser ses buts, en se basant sur ses croyances, modélisé par une liste chaînée d'actions.

2 Programme

2.1 Déroulement du programme

Notre programme débute par la création de l'environnement (à noter que lors de cette création, on génère 10 poussières et 5 bijoux, comme spécifié dans le constructeur). On créé ensuite l'agent que l'on place aléatoirement dans l'environnement.
On va par la suite diviser le programme en deux processus fonctionnant en parallèle : un pour l'environnement et un pour l'agent afin de garantir l'indépendance de ces deux éléments.

L'environnement s'affiche déjà une première fois dans le terminal afin de montrer qu'il a commencé à fonctionner. On va par la suite fonctionner à l'intérieur d'une boucle infinie, que l'on devra arrêter manuellement quand on le désirera. Au sein de cette boucle, l'environnement va simplement générer aléatoirement de la poussière et des bijoux à des fréquences données.

Le cycle de vie de l'agent va également avoir lieu au sein d'une boucle infinie. Cest le cycle de vie classique d'un agent basé sur les buts :

— Observe l'environnement et met à jour son état interne : méthode `updtateAgentState()`
— Choisit un plan d'action (par le biais d'algorithmes d'exploration)
— Execute les actions : méthode `act()`

2.2 Algorithmes d'exploration

Afin de réaliser notre exploration, nous proposons deux algorithmes différents, un de type non informé et l'autre de type informé. Il est possible de changer l'algorithme que l'on souhaite utiliser en changeant tout simplement la valeur de la variable `explorationAlgorithm` dans le main. Nous modélisons la structure à explorer par un arbre, composé de noeuds (classe `Node`) que l'on peut voir sur la **??**, cependant cette modélisation reste implicite et faite au fur et à mesure car nous ne souhaitons pas garder l'arbre complet en mémoire.

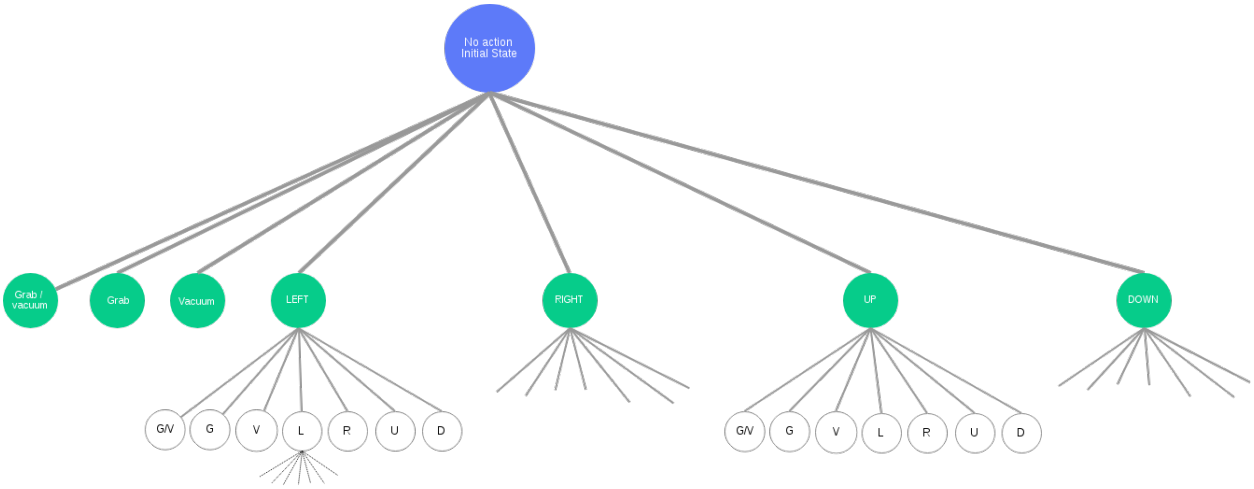

FIGURE 2 – Arbre utilisé pour l'exploration

Ce qui diffère entre les 2 algorithmes est la frontière à explorer, qui est créée de manière différente. On va donc dans les deux cas utiliser une liste chaînée permettant de modéliser cette frontière et l'ordre dans lequel elle doit être parcourue.

2.2.1 Exploration non informée

Il existe différents types d'algorithmes d'exploration. Dans le cas de notre modélisation, nous considérons que les coûts entre les liens sont identiques, et l'on souhaite un minimum d'actions pour notre agent afin qu'il perde un minimum d'énergie. Pour cela, il est plus judicieux d'utiliser un algorithme de parcours en largeur Breadth-First Search. On va parcourir les noeuds enfants dans l'ordre dans lequel ils sont ajoutés à la frontière. Afin de garantir l'optimalité de l'algorithme, on va cependant générer la frontière toujours en commençant par les actions qui seront des feuilles (et qui pourront potentiellement rapporter des points) : `GRABVACUUM`, `GRAB` et `VACUUM`.

2.2.2 Exploration informée

Dans le cas de l'exploration informée, on ajoute des informations extérieures aux noeuds, permettant de trouver une solution plus efficace. On créé donc une variante de nos noeuds : `NodeInformed`, auxquels on rajoute un attribut `heuristic`. Une fois de plus, le coût des arcs de l'arbre étant uniforme, on peut se contenter de considérer l'heuristique afin de déterminer l'ordre de la frontière. On va donc utiliser une fonction de tri de notre frontière en fonction de l'attribut `heuristic`. On va donc pouvoir parcourir nos noeuds dans la frontière en commençant par celui ayant l'heuristique la plus petite : `GRABVACUUM` car c'est le plus intéressant au niveau de la performance.

2.2.3 Condition d'arrêt et création des intentions

La condition d'arrêt pour l'exploration est testée au sein même des noeuds par une méthode `isSuccess()` qui va tester la corrélation entre l'action représentée par le noeud et l'état de l'environnement à la position de l'agent. En cas de succès, l'algorithme d'exploration s'arrête et l'on peut générer la liste d'intentions (le plan d'action) de l'agent. Pour cela on remonte dans l'arbre à partir du noeud courant. Cela est rendu possible par le fait que chaque noeud dispose d'un attribut `parent`. Pour chaque noeud de l'arbre parcouru, on ajoute l'action à la liste d'intentions de l'agent. On obtient donc, dans l'ordre, les différentes actions nécessaires pour que l'agent atteigne un succès (pour que l'agent remporte des points de performance).

3 Performance de l'agent

3.1 Calcul de la performance

Le calcul de la performance a lieu au niveau de l'environnement. La performance dépend à la fois de l'action effectuée par l'agent et de l'état de la case où il se trouve.
Cela permet de pénaliser l'action `VACCUM` si elle s'effectue sur une case où se trouve également un bijou. On associe alors une performance de 15 à l'action `VACCUM` si la case contient juste de la poussière, mais de -30 si il y a un bijou. Pour les action `GRAB` et `VACCUM`, les performances

associées sont respectivement 17 et 22.
On a de plus associé une performance de -5 aux actions `LEFT`, `RIGHT`, `UP` et `DOWN` pour représenter la consommation d'électricité. La différence entre les performance de `GRABVACCUM` et `VACCUM` réussi est volontairement faible, afin de privilégier systématique le nettoyage d'une case, et non le fait de ramasser des bijoux.

3.2 Apprentissage

Afin de permettre à l'agent d'agir dans l'environnement et d'être meilleur avec le temps pour atteindre son but, l'agent doit être capable d'apprendre pour améliorer ses performances. Pour cela, l'agent possède un tableau gardant en mémoire les différentes fréquences d'exploration que l'on utilise ainsi que la performance associé à cette fréquence. Ce tableau est mis à jour lorsque le capteur de l'agent récupère la performance dans la méthode `updateAgentState()`. Ensuite lors des 5 premières explorations, nous faisons des sauts aléatoires pour trouver différentes fréquences et différentes performances. Ainsi nous pouvons prendre en compte la meilleure performance et utiliser la fréquence associée tout en continuant de réaliser des sauts, plus petits cette fois, pour que l'agent continue d'apprendre et de perfectionner sa fréquence d'exploration. Cette fréquence intervient dans la boucle de la fonction `doAction()` de l'effecteur. Pour ce faire nous avons ajouté un timer permettant de vider la liste d'intentions et recommencer une exploration si la fréquence d'exploration est moins élevée que le temps que ça aurait prit à l'agent de réaliser toutes ses intentions.

Conclusion

Nous avons pu nous familiariser avec les différentes notions abordées en cours et ainsi pu les implémenter au travers de ce projet. On peut noter cependant qu'il s'agit d'une modification simplifiée et limitée (100 « pièces ») du fonctionnement d'un agent aspirateur intelligent. Les fonctions d'exploration et d'apprentissage prendraient plus de sens dans un modèle de taille plus importante.

Le code de notre application est disponible ici : `https://github.com/Orann/ia_work1`