La Mine Hantée

Eliott Jacopin, Anaëlle Badier, Augustin Borderon,
Liliana Jalpa Pineda, Abel Masson

Janvier 2020

1 Introduction

L'objectif de ce rapport est de présenter le jeu de la Mine Hantée que nous avons développé, sous la forme d'une application Python. Il a pour objectif de présenter les choix de conception que nous avons fait en fonction éventuellement des contraintes que nous avons rencontrées. Pour chacun de ces choix, nous argumenterons autour de sa raison d'être et préciserons ses conséquences pour le reste des fonctionnalités développées et/ou ses retombées, positives comme négatives, sur l'ensemble de l'application.

2 Choix de conception

2.1 Choix de la partition en classes pour une programmation orientée objet

Pour développer l'application, nous avons choisi une programmation orientée objet. Cette approche nous a semblé meilleure que la programmation procédurale, parce qu'elle est modulaire et donc plus facilement compréhensible lorsque les codes deviennent conséquents, mais aussi parce qu'elle permet d'effectuer des modifications sur certains modules sans affecter les autres, ce qui est un très grand atout lorsqu'il s'agit de coder à 5. La définition de classes munies d'attributs et de méthodes permet de modifier les instances de classe plus facilement et de façon plus économe en temps et en mémoire. Cet avantage nous a paru particulièrement intéressant dans le contexte du développement d'un jeu de plateau, qui par définition est amené à évoluer, et donc à être très fréquemment modifié, au fil des interactions avec les joueurs.

Dans ce contexte nous avons définis les classes suivante :

- La classe "Carte", dont les instances représentent les différentes cartes présentes sur le plateau (leur type, leur position, leur orientation et leur contenu). Pour gérer l'orientation d'une carte, nous avons défini 4 directions : *(Nord, Sud, Est, Ouest)*, à chaque type de carte nous avons une liste des connectivités possibles en fonction de l'orientation de la carte. Ces connectivités étant représentées par une liste de 4 chiffres binaires de valeur

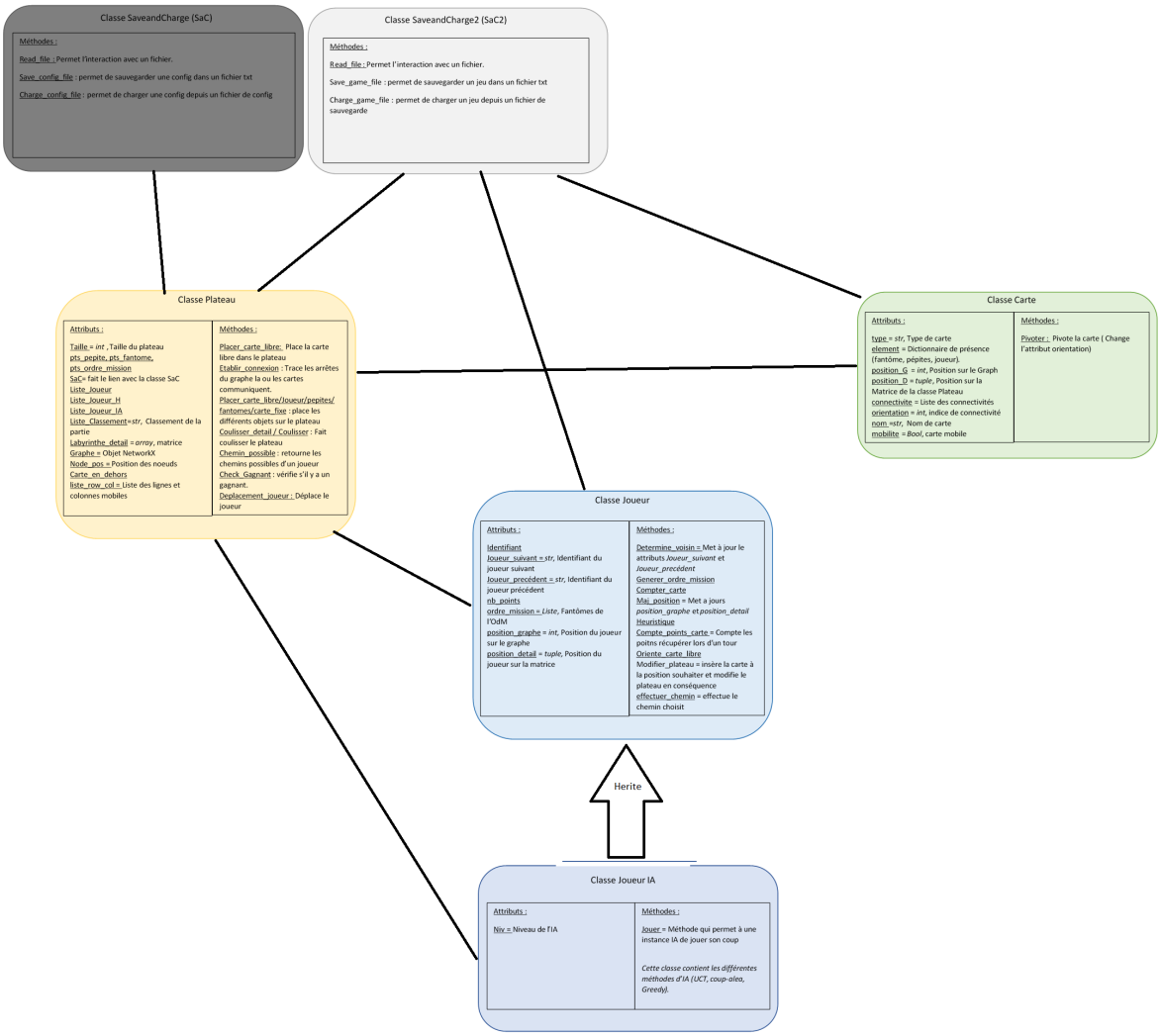
0 si la carte n'est pas ouverte sur la direction correspondante, 1 sinon. L'orientation était ensuite définie comme un indice de cette liste. Cette approche a facilité la gestion de la rotation des cartes et l'établissement des chemins entre les cartes.

- La classe "Joueur", dont hérite la classe "JoueurIA" et dont les instances représentent les joueurs de la partie (leur ordre de mission, leur position sur le plateau, et leur gain). La classe joueur IA comporte les différentes approches IA utilisées et fait appel aux différentes méthodes de "Joueur" qui permettent de faire tourner la carte libre, d'insérer la carte où le joueur le souhaite et de se déplacer.

- La classe "Plateau" dont chaque instance représente le plateau et qui garde en attribut tous les éléments présent sur ses différentes cases. Ses méthodes permettent de faire coulisser le plateau (à l'appel d'un élément de la classe "Joueur" ou "JoueurIA"), d'établir le classement et de vérifier si il y a un gagnant. Elle comporte également des méthodes et des attributs permettant de faire le lien avec le graphe représentant le plateau et de le mettre à jour.

- Les classes SaC and SaC2 (pour Save and Charge) ont pour rôle respectif la gestion (écriture / lecture) des fichiers de configuration du jeu et la gestion des fichiers de sauvegarde. Nous avons fait le choix de créer des fichier de type *.txt*. Nous aurions pu dans le cadre du fichier de sauvegarde utiliser les types fichiers et fonctionnalités prévus spécialement par python pour sauvegarder des objets, mais nous avons fait le choix de produire un fichier texte pour des raisons de lisibilité et car cela autorise la modification du fichier si l'on veut changer les différents éléments de la partie.

En résumé nous avions les relations suivantes entre les classes :



Enfin il était prévu à l'origine une classe simple fantôme qui fut par la suite inutilisée.

Au cours de l'avancement de notre projet certains attributs se sont finalement retrouvés inutilisés car il portait une information inutile au accessible autrement. Nous avons également constaté des doublons dans des attributs portant la même information mais étant tous deux utilisés dans des méthodes diverses. On peut citer à titre d'exemple les attributs de la classe carte *type* et *nom* qui portent tous deux l'information du type de l'instance carte ("Couloir", "Carrefour" et "Coin").

2.2 Choix de l'utilisation d'un graphe NetworkX pour représenter le plateau de jeu et repérer les chemins empruntables par les joueurs.

Nous avons choisi de voir le plateau comme un réseau de cartes, pouvant être reliées par un ou plusieurs chemin. Nous avons donc utiliser la bibliothèque NetworkX, de visualisation, manipulation et parcours de graphe. Les fonctionnalités de cette bibliothèque nous ont permis d'identifier très rapidement les liens entre les cartes, et de déterminer les différents chemins empruntables par un joueur donné à un tour donné. Sans les fonctions de parcours de graphe de la bibliothèque NetworkX, ce travail aurait été plus fastidieux et surtout plus coûteux en temps et en mémoire.

3 Choix dans le moteur de jeu et dans la visualisation

3.1 Paramètrage et début du jeu

Il était laissé libre aux joueurs de paramétrer eux mêmes (avec des valeurs par défaut) tous les éléments du jeux (taille du plateau, nombre de fantômes, nombre de fantômes par ordre de mission, nombre de pépites, points attribués pour une pépite, un fantôme et un fantôme de l'ordre de mission). En revanche nous avons limité le plateau à une taille de 15x15 et le nombre de fantôme par OdM à 10 pour des raisons d'encombrement visuel et de performance. Enfin les paramètres sont reliés pas des relations logiques : il ne peut pas y avoir plus de fantômes que le plateau peut en accueillir, et plus de fantômes par OdM que de nombre de fantômes.

Le paramètrage et l'initialisation du jeu est réalisé via le code *"Launcher"* qui utilise le module TkInter dans son jeu de fenêtre, nous avons choisi d'utiliser ce module plutôt que Pygame par exemple car il possède des méthodes et des outils prédéfinis facilitant la navigation entre les fenêtres et le recueil des paramètres.

3.2 Moteur et Visualisation

Nous avons fait le choix de réaliser la visualisation et le moteur du jeu dans deux fichiers distincts. Ce qui facilite grandement la gestion du jeu. Le code de la visualisation (*vizpygame*) utilise le module Pygame proposé par Spyder spécialement conçu pour l'interface graphique de jeu, en revanche il est codé en programmation procédurale ce qui le rend un peu lourd. Le moteur de jeu met à jour les différents éléments des classes présentée en partie 2 en fonction des actions sur l'interface graphique. A chaque étape, la visualisation est mise à jour ce qui nécessite une relecture intégrale de la matrice de l'instance de la classe *plateau*. Cela entraîne des mises à jour visuelles lentes, particulièrement quand la taille du plateau, et donc la taille de la matrice à parcourir, augmente.

Nous avons choisi une visualisation assez simple et à l'utilisation très intuitive pour un joueur humain :

En revanche, un joueur IA lors de son tour va directement se téléporter à sa case d'arrivée en ramassant les différents fantômes et pépites qu'il peut ramasser sur le chemin. Il sera donc difficile pour un joueur humain de suivre parfaitement les coups joués par une IA.

Enfin, nous avons choisi comme critère d'arrêt de jeu, le fait que le joueur en tête ne puisse plus être rattrapé mathématiquement par les autres joueurs. Ainsi après chaque tour, le moteur de jeu vérifie si le joueur peut encore être rattrapé avec les points restant sur le plateau. Sinon il arrête le jeu et provoque la visualisation d'une nouvelle fenêtre, celle de la fin du jeu :

4 Choix dans les algorithmes définissant le jeu des Intelligences Artificielles (IAs)

- UCB
- Min/Max
- Approche Glouton
- Coup au hasard

Nous avons choisi de coder plusieurs algorithmes pour définir le jeu de différentes IA. L'utilisation de plusieurs algorithmes différents nous a permis de définir plusieurs niveau de jeu pour les IA ; l'algorithme jouant au hasard correspond au niveau de difficulté "débutant", l'algorithme par Greedy Approach correspond au niveau "intermédiaire", et l'algorithme UCB correspond au niveau de difficulté "expert".

L'algorithme jouant au hasard choisit à chaque tour un chemin au hasard parmi les chemins empruntables par le joueur. Il n'est pas optimisé, et constitue donc le premier échelon de difficulté. L'algorithme par approche gloutonne choisit à chaque tour le chemin qui maximise les gains du joueur, i.e le chemin qui permet au joueur de collecter le maximum de pépites et d'attraper le maximum de fantômes. Rien ne garantit que cette approche soit optimale pour le joueur, lorsqu'il s'agit de gagner la partie. En effet, cette approche ne permet pas de définir une stratégie sur plusieurs tours, à fortiori sur l'intégralité de la partie. Cet algorithme constitue donc le niveau de difficulté intermédiaire. Enfin l'algorithme UCB (Upper Confidence Bound) choisit le chemin que le joueur emprunte en fonction du gain maximal qu'il pourra en tirer après plusieurs

tours, en calculant ce gain à l'aide d'une Monte Carlo Tree Search (MCTS). Pour des raisons pratiques, car la MCTS peut s'avérer très longue dans le cas de notre jeu, nous n'avons pas fixé un nombre de tours, mais plutot un temps de calcul de 10 secondes pour chaque coup à jouer. Nous notons x le nombre de tours pris en compte par la MCTS dans la limite des 10 secondes. Pour chaque chemin empruntable, l'algorithme simule le choix de ce chemin ainsi que les x tours suivants et estime les gains que peut espérer le joueur à l'issue de ces x tours. Il retournera en définitive le chemin qui conduit au gain maximal possible. Cet algorithme est donc le seul à même de définir une stratégie de jeu sur plusieurs tours. Il constitue le niveau de difficulté expert.

Une implémentation de l'algorithme mini max était prévue. Cette approche équivaut à une approche gloutonne sur plusieurs tours, elle maximise les points de celui qui l'utilise et minimise le points du reste des joueurs. Cet algorithme aurait permis d'avoir une autre méthode qui prévoit plusieurs coups, en fonction de la profondeur que l'on souhaite explorer, mais l'implémentation n'a pas pu être finie dans le temps imparti. Cependant, étant donné que le facteur de branchement est assez élevé et que la copie du plateau peut-être coûteuse en mémoire, le temps de calcul peut vite devenir élevé même en utilisant de l'élagage alpha-bêta. La performance de cette méthode serait donc à comparer à l'approche gloutonne qui correspond à l'utilisation d'un mini max sur une profondeur de 1. Nous avons supposé que le rapport mémoire-temps et gain pour le joueur est similaire dans les deux cas : l'implémentation gloutonne satisfait, dans une certaine mesure, les critères souhaités pour l'approche mini max.

- Test du bien-fondé de la définition des niveaux de difficulté par la simulation de parties :

Au cours de nos essais de nos parties, nous nous sommes rendu compte que l'approche gloutonne était d'une efficacité redoutable, alors que l'approche UCT est lente et donc peu efficace dans le cadre de nos capacités de calcul limitées. Nous avons donc décidé de transférer l'UCT en difficulté normale et l'approche gloutonne (Trop ardue à battre pour un niveau moyen) en niveau difficile.

Les niveaux d'IA finaux sont donc les suivants :

- Niveau débutant : Approche Aléatoire
- Niveau intermédiaire : Approche UCT
- Niveau difficile : Approche Gloutonne

5 Choix dans la conception du réseau

Le réseau se base sur la bibliothèque bas niveau intitulée "socket". Le serveur ainsi que chaque client sont instanciés dans des thread séparés et communiquent via des chaînes de caractères encodées en bit. Dans le cas de la mine hantée,

notre objectif était de construire un serveur qui ne soit présent que pour transmettre les informations d'un joueur à l'autre. Tous le moteur du jeu ainsi que la visualisation devait être contenue au n. Lorsqu'un joueur (i.e. client) jouait un coup sur son plateau, il envoyait alors la composition de ce coup au serveur qui le communiquait ensuite aux autres clients. Le coup était décodé et le plateau de chacun des clients était mis à jour en fonction de ce dernier.
Pour les IAs, nous avions prévue de les instancier avec le client créateur du serveur. C'est-à-dire que 'lorsque le joueur cliquait sur le bouton "Connexion au serveur", il créait des clients qui correspondaient aux IAs.
Malheureusement, même si nous avons réussi à créer le serveur et instancier le serveur les clients sans difficultés (pas pour les IAs à ce stade), l'instanciation de la visualisation PyGame était très instable. En effet, très souvent, PyGame "tournait en rond" sur une fenêtre noir avant même d'accéder au code que nous avions écrit ; et de temps en temps, tous ce lançait sans problème.
Nous supposons que cela a un lien avec l'instanciation des thread pour les clients et le serveur car c'est la seule différence avec le cas d'une partie locale (où PyGame se lance sans problèmes).
En raison de cette complication, nous n'avons pas pu implémenter la communication entre un clients et tous les autres lorsqu'il effectue un coup. Il est donc impossible d'effectuer une partie en réseau même si les clients sont bien connectés.