TP4: Les apothicaires

Cédric Buron

14 décembre 2020

Introduction

Dans de TP, nous allons nous intéresser à la problématique suivante : quelques apothicaires ont décidé de coopérer afin de produire leur fameuse liqueur d'Anablardone. Nous allons pour cela utiliser le paradigme BDI. L'anablardone est une liqueur qui se compose principalement de deux plantes : l'anis et la blardone. Ces deux herbes sont disponibles en forêt. Nos herboristes ont pour objectif de récolter anis et blardone en quantité égale afin de distiller leur brevage. Pour cela, ils doivent choisir à chaque moment un endroit de la forêt où faire leur récolte parmi les 10.

Il n'existe pas de bibliothèque python qui permette de faire du BDI à elle toute seule. Nous allons donc utiliser une bibliothèque qui permet de traiter des fichiers écrits en AgentSpeak, le même langage qui est utilisé avec Jason. Ce TP va donc se concentrer sur la partie AgentSpeak ; nous allons nous concentrer sur le fichier `.asl`. Contrairement aux autres TP, celui-ci sera davantage guidé, car il se focalisera sur un autre langage. Que vous fassiez ou non ce TP sur jupyter, il vous faudra éditer le fichier `agent.asl`. Avant de démarrer, il vous faudra installer le package agentspeak :

```
pip install agentspeak
```

Si vous avez besoin de fonctions qui ne sont pas expliquées dans le texte du TP, vous pouvez vous référer à cette notice. Attention cependant, certaines fonctions ne sont pas encore implémentées dans le package python. Vous trouverez la liste des fonctions dans le stdlib du package.

Nous allons commencer par le code python. D'abord les imports :

```
import math
import random
from typing import List

import agentspeak
from agentspeak import Literal
import agentspeak.runtime
import agentspeak.stdlib

import os
```

Nous allons maintenant devoir définir les actions qui peuvent être utilisées par nos agents. Avant d'ajouter les actions spécifiques à notre TP, nous allons inclure les fonctions standard :

```
actions = agentspeak.Actions(agentspeak.stdlib.actions)
```

Et on ajoute le nombre de ressources qui seront utilisées dans le cadre de notre TP : 2 (l'anis et la blardone).

```
N_RESOURCES = 2
```

Nous déclarons ensuite une fonction dite *Upper Confidence Bound* qui nous permettra de décider quelle région de la forêt chaque agent ira explorer. Nous ajouterons cette fonction aux actions disponibles pour les agents.

```
def ucb(mu, ni, n):
    if n == 0:
        return 200
    elif ni == 0:
        return 200
    else:
        return mu/n + math.log(2*n/ni)
```

Il nous faut aussi définir les ressources elle-même. Chaque région a une certaine quantité de chaque ressource en stock, et une certaine moyenne, qui sert à générer une quantité de chaque ressource, via la fonction `exploit`.

```
class Resource:
    def __init__(self, mu_anis, mu_bardane, quantity):
        self.mu_anis = mu_anis
        self.mu_bardane = mu_bardane
        self.quantity_anis = quantity
        self.quantity_bardane = quantity

    def exploit(self):
        anis = min(self.quantity_anis, random.triangular(0, 1, self.mu_anis))
        bardane = min(self.quantity_bardane, random.triangular(0, 1, self.mu_bardane))
        self.quantity_anis = self.quantity_anis - anis
        self.quantity_bardane = self.quantity_bardane - bardane
        return anis, bardane

    def __str__(self):
        return str(self.__dict__)
```

Enfin, nous définissons l'environnement, en fonction des éléments précédemment cités. Notons que c'est aussi ici que nous devrons définir les actions supplémentaires pour notre agent. Nous instancions également l'environnement et les agents :

```
class ResourceEnvironment(agentspeak.runtime.Environment):
    resources: List[Resource]

    def __init__(self):
        super().__init__()
        self.resources = []
        for i in range(N_RESOURCES):
            self.resources.append(Resource(random.random(), random.random(),
            15 + 5 * random.random()))

env = ResourceEnvironment()

with open(os.path.join(os.path.abspath(''), "agent.asl")) as source:
    agents = env.build_agents(source, 5, actions)
```

Enfin, nous lançons le programme :

```
if __name__ == "__main__":
    random.seed(0)
    env.run()
```

Implémentation des fonctions

Nous allons avoir besoin de deux fonctions non standard, qui devront être ajoutées à la base des actions accessibles par les agents dans le fichier `main.py`. La première est une fonction à proprement parler, l'autre une action. La fonction n'est que la fonction `ucb` déjà définie. Pour ajouter une fonction ou une action à l'ensemble des actions à disposition de l'agent, on utilise une annotation : `@actions.add_function('.name', input_type)`. Ici, nous déclarons la fonction de nom `'.ucb'` (notez que le nom donné à la fonction python et le nom utilisé en AgentSpeak n'ont pas à être identiques, mais on les gardera identiques ici pour faciliter la lecture du code). Le type d'entrée doit être donnée sous la forme d'un tuple des types d'entrées.

Le seconde action est une action à part entière. **Elle doit être implémentée dans l'environnement**. Il y a là quelque chose d'assez particulier. La fonction est tout d'abord précédée d'une annotation, un peu différente : `@actions.add(".name", nb_arguments)`. `nb_arguments` correspond au nombre d'arguments de la fonction.

Les fonctions représentant les actions des agents ont toujours la même signature :

```
@actions.add(".name", nb_arguments)
def name(self, term, intention):
```

Implémentez la fonction permettant à un agent de choisir une région de la forêt et d'en obtenir les ressources. Pour récupérer les arguments de la fonctions, vous devez accéder à `term.args`, qui stocke les arguments en tant que liste. La principale particularité de la fonction cependant est que, bien que dans la classe héritant de l'`Environment`, son `self` fait référence à l'agent. Ainsi, pour accéder aux ressources par exemple, il vous faudra appeler `self.env.resources`.

La fonction doit faire les choses suivanets : récupérer le numéro de région appelé par l'agent, puis appeler `exploit` sur le numéro de la ressource. Ensuite, récupérez la valeur de ressources actuelle de l'agent pour chacune des deux ressources. Pour cela, vous pouvez utiliser le champ `beliefs` de l'agent. Le champ `beliefs` est un dictionnaire qui contient toutes les croyances de l'agent. La clé est `('belief_name', n_args)` et contient le set de croyances de l'agents. Ainsi, pour accéder à la croyance déjà implémentée, `i_believe(0)`, on utilisera `list(self.beliefs[('i_believe, 1)])[0]`. Un bielief est un `Literal`, dont il est possible de récupérer les argmuents *via* le champ `args`. Ce champ est une liste des arguments de la croyance. Ainsi, `list(self.beliefs[('i_believe, 1)])[0]` renverra 0. Il est nécessaire de transformer le set en list pour pouvoir accéder aux éléments. Une croyance est définie comme un `Literal`. Le constructeur du `Literal` est le suivant : `Literal('belief_name', args)`. La fonction doit changer les beliefs `anis` et `blardone` de l'agent en ajoutant les sorties de la fonction `exploit`. **Enfin, la fonction doit se terminer par** `yield`.

Le langage AgentSpeak

Passons maintenant au gros du TP, l'implémentation du code AgentSpeak. Nos agents sont des agents BDI. Ils sont dotés de croyances, de buts (les "désirs" dans le paradigme donné en cours), de plans et d'intentions (le but auquel le plan en cours répond).

Avant toute chose, il faut savoir qu'en AgentSpeak, **les variables doivent être écrites en majuscule, et les litéraux (plans, croyances, buts) en minuscule ; de même il est impossible de modifier la valeur d'une variable qui a déjà été affectée**. AgentSpeak est un langage déclaratif. On y déclare les désirs, les plans, les croyance de l'agent.

Les buts

Il existe deux types de buts. Les buts de réalisation ou *achievement goals* sont les buts véritablement poursuivis par l'agent, alors que les objectifs de test ou *test goals* concernent la collect d'information.

Les buts de réalisation se déclarent simplement au moyen d'un point d'exclamation. Vous avez pour exemple le but `!start.` déclaré au début du fichier `agent.asl`. Chaque élément déclaré doit s'achever par un point.

Les buts de test permettent de récupérer des valeurs inconnues. Ils se déclarent ainsi :

```
?goal_name(v1, v2,..., vn).
```

Ces buts sont accompagnés de valeurs `v1,...,vn`. Si une varaible non affectée est passée en argument, elle pourra être affectée dans un plan répondant au plan en question. C'est ainsi que vous pourrez utiliser un plan vous permettant de calculer certains résultats intermédiaires. Notez qu'il est possible de requêter les beliefs avec ces buts. Ainsi `?i_believe(N);` assignera 0 à N.

Les croyances

Une croyance initiale se crée simplement *via* un litéral. D'autre part, il est possible de donner des arguments à une croyance, comme cela est fait au début du fichier `agent.asl`. Au cours d'un plan, ajouter une croyance se fait avec `+bel`, en retirer une se fait avec `-bel`. Retirer une croyance d'une certaine forme se fera au moyen de _. Ainsi :

```
+foo(5);
-foo(_);
?foo(N);
```

renverra une erreur.

Les plans

Un plan est une suite d'instructions résolvant un but lorsque certaines croyances sont vraies. Un plan se déclare ainsi :

```
+!goal: beliefs <-
    .instructions;
```

Notez que s'il s'agit d'un but de test, il faudra écrire `+?goal(...)`.

Lorsqu'un but est déclaré, on recherche immédiatement quel plan peut y répondre et on suspend l'intention en cours. Si aucun plan n'est susceptible de résoudre le but, une erreur apparaîtra.

Les actions sont généralement appelées *via* un nom précédé d'un point, par exemple `.print(X);` permet d'afficher X. Notez que la fonction print peut afficher plusieurs éléments *e.g.* `.print(X, 2, 'A');`. Notez aussi que certaines fonctions sont utilisées pour faire un calcul. Dans ce cas, on procède comme pour les buts de test, en passant des arguments non instanciés.

Il existe de nombreuses fonctions et actions préexistantes. Parmi celles que vous devrez utiliser pendant le TP :

- `.broadcast(achieve, literal);` crée le désir et la croyance `literal` chez tous les autres agents,
- `.min([X1,...,Xn], Y)` stocke le minimum de $X_1, \ldots, X_n$ dans Y,
- `.max([X1,...,Xn], Y)` stocke le maximum de $X_1, \ldots, X_n$ dans Y,
- `a mod b` renvoie le reste de la division euclidienne de a par b.

Notez que l'affectation se fait en utilisant l'opérateur =. Il est aussi possible d'utiliser des structures conditionnelles :

```
if(cond){
    .action1;
}
else{
    .action2;
}
```

et les boucles ; il est notamment possible de boucler sur tous les beliefs d'une certaine forme. Ainsi :

```
for(i_believe(K)){
    .print(K);
}.
```

imprimera 0.

À implémenter

3.4.1 Le plan principal

Le but du TP est le suivant : les agents commencent avec une quantité nulle d'anis et de bardane. À chaque tour, ils choisissent la région qui maximise l'UCB à partir de la valeur minimale entre l'anis et la blardone, et l'exploitent. Ils changent la valeur de leurs croyance sur les ressources qu'ils possèdent, ainsi que sur les valeurs de chacune des régions. On implémentera aussi un mécanisme de partage d'information. Quand plus aucune ressource n'est disponible, les agents arrêtent la collecte.

Avant de nous intéresser au code agentspeak, il faut créer des croyances pour chacun des agents sur les ressources en cours. Pour chaque agent, pour chaque resource i, on ajoute aux croyances de l'agent la valeur suivante :

```
Literal('values_r', [i, 0, 0, 0])
```

Cette croyance nous permet d'indiquer, pour chaque région i, la quantité moyenne d'anis, de bardane et le nombre de fois où la région a été visitée par l'apothicaire. Je vous conceille de faire le code en python afin de pouvoir l'adapter plus facilement au nombre de ressources.

Ensuite, dans le code AgentSpeak, on crée une croyance `resources_available` un désir `get_resources`, et un plan qui y est associé. Ce plan commence par rechercher la meilleure ressource et fait pour cela appel à un autre désir, test celui-ci. Puis il appelle la fonction qui permet de récolter les ressources. Enfin, il met à jour ses croyances. On commence par vérifier si on a obtenu de l'anis et de la bardane. Si ce n'est pas le cas pour une des ressources, on met la valeur moyenne de cette ressource pour cette ressource à 0. Sinon, on fait la moyenne entre le contenu de la croyance précédente et ce qu'on a obtenu en allant faire la collecte.

Note : Il est possible d'obtenir une information sur ce qui a été obtenu en comparant l'ancienne quantité de la ressource et la quantité mise à jour après la collecte.

Une fois les croyances sur les ressources mises à jour, on vérifie si le nombre de fois où la région a été visitée est un multiple de 10. Si c'est le cas, on envoie à tous les autres agents une croyance sur la région en question contenant sa valeur moyenne de a, de b et le nombre de visites. Attention ! Évitez d'utiliser une croyance ayant le même nom que les croyances déjà présentes chez l'agent. Cela pourrait se confondre avec lesdites croyances. Je vous conseille d'envoyer une croyance avec un nouveau littéral, mais avec la même "signature".

La réception se fait en utilisant le désir créé chez l'agent recevant le message, à la manière d'un désir de test, mais avec la syntaxe et le point d'exclamation d'un but de réalisation. Ainsi, on aura par exemple :

```
...
    .broadcast(achieve, foo(5));
...

!foo(X) <-
    .print(X).
```

La dernière étape du plan est de vérifier s'il reste des ressources pour lesquelles on a une croyance que cette ressource n'a pas une moyenne nulle en anis et en bardane. S'il en reste au

moins une, on s'assure que la croyance `resources_available` est toujours présente et on renouvelle le désir `get_resources`.

Ajoutez un plan qui affiche les quantités d'anis et de blardone quand la croyance `resources_available` n'est plus présente (on note alors `!+goal:not bel`).

3.4.2 Le plan test

Nous avons besoin de déterminer quelle est la meilleure région. Pour ce faire, on va créer une croyance. On va commencer par s'assurer qu'on a bien supprimé le désir des itérations précédentes. On va ensuite itérer sur toutes les croyances concernant les régions, calculer la valeur actuelle de chaque ressource de l'agent plus l'estimation de gain pour la région.

Note : Pour calculer la région de plus haute valeur, je vous recommande de faire une boucle for et de claculer UCB à chaque fois, en mettant à jour les informations concernant la meilleure région rencontrée jusqu'à présent.

Question 1– Quelle différence voyez-vous avec l'architecture PRS vue en cours ? Argumentez.

Votre réponse ici

Question 2– Testez votre code avec et sans le partage d'information. Qu'apporte ce dernier ?

Votre réponse ici

Question 3– (Bonus) Dupliquez le fichier `agent.asl` et modifiez-le pour que cet agent mente lorsqu'il envoit des informations. Faites en sorte qu'il envoie une croyance sur la région maximisant la somme d'anis et bardane et envoie la croyance que la moyenne de cette région est de 0. Quelle est la conséquence sur la quantité d'anis et de bardane recueilli par cet agent ? Par les autres ?

Votre réponse ici