Université Claude Bernard Lyon 1

Multi-Agents and Self-* Systems
TP

Rapport TP Tri collectif

Élèves :
Paul Flagel
Nathan Etourneau

Enseignant :
Salima Hassas

8 décembre 2021

Table des matières

Introduction

Ce rapport présente le travail effectué pour la deuxième partie du TP concernant le tri collectif, qui consiste à répartir des objets et des agent sur une grille, et à montrer que à partir de règles locales et très simples, il est possible de trier les objets et de faire émerger une intelligence collective.

Ce projet a été implémenté en Python. Les classes implémentées ont été détaillées dans la partie 1 du rapport. Elles seront moins détaillées ici pour laisser de la place à l'implémentation de la stratégie collective, mais ce rapport peut être lu indépendamment du premier

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, un nombre n_b d'objets de classe B, un nombre n_c d'objets de type C. Une instance de la classe Objet est créée pour chacun des $n_a + n_b + n_c$ 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.

La classe objet est inchangée par rapport à la version 1. Toutefois, un objet peut cette fois être la classe C.

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.
— pheromone : la valeur de la quantité de phéromones en cette cellule

Nous avons simplement rajouté un attribut phéromone dont le calcul est détaillé dans la section sur la classe Source.

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'environ-

nement 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', 'B' ou 'C' correspondent à un objet de type A, B ou C, et '0' correspond à une absence d'objet.
- object : si l'agent porte un objet, contient l'instance de la classe Object associée à cet objet. Sinon, contient None.

Les nouveaux attributs sont :

- patience : le temps courant d'attente (en rounds) de l'agent pour de l'aide pour un objet de type C
- max_patience : le temps maximum que l'agent est prêt à attendre.
- is_helped_by : le cas échéant, l'instance de l'agent qui aide. Sinon, None
- is_helping : le cas échéant, l'instance de l'agent que cet agent est en train d'aider. None sinon.

L'objet Agent a également les méthodes suivantes :

- perception(environment) : retourne un dictionnaire ayant pour clef les directions menant à des cellules vides autour de l'agent, et pour valeur la quantité de phéromone dans ces cellules.
- 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.

Par rapport à la version 1, les modifications sont les suivantes :

- Pour introduire le comportement collectif, il faut avoir des attributs pour préciser si l'agent est en train d'attendre de l'aide, en train de se faire aider, ou en train d'aider
- La fonction action a été largement modifiée, pour tenir compte du comportement collectif. Il faut ainsi distinguer dans le code si la catégorie des objets est de type C, et agir en conséquence.
- La fonction perception renvoie cette fois un dictionnaire : un agent ne se dirige plus nécessairement de manière aléatoire, s'il détecte des phéromones autour de lui, il se dirige vers la cellule contenant le maximum de phéromones.

Pour ne pas stocker la position au sein de l'objet Agent, nous avons implémenté un objet AgentData. Cet objet permet de stocker les agents ainsi que leur position, séparément. Cet objet a les attributs suivants :

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

— position : la position de l'agent

4 Modification de la méthode action

La méthode action a du être modifiée pour tenir compte de la présence d'objets de type C. Le comportement est le suivant :

— Si un agent 1 est lié à un autre agent 2 (pour l'aider à porter un objet de type C, alors lors de la boucle principale, la méthode agent1.action() n'effectuera rien, toutes les actions de l'agent 1 seront effectuées dans la boucle de l'agent 2.
— Si l'agent croise un objet de type A ou B, alors il agit normalement.
— Si un agent 1 croise un objet de type C, alors il peut décider de le ramasser ou non. S'il décide de le ramasser, il cherche un agent 2 à portée immédiate pour l'aider. Si un agent est à portée immédiate pour l'aider, alors l'agent 1 et l'agent 2 sont reliés, et lorsque l'agent 1 se déplace dans une cellule, l'agent 2 se déplace à l'ancienne position de l'agent 1. Si aucun agent n'est à portée pour l'aider, alors il émet des phéromones (en s'inscrivant à la liste des émetteurs de l'environnement) et commence à attendre.
— Si un agent 1 croise un objet de type C, alors il peut décider de le ramasser ou non. S'il décide de le ramasser, il cherche un agent 2 à portée immédiate pour l'aider. Si un agent est à portée immédiate pour l'aider, alors l'agent 1 et l'agent 2 sont reliés, et lorsque l'agent 1 se déplace dans une cellule, l'agent 2 se déplace à l'ancienne position de l'agent 1. Si aucun agent n'est à portée pour l'aider, alors il émet des phéromones (en s'inscrivant à la liste des émetteurs de l'environnement) et commence à attendre.
— Si l'agent est en train d'attendre, il cherche un autre agent 2. Si il y en a un, il se lie à lui et agit comme ci-dessus. Sinon, il incrémente son compteur d'attente. Si cela fait trop longtemps qu'il attend, il effectue un mouvement aléatoire et arrête d'attendre.

5 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.
— sources : la liste des sources

L'objet de classe Environment a également les méthodes suivantes :
— init_grid(na, nb, nc, 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, nc) : 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.
— agent_able_to_help(key) : si cela est possible, retourne une instance de l'objet Agent associée à un agent en position d'aider l'agent ayant la clef key. Pour pouvoir aider, l'agent doit être à une distance d'une case et ne pas porter d'objet.
— register_pheromone_source(key, patience) : créée un objet Source avec la position de l'agent ayant la clef key, avec la valeur de patience patience, et l'ajoute à la liste des sources.
— update_pheromones() : met à jour l'ensemble des cellules, en recalculant la valeur des phéromones à partir de la liste de sources.

6 Classe Source et calcul des phéromones

Nous avons créé une classe Source, ayant comme attributs la position et la "patience" (le nombre de rounds passés à attendre l'aide d'un autre agent) d'un agent.

Chaque fois qu'un agent souhaite émettre des phéromones, on crée un objet source que l'on ajoute dans une liste de sources de l'environnement. En pratique, un agent appelle la méthode register_source(key, patience) de l'environnement.

À la fin de chaque round, cette liste de sources est parcourue, et la valeur des phéromones de chaque cellule est recalculée selon la formule :

$$P(cell) = \sum_{\text{source} \in S} \gamma^{t[source]} \cdot \max(1 - \frac{\text{distance}(source, cell)}{\text{signal\_range}}, 0)$$

Avec γ le taux de décroissance temporelle du signal (env.ratio dans le code), t[source] le temps d'attente de la source, distance la distance L^∞ entre deux cellules : le maximum entre l'écart horizontal et l'écart vertical entre deux cellules, signal_range la portée du signal (env.signal_range dans le code).

La liste des sources est ensuite vidée, et la valeur des phéromones de chaque cellule est actualisée selon la formule ci-dessus. Ceci est effectué par un appel à la méthode update_pheromones() de l'environnement.

Au round suivant, chaque agent qui souhaite émettre des phéromones crée un objet source contenant sa position et sa patience. À la fin du round, les phéromone correspondants seront émis.

7 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 1: Boucle principale

```
require N, M, na, nb, nc, n_agents, k_plus, k_minus, memory_buffer_size,
  n_rounds, ratio, signal_range, max_patience

env = Environnement(
            N, M, na, nb, nc, n_agents, k_plus, k_minus,
            memory_buffer_size, ratio, signal_range, max_patience
            )

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)
    end
    env.update_pheromone()
    update_plot()
end

Return the plot
```

8 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 ou Java.

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/V1/app.py` pour la V1,
`https://share.streamlit.io/nathanetourneau/sma-tp3-tri_collectif/V2/app.py` pour la V2.

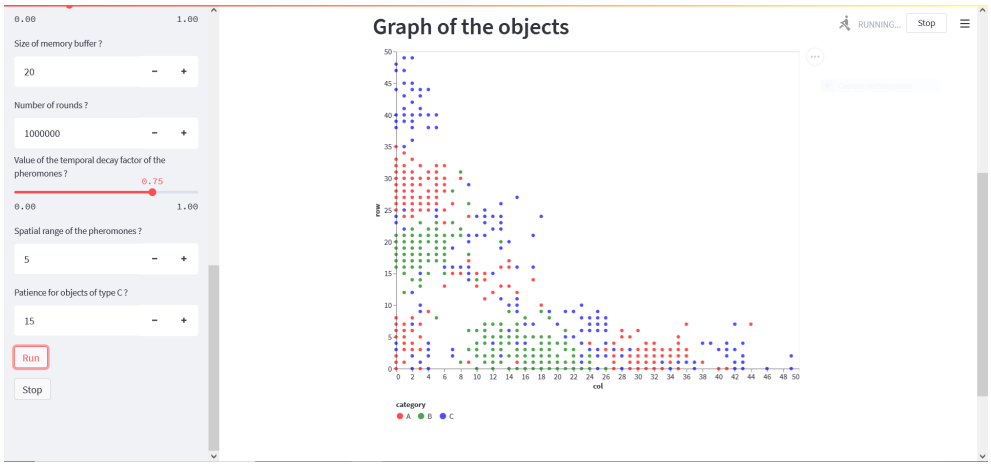

FIGURE 1 – Capture d'écran de l'application

L'application codée comporte une sidebar pour les réglages, un bouton start pour commencer la simulation, et un espace au centre pour afficher les résultats

9 Exécution, test et analyse

Pour exécuter notre code, il y a 3 possibilités :

— Accéder à l'une des deux versions accessibles en ligne :
`https://share.streamlit.io/nathanetourneau/sma-tp3-tri_collectif/V1/app.py` pour la V1
`https://share.streamlit.io/nathanetourneau/sma-tp3-tri_collectif/V2/app.py` pour la V2
Ceci est plus lent que de les faire tourner en local.

— Récupérer notre code sur github, aller dans le dossier d'intérêt (V1 ou V2), installer les dépendances dans requirements.txt, puis effectuer l'instruction **streamlit run app.py**

— Récupérer notre code sur github, aller dans le dossier d'intérêt (V1 ou V2), installer les dépendances dans requirements.txt, puis effectuer l'instruction **python app.py**. Cette version est moins jolie et pratique que l'app streamlit

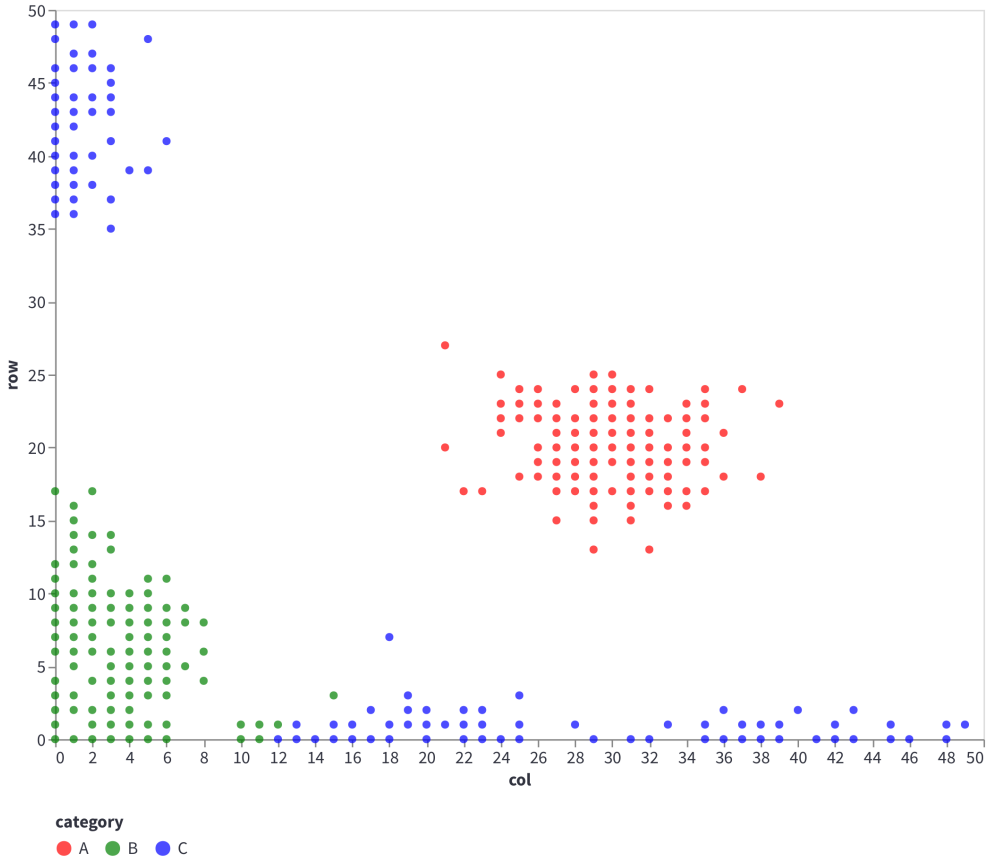

FIGURE 2 – Position des objets après 1 000 000 rounds avec une grille 50×50, 100 objets de chaque classe, 20 ALR, memory = 10, portée des phéromones = 2, patience = 15, ratio = 0.75, k+ = 0.1, k- = 0.3

La convergence est bien plus lente que dans la version sans agent de type C : c'est normal, les objets de type C ralentissent considérablement le processus de clusterisation. En effet, pour déplacer un objet de type C, il faut la présence d'un agent à proximité. Si la portée des phéromones est faible, le tri des objets de type C est lent, mais le tri des objets de type A et B est rapide et n'est que peu perturbé par les objets de type C. En revanche, si la portée des phéromones est plus grande, le tri des objets de type C est plus rapide mais les phéromones gênent le tri des objets de type A et B.

On remarque que lorsque l'on met 0 objet de type C, le tri fonctionne parfaitement, et le tri s'effectue à la même vitesse que avec la version 1 du code.

Conclusion

Dans cette partie 2, nous avons mis en place un comportement collectif pour les agents, à partir de règles locales, et avec un système multi-agents décentralisé. Nous remarquons que le tri fonctionne toujours, cela signifie que nous avons réussi à créer de l'intelligence globale à partir de règles basiques et locales.