

TP2 Version 1 - SMA

Polytech Lyon

MANG Quentin | Système multi-agent | 08/12/2021

Table des matières

[1. Fonctionnement du TP 1](#_Toc89875249)

[2. Version 1 sans taux d’erreur 2](#_Toc89875250)

[3. Version 1 avec taux d’erreur 5](#_Toc89875251)

[4. Démonstration 6](#_Toc89875252)

# Fonctionnement du TP

Le projet pour le deuxième TP de la matière « Système Multi-Agent » est décomposé en 4 classes java (*Main, Environnement, Agent et Direction*), développée sous IntelliJ. Un objet est représenté par un caractère tandis que l’agent est représenté par la classe *Agent*.

Les 4 classes et leurs fonctionnalités :

* *Main.java* : Classe principale de l’application permettant d’exécuter le code du TP. Dans cette classe, nous retrouvons la fenêtre Java Swing montrant le déplacement des objets par les agents en temps réel.   
  De plus, cette classe contient la variable « *e* » de type « *Environnement* » initialisant les différentes variables du TP comme la taille de la carte, le nombre d’itération, le nombre d’agent, le nombre d’objet A et B, k-moins et k-plus, la taille de la mémoire et le pourcentage d’erreur dans la reconnaissance des objets.
* *Environnement.java* : Classe de l’environnement dans lequel évolue les agents. L’architecture importante de cette classe est composée de deux « *map* » : une représentant les agents et une les objets.   
  En effet, représenté les agents et les objets sur la même *map* rendait peu lisible le déplacement de ces derniers. Pour repérer de manière unique les agents au sein de l’environnement, nous avons utilisé une *Hashmap* prenant pour clé un « *agent* » et en valeur, ces coordonnées x et y.  
  Les fonctions principales de cette classes sont :
  + *Void initObjet()*: prenant en entrée le nombre d’objet A et B et permet de placer les objets de manière aléatoire sur la carte.
  + *initAgent()*: prenant en entrée le nombre d’agent et permet de placer les agents de manière aléatoire sur la carte. Cette fonction retourne la *Hashmap* d’agent avec ces coordonnées.

Pour respecter la conception environnement / agent, nous avons implémenter les fonctions suivantes :

* *PerceptionSeDeplacer()*: prenant en entrée un agent. Cette fonction retourne une *hashmap* de type « *Direction* » désignant les directions possibles de l’agent en fonction de son emplacement actuel sur la carte.
* *Void deplacement()*: prenant en entrée un agent et une direction choisi de manière aléatoire par l’agent (le choix se déroule dans la classe *agent*). Cette fonction va actualiser la carte, la carte d’agent ainsi que les coordonnées de l’agent dans la *hashmap*.
* *PerceptionPrendre()*: prenant en entrée un agent et retournant la valeur (un caractère) de la case où s’est déplacé l’agent pouvant être un objet A, B ou rien si la case est vide.
* *Void prendre() :* prenant en entrée un agent et permettant d’actualiser la carte selon la décision de l’agent (prendre ou non).
* *PerceptionDeposer()*: prenant un agent en entrée et retournant la valeur (un caractère) de la case où s’est déplacé l’agent pouvant être un objet A, B ou rien si la case est vide.
* *Void depot()*: prenant en entrée un agent et l’objet à poser, permettant à l’agent de déposer l’objet qu’il porte.
* *Agent.java* : Classe d’un agent prenant les paramètres pour le calcul de probabilité (k-mois, k-plus, l’objet porté, et la mémoire). Cette classe va diriger les décisions de l’agent sans que ce dernier ne sache où il se trouve sur la carte, afin de respecter le concept environnement / agent.
  + *Void action()*: Cette fonction va permettre à l’agent de se déplacer sur la carte et savoir s’il doit prendre un objet ou en déposer un selon la probabilité de la décision calculée.

Les fonctions *prendre()* et *déposer()* correspondent aux calcul des valeurs de pPrise et pDepot afin de créer des clusters entre les objets grâce à la mémoire de l’agent et à la fonction *calculerF()*. Pour gérer la mémoire, nous avons décidé d’initialiser une ArrayDeque de caractère correspondant aux objets où l’agent est déjà passé.

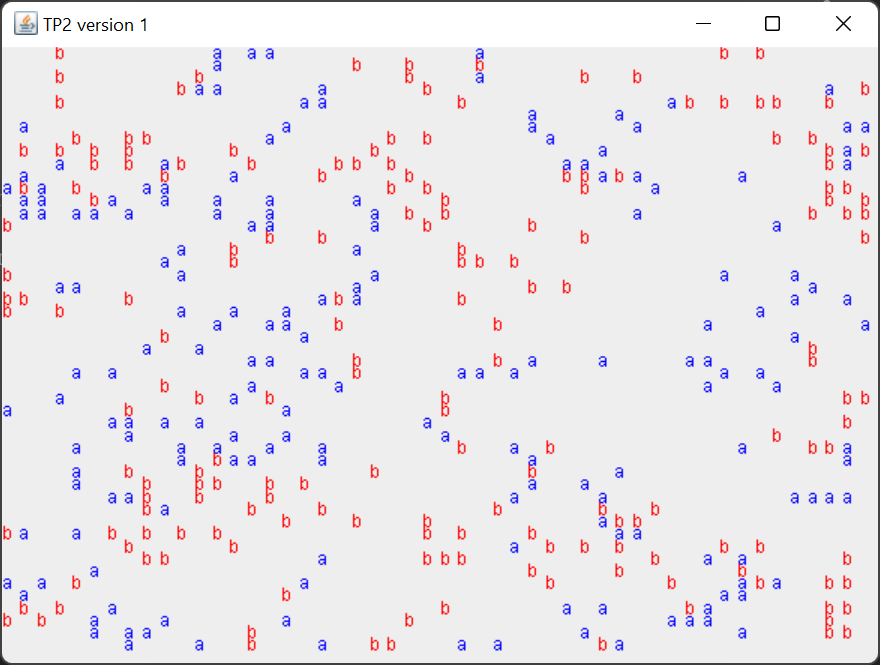
* *Direction.java*: Cette classe sert juste à stocker les coordonnées x et y qu’un agent peut prendre. Nous sommes conscients que nous aurions pu nous passer de cette classe, mais nous souhaitions avoir une classe bien distinctive pour identifier les déplacements.

# Version 1 sans taux d’erreur

Voici notre système avec les valeurs suivantes :

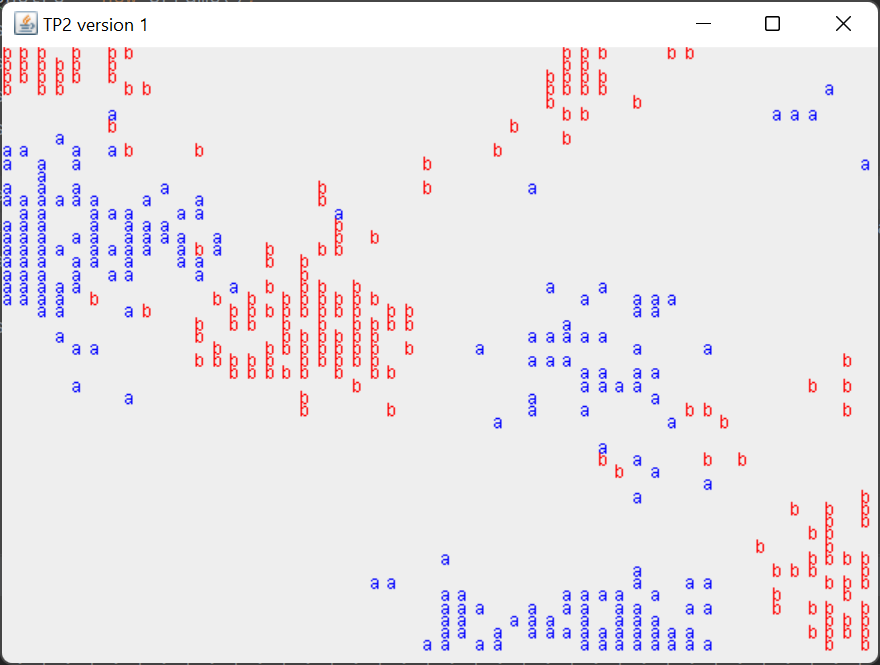
Une grille de 50x50, i=1, 20 agents, k-plus = 0.1 et k-mois = 0.3, 200 objets A et 200 objets B.

Voici le résultat au bout de 1000 itérations :



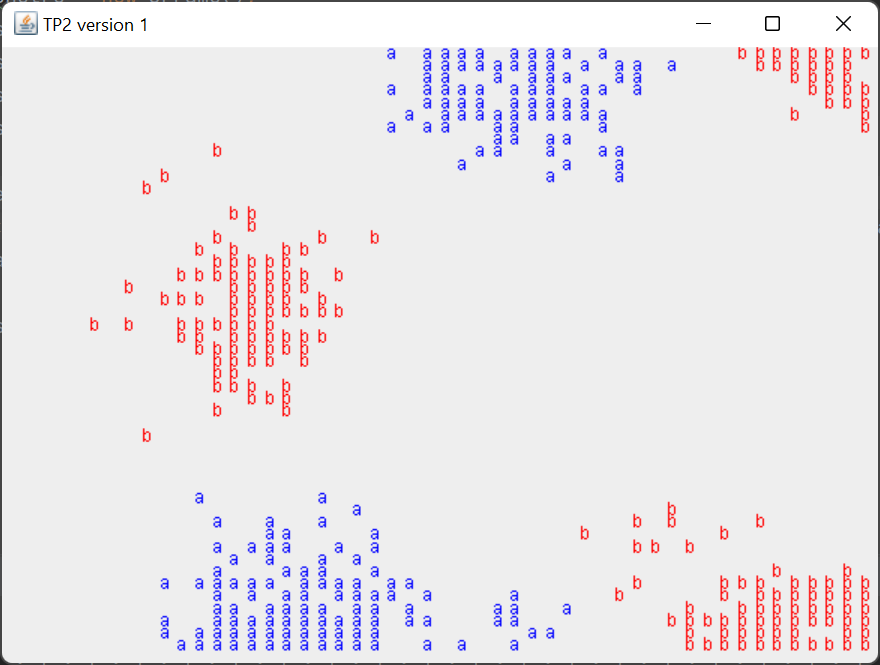
On remarque que les objets A et B sont plutôt dispersé sur la carte et que les agents n’ont pas réussis à former des clusters d’objet, notamment car le nombre d’itération n’est pas assez élevé.

Voici le résultat pour les mêmes valeurs avec 100 000 itérations :



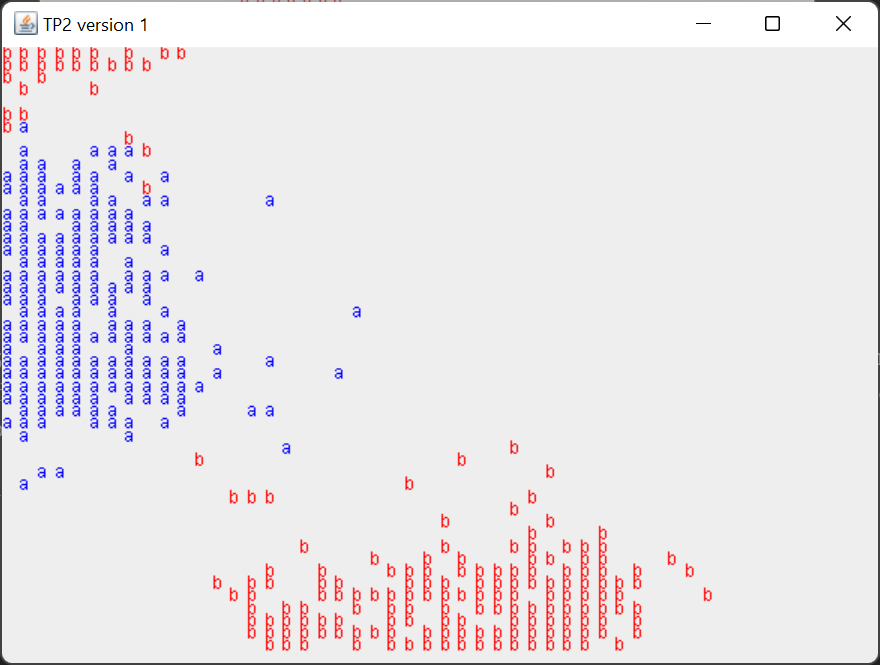
On remarque maintenant que plusieurs clusters d’objet se sont créés mais que certains objets sont « perdus » dans un cluster d’objet différent.

Voici le résultat pour les mêmes valeurs avec 500 000 itérations :



Avec 500 000 itérations, on peut voir que les clusters d’objet sont bien regroupés en objet du même type et bien délimité. Les agents prennent et déposent les objets selon les probabilités calculés ce qui permet de bien séparé les objets.

Voici le résultat pour les mêmes valeurs avec 1 000 000 itérations :



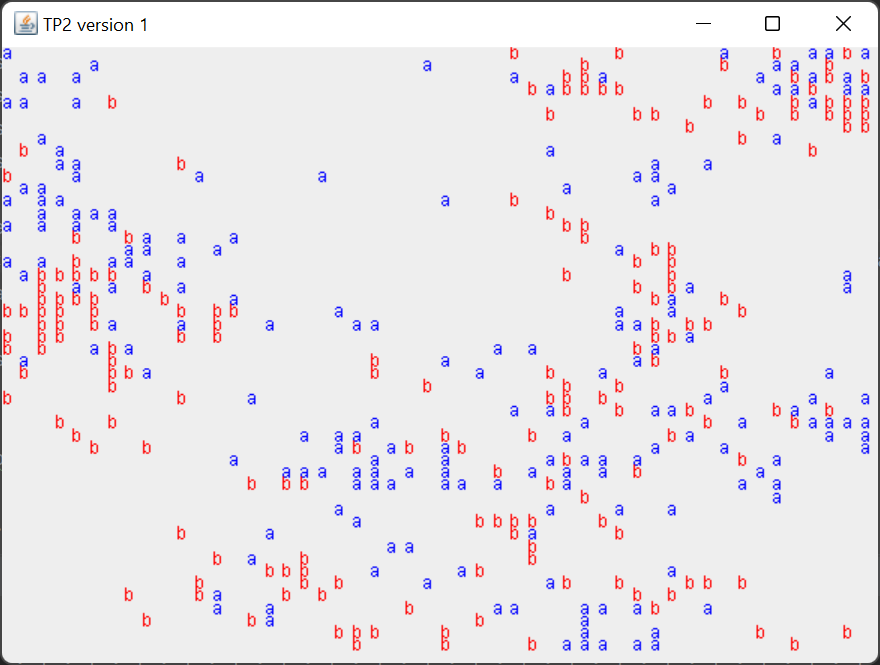
Pour conclure sur cette première partie, la formation de cluster d’objet va dépendre du nombre d’agent et d’objets présents sur la carte. En effet, plus il y aura d’agent, plus la formation de cluster va se faire en peu d’itération pour un même nombre d’objet. De plus, plus le nombre d’itération sera important, plus les agents auront le temps de se déplacer sur la carte et de prendre ou déposer les objets donc former des clusters d’objet.

# Version 1 avec taux d’erreur

Voici notre système avec les valeurs suivantes :

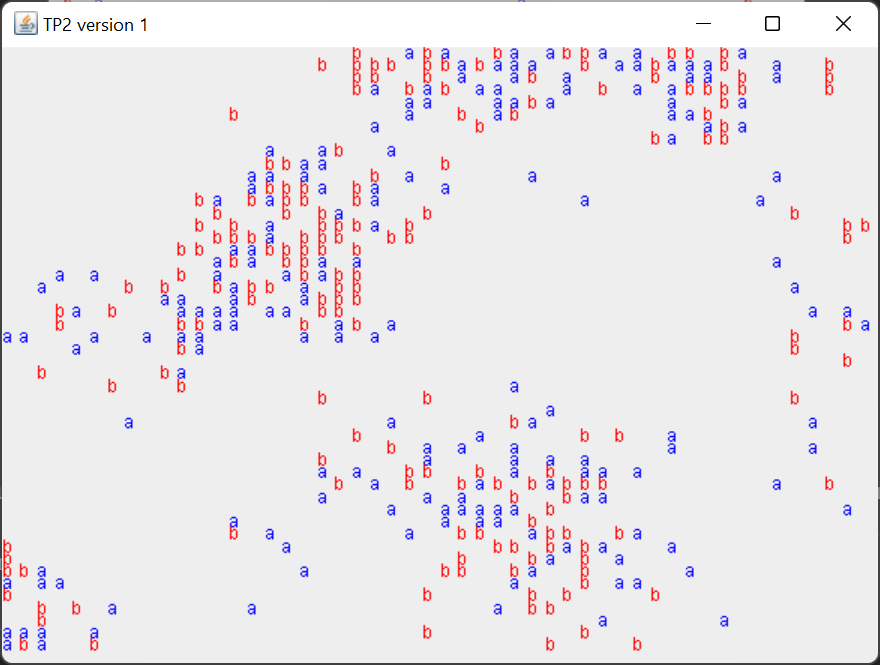
Une grille de 50x50, i=1, 20 agents, k-plus = 0.1 et k-mois = 0.3, 200 objets A et 200 objets B avec un taux d’erreur variable.

Voici le résultat avec un taux d’erreur égale à 0.2 et 30 000 itérations :

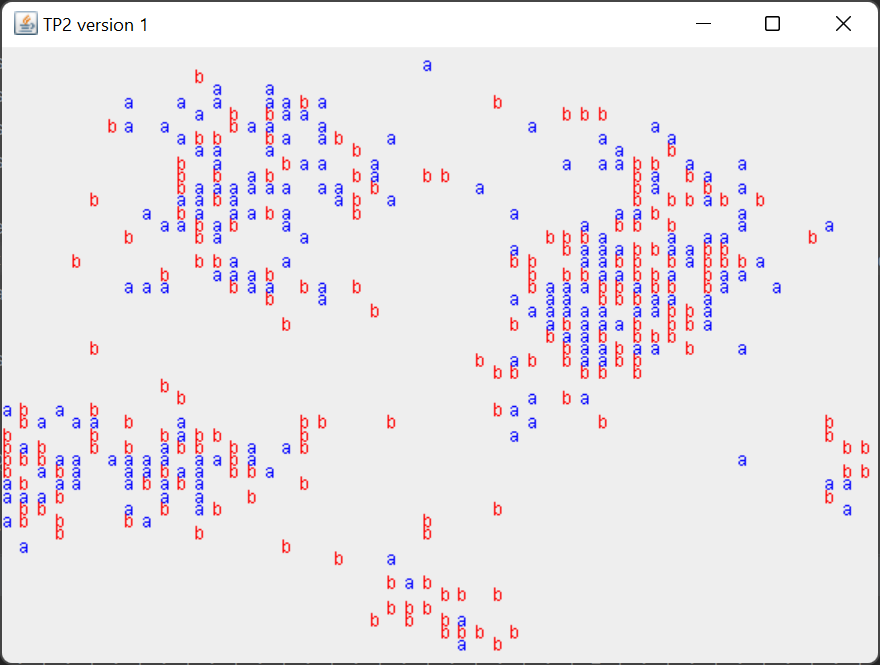


Le résultat obtenu montre que les objets vont être déposer de en tenant compte d’une erreur possible dans la mémoire. Ainsi, on remarque que les agents ont plus de mal à former des clusters d’objets.

Voici le résultat avec un taux d’erreur égale à 0.5 et 30 000 itérations :



Voici le résultat avec un taux d’erreur égale à 1 et 30 000 itérations :



L’introduction du pourcentage d’erreur va entraîner une conséquence importante, cette différence avec la partie ci-dessus est que les agents vont prendre et déposer des objets de manière plus « aléatoire » à cause du taux d’erreur qui trouble le calcul des probabilités de pPrise et pDepôt.

Par exemple, si le taux d’erreur est de 0.2, l’agent va considérer prendre ou déposer le mauvais objet 2 fois sur 10.

# Démonstration

Voici un gif du fonctionnement de notre code. Nous nous excusons pour la qualité de l’image. Cet démonstration s’est faites avec 300 000 itérations, k-plus égale à 0.1 et k-moins égale à 0.3, 500 agents, 250 objets A et 250 objets B avec un mémoire de taille 10.

