Rapport TP Simulation Multi-Agents
Projet de 2ème année d'Architectures Logicielles et Qualité
Filière : Génie logiciel et systèmes informatiques

Réalisation d'une simulation multi-agents

Présenté par : **DESNOS Nathan**
VILLEDIEU DE TORCY Mathieu

Campus des Cézeaux. 1 rue de la Chébarde. TSA 60125. 63178 Aubière CEDEX

Table des matières

1 Objectif

Dans le cadre de ce TP de simulation multi-agent (SMA), nous avons choisi de modéliser la compétition entre 2 groupes d'agents évoluant sur une grille hexagonale. L'objectif de chaque groupe est de s'étendre et de contrôler la plus grande part de la grille, les agents d'un même groupe capturant les cases de la grille en se déplaçant dessus. En plus d'être en compétition pour le contrôle des différentes cases, les agents d'une même équipe peuvent se multiplier et éliminer les agents adverses au travers d'affrontements collaboratifs. De plus les actions passées d'un agent ainsi que des agents alentours influent sur son comportement futur.

Ce compte rendu présente de façon synthétique la manière dont nous avons conçu et implémenté notre projet, les problèmes rencontrés, les résultats statistiques obtenus, ainsi que les changements effectués en cours de route.

2 Patrons utilisés

2.1 Singleton

Nous sommes partis sur l'utilisation d'une classe *Manager* qui sera chargée de gérer les actions des agents à chaque tour. Ce manager, qui contient la liste des agents ainsi que la carte sur laquelle les agents agissent, doit être unique. En effet, il ne faut pas que lors de l'exécution deux instances de cette classe soient instanciées. C'est pourquoi, nous utilisons le patron de *singleton* [1] pour cette classe.

La classe manager contient donc un attribut de classe qui est un pointeur vers le seul et unique objet manager instancié. Le constructeur de cette classe est privé afin que l'utilisateur ne puisse pas instancier lui-même un manager. Il doit passer par la méthode de classe *getInstance()* lui retournant un pointeur sur le manager.

Dans cette méthode *getInstance()*, si l'attribut de classe contient le pointeur *nullptr* alors elle appellera le constructeur pour instancier l'objet et retournera cette instance sur laquelle l'attribut de classe pointera.

2.2 Stratégie

Un patron que nous aurions aimé implémenter avec plus de temps est le patron *Stratégie*. Ce patron permet de définir des algorithmes similaires et interchangeables, tout en les dissociant des classes sur lesquelles ils sont appliqués.

Les agents que nous manipulons font appel à des algorithme identiques dans leur forme, et ne variant que par la façon dont les données sont traitées selon l'équipe à laquelle appartient l'agent. Actuellement, ces algorithmes sont intégrés directement dans le code de la classe *Memoire*, et la bonne méthode est appelée dans une déclaration conditionnelle. Cependant, il aurait été intéressant de les séparer en utilisant la logique de ce patron.

3 Détails du code

3.1 Organisation des fichiers

Le dépôt en ligne [2] est composé de plusieurs fichiers :

— Fichier Doxyfile qui contient la configuration nécessaire à la génération de la documentation par doxygen.
— Le dossier *documentation*
— Le dossier *src*

Le dossier *documentation* contient :

— Le site documentation html généré par doxygen
— Les comptes rendus du TP
— Les présentations des outils utilisés

Le dossier *src* contient :

— Un *makefile*
— Les fichiers des classes *(.cpp et .hpp)*
— Le *main.cpp*
— Le fichier contenant les fonctions du générateur MT *(mt.cpp et mt.hpp)*

Le fichier *main.cxx* contient le point d'entrée du programme permettant de lancer la simulation et de calculer les moyennes et intervalles de confiance de la répétition des simulations. Le programme est généré en utilisant la commande : *make* et le programme peut être lancée avec : *./prog*

Les fichiers *mt.cpp* et *mt.hpp* contiennent les fonctions du générateur aléatoire de Mersenne Twister. [3]

Les fichiers *main_test_agent_memoire.cxx* et *tests_catch.cxx* servent respectivement à lancer les tests et contenir les tests unitaires des principales méthodes sur les classes Agent, Carte et Memoire. Les tests unitaires sont lancés avec la commande : *make test*, puis *./test*

3.2 Carte

Les agents évoluent sur une carte hexagonale. Dans notre implémentation, l'objet *carte* est composé de 2 matrices carrées de taille la dimension de la carte. La première matrice contient la coloration de chaque case de la carte : *Non colorée* par défaut, *rouge* si capturée par l'équipe rouge, et *bleu* si capturée par l'équipe bleue. La seconde matrice est un tableau de pointeurs vers les agents évoluant sur la carte, il est ainsi facile de savoir si une case est occupée ou d'accéder à un agent à partir de sa position dans la carte.

L'image ci-dessous illustre la correspondance entre les cases de la matrice et les cases réelles de la carte. La logique de correspondance est gérée directement dans le code, à l'aide d'une *énumération* donnant la direction d'une case par rapport à une autre. De plus, la carte est un *tore*, ainsi les cases d'une extrémité sont connectées à l'autre extrémité.

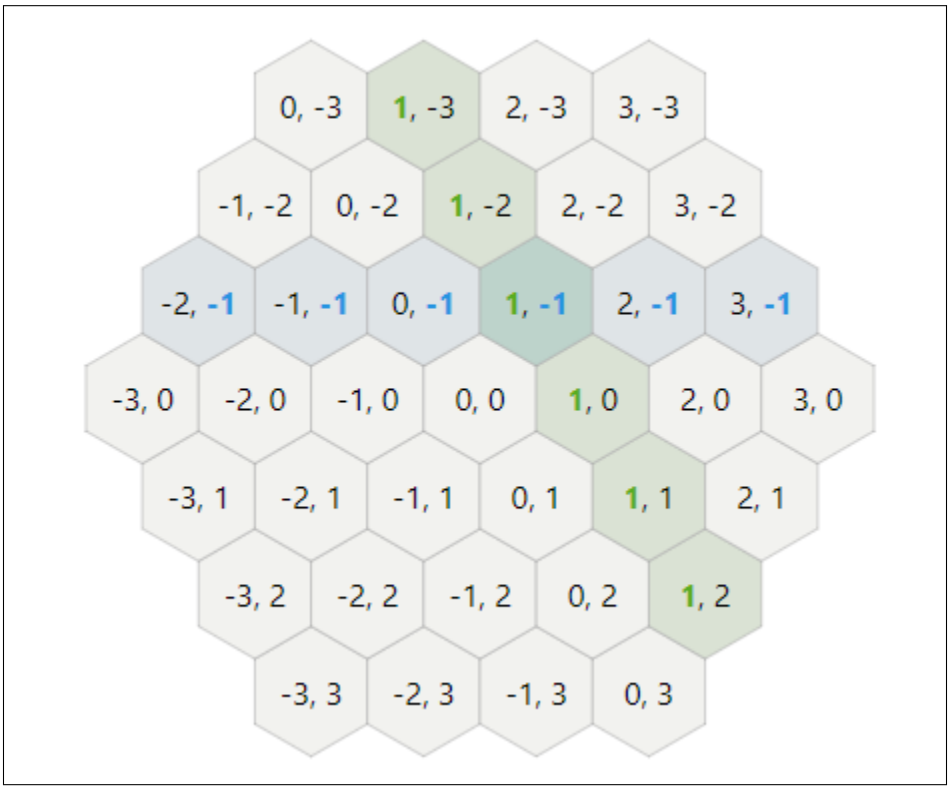

FIGURE 1 – Correspondance matrice / plateau hexagonal

A partir de cette carte, il est possible de récupérer toutes les informations nécessaires à un agent : quelles cases adjacentes sont libres, la présence d'agents amis ou ennemis adjacents, ou encore la couleur des cases. La position de chaque agent est actualisée à chacune de ses actions. Afin de visualiser l'évolution du système sur une itération, il est possible d'afficher une représentation de la carte en ligne de commande.

3.3 Algorithme principal

La simulation fonctionne par tour. Durant chaque tour, tous les agents *agissent*, puis une fois que ceux-ci ont tous agit, ils *communiquent* entre eux pour partager leur dernière action effectuée.

Au début de chaque tour, la liste contenant tous les agents, des deux équipes, est mélangée. (A l'aide de l'algorithme Fisher Yates [4]) Cela permet de ne pas avantager une équipe qui agirait toujours en premier si l'ordre dans la liste des agents ne change pas. Les avantages que certains agents ont en agissant avant les autres sont alors compensés sur plusieurs tours.

C'est la classe manager qui se charge de parcourir cette liste d'agents, de faire agir et communiquer les agents.

Après l'action d'un agent, si celui-ci est mort, il est retiré de la liste des agents pour ne pas impacter la suite du tour.

Lors d'une division d'un agent, un nouvel agent est créé et ajouté dans la liste d'agents. Ce nouvel agent agit directement après être instancié avec un déplacement. Puisqu'il est instancié sur la même case que l'agent qui se divise il doit donc se déplacer sur une case adjacente libre. (Il n'y a pas de division s'il n'y a pas de cases libres sur le voisinage de l'agent)
Pour que l'agent nouvellement créée n'agissent pas une deuxième fois, il est ajouté à la fin de la liste d'agent et le nombre d'agents total n'est actualisé quand début de tour. Ainsi le parcours de la liste s'effectue uniquement sur les anciens agents lors du parcours dans la liste contenant tous les agents.

A chaque fois que l'agent doit agir ou communiquer, un voisinage autour de lui est demandé à la carte qui lui retourne l'état des 6 cases adjacentes. (couleur et agents adjacents) Cela lui permet d'agir, interagir avec son environnement proche.

3.4 Comportement des agents

Le comportement des agents est conditionné par un de leurs attributs complexe appelé « mémoire ». L'agent peut effectuer une des 3 action de base suivante : *se déplacer*, *prendre un niveau*, et *se diviser* en deux agents. La quatrième action possible, *le combat*, et uniquement disponible lorsque l'agent est adjacent à un agent ennemi , auquel cas elle est obligatoire.

La mémoire de l'agent contient son pourcentage de chance d'effectuer l'une des 3 actions de base. Ainsi, lorsqu'un agent agit, il tire un nombre aléatoire entre 0 et 1 et détermine quelle action effectuer en conséquence. Afin d'avoir une bonne génération statistique de nombre aléatoires, nous avons choisi d'utiliser le Mersenne Twister développé par Makoto Matsumoto et Takuji Nishimura. [3]

Une fois que l'agent à agit, sa mémoire est mise à jour en fonction de l'action effectuée et des résultats qui en ont découlé. Par exemple, si l'agent s'est divisé, alors ses chances de se diviser diminuent alors que les deux autres champs augmentent. S'il s'est déplacé mais qu'il a atteint une case qui était déjà de sa couleur, alors ses chances de se déplacer augmentent afin de chercher à capturer une nouvelle case.

De plus, à la fin de chaque tour et une fois que tous les agents ont agis, chaque agent communique à ses voisins direct l'action qu'il vient d'effectuer, ce qui affecte encore une fois la mémoire de chaque agent.

L'action la plus commune est le déplacement, car sans elle les agents ne pourraient pas chercher à atteindre leur objectif. La division a de son côté une chance très faible de se produire, car le nombre d'agents croit exponentiellement. L'action de prise de niveau est un peu plus complexe. Chaque agent dispose d'un niveau maximum, et plus son niveau actuel est proche de son niveau maximum, plus ses chances de prendre un niveau sont basses. Un agent avec un niveau élevé a aussi plus de chances de se diviser. Il est également possible qu'un agent ne puisse pas effectuer l'action choisie, par exemple si l'ensemble des cases adjacentes sont occupées. Dans ce cas, il reste passif, mais sa mémoire évolue tout de même en conséquence.

La dynamique de combat est plutôt simple. Lorsque qu'un agent doit agir et est pris dans un combat, on détermine s'il survit au combat en comparant la somme de son niveau et du niveau des alliés adjacents avec la somme du niveau des ennemis adjacents. Une très légère notion d'aléatoire permet d'éviter des situations bloquantes où les attaquants et les défenseurs ont des sommes de niveaux identiques.

L'utilisation de ces mécanismes permet d'influencer de diverses façons sur le comportement des groupes d'agents en modifiant les méthodes d'évolution de la mémoire. Ainsi certains comportements vont donner lieu à une importante reproduction des agents, alors que dans d'autre cas les agents peuvent devenir des explorateurs frénétiques.

3.5 Différence entre les équipes

Lors de la phase de communication, chaque agent apprend des autres agents adjacents de la même équipe. Pour différencier les deux équipes et dynamiser la simulation, l'apprentissage des agents dépend de leur équipe.

Les *agents bleus* favorise le *déplacement* et ont donc plus tendance à se déplacer.

Tandis que les *agents rouges* ne favorise *aucune action spécifique* lors de l'apprentissage.

4 Répartition du travail

Le travail a été équitablement réparti entre les deux membres du binôme. Mathieu a essentiellement travaillé sur la classe *Manager* chargée de dérouler l'ensemble de la simulation, et sur la mise en place de l'étude statistique. Nathan s'est principalement occupé de l'implémentation de la classe *Carte*.

Les deux membres du binôme ont travaillé sur les classes *Agent* et *Mémoire*. Mathieu s'est essentiellement chargé de la mise en place des méthodes de base et de la logique d'utilisation de ces classe, alors que Nathan s'est concentré sur le développement des stratégies des agents et sur leur apprentissage.

Mathieu a dans un premier temps développé une communication entre les agents prenant en compte la mémoire de chacun pour en faire une moyenne pondérée en fonction de la différence de niveau entre les agents.

Nathan a par la suite développé la communication entre les agents en prenant en compte la différence de niveau et la dernière action effectué par les agents. Il a également implémenté un apprentissage différent lors de la communication entre agent suivant leur équipe. C'est cette dernière version de communication, avec un apprentissage différent suivant l'équipe, qui a été utilisée par la suite dans les simulations.

5 Résultats

5.1 Résultat d'une simulation

Après avoir réalisé une simulation, nous affichons la carte afin d'avoir une meilleure représentation du la simulation. Voici un exemple de résultat après une simulation, de 1000 tours d'action des agents, partant de seulement 1 agent de chaque équipe placée de façon aléatoire sur la carte.

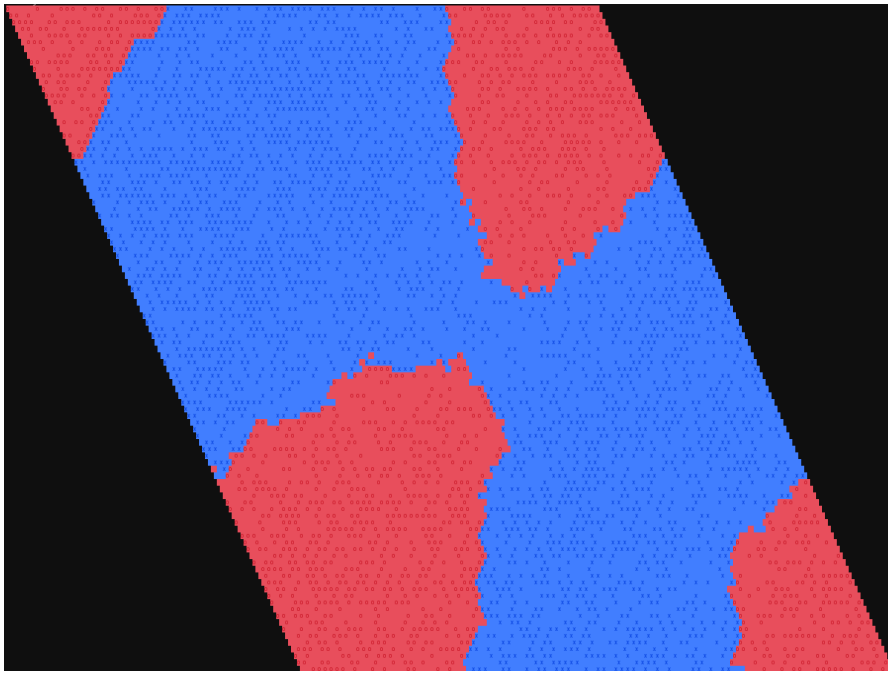

FIGURE 2 – Répartition des agents pour une simulation de 1000 tours

5.2 Résultats sur plusieurs simulations

Nous avons effectué plusieurs simulations afin d'obtenir une tendance du comportement des agents qui s'affrontent. Nous avons effectué des simulations de 1000 tours et relevé le nombre de cases capturées à la fin de chaque simulation.

Pour 120 simulations nous obtenons :

```
Pour 120 simulations
Sur une carte hexagonale de 100 par 100
Ecart moyen du nombre de cases capturees entre les deux equipes : 5964.34


Intervale de confiance a 95% du nombres de cases capturees par equipe :

BLEU :  [ 7940.74 - 221.85; 7940.74 + 221.85 ] => [ 7718.89 ; 8162.59 ]
ROUGE : [ 1976.40 - 181.65; 1976.40 + 181.65 ] => [ 1794.75 ; 2158.05 ]
```

FIGURE 3 – Résultats pour 120 simulations de 1000 tours

On peut voir que l'équipe BLEU semble capturer plus de cases que l'autre équipe. La stratégie définit dans la communication des agents, affectant leur mémoire, semble plus efficace pour se répandre face à l'autre équipe d'agents.

Références

[1] *Patron singleton*. URL : http://www.goprod.bouhours.net/?lang=fr&page=pattern&pat_id=19.

[2] *Gitlab du projet*. URL : https://gitlab.isima.fr/mavilledie4/sma/-/tree/main.

[3] *Mersenne Twister*. URL : http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/MT2002/emt19937ar.html.

[4] *Fisher-Yates*. URL : https://fr.wikipedia.org/wiki/M%C3%A9lange_de_Fisher-Yates.