UQAC

Luke ABOUGIT, Erwan Dufour, Mathis Certenais

Intelligence Artificielle

Compte rendu TP1 : Création d'un agent intelligent

L'objectif de ce TP est de générer un agent intelligent omniscient ayant pour rôle d'aspirer de la poussière et de ramasser des bijoux dans un espace modifié en continu. L'environnement sera constitué d'une carte contenant 25 pièces sur laquelle il est posé de manière aléatoire des bijoux et poussière dans une boucle à l'infinie. L'agent doit contenir deux types d'algorithme d'exploration : un non-informé et un informé, pour pouvoir nettoyer l'environnement.

Introduction

Explication de la méthodologie appliquée : construction de l'environnement, affichage dans un premier lieu, puis construction de l'arbre et exploration.

La conception de notre programme se base sur le fait que nous souhaitions qu'il n'y ait aucune communication directe entre l'environnement (génération de bijoux et poussières aléatoires dans le manoir) et le robot (capture du manoir, choix de son chemin basé sur un algorithme d'exploration). Nous avons donc trois threads dans notre programme: TR pour ThreadRobot qui s'occupera du comportement du robot, TE pour ThreadEnvironnement qui permet de générer les poussières et bijoux aléatoirement puis TI pour ThreadInterface qui gère l'affichage et s'occupe de gérer le manoir et le score. TI peut-être vu comme un pont entre TR et TE pour qu'ils puissent être sur le même manoir.
La communication entre les threads se fera à travers une queue synchronisée que partagera chacun de ces threads. Nous avons donc TE et TR comme producteur dans la queue et TI comme consommateur de la queue.

N'oubliez pas de lire le README pour les instructions du lancement.
Lien github: [wall-e]

Nous pouvons apercevoir ci-dessous les images de notre programme en fonctionnement:

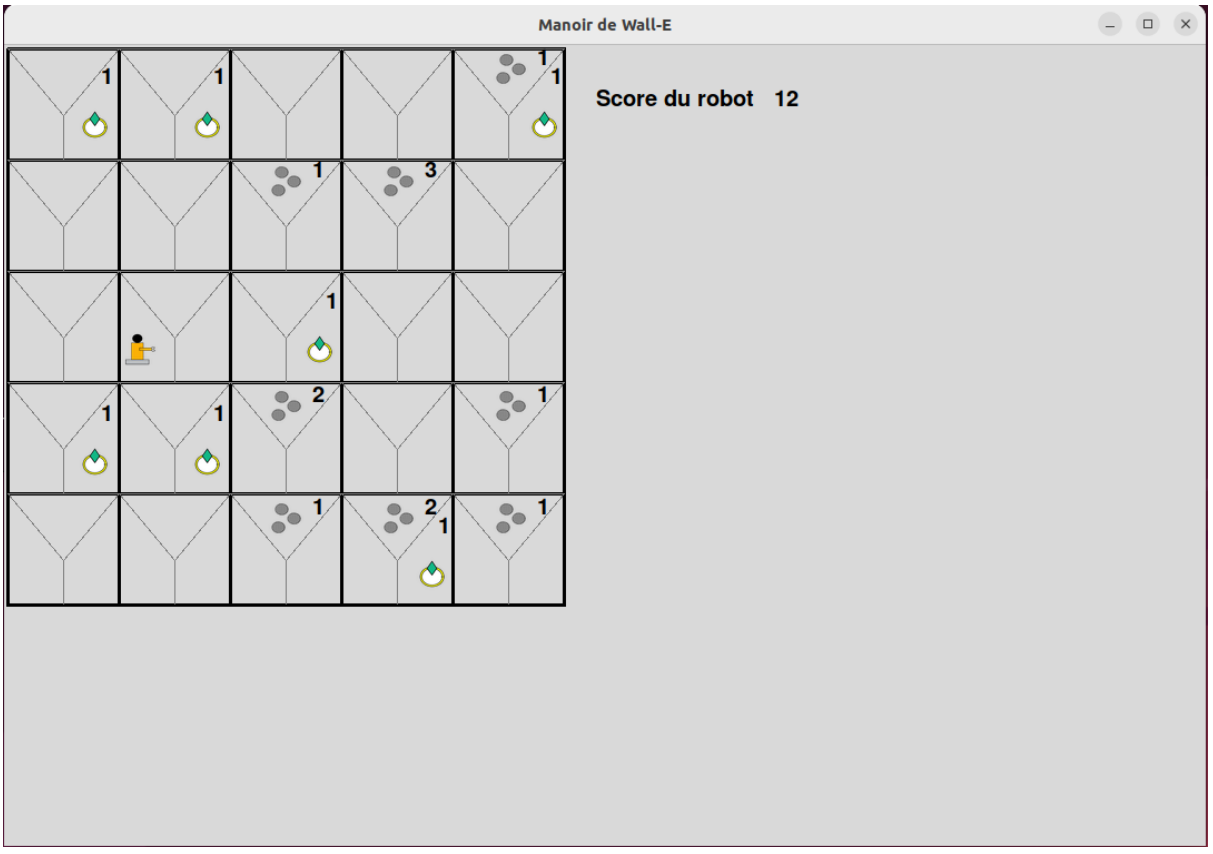

Figure 1: Capture d'écran de l'interface du projet dans les débuts du lancement

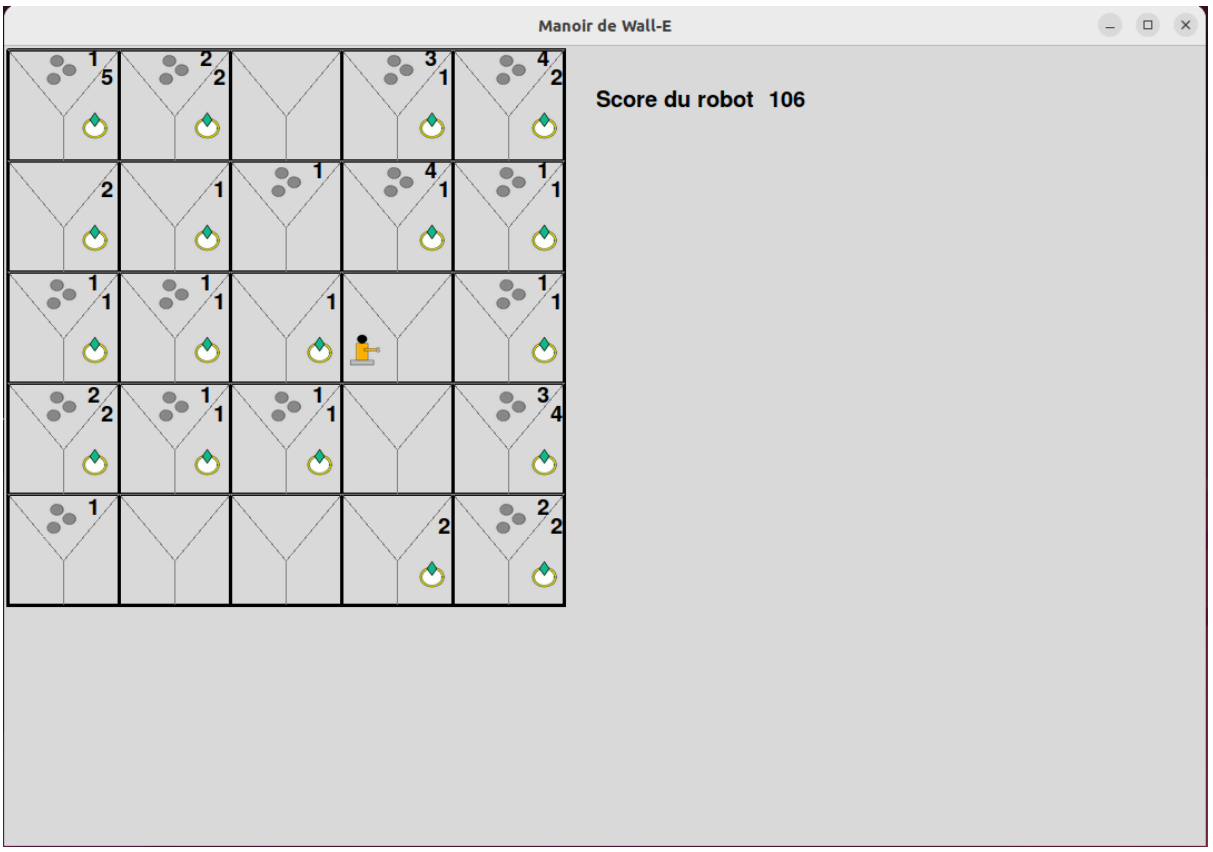

Figure 1: Capture d'écran de l'interface du projet dans un état avancé

Description de l'environnement

Propriété

L'environnement est le Thread le plus simple du projet. Il consiste uniquement à gérer de façon aléatoire des poussières et des bijoux. Ces deux derniers éléments possèdent chacun une classe à eux qui hérite de la classe 'Object_scene'. La classe bijoux ou poussières possèdent leurs propres attributs respectifs (chemin de l'image, nombre de points quand aspirer ou ramasser, leur position dans la matrice du manoir). Nous avons aussi une classe aspirateur qui hérite de 'Object_scene' permettant l'affichage de l'aspirateur.

Structure

La structure de l'environnement est représentée d'une classe qui hérite de l'objet Thread.threading, qui se nomme ThreadEnvironnement, et qui permet dans une boucle infinie de générer des poussières et objets avec un sleep aléatoire entre x et y secondes. Nous avons donc la génération du futur objet qui est aléatoire (peut être une poussière, un bijoux ou rien) et le temps entre deux apparitions (le temps est compris entre x et y secondes), X et Y deux variables définies par l'utilisateur dans le code.

L’environnement est un tableau à 3 dimensions contenant dans chacune de ces cases une liste d'objets ‘Object_scene’. La classe ‘Object_scene’ est la classe mère des poussière, bijoux et aspirateur.

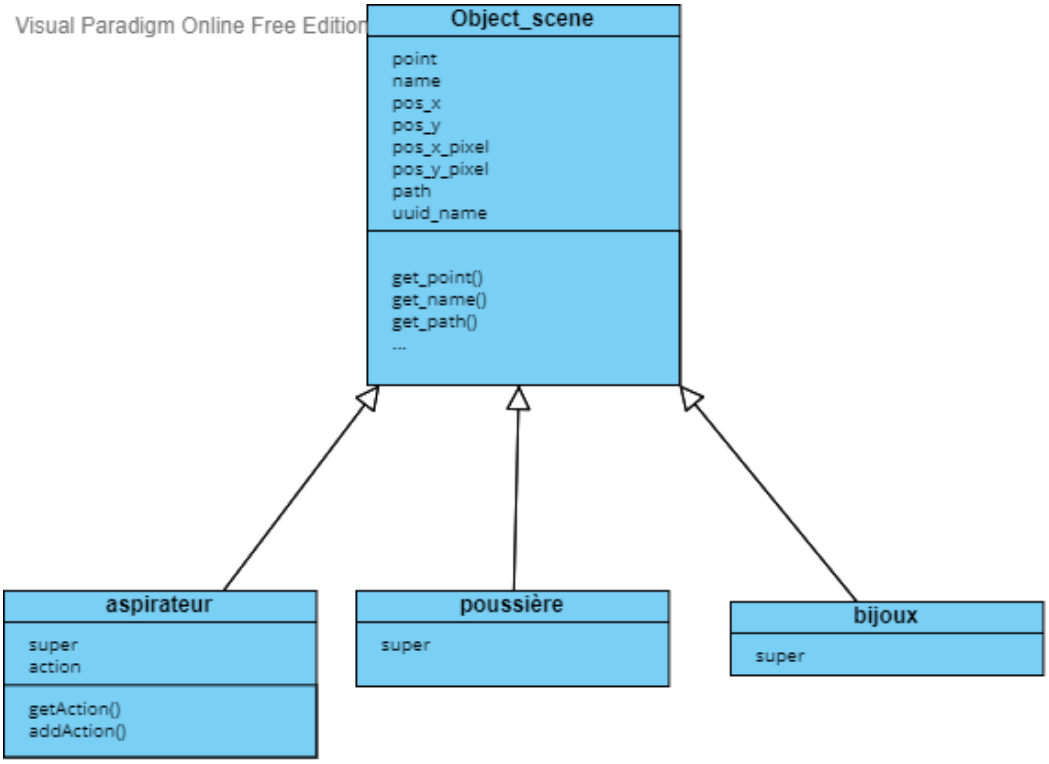

Figure 3: Diagramme UML de la classe Object_scene avec ses classes filles

Description de l'agent

Propriété et type d’agent

Le ThreadRobot est une classe qui hérite de Thread.threading. Cette classe a pour rôle, dans une boucle infinie, de réaliser les 4 étapes nécessaires au fonctionnement de notre agent intelligent. Il appelle la fonction capture, puis transforme le résultat en arbre, et ainsi exécute l’algorithme d’exploration avant d’exécuter le tout.
Ces étapes sont décrites de façon plus précise dans la partie Modélisation.
Cette classe permet aussi de stocker les attributs nécessaires à ces étapes.

Structure et fonction d’agent

Lorsque le Robot capte son environnement, il procède par une construction d’un arbre représentant l’environnement de manière hiérarchique, pour ensuite y appliquer une exploration.

Description de l’interface

Propriété

L’interface est un élément majeur qui permet d'interpréter graphiquement le code de l’aspirateur intelligent. Son rôle est de faciliter la compréhension du système par l’utilisateur en le rendant ergonomique, par exemple en remplaçant le tableau qui constitue le manoir par l’affichage d’un quadrillage, ou encore l'affichages des images des éléments qui constitue le manoir, c’est-à-dire le robot, les poussières et les bijoux.

L’interface, qui se situe intégralement dans le fichier “interface.py”, est le consommateur en recevant les informations du robot et de l'environnement qui sont producteurs, en utilisant un bus(queue) synchronisé.

Structure

Pour ce premier travail pratique, nous avons utilisé la librairie Tkinter, autrement dit Tool kit interface, qui permet la création d'interfaces graphiques pour le langage Python. Cette librairie compte un nombre important de widgets, permettant de générer une interface graphique complète et intuitive. Nous allons voir à la suite de ce rapport les différents widgets utilisés sur le projet.

Canvas
Le widget Canvas nous permet d'afficher divers éléments graphiques qui sont expliqués plus bas en détail .

Canvas text objects
Ce widget permet de créer des zones de textes, que l'on utilise pour l'affichage du compteur de poussière et de bijoux présent dans chaque case du manoir, mais aussi du score de l’aspirateur. Une des fonctionnalités du widget est d’utiliser un tag qui permet de donner un identifiant à un élément. Dans notre cas, nous les utilisons pour maintenir les différents compteurs à jour en fonction des actions du robot (déplacement, aspiration, ramassage) et du manoir (génération d’élément).

Canvas line objects
Afin de simuler le décor du manoir, nous avons utilisé le widget ligne qui permet de créer, comme son nom l’indique, des lignes, constitué d’un certain nombre de segments, qui peuvent être droit ou courbé, et sont reliés bout à bout. Ce widget dispose de plusieurs fonctionnalités comme le changement de l'épaisseur du trait ou encore la couleur.

Sur notre projet d’aspirateur autonome, en fonction des intentions de l'utilisateur quant à la taille du manoir qu’il indiquera, le tracé du quadrillage en plusieurs cases sera plus ou moins grand. De plus, nous avons utilisé ce dernier widget pour séparer chaque case en 3 parties pour y incorporer les images des poussières, des bijoux et du robot.

Label

Le label est une étiquette qui est utilisée pour mettre en place des boîtes d'affichage pour y incorporer des éléments comme du texte ou encore des images, et qui peuvent être modifiés à tout moment. Il est important de noter qu'une étiquette ne peut utiliser qu'une seule police à la fois pour afficher le texte. On notera qu'il s'agit d'un tag plus sophistiqué.

Nous utilisons ce widget afin d'identifier les éléments qui sont générés dans le manoir, c'est-à-dire les poussières ou les bijoux, ainsi que le robot. Lorsque le robot effectue un mouvement, nous allons donc, grâce au label, supprimer l'image du robot qui se situe à l'emplacement précédent, et en générer un nouveau à l'état actuel. Même combat pour les images des poussières et des bijoux qui seront supprimés lorsqu'ils seront respectivement aspirés et ramassés.

Modélisation

Nous avons utilisé une méthode BFI pour notre agent. De ce fait, nous avons donc quatre étapes importantes. La première étape est celle de prendre conscience de son environnement à l'aide de ses capteurs. La seconde est de transformer cette capture en un objet qui permettra aux algorithmes d'exploration de fonctionner dessus facilement, ainsi faciliter le choix entre deux algorithmes d'exploration. La troisième étape consiste à exécuter son algorithme d'exploration pour définir les actions à faire. La dernière étape consiste uniquement à exécuter le plan défini auparavant.

Modélisation de la perception

Pour que le capteur prenne conscience de son environnement, nous voulons qu'il fasse une capture de cet environnement à un instant T et si un changement de l'environnement arrive, l'image que le robot a de l'environnement ne doit pas être modifiée. Nous avons donc le ThreadRobot qui appelle une fonction dans l'objet ThreadInterface, et qui renvoie une copie profonde de la matrice d'environnement. Ainsi, notre objet capturé par le Robot ne sera pas modifié par le ThreadInterface étant donné que nous en avons une copie.

Modélisation de l'arbre

Pour pouvoir exécuter nos algorithmes d'exploration facilement nous avons créé une couche d'abstraction qui se matérialise par un arbre. Nous avons donc une classe noeud qui contient comme attribut le noeud racine. Ainsi nous avons aussi une classe Noeud qui a 4 fils: Nord, Sud, Est, Ouest. Nous avons choisi cet ordre arbitrairement, mais il a une importance sur le comportement des algorithmes d'explorations.
Cette couche d'abstraction permet d'implémenter les algorithmes de façon récursive plus facilement.

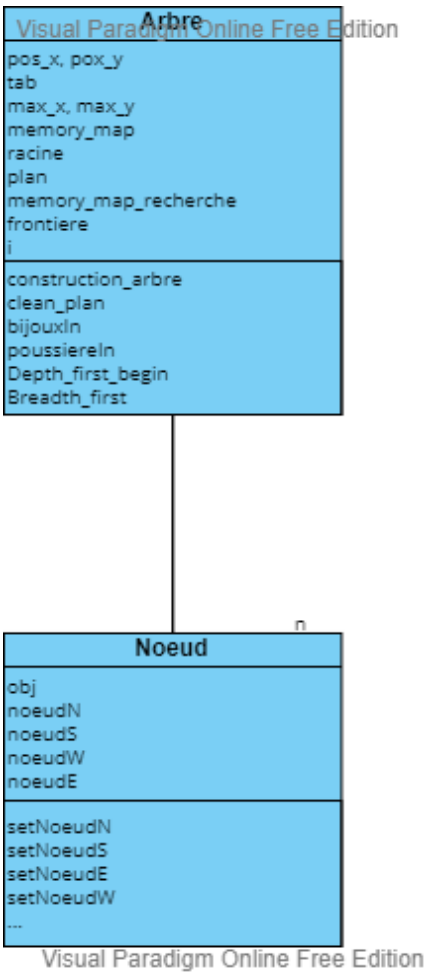

Figure 4: Diagramme UML de la structure de l'arbre

La création de l'arbre se fait avec une méthode récursive. Pour éviter les boucles un attribut sous forme de dictionnaire, memory_map, est nécessaire. Il permet de se rappeler des nœuds déjà visités et ainsi de ne plus les visiter. De plus, notre arbre part du principe que chaque déplacement coûte 1 en énergie. Si le Nord d'un nœud est en dehors de la matrice du manoir, nous lui assignons une valeur None pour représenter ce cas.

Modélisation de l'action

Afin de changer d'algorithme, nous devons changer la fonction qui est appelée dans le fichier Robot.py -> classe ThreadRobot -> fonction run -> changer la fonction entre la création de l'arbre et l'appel de la fonction JustDoIt().

Algorithme non informé

Trois algorithmes non informés d'explorations sont implémentés: Gready_search, Breadth_first et Depth_first qui se trouvent dans la classe Arbre.py.

Algorithme informé

Ici nous avons fait le choix d'implémenter la méthode d'exploration 'Greedy Search' qui est estimée à l'aide de la distance de manhattan comme fonction heuristique. Le principe est d'étendre notre exploration au nœud qui semble le plus près du but. Pour ce faire, l'algorithme est divisé en deux parties :

- Trouver le but (donc poussière la plus proche de l'aspirateur)
- Explorer notre arbre hiérarchique en utilisant la distance de Manhattan comme choix d'un nœud fils.

De la même manière que les algorithmes de recherche précédent, après construction de l'arbre, l'aspirateur explore les nœuds fils d'une racine en y choisissant celle minimisant la distance de manhattan jusqu'à trouver son but.
La distance de Manhattan a pour but de minimiser les coûts de déplacement en ayant comme trajectoire en forme de L.

Modélisation de l'apprentissage

Nous allons expliquer le principe de l'apprentissage par le système de points négatif/positif et la fréquence étudiée optimale à la suite de ce rapport.
Nous avons mis en place comme règle: une poussière ramassée donne 5 points, un bijoux ramassé donne 10 points et un bijoux aspiré -10 points. Quand l'aspirateur aspire, il aspire toutes les poussières et bijoux présents dans la case. Quand l'aspirateur ramasse, il ramasse tous les bijoux présents sur la case.
La génération de poussières ou bijoux se fait toutes les 0.1 secondes. Chaque action du robot dure 0.5 seconde. Entre deux boucles du robot, nous avons un temps de refroidissement qui est de 0.3 seconde.
Après plusieurs essais, nous avons recalculer le chemin après 1, 2, 3, 4 ou une infinité d'actions, nous obtenons les résultats ci-contre:

Min \ Nb d'actions	1	2	3	4	∞
2	1564	1500	1533	1590	1400
4	3800	3500	3492	3900	2950
6	5000	5348	5200	5870	5727
8	7000	9000	9000	8477	8800

Figure 5: Tableau de score en fonction du nombre de minutes et du nombre d'actions avant la réévaluation

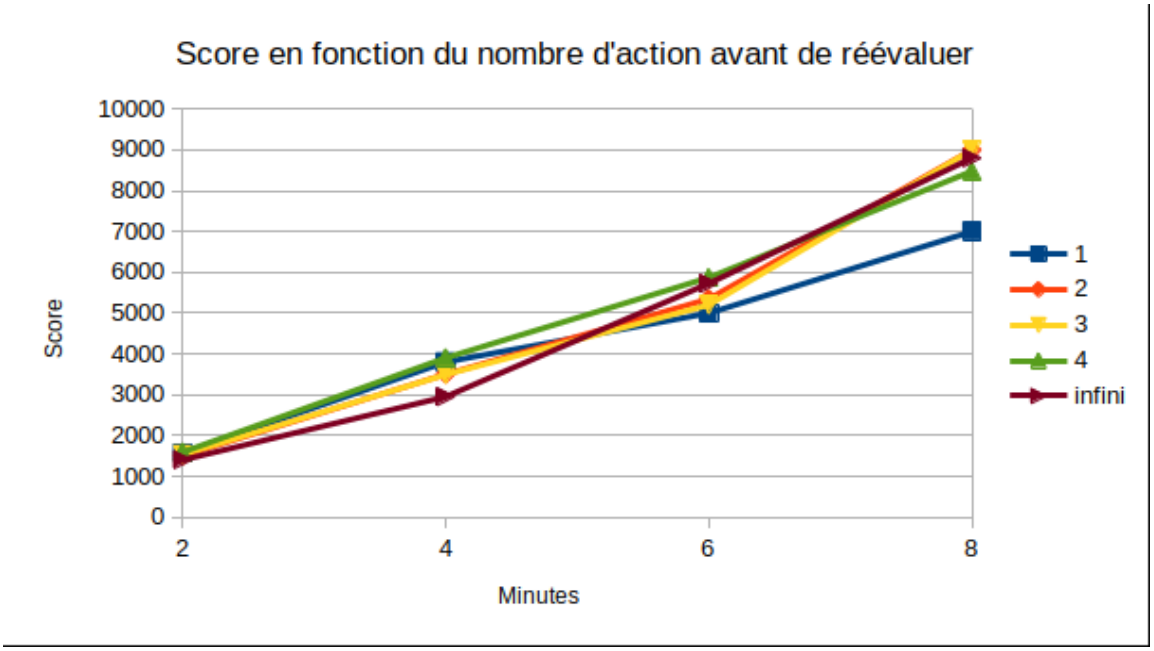

Figure 5: Graphique des scores en fonction du nombre d'action avant la réévaluation

Ce graphique nous permet de dire que le mieux est de relancer la capture et la recherche de chemin toutes les 2 actions du robot. Ce résultat peut varier énormément si on modifie la durée pour l'apparition des poussières ou bijoux, mais aussi le temps entre chaque action, le temps entre chaque boucle du robot.