DESQUENNE Vincent
GOUBIN Fabien

TP SMA
Exemple de planification multi-agents : le monde des blocs

Lien Git : https://github.com/VincentDesquenne/MultiAgentsBloc

1ère partie

Dans cette première partie, nous avions pour objectif de programmer un système multi-agents non communicatif. Chaque agent a comme objectif son emplacement dans la solution finale et peut exécuter deux actions : *pousser* si il y a un bloc au-dessus de lui, et se *déplacer* dans le cas contraire. Comme ce système est non communicatif, il n'y a donc pour l'instant pas de coordination des actions des agents et il y a une possibilité de redondance des actions individuelles. Les agents ne savent pas où ils sont, ni où sont les autres agents, ils possèdent simplement une fonction *perception* leur permettant de demander des informations à l'environnement, mais sans pouvoir stocker ces informations.
Dans nos explications, un agent correspond à un bloc.

Nous avons donc programmé de telle sorte que les blocs se déplacent jusqu'à obtenir la solution finale. Voici les différentes étapes de chaque itération :

1. Sélection aléatoire d'un bloc, même si ce dernier est bien placé.
2. L'agent va alors demander plusieurs informations à l'environnement, via sa fonction *perception*, afin de savoir quelle décision prendre.
   Il va tout d'abord lui demander s'il est bien placé, afin de savoir s'il a besoin de bouger, ce qui va mettre à jour sa condition de satisfaction.
   Ensuite l'agent va en premier regarder s'il est poussé, car même si le bloc en dessous de lui est le bon, s'il est poussé il doit bouger quoi qu'il arrive.
   Si le bloc est poussé, il va regarder s'il peut bouger, et dans ce cas, il va changer aléatoirement de position. Si le bloc ne peut pas bouger, il va alors pousser celui au-dessus de lui.
   Maintenant passons au cas où l'agent n'est pas poussé, mais n'est pas satisfait. Dans ce cas, s'il peut bouger, alors il se déplace aléatoirement. S'il ne peut pas bouger, il va simplement pousser le bloc au-dessus de lui.
3. A la fin de chaque itération, on regarde si la solution finale a été atteinte grâce à une méthode de l'environnement qui vérifie l'état actuel de l'environnement et la solution finale.
4. Si la solution finale est atteinte, le système se termine.

Comme expliqué auparavant, ce système n'est pas performant puisqu'il n'y a aucune coordination des actions des agents, et il peut y avoir un bouclage par la redondance des actions individuelles. Cela prend donc plus d'itérations et plus de temps pour atteindre l'objectif.

Pour lancer notre solution pour cette première partie, il suffit simplement d'exécuter le code, et de rentrer le chiffre "1" au clavier. Ensuite vous devez entrer le nombre d'essais que vous voulez faire (au minimum 1).

En exécutant notre programme, la première pile est faite aléatoirement (et ne correspond pas à la solution finale bien sûr), et le résultat de la moyenne du nombre d'itérations pour obtenir la solution finale après votre nombre d'essais choisis sera affiché dans la console. La console affiche également l'état de l'environnement à chaque itération. Il est donc possible de voir le détail de chaque itération et l'action effectuée par le bloc sélectionné lors d'une itération comme vous pouvez le voir sur la figure 1 (ne pas prendre en compte les "Priorite = 0", car il s'agit d'une fonctionnalité de la question 2). A noter que nous avons défini la solution finale par "A,B,C,D" comme affiché dans l'énoncé.

```
------------------------------------------------------
NB Iteration : 55
POSITION : 0
POSITION : 1
|| Bloc{, nom='D', Priorite ='0'} ||
|| Bloc{, nom='C', Priorite ='0'} ||
|| Bloc{, nom='B', Priorite ='0'} ||
POSITION : 2
|| Bloc{, nom='A', Priorite ='0'} ||
END
------------------------------------------------------
```

Figure 1 : affichage de l'état de l'environnement à chaque itération

Après avoir effectué de nombreux essais en exécutant notre programme, on trouve une moyenne du nombre d'itérations entre 60 et 65 pour obtenir la solution finale avec ce système. C'est un nombre assez élevé, mais qui est logique au vu de notre système non performant et assez aléatoire.

2ème partie

Dans cette seconde partie, nous avions pour objectif de programmer un système multi-agents communicatif à base de stratégies de coordination. Chaque agent a comme objectif son emplacement final et peut exécuter deux actions comme à la question 1 : pousser si il y a un bloc au-dessus de lui, et se déplacer dans le cas contraire. Il a encore une fonction perception comme expliqué précédemment, afin de communiquer avec l'environnement. La différence va être que cette fois-ci, les agents vont communiquer entre eux.

Étant donné que chaque agent connaît uniquement son objectif final, nous avons fait communiquer nos agents entre eux juste après l'initialisation de l'environnement. Ces derniers vont alors s'attribuer une priorité selon leur placement via une priorité. Par exemple, l'agent qui a pour objectif de ne pas avoir de bloc au-dessous de lui, va prendre une priorité de 4. Ensuite la priorité est diminuée de 1, et l'environnement va stocker en mémoire cet agent, puis tous les autres agents vont alors enquêter l'environnement, afin de savoir si leur objectif (donc leur bloc en dessous d'eux) correspond à ce bloc stocké. Une fois que l'agent stocké correspond avec l'objectif d'un agent, ce dernier met à jour sa priorité, et l'opération se répète jusqu'à que les 4 agents aient une priorité (allant donc de 4 à 1).
Cela correspond à notre fonction *communicationAgent*.

Une fois cela fait, on ne va pas choisir aléatoirement un agent, mais on va prendre celui qui a la priorité la plus haute. Une fois qu'un agent est bien placé, sa priorité passe à 0, permettant donc de sélectionner un autre agent. L'agent avec la plus haute priorité va donc être sélectionné jusqu'à qu'il soit bien placé.

Nous ne nous sommes pas limités simplement à cette stratégie, nos agents sont aussi plus performant, et communiquent mieux avec l'environnement. Par exemple, si un agent est poussé, ce dernier va alors demander à l'environnement si le premier bloc a été placé, et auquel cas il va essayer de lui laisser une position libre, afin qu'il soit situé en premier (tout en bas).
Cela correspond à notre fonction *calculerBestMove*, où les agents vont discuter avec l'environnement afin de se placer du mieux qu'ils peuvent.

Avec cette communication, nous arrivons à une moyenne de 5 itérations afin d'obtenir la solution finale, ce qui est très performant.