Introduction aux Systèmes Multi-Agents
Le Dîner des Philosophes

FRANÇOIS HERNANDEZ - LÉO PONS
CentraleSupélec
January 28, 2017

Contents

Introduction

Un Système Multi-Agents est un système composé d'un ensemble d'agents, situés dans un certain environnement et interagissant selon certaines relations. Un agent est une entité caractérisée par le fait qu'elle est, au moins partiellement, autonome. Le fonctionnement du système est défini par le fonctionnement des agents, le système central ne fait qu'office d'environnement d'évolution et de communication des différents agents. On parle d'intelligence artificielle distribuée.

Ce TD a pour objectif de construire une plateforme générique permettant de représenter et simuler des systèmes multi-agents de différentes natures. Un premier SMA simulé sera celui du dîner des philosophes.

Le problème des philosophes peut être défini de différentes manières, mais l'idée générale est toujours la même.

La situation est la suivante :

- Un certain nombre de philosophes se trouvent autour d'une table.
- Chacun des philosophes a devant lui un plat de spaghetti.
- À gauche de chaque plat de spaghetti se trouve une fourchette.

Un philosophe a un certain nombre d'états possibles, par exemple :

- Il peut penser, ce qui lui donne faim.
- Il peut être dans un état de famine, s'il a faim et qu'il ne peut pas manger.
- Il peut être en train de manger.

Des contraintes extérieures s'imposent à cette situation :

- Pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à droite (c'est-à-dire les deux fourchettes qui entourent sa propre assiette).
- Quand un philosophe a faim, il attend que les fourchettes soient libres.

Le problème consiste à définir le comportement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

I Plateforme SMA Synchrone

Dans un premier temps, nous créons un package 'plateforme' contenant les éléments principaux de la représentation générique d'un SMA.

Notre plateforme fonctionne de manière synchrone : Le premier agent observe son environnement et effectue une action dont les critères d'exécution sont satisfaits par les données observées, puis c'est au tour du second et ainsi de suite. Arrivé au dernier agent, on recommence en repartant du premier.

Notre package contient les classes suivantes :

- `Agent` : Représente chaque agent du système, liés à un `Environnement`, à une liste d'actions possibles et à une boîte aux lettres (pour la réception des messages, détaillée plus loin).
- `Environnement` : Représente l'environnement dans lequel évoluent les agents, contenant des `Donnee` et une liste d'agents `Agent`.
- `Donnee` : Classe générique représentant les différentes données d'un problème.
- `Action` : Classe abstraite destinée à représenter les différentes actions possibles, chaque action sera une sous-classe de celle-ci. Deux méthodes cruciales définissent une action : `check_conditions` qui évalue les critères d'exécution de l'action, et `effectuer` qui exécute l'action proprement dit et modifie donc les données affectées par cette action.
- `Message` : Les messages seront décrits dans un deuxième temps. Ils permettent la communication entre les agents.

Vous trouverez dans le code Java des commentaires détaillant l'implémentation des différentes classes et de leurs méthodes.

II Application au problème du dîner des philosophes

II.1 Implémentation Générale

Afin d'implémenter le problème des philosophes, nous créons un nouveau package 'philosophes' contenant différentes classes héritant des classes génériques de la plateforme, ainsi que de nouvelles destinées à représenter d'autres aspects spécifiques au problème.

Les classes héritant de la plateforme sont les suivantes :

- `Philosophe`, héritant de la classe `Agent`.
- `Table`, héritant de la classe `Environnement`.
- `philosophes.actions`, package de classes héritant de la classe `Action`. Chacune de ces classes définit une action exécutable par les philosophes.
- `Fourchettes`, héritant de la classe `Donnee` et contenant la représentation de l'état des N fourchettes de l'environnement.

Les classes supplémentaires sont les suivantes :

- `Etat`, type énuméré représentant l'état d'un `Philosophe`.
- `main`, permettant de faire tourner le système.

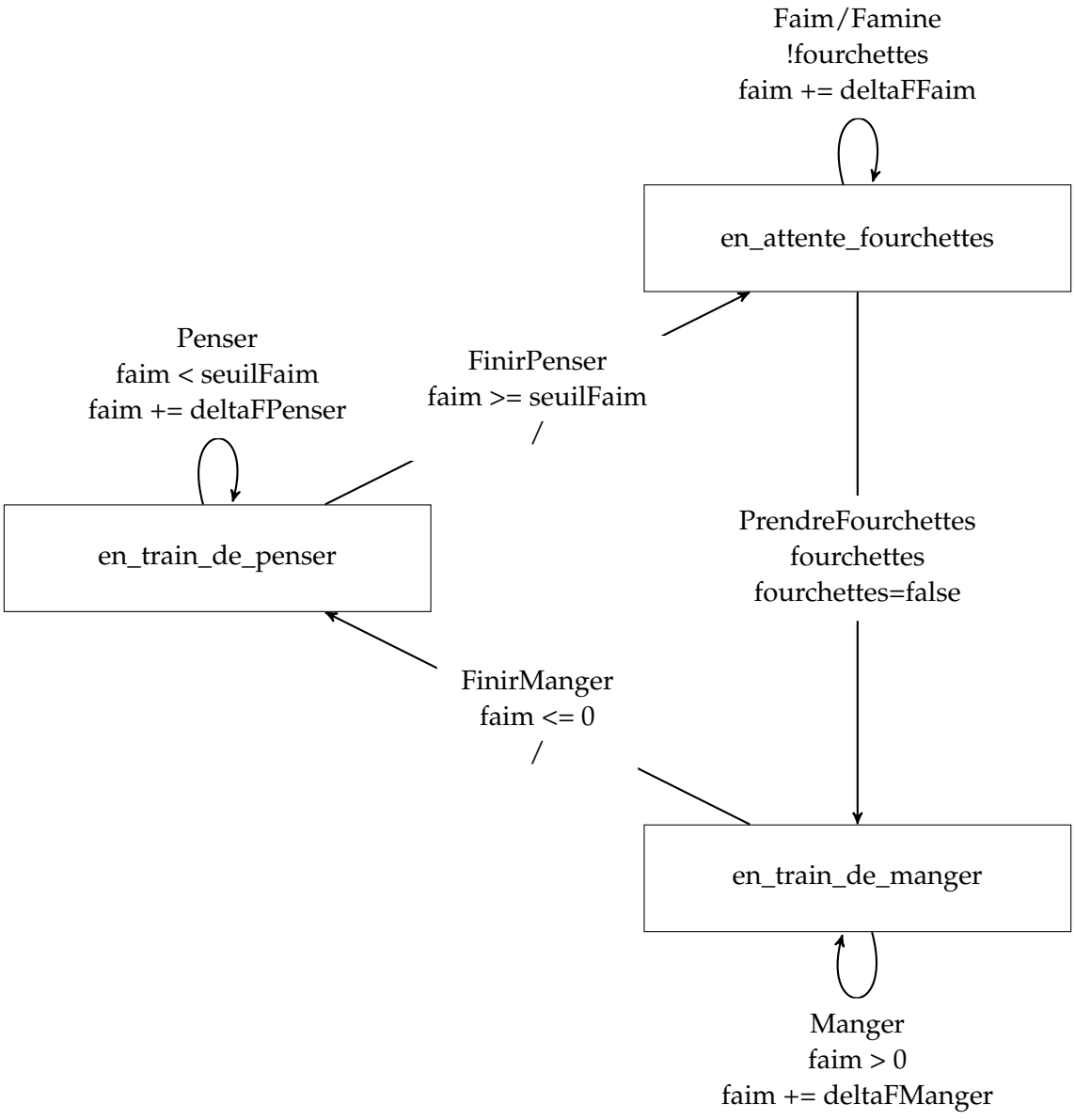
Comme précisé en introduction, le problème du dîner des philosophes peut être construit de différentes manières. Voici quelques points de comportement que nous avons décidé d'adopter :

- Le nombre de philosophes est défini par l'attribut `effectif` de la `Table`.
- Les philosophes commencent tous avec un compteur de faim `effectif` à 0, et leur état est `en_train_de_penser`.
- Quand un philosophe pense un compteur de mesure `compteurPensee` est incrémenté à chaque tour au niveau de la `Table`.
- Le fait de penser donne faim aux philosophes, ce qui est quantifié par l'attribut `deltaFPenser` de la `Table`.
- Passé un certain seuil de faim défini par l'attribut `seuilFaim` de la `Table`, le philosophe ne peut plus penser et a faim, il essaie alors de ramasser ses fourchettes pour manger.
- Lorsqu'un philosophe a faim et qu'il ne mange pas, sa faim s'aggrave, ce qui est quantifié par l'attribut `deltaFFaim` de la `Table`.
- Passé un certain seuil de faim défini par l'attribut `seuilFamine` de la `Table`, le philosophe est en état critique de famine : un compteur de mesure `compteurFamine` est incrémenté à chaque tour au niveau de la `Table`.

- Lorsqu'un philosophe mange, sa faim diminue, ce qui est quantifié par l'attribut `deltaFManger` de la `Table`.
- Passé le seuil de satiété défini par défaut à 0, le philosophe s'arrête de manger et peut recommencer à penser.

II.2 Machine à états

Nous avons défini quelques actions de base pour nos philosophes. À chaque tour, selon l'état du philosophe et l'observation des données, une action est exécutée. Pour plus de clarté, nous avons représenté l'ensemble des actions implémentées sur une machine à états simplifiée. Les messages ne sont pas encore pris en compte.



II.3 Rôle des paramètres

Bien évidemment, le comportement du système est largement affecté par le choix des constantes de l'environnement. Par exemple, un choix de `deltaFManger` démesurément grand (en valeur absolue) par rapport à `deltaFPenser` et `deltaFFaim` a pour conséquence de simplifier grandement le problème : chaque philosophe peut passer son temps à penser et ne manger qu'une fois de temps pour rattraper son retard. Les fourchettes sont alors presque tout le temps disponible.

L'intérêt bien sûr n'est pas de regarder ses philosophes manger mais bien de définir des constantes équilibrées pour étudier un système nuancé. Après quelques essais, nous avons décidé de travailler dans un premier temps avec xxxxx var. On remarquera que le choix de `deltaFFamine` ne joue que sur la variable de mesure `compteurFamine` et non sur le fonctionnement même du système.

graphe évolution

Le choix de `deltaFFamine` est donc un problème de convention. Pour le contexte donné en fig.X, on trouve par exemple xxxxx. On choisira `deltaFFamine=60` pour les prochaines simulations.

Le choix du nombre de philosophe est nettement plus intéressant. En effet, on remarque vite que le comportement du système est assez chaotique : les scores de famine et de pensée peuvent varier drastiquement d'une configuration à l'autre, ceci sans que l'on ai pu identifier de relation logique entre l'effectif et ces scores. Voir figure XXX

graphe nb de philosophes

Ce comportement est problématique. Dans le cas où l'on a 8 philosophes, le comportement du système est désastreux. Il faut donc mettre en place une solution pour régulariser ce genre de cas, et pourvoir travailler avec un système moins imprévisible.

II.4 Ajout des messages

Le problème de beaucoup de configurations malheureuses, c'est que certains philosophes s'accaparent les couverts pendant que d'autres non jamais l'occasion de les ramasser. On peut essayer d'éviter cela en mettant en place un système de communication entre agents.

La communication entre agents passe par la classe `Message`. Un agent peut envoyer un message à un autre agent en créant une instance de `Message` spécifiant son identifiant, l'identifiant du destinataire, et le contenu du message. L'envoi est effectué via l'objet d'environnement avec la méthode `send` qui redistribue le message au destinataire.

Le message est alors stocké dans la boite aux lettres du destinataire dans l'attente d'être traité. Les messages sont ensuite régulièrement supprimés (par exemple à la fin de chaque tour) pour éviter d'accumuler de vieux messages. Pour le traitement, plusieurs solutions sont envisageables au niveau de l'implémentation d'un problème particulier.

Pour nos philosophes, nous avons décidé de mettre en place une solution simple : au début de son tour le philosophe consulte le dernier message qu'il a reçu, les autres sont ignorés. Le contenu du message intervient alors au niveau des conditions d'exécution des

actions, au même titre que les autres données du système.

Un premier type de message a été mis en place pour diminuer les scores de famine : `Help`. Cette démarche a été très fructueuse, donc nous nous sommes contentés de ce type de message dans un premier temps. Le message est émis par un philosophe lorsqu'il passe le seuil de famine, et est envoyé à ses deux voisins pour leur demander de lâcher leurs fourchettes. Ainsi, au tour suivant, le philosophe peut manger et on évite un état de famine prolongé.

même graphe nombre de philosophes

Conclusion