Rapport Projet SYMA

delhav_a & marty_l

09 Juin 2017

Contents

0.1 Préface

Ce document a été rédigé par deux étudiant en 4ème année à l'EPITA dans le cadre de leurs études. Il regroupe des informations relatives au cours de Système Multi Agents et est destiné à être utilisé à des fins pédagogiques. Le projet a été réalisé à l'aide du framework **Repast Symphony** et vise à observer les interactions d'humains dans un quartier.

Chapter 1

Choix de conception

1.1 Description de l'écosystème

Le principal objectif du projet est d'observer l'évolution de plusieurs agents dans un même écosystème au cours du temps.

Nous avons pris la décision de ne pas étudier un seul écosystème mais d'en générer automatiquement. Pour ce faire le programme prend en entré un fichier texte contenant les informations relatives aux infrastructures de notre écosystème(cf.chapter 4) pour ensuite générer avec une part d'aléatoire les différentes éléments composant l'écosystème. Ainsi il est très facile de faire varier les modèles à étudier et donc d'observer différentes interactions entre les agents.

L'ensemble du quartier est modélisé dans une grille de dimension fixée dans le fichier passé en input et généré procéduralement.

Pour représenter l'activité d'un quartier nous avons repris l'idée du jeu **SimCity**. Le but de ce jeu est de créer et gérer sa ville. Pour cela il faut à la fois avoir une gestion macro mais aussi micro des différentes infrastructures. Nous reprenons ici la même idée pour notre écosystème.
Le quartier est représenté sous formes de zones:

- zone pavillonaire
- zone commerciale
- zone d'activité économique(bureaux)

Ces zones sont composées d'un certains nombre de bâtiments qui ont tous la même fonction (loger, vendre un produit, accueillir des employés). Cependant chacun des bâtiments d'une zones possède des caractéristiques différentes des autres bâtiments de sa zone(capacité d'accueil, nourriture restante,etc). De même que les agents qui se déplacent dans le quartier en plus d'être humain possèdent des attributs qui leurs sont propres(âge, argent, etc).

De plus les zones sont reliées automatiquement entre elles par des routes, permettant le déplacement en voiture.

De plus afin de rythmer la vie de ces agents nous avons choisi de représenter le temps sous la forme d'un calendrier qui débute à partir du début de l'an 2000. L'unité utilisée pour représenter le temps est fixé à 60 ticks par heure. Ainsi chacun des habitants sait à quel moment il doit aller travailler, à quel moment il est en week end, à quel moment il recoit sa paye et à quel moment c'est son anniversaire.

Ainsi en laissant tourner le programme on s'aperçoit que chaque agent participe à la cohérence de l'ensemble de l'écosystème dans lequel il se développe.

1.2 Comportement des humains

Pour rendre la simulation réaliste nous avons décidé d'implémenter des comportements q'uon retrouve dans la réalité aux agents *Human*.

1.2.1 Le travail

La principale activité des humains est le travail. Dans le fichier d'input nous définisson un nombre d'humain par bureaux et un nombre d'humain par maison. Si le nombre d'humain dépasse la capacité d'accueil de l'ensemble des bureaux alors une partie des humains n'auront pas de travail.

Le travail possède une heure d'ouverture et une heure de fermeture. Entre ces deux heures les habitants viennent travailler.

Tous les débuts de semaine un income fixé aléatoirement au début de la simulation est reversé aux employés. Cet inncome est considéré comme leur salaire.

1.2.2 Les courses

Comme tout être humain, les habitants du quariter doivent se nourrir. Heureusement de base leur maison est approvisionné en nourriture. Cependant chaque jour l'ensemble des habitants est contraint de se nourrir et donc de consommer la nourriture de la maison dans laquelle ils habitent. S'il ne reste plus de nourriture il perdront des points de santé. Si sa santé est nulle, l'agent meurt.

Aussi pour éviter cette situation les habitants peuvent aller faire les courses sur leur temps libre(aucune activité en cours). Pour cela ils doivent se rendre dans un magasin qui lui aussi à des horaires d'ouverture afin d'échanger son argent durement gagner contre des vivres.

1.2.3 Les activités

Bien que les agents occuepnt la majeure partie de leurs journée à travailler se nourri et faire les courses, il leur reste néanmoins du temps libre le soir et notamment le week end. Tout ce temps libre peut être ennuyeux. Il a donc fallut implémenter une jauge d'ennui. Lorsque cette jauge d'ennui est remplie, les Humain pratiquent une activité qui va les distraire. Dans les quartiers générés cette activité est le cinéma. Ainsi en échange d'argent les habitants du quartier peuvent aller au cinéma ce qui va les distraires et réduire leur jauge d'ennui.

1.2.4 Quotidien

Au quotidien les humains agissent donc en fonction de leurs besoins et rythme leur semaine entre faire les courses, aller au travail et se distraire au cinéma. De plus au cours du temps ils veillissent. Cependant tous les humains n'ont pas les mêmes revenus et ne sont donc pas égaux. Certain human peuvent même se retrouver sans travail et donc sans revenu.

Chapter 2

Implémentation

2.1 Agents

Pour pouvoir implémenter ce programme, nous avions décidé de considérer tous les éléments qui seraient voués à évoluer lors de la simulation comme des classes qui *extends* la classe abstraite Agent. Nous avons déterminé trois grand types d'agents qui échangent ensemble:

- les habitants
- les bâtiments
- les véhicules

  Le schéma suivant représent l'implémentation des différents *Agents* de notre système.

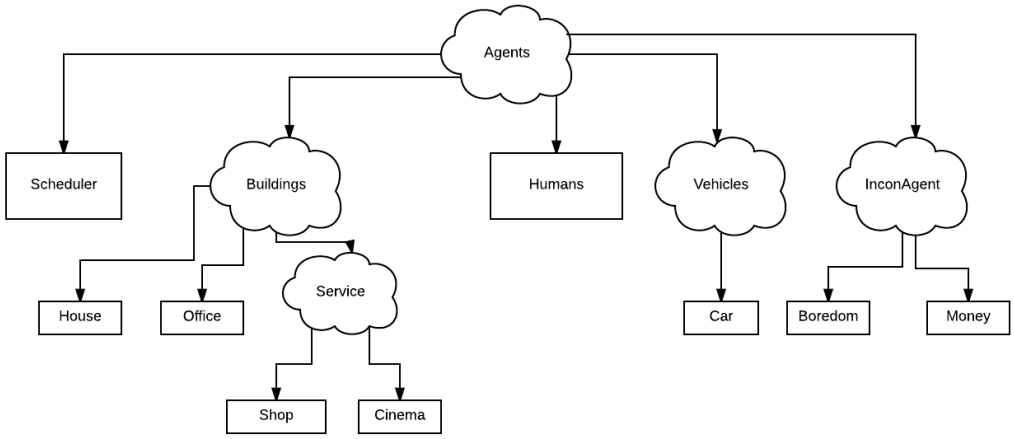

Figure 2.1: Représentation des Agents.

2.2 IconAgents

Nous avons considéré des élémént comme la *Road* et les *IconAgent* comme des agents uniquement pour pouvoir les afficher dans Repast. Les icônes permettent de visualiser respectivement quand un Humain gagne ou perd de l'argent mais aussi quand il s'ennuie ou se distrait.

2.3 Les véhicules

Afin de se déplacer plus rapidements, les humains les plus riches du quartier possèdent une voiture. Ce moyen de transport s'active lorsqu'ils sortent du bâtiment dans lequel ils étaient.

2.4 Déplacements des humains

Les humains utilisent le plus court chemin pour se rendre d'un point A à un point B. Pour se faire nous avons implémenter l'agorithme DJikstra qui prend comme poid le nombre de ticks pour aller d'un point à un autre. Un humain peut bien évidemment utiliser sa voiture pour se déplacer plus rapidement.

Les deux modes de transport disponibles sont: - La voiture qui n'est possible que sur les routes - La marche qui est possible de partour mais est plus lente

2.5 Les bâtiments

Aussi même s'il ne bouge pas, un bâtiment est voué à évoluer au cours de la journée notamment par le nombre de personnes qu'il accueil ou par ses horaires d'ouvertures par exemple.

Les bâtiments de service communiquent avec le Scheduler afin de connaître à quelle heure ils doivent ouvrir et fermer afin de communiquer cette information aux habitant.

Les *House* bien qu'elles n'aient pas besoin d'ouvrir et fermer, possèdent un compteur de nourriture qui permet de nourrir ses habitants.

2.6 Scheduler

On remarquera qu'en plus des trois types cités précédemment on retrouve la classe *Scheduler*. Cette classe sert d'interface pour traduire le *TickCount* dans un sytème plus compréhensible pour l'homme. Elle n'a pas vocation à être affichée mais permet aux autres agent de connaître l'heure et la date actuelle afin de guider leurs actions. C'est lui qui rythme les interaction des Agents et l'évolution de la simulation.

2.7 Communication entre agents

Afin que les agents effectuent des actions nous avons utilisé les deux types de méthodes suivant:

- @ScheduledMethod
- @Watch

La première méthode permet de donner à l'agent une action à réalisé avec l'interval souhaité tandis que la seconde est utilisée pour que l'agent puisse "réagir" à une action réalisé par un autre agent. Avec ces methodes les agents peuvent à la fois évoluer en autonomie mais aussi s'adpater aux autres agents.

Par exmple un humain *Watch* l'office dans lequel il travaille et suit son ouverture et sa fermeture afin de déclencher les méthodes *goHome* ou *goToWork*. Aussi les différents bâtiments de service utilient cette même méthode sur le *Scheduler* pour connaître les heures d'ouverture et de fermeture du service. Ainsi tout le quartier "vit" en adéquation avec l'heure de la journée ce qui créer une cohérence dans l'activité de 'écosystème.

De plus nous avons considérer que chaque humain est relié à une seule maison et un seul office. Celà permet d'empêcher que des inconnus pénêtre dans nôtre maison ou qu'il fasse notre travail à nôtre place(même si l'on aimerait bien).

Pour ce faire nous avons utilisé les *Network* afin de lier chaque humain à une maison(s'il en possède) et un bureau(s'il en possède). Ainsi chacun sait où il doit se rendre lorsqu'une des fonctions de déplacement comme *goToWork* est appelée.

Ci-dessous un exemple de relation entres des agents *Human* , *Office* et *House*:

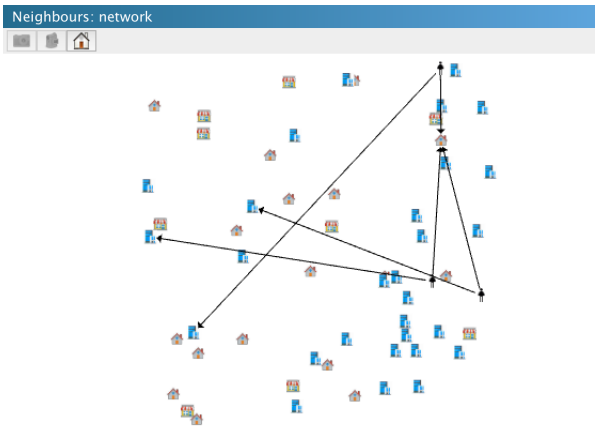

Figure 2.2: Illustration network

2.8 Parser

Comme évoquer prédemment le but de ce projet est de générer procéduralement le système dans lequel vont évoluer les agents. Au lieu de représenter notre quartier sous la forme d'un tableau, l'input prend la forme d'un seul fichier texte composé d'une suite d'instruction comme ci-suit:

```
1 grid: 50, 50
2 zone: house, 5, 22, 1, 3, 1
3 zone: office, 4, 4, 3, 1
4 zone: house, 5, 10, 4, 4, 54
5 zone: shop, 19, 16, 3, 1
6 zone: office, 4, 16, 2, 2
7 zone: office, 17, 4, 4, 2
```

Figure 2.3: Exemple de fichier d'input

Le fichier d'input est très modulable et simple d'utilisation(cf chapter 4). Il suffit d'ajouter une zone pour que les bâtiments et les routes reliant la zone aux autres zones soient généré automatiquement. La génération des routes se fait grâce à l'algorithme Djikstra.

2.9 Singleton

Dans notre projet nous avons fait le choix d'implémenter un singleton nommé *MainContext*. Il a pour but de coordonner l'ensemble du système dans lequel les agents évoluent. Ainsi chaque agent a à sa dispositon les informations relatives à l'écosystème du quartier généré.

Pour remplir cette mission nous avons doté le singleton des informations suivantes:

- le contexte
- la grille et ses dimensions
- les listes des différents *BuildingZone*

Ainsi il est possible de modifier les informations relatives aux Agents depuis n'importe où dans le programme.

Le *MainContext* a aussi le rôle de créer et d'initialiser les attributs des différents Agents. Pour proposer de la diversité entre les agents nous utilison la classe *RandomHelper* afin de généré "aléatoirement" les caractéristique de chacun des éléments.

Chapter 3

Difficultés rencontrées

Une difficulté rencontrée par le groupe au niveau de la conception a été l'emploi du temps. En effet le groupe s'est formé dans une période où les rendus dans d'autres matières sont courants. Il a donc fallut gérer l'emploi du temps afin de faire au mieux pour chacune des tâches qu'il nous avait été attribuées dans l'ensemble des matières. Pour surmonter celà nous avons profiter des outils comme *Git* et *Slack* afin de communiquer sur l'avancement du projet.

Au niveau du développement une des majeure complication sur le projet a été l'utilisation des *Watchers*. En effet il s'est avéré qu'à plusieurs reprises les fonctions disposant d'un @Watch n'était pas déclenchées comme nous le souhaitions. Aussi malgré une documentation assez claire il existe peu de topics sur le sujet pemettant de confronter des problèmes ou discuter d'exemples d'utilisation. Finalement une des erreur provenait du fait que la fonction à laquelle était rattaché le @Watch était *private*. Bien que l'erreur soit plutôt basique, aucun log ne permettait d'identifier sa provenance, en fait aucune erreur n'était déclenchée.

Ce qui amène à l'une des plus grande difficultée, la plateforme **Repast Symphony** en elle-même. Bien que l'outil offre des possibilités modulables pour représenter son modèle et possède une interface assez intuitive pour l'utilisateur il existe trop de problème qui sont gênant:

- L'outil n'est pas assez userfriendly: pafois lorsqu'une action est effectuée aucun retour n'est affiché à l'écran. On ne sait pas si l'action s'est bien déroulé ou si un élément quelconque a entrainé un *Fail* de l'action. De plus certains boutons ont des icônes très peu explicites et ce n'est pas la documentation pourrait aider à expliquer leur utilisation.
- La documentation est le deuxième point négatif. En effet il est parfois dur d'obtenir une information rapidement grâce à internet. La documentation des fonctionnalités est illustrée par des exemples qui mettent en scène des situations et qui énumère des suites d'instructions sans expliquer les fonctionnalités de l'outil. La documentation de l'API est très basique et peu détaillée. De plus les posts sur les forums sont peu nombreux et en général mal présentés.
- Il subsiste quelques bugs mineurs

Néanmoins ces difficultés s'apparentent plus à une gêne. Bien que des améliorations pourraient être apportées, l'outil permet d'avoir des résultats assez satisfaisant.

Chapter 4

Usage

Pour tester le programme il suffit de modifier le fichier nommé **test.in** situé à la racine deu projet.

Le quartier est généré procéduralement en suivant les normes suivantes:

- définir dimensions du quartier:
  Usage : *grid: X, Y*
- définir les différentes zones:
  - house:
    Usage : *zone: house, Xorigin, Yorigin, sideLenght, capacity, humansNumber*
  - office, shop, movie:
    Usage : *zone: zoneType, Xorigin, Yorigin, sideLenght, capacity*

Les différentes instructions sont indépendantes et peuvent être déclarées dans l'odre souhaité par l'utilisateur. Malheureusement, tous les cas d'erreur ne sont pas traités nous vous conseillons donc d'utiliser le programme avec des exemples qui visent à ce que la simulation fonctionne.