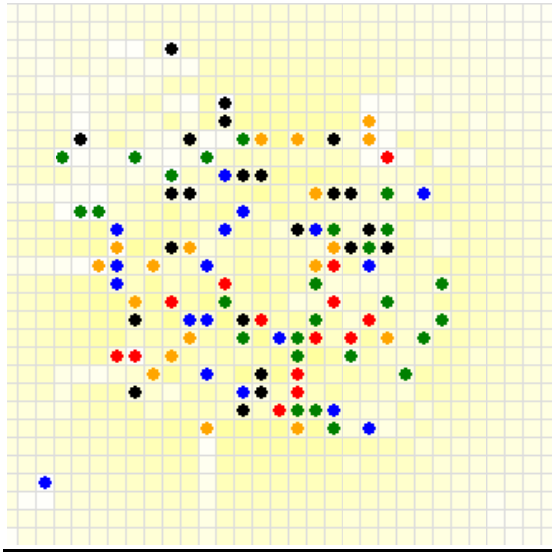
# Projet info 2014

## *Système multi-agents*

**Cornil Bruno** **Fara Sam**



## *Sommaire :*

**Remarque importante :** plusieurs modules ( mas, env ... ) ont été modifiés. Afin d'avoir un programme opérationnel, il faut donc lancer le code à partir de nos modules.

## 1. Introduction

Afin de mener à bien le projet d'informatique notre démarche s'est rythmée en deux temps : la compréhension du code fourni et la rédaction personnelle d'un code complémentaire.

## 2. Méthode récurrente

Initialement, après avoir étudié attentivement le code annexe, nous avons opté pour une approche par défaut. C'est à dire un environnement dans lequel les agents se déplaçaient de manière totalement aléatoires et ceci dans le but de tester notre maîtrise du code de base.

Le premier objectif réussi, nous nous sommes lancés dans l'implémentation des règles concernant les agents. Une approche basée sur la logique et sur l'efficacité étant de rigueur, notre démarche consistait à se poser les bonnes questions : où voulons-nous aller et comment aborder le problème?

## 3. Règles des agents

### 3.1 RA1

La création de fonctions qui permettent de parcourir la grille pour atteindre les différents taux de sucre, qui les mettent à jour, qui déplacent les agents et la configuration du dossier " test.py ", nous ont naturellement menés à RA1. Bien vite un soucis s'est révélé. Notre programme ne faisait pas ce qu'on espérait.

En effet, tous nos agents se rejoignaient sur les pics de la distribution gaussienne ce qui se traduisait par un regroupement des " points rouges " sur notre grille en deux endroits distincts. L'évidence nous a alors frappés ; nous avions oublié de rédiger les fonctions gérant les interactions entre agents. Pour ce faire, nous avons opté pour une stratégie liant 2 listes en parallèle. La première contient les agents présents dans l'environnement. La seconde est une liste vide en chaque début de cycle qui reçoit pour chaque agent la position convoitée par celui-ci. L'intérêt de cette méthode se comprend lorsque

l'on se rappelle que le code utilise des boucles for. En effet, nous parcourons notre liste d'agent un à un en leur appliquant individuellement des modifications. Ce qui fait que l'agent en position i dans la liste se voit modifié avant celui en i+1. Ainsi, nous pouvions tester si la position que convoite l'agent " i+1 " n'avait pas déjà été attribuée précédemment.

Après cet apport plus que nécessaire, notre SMA tournait de manière correcte. Nous sommes donc passés à la programmation des autres règles relatives aux agents.

### 3.2 RA2

Afin de pouvoir l'exécuter correctement, nous avons dû modifier la liste des agents. Cette évidence nous est apparue assez rapidement après analyse de la question " Comment gérer la mémoire? ". Dès lors, la liste de l'agent s'est vue rallonger d'un paramètre. Ayant déjà programmé RA1, l'utiliser pour RA2 semblait une solution adéquate. Nous avons donc stocké dans la liste agent à l'emplacement " mémoire " la position renvoyée par la première règle.

Bien que ce défi ne nous semblait pas insurmontable, nous nous sommes heurtés à un soucis concernant le déplacement. En effet, réutiliser la méthode précédemment codée était infaisable. Auparavant, l'agent pouvait voir sa position changer de plusieurs cellules à la fois tandis que pour cette règle nous avions besoin d'une modification systématique d'une seule case. Une première approche était de jouer avec la vision en la définissant égale à 1 et à chaque cycle déplacer l'agent d'aussi loin que portait sa vue. Cette idée fut immédiatement oubliée. En effet, le concept de la mémoire n'avait désormais plus aucun sens avec une vision de un. Pour résoudre ce problème, nous avons simplement modifié la position de l'agent d'une ligne ou d'une colonne à chaque cycle pour atteindre la cellule enregistrée dans la mémoire.

Notre règle deux marchait désormais correctement.

### 3.3 RA3

Sur base de tout ce qui avait déjà été codé, RA3 n'a pas été un gros challenge. Nous voulions que l'agent se rende sur la cellule où se trouvait le plus petit taux de sucre supérieur à son métabolisme dans son voisinage. Globalement, cela s'est traduit par une simple inégalité à rajouter.

## 3.4 RA4

Afin d'avoir une règle 4 opérationnelle, nous devions avant tout analyser de

manière efficace le voisinage d'une position donnée. Au lieu d'épiloguer durant de longues phrases, laissons parler le code :

```
for distance in range(1,vision+1):
    for direction in [(0,1),(0,-1),(1,0),(-1,0)]:
        current_cell_position = lig+distance*direction[0], col+distance*direction[1]
        if p.occupied_cell_agent(pop, current_cell_position) and position != current_cell_position:
            agent_sugar_add += get_sugar_level(p.get_agent_position_list(liste,current_cell_position))
            agent_number+=1
if agent_number !=0:
    agent_average_sugar = agent_sugar_add / agent_number
    if agent_average_sugar < get_sugar_level(agent):
        is_completed = False
        forbidden_position.append((lig,col))
```

---

Cette image représente assez bien la partie cruciale de RA4 ; si l'agent possède un taux de sucre supérieur à la moyenne locale, il se voit obligé de trouver une autre cellule. Comme pour RA3, la structure du code étant déjà mise en place grâce aux deux premières règles, il suffisait d'implémenter cette fonction au bon endroit pour profiter des stratégies précedemment élaborées concernant, par exemple, les interactions entre agents.

# 4. Règles de la population

## 4.1 OA1

Comme expliqué précedemment ( c.f 3.1 ), notre programme est basé sur une boucle for parcourant notre liste d'agents. Pour travailler correctement avec OA1, il nous suffisait de légèrement modifier notre approche en passant d'une boucle for à une boucle while. En effet, grâce à un " random " nous sélectionnons au hasard un agent et excécutons le reste du code. Ensuite, nous le supprimons de la liste. C'est ainsi que notre while prend son sens : tant que notre liste n'est pas vide, la méthode explicitée ci-dessus est d'application.

## 4.2 OA2

Afin de pouvoir utiliser correctement la " méthode boucle for ", nous nous sommes dit que trier la liste dans l'ordre croissant des taux de sucre était la meilleure chose à faire. C'est dans ce but que nous avons retranscrit et adapté le code du tri par séléction pour pouvoir l'appliquer à notre liste d'agents en chaque début de cycle.

Les règles de base concernant les agents et la population étant désormais créées, nous sommes passés à la partie plus amusante du projet : les ajouts personnels.

## 5. Les apports personnels

Une fois en possession d'un SMA fonctionnel, nous avons décidé de rajouter quelques concepts. C'est surtout dans l'interaction entre agents que nous nous sommes divertis.

Dans un premier temps, nous avions imaginé le scénario suivant : si deux agents se retrouvent sur une même case, ils se battent pour celle-ci. L'idée cachée derrière le combat était simplement une version informatisée d'un lancé de dés. Pour chacun des agents, un random de 1 à 6 était attribué et celui qui recevait la valeur la plus grande se voyait octroyer la cellule disputée. Par la suite, la relation violente entre agents a laissé place à un peu plus de tendresse : la reproduction sexuée. Un enfant naissait de la réunion de deux agents sur une même case. Les caractéristiques des parents étaient attribuées avec un random au nouveau venu.

Une notion de vieillissement a également été introduite. Une fois l'agent arrivé en fin de vie, il mourrait. Ce qui se traduit par sa suppression dans la liste d'agent.

Enfin, nous nous sommes lancés dans le codage de la co-existence de plusieurs communautés sur un même environnement. Cet ajout représentait notre défi initial. En effet, dès le départ, c'est cet objectif que nous nous étions fixés et c'est avec grande fierté que nous l'avons atteint.

## 6. Conclusions

Après quelques semaines passées sur ce programme, différentes conclusions peuvent être tirées.

Le projet d'info nous a permis de nous rendre compte qu'au fond, avec seulement trois structures ( if, while, for ) il est possible de faire énormément de choses. C'est une réalité à laquelle nous avons été sensibles.

De plus, le travail en binôme a été enrichissant dans la mesure où plusieurs idées se confrontent. Grâce aux discussions et au bon sens, celles-ci peuvent toujours être combinées pour tirer le meilleur des différentes approches.

Pour terminer, le projet informatique a été bénéfique pour nous familiariser avec du code et nous préparer aux épreuves de janvier.