8INF846 Intelligence Artificielle
Travail No. 1
Création d'un agent aspirateur

Florian BARTHOLIN - BARF21129805
Ahlam HABBARI - HABA19539806

Compilation

Le projet a été réalisé sur Eclipse avec le JDK 12.

Le dossier Ressources doit être au même niveau que le dossier src.

Type d’environnement (propriété)

L'environnement est complètement observable, l’agent a connaissance de sa position ainsi que de l’état de chaque pièce du manoir. L'environnement est déterministe et séquentiel, chaque action a un résultat connu. L'environnement est dynamique, de la poussière ou des bijoux peuvent apparaître à tout moment. Enfin, l'environnement est discret et est composé d’un seul agent.

Modélisation de l'environnement

L'environnement implémente le design pattern du singleton car il ne peut y avoir qu’un seul environnement. Il est exécuté dans un thread différent. Il contient le module de calcul de performance ainsi que la modélisation de l'environnement réel dans lequel évolue l’agent aspirateur. L'environnement réel (un manoir) est représenté par une unique classe qui contient la position du robot et deux tableaux à deux dimensions de booléen, une pour la poussière et une pour les bijoux. Les deux tableaux sont de dimensions identiques et contiennent une case par pièce dans le manoir.

Lorsque l’objet de la classe Manor qui contient la représentation du manoir est récupéré par un autre objet, c’est toujours une copie de l’objet Manor qui est passée. Le véritable environnement ne peut être modifié que par la classe Environnement.

Type d’agent

L’agent créé est un agent basé sur les buts. Pour déterminer les actions qu’il doit effectuer, il utilise soit un algorithme d’exploration non-informé (Iterative deepening search), soit un algorithme d’exploration Informé (A*). Un petit module d’apprentissage lui permet de déterminer s'il doit exécuter les solutions retournées par son exploration jusqu’à la fin ou s'il doit réaliser son exploration plus souvent.

Modélisation de l’agent (fonction d’agent)

L’agent est modélisé avec un état mental sous la forme BDI. Son but (désir) est représenté par un objet de la classe Manor ainsi que quelques conventions. Par exemple, si la position du robot n’a pas d’importance alors dans le but celle-ci est à -1.

Ses croyances (beliefs) sont aussi représentées par un objet de la classe Manor.

Ses intentions sont représentées avec une pile (liste LIFO). Cette pile contient des éléments d’une énumération contenant toutes les actions possibles de l’agent.

Pour accéder à l'environnement, l'agent doit forcément passer par ses actionneurs ou ses capteurs, ils sont modélisés par deux classes.

Chaque fois que l'agent veut réaliser une exploration, il crée un problème à partir de son but et de ses croyances. Le problème est modélisé avec une classe (voir partie sur la modélisation du problème). Ce problème est ensuite passé à un des deux algorithmes d'exploration qui le résoudra et renverra une pile d'actions à effectuer.

Modélisation de l'action

Pour réaliser une action l'agent passe par la classe Effector. Cette classe contient une méthode par action que l'agent peut réaliser. Chacune de ses actions prend du temps.

Une énumération est utilisée pour connaître toutes les actions possibles.

Modélisation de la perception

Pour percevoir son environnement l'agent passe par la classe Sensor. Cette classe récupère et enregistre toutes les informations de l'environnement lorsque l'agent le demande. L'agent peut alors récupérer ces informations pour mettre à jour son état mental.

Modélisation du problème

Le problème est modélisé avec la classe *Problem*. Il contient un état initial, le but, la liste de toute action possible ainsi qu'une méthode pour tester si un état correspond au but.

L'état initial est de la classe *State*. Cette classe est une classe générique qui permet aux algorithmes d'exploration de traiter tout problème modélisé avec cette classe. Notre état initial étant de la classe *Manor* cette classe hérite de *State* et implémente les méthodes *equals()* et *hashCode()*.

Le but est aussi un élément de la classe *Manor*.

La méthode qui teste si un état correspond au but permet de faire la comparaison de façon plus laxiste que la méthode *equals()*. Cela permet par exemple d'avoir un but qui correspond à plusieurs états. Si cette méthode est bien réalisée, elle peut être utilisée pour plusieurs buts différents (ex : ramasser que la poussière ou ramasser toute la poussière et tous les bijoux ou se rendre dans une pièce précise).

La liste des actions possibles contient des classes héritant de la classe *Action*. Chacune de ses classes filles permet de donner l'état résultant à la réalisation d'une action sur un état quelconque, ainsi que le coût de cette action. Il y a donc une classe fille de la classe Action par action que peut réaliser l'agent.

Exploration non-informée

Pour l'exploration non-informée, on a utilisé l'algorithme *Iterative deepening search*. Nous l'avons implémenté de façon récursive. De plus, nous avons ajouté une stratégie pour éviter les boucles. Cette stratégie est de vérifier si l'état actuel n'est pas aussi un état d'un ancêtre à cet état, si cela est le cas, nous ne réalisons pas l'expansion de ce nœud.

Lorsque la solution est en profondeur 19 dans l'arbre, trouver la solution avec notre implémentation de *Iterative deepening search* cela peut prendre plus d'une minute pour trouver la solution. Au delà de cette profondeur, l'algorithme devient trop long par rapport à la vitesse d'apparition de la poussière dans la simulation de l'environnement.

Exploration informée

Pour l'exploration informée on a utilisé l'algorithme A*. Nous avons implémenté cet algorithme selon l'algorithme de *graph search* pour éviter que des états soit explorés plusieurs fois.

L'heuristique utilisée est : le nombre de pièces sales, plus le nombre de bijoux qui sont dans une pièce sale, plus la longueur du chemin entre le robot et la pièce sale la plus éloignée du robot.

Mesure de performance

La mesure de performance est réalisée dans l'environnement, elle est calculée sur des périodes de 10 secondes. Son calcul est basé sur le ratio du nombre de pièces nettoyées sur la quantité d'énergie consommée, pondéré par le pourcentage de pièces propres, multiplié par 100, plus un malus pour chaque bijoux aspiré.
Performance = ((nbPiècesNettoyées / EnergieConsomée) * (nombreDePiècesPropres/25))*10 - 5*nombreDeBijouxAspirés

Apprentissage

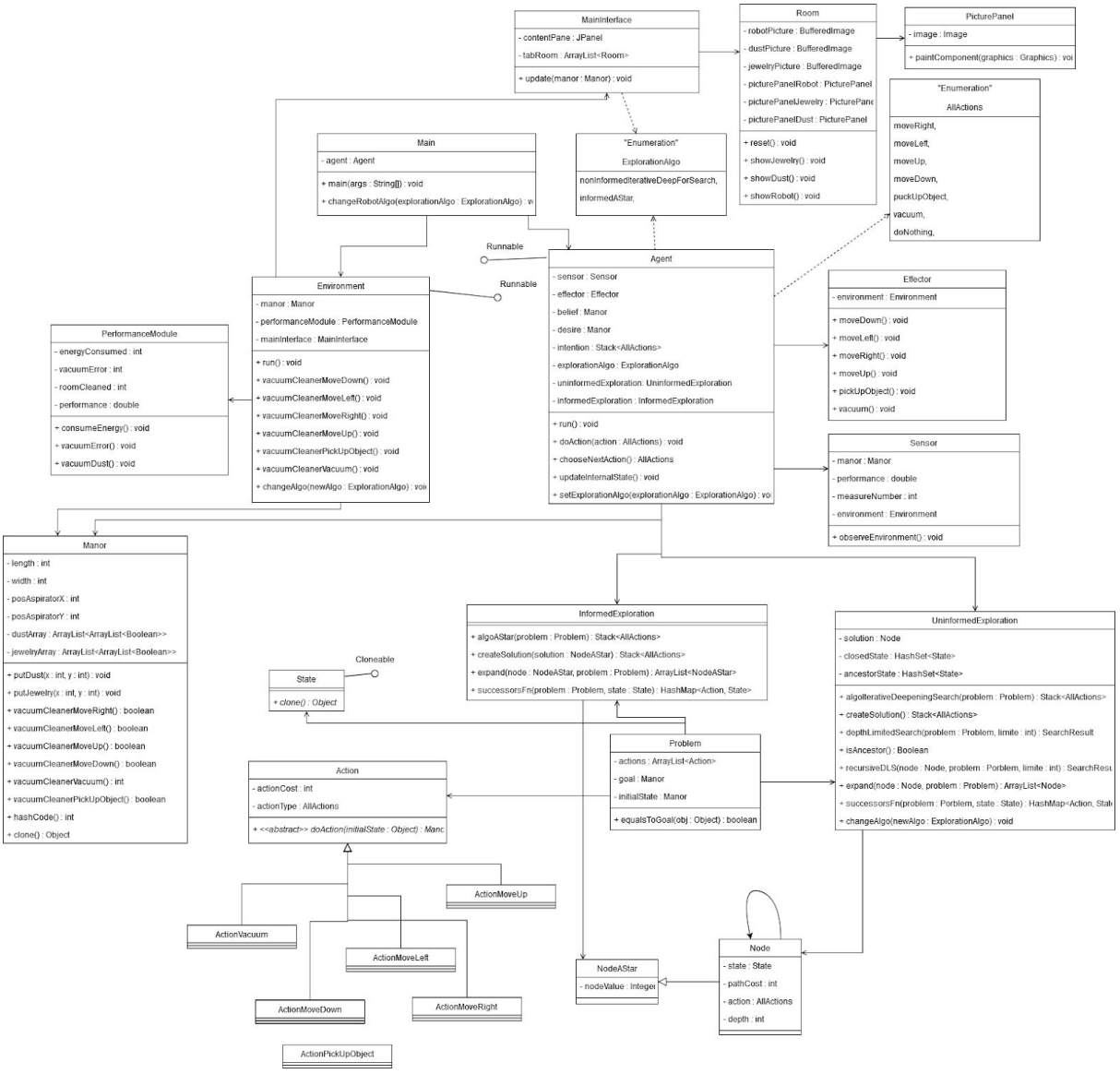
Le but de l'apprentissage est de savoir s'il faut exécuter l'exploration après avoir effectué tout le plan d'actions fourni par l'exploration précédente, ou s'il faut recommencer l'exploration après par exemple au maximum 5 actions.

Notre implémentation de l'apprentissage est simple. Pour différentes valeurs on effectue trois mesures de performance (une mesure toutes les 10 secondes). Après avoir effectué 3 mesures pour toutes les valeurs que l'on veut tester, on réalise la moyenne des performances pour chaque valeur, et on prend la meilleure valeur pour la suite de la vie de l'agent.

Lorsque l'algorithme d'exploration change, l'apprentissage recommence à zéro.

Annexe

Voir diagramme de classes UML représentant la structure du projet.


MainInterface
- contentPane : JPanel
- tabRoom : ArrayList<Room>
+ update(manor : Manor) : void
Room
- robotPicture : BufferedImage
- dustPicture : BufferedImage
- jewelryPicture : BufferedImage
- picturePanelRobot : PicturePanel
- picturePanelJewelry : PicturePane
- picturePanelDust : PicturePanel
+ reset() : void
+ showJewelry() : void
+ showDust() : void
+ showRobot() : void
PicturePanel
- image : Image
+ paintComponent(graphics : Graphics) : voi
"Enumeration"
AllActions
moveRight,
moveLeft,
moveUp,
moveDown,
puckUpObject,
vacuum,
doNothing,
Main
- agent : Agent
+ main(args : String[]) : void
+ changeRobotAlgo(explorationAlgo : ExplorationAlgo) : vo
"Enumeration"
ExplorationAlgo
nonInformediterativeDeepForSearch,
informedAStar,
Runnable
Runnable
Agent
- sensor : Sensor
- effector : Effector
- belief : Manor
- desire : Manor
- intention : Stack<AllActions>
- explorationAlgo : ExplorationAlgo
- uninformedExploration: UninformedExploration
- informedExploration : InformedExploration
+ run() : void
+ doAction(action : AllActions) : void
+ chooseNextAction() : AllActions
+ updateInternalState() : void
+ setExplorationAlgo(explorationAlgo : ExplorationAlgo) : voi
Environment
- manor : Manor
- performanceModule : PerformanceModule
- mainInterface : MainInterface
+ run() : void
+ vacuumCleanerMoveDown() : void
+ vacuumCleanerMoveLeft() : void
+ vacuumCleanerMoveRight() : void
+ vacuumCleanerMoveUp() : void
+ vacuumCleanerPickUpObject() : void
+ vacuumCleanerVacuum() : void
+ changeAlgo(newAlgo : ExplorationAlgo) : voi
Effector
- environment : Environment
+ moveDown() : void
+ moveLeft() : void
+ moveRight() : void
+ moveUp() : void
+ pickUpObject() : void
+ vacuum() : void
PerformanceModule
- energyConsumed : int
- vacuumError : int
- roomCleaned : int
- performance : double
+ consumeEnergy() : void
+ vacuumError() : void
+ vacuumDust() : void
Sensor
- manor : Manor
- performance : double
- measureNumber : int
- environment : Environment
+ observeEnvironment() : void
Manor
- length : int
- width : int
- posAspiratorX : int
- posAspiratorY : int
- dustArray : ArrayList<ArrayList<Boolean>>
- jewelryArray : ArrayList<ArrayList<Boolean>>
+ putDust(x : int, y : int) : void
+ putJewelry(x : int, y : int) : void
+ vacuumCleanerMoveRight() : boolean
+ vacuumCleanerMoveLeft() : boolean
+ vacuumCleanerMoveUp() : boolean
+ vacuumCleanerMoveDown() : boolean
+ vacuumCleanerVacuum() : int
+ vacuumCleanerPickUpObject() : boolean
+ hashCode() : int
+ clone() : Object
InformedExploration
+ algoAStar(problem : Problem) : Stack<AllActions>
+ createSolution(solution : NodeAStar) : Stack<AllActions>
+ expand(node : NodeAStar, problem : Problem) : ArrayList<NodeAStar>
+ successorsFn(problem : Problem, state : State) : HashMap<Action, State>
UninformedExploration
- solution : Node
- closedState : HashSet<State>
- ancestorState : HashSet<State>
+ algoIterativeDeepeningSearch(problem : Problem) : Stack<AllActions>
+ createSolution() : Stack<AllActions>
+ depthLimitedSearch(problem : Problem, limite : int) : SearchResult
+ isAncestor() : Boolean
+ recursiveDLS(node : Node, problem : Porblem, limite : int) : SearchResu
+ expand(node : Node, problem : Problem) : ArrayList<Node>
+ successorsFn(problem : Porblem, state : State) : HashMap<Action, Stat
+ changeAlgo(newAlgo : ExplorationAlgo) : void
Cloneable
State
+ clone() : Object
Problem
- actions : ArrayList<Action>
- goal : Manor
- initialState : Manor
+ equalsToGoal(obj : Object) : boolean
Action
- actionCost : int
- actionType : AllActions
+ <<abstract>> doAction(initialState : Object) : Manc
ActionMoveUp
ActionVacuum
ActionMoveLeft
ActionMoveDown
ActionMoveRight
ActionPickUpObject
NodeAStar
- nodeValue : Integer
Node
- state : State
- pathCost : int
- action : AllActions
- depth : int