Université Claude Bernard Lyon 1

Multi-Agents and Self-* Systems
TP

Rapport TP Tri collectif

Élèves :
Paul Flagel
Nathan Etourneau

Enseignant :
Salima Hassas

9 novembre 2021

Table des matières

Introduction

Le but de ce TP est d'implémenter un système multi-agents impliquant des agents répartis sur une grille, ainsi que des objets de deux classes (A et B). Ce système multi-agents a pour but de regrouper les objets par type d'objet, tout en s'assurant que les agents n'ont qu'une perception locale de l'environnement. Les propriétés des agents sont tirées de cet article de recherche : *Deneubourg, Jean-Louis et al.* The dynamics of collective sorting robot-like ants and ant-like robots.

L'intérêt de ce TP est de montrer que même avec une perception extrêmement réduite de l'environnement (absence de communication entre agents, perception approximative de l'environnement, mémoire faible), les agents sont capables de trier les objets par classe, et une intelligence collective émerge de comportements individuels très simplistes.

Dans ce rapport, nous allons présenter les choix d'implémentation effectués pour les multiples classes du sujet.

1 Classe Object

La première classe présentée ici est la classe Objet. Sur la grille, il y a un nombre n_a d'objets de classe A, et un nombre n_b d'objets de classe B. Une instance de la classe Objet est créée pour chacun des $n_a + n_b$ objets. Cette classe est très simple, c'est un simple objet encapsulant les attributs suivants :

- key : la clé d'identification de l'objet
- category : la catégorie de l'objet représenté par la classe Objet.
- position : la position de l'objet sur la grille sous forme de tuple (ligne, colonne). Il y a une petite subtilité : lorsque l'objet est porté par un agent, la position est indéfinie et a donc pour valeur None.
- parent : le type d'objet dans lequel la classe objet réside. Cela peut être un objet de classe Cell ou un objet de class Agent, en fonction de si l'objet est sur la grille ou porté par un agent.

Un objet est totalement passif et n'a donc aucune méthode particulière associée.

2 Classe Cell

De manière analogue à la classe Objet, on définie une classe Cell. Il y aura autant d'objets de la classe Cell instanciés que de cellules dans la grille (c'est à dire $N \times M$). Cette classe a les attributs suivants :

- position : La position de la cellule sous forme de tuple d'entiers (ligne, colonne).
- agent : un attribut contenant le cas échéant, l'instance de la classe Agent associée à l'agent présent dans la cellule. S'il n'y a pas d'agent, cet attribut contient None.
- objet : le cas échéant, l'instance de la classe Objet associée à l'objet contenu par la cellule. S'il n'y a pas d'objet, cet attribut est None.

Une cellule est totalement passive et n'a donc aucune méthode particulière associée.

3 Classes Agent et AgentData

Nous avons ensuite implémenté 2 classes : Agent et AgentData. La classe Agent comporte un ensemble de méthodes et d'attributs modélisant la connaissance de l'environnement de l'agent, ainsi que les actions qu'il est à même de réaliser. Les attributs que possède l'objet Agent sont :

- key : la clé de l'agent, pour identification future
- kplus : la valeur de la variable k_+ utilisée ici
- kminus : la valeur de la variable k_- utilisée ici
- memory_buffer_size : la taille maximale de la mémoire de l'agent
- memory : la chaîne de caractère représentant la mémoire de l'agent : ce sont la liste des objets rencontrés sur les dernières cellules visitées. Les caractères 'A' ou 'B' correspondent à un objet de type A ou B, et '0' correspond à une absence d'objet.
- error_rate : le taux d'erreur de classification des objets.
- object : si l'agent porte un objet, contient l'instance de la classe Object associée à cet objet. Sinon, contient None.

L'objet Agent a également les méthodes suivantes :

- perception(environment) : retourne la liste des cellules vides autour.
- action(environment, empty_cells) : effectue une étape d'action.
- update_memory(category) : met à jour la mémoire avec la dernière cellule rencontrée.
- get_frequency(category) : renvoie la fréquence de la catégorie donnée dans la mémoire, prenant en compte le taux d'erreur si besoin.

Il nous a semblé très important de faire en sorte que l'objet agent ne contienne pas l'attribut position, afin de représenter le fait que les agents n'ont pas une connaissance de haut niveau de leur environnement, seulement une perception locale. Cette perception locale est ici obtenue en requêtant un objet environnement, qui sera présenté après.

Nous rappelons que lorsqu'un ALR porte un objet, il est possible qu'il passe sur une cellule contenant également un objet, comme on peut le lire dans la partie 2 de l'article "If it has picked up an object, then at each step that it finds itself in an unoccupied point it decides whether or not to put the object down". Nous allons ici détailler en pseudo-code la méthode action de l'objet agent, pour mieux comprendre l'implémentation.

Algorithm 1: Méthode action

```
require empty_cells, environment

if empty_cells ≠ [ ] then
    destination = pick_random_element_in(empty_cells)
    environment.move(this.key, destination)
cell = environment.get_agent_cell(this.key)

if cell.object is not None then
    to_push = cell.object.category
else
    to_push = '0'

if this.object is not None then
    if cell.object is None and this.will_drop(this.object.category) then
        cell.object = this.object
        this.object = None
        cell.object.parent = cell
        cell.object.position = cell.position
else if cell.object is not None and this.will_pick(cell.object.category then
    this.object = cell.object
    cell.object = None
    this.object.position = None
    this.object.parent = this

this.update_memory(to_push)
```

Par conséquent, pour des raisons de commodité, nous avons implémenté un objet AgentData, permettant de stocker les agents ainsi que leur position, séparément. Pour pallier ce problème, nous avons ainsi créé l'objet AgentData, qui contient les attributs suivants :

— agent : l'instance de l'objet Agent pour lequel on souhaite stocker la position
— position : la position de l'agent

4 Classe Environment

La dernière classe que nous avons implémenté est la classe Environnement. L'objet Environnement est instancié une seule fois et sert à encapsuler l'ensemble de la logique : il contient les agents, les objets, les cellules. Lorsque les agents ont besoin de connaître des propriétés de leur environnement, ils requêtent l'objet environnement.

Les méthodes de l'objet environnement sont les suivantes :

— N : le nombre de lignes de la grille
— M : le nombre de colonnes de la grille
— grid : un tableau de taille $N \times M$ comportant à chaque case (i,j) un objet de classe Cellule, comportant éventuellement un objet de classe Object et/ou un agent de classe Agent associé.
— agents : un dictionnaire ayant pour clefs les clefs des agents et pour valeur les instances de la classe AgentsData associée.

— objects : un dictionnaire ayant pour clefs les clefs des objets et pour valeur les instances de la classe Objet associés.

L'objet de classe Environment a également les méthodes suivantes :
— init_grid(na, nb, n_agents, kplus, kminus, memory_buffer_size, error_rate) : initialise les cellules, les agents, les objets, la grille et les dictionnaires d'agents et d'objets.
— init_objects(na, nb) : initialise les objets sur la grille, les objets
— init_agents(n_agents, kplus, kminus, memory_buffer_size, error_rate)
— valid_cell(row, col) : vérifie qu'une cellule est bien dans la grille et n'est pas hors limites
— empty_cells(key) : retourne les cellules vides (sans agent dedans) autour de l'agent ayant pour clef key
— move(key, direction) : déplace l'agent ayant pour clef key dans la direction donnée en argument (ex : pour la direction Nord-Ouest, $direction = (1, 1)$, pour la direction Sud, $direction = (-1, 0)$.
— get_agent_cell(key) : renvoie la cellule contenant l'agent ayant la clef key.
— will_pick(category) : retourne True si l'on ramasse l'objet de la catégorie donnée, compte tenu de la mémoire, et False sinon. Les probabilités associées sont celles de l'énoncé du sujet.
— will_drop(category) : retourne True si l'on dépose l'objet de la catégorie donnée, compte tenu de la mémoire, et False sinon. Les probabilités associées sont celles de l'énoncé du sujet.

5 Script principal

Notre code étant structuré en classes, ces dernières permettent une programmation transparente du code : tous les détails et la logique sont encapsulés dans la programmation des classes et des méthodes. Nous écrivons ici la boucle principale en pseudo-code, on peut

constater que la lecture est transparente.

Algorithm 2: Boucle principale

```
require N, M, na, nb, n_agents, k_plus, k_minus, memory_buffer_size,
  n_rounds, error_rate

env = Environnement(
            N, M, na, nb, n_agents, k_plus, k_minus,
            memory_buffer_size, error_rate
            )

for round = 1 To nb_rounds do
    for agent_data in shuffle(env.agents) do
        agent = agent_data.agent
        empty_cells = agent.perception(env)
        agent.action(env, empty_cells)
        update_plot()
    end
end

Return the plot
```

6 Interface graphique

Afin de visualiser le tri collectif, nous avons décidé que tous les K rounds, nous afficherions une image représentant la position des objets. Nous n'affichons pas les agents car ils représentent un moyen afin d'arriver à l'objectif qui est de trier les objets, et donc ils parasitent la visualisation.

Nous aurions aimé afficher en temps réel mais compte tenu du nombre d'étapes nécessaires afin de trier les objets, un tel affichage demanderait trop de puissance pour être implémenté en Python, il faudrait le faire en C.

À défaut d'afficher en temps réel, nous avons décidé d'implémenter une petite application permettant de régler les paramètres et d'afficher de manière claire l'apprentissage. Cette application est disponible en ligne à l'adresse `https://share.streamlit.io/nathanetourneau/sma-tp3-tri_collectif/app.py`.

Après 500 000 rounds, on obtient une excellente classification des objets

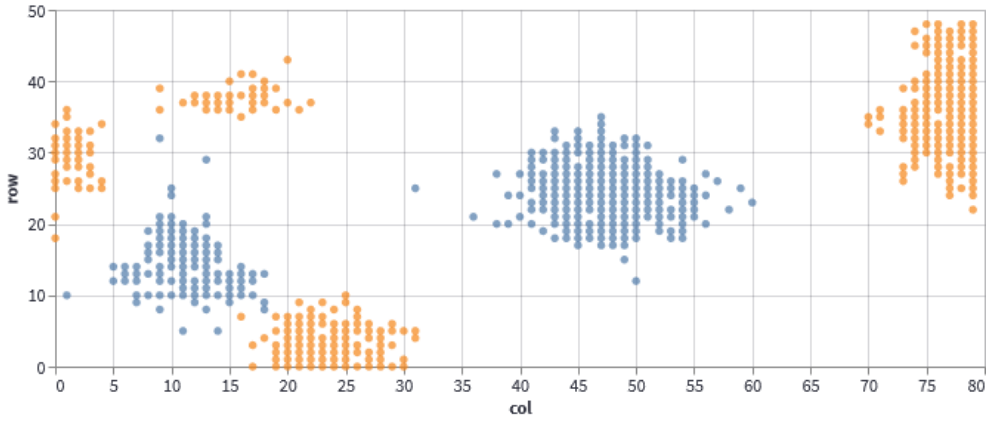

FIGURE 1 – Position des objets après 500 000 rounds avec une grille 80×49, 300 objets de chaque classe, 20 ALR, m = 15

7 Introduction d'une erreur de classification des objets

Nous avons introduit une erreur de classification dans la reconnaissance d'objets : lorsqu'un objet passe sur une cellule, il a une probabilité e de se tromper dans la classification de l'objet. Ainsi, pour calculer la fréquence des objets dans la mémoire, nous tenons compte de cette éventuelle erreur. En notant n_A le nombre d'objets de la classe A, n_B le nombre d'objets de classe B, et m la longueur de la mémoire, la formule pour calculer la fréquence de la classe A est :

$$f_A = \frac{n_A + e \cdot n_B}{m}$$

Cette erreur de classification rend les clusters mal définis : des clusters se forment, et ils se forme d'autres groupes au sein du cluster. Nous ne trouvons ici pas les mêmes résultats que l'article.

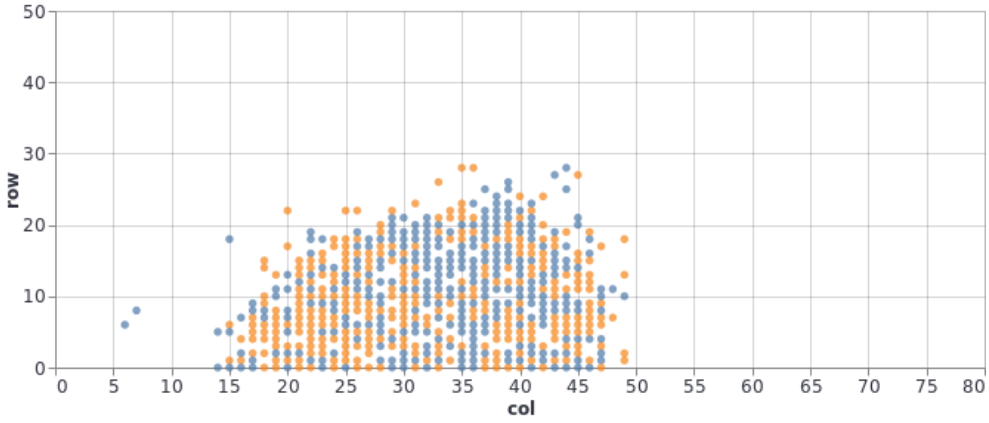

FIGURE 2 – Position des objets après 5 000 000 rounds avec une grille 80×49, un taux d'erreur e = 0.2, 300 objets de chaque classe, 20 ALR, m = 15

Conclusion

Ce projet informatique était une première approche de systèmes multi-agents avec des agents réactifs. On remarque qu'à partir de règles élémentaires et de simple perception locale, on obtient une intelligence collective. Nous pouvons être satisfaits du code produits car le code reproduit les résultats de l'article de recherche initial.