Rapport SIM 202 : Jeu d'échecs

Jonas Benhamou, Hugo Blanc, Pierre Cortambert, Jean-Michel Mamfoumbi
Année 2020-2021

Table des matières

1 Introduction

L'objectif du projet est d'implémenter un jeu d'échecs permettant à un joueur d'affronter un ordinateur. Le projet se divise en plusieurs parties qu'on peut réaliser simultanément en se répartissant correctement le travail : l'implémentation de classes nécessaires au jeu d'échec (échiquier, pièces...), l'implémentation de l'algorithme du min-max qui permettra à l'ordinateur de jouer, le test de cet algorithme à un jeu simple comme le tictactoe et enfin son application pour le jeu d'échecs.

2 Implémentation du jeu d'échecs

Pour la structure du jeu d'échec, nous avons élaboré plusieurs classes permettant de construire les différentes composantes du jeu d'échec. Nous avons notamment créé une classe permettant de construire une position du jeu d'échec, cette classe contient un objet échiquier, contenant lui-même les pièces qui seront des objets de la classe **pièce**. Par ailleurs chacune des pièces a un type qui sera lui-même un objet de la classe **type-pièce**. Aussi, nous détaillerons par la suite la construction de ces différentes classes.

2.1 Définition de l'échiquier et des pièces

```
#ifndef ECHIQUIER_H
#define ECHIQUIER_H
#include "piece.h"
#include "coup.h"
#include <iostream>
#include <vector>

using namespace std;


class echiquier
{
    private:
        vector<vector<piece*>>  plateau; //Plateau avec les différentes pièces restantes
    public:
        echiquier(); //Permet d'initialiser un plateau de début de jeu
        echiquier(vector<coup> liste);  //Permet de représenter le plateau après un enchaînement de coup
        vector<vector<piece*>> getPlateau(){return plateau;} //Accesseur du plateau
        piece* getPiece(int i,int j){return plateau[i-1][j-1];} //Accesseur de la pièce aux coordonnées i et j
        void print(); //Permet d'afficher le plateau
        bool isaPiece(int i, int j); //Vérifie si la case est vide
        int istwopawns();
};

#endif // ECHIQUIER_H
```

Définition de la classe échiquier

Avec cette classe on a défini le plateau comme un vecteur de vecteur dimension 8x8 et chaque composante de ce vecteur renvoie à une case qui peut être occupée par une pièce ou simplement être vide. On y retrouve des fonctions d'initialisation ou de modification d'un échiquier et une fonction d'affichage de l'échiquier. Dans le même temps nous avons défini les classes **pièce** et **type pièce** qui nous permettent de représenter les pièces de l'échiquier :

```cpp
#ifndef PIECE_H
#define PIECE_H
#include "type_piece.h"

class echiquier;
enum Lettre{a=1, b=2,c=3,d=4,e=5,f=6,g=7,h=8};
enum Couleur{Noir, Blanc};
class piece
{
    private:
        type_piece type;
        Couleur couleur;
        vector<int> position;

    public:
        piece():type(){}
        piece(type_piece t, Couleur c, vector<int> p):type(t),couleur(c),position(p){}
        piece(const piece& a){type=a.type;couleur=a.couleur; position=a.position;}
        Couleur getCouleur(){return couleur;}
        vector<int> getPosition(){return position;}
        type_piece getType(){return type;}
        void setCouleur(Couleur c){couleur=c;}
        void setType(type_piece t){type=t;}
        void setPosition(int i,int j){position=vector<int>(2);position[0]=i;position[1]=j;}
        piece& operator=(const piece& a){type=a.type;couleur=a.couleur; position=a.position; return *this;}
        int val();
        int control(echiquier chess);
        bool onBoard(vector<int> relativ); //vérifie si un coup relatif reste dans le plateau
        bool access(echiquier chess,vector<int> relativ); //vérifie si la pièce accès à une zone du plateau
        bool isanAlly(echiquier chess,vector<int> relativ);
        bool isanOpponent(echiquier chess,vector<int> relativ);
        bool accessMove(echiquier chess,vector<int> relativ);
        void print();

};

#endif // PIECE_H
```

Définition de la classe pièce

La classe pièce permet de représenter une pièce avec sa couleur, sa position actuelle et son type (objet type_pièce qui sera décrit par la suite).

```cpp
#ifndef TYPE_PIECE_H
#define TYPE_PIECE_H
#include <vector>
using namespace std;

enum PIECE{Pion, Cavalier, Fou, Tour, Dame, Roi,None};
class type_piece
{
    private:
        PIECE type;
        int valeur;
        bool moved=false;
        vector<vector<int> > relativattack;
        vector<vector<int>> relativmove;

    public:
        type_piece():type(None),valeur(0){}
        type_piece(PIECE P, int v, vector<vector<int> > attack, vector<vector<int>> move):type(P),valeur(v), relativattack(attack),relativmove(move){}
        type_piece(const type_piece& P):type(P.type),valeur(P.valeur),relativattack(P.relativattack),relativmove(P.relativmove),moved(P.moved){}
        type_piece & operator=(const type_piece& t){type=t.type;valeur=t.valeur; relativattack=t.relativattack; relativmove=t.relativmove; moved=t.moved;return *this;}
        PIECE getPIECE(){return type;}
        int getValeur(){return valeur;}
        bool getMoved(){return moved;}
        void setMoved(bool c){moved=c;}
        vector<vector<int>> getAttack(){return relativattack;}
        vector<vector<int>> getMove(){return relativmove;}
        void setPIECE(PIECE t){type=t;}
        void setValeur(int v){valeur=v;}

};

#endif // TYPE_PIECE_H
```

Définition de la classe type-pièce

Les attributs sont en privé donc il faut définir des acceseurs pour les modifier ou les utiliser. Nous avons aussi créé des fonctions qui nous assurent qu'un coup est autorisé. Dans ces classes on retrouvera toutes des informations relatives à une pièce : type de pièce, position et mouvements

relatifs (de déplacement classique ou d'attaque). Ainsi quand on s'intéressera à une case du plateau on pourra savoir si elle possède une pièce et dans cette situation déplacer la pièce sur l'échiquier.

Pour un aspect pratique on voudrait travailler avec un échiquier classique dont les cases sont définies par :

— une lettre entre a et h
— un chiffre entre 1 et 8

On définit donc grâce à un enum les caractères entre a et h ce qui permet de garder le formalisme habituel des échecs.

Il reste ensuite à créer une classe pour créer et suivre les coups, ainsi que créer la classe **position echec** qui est semblable à **position**. C'est avec cette classe qu'on pourra utiliser l'algorithme du min-max et faire jouer l'ordinateur.

```
#ifndef COUP_H
#define COUP_H
#include "piece.h"

enum special{PR, GR, Usual };

class coup
{
    public:
        piece playpiece;
        vector<int> lastpos;
        vector<int> newpos;
        piece pieceprise;
        special coupspecial;
        coup();
        coup(piece p,vector<int> a, vector<int> b,piece prise, special c){playpiece=p; lastpos=a; newpos=b;pieceprise=prise;coupspecial=c;}
        piece getPlaypiece(){return playpiece;} //Retourne la pièce joué par ce coup
};

#endif // COUP_H
```

Définition de la classe coup

```
#ifndef POSITION_ECHEC_H
#define POSITION_ECHEC_H

#include <vector>
#include "echiquier.h"

class position_echec
{
    public:
        vector<coup> liste;
        position_echec* pfille;
        position_echec* psoeur;
        int joueur; //On considère que le joueur blanc a pour indicateur 1 et le noir a pour indicateur 0, joueur qui doit jouer la position

        ~position_echec();
        position_echec(){liste={};pfille=0; psoeur=0;}
        position_echec(vector<coup> l,int joueur):liste(l),joueur(joueur){pfille=0; psoeur=0;}
        //les fonctions
        position_echec* getSoeur(){return psoeur;}
        position_echec* getFille(){return pfille;}
        float valeur(float alpha=1,float beta=1,float gamma=1, Couleur ordiCouleur=Blanc);
        position_echec* genererpositionfils();
        void print(){echiquier chess(liste);chess.print();}
        position_echec& operator=(const position_echec& pos){liste=pos.liste; pfille=pos.pfille;psoeur=pos.psoeur;joueur=pos.joueur;}
        bool check();
        bool checkmate();
};
```

Définition de la classe **position-échec**

Grâce à cette classe, il sera possible de générer les positions résultantes d'une position initiale (en observant tous les déplacements possibles pour les pièces du joueur dont c'est le tour).

Ensuite à partir du **min-max** il sera possible de faire jouer l'ordinateur contre nous. La fonction valeur déterminera le style de jeu de l'ordinateur (défensif, agressif, misant sur l'échange de pièces...).

2.2 Problèmes rencontrés

Nous avons eu plusieurs principaux problèmes pour l'implémentation du jeu d'échec

Le choix de structures adaptés

Au début du projet il était compliqué de savoir ce dont nous aurions besoin et donc de trouver des structures parfaitement adaptées. Il a fallu réfléchir aux fonctions essentielles notamment comment vérifier de façon efficace la légalité des coups. Par ailleurs, il était compliqué de se projeter dans les différentes structures, et de comprendre comment les différentes classes allaient pouvoir interagir les unes avec les autres.

La gestion des coups spéciaux

Il a été complexe de savoir où nous pourrions faire intervenir la gestion des coups spéciaux, en effet ceci pouvait intervenir dans la classe coup, ou encore être géré directement lors de la génération des positions filles en effectuant des tests sur l'échiquier (par exemple pour vérifier si le roque était possible).

La gestion de la promotion des pions

Il a fallu implémenter cette règle des échecs qui stipule que si un pion arrive sur la dernière ligne face à lui, le joueur peut transformer le pion en la pièce de son choix. Nous avons pensé d'abord à réaliser un test à chaque tour de jeu pour savoir si un pion (blanc ou noir) avait atteint la position en question et auquel cas effectuer le changement indiqué plus haut. Le problème est que cela impliquait une recherche sur 16 cases à chaque tour dès le départ. C'est un problème car cette méthode très simpliste rajoute beaucoup de calculs et dans le cas de la génération de positions à une profondeur élevée, cela aurait demandé de vérifier cette condition pour chaque position fille ce qui semblait très lourd.
Nous avons donc décidé de voir cette promotion de pièce comme un coup spécial divisé en deux étapes : atteinte de la dernière ligne et transformation du pion. En voyant la promotion de cette manière nous avons trouvé une méthode plus efficace pour l'implémenter. On réalise des test sur la pièce déplacée (en vérifiant son type, sa position et ses positions accessibles) puis nous modifions son type et sa valeur.
Enfin, nous avons choisi d'octroyer le type dame au pion lors de sa promotion pour simplifier la méthode : c'est aussi le choix des joueurs experimentés.

3 Implémentation du min-max

L'algorithme du min-max permet de faire jouer un ordinateur contre un humain sur un grand nombre de jeux à deux joueurs. Pour faire fonctionner l'algorithme il faut disposer d'une fonction qui note l'état de la partie pour l'ordinateur. Cette fonction d'évaluation joue un rôle essentiel dans notre algorithme. Obtenir une fonction optimale peut s'avérer difficile voir impossible.
Rentrons dans le détail de l'algorithme. Nous partons d'une position donnée, nous générons tout les coups possibles depuis cette position. Ensuite pour chaque coups possible nous générons à nouveau l'ensemble des cous possible et ceci jusqu'à un profondeur donnée. Nous obtenons

l'arbre des coups possibles. Une fois que l'on a généré toutes les position possible nous utilisons la fonction d'évaluation pour noter l'état de la partie sur les derniers postions de l'arbre. Puis nous remontons d'un étage. Si c'est à l'ordinateur de jouer on attribue la valeur max à la position précédente et si c'est au joueur on attribue la valeur min. Voici la représentation de l'algorithme.

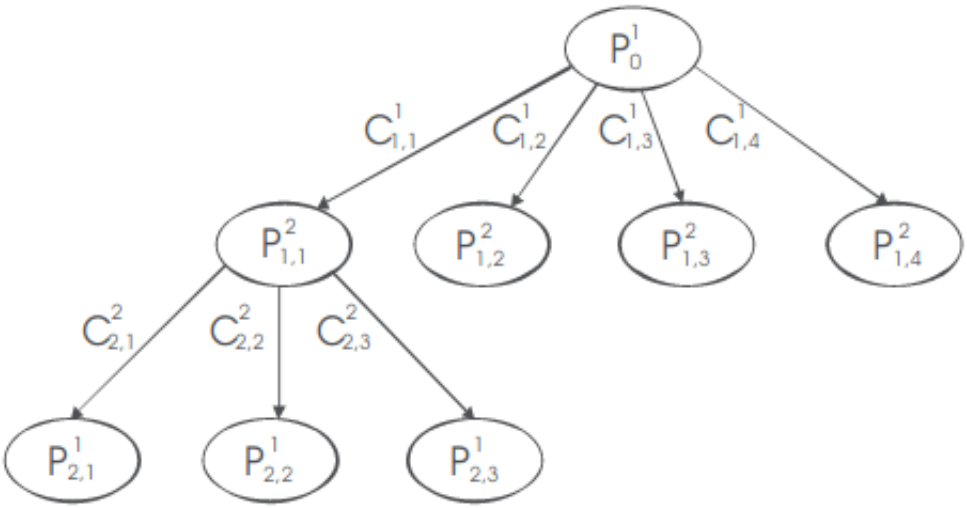


Schéma de l'algorithme du min-max
Ce graphique est issu du rapport de présentation du projet

Pour l'implémentation de l'algorithme du min-max, nous allons écrire une classe abstraite que l'on appel **Position**. Notre algorithme du min-max va fonctionner sur cette classe abstraite.

```
class position
{
public:
    position* psoeur;
    position* pfille;
    int joueur;
    //les fonctions
    virtual float valeur()=0;
    virtual position* genererposition()=0
};
```

Définition de la classe position

De façon abstraite l'objet position représente les cercles. L'attribut pfille est un pointeur de la classe position qui pointe vers le premier coup possible. Ensuite pfille permet de chaîner les différentes positions possibles. Enfin, nous avons deux fonction virtuelles pures qui sont propres à chaque type de jeu et nécessaire à l'algorithme du min-max.

Pour implémenter l'algorithme du min-max, nous distinguons la recherche du max sur le coups suivant et l'algorithme min-max que nous avons décrit. Nous avons choisi de découper l'algorithme de la façon suivante car le coup $n+1$ doit être retenu tandis que les coups suivants sont

utiles seulement pour calculer la valeur de la position.
Voci l'algorithme du min-max qui renvoie la valeur d'une position et descend jusqu'à un profondeur donnée depuis une position donnée.

```
double minmax_val(int deep,position* P){
    if (deep==0){return P->valeur();}
    P->genererposition();
    if (P->pfille ==0){return P->valeur();}
    double extremum=minmax_val(deep-1,P->pfille);
    position* courant=(P->pfille)->psoeur;

    if (courant==0){return extremum;}
    if (courant->joueur==1){//joueur=1 C'est une machine
        //max
        while(courant!=0){//joueur=0 C'est un humain
            double temp=minmax_val(deep-1,courant);
            if(temp >= extremum){extremum=temp;}
                courant=courant->psoeur;
            }
        }

    if (courant==0){return extremum;}
    if (courant->joueur==0){
    //min
        while(courant!=0){
            double temp =minmax_val(deep-1,courant);
            if(temp <= extremum){extremum=temp;}
                courant=courant->psoeur;
            }
    }
    return extremum;
}
```

Algorithme qui donne la valeur d'une position

Voici l'algorithme global du min max qui renvoie la nouvelle position de notre jeu après le coup de l'ordinateur.

```
position* minmax(int deep,position* P){
    P->genererposition();
    position* extremum=P->pfille;
    if(extremum==0){return P;}
    position* courant=(P->pfille)->psoeur;
    while(courant!=0){
        //max
        if(minmax_val(deep-1,extremum)<=minmax_val(deep-1,courant))
            {extremum=courant;}
        courant=courant->psoeur;
    }
    return extremum;
}
```

Algorithme du min-max

Lors de l'implémentation de l'algorithme, nous n'avons pas directement pensé à séparer la première recherche de max des autres. Nous avons donc eu du mal à coder cette partie.
De plus, la classe étant abstraite, il nous a été difficile de comprendre comment elle fonctionnait. Enfin, nous avons mis longtemps a corriger une erreur. Nous essayions d'accéder à un attribue d'un pointeur qui était en réalité nul. Le debbugueur nous a permis de comprendre cette erreur. Pour l'enlever nous avons rajouté une condition qui teste si le pointeur est nul. Nous retenons de cette erreur l'utilité du debbugeur pour les erreurs liées à un mauvaise utilisation de la mémoire ainsi que l'importance de vérifier que nos variables ne sont pas nulles.

4 Application du min-max

4.1 Validation avec le tic-tac-toe

En parallèle de l'implémentation de l'algorithme du min-max on a crée une classe **position tic-tac-toe** qui hérite de la classe position (classe utile dans l'algo du min-max). L'héritage va nous permettre de valider le min-max en l'appliquant au jeu du tic-tac-toe.

```
108  class position_tictactoe : public position
109  {
110    public:
111    grille position_actuelle ;
112    virtual float valeur();
113    position_tictactoe(int ,grille);
114          virtual position_tictactoe* genererposition();
115          ~position_tictactoe(){};
116    virtual void affiche(){position_actuelle.afficher_grille();}
117
118  };
```

Définition de la classe position-tictactoe

La classe a trois attributs, les deux dont elle hérite par la classe position (pointeurs vers la position soeur et vers la position fille) et une grille qui sera modifiée au cours de la partie. Les fonctions **valeur** et **genererposition** permettent de déterminer le coup suivant à partir de l'algorithme du min-max.

Contrairement au jeu d'échecs il y a une faible quantité d'information donc nous avons estimé qu'une classe **grille** suffisait à définir l'avancement de la partie en cours. L'attribut grille de notre classe **position tictactoe** est un élément de cette classe.

```
//La case vaut -1 si elle n'est pas occupée
class grille
{
  public :
  int grille_[9]={-1,-1,-1,-1,-1,-1,-1,-1,-1};
  //A faire un constructeur par copie
  int & operator()(int i,int j){return grille_[co_to_int(i,j)];}
  bool jouer(int i,int j,int joueur);
  int valeur();
  ~grille(){}
  void afficher_case(int,int);
  void afficher_grille();
  bool grille_pleine();
};
```

Définition de la classe **grille**

Un élément de la classe **grille** est un tableau d'entiers dont on modifiera les valeurs en fonction des coups joués par l'utilisateur et de l'ordinateur grâce aux fonctions **jouer** et **grille pleine**. La fonction **affiche grille** permet d'afficher la partie en cours.

4.1.1 Problèmes rencontrés

Nous avons rencontré divers problèmes. Le premier problème était le choix de structures adaptées pour implémenter une partie de tictactoe : nous ne voulions pas faire de classes ou de fonctions superflues. Si pour la classe échec il était important de garder une trace des coups joués, nous avons pensé que pour le tictactoe, la grille (tableau d'entiers) suffisait étant donné qu'il n'y avait que neuf cases de jeu. Aussi, nous avons préféré travaillé avec un liste de taille 9 plutôt qu'une matrice 2d de taille 3x3.

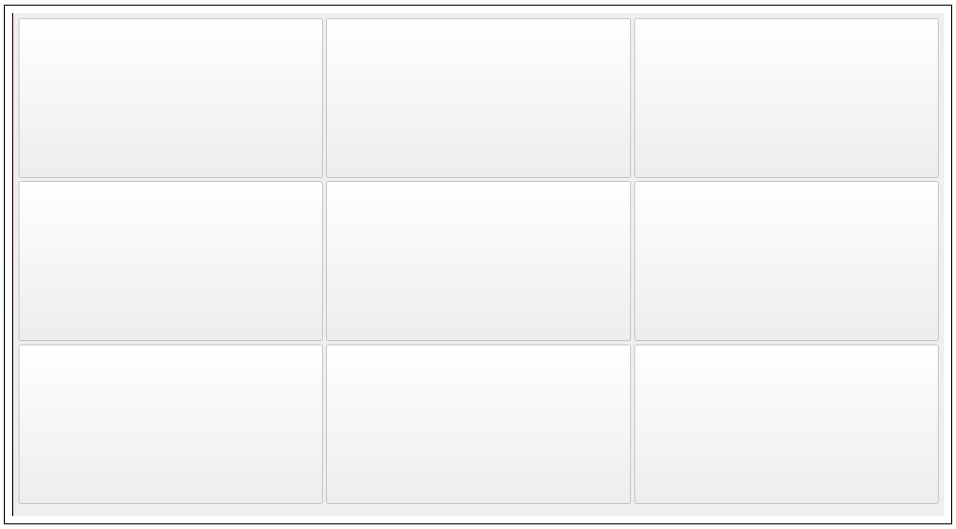
Ensuite nous avons dû décider quelles fonctions étaient essentielles pour le jeu : par exemple la fonction **grille pleine** nous indique si toutes les cases sont remplies ou non, c'est cette fonction que nous utilisons pour détecter les matchs nuls. Nous savons qu'il est possible de détecter les matchs nuls plus tôt et nous nous sommes demandés comment les détecter plus tôt et s'il y avait une réelle valeur ajoutée avec cette fonctionnalité. Nous avons conclu que non dans la mesure où le tictactoe sert à la validation du modèle pour l'application aux échecs.

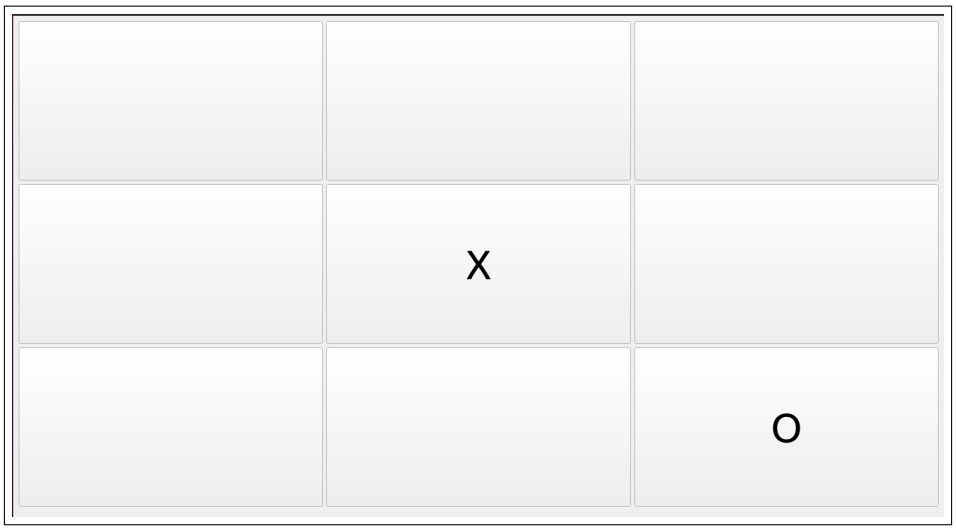
Enfin, ce qui a été le plus difficile pour nous a été d'implémenter les fonctions liées à l'héritage de la classe **position** notamment la fonction **genererposition**. Dans cette fonction l'objet **position tictactoe** est vu soit comme un un objet de la classe mère **position** soit comme un objet de la classe fille **position tictactoe**. Afin de changer la façon dont l'objet est considéré, nous utilisons la fonction dynamiyc_cast<>. Ainsi il est primordiale de savoir exactement à quel objet on fait référence cela nous a posé de grandes difficultés.

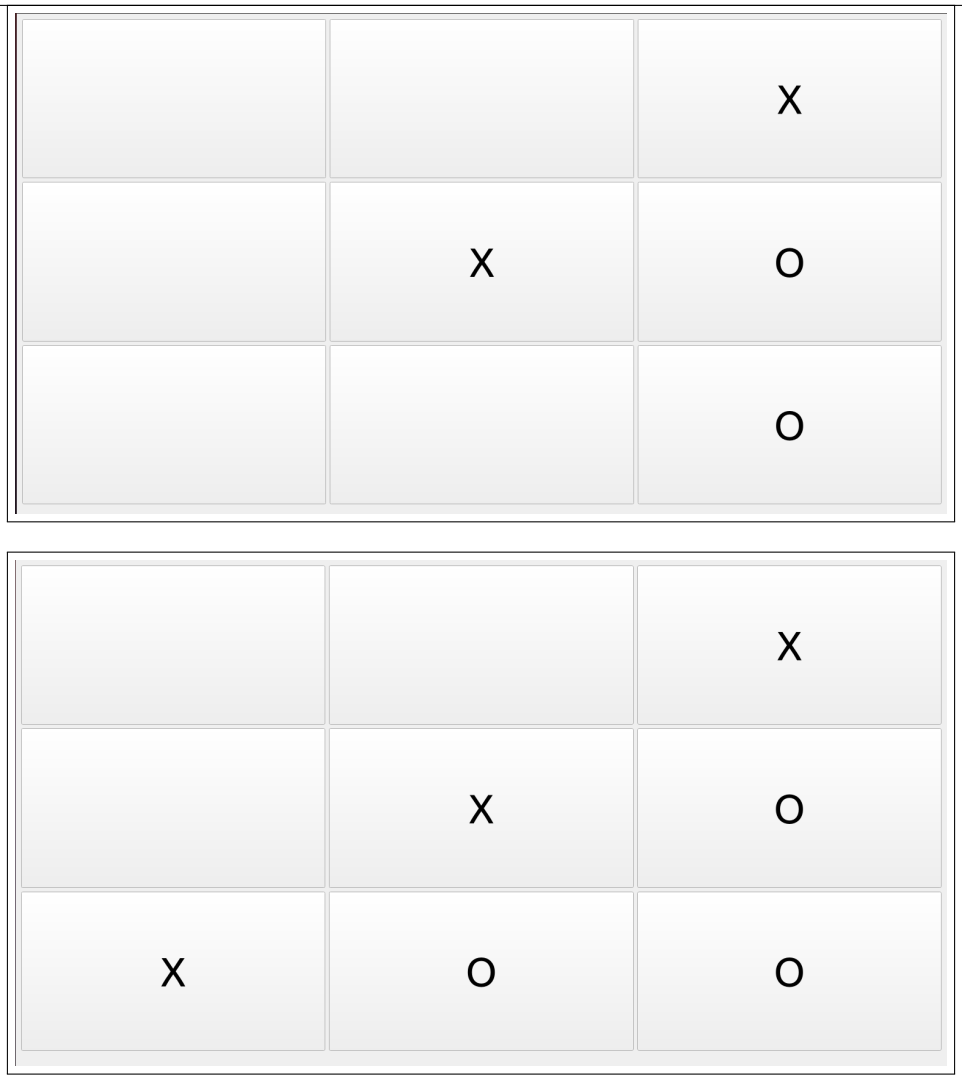
4.1.2 Lancement d'une partie

Pour commencer le jeu de tictactoe, l'utilisateur peut bénéficier de l'interface QT en mettant play sur le fichier projet correspondant.

Une fenêtre apparaît alors sur l'écran :



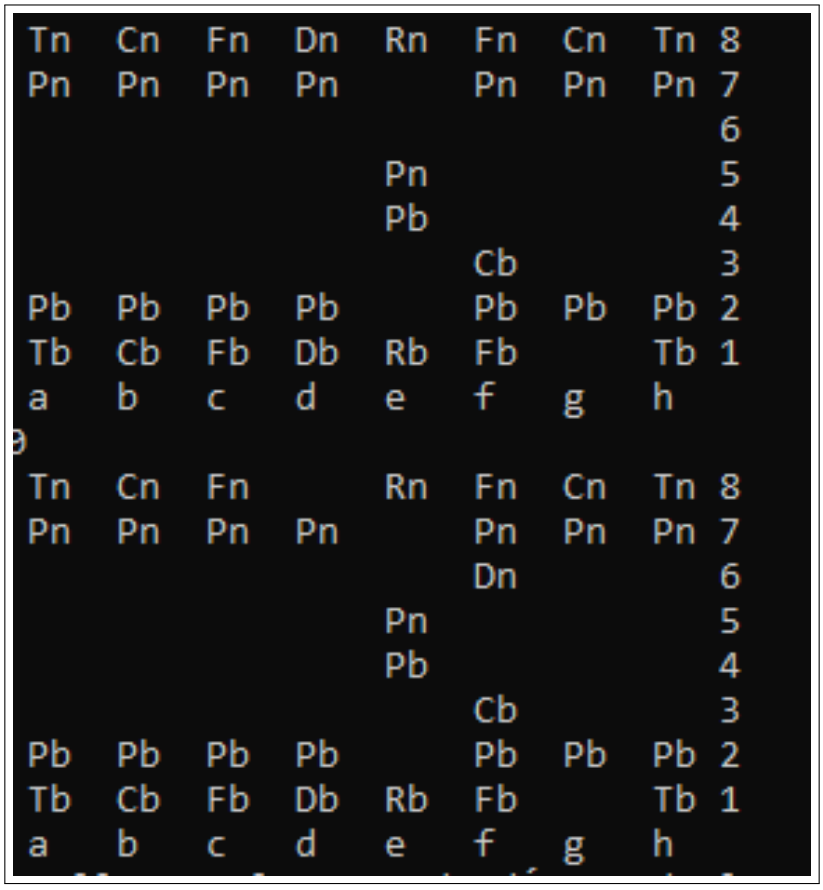




Fin de partie de tictactoe

L'affichage fonctionne en double coup :
pour chaque case jouée par le joueur, c'est à dire cliquée, l'ordinateur joue automatiquement.

4.2 Application aux échecs



Partie en cours : l'ordinateur joue les noirs

La fonction valeur semble relativement cohérente, l'ordinateur cherche à se développer en avançant un pion du centre. La dame jouée est assez originale mais il ne fait rien d'absurde. Il y aurait évidemment beaucoup de méthodes pour améliorer l'heuristique et nous évoquerons quelques idées dans la piste d'amélioration.

5 Pistes d'améliorations

5.1 Algorithmie

5.1.1 Amélioration de la robustesse

Nous avons veillé à vérifier la légalité des coups joués par le joueur, ainsi s'ils rentrent des coordonnées invalides l'ordinateur ne prend pas en compte son entrée et lui redemande de jouer. La plupart du temps si l'entrée n'est pas conforme à ce qui est attendu l'ordinateur ne la prend pas en compte, cependant si le joueur s'amuse à rentrer une chaîne de caractère on peut constater un bugs. Avec plus de temps, nous aurions pu améliorer la robustesse du code de manière à ce qu'il gère tout type d'entrées invalides.

5.1.2 Amélioration de l'heuristique

Nous avons pénalisé les colonnes doublées car elles empêchent un bon développement des pièces. Cependant ce n'est pas le seul paramètre qu'on aurait pu prendre en compte. Par exemple des pions isolés sont souvent très faibles. Du côté offensif des attaques comme la fourchette (attaque de deux pièces différentes) et le clouage auraient pu être plus valorisées.

5.2 Interface Graphique

Bien que nous n'ayons pas vu en cours l'interface QT en C++, notre professeur référent a apporté une aide précieuse tout le long du projet.

La partie graphique a cependant pris du retard car elle ne commençait qu'à partir des premières fonctions créées. D'autres problèmes ont aussi survenu pendant le projet : difficulté de trouver des bons tutoriels pour utiliser l'interface QT, crash d'ordinateurs.

L'interface graphique a été pensée de manière qu'elle soit ergonomique. Nous n'avons cependant pas rajouté des éléments qui auraient rendu le jeu plus agréable pour l'utilisateur. Nous pouvons par exemple nous référer au site d'échec 'chess.com' qui a une interface rendant le jeu agréable. Ce jeu d'échec montre par exemple les coups possible à jouer dès qu'une pièce est séléctionnée. Lorsque l'on joue contre l'ordinateur, il y a une animation entre le moment où le joueur joue et le jeu de l'ordinateur. Le joueur peut aussi pré-sélectionner des coups à jouer. Une horloge est présente dans le jeu et permets une mise en condition réelle pour les tournois d'échec où l'on peut gagner sur le temps : si le joueur adverse a une horloge sans temps restant et les pièces nécessaire à faire échec et math.

6 Annexe

On présente ici un court descriptif des fonctions beaucoup utilisées dans ce projet en complément des commentaires de code.

6.1 Minmax

— **minmax_val(int deep,position* P)** Cette fonction renvoie la valeur d'une position calculée à une profondeur deep définie.

— **min_max()** Renvoie la meilleur position calculée avec un profondeur donnée.

6.2 Tictactoe

— **co_to_int(int i,int j)** Permet de convertir un couple en un indexe. Nous avons codé la grille de jeu comme un tableau statique unidimensionnel.
— **afficher_grille()** cette fonction membre de la classe grille affiche une grille dans la console.
— **jouer(int i,int j,int joueur)** est un fonction membre de la classe grille. Cette fonction vérifie si le coup peut être jouer. Si c'est un coup possible on joue et on renvoie true. Sinon on renvoie false.

6.3 Echecs

— **valeur()** attribue une valeur à une position et la renvoie
— **generepositionfils()** génère toutes les positions possibles à partir d'une position donnée
— **isaPiece()** vérifie si la case est occupée par une pièce et renvoie un booleen
— **control()** renvoie le nombre de cases contrôlées par une pièce
— **onBoard()** vérifie si la pièce est bien sur l'échiquier (lorsque l'on demande un certain déplacement par exemple)
— **isanAlly()** pour une pièce donnée vérifie si une pièce indiquée est de la même couleur que la pièce choisie