Intelligence Artificielle - TP1

UQAC | Kevin BOUCHARD

Creation d’un agent intelligent d’aspirateur

Killian LACHAUX | LEO SCHIRVANIAN

2020

# Details techniques

## Environnement

L’environnement est **Complètement Observable** (senseurs omniscients).

L’environnement est **Stochastique** car les poussières et bijoux sont générés à un rythme et des positions aléatoires.

L’environnement est **Episodique** car les états précédents ne sont pas pris en compte dans la prise de décision.

L’environnement est **Dynamique** car les threads de générations sont actifs durant la délibération de l’agent.

L’environnement est en théorie **Discret** car il comporte un nombre fini d’état, cependant, le nombre d’enchainements d’actions possibles par l’agent est infini.

## Agent

L’agent est **Autonome** car l’aspirateur évolue sans intervention externe et dispose de ses propres ressources.

L’agent présente une habileté sociale **limitée** car la seule véritable interaction de l’agent consiste en l’envoi d’ordre au sol du manoir.

L’agent est **Pro-actif** car il établit constamment une liste d’instructions à exécuter à chaque itération et non seulement quand l’état de l’environnement change.

L’agent est donc un **Agent reflexe avec état interne** car il utilise un état.

## Modélisation

L’affichage et l’état du problème sont représentés par une grille d’entier allant de 0 à 7 :

0 – Case vide

1 – Aspirateur

2 – Bijoux

3 – 1 + 2

4 – Poussière

5 – 4+1

6 – 4+2

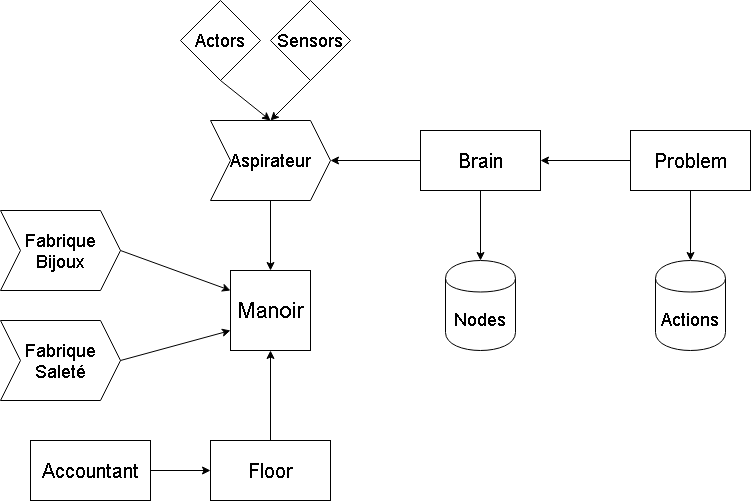
7 – 4+2+1 (Tout)

## Répartition des taches

Killian : Structure du code et Algorithme de recherche non informé (Bidirectional BFS)

Leo : Algorithme de recherche informé (NOM DE TON ALGO)

# Structure du code



## Explications

Le cœur du code est le **Manoir**, qui est un singleton recueillant la boucle principale d’exécution.

Il contient le **Floor**, objet contenant l’état des 25 salles du manoir ainsi que les méthodes pour modifier cet état. Il contient également l’**Accountant** dont le rôle est d’établir la mesure de performance.

Les deux **Fabriques** sont deux threads interagissant avec le sol du manoir pour générer bijoux et saleté de manière aléatoires.

Le manoir contient également l’**Aspirateur**, autre thread et agent intelligent du problème. L’aspirateur utilise ses **Sensors** pour mettre à jour son état interne, et utilises les **Actors** pour interagir avec le manoir.

La classe **Brain** est la classe dont le rôle est d’effectuer l’exploration en utilisant le **Problem**, objet représentant le problème à résoudre et contenant les méthodes de succession et retro succession. Le Brain va effectuer l’exploration pour aller de l’état courant a l’état désiré et retourner une liste d’action à effectuer. Le brain est l’objet qui conserve l’arbre d’exploration courant, une liste de nodes.

# Exploration

## Exploration non-informée

L’agent effectue son exploration non-informée en utilisant un algorithme de recherche en largeur bidirectionnelle, c’est-à-dire qu’il recherche et construit sa chaine d’action en partant en même temps de l’état initial et de l’état désiré puis en joignant les résultats lorsqu’un nœud commun est atteint.

L’exploration s’effectue donc en explorant les nœuds sur le modèle FIFO. Afin d’optimiser la vitesse d’exécution, l’algorithme va supprimer automatiquement les nœuds qui n’apportent pas de modification à l’état parent (ex : nettoyer une case vide ou se déplacer en direction d’un mur). Il va également ignorer les nœuds enfants d’une action nulle (afin de limiter le nombre d’enfants ‘inutiles’ le seul enfant de l’action ‘nothing’ sera toujours ‘nothing’, cela afin de limiter l’exploration des chemins ou plus d’une des actions est nulle, sauf si l’état courant est celui désiré).

Dans un souci de lisibilité et afin d’avoir un affichage clair, l’aspirateur est restreint et ne peut effectuer que 2 actions à chaque itération, avec 0.5 seconde entre chaque. Cela permet un affichage plus fluide, cependant, il serait tout à fait possible (et même plus efficace) de faire exécuter toutes ses actions simultanément à l’aspirateur des qu’un état est résolu (cela évite que l’état soit plus complexe si un item apparait sur le sol durant l’exécution des taches).

### Limites

A l’heure actuelle, l’algorithme est capable d’explorer un manoir 5x5 et tourne sans problème pendant 2h et a atteint une mesure de performance de plus de 200 000, avec un taux d’apparition d’environ une poussière et un bijou toutes les 4 secondes.

Cependant, il existe des états spécifiques bloquants. Par exemple lorsque l’aspirateur est dans un coin et que le sol contient 2 items ou plus situé a des coins opposés de l’aspirateur et l’un de l’autre, le temps de résolution est plus grand que le temps d’actualisation de l’état et l’état deviens donc de plus en plus complexe, rendant la résolution plus longue que la prochaine actualisation etc…

La fonction d’exploration non-informée ne prend actuellement pas en compte le cout mesuré par la mesure de performance. Le fonctionnement de Bidirectional BFS nous assure que le chemin pris sera toujours le nombre d’actions le plus court mais pas nécessairement les actions les moins couteuses (ex : l’algorithme trouveras toujours le chemin de déplacement optimal, mais pour lui, il est plus efficace pour se rendre à l’état désiré de simplement aspirer une case contenante a la foi de la poussière et des bijoux, que de d’abord ramasser les bijoux, puis d’aspirer.)

Cela pourrait être résolu simplement en triant les nœuds de la frontière selon leur cout a chaque itération, mais se ferait au cout de vitesse d’exécution, car a chaque exécution il faut trier les éléments de deux listes (frontière depuis l’état initial et le goal), et qu’en raison du branching factor moyen, ces listes se remplissent vite. De plus cela est fait par l’exécution informée.

### Optimisation

Afin d’améliorer les performances de l’algorithme et de supprimer le blocage par état complexe, nous pourrions envisager une méthode de séparation de l’état courant en plusieurs sous-état chacun contenant un seul item. Puis l’agent pourra résoudre chaque état séparément (beaucoup moins de nœuds à explorer car profondeur plus petite) et exécuter les opérations de chaque résolution a la chaine. Afin d’éviter le retour au point de départ dans un tel scenario, il suffirait de modifier l’état initial pour correspondre à la position attendue de l’aspirateur après résolution du sous-état précèdent.

Le fonctionnement des fabriques utilise un état interne fixe, à savoir une matrice de probabilité représentant les chances d’apparitions. Nous souhaitions implémenter, afin de développer l’aspect apprenant de notre agent, une méthode pour lui de déterminer un état désiré optimal en se basant sur les apparitions observées des objets. L’agent aurait pu pour ça créer une matrice mise à jour à chaque changement d’état ou chaque case possède une valeur, la plus élevée étant la plus désirable. En combinant le nombre d’apparition et un flou gaussien on peut ainsi obtenir la case présentant le plus d’intérêt pour permettre à l’aspirateur d’attendre au plus proche des prochaines apparitions. Cela n’a pas été fait par manque de temps.