Systèmes multi-agents : la bille, le poisson et l'avatar

Matthieu CARON - Alexandre MOEVI

3 octobre 2016

1 Architecture et utilisation

1.1 Usage

En ouvrant l'archive, on peut constater dans le la présence de 4 répertoires : un `core` qui contient la base d'un système multi-agents et un pour chaque simulation (`particules`, `wator` et `game`). Chacun des trois simulations contient une classe Main pour lancer le programme.

Pour réaliser les différents simulations, il a été décidé d'utiliser la langage Python 3 et la bibliothèque graphique `Tkinter`. On peut vérifier l'installation de Tkinter en lançant la commande `python3 -m tkinter` (ou `python -m tkinter`) ou l'installer avec `sudo apt-get install python3-tk` (ou chercher le nom du paquet avec `sudo apt-cache search tk`).

La modification de la variable `$PYTHONPATH` se fait directement dans chaque Main. Néanmoins si ça ne marche pas il faut faire les étapes suivantes. Afin de permettre une architecture sous forme de paquetage en Python, il faut modifier la variable d'environnement `$PYTHONPATH` en ajoutant le chemin vers le répertoire `sci`

```
export PYTHONPATH=$PYTHONPATH:/home/pmatthieu/example/tps/sci-caron-moevi
```

Sans cette modification, les imports ne marcheront pas. Cette modification peut être faite dans un terminal (méthode temporaire) ou directement dans les fichiers `.bashrc` et/ou `.bash_profile`.

modifier les paramètres dans MainXXX.py, etc.

1.2 Paquetage `core`

Avant d'expliquer les trois simulations effectuées, il est nécessaire de présenter le paquetage `core`. `core` contient la description générique des objets nécessaires à la mise d'un SMA (système multi-agents). Le paquetage contient cinq classes.

La classe `Agent` qui représente un agent du système. Placé dans un `Environnement`, il connaît ses coordonnées (`self.x` et `self.x` en Python, qui correspondent à `this.x` et `this.y` en Java). Quand le `SMA` lui « donne la parole », l'agent décide en fonction de stratégie (méthode `self.decide()`) puis se met à jour dans son environnement (`self.update()`). L'agent possède également une méthode `place_agent()` qui concerne son affichage dans la fenêtre (classe `Window`).

La classe `AgentCreator` génère un ou plusieurs `Agent` du système. Il peut créer un seul type d'agent (uniquement des billes) ou plusieurs types d'agents (poissons et requins).

La classe `Environnement` représente l'espace du système sous forme de grille. Cette grille contient l'ensemble des agents et leur coordonnées. L'environnement peut être torique ou non.

La classe `SMA` est la classe qui controle tout, à l'initialisation elle va créer un environnement et va faire nbAgents appels à la méthode create de la classe AgentCreator. La classe `SMA` contient la méthode `run()` qui effectue le tour de parole (ou tick). Dans un tick, le `SMA` appelle un ou plusieurs agents avec la méthode `agent.decide()` en fonction du scheduling (`self.scheduling`). Le scheduling peut être séquentiel, équitable ou aléatoire. Un fois le tour fini, l'affichage est mis à jour avec la méthode `SMA.updateDisplay()`.

La classe `Window` concerne tout ce qui l'affichage. En se basant sur la bibliothèque graphique `Tkinter`, la classe permet l'affichage du système et l'affichage d'une grille si l'utilisateur le souhaite.

2 Simulations

2.1 Tube à particules

La première simulation, le paquetage `particules`, reproduit le comportement de billes (ou particules) dans un espace. Cet espace peut torique ou non ; s'il ne l'est pas, les particules rebondissent contre les murs. Les particules rebondissent également en collision.

La classe `Bille` hérite la classe `Agent` du paquetage `core`. L'agent `Bille` connaît son environnement, sa position (entiers `x` et `y`) et sa direction (vecteur à 2 dimensions `[a, b]`).

La stratégie d'une particule est la suivante :

— Dans la méthode `nextPos()`, la particule calcule sa prochaine possible destination en fonction de l'espace torique (apparition de « l'autre côté » de l'écran) ou non (rebond).

— Dans `decide()`, la particule regarde sa prochaine destination. Si la voie est libre, elle se prépare à se déplacer (`particule.bougera = True`). Sinon ça veut dire qu'elle essaye de se rendre dans une case occupée par une autre particule : c'est une collision.

— Dans `update()`, si la particule peut bouger, elle se met à sa nouvelle position en laissant libre son ancienne place et l'environnement se met à jour.

Dans le cas d'une collision (`particule1.collision(particule2)`), la bille « incidente », c'est-à-dire celle qui essaye d'aller dans une case déjà occupée, échange sa direction avec l'autre particule. Il a été décidé que la bille incidente ne se déplace pas pendant le tick de collision, malgré sa nouvelle direction. Afin d'avoir un comportement cohérent lorsque les billes remplissent la grille et donc qu'elles ne peuvent pas bouger.

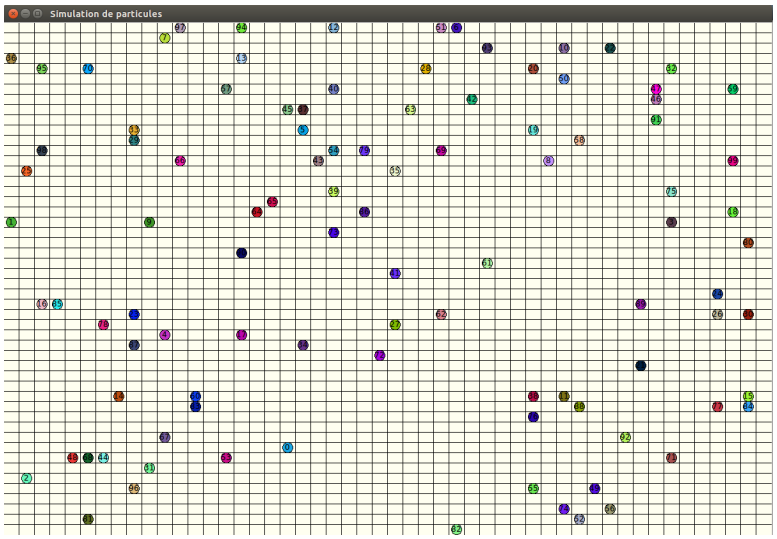

FIGURE 1 – Fenêtre d'exécution du programme `MainBille`. Les billes sont assignées d'une couleur aléatoire et d'un identifiant pour faciliter leur suivi.

2.2 Poissons et requins dans le golfe (du Bénin)

La deuxième simulation concerne le paquetage `wator`. Elle a pour but de voir comment des poissons (les agents `Fish` dans notre simulation) et des requins (agents `Shark`) cohabitent dans une zone.

Un agent `Fish` se déplace de façon aléatoire et se reproduit tous les `fishBreedTime` ticks. Pour se reproduire, le poisson doit se déplacer et donne naissance à un autre poisson sur la case où il était précédemment. Il observe donc dans le voisinage de moore les cases qui sont libres et en prend une au hasard si c'est possible. Enfin l'age augmente de 1 à chaque tick.

Un `Shark` est affamé, il a `dontStarve` ticks pour manger un `Fish`. S'il n'y arrive pas à temps, il meurt. À l'instar du poisson, il peut se reproduire tous les `sharkBreedTime` et donne naissance à un nouveau requin sur la case qu'il occupait avant.

Comme pour l'agent `Bille` de la simulation précédente, les agents `Fish` et `Shark` ont conscience de leur environnement et connaissent leur coordonnées dans l'espace.

Avec ces paramètres, le but est de trouver d'atteindre une situation d'équilibre. On veut éviter une pénurie de poissons (plus de poissons = mort des requins affamés = zone vierge de tout animal) et une absence de requins (les poissons vont se reproduire jusqu'à remplir entièrement la zone).

La classe FishAndSharkCreator prend un nombre de Shark, un nombre de Fish et à chaque appel à create renvoie aléatoirement un Fish ou un Shark dans une place libre trouvé aléatoirement.

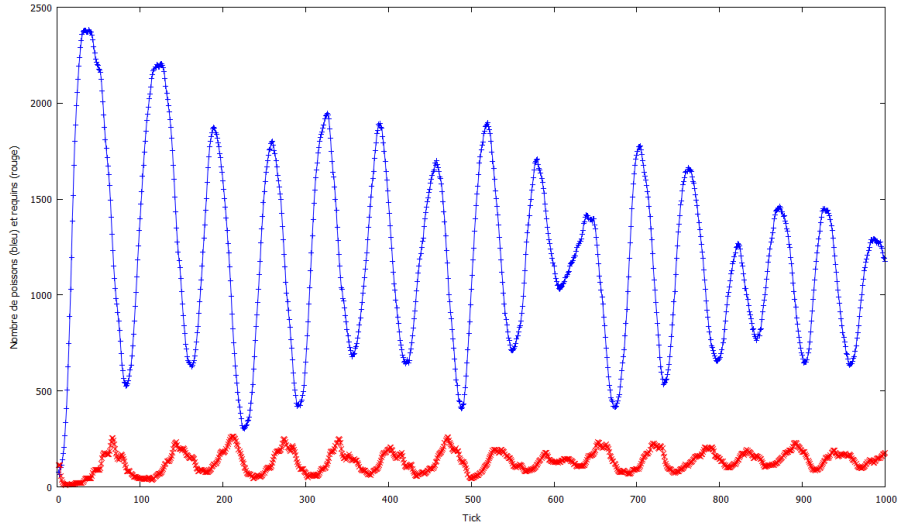

FIGURE 2 – Courbes de population.

2.3 Avatar et Hunters

Enfin dans cette simulation correspondant au package `game`, le but en tant qu'Avatar est de ne pas se faire attraper par les Hunters sinon la partie est perdue. En revanche pour gagner la partie si vous attraper un bonus d'invicibilité, vous pouvez pendant un court momment attrapper les Hunters qui tentent de fuir. Une fois tous les Hunters attrapés, la partie est gagnée.

Un agent `Avatar` se déplace en fonction du dernier imput (gauche,droite,haut ou bas) du joueur. Si l'Avatar fonce dans un mur il ne se déplace plus, si l'Avatar fonce dans un Hunter il meurt. Enfin à chaque appel de update il va calculer sa matrice de Dijkstra qui sera enregistrée à l'environnement ainsi tous les Hunter peuvent y acceder.

Un agent `Hunter` se déplace en fonction de la matrice de Dijkstra et des autres agents (il n'ira pas dans un mur ni dans un autre Hunter mais ira tuer l'Avatar) en choisissant uniquement une position améliorante (avec un score strictement inférieur à sa position actuelle, l'inverse si l'Avatar est invulnérable). Parmis les futurs position améliorante il en choisit une de libre au hasard pour éviter de faire d'abord tous les déplacements vers le bas puis tous les déplacements vers la droite (par exemple).

Un agent `Defender` apparait à un emplacement aléatoire vide sur la map et reste en vie n tours. Il permet de donner l'invicibilité.

Enfin un agent `Wall` qui est placé à la création de l'environnement et influe sur le calcule de la matrice de Dijkstra. Les murs sont placés de manière aléatoire.

2.3.1 Remarques sur le GameAgentCreator

Le SMA à l'initialisation appelle nbAgents fois la méthode create de la classe GameAgentCreator et l'ajoute à l'environnement avec la méthode ajouteAgent(Agent) de la classe environnement. Le seul soucis actuel c'est que la

classe SMA appelle decide() sur tous les agents y compris les murs... Néanmoins le jeu et fluide quand même. Donc le GameAgentCreator crée d'abord tous les murs et ensuite les Hunters puis l'Avatar. Problème on peut se retrouver enfermé.