POLYTECH LYON

Rapport de conception du Jeu d'échecs

Mark KPAMY & Fabien OGLI & Christian TAGGUEJOU

Table des matières

Introduction

Ce rapport présente nos travaux sur la réalisation d'un jeu d'échecs doté d'une intelligence artificielle avec plusieurs niveaux de difficultés.

Modèle de conception

Nous avons créé le jeu d'Echec de façon modulaire. La classe Plateau est composée essentiellement composé d'une grille et deux joueurs.

La Case

Chaque case de la grille est un objet de la classe Case, elles possèdent tous un booléen "occupied", d'un entier "couleur", d'un caractère "id" et d'une coordonnée.

Le booléen occupied renvoie vrai si la case est occupé. Si la case est occupée, l'entier "couleur" renvoie 0 si la case est occupé par une pièce noir, 1 si la case est occupée par une pièce blanche, sinon il renvoie -1. Le caractère "id" permet de renvoyer l'identifiant de la pièce présente sur la case, si la case n'est pas occupée, l'identifiant est 'N'.

La Grille

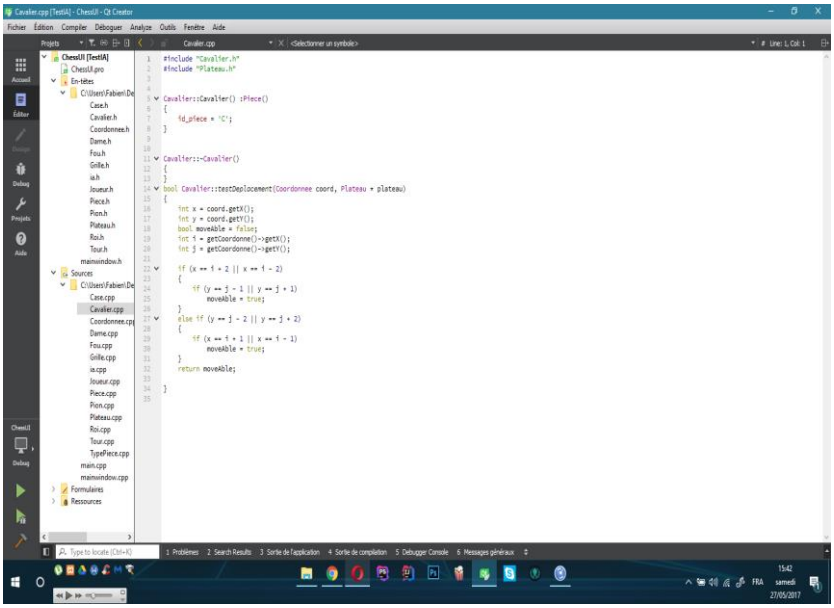
La classe Grille possède un tableau de pointeur d'objet "Case". Nous avons privilégié les pointeurs aux objets afin d'avoir un coût moindre en mémoire et pour pouvoir modifier directement les données des objets.

La méthode "putPiece" prend en paramètre un pointeur de pièce et permet de changer les valeurs de la case avec les coordonnées correspondantes pour qu'ils concordent avec les attributs de la pièce.

La méthode "removePiece" quant à elle fait l'inverse, elle remet les valeurs de la cases à son état initial.

Les Pièces

L'ensemble des pièces du jeux d'échec ont leur classe spécifique et héritent tous de la classe "Piece". Les classes spécifiques des pièces nous servent à spécifier le déplacement de chacune de pièce, ainsi que leur identifiant. En effet, à chaque objet de la classe "Piece" à un caractère "id_piece" permettant de savoir de quel type de Pièce est l'objet. Par exemple, à l'instanciation d'un Cavalier, l'identifiant est 'C'.



Chaque objet de la classe "Piece" a un pointeur vers un objet Coordonne, un booléen "alive" qui devient faux lorsque la pièce est attrapée, un entier couleur qui permet de définir si la pièce est noir ou blanche et ainsi qu'un caractère "id_piece" qui a été expliqué plus haut.

Les méthodes Essentielles de la classe "Piece"

La pièce se déplace via la méthode "move". Celle-ci emploie la méthode "testDeplacement", si cette méthode renvoie vrai, alors on modifie les coordonnées de la pièce, sinon on affiche une fenêtre indiquant que le déplacement n'est pas possible. Si la pièce peut se déplacer sur une case occupée par une pièce adverse, la méthode "move" effectue la méthode "kill" en mettant en argument la pièce adverse. La méthode "kill" active le destructeur de la Piece passé en paramètre.

La classe "Piece" possède une méthode "deplacementPossible" qui prend en paramètre l'entier qui définit le joueur et la plateau. Cette méthode parcourt la grille et effectue un "testdeplacement" toutes les cases. Elle renvoie un QVector de QPoint qui comporte les coordonnées des cases où les déplacements sont possibles.

Le Joueur

La classe "Joueur" a en attribut en entier "id" qui définit si c'est le joueur qui possède les pièces noires ou blanches, une coordonnée origin qui permet de positionner les pièces sur la grille et un entier "wayToMove" qui indique dans quel sens doivent se déplacer les pions.

Le joueur possède un vecteur de pointeur de pièce "deck" qui répertorie les différentes pièces d'un joueur. La méthode "isAnyPiece" est essentielle pour le jeu d'échec. Elle renvoie un entier correspondant à la position de la pièce dans le vecteur "deck". Si le joueur ne possède pas la pièce, la méthode renvoie -1.

Le Plateau

Lorsque le plateau est instancié, on instancie la grille, les joueurs et l'ia.

Lorsque l'on appuie sur l'onglet "Lancer Partie", la méthode "displayPlateau" est appelée, ce qui permet d'afficher toutes les pièces sur le plateau via la méthode "affichSuppInit".

Cette dernière méthode permet d'afficher et d'effacer une pièce. Elle prend en paramètre la pièce, l'id du joueur ainsi qu'un entier qui est 0 pour afficher, 1 pour effacer.

La méthode "movePiece" s'effectue lorsque l'on a choisi le départ et l'arrivée de la pièce et que l'on appuie sur le bouton "OK". Elle effectue le déplacement de la pièce si la pièce peut bouger sinon on fait apparaître un fenêtre indiquant un mauvais déplacement. Elle permet aussi de faire jouer l'IA lorsque le déplacement est bon.

Algorithmes d'IA

Les algorithmes d'IA que nous avons sont principalement l'algorithme MIN-MAX combiné avec l'algorithme alpha-bêta. Ces deux algorithmes sont les plus utilisés pour la recherche du meilleur coup à jouer. Il est donc notamment très en vogue chez les développeurs de jeux de réflexion (échecs, dames, go, etc). En dépit de ses avantages et sa popularité, l'algorithme MIN-MAX tient son inconvénient dans la récursivité. Le temps de parcours peut s'avérer très long si la profondeur est élevée. Dans notre cas :

- Pour une profondeur de 2 coups, l'IA fait son choix en moins de 3 secondes en début de partie
- Pour une profondeur de 4 coups, l'IA fait son choix en environ 10 secondes en début de partie
- Pour une profondeur de 6 coups, on avoisine à peu près 20 secondes

Nous allons maintenant présenter notre classe IA.

Présentation de la classe IA

Notre classe IA est le moteur d'intelligence artificielle de notre jeu d'échecs. Ses principales méthodes sont :

```
ia(int level);
    void setLevel(int i);
    int getLevel();
    QVector<QPoint> jouer(Joueur *joueur,Plateau *plateau);
    int eval(int couleur[8][8],char idPiece[8][8]);
    QVector<QPoint> calc_echec_et_mat(int idJoueur,QPoint pos_rois_joueur);
```

```
    void initTableauTmp(Plateau * plateau);
    int max(int idJoueur,int profondeur,int alpha,int beta,int
couleur[8][8],char idPiece[8][8]);
    int min(int idJoueur,int profondeur,int alpha,int beta,int
couleur[8][8],char idPiece[8][8]);
    int gagnantEnCours(int idJoueur);
  ~ia();
```

Le détail des paramètres de ces fonctions pourra être retrouvé dans le code source.

Fonction jouer()

Cette fonction constitue le point d'entrée de l'algorithme. C'est lui qui parcourt les pièces du joueur et élague les branches.

Voici un pseudo-code de la fonction de laquelle est inspirée l'algorithme que nous utilisons :

```
fonction ALPHABETA(P, alpha, beta) /* alpha est toujours inférieur à beta
*/
   si P est <= 0 alors
       retourner la valeur de P
   sinon
       si P est un nœud Min alors
           Val = infini
           pour tout enfant Pi de P faire
               Val = Min(Val, ALPHABETA(Pi, alpha, beta))
               si alpha ≥ Val alors  /* coupure alpha */
                   retourner Val
               finsi
               beta = Min(beta, Val)
           finpour
       sinon
           Val = -infini
           pour tout enfant Pi de P faire
               Val = Max(Val, ALPHABETA(Pi, alpha, beta))
               si Val ≥ beta alors /* coupure beta */
                   retourner Val
               finsi
               alpha = Max(alpha, Val)
           finpour
       finsi
   retourner Val
   finsi
fin
```

Notre fonction jouer() se comporte comme suit :

```
fonction jouer(etat du jeu) : void

     max_val <- -infini

     Pour tous les coups possibles
          simuler(coup_actuel)
          val <- Min(etat_du_jeu, profondeur)

          si val > max_val alors
               max_val <- val
               meilleur_coup <- coup_actuel
          fin si
```

```
          annuler_coup(coup_actuel)
     fin pour

     retourner(meilleur_coup)
fin fonction
```

Il s'agit donc pour chaque coup de le simuler et de choisir à la fin le plus rentable.

Fonction min()

Cette fonction permet de simuler tous les coups possibles de l'adversaire. Le but de la cette function est de trouver le minimum des nœuds-enfants du nœud envoyé en paramètre.

```
fonction Min(etat du jeu) : entier

     si profondeur = 0 OU fin du jeu alors
          renvoyer eval(etat_du_jeu) OU valeur de fin de jeu

     min_val <- infini

     Pour tous les coups possibles
          simuler(coup_actuel)
          val <- Max(etat_du_jeu, profondeur-1)

          si val < min_val alors
               min_val <- val
          fin si

          annuler coup(coup actuel)
     fin pour

     renvoyer min_val
fin fonction
```

Fonction max()

Cette fonction permet de simuler tous les coups possibles de l'IA. A la différence du min(), ici on cherche les nœuds max.

```
fonction Max(etat du jeu) : entier

     si profondeur = 0 OU fin du jeu alors
          renvoyer eval(etat_du_jeu) OU valeur de fin de jeu

     max_val <- -infini

     Pour tous les coups possibles
          simuler(coup_actuel)
          val <- Min(etat_du_jeu, profondeur-1)

          si val > max_val alors
               max_val <- val
          fin si

          annuler_coup(coup_actuel)
     fin pour

     renvoyer max_val
fin fonction
```

Fonction gagnantEncours()

Cette fonction permet de savoir à un instant donné qui a l'avantage, c'est-à-dire le gagnant de la situation.

Cette fonction permet de calculer la fin ou la continuation de la partie. Si le jeu n'est pas encore terminé car le joueur adverse ou l'IA ont encore des coups possibles, il retourne 0 sinon il retourne -1000 ou 1000 suivant que le joueur qui joue a perdu ou gagné.

Fonction calc_echec_et_mat()

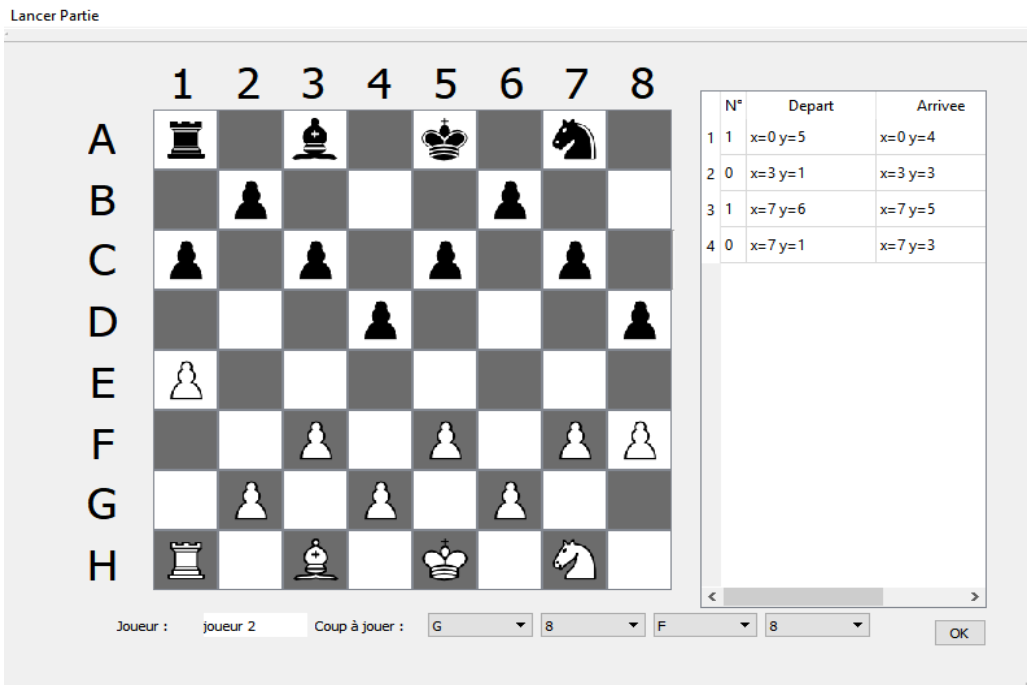
Cette fonction permet de savoir si le joueur adverse est en échec et mat ou non. Si c'est le cas, il renvoie un vecteur de déplacements possibles et dans le cas contraire un vecteur vide.

Fonction eval()

Cette fonction permet d'évaluer le plateau. On attribue à chaque pièce une valeur fixe (Pion = 1, Fou=3, T=5). On n'ajoute pas le roi car il ne peut être pris. Ensuite, on calcule le score obtenu par chaque joueur à travers ses pièces, cela permet d'avoir une idée du sort pour chaque joueur et donc de comparer les situations pour en choisir la plus avantageuses.

Interface graphique

L'interface graphique de notre jeu a été réalisée sous Qt Creator. Cette interface permet de visualiser et faire les déplacements, de lister les déplacements effectués, faire le choix du niveau de difficulté. Cette interface reste assez basique car elle n'était pas le but principal du projet. Nous avons tout de même apprécié utiliser ce logiciel car sa librairie graphique est très facile à prendre en main et la gestion des signaux est plus que jamais simplifiée.



Conclusion

La réalisation de ce jeu nous a permis de mûrir nos réflexions sur le fonctionnement et l'implémentation d'une intelligence artificielle. Il fût aussi pour nous un challenge car nous avons décidé de le coder en C++ avec une interface graphique réalisé avec Qt Creator. Nous avons vite compris les défis auxquelles se heurtent les développeurs de jeu en termes de temps de réponse, d'efficacité et de complexité. Les problèmes auxquelles nous avons été confrontés sont principalement l'amélioration du temps de décision de l'IA. Pour résoudre cela, nous avons simplifier tout notre code et utiliser des variables assez légères comme des tableaux d'entiers et de caractères.