

2015-2016

3ème année de Licence Informatique

Pathfinding multi-agents à l'aide de méthodes stochastiques

Nicolas Roux

Théo Voillemin

Corentin Talarmain

Sous la direction de M.

Stéphan Igor

Soutenu publiquement le :
6 Juin 2016



L'auteur du présent document vous autorise à le partager, reproduire, distribuer et communiquer selon les conditions suivantes :



- Vous devez le citer en l'attribuant de la manière indiquée par l'auteur (mais pas d'une manière qui suggérerait qu'il approuve votre utilisation de l'œuvre).
- Vous n'avez pas le droit d'utiliser ce document à des fins commerciales.
- Vous n'avez pas le droit de le modifier, de le transformer ou de l'adapter.

Consulter la licence creative commons complète en français :
<http://creativecommons.org/licences/by-nc-nd/2.0/fr/>

Ces conditions d'utilisation (attribution, pas d'utilisation commerciale, pas de modification) sont symbolisées par les icônes positionnées en pied de page.



REMERCIEMENTS

Avant toute chose, nous souhaitons remercier notre responsable de stage Monsieur Igor Stéphan pour son écoute et sa disponibilité tout au long de notre stage et d'avoir accepté notre demande d'étude sur la recherche de chemin en informatique.

Nous souhaitons remercier aussi le département informatique pour l'accès aux salles informatiques et l'équipe technique pour l'installation des bibliothèques.

Sommaire

INTRODUCTION

1 Présentation du sujet de stage

- 1.1. La recherche de chemin
- 1.2. Les méthodes stochastiques

2 Analyse du problème

- 2.1. État de l'art
- 2.2. Contraintes
 - 2.2.1. Temps réel
 - 2.2.2. API
 - 2.2.3. Solution générale
- 2.3. Choix possibles
 - 2.3.1. Algorithme de Monte Carlo
 - 2.3.2. Algorithme génétique

3 Planning du travail effectué

4 Présentation des solutions adoptées

- 4.1. Choix effectués
 - 4.1.1. Algorithme génétiques
 - 4.1.2. Controller (communication)
- 4.2. Difficultés rencontrés
 - 4.2.1. Conception de la base de représentation d'un problème de Pathfinding
 - 4.2.2. Conception d'un algorithme génétique
 - 4.2.3. Gestion du temps réel
 - 4.2.4. Création d'une API
- 4.3. Outils utilisés
 - 4.3.1. Librairies
 - 4.3.2. Environnement de développement
 - 4.3.3. Débugage
 - 4.3.4. Travail collaboratif

5 Tests effectués

- 5.1.1. Protocole
- 5.1.2. Carte 1
- 5.1.3. Carte 2
- 5.1.4. Carte 3

CONCLUSION

BIBLIOGRAPHIE

ANNEXES

Résultats des tests

Introduction

Dans le cadre de notre licence Informatique à l'université d'Angers, nous devons réaliser un stage de deux mois pour valider notre diplôme. Nous sommes tous les trois intéressés par une orientation vers la recherche dans la suite de nos études. C'est pour cette raison que nous avons souhaité profiter de cette occasion pour explorer un domaine qui nous intéressait tous, la recherche de chemin dans la théorie des graphes. En lien avec un autre stage d'image de synthèse, nous avons donc exploré ce domaine au travers des méthodes stochastiques.

Le but de ce stage est de fournir une interface de programmation applicative (API) utilisable par des développeurs de jeux de stratégies en temps réel (RTS) pour appliquer un algorithme de recherche de chemins pour leur jeu. Cette API doit s'adapter à un spectre large de demandes en matière de RTS, et doit présenter une utilisation simple.

1 Présentation du sujet de stage

Notre sujet de stage, la réalisation d'un pathfinding à l'aide de méthodes stochastiques, se décompose en deux sous-parties : qu'est-ce que la recherche de chemin, et qu'est-ce qu'une méthode stochastique.

1.1. La recherche de chemin

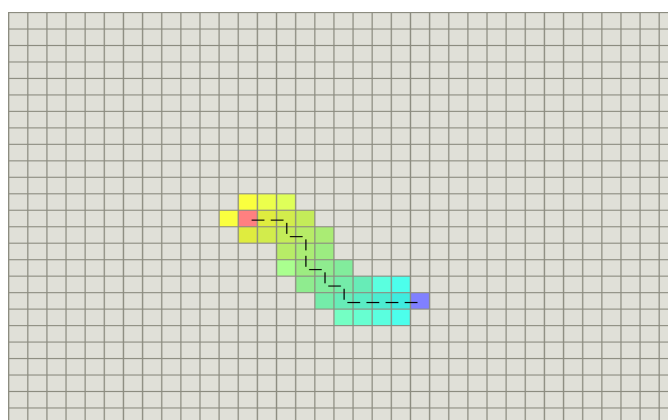
C'est à l'intelligence artificielle et plus particulièrement au domaine de la recherche de solution qu'est rattachée la recherche de chemin. De manière basique, on peut élargir ce problème à la recherche du meilleur chemin entre deux nœuds d'un graphe. C'est en rajoutant des contraintes sur le graphe tels que des obstacles ou de la recherche multiple avec gestion de conflits que le problème se complexifie. Par ailleurs, la contrainte du temps réel impose de renvoyer des débuts de chemins immédiatement, que l'on ait déjà trouvé un chemin arrivant au point voulu ou pas.

1.2. Les méthodes stochastiques

Pour la résolution des problèmes concernant la recherche de chemin, l'algorithme de Dijkstra ou l'algorithme A* sont très souvent utilisés, car répondant déjà au problème de manière efficace, particulièrement A* qui présente un bon compromis entre coût de calcul et optimalité de la solution.

Cependant, nous devons répondre à une exigence du temps réel pour notre solution, exigence à laquelle ces deux algorithmes ne répondent pas, c'est pour cela qu'il était nécessaire de mettre en place un algorithme stochastique.

Un algorithme dit stochastique suit une méthode de recherche aléatoire permettant de converger vers une solution au problème. Cette méthode sera dans de très rares cas aussi efficace qu'un algorithme A*, mais sa particularité est qu'elle pourra renvoyer la meilleure solution trouvée à tout moment, même en étant interrompue.



*Illustration 1: Exemple de pathfinding avec A**

2 Analyse du problème

La principale problématique de notre stage était de pouvoir répondre au problème de recherche de chemin d'un jeu de stratégie en temps réel. Ce problème implique certaines contraintes qu'il nous a fallu respecter tout au long du stage.

2.1. État de l'art

Nos nombreuses recherches sur le sujet ont débouché sur de nombreux résultats sur le pathfinding d'une part, et sur les méthodes stochastiques de l'autre. Il reste assez rare de trouver des applications des méthodes stochastiques sur le problème du pathfinding. La majorité de ces applications concerne l'algorithme des fourmis et l'algorithme de Monte-Carlo. Sur l'algorithme de Monte-Carlo, on peut citer le papier [MCPPTSG] et sur l'algorithme des fourmis, on peut citer le papier [AADO]. L'algorithme génétique est la méthode stochastique qui nous paraît être la moins appliquée sur le problème du pathfinding suite à nos recherches. Elle nous a semblé de base plus originale et attrayante que les autres solutions, malgré sa faible utilisation dans le domaine du pathfinding multi-agents (que ce soit en temps réel ou non). Ainsi, la plupart des papiers que nous avons trouvés nous ont permis de comprendre le principe fondamental de l'algorithme génétique, mais ne traite que sur des mono-agents. De ces papiers on peut citer [RTPGA], [DGASSPP] ou encore [GAGP]. La dernière utilisation d'un algorithme génétique dans l'utilisation du pathfinding d'un jeu vidéo semble être le jeu Black & White qui date de 2001. Nous n'avons trouvé aussi que très peu de benchmarks, que ce soit sur l'algorithme génétique comme sur le pathfinding en général. Le seul trouvé et comparable n'expose que des résultats de longueur de chemin optimale, ce qui nous semble être intéressant mais trop peu explicite de la qualité d'un algorithme de pathfinding temps réel. Nous avons donc décidé de développer, suite à ce stage, notre propre site web qui exposera dans un premier temps, notre propre algorithme, puis présentera un benchmark forgé par nos soins qui partagera les maps utilisées ainsi que le temps pour trouver un chemin en temps réel et la qualité du chemin.

2.2. Contraintes

Le sujet de notre stage nous a imposé certaines contraintes qu'il nous a fallu respecter, de la modélisation au développement de notre projet.

2.2.1. Temps réel

La contrainte du temps réel a été majeure dans notre projet, justifiant majoritairement le choix de l'utilisation de méthodes stochastiques. En effet, la particularité des jeux de stratégie en temps réel est que le joueur doit pouvoir arrêter, modifier et ajouter une demande de déplacement à tout moment. Ceci

demande la mise en place d'un algorithme interruptible, qui serait en mesure de renvoyer une solution quelque soit l'avancée du calcul.

De plus, le temps réel implique une demande d'efficacité de notre algorithme quant au temps d'exécution. Le plus important pour les développeurs du jeu en question, est de savoir quelle est la prochaine case d'un personnage à intervalles réguliers courts. Il était donc nécessaire pour nous de développer une solution capable de renvoyer de manière régulière une solution pour chaque personnage du jeu.

2.2.2. API

La finalité de notre stage était de fournir un outil utilisable par les développeurs. C'est en ce sens qu'il était nécessaire que nous rendions notre projet sous forme d'API fonctionnelle. Cette contrainte a orienté la conception de notre projet, notamment sur les moyens que nous devons fournir pour communiquer avec notre algorithme. La construction d'une API fonctionnelle est une contrainte régulière en informatique, que nous rencontrerons sans doute à nouveau, de ce fait il a été très formateur pour nous de s'y adapter.

2.2.3. Solution générale

Pour que l'API ne soit pas restrictive, nous avons eu comme contrainte de fournir un outil adaptable aux souhaits des développeurs. En ce sens, nous avons eu à impliquer cette contrainte dans nos réflexions de conception, modélisation et développement.

2.3. Choix possibles

Nos recherches sur les méthodes stochastiques ont mis en exergue quelques algorithmes existants qui ont pu faire leur preuve sur différentes applications.

2.3.1. Algorithme de Monte Carlo

Le premier algorithme et le plus connu est l'algorithme de Monte-Carlo. Cet algorithme a deux caractéristiques : Il utilise de l'aléatoire au cours de son calcul mais s'exécute avec un temps déterministe. Cet algorithme était à considérer car son temps d'exécution déterministe permettait d'accéder à la contrainte du temps réel. En effet, il aurait été alors possible de renvoyer un résultat à tous les intervalles voulus par les développeurs. Le défaut de cet algorithme est que le résultat renvoyé peut être incorrect avec une certaine probabilité.

2.3.2. Algorithme génétique

Moins connu mais sûrement plus intéressant, le second algorithme est l'algorithme génétique. Basé sur les travaux de sélection naturelle et d'évolution de Charles Darwin, tel Monte-Carlo, il génère des solutions principalement grâce à l'aléatoire.

Ainsi, l'algorithme possède une population où chaque individu propose une solution de chemins au problème de pathfinding, ceux-ci sont par la suite modifiés à l'aide de mutations où s'y effectuent

diverses insertions, modifications ou suppressions de bouts de chemin basé sur l'aléatoire. Ces individus sont ainsi évalués grâce à une fonction particulière qui, après exécution, attribue une fitness à chaque membres de la population, valeur qui reflète la qualité des chemins proposés.

Cette fitness permet des modifications de plus grande envergure, notamment sur la population entière où l'on supprime les agents les moins bons ou alors où l'on croise les différents individus pour créer de nouveaux membres ayant des chemins générés par combinaisons de ses individus « parents ».

Cet algorithme permet de retourner à n'importe quel moment une solution ou une ébauche de solution, permettant ainsi, par exemple, un déplacement en temps réel des différents agents du problème de pathfinding. Son principal défaut vient du calcul de la fitness qui peut être différent selon les conditions de la carte. Une carte ouverte et une carte en labyrinthe ayant des évaluations de chemins sensiblement différentes.

3 Planning du travail effectué

	Théo	Nicolas	Corentin	N° Semaine
Conception	X	X	X	1
Développement carte				1,2
• Création des classes	X	X	X	
Développement Communication				1,2
• Règles (xml)			X	
• Cartes (txt)	X			
• Classe (Controller)		X		
Développement Dijkstra				2,3
• Récuratif		X		
• Non Récuratif	X		X	
• Débugage & amélioration	X	X	X	
Mono-Agent génétique				3,4
• Évaluation	X	X	X	
• Crossover		X		
• Élimination			X	
• Mutations	X			
• Initialisation		X		
• Dijkstra → A*	X			
• Débugage & amélioration	X	X	X	
Multi-Agent génétique				5,6,7,8
• Évaluation	X			
• Évaluation Sous Minion	X			
• Zones	X			
• Crossover		X		
• Superman		X		
• Mutations		X		
• Débugage & amélioration	X	X	X	
Développement interface graphique(Qt)			X	7,8
Rédaction du rapport de stage	X	X	X	7,8

Tableau 1 : Planning du travail effectué

4 Présentation des solutions adoptées

Après réflexion autour de toutes les possibilités accessibles et de toutes les contraintes, nous avons mis en place les solutions les plus cohérentes selon nous.

4.1. Choix effectués

4.1.1. Algorithme génétique

Nous avons décidé, pour développer un pathfinding par méthode stochastique, de développer un algorithme génétique. Celui-ci est programmé dans une classe « Algogen » qui contiendra et gèrera les principes fondamentaux de l'algorithme génétique.

Les chemins, ou génomes pour notre algorithme, sont stockés sous la forme d'une suite de déplacements (haut, bas, gauche, droite, ou « rien »). Cette représentation engendre une évaluation coûteuse, car nous sommes obligés de parcourir tout le génome pour déterminer quelle est la case d'arrivée. Toutefois nous pensons que c'est un choix pertinent pour l'algorithme génétique, surtout en temps réel (voir Évaluation et fitness).

L'autre possibilité était de représenter les chemins par une suite de cases et non de déplacements. Ceci aurait grandement accéléré l'évaluation, mais nous aurions alors dû effectuer des réparations (extrêmement coûteuses) lors des crossovers (ou « reproductions »). En considérant une application temps-réel, cette méthode implique aussi de « jeter », après le déplacement d'une unité sur le chemin, tous les chemins potentiels qui ne passaient pas par cette case.

a) Demande de chemin et Initialisation

Dans un premier lieu, étudions la population. Elle contient un certain nombre de génomes, chacun définissant le déplacement de chaque agent. C'est un groupement de génomes, chacun donnant le déplacement d'un agent. Ce groupement, nous l'avons appelé « SurMinion », contenant des objets « Minion » ayant chacun un génome. Voici un exemple de génome d'un « SurMinion » :

Agent 1 / Minion 1							Agent 2 / Minion 2					Agent 3 / Minion 3					
01	11	00	10	00	11	01	11	00	10	01	10	00	00	11	11	01	10

Tableau 2 : Exemple d'un SurMinion

Cette population est la base de l'algorithme et toute base doit être instanciée. Dans notre cas, nous créons la population lors de l'ajout du tout premier agent du pathfinding, puis, pour chaque nouvel ajout, nous ne faisons que rajouter un Minion à chaque membre de la population. De la même manière, lorsqu'un agent a atteint son chemin, nous supprimons le Minion correspondant dans tous les SurMinions.

Lorsqu'une demande de chemin est effectuée, on rajoute dans 3 SurMinions des débuts de chemins courts (6 déplacements) calculés via l'algorithme A*, et dans les autres des chemins aléatoires (aussi 6 déplacements).

Nous avons ici aussi apporté une petite optimisation : si un autre déplacement a déjà été demandé vers la même case, que l'unité de cet autre déplacement n'est pas encore arrivée au bout de son chemin, et que la case d'origine du nouveau déplacement est située dans une petite zone que nous calculons autour de la case d'origine de l'autre déplacement, alors le nouveau déplacement n'en sera pas un à part entière mais un SousMinion.

C'est une autre classe que nous avons écrite, et son comportement est assez simple : elle représente un « suiveur » du déplacement original. Elle va commencer par rejoindre la case de départ du déplacement original (via un A*, garanti non coûteux car notre calcul des « zones » les construit de manière garantie sans obstacle direct), puis à chaque fois que le déplacement original va effectuer un mouvement, ce mouvement va se greffer à tous les « suiveurs ». Ces mouvements hérités d'un chemin sont temporairement stockés pour éviter qu'un SousMinion n'hérite que d'une partie du chemin original si le Minion leader venait à être supprimé précocement après avoir atteint son objectif.

Les SousMinions ne sont pas mutés ni supprimés, ils ne servent qu'à économiser quelques calculs de chemins. Ce concept a été pensé pour répondre à une demande de notre groupe partenaire, qui voulait pouvoir effectuer efficacement le déplacement d'un grand nombre d'unités groupées vers une même case.

b) Mutation

Un nombre de SurMinions (ratio variable) est sélectionné, et chacun de ces SurMinions est muté. La mutation peut prendre deux formes selon le rang du SurMinion (rang déterminé en fonction de sa fitness globale, somme de la fitness de chacun des minions qui le composent).

Si le SurMinion est un élite (dans les 20 % meilleurs), il ne fait que recevoir de nouveaux chromosomes à la fin de son génome. Sinon, il subit quelques suppressions de chromosomes, d'autre sont modifiés, et enfin d'autres sont ajoutés en fin de parcours.




Agent 1							Agent 2					Agent 3					
01	11	00	10	00	11	01	11	00	10	01	10	00	00	11	11	01	10
																	
Agent 1							Agent 2					Agent 3					
00	11	10	00	00	01	01	01	00	00	00	11	01	01	01	10	11	11

Tableau 1: Rouge : Modifications | Vert : Ajouts

c) Élitisme

Nous avons développé la notion d'élitisme, souvent utilisée dans les algorithmes génétiques. Dans notre cas, l'élitisme se décline en 3 parties :

- Une partie de la population, considérée « élite », ne subit pas de mutations ni de suppressions de chromosomes d'une génération à l'autre, mais seulement des ajouts, à la fin de son génome. De manière plus classique, cette « élite » va aussi systématiquement survivre d'une génération à l'autre, contrairement au reste des SurMinions qui ont une chance d'être supprimés.

- Pour garantir qu'une solution est toujours disponible, nous prenons à chaque génération le meilleur (selon la fonction d'évaluation) SurMinion, et lui attribuons le statut de « président ». Le président n'est ni altéré ni supprimé d'une génération à l'autre. Ceci permet ainsi, une fois une solution globale trouvée, d'être assuré de pouvoir la renvoyer en l'état sans risquer des modifications destructrices entre temps.

- Les SurMinions évoluent indépendamment les uns des autres, il est donc extrêmement improbable qu'un SurMinion regroupe chacun des meilleurs chemins trouvés pour chaque demande. Or, c'est précisément de ceci dont nous avons besoin pour renvoyer une solution. Pour répondre à ce problème, nous créons à chaque génération un SurMinion (« Superman ») qui regroupe les meilleurs chemins trouvés pour chaque demande. C'est un nouvel individu à part entière et non juste un statut de SurMinion (par opposition à « président »).

d) Reproduction

A chaque génération, en plus de la création du « Superman », il y a une phase de crossover dans laquelle 3 parents sont sélectionnés avec une probabilité exponentiellement décroissante dans l'ordre de fitness (la formule est $1/(2^{\text{rang}})$). Cette méthode de sélection de parents par rang et exponentielle ([Bickel & Thiele, "Selection Schemes in GA"]) ainsi que leur nombre de 3 ([Eiben, Raué & Ruttkay, "GA with multi-parent recombination"]) semble offrir un bon compromis entre exploration et intensification.

Le nombre d'enfants à générer par trio de parents est peu documenté, nous avons donc effectuée quelques tests et les résultats sont non significatifs. Instinctivement, nous pensons qu'un nombre faible va favoriser l'exploration (car dans ce cas, plus de trios de parents seront sélectionnés par génération) et un nombre plus élevé va favoriser l'intensification (car effet inverse).

Lorsque 3 parents ont été sélectionnés, nous procédons à un crossover selon une méthode hybride uniforme / single-point (fortement uniforme quand même). Les crossovers ne sont effectués qu'entre les Minions représentant le même chemin. Notre choix de représentation des chemins implique que les parents ont quasi-systématiquement des génomes de tailles différentes. Les enfants auront un génome de taille égale à celui du parent le plus long. Pour un chromosome à l'indice i , nous sélectionnons d'abord parmi les 3 parents ceux (ou celui) qui a encore un chromosome à cet indice. Un choix aléatoire est alors effectué entre les parents sélectionnés pour déterminer lequel fournira ce chromosome,

conformément à la méthode uniforme. La fin du génome des enfants sera donc une copie de cette même fin dans le génome du parent le plus long, ce qui peut être apparenté à du single-point.


Parent 1														
Agent 1					Agent 2					Agent 3				
11	00	10	01		11	11	11	10	10	11	11	11	00	00
Parent 2														
Agent 1					Agent 2					Agent 3				
11	00	00	11	00	11	01	00	00	10	10	01	01	00	11
Parent 3														
Agent 1					Agent 2					Agent 3				
00	11	10	00	00	11	00	00	00	11	01	00	01	10	11
														
Enfant														
Agent 1					Agent 2					Agent 3				
00	00	10	00	00	11	00	11	00	10	11	01	01	00	10

Tableau 2: Crossover (Reproduction)

e) Évaluation et fitness

La fonction d'évaluation calcule un nombre, la fitness, pour chaque Minion de chaque SurMinion. Elle correspond à la distance de Manhattan¹ [voir fitness, en cours de modification]. La fitness d'un SurMinion est la somme des fitness de ses Minions.

Pour déterminer la fitness d'un chemin, au vu de notre représentation des chemins (par déplacement et non par cases), nous devons parcourir tout le génome pour savoir quelle est la dernière case du chemin. Nous avons choisi de profiter de ce parcours pour effectuer des corrections sur le génome :

- Si une des cases parcourues a déjà été parcourue dans le même chemin, c'est une boucle, et nous supprimons cette boucle. Il est possible qu'une partie des déplacements de la boucle soit utile, mais déterminer laquelle serait encore très coûteux, et avec les ajouts de déplacements aléatoires il se trouve

1 Si (x_1, y_1) et (x_2, y_2) sont les coordonnées respectives de la dernière case du chemin et de la case cible, alors la distance de Manhattan est égale à $|x_2 - x_1| + |y_2 - y_1|$. C'est un calcul rapide et efficace sur les cartes ouvertes.

que des boucles sont formées à chaque itération et pour chaque minion. Nous avons donc pris le parti de simplement les supprimer.

- Si un déplacement amène sur une case obstacle ou en dehors de la carte, nous supprimons ce déplacement.

- Considérant un SurMinion comme étant une solution à tous les chemins demandés, notre gestion des conflits se réalise de façon interne à chaque SurMinion. Si le chemin d'un Minion l'amène à être sur la même case et au même instant qu'une autre Minion (du même SurMinion) déjà évalué, alors nous ajoutons un déplacement neutre (un arrêt) dans le génome du juste avant ce conflit.

De fait, la fonction d'évaluation sert à évaluer et aussi à corriger les chemins, avec un surcoût de temps de calcul faible par rapport à l'utilisation d'une autre méthode/fonction pour faire les corrections.

f) Suppressions

Une fois par génération, nous supprimons un certain nombre de SurMinions non-élites. Ici aussi, ce nombre est à affiner car un grand nombre de suppressions favorise l'intensification en permettant plus de crossovers entre les minions existants, tandis qu'un petit nombre de suppressions favorise l'exploration en laissant évoluer la population existante. Un ratio de suppressions de SurMinions aux alentours de 10 % semble produire les meilleurs résultats pour notre problème et avec notre algorithme.

g) Renvoi d'une solution et déplacement d'une unité.

Après une itération, nous possédons un chemin – qui ne va pas nécessairement à la cible – pour chaque déplacement demandé. Nous sommes donc en mesure de renvoyer un certain nombre de déplacements (principalement dépendant du nombre d'ajouts dans les mutations) pour chaque unité.

Pour s'assurer que les chemins soient cohérents, il est important de ne pas renvoyer de résultat pendant une itération de l'algorithme. Nous avons forcé ce comportement via l'utilisation de mutex.

Sur requête de l'utilisateur (contenant l'identifiant de l'unité), nous présentons ce résultat case par case parce que notre algorithme sert de « moteur » de pathfinding, et nous estimons que le retour le plus simple pour un utilisateur de notre API est une seule case, et cela nous permet aussi de faire évoluer le chemin en permanence.

L'utilisateur peut alors nous transmettre le déplacement effectif de cette unité sur la case en question au moment qu'il choisit. Lorsque cela arrive, 2 cas de figure se présentent :

- Si c'est un SousMinion (un « suiveur » qui n'a pas de chemin propre) alors nous supprimons simplement son premier déplacement.

- Si c'est un Minion, alors nous ôtons le premier déplacement de son génome, et l'ajoutons à la fin du génome de tous les SousMinions qui le suivent. Nous changeons ensuite la case d'origine du chemin de ce Minion vers la nouvelle case précisée par l'utilisateur. Ceci, couplé à notre représentation du chemin

sous forme de déplacements et non de cases, nous permet en fait de « recycler » notre population, au lieu de repartir de zéro à chaque déplacement.

Lorsqu'un SousMinion atteint sa cible, nous le détruisons. Pour un Minion, nous détruisons aussi la Zone qui lui est associée.

4.1.2. Controller (communication)

Pour permettre à des développeurs d'un jeu de stratégie en temps réel d'utiliser notre algorithme de recherche de chemin nous avons décidé de mettre en place un « controller », faisant référence au modèle Moteur-Vue-Controller (MVC) dont le principe dans ce modèle est de ne donner accès qu'à quelques méthodes spécifiquement choisies par les développeurs. Cette méthode nous permet de contrôler de manière efficace la façon d'utiliser notre algorithme. De plus, cette conception nous a permis de notre côté de mettre en place le projet sous ce modèle.

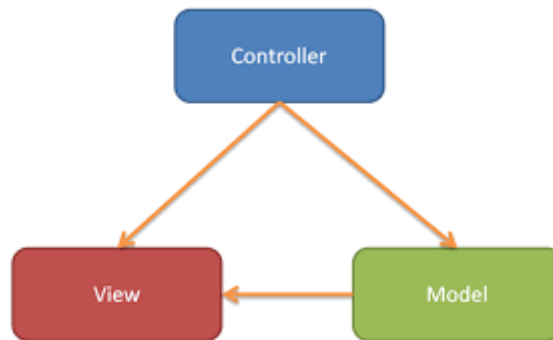


Illustration 2: Modèle MVC

4.2. Difficultés rencontrées

Ce stage nous a permis à tous les trois d'explorer un domaine inconnu pour nous jusqu'ici. De ce fait, tout au long du développement, nous avons fait face à des difficultés, que ce soit dans la programmation ou la conception, qui nous ont aidé à progresser.

4.2.1. Conception de la base de représentation d'un problème de Pathfinding

Les prémisses de notre stage n'ont pas directement commencé avec la conception d'un algorithme permettant un pathfinding. En effet, il nous fallait dans un premier temps concevoir une base solide permettant la représentation d'un problème (représentation d'une carte, des unités, agents, terrains...) auquel on souhaite appliquer sa résolution par le biais d'un pathfinding. Cependant, il nous était demandé de concevoir un API, ce qui nous incombait de concevoir cette base la plus générale possible pour que, une fois programmé, tout problèmes extérieurs, quel qu'il soit, puissent être représenté sans avoir à toucher à cette base. Ainsi, même si 75 % de notre base est inutilisée par le RTS programmé par le groupe partenaire, il devrait normalement être possible d'abstraire n'importe quelle contrainte possible sur un problème pour qu'elle soit exploitée par notre base. Cette conception a ainsi demandé beaucoup de réflexion, de compromis et de temps pour aboutir à celle que nous avons obtenu. Par contre, il est tout de même à noter que même s'ils nous a fallu beaucoup de temps à sa conception, nous en avons pris assez pour produire une base très solide puisque nous n'avons pas eu à y retoucher de tout le stage malgré les éventuels changements du RTS partenaire.

4.2.2. Conception d'un algorithme génétique

La première difficulté rencontrée pour la conception d'un algorithme génétique est la compréhension même de son fonctionnement. Certes il est aisé de faire le lien avec les travaux sur l'évolution de Darwin, mais il fût plus compliqué de concevoir son fonctionnement au travers de code.

Comment représenter la population, le chemin proposé de chaque individu, etc. La difficulté principale suite à la conception, vient d'un problème propre à l'algorithme génétique dont la plupart des recherches effectuées sur ce domaine aujourd'hui ne propose pas de solution optimale. Il s'agit du problème d'évaluation et de fitness. Effectivement, même s'il existe peut être une fitness non optimale mais qui fonctionnerait sur tout type de terrains (terrain ouvert, labyrinthe...), il est tout de même préférable de coder plusieurs fitness optimales pour chaque type de terrains et choisir laquelle utiliser en conséquence.

Or, réfléchir à une fitness est tout de même très abstrait et demande beaucoup de temps, malheureusement, sur deux mois de stage, nous n'avons pas eu le temps de trop nous pencher dessus. Ainsi, la fitness que nous proposons est loin d'être parfaite et nos agents ont souvent tendances à se bloquer dans des « cuvettes » où autre particularité de terrain forçant les agents à faire demi-tour s'ils s'y engouffrent. Ainsi, maintenant notre algorithme génétique fini, nous pensons qu'il s'agit du point sur lequel il faudrait se pencher si nous voulions l'améliorer.

L'autre problème rencontré est celui des « ratios » (de modifications, ajouts, suppressions...), celui-ci aussi fait l'objet de nombreuses recherches pour déterminer quels seraient les ratios optimaux à la résolution de problèmes spécifiques, ce problème joint dès lors avec un problème similaire, celui de l'adaptabilité, c'est à dire à savoir s'il faudrait modifier ces ratios en temps réel afin d'adapter le problème au fur et à mesure de sa résolution.

Ces trois derniers problèmes exposés, comme nous l'avons déjà dit, font tous l'objet de recherche dans ce domaine, mais si nous avions eu plus de temps, nous aurions sans aucun doute effectué de nombreux tests pour essayer de déterminer comment les résoudre et ainsi améliorer fortement les performances de notre algorithme génétique.

4.2.3. Gestion du temps réel

Le temps réel peut ne pas être une difficulté si l'algorithme mis en place possède une complexité linéaire, mais dans notre cas, le temps réel nous a imposé une exigence sur l'optimisation de notre code. En effet, le problème de la recherche de chemin à l'aide de méthodes stochastiques est un problème à complexité NP-Complet, de ce fait, il était primordial qu'on ait une attention particulière à l'optimisation de notre code.

Régulièrement au cours de notre projet, nous avons estimé le temps d'exécution de notre programme, avec des graphes à complexités variables et des déplacements plus au moins lourds, ceux-ci nous ont permis de considérer si, quelque soit l'avancée de notre projet, nous pouvions répondre à la contrainte du temps réel.

La gestion du temps réel est une contrainte que nous viendrons sans doute à rencontrer à nouveau au cours de notre carrière, puisque cela peut se retrouver dans les jeux vidéos comme ici, mais aussi dans certains systèmes d'exploitation.

4.2.4. Création d'une API

L'objectif du projet étant de fournir un outil, c'est sous forme d'API que notre projet peut être utilisé. Cette difficulté s'est ressentie dans la conception du projet où en plus de devoir penser à notre algorithme, nous devons garder à l'esprit de pouvoir le faire communiquer si voulu avec une application extérieure.

4.3. Outils utilisés

4.3.1. Librairies

Certaines librairies gratuites nous ont permis de simplifier la mise en place de notre développement.

a) Libxml++

La conception de la communication avec notre API est passée notamment par l'échange d'informations entre notre programme et le jeu extérieur. Parmi ces informations on peut notamment trouver la carte du jeu souhaitée par les développeurs. Pour recevoir cette information, nous avons décidé de mettre en place un transfert de fichiers au format XML nous donnant toutes les informations

nécessaires sur tout ce qu'on pourra trouver sur la carte, des forêts, des marais, des montagnes et tous les types de terrains qui peuvent intéresser les utilisateurs, le format XML correspondant à ce type d'utilisation.

Pour se faire, plusieurs librairies C++ sont disponibles telles que Xerces, RapidXML, TinyXML et LibXML++. Notre choix s'est porté sur la dernière, LibXML++, car c'est une bibliothèque très complète, souple et bien documentée, qui implémente un support Xpath et qui permet de lire des fichiers XML volumineux. Notre objectif est de donner un outil souple quand à son utilisation, cela implique qu'on ne connait la quantité de données reçu.

b) Glibmm

Libxml++ a été adaptée en C++ à partir de la librairie libxml2 codée en C. De ce fait, pour cette adaptation, il est nécessaire d'utiliser la librairie Glibmm, qui est l'interface multi-plateforme pour la librairie Glib, utilisée dans la librairie libxml2.

4.3.2. Environnement de développement

Les IDE (Environnement de développement) sont des outils qui se révèlent très pratiques pour les développeurs, simplifiant la programmation.

a) Kdevelop

KDevelop est un IDE dont le développement a commencé en 1998 par Sandy Meier. Parmi les avantages qu'il apporte, on peut citer entre autres sa gestion de multiples langages de programmation, sa gestion automatique des projets avec automake. Nous avons choisi d'utiliser cet IDE car dans la première partie de l'année, nos cours de C++ nous ont permis d'appréhender cet outil, et d'apprendre à l'utiliser pour optimiser notre programmation dans ce langage, de ce fait, son utilisation nous a permis de gagner du temps sur des éléments simples et répétitifs de la programmation tel que la déclaration de méthode ou l'inclusion de fichiers.

4.3.3. Débugage

Un projet d'informatique passe forcément par une phase de débbugage du code, quelque soit le langage. Ici aussi, nous avons pu trouver des outils qui nous ont été utile au débbugage efficace de notre projet.

a) Valgrind

Valgrind est un outil bien connu dans le monde de la programmation en C et C++ car ces deux langages intervenant directement avec la mémoire de l'ordinateur, il est facile d'avoir des fuites mémoires à la création, suppression et recopie d'objets. Nous avons eu besoin de cet outil régulièrement pour être sûr que notre projet n'enclenchait aucune fuite mémoire sur l'ordinateur de l'utilisateur.

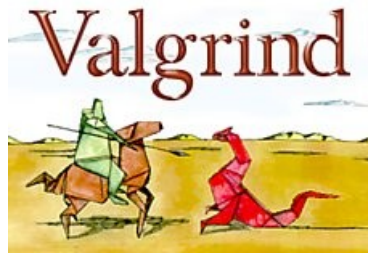


Illustration 3: Valgrind

b) GDB

GDB ou GNU Project Debugger est un utilitaire permettant d'exécuter un programme et de voir ce qu'il s'y passe. Il nous a été grandement utile pour régler les problèmes tels que les erreurs segmentation ou les exceptions sur nos vecteurs. Il permet notamment de retracer la pile d'appels, et donc de savoir dans quel cas une méthode pose problème.

4.3.4. Travail collaboratif

Pour mener un projet à bien, il est nécessaire de mettre en place des outils permettant de mieux communiquer entre les différents développeurs.

a) Git (Github)

Git est le logiciel de gestion de versions le plus populaire. Utilisé par plus de deux millions de personnes, il est une référence dans la mise en place d'un projet de développement. Ce logiciel nous a été grandement utile pour travailler ensemble et mener à bien ce projet.



Illustration 4: Logo du logiciel git

5 Tests effectués

5.1.1. Protocole

Notre algorithme a été pensé pour résoudre des problèmes de pathfinding sur des cartes relativement ouvertes. Nous l'avons testé sur des cartes différentes afin d'observer son comportement et l'influence de certains paramètres. Les cartes utilisées pour les tests qui suivent font 200x200 cases. Nous avons appliqué avec succès notre algorithme sur des cartes de taille allant jusqu'à 1000x1000 cases.

Le temps d'exécution cité ci-dessous ne correspond pas forcément au moment où un chemin a été trouvé pour chaque agent, mais plutôt au moment où les agents se sont déplacés jusqu'à leurs destinations respectives. Les chemins auront potentiellement été trouvés plus tôt, mais dans un contexte temps-réel, nous estimons qu'inclure les déplacements est plus représentatif.

Il est important de noter que notre algorithme est au moins aussi performant en « offline » (non-temps réel), car si nous avons fait en sorte de ne pas augmenter la complexité des calculs lors des déplacements, l'énoncé du problème peut devenir pire qu'au lancement du calcul. Nous avons observé ce phénomène notamment lorsqu'il existe des « cuvettes » dans les cartes, ce qui correspond, d'un point de vue plus abstrait, à des optimums locaux dans l'espace de recherche.

Si une telle cuvette est sur le chemin direct de l'origine à la cible, il y a de très fortes chances que le meilleur chemin renvoyé lors des premiers déplacements pointe vers cette cuvette. La totalité des chemins calculés se fera alors à partir de cette cuvette. En offline, le calcul se fait toujours à partir du point d'origine. Avec une population de taille suffisante, il est probable que quelques chemins a priori sous-optimaux survivent assez longtemps pour contourner la cuvette et être au final meilleurs que les chemins en ligne droite.

Il est à préciser la configuration de l'ordinateur utilisé pour pouvoir convenablement comparer les résultats :

CPU : Intel Core i5 4200H 2.8Ghz

Mémoire RAM : 4GB

GPU : Nvidia Geforce GTX 950m

5.1.2. Carte 1

Cette carte est un exemple relativement aisé à résoudre pour notre algorithme, car aucun des débuts de chemins ne mène rapidement à un optimum local non-solution. Concrètement, cela signifie que l'agent bleu (partant du coin en haut à gauche) a le temps de trouver un chemin qui contourne l'obstacle central avant d'y arriver et donc de se « perdre » dans cet obstacle. Cela arrive parfois tout de même, et il finit toujours par en sortir, mais en plus longtemps.

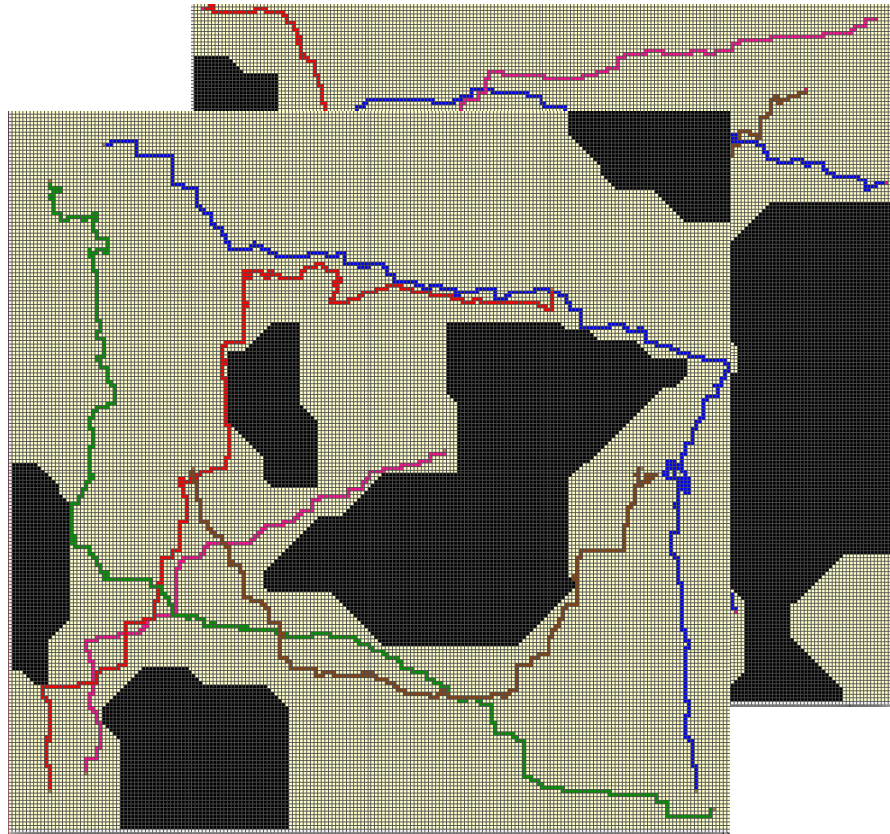
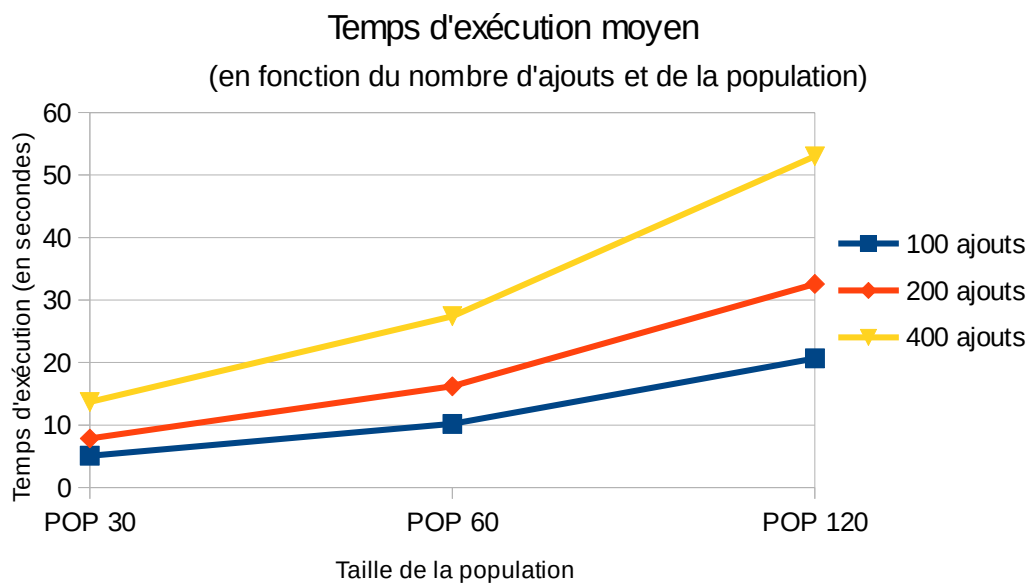


Illustration 5: Carte 1

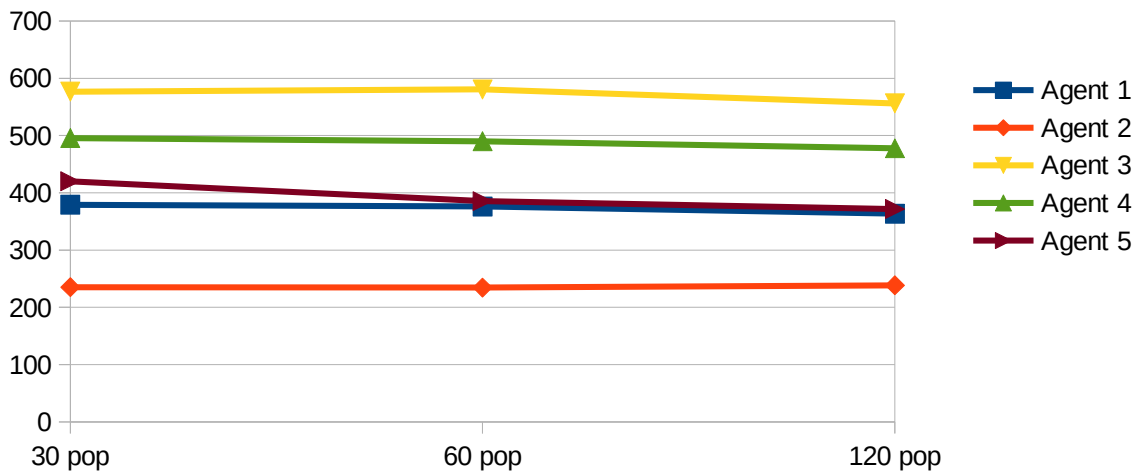


Pour cette carte, la comparaison des temps d'exécution est sans appel : augmenter le nombre d'ajouts ou la taille de la population est coûteux en temps de calcul. Ceci s'explique par l'absence quasi-systématique de déplacements dans des optimums locaux, desquels une grande population et un grand nombre d'ajouts aident à sortir.

A titre indicatif, nous avons calculé le nombre de déplacements correspondant aux chemins optimaux des 5 agents :

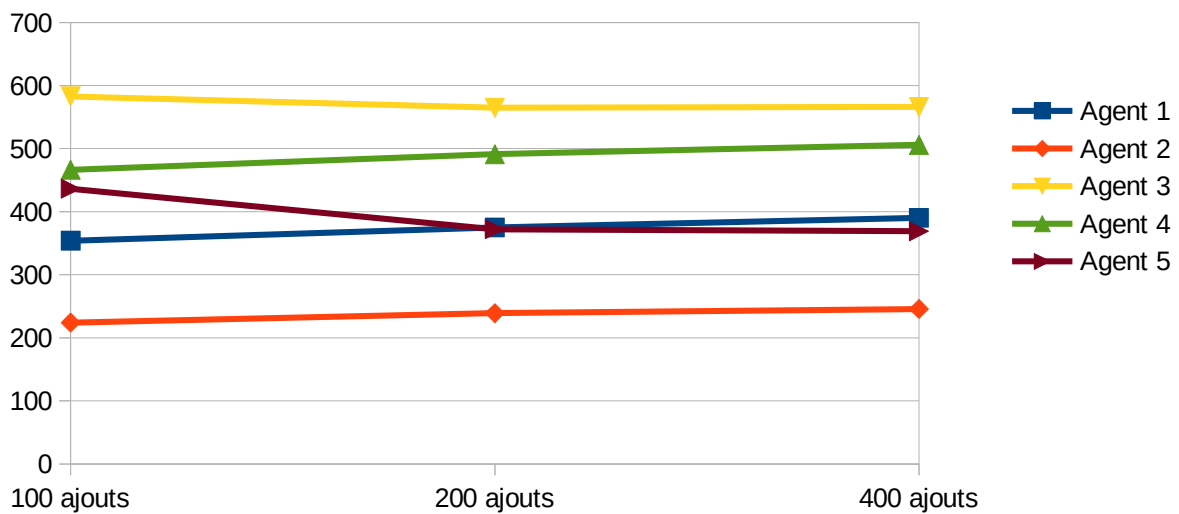
	Optimal
Agent 1	280
Agent 2	190
Agent 3	345
Agent 4	360
Agent 5	225

Comparaison qualité chemin
(en fonction de la taille de la population)



La qualité des chemins trouvés est assez stable, quelle que soit la taille de population choisie. Ici encore, pas d'avantage particulier à avoir une exploration plus importante de l'espace de recherche.

Comparaison qualité chemin
(en fonction du nombre d'ajouts)



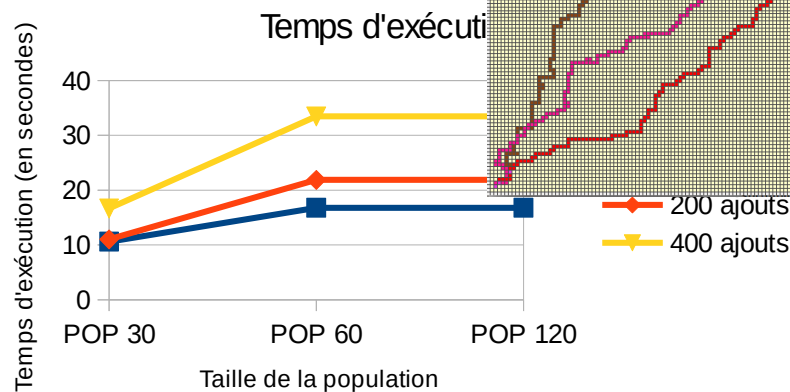
Le nombre d'ajouts ne semble pas avoir d'influence sur la qualité des chemins trouvés.

Pour cette carte, au vu des tests effectués, il ne semble pas utile de favoriser l'exploration de l'espace de recherche. Le temps de calcul peut donc être fortement réduit en diminuant la taille de la population et le nombre d'ajouts, sans conséquences notables sur les solutions renvoyées.

5.1.3. Carte 2

Cette carte est, tout comme la première, aisément solvable. La seule difficulté que l'on pourrait mentionner est celle du parcours du chemin nord-sud vert. En effet, on peut noter une légère « cuvette », cependant, elle est assez peu profonde pour être facilement contournée. Le reste de la carte est ouverte et la plupart des chemins que les agents ont à trouver peuvent s'effectuer en ligne droite.

Illustration 7: Carte 3

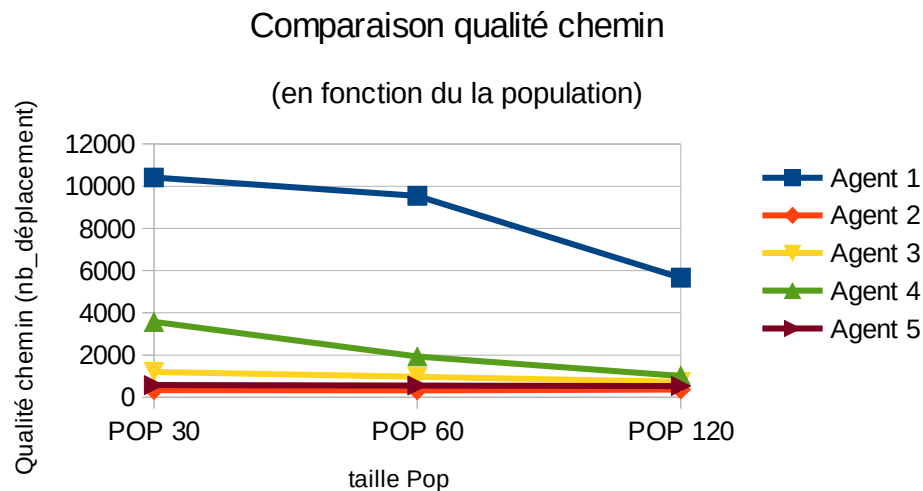


Comme pour la carte précédente, nous pouvons très clairement remarquer qu'en augmentant la taille de la population ainsi que le nombre d'ajout accentue le temps d'exécution. Cependant on remarque une croissance moins exponentielle que la précédente, ici, le temps d'exécution semble stagner une fois le palier de 60 pour la taille de population et de 200 pour le nombre d'ajouts atteints.

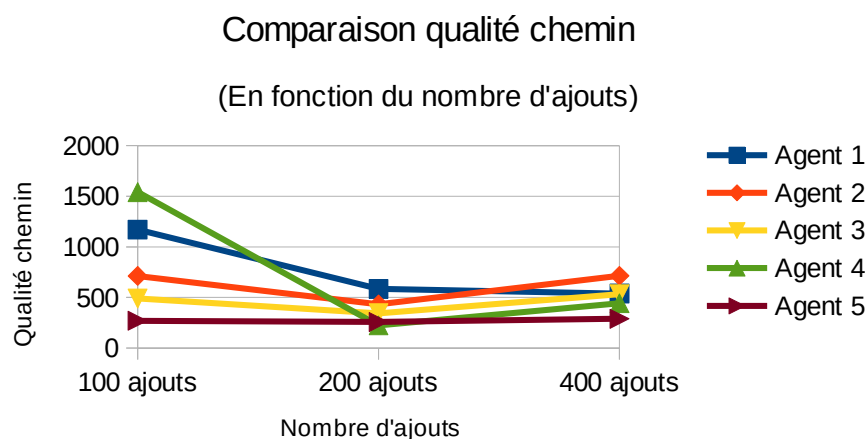
Comparons dès à présent la qualité des chemins trouvés avec en fonction des paramètres ainsi que des chemins optimaux pouvant être obtenu :

	Optimal
A1	312
A2	380
A3	288
A4	177
A5	248

	Optimal
a1	386
a2	283
a3	196
a4	272
a5	340



Ici, nous pouvons remarquer une situation semblable à la précédente, la qualité des chemins est assez stables, même si on remarque une augmentation de la qualité du chemin de l'agent 4 (alias chemin vert). Mais on remarque aisément que la taille de la population a une importance moindre que le nombre d'ajouts, comme nous allons le voir.



En effet, dans ce cas, on remarque qu'augmenter le nombre d'ajout de 100 à 200 a énormément permis l'affinement d'un chemin pour optimal. Cela est dû au fait que les Minions produisant les chemins de l'agent 4 ont plus de chance d'en créer un qui contourne plus proprement la « cuvette ». Cependant, on remarque que l'augmenter de nouveau, de 200 à 400, diminue de nouveau la qualité. Ceci est explicable par le fait que notre fitness se base uniquement sur des distances entre le point d'arriver du chemin généré par le génome, et point d'arriver final de l'agent. Ainsi, un chemin faisant un grand détour mais arrivant plus près du point d'arriver sera mieux considéré qu'un autre pouvant être plus « propre » mais plus éloigné. Or, un grand nombre d'ajout accentue la génération de ce genre de cas de figure.

5.1.4. Carte 3

Cette dernière carte est un peu plus compliquée que les précédentes à résoudre pour notre algorithme génétique. On peut en effet apercevoir dans l'exemple ci-contre de résolution, la présence de deux « cuvettes », une pour le chemin rouge et une pour le chemin vert qui se déplacent tout deux du nord au sud. Le problème de ces cuvettes a déjà été expliqué plus tôt dans la section difficultés rencontrées, les agents s'engouffrent dans la cuvette, mais puisqu'il faut faire demi-tour pour pouvoir en sortir et que nous n'avons pas eu le temps de mettre en place une résolution indépendante de la fitness pour résoudre ce problème, les agents tournent en rond dans ces pièges jusqu'à qu'un autre chemin est « aléatoirement » contourné cette cuvette.

Dans un premier temps, comme pour les deux cas précédents, nous pouvons remarquer que le temps d'exécutions s'accroît au fur et à mesure du grandissement de la taille de la population. Cependant, fait plus étrange, ce temps diminue fortement (près de 65%) lorsque nous avons augmenté le nombre d'ajouts de 100 à 200. Cela peut logiquement s'expliquer car, plus le nombre d'ajout est grand, plus les Minions de l'algorithme ont des chances de générer un génome s'extirpant d'une cuvette. Nous pouvons remarquer tout de même qu'un nombre d'ajout d'environ 200 semble être optimal puisque si nous en mettions 400, le temps d'exécution recommencerait à augmenter. Suite à cela, comme précédemment, comparons les qualités de chemins trouvés en fonction des paramètres ainsi que des chemins optimaux calculés :

Avant tout, précisons que l'agent 1 correspond au chemin rouge et l'agent 4 au chemin vert. On remarque ainsi aisément qu'à faible population, ceux deux agents ont énormément de mal à trouver des chemins de qualité optimale. En effet, ils ont tendance à rester coincés dans les cuvettes, et vu qu'ils doivent constamment renvoyer un chemin, ils finissent par longtemps tourner en rond avant de trouver une échappatoire.

Le cas du nombre d'ajouts de déplacements à chaque itérations est beaucoup plus flagrant. De la même manière que précédemment, on remarque une énorme augmentation de la qualité du chemin avec l'augmentation du nombre d'ajout. Or, précédemment, il s'agissait plus de trouver un chemin sortant (grâce à une augmentation dans la population on obtient plus de chance d'en trouver), ici, il s'agit plus que l'on a de plus en plus de chance d'en créer, le nombre d'ajout étant haut, on a d'autant plus de chance de produire un chemin contournant ou sortant du piège.

Conclusion

Ce stage a été pour nous l'occasion de nombreux apprentissages. Nous avons collaboré en interne sur tous les niveaux : organisation du temps de travail, répartition des tâches, modélisation du programme, implémentation des concepts des algorithmes de Dijkstra, A* et génétique, écriture du code, calculs et débogage. Nous avons aussi travaillé avec le groupe de T.Calatayud et V.Hénaux afin de leur fournir une API viable pour leur jeu de stratégie en temps réel. La présence physique de nos deux groupes dans les mêmes locaux et l'utilisation d'outils adéquats a grandement facilité cet aspect du stage.

Nous avons eu la chance de pouvoir définir nous-même notre sujet (avec les conseils de Mr Stéphane), totalement libres dans le choix du langage de programmation, de la méthode stochastique à employer, et nous ressortons très satisfaits de notre première approche de ces méthodes. Les algorithmes génétiques sont un concept puissant et adaptable à des problèmes variés.

Nous souhaitons par la suite pouvoir réutiliser et développer nos connaissances sur ce sujet en affinant notre travail, en l'appliquant à d'autres problèmes ou mieux encore en remontant d'un niveau d'abstraction et utiliser ces concepts dans la programmation génétique. Nous pensons que les sujets très actuels de l'apprentissage machine et de la génération automatisée de code peuvent être abordés en réutilisant les concepts des algorithmes génétiques.

Bibliographie

Bibliographie

MCPPRTSG: Munir Naveed, Diane Kitchen, Andrew Crampton, Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games,

AADO: Marco Dorigo, Gianni Di Caro, Luca M. Gambardella, Ant Algorithms for Discret Optimization,

RTPGA: Alex Fernandes da V.Machado, Ulysses O. Santos, Higo Vale, Rubens Gonçalves, Tiago Neves, Luiz Satoru Ochi, Esteban Walter Gonzales Clua , Real Time Pathfinding with Genetic Algorithm,

DGASSPP: Saeed Behzadi, Ali A. Alesheikh, Developing a Genetic Algorithm for Solving Shortest Path Problem, 2008

GAGP: Steve Gargolinski, Genetic Algorithms for Game Programming,

Blickle & Thiele, "Selection Schemes in GA": Tobias Blickle & Lothar Thiele, A Comparison of Selection Schemes used in Genetic Algorithms, 1995

Eiben, Raué & Ruttkay, "GA with multi-parent recombination": A.E Eiben, P-E. Raué, Zs. Ruttkay, Genetic algorithms with multi-parent recombination, 1994

Les méthodes stochastiques permettent de trouver des solutions au problème de la recherche de chemin
mots-clés : Stochastique, algorithme génétique, fonction d'évaluation, exploration, intensification.

Les algorithmes génétiques sont une façon intéressante de résoudre cette classe de problème. Leur efficacité dépend grandement des choix effectués pour la représentation des solutions, la fonction d'évaluation et les paramètres internes de l'algorithme.

Ces choix représentent souvent un compromis entre exploration, intensification et temps de calcul. Leur influence respective est variable selon la représentation des solution et la complexité du problème à résoudre.

Annexes

Résultats des tests

Carte 1:

		100 ajouts			200 ajouts			400 ajouts		
		min	max	moy	min	max	moy	min	max	moy
POP 30	a1	289	507	358	305	619	380	305	667	399,46
	a2	191	627	225,38	201	937	237,88	199	647	242,45
	a3	378	1026	583,37	381	1024	579,97	382	814	566,55
	a4	379	719	470,58	379	715	497,26	395	847	518,98
	a5	266	1051	488,94	264	878	377,28	260	784	394,34
	Temps	3,92	6,65	5,09	5,89	10,91	7,84	10,71	17,39	13,69
POP 60	a1	299	479	348,08	293	619	380,26	301	687	400,24
	a2	193	1011	223,15	195	755	238,9	205	351	241,88
	a3	368	990	599,93	376	883	568,4	382	842	573,65
	a4	373	656	462,97	383	780	495,64	395	773	511,25
	a5	250	882	415,77	254	577	367,71	256	722	373,42
	Temps	8,14	15,67	10,19	11,22	20,94	16,20	21,35	34,66	27,41
POP 120	a1	295	571	355,41	295	553	364,47	299	589	370,73
	a2	197	435	223,3	195	1089	240,02	197	759	252,46
	a3	358	978	564,99	362	790	545,8	374	771	557,41
	a4	375	721	465,83	385	737	480,35	391	778	487,45
	a5	260	781	404,71	262	650	371,23	252	604	339,12
	Temps	14,30	27,15	20,66	26,18	53,05	32,57	43,24	63,89	53,00

Carte 2:

		100 ajouts			200 ajouts			400 ajouts		
		min	max	moy	min	max	moy	min	max	moy
POP 30	a1	437	6513	1548,01	407	1242	633,93	397	785	545,23
	a2	425	996	709,48	425	981	709,18	455	1022	737,05
	a3	393	593	477,34	447	911	531,94	329	765	538,62
	a4	374	8471	2285,78	280	1563	699,96	234	1143	516,37
	a5	255	301	272,8	253	313	278,92	263	872	293,75
	Temps	6,08	20,09	10,60	8,36	15,22	11,03	12,41	20,91	16,64
POP 60	a1	335	3885	1124,96	377	969	575,93	377	803	544,94
	a2	425	939	714,47	411	927	741,07	419	1018	724,15
	a3	433	583	498	323	635	506,26	359	707	526,33
	a4	380	5520	1370,38	262	1129	710,52	224	1042	426,54
	a5	253	297	271	255	741	280,86	257	337	287,72
	Temps	10,76	28,22	16,78	17,88	29,87	21,87	26,14	42,49	33,43
POP 120	a1	1593	405	839,22	397	789	546,47	391	729	527,29
	a2	1040	417	713,12	405	1006	711,81	423	916	682,62
	a3	583	323	494,21	327	773	521,52	341	759	530,82
	a4	2114	328	968,54	234	1132	659,25	216	898	387,31
	a5	297	251	266	251	719	279,96	255	371	286,12
	Temps	41,62	24,64	32,35	34,31	53,50	44,25	50,30	93,16	66,27

Carte 3:

		100 ajouts			200 ajouts			400 ajouts		
		min	max	moy	min	max	moy	min	max	moy
POP 30	a1	715,00	105563,00	27543,06	503,00	16313,00	2814,45	549,00	2183,00	874,16
	a2	296,00	366,00	318,78	298,00	3298,00	353,12	298,00	1382,00	351,02
	a3	477,00	10240,00	2282,06	243,00	1709,00	795,56	223,00	1344,00	514,64
	a4	345,00	29314,00	9076,05	319,00	1929,00	970,26	323,00	1169,00	670,78
	a5	445,00	1191,00	601,42	377,00	981,00	570,49	387,00	1303,00	554,46
	temps	6,25	190,06	65,09	10,46	47,62	17,82	13,30	28,15	19,51
POP 60	a1	543,00	131272,00	25917,65	535,00	6385,00	1918,67	429,00	1235,00	802,15
	a2	290,00	360,00	314,32	288,00	372,00	320,56	296,00	1516,00	361,87
	a3	445,00	7496,00	1562,51	261,00	1891,00	774,05	227,00	1146,00	571,87
	a4	335,00	20953,00	4502,23	339,00	1323,00	732,27	325,00	913,00	563,80
	a5	453,00	1023,00	585,25	371,00	1073,00	551,06	371,00	1035,00	534,61
	Temps	16,70	397,43	100,49	21,31	47,49	30,75	30,33	54,54	39,31
POP 120	a1	519,00	141003,00	15173,39	517,00	4262,00	1093,18	486,00	1013,00	720,18
	a2	288,00	10311,00	418,65	300,00	1766,00	344,58	304,00	1108,00	350,01
	a3	375,00	3166,00	967,48	279,00	1314,00	706,72	227,00	988,00	504,37
	a4	321,00	9760,00	1904,30	295,00	1190,00	646,72	317,00	961,00	478,68
	a5	393,00	957,00	569,32	381,00	759,00	531,59	355,00	889,00	490,27
	Temps	27,42	889,25	129,98	39,11	89,25	54,68	46,77	102,22	68,91

ENGAGEMENT DE NON PLAGIAT

Nous, soussignés Théo Voillemin, Nicolas Roux, Corentin Talarmain
déclarons être pleinement conscients que le plagiat de documents ou d'une
partie d'un document publiée sur toutes formes de support, y compris l'internet,
constitue une violation des droits d'auteur ainsi qu'une fraude caractérisée.
En conséquence, nous nous engageons à citer toutes les sources que nous avons utilisées
pour écrire ce rapport.

signé par les étudiants le 30/06/2016

**Cet engagement de non plagiat doit être signé et joint
à tous les rapports, dossiers, mémoires.**

Présidence de l'université
40 rue de rennes – BP 73532
49035 Angers cedex
Tél. 02 41 96 23 23 | Fax 02 41 96 23 00

