|  |
| --- |
| École Polytechnique de Montréal |



**INF4215**

**Introduction à l'intelligence artificielle**

**Hiver 2015**

**Travail pratique 3**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Remis par :**

**Francis Trudeau-Beaulieu #1632734**

**Kévin Alexandre Boissonneault #1631079**

**Gabriel Loyer # 1637122**

**Gabriel St-Laurent # 1631687**

**À :**

**Cédric Barbez**

**Le 17 avril 2015**

# Introduction

Dans le cadre du travail pratique final du cours d'introduction à l'intelligence artificielle, nous avions la possibilité de réaliser une application d'intelligence artificielle de notre choix, tant qu'elle consiste en un niveau de difficulté adéquat. Nous avons décidé de nous mettre en équipe de 4 personnes pour réaliser deux agents intelligents qui assumeraient le rôle des joueurs dans un jeu développé par Kevin. Les deux agents intelligents ont été développés indépendamment par deux sous-groupes de notre équipe: l'implémentation du joueur un a été faite par Kévin et Francis, alors que le joueur deux a été fait par les Gabriel. Le but était de développer deux agents qui apprennent par Q-Learning et sont en compétition un et l'autre, puis d'observer le résultat. Le code du jeu et un algorithme de recherche de chemin A\* est partagé par les deux équipes, mais le reste de l'implémentation des agents intelligents différents.

# Description du jeu

Tout d'abord, nous allons expliquer les principes généraux du jeu. Celui-ci est relativement simple. Au démarrage, une carte de 30x30 tuiles est générée. Le joueur peut se déplacer dans 8 directions, d'une tuile à l'autre. Chaque carte possiblement générée n'a aucune tuile isolée, chaque tuile du jeu peut donc être accédée. Il y a 5 types de tuiles: plancher, mur, monstre, arme et trésor. Les murs sont des cases noircies inaccessibles, les quatre autres types sont des cases franchissables par le joueur. En allant sur une case trésor, le joueur récolte de l'argent, qu'il doit accumuler pour gagner une partie, qui se termine lorsqu'aucun monstre ou trésor ne reste sur la carte. En plus de son pointage, le joueur possède un attribut de force, qui augmente ses probabilités de gagner un combat. Il peut démarrer un combat avec un monstre ou l'autre joueur, qui auront aussi une force. En cas de victoire contre un joueur adverse, le joueur récolte une certaine somme d'argent, égale à la différence de son roulement de force. C'est le même qui s'applique en cas de défaite, où le joueur perd de l'argent et réapparaitra à un des deux spawn points, celui le plus loin du joueur ennemi. Chaque monstre possède initialement une somme d'argent égale à sa force, et peut en accumuler plus lorsqu'il bat un joueur. Lorsqu'un joueur bat un monstre, il récolte tout l'argent que le monstre avait à ce moment. Quand un joueur marche sur une case arme, des points de forces sont ajoutés à la sienne: il augmente ainsi ses chances de battre un monstre ou l'adversaire pour lui voler des points. Le jeu est tour par tour, le jeu attend donc une action des deux joueurs avant de se mettre à jour. À la fin d'une partie, on peut appuyer sur espace et une autre partie sur la même carte recommencera. Les deux agents intelligents apprendront donc à s'améliorer dans le même environnement partie après partie. Veuillez noter que différents arguments peuvent être entrés en partant le jeu en ligne de commande pour modifier son comportement. On peut entrer "size" pour spécifier la taille de la salle, "seed" pour entrer le seed qui génère une map manuellement, "max\_trail" pour spécifier le nombre de trailes indiquant les déplacements passés des joueurs, "window\_size" pour spécifier la taille de la fenêtre. Vous pouvez vous référer à la fonction ParseArgs dans le fichier Game.cpp pour l'utilisation des paramètres.

# Agent intelligent: Kévin et Francis

Pour arriver à développer un agent intelligent qui allait arriver à s'améliorer après un certain nombre de parties sur une même carte de jeu, nous avons utilisé l'apprentissage machine Q-Learning vu en classe. Au lieu de définir une action choisie par l'IA comme étant une direction, nous avons développé une recherche de chemin A\* et avons décidé qu'une action représente une tuile vers laquelle se dirige l'agent. Lorsqu'il prend une décision aléatoire pendant son apprentissage, on prend une case monstre, trésor ou arme au hasard qui est définie comme son but. On envoie cette case à l'algorithme A\*, qui bâtit alors une pile de directions à aller pour que l'IA atteigne son but. À chaque tour, l'IA dépile une direction de la pile et l'exécute. Nous avons ensuite différentes valeurs de récompenses, positives et négatives qui sont associées aux actions que l'IA fait selon le résultat. Avec l'algorithme de Q-Learning, nous avons mis une valeur epsilon de 0.3, faisant en sorte que l'agent effectuera une action aléatoire 3 fois sur 10 en moyenne, afin d'aider son apprentissage. Dans les 7 autres cas, il effectuera l'action avec une récompense la plus élevée associée à cet état selon son expérience. Un état est différent d'un autre si la position d'un joueur ou le type d'une tuile diffère. Avec le Q-Learning, notre IA devrait donc en théorie s'améliorer à chaque partie, car il va de plus en plus se retrouver dans des états qu'il connait et il prendra l'action qui lui donne le plus de récompense.

Nous avons rencontré plusieurs difficultés lors de l'implémentation de notre agent intelligent. La première était que nous avions très peu de temps, car nous avions tous une fin de session très occupée avec la remise du projet 3 et d'autres travaux de sessions. Nous avons sous-estimé le temps requis pour implémenter du Q-Learning dans ce projet, car nous pensions originalement qu'une action correspondrait à une direction effectuée par le joueur. Nous avons à mi-chemin réalisé qu'avec une telle approche, notre IA n'apprendrait jamais, car le jeu possède beaucoup trop d'actions qui mèneraient à une tuile vide, et donc sans récompense. L'agent va donc tourner en rond pendant des centaines de parties interminables sans apprendre. Pour remédier à cette situation, nous avons donc implémenté un algorithme de pathfinding A\* et encapsulé les actions comme étant une tuile clé visée par l'IA. Cela nous a demandé beaucoup d'efforts non prévus, mais a été fructueux au final. Une autre difficulté est qu'à la fin, c'est tout de même très difficile de tester si notre IA s'améliore réellement de partie en partie, car plusieurs résultats dans le jeu dépendent de hasard et l'avancement du niveau du joueur à chaque partie est difficile à mesurer.

Notre agent intelligent possède aussi plusieurs limites. La plus grande est que nous avions initialement prévu qu'il irait faire des combats avec l'autre joueur s'il jugeait que cela pourrait lui rapporter des récompenses, mais nous avons ensuite réalisé que c'est très complexe, car l'autre joueur se déplace de façon imprévisible à chaque tour. Nous avons donc abandonné cette idée, et notre IA est incapable d'engager volontairement un combat, peu importe les bénéfices potentiels, sauf s'il fonce dans l'autre joueur simplement par hasard. Une autre limite est qu'il prend horriblement de temps à s'améliorer à cause d'un beaucoup trop grand nombre d'états différents faisant en sorte qu'il se retrouve rarement dans un état dont il connait l'action optimale. Comme amélioration possible, nous pourrions combler la première limite expliquée et essayer de faire une action pour laquelle l'IA se dirige vers l'autre joueur pour initier un combat lorsque c'est bénéfique, en essayant de prédire dans quelle direction va se diriger l'autre joueur pour l'intercepter et initier un combat.

# Agent intelligent: Gabriel et Gabriel

Pour développer notre agent intelligent, nous avons tout d’abord fait quelques recherches pour les algorithmes connus. Nous avons opté pour l’algorithme de QLearning car il nous semblait le plus facile et efficace pour notre situation. L’agent QLearning doit pouvoir comprendre deux concepts : State et Action.

Pour le concept State, qui doit représenter un état de jeu, il nous semblait nécessaire de créer une abstraction du jeu réel pour se soulager de données superflues (les cases vides, par exemple). Nous avons opté pour une liste des cases importantes (trésors, monstres, autre joueur), dans une représentation spéciale, avec le type, la valeur de trésor, d’arme, la distance et la direction par rapport à l’agent. Ainsi, on souhaite produire un agent qui pourra réutiliser sa connaissance dans des parties différentes, car les données conservées concernent des états qui ne sont pas liés à la partie. Autrement dit, les états peuvent être conservés d’une partie à l’autre.

Pour le concept Action, nous avons opté pour définir une action comme une cible à atteindre, où une cible est un élément important (trésor, monstre ou autre joueur). Quand il choisit cette cible, il se dirige vers elle à l’aide d’un algorithme de calcul de chemin A\*. Cela lui permet de faire abstraction de la disposition de la zone de jeu.

Pour choisir une action (cible), notre agent choisis d’abord la cible qu’il pense la plus intéressante (selon sa valeur de Q). Aussi, s’il décide de changer de cible, il a une chance de choisir une cible au hasard (pour explorer ses options). Cette chance est proportionnelle à la valeur Alpha de sa cible, qui indique l’’expérience qu’à l’agent avec cette décision dans l’état donné. Ainsi, pour une action qu’il connait bien, il risque moins d’explorer les autres options.

Les problèmes rencontrés dans le développement de notre agent furent assez nombreux, mais heureusement nous avons trouvé des solutions adéquates. L’un des plus grands problèmes fut de trouver une abstraction correcte de l’environnement du jeu pour permettre à l’agent d’être réutilisable dans plusieurs parties, d’être rapide et d’être efficace. Pour y arriver, nous avons discuté ensemble de différentes solutions, et avons fait une analyse des caractéristiques de chacune, pour en arriver à la solution décrite plus haut.

Les grandes limites de notre agent intelligent sont que sa vision de l’état de jeu ne contient pas toutes les informations, et que les actions sont restreintes à celles définies. Notre agent ne peut donc pas effectuer d’évitement de joueur ou de monstre. Puisque la possibilité d’état est relativement grande, il requiert également un grand nombre d’itérations avant d’être capable de propager les récompenses au travers des états afin de discerner les actions bénéfiques.