

8INF846 - Intelligence Artificielle



Université du Québec
à Chicoutimi

Travail Pratique n°3

Agent logique – Le simulateur d'un robot qui sauve des vies

Groupe 1 :

Mathias DURAND (DURM13099909)

Sophie GERMAIN (GERS09559909)

Pierre LACLAVERIE (LACP03119904)

Barnabé PORTIER (PORB26070006)

I. Contextualisation

A. Rappel du problème

On souhaite modéliser une intelligence artificielle, sous la forme d'un système expert, capable de diriger un robot dans une zone dévastée notamment par des incendies et des décombres, afin d'y trouver un survivant. Pour accomplir sa tâche, le robot dispose de capteurs capables de détecter de la chaleur, des incendies, des cris, de la poussière et des décombres. Dans notre modélisation, ses déplacements sont limités. Le robot se déplace sur une grille et ne peut aller que vers le haut, le bas, la droite ou la gauche, et pas en diagonale. L'agent peut échouer s'il s'enlise dans les décombres mais peut éteindre un feu. Le robot n'a aucune limite de temps ou d'action, il peut continuer jusqu'à être coincé par les décombres ou qu'il ait trouvé le survivant.

B. Exécution de notre programme

Voici les caractéristiques d'exécution de notre programme :

- Version de Python > 3.8
- Fichier Python à exécuter : Main.py sans arguments
- Interaction avec le programme : après avoir atteint la victime, le robot a fini le niveau. Pour passer au niveau suivant, appuyez sur la touche Entrée.
- Arrêt du programme : lorsque l'agent se retrouve coincé sous les décombres, ou Ctrl+C n'importe quand pendant l'exécution du programme.

II. Modélisation de la solution

Nous avons modélisé la solution à l'aide des classes Python suivantes.

- Environnement : définit l'environnement (carte des lieux, probabilités ...)
- Agent : définit l'agent intelligent, qui analyse son environnement proche avec son Capteur, se déplace et peut interagir avec l'environnement grâce à son Effecteur
- Capteur : permet à l'agent d'analyser l'environnement et ses cases
- Effecteur : permet à l'agent d'agir sur l'environnement : se déplacer, éteindre un feu, sauver la victime ...

L'Agent commence par charger son environnement. Il utilise ensuite son Capteur pour déterminer l'état des cases sur lesquelles il peut se déplacer. L'Agent décide ensuite de sa prochaine action, il appelle alors l'Effecteur pour la réaliser.

III. Code et explications

Il est à noter que les captures d'écran ont été légèrement modifiées par rapport au code afin d'en simplifier la lecture. Des fonctions ont été renommées pour plus de cohérence et des commentaires ainsi que d'autres détails sont présents dans le code source mais pas ici, ces captures n'ayant pour vocation qu'à illustrer les propos tenus.

A. Règles de déduction (if, for)

Le robot n'a qu'une connaissance limitée de son environnement, il n'est certain que de l'état des cases contre lui. Ainsi, il doit tout d'abord prendre en considération ce qu'il voit i.e. ce qui est présent autour de lui. La fonction `check_voisins` permet d'analyser les cases en question. (voir partie étapes du cycle d'inférence).

Cependant, le robot doit aussi prendre en compte ce qui n'est pas visible dans son entourage immédiat en émettant des hypothèses sur les voisins de ses voisins. Car en effet, cela apporte autant d'information de savoir ce qui est présent dans l'entourage proche que ce qui peut être présent dans les cases qui suivent l'entourage direct.

Cela est notamment géré grâce à la fonction `check_positive_subsequent_cases` qui permet d'émettre des déductions sur ce que peuvent contenir les cases entourant la case voisine à l'agent. Ci-dessous la fonction :

```
def check_positive_subsequent_cases(self, a, b, horizontal, element, possibilities):
    for m in [1, -1]:
        if(horizontal):
            b += m
        else:
            a += m
            # la case regardée n'est dans le tableau ou la probabilité de la case est déjà fixée
            # ou la case a déjà été visitée
            if(not self.envIRONNEMENT.isInBornes((a, b)) or possibilities[a][b]["determined"] or possibilities[a][b]
["visited"]):
                break
            if(element == Element.HEAT and Element.FIRE not in possibilities[a][b]["nonpossibilities"]):
                possibilities[a][b]["possibilities"].append(Element.FIRE)
            elif(element == Element.CRIES and Element.CRIES not in possibilities[a][b]["nonpossibilities"]):
                possibilities[a][b]["possibilities"].append(Element.VICTIM)
            elif(element == Element.DIRT and Element.DIRT not in possibilities[a][b]["nonpossibilities"]):
                possibilities[a][b]["possibilities"].append(Element.RUBBLE)
```

Ci-dessous la fonction `check_negative_subsequent_cases` qui est l'organe permettant d'émettre des déductions de ce qu'il ne peut pas y avoir sur les voisins de son entourage proche.

```

def check_negative_subsequent_cases(self, a, b, horizontal, element, possibilities):
    random.shuffle(self.listthehe)
    # on regarde les voisins
    for (k, l) in self.listthehe:
        # on regarde si la cible est bien dans le tableau
        if(not self.environnement.isInBornes((a + k, b + l))):
            break
        if(element != Element.HEAT):
            possibilities[a + k][b +
                            l]["nonpossibilities"].append(Element.FIRE)
        if(element != Element.CRIS):
            possibilities[a + k][b +
                            l]["nonpossibilities"].append(Element.VICTIM)
        if(element != Element.DIRT):
            possibilities[a + k][b +
                            l]["nonpossibilities"].append(Element.RUBBLE)

```

carbon
carbon.now.sh

B. Structure de données qui compose les faits

À chaque étude d'une case, un tableau 2D stocke pour chaque case les possibilités et les "impossibilités" (ce qu'il ne peut pas y avoir) d'éléments dans celles-ci selon nos déductions.

Cette variable est remplie lors des déductions effectuées sur les cases adjacentes aux cases voisines directes de l'agent. Elle est ensuite utilisée lors de la prise de décision concernant le choix de la case vers laquelle se diriger.

Ci-dessous, la déclaration de cette variable, disponible dans `check_voisins` de Capteur :

```

possibilities = [{"position": (o, p), "possibilities": [], "nonpossibilities": [],
                  "visited": False,
                  "determined": False}
                 for o in range(self.environnement.size)]
                 for p in range(self.environnement.size)]

```

carbon
carbon.now.sh

C. Algorithme de raisonnement déductif

C'est en prenant en compte ce que l'on sait, ce que l'on déduit et les règles, que nous pouvons avoir un système expert. Celui-ci va prendre en compte les différentes règles ainsi que ses déductions pour pouvoir choisir quelle sera la prochaine action à faire.

La déduction se fait dans la méthode `where_to_go` de Agent :

```

def where_to_go(self, possibilities):
    higher_proba: float = -1.0
    higher_x = self.current_position[0]
    higher_y = self.current_position[1]

    random.shuffle(self.listehe)
    #on regarde aléatoirement un voisin
    for (k, l) in self.listehe:
        # s'il est dans le tableau et qu'il n'est pas l'agent
        if (self.environnement.isInBornes(
            (self.current_position[0] + k, self.current_position[1] + l)) and not Element.AGENT in
            self.memoire[
                self.current_position[0] + k][self.current_position[1] + l]["elements"]):
            proba = self.get_proba((self.current_position[0] + k, self.current_position[1] + l),
                                   self.memoire[self.current_position[0] + k][self.current_position[1] + l][
                                       "elements"], possibilities)
            # on calcule sa nouvelle valeur de proba
            if (self.memoire[self.current_position[0] + k][self.current_position[1] + l]["visited"]):
                proba *= Probabilities.probabilities["visited"]
            if proba > higher_proba:
                higher_proba = proba
                higher_x = self.current_position[0] + k
                higher_y = self.current_position[1] + l

    # print(higher_x, higher_y, higher_proba)
    return (higher_x, higher_y), self.effetueur.do_action((higher_x, higher_y))

```

D. Méthodes de chaque étape du cycle du moteur d'inférence

Pour pouvoir bâtir un cycle de moteur d'inférence on va devoir réaliser trois grandes étapes majeures : le filtrage, le choix d'une règle et l'application de la règle.

Filtrage

Cette étape consiste en l'élimination des règles qui ne sont pas pertinentes.

Cela est fait dans la fonction `check_voisins`. En effet, cette méthode va commencer par vérifier si les indices étudiés correspondent à une case du tableau. Si ce n'est pas le cas, cette case sera écartée car elle n'est pas pertinente pour le programme. Sinon, il va stocker tout élément pertinent sur lequel il va pouvoir émettre des hypothèses (dans le cas d'une poussière, d'un cris ou de la chaleur, un de ces éléments dans la case étudiée signifie que chacune de ses cases voisine est *suspectée* de contenir respectivement des décombres, la victime ou du feu) ou alors avoir des certitudes (dans le cas d'un feu, de combles ou de la victime dans la case étudiée, on est *certain* que les cases voisines contiennent respectivement de la poussière, des cris ou de la chaleur). Cela va permettre d'alimenter les certitudes du robot et ses suppositions sur ce qui se trouve sur les voisins des cases visibles par le robot.

Ci-dessous le code expliqué et simplifié :

```

def check_voisins(self, agent_memoire, current_position):

    i, j = current_position
    possibilities = [{"position": (o, p), "possibilities": [], "nonpossibilities": [
], "visited": False, "determined": False} for o in range(self.environnement.size)] for p in
range(self.environnement.size)]
    agent_memoire[i][j]["visited"] = True
    possibilities[i][j]["visited"] = True

    random.shuffle(self.listehe)
    # on regarde aléatoirement un voisin
    for (k, l) in self.listehe:
        if(self.environnement.isInBornes((i + k, j + l))):
            if(self.environnement.map[i + k][j + l]["contents"] == [Element.EMPTY]):
                break

    for element in self.environnement.map[i + k][j + l]["contents"]:
        if(element == Element.AGENT):
            break

    agent_memoire[i + k][j + l]["elements"].append(element)
    # Si c'est la victime on finit le niveau
    if(element == Element.VICTIM):
        self.agent.end_level((i + k, j + l))
    elif(element == Element.FIRE or Element.RUBBLE):
        possibilities[i + k][j +
            l]["possibilities"] = [element]
        possibilities[i + k][j + l]["determined"] = True
    if(element != Element.EMPTY):
        self.check_positive_subsequent_cases(
            i + k, j + l, abs(k) == 1, element, possibilities) # supposition
        self.check_negative_subsequent_cases(
            i + k, j + l, abs(k) == 1, element, possibilities) # supposition
    return possibilities

```

Choix d'une règle

Pour pouvoir choisir une règle, on met des poids sur chacun des éléments qui composent notre système, on modélise ainsi les probabilités, soit la facilité avec laquelle l'agent pourra a priori poursuivre son chemin après l'élément. L'agent va calculer les poids de chaque possibilité de mouvement et va choisir celle qui possède la plus grande pour pouvoir par la suite s'y déplacer. Ci-dessous la fonction `get_proba` qui permet de mettre en place cette mécanique :

```

def get_proba(self, pos, elements, possibilities):
    proba: float = 1
    for element in elements:
        proba *= Probabilities.probabilities[element]

    # partie adjacents de l'adjacent
    prob = 0
    nb = 0
    # on prend les probabilités de chaque élément et on les pondère avec les poids
    # des éléments adjacents
    for (k, l) in self.lishehe:
        if (self.enviennement.isInBornes((pos[0] + k, pos[1] + l))):
            prrr = 1
            nb += 1
            for element in possibilities[pos[0] + k][pos[1] + l]["possibilities"]:
                prrr *= Probabilities.probabilities[element]
            if (possibilities[pos[0] + k][pos[1] + l]["visited"]):
                prrr *= Probabilities.probabilities["visited"]
            prob += prrr

    proba *= prob / nb
    return proba

```

carbon
carbon.now.sh

Appliquer une règle

Maintenant que le robot a filtré les possibilités et choisi une règle à appliquer, il faut donc... l'appliquer. C'est ce que `go_there` et `do_action` sont là pour mettre en place.

```

def go_there(self, pos, action=""):
    print("Je suis en " + str(self.agent.current_position))
    if(action != ""):
        print("Je fais " + action + " sur la position " + str(pos))
    # On met les informations à jour pour le robot et on affiche à l'écran
    # les informations utiles pour l'utilisateur
    self.agent.memoire[self.agent.current_position[0]][self.agent.current_position[1]]
["elements"].remove(Element.AGENT)
    self.agent.current_position = pos
    self.agent.memoire[pos[0]][pos[1]]["visited"] = True
    self.agent.memoire[pos[0]][pos[1]]["elements"].append(Element.AGENT)

    print("Je vais à " + str(pos) + "\n-----")
    if(action == "mourir"):
        print("Fin du jeu : je suis tombé sous les décombres. Adieu...")
        exit(1)

```

`go_there` va mettre en place le mouvement du robot sur la map et il va notifier l'utilisateur via la console pour y décrire l'action faite.

En effet, à chaque déplacement, l'agent peut faire des actions. Ces actions peuvent être de juste se déplacer, d'éteindre le feu, de sauver la personne ou de mourir sous les décombres. Pour gagner en lisibilité et en flexibilité, nous avons séparé le fait d'éteindre le feu et celui de mourir dans la méthode `do_action`. Dans le cas où le robot tombe dans les décombres, cela va arrêter le programme. `do_action` va être appelée avant `go_there` afin que le robot sache s'il doit faire une action supplémentaire après s'être déplacé et s'il ne trouve pas la victime. L'action résultante sera inscrite dans le paramètre `action` de `do_action`.

```
def do_action(self, pos):
    action = ""
    if (Element.FIRE in self.agent.memoire[pos[0]][pos[1]]["elements"]):
        action = "éteindre le feu"
        self.extinguish_fire(pos)
    elif (Element.RUBBLE in self.agent.memoire[pos[0]][pos[1]]["elements"]):
        action = "mourir"
    return action
```

E. Capteurs et effecteur


L'agent possède des capteurs pour obtenir des informations sur son environnement ainsi que des actionneurs pour exécuter les actions adéquates.

Ces deux parties sont modélisées grâce aux classes Capteur et Effecteur visibles ci-dessous :

```
class Capteur():
    def __init__(self, environnement, listhehe, agent):
        self.environnement = environnement
        self.listhehe = listhehe
        self.agent = agent
        random.seed(datetime.now())

    def check_voisins(self, agent_memoire, current_position):
        ''' Filtrage '''

    def check_positive_subsequent_cases(self, a, b, horizontal, element, possibilities):
        ''' emettre suppositions'''
    def check_negative_subsequent_cases(self, a, b, horizontal, element, possibilities):
        '''Emettre des suppositions'''
```



```
class Effecteur():
    def __init__(self, environnement, listehe, agent):
        self.environnement = environnement
        self.listehe = listehe
        self.agent = agent

    def go_there(self, pos, action=""):
        ''' se déplacer '''

    def do_action(self, pos):
        '''Faire une action '''

    def extinguish_fire(self, fire_pos):
        ''' eteindre le feu'''
```

carbon
carbon.now.sh

F. Environnement

Assez similaire dans sa conception à la classe homonyme du premier TP, la classe Board définit la carte dans laquelle l'agent va évoluer :

```
class Board:
    def __init__(self, size):
        """
        Constructeur de la classe Board
        """

    def random_valid_board(self):
        """
        Genere une grille dans laquelle va représenter l'environnement

        :return:
        """

    def fill_with_heat(self, x, y):
        """
        Un feu se trouve en (x,y) on remplit les cotes adjascent par de la chaleur
        :param x: ligne
        :param y: colonne
        :return: None (mets a jour self.map)
        """

    def fill_with_dust(self, x, y):
        """
        Un debris se trouve en x,y on remplit les cotes adjascent par de la poussiere
        :param x: ligne
        :param y: colonne
        :return: None (mets a jour self.map)
        """

    def fill_with_cries(self, x, y):
        """
        Une victime se trouve en x,y , remplit les cotes adjascents de ses cris
        :param x: ligne
        :param y: colonne
        :return: None (mets a jour la map)
        """

    def isInBornes(self, pos):
        """
        Verifie si la position est dans les bornes de la grille
        :param pos: la position a verifier
        :return: True si dans les bornes, False sinon
        """

        return pos[0] ≥ 0 and pos[0] < self.size and pos[1] ≥ 0 and pos[1] < self.size
```

carbon
carbon.now.sh

G. Programme principal

Le programme principal, lancé sans arguments, fait débiter l'agent au niveau de taille 3. À la fin de chaque niveau (victime trouvée), le programme demande à l'utilisateur d'appuyer sur Entrée afin de passer au niveau suivant.

```
def main():
    A = Agent()
    i = 0
    while(True):
        run_level(A, 3 + i)
        i+=1

def run_level(A, size):
    print("Niveau de taille " + str(size))
    B = Board(size)
    print(B)
    try:
        A.run(B)
    except Exception as e:
        print(e)
        exit(1)
    inp = input("Passage au niveau suivant ?")
```

carbon
carbon.now.sh

Note : le programme ne se ferme tout seul que lorsque l'agent rentre dans des décombres. L'utilisateur doit appuyer sur Entrée pour passer au niveau suivant lorsque le niveau en cours se termine, et doit utiliser Ctrl+C pour mettre fin à l'exécution lorsqu'il souhaite forcer l'arrêt du programme.