Intelligence Artificielle- Projet 1 : VacuumAI

Étudiants :

Lucas BOUSSELET
Nicolas GILBERT
Ramachandran SUNTHARASARMA

Enseignant :

Kévin BOUCHARD

Table des matières

Préface

Monsieur BOUCHARD,

Lors de notre dernière entrevue dans votre manoir, vous nous aviez fait part de vos difficultés à maintenir votre demeure propre et ordonnée.

Nous avons donc l'honneur de vous présenter un premier concept de notre prototype appelé VacuumAI. Ce logiciel simule le comportement de votre futur Aspirobot T-0.1, il contient une présentation du robot, ainsi qu'une représentation visuelle de son évolution dans un modèle représentant votre demeure de 11 pièces, tout en veillant à ramasser les bijoux qui auraient pu être abandonnés par mégarde.

Lancement du programme

Le programme est disponible sur un Git à l'adresse suivante : https://github.com/Ichinin/VaacumAI.

Il est rédigé en C# et nécessite Visual Studio pour l'édition ainsi que Microsoft .NET Framework 4.5 pour son bon fonctionnement.

Le fichier .sln à lancer avec Visual Studio est localisé dans VacuumAI et se nomme VacuumAI.sln

Il est également possible de lancer directement le programme en démarrant le script Start_VacuumAI.bat localisé dans le dossier VacuumAI.

Pour terminer le programme, il suffit de fermer la console du programme.

1. Description du problème

Dans un environnement, constitué de 11 pièces, apparaissent aléatoirement de la poussière et des bijoux. La problématique est de concevoir un agent intelligent capable d'effectuer trois tâches distinctes, ramasser ces bijoux, aspirer la poussière et pouvoir se déplacer.

Une pénalité est appliquée si l'aspirateur aspire les bijoux et un gain de points lorsqu'il parvient à maintenir l'environnement propre, débarrassé de toutes poussières. Basé sur des buts, l'agent doit intégrer un état mental, calqué sur le modèle BDI (Belief-Desire-Intention).

Afin de garantir la totale autonomie de fonctionnement du robot, il sera exécuté dans un Thread unique, séparé du Thread contenant l'environnement. Le tout sera exécuté depuis un programme commun *main().*

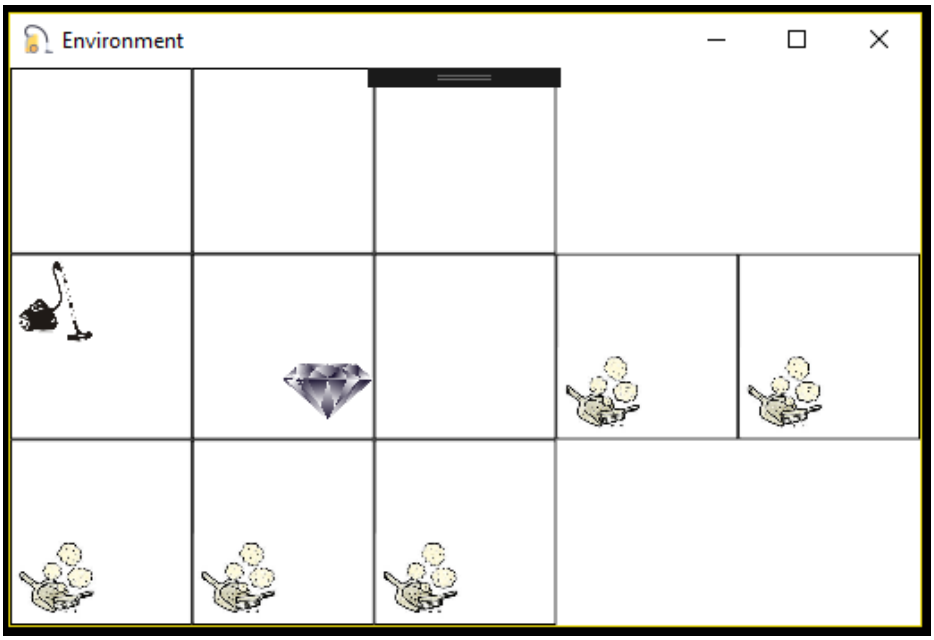
L'agent intelligent peut consulter et modifier à volonté la case de l'environnement dans laquelle il se trouve. Il n'a aucune influence sur la génération d'éléments ni la possibilité d'anticiper l'évolution de l'environnement dans lequel il évolue car il connaît uniquement le contenu de la case sur laquelle il se trouve. L'un des choix de conception que nous avons fait est l'impossibilité pour l'environnement de faire apparaître un bijou sur la case du robot. En effet, dans la réalité, un bijou placé directement sous l'aspirateur n'est pas vraiment un cas plausible.

2. Description de l'environnement

Dans cette partie, nous allons nous attacher à décrire l'environnement dans lequel le robot évolue.

2.1. Description générique de l'environnement

Ci-dessous le schéma de l'environnement :



Cet environnement est partiellement observable grâce aux capteurs de notre agent qui lui donnent accès à des informations concernant la case où il se trouve. L'environnement est également stochastique, car il va faire apparaître des éléments dans les pièces, sans consulter l'agent. Le prochain état de l'environnement n'est donc pas seulement déterminé par l'état courant et l'action exécutée par l'agent. L'enchaînement des actions est épisodique.

Il s'agit d'un environnement dynamique, car on souhaite éviter un déroulement au "tour par tour". En effet, l'ajout de poussière et de bijoux doit pouvoir se faire n'importe quand, quel que soit l'état mental courant du robot aspirateur, même pendant qu'il délibère sur l'action à prendre. Dans un souci de simplicité, l'environnement est discret, cela signifie que seul un nombre fini d'actions, de perceptions et d'états est possible dans notre simulation. Nous sommes également sur un modèle mono-agent.

2.2. Génération aléatoire des objets

Dans la conception de l'environnement, nous avons fixés les choix techniques suivant :

- L'environnement est constitué de cases. Ces cases peuvent présenter poussière, bijoux, les deux ou aucun des deux ;
- La présence de contenu est booléenne. La valeur est donc discrète ;

- Par soucis de réalisme, un coefficient est fixé pour que l'environnement génère plus de poussière que de bijoux.

La méthode de génération des éléments est appelée *GenerateObject(int c)*. *c* est le coefficient représentant $\frac{n_{poussière}}{n_{bijoux}}$. Une première variable locale stocke un chiffre aléatoire généré entre 1 et 13 – il y a onze cases indexés de 1 à 13 et deux cases interdites portant les index 4 et 5.

Après avoir testé si la valeur prise par le chiffre aléatoire est autorisé, nous testons le rapport $\frac{n_{poussière}}{n_{bijoux}}$. En fonction de ce rapport, nous ajoutons un objet dans la case choisie aléatoirement.

Ce mécanisme simple et efficace permet de générer un contenu sur une case aléatoire de la matrice environnement.

```
private void GenerateObjects(int p_iDustJewelRatio)
{
    Random rand = new Random();
    int index = rand.Next(0, 13);

    if ((index != 3) && (index != 4))
    {
        if ((m_iDustNumber / (m_iJewelNumber + 1)) <= p_iDustJewelRatio)
        {
            AddDust(index);
        }
        else
        {
            AddJewel(index);
        }
    }
    else
    {
        index = rand.Next(0, 13);
    }
}
```

3. Description du robot / BDI

3.1. Le type d'agent

Parmi les différents types d'agents existants, nous développons dans ce projet un agent basé sur les buts. L'un des points essentiels qui le définit est que, pour deux situations identiques impliquant le même état de l'environnement, notre programme ne décidera pas forcément d'effectuer la même action. La distinction se fera sur les buts de l'agent à l'instant de la décision, et c'est cette nuance qui le différencie du simple agent réflexe.

Ainsi notre robot va observer la même suite d'instructions en permanence, depuis son état initial jusqu'à ce que son but soit atteint (ou non atteint) :

- Il observe son environnement ;

- Il met à jour ses perceptions et son état interne ;
- Il choisit l'action la plus susceptible de le mener à son but ;
- Il exécute cette action.

Chacune de ses quatre instructions sont relativement vagues, et fait donc appel à des sous-fonctions pour donner "vie" au robot.

Points importants de code :

Nous avons utilisé la structure basique d'agent vue en cours, qui fait intervenir une boucle *while*, qui tourne tant que le robot est opérationnel. Pour notre application cela signifie que son compteur de points n'est pas tombé à zéro. Des points sont octroyés quand le robot aspire de la poussière (+1 point) ou ramasse un bijou (+6 points), et retirés quand un effecteur est utilisé (-1 point quand le robot se déplace, aspire, ramasse).

```
/// <summary>
/// Thread to run the robot through the manor.
/// </summary>
private static void ThreadRobotStartingPoint()
{
    while (raiJojo.AmIAlive())
    {
        Thread.Sleep(1000);
        /* The function call execute the BDI model.
         * - First we call GetEnvironmentState() which return the state of the environment.
         * - The we call DetermineActionUponMyGoal() which determines which action will bring
         * the robot to its goal.
         * Finally we call DoAction() which executes the action which has been chose,.
         */
        raiJojo.GetEnvironmentState();
        raiJojo.DoAction(raiJojo.DetermineActionUponMyGoal(raiJojo.UpdateMyState()));
    }

    MessageBox.Show("The robot doesn't have any points left", "Robot died...", MessageBoxButton.OK, MessageBoxImage.Error, MessageBoxResult.OK);
}
```

On peut ainsi suivre le cheminement de l'état du robot. S'il est en vie, il récupère les données de l'environnement, puis choisit l'action la plus pertinente pour atteindre son but, et finit par l'exécution de cette action. Une temporisation d'une seconde a été placée pour permettre à l'utilisateur de voir les changements de manière graphique.

3.2. Modélisation des percepts

Les percepts du robot sont modélisés dans le programme par des méthodes accédant aux variables appartenant à la classe *Square*, représentant une case de l'environnement.

Ainsi l'agent peut savoir si la case dans laquelle il se trouve contient des bijoux ou de la poussière.

Nous avons donc choisi de placer ces méthodes dans une classe *Sensor*. Chacune des méthodes de cette classe prend comme paramètre un indice de case et retourne le booléen contenu dans la case de l'environnement.

Une fois l'état de la case perçue, le robot enregistre l'état de la case sous forme d'un entier pouvant prendre 4 valeurs :

- 0 : Case vide ;
- 1 : Case avec poussière ;
- 2 : Case avec bijoux ;
- 3 : Case avec bijoux et poussière.

Une fois ces situations prises en compte, le robot permet de mettre en place un cheminement de pensée le menant vers une prise de décision.

```
public static class Sensor
{
    /// <summary>
    /// Return true if the current case contains some dust, false otherwise.
    /// </summary>
    /// <param name="p_sSquare"> The case to scan. </param>
    /// <returns></returns>
    1 référence | Nicolas G, Il y a 14 heures | 1 auteur, 3 modifications
    public static bool HasDust(Square p_sSquare)
    {
        return p_sSquare.HasDust;
    }

    /// <summary>
    /// Return true if the current case contains some jewels, false otherwise.
    /// </summary>
    /// <param name="p_sSquare"> The case to scan. </param>
    /// <returns></returns>
    1 référence | Nicolas G, Il y a 14 heures | 1 auteur, 3 modifications
    public static bool HasJewel(Square p_sSquare)
    {
        return p_sSquare.HasJewel;
    }
}
```

3.3. Modélisation des actions

Dans le cadre de notre simulation, le nombre d'actions que peut effectuer l'aspirateur est relativement limité. Ainsi on distingue trois actions possibles (en plus de l'action de ne rien faire), que l'on va formaliser sous le format "STRIP". Ce modèle consiste à associer à chaque action, un couple prémisses-conséquences. Les prémisses sont les conditions requises par l'environnement pour que l'agent soit en mesure d'effectuer son geste. Les conséquences décrivent les changements apportés à l'environnement après exécution de l'action. Ainsi on définit :

Action 1 - Aspirer

- ❖ Prémisses : robot en attente sur une case du plateau
- ➢ Conséquences : disparition des poussières et des bijoux de la case

Action 2 - Déplacer le robot

- ❖ Prémisses : robot en attente sur une case du plateau
- ➢ Conséquences : robot sur une autre case du plateau

Action 3 - Ramasser

- ❖ Prémisses : robot en attente sur une case du plateau
- ➢ Conséquences : disparition des bijoux sur la case

Points importants de code :

```
/// <summary>
/// A possible action is to move the robot.
/// </summary>
public class MoveRobot : ActionPossible
{
    private string m_sName = "MoveRobot";

    public override string Name()
    {
        return m_sName;
    }

    public MoveRobot(VacuumRobot.RobotAI p_RobotAI) : base(p_RobotAI)
    {
    }

    public override void Act()
    {
        m_RobotAI.Move();
    }
}
```

Afin de modéliser les actions, nous avons défini une classe abstraite *ActionPossible*, et nous utilisons l'héritage pour dériver des classes d'actions pertinentes pour l'aspirateur, telles que "MoveRobot" ou "Grab".

Toutes les classes d'actions possèdent en attribut le robot, qui va ainsi pouvoir les exécuter.

Les prémisses sont absentes car toutes les actions sont théoriquement possibles dans tous les états de l'environnement (le chemin est prédéfini).

Par exemple, si la décision est de se déplacer, la fonction appelée via *Act()* correspond à la méthode *move()* du robot, lui permettant de changer de case.

3.4. État mental BDI

En plus de ses capteurs et actionneurs, l'agent intelligent doit présenter un état mental, correspondant à la représentation qu'il se fait de l'environnement, des possibilités qui l'entourent et des objectifs qui lui sont fixés. Le modèle mental réalisé dans ce projet est un des plus courants, le modèle Belief-Desire-Intention (BDI) que nous avons adapté au problème de notre agent aspirateur.

Belief : Ce sont les croyances de notre robot par rapport à son environnement, résultats des données perçues via ses capteurs. Dans notre cas, il s'agit de l'état de propreté de la case sur laquelle se trouve l'aspirateur, une donnée fournie par les capteurs. La seconde croyance de l'agent est le numéro de la case qu'il occupe, celle-ci est déterministe et dépend uniquement des actions entreprises par le robot. De plus, nous avons fait le choix d'un chemin prédéfini que notre agent parcourt indéfiniment.

Desire : Les désirs sont les buts qui sont fixés à notre agent. Ce ou ces buts doivent guider le choix des actions entreprises par l'aspirateur, Pour notre problème, les buts sont très simples et ne requièrent pas de formuler une liste d'actions pour y parvenir. En effet, les désirs que l'on définit sont parmi les suivants:

- Rendre la pièce propre en faisant attention de ramasser les bijoux ;
- Rendre la pièce la plus propre le plus rapidement possible ;
- Economiser le plus d'électricité possible.

Ces désirs vont conditionner l'action prise par l'agent, par exemple si celui-ci se trouve sur une case avec de la poussière et des bijoux, il pourra :

- Aspirer les deux sans faire de distinction, si son but est de rendre la pièce propre rapidement ;
- Ramasser le bijou, et ensuite aspirer la poussière, si son but est de prendre soin des bijoux ;
- Ne rien faire si son but est d'économiser de l'électricité.

Ainsi on respecte la définition de l'agent basé sur les buts, qui peut entreprendre deux actions distinctes pour atteindre deux buts différents, sur des données identiques de l'environnement.

Intention : Les intentions de l'agent sont les actions intermédiaires qu'il compte entreprendre pour se rapprocher de son but final. Il réfléchit sur les conséquences futures de ses éventuelles actions, et décide de réellement exécuter celle qui le rapproche le plus de ses désirs. Dans le cas présent de l'aspirateur, aucun des buts fixés au robot ne prendra plus d'une action pour être rempli ou échoué, alors les intentions de l'agent se limitent au "temps de réflexion" qu'il mettra à choisir l'action la plus adéquate.

Points importants de code :

La partie Belief est principalement codée dans la méthode *GetEnvironmentState()*, qui permet à l'aspirateur d'utiliser ses capteurs pour aller récupérer l'état de l'environnement.

```
/// <summary>
/// Get the Environment state (Dust or not ? Jewel or not ?).
/// </summary>
public void GetEnvironmentState()
{
    m_bDustDetected = Sensor.HasDust(m_asPathArray[m_iCurrentPositionIndex]);
    m_bJewelDetected = Sensor.HasJewel(m_asPathArray[m_iCurrentPositionIndex]);

    SetControlText(textBoxDust, m_bDustDetected.ToString());
    SetControlText(textBoxJewel, m_bJewelDetected.ToString());
}
```

On a ensuite la fonction *UpdateMyState()* qui permet de mettre à jour l'état mental du robot pour lui permettre de prendre sa décision.

```
/// <summary>
/// Return the current state of the Environment from the robot's point of view.
/// The state of the Environment can return the following values :
/// 0 : Nothing on the Square.
/// 1 : Only dust on the Square.
/// 2 : Only jewels on the Square.
/// 3 : Both dust and jewel on the Square.
/// </summary>
/// </summary>
/// <returns> Return an int array as followed [Index of the current Square, State of the current Square]. </returns>
public int[] UpdateMyState()
{
    int iResultState = 0;

    if ((m_bDustDetected == true) && (m_bJewelDetected == false))
    {
        iResultState = 1;
    }
    if ((m_bDustDetected == false) && (m_bJewelDetected == true))
    {
        iResultState = 2;
    }
    if ((m_bDustDetected == true) && (m_bJewelDetected == true))
    {
        iResultState = 3;
    }

    int[] aiResult = { m_iCurrentPositionIndex, iResultState };
    return aiResult;
}
```

La partie Desire du modèle comprend plusieurs méthodes, ci-dessous se trouve la capture de la plus importante. La fonction *DetermineActionUponMyGoal()* prend en paramètre l'état de l'environnement et le but de l'agent.

```
/// <summary>
/// Choose an action for the agent to perform, based on its goal and environment.
/// </summary>
/// <param name="p_aiStateEnv"> Current state of the environment surrounding the agent. </param>
/// <param name="p_iMyGoal"> Current goal the agent is trying to achieve. </param>
/// <returns> The best choice action in the current context, that will help the agent to achieve its goal. </returns>
public ActionPossible DetermineActionUponMyGoal(int[] p_aiStateEnv, int p_iMyGoal)
{
    // This list contains each possible action for the agent, regardless of their relevance
    List<ActionPossible> lapListActionPossible = ActionDeclenchable(p_aiStateEnv);

    // Initialises the index used to keep track of the best action to perform.
    int iIndexActionToDo = -1;

    // Each iteration, initialises the worthiness of the action currently evaluated.
    int iWorthiness = -1;

    for (int i = 0; i < lapListActionPossible.Count; i++)
    {
        if (iWorthiness < CalculateWorthiness(lapListActionPossible[i], p_iMyGoal, p_aiStateEnv))
        {
            iWorthiness = CalculateWorthiness(lapListActionPossible[i], p_iMyGoal, p_aiStateEnv);
            // If the current action is the most relevant, we keep its index in the list.
            iIndexActionToDo = i;
        }
    }
    // When we went through the whole list, the index returned is the one of the most relevant action.
    return lapListActionPossible[iIndexActionToDo];
}
```

Tout d'abord on utilise la fonction *ActionDeclanchable()*, qui permet d'obtenir la liste des actions qu'il est possible de réaliser. On distingue les actions possibles et pertinentes, dans notre cas très simple, aucune

action n'est impossible. En revanche si l'agent se déplaçait aléatoirement, et qu'il se trouvait sur une case sur le bord gauche du plateau, l'action *MoveToLeft()* ne serait pas permise.

Une fois cette liste d'actions récupérée, on itère à travers celle-ci pour calculer la pertinence de chaque action, évaluée par la variable *iWorthiness* renvoyée par *CalculateWorthiness()*. Celle qui aura donné le meilleur résultat sera retenue, et renvoyée par *DetermineActionUponMyGoal()* pour permettre l'exécution de l'action.

Conclusion

A travers ce projet, nous avons pu concevoir, implémenter et tester le modèle théorique d'agent intelligent basé sur des buts. Prenant la forme d'un aspirateur, notre agent intelligent est capable de se déplacer en suivant un itinéraire prédéfini, tout en indiquant en console chacune des étapes franchies, et le cheminement de pensée.

Dans chacune des cases du plateau, l'intelligence artificielle est capable, via le modèle BDI, de percevoir l'état de son environnement, de déterminer l'action à prendre basé sur son objectif afin de maximiser ses chances de succès. Les trois actions qui lui sont actuellement possibles sont : aspirer la poussière, ramasser des bijoux ou ne rien faire. Avec ce modèle en place, l'ajout d'actions supplémentaires sera relativement simple, par exemple les déplacements aléatoires.

Au cours de la conception, nous avons dû faire face à différents défis techniques et humains. La première difficulté a été d'asseoir des choix techniques qui conviennent à l'intégralité de l'équipe, notamment concernant l'environnement, l'itinéraire du robot et ses agissements. Ensuite le principal défi a été de traduire le modèle BDI en concept algorithmique concret. Nous avons alors choisi de faire part à l'utilisateur de l'état mental du robot, via une fenêtre de console pour avoir un regard sur la validité de l'implémentation du modèle BDI.

Sur un aspect purement technique, c'était la première fois que nous travaillions en équipe sur un projet C# WPF synchronisé via un git. Cet aspect a été un élément de formation où nous avons pu acquérir de nouvelles compétences.

Ensuite, nous avons dû passer beaucoup de temps pour traiter les difficultés causées par le multithreading. Concernant l'environnement, nous avons effectué des recherches pour trouver un moyen de générer périodiquement des objets aléatoire dans le GUI, notamment la gestion des timers cadençant les actions du robot et la génération d'objets. Il nous a donc fallu chercher une solution alternative pour palier à cette limitation. Ce point nous permet de prendre conscience à quel point le multithreading demande une appréhension poussée de la logique d'exécution du programme afin de parvenir à atteindre notre objectif.

Pour finir, grâce à ce projet nous avons pu prendre connaissance d'un modèle d'agent intelligent élémentaire tout en faisant face aux défis techniques de l'implémentation d'une telle solution.