Rapport de projet

Jeu d'Echecs

Faveraux Adrien – Lechalier Jc

Sommaire

1. Analyse des besoins

a. Besoins lié au jeu

Nous devons réaliser une application de jeu d'échec. Elle doit permettre de jouer une partie, soit à deux soit à un joueur (contre un ordinateur).

Dans le jeu d'Echecs, il nous faut savoir quelles sont toutes les possibilités de mouvements à toute étape du jeu. À savoir, il faut connaître la pièce que l'on souhaite déplacer ainsi que les déplacements propres à cette dernière.

Il y a que les pièces suivantes :

→ pion : il peut se déplacer d'une case en avant, mange les autres pièces sur le côté.

Il peut se déplacer de deux cases en avant lors de son premier déplacement.

→ tour : elle peut se déplacer uniquement en vertical ou en uniquement en horizontal.

→ cavalier : il peut se déplacer de une case avant/arrière/droite/gauche et une case sur un coté suivant.

→ fou : il se déplace seulement en diagonale.

→ reine : possède le déplacement du fou et de la tour.

→ roi : peut se déplacer vers n'importe qu'elle case voisine.

De plus dans le jeu d'Echecs, il y a, à des moments précis, des coups qui sont rendu possible.

Ce sont des coups spéciaux et ils sont au nombre de trois :

→ Le pion peut, dans certain cas prendre une pièce qui est juste à côté de lui en se déplaçant en diagonale du côté de la pièce à prendre.
Ce coup n'est possible que sur la ligne 4 pour le pion noir et sur la ligne 5 pour les pions blancs. Ce coup spécial s'appelle la prise en passant.

→ Le pion, quand il est arrivé à l'autre bout de l'échiquier, il peut se transformer en n'importe quelle pièce du jeu. C'est la promotion.

→ Le roque peut se faire dans certaine condition, à savoir, qu'entre la tour et le roi il n'y ai pas de pièce et que là où va aller le roi, il ne sera pas en échec.
Ainsi le roi peut se déplacer de deux case sur le côté et la tour se place de l'autre côté du roi.

Lors du déroulement du jeu, à chaque coup, on doit être en mesure de savoir si le jeu est fini.

La partie peut être finie pour plusieurs raisons :

→ Soit on a dépassé le nombre maximum de coups de la partie.

→ Soit le roi un roi est en échec et ne peut plus se déplacer : pat

→ Soit il n'y a plus assez de pièces pour terminer la partie pour bloquer le roi adverse.

Nous devons être en mesure de reconnaître ces différents cas pour le jeu d'Echecs.

Mais nous avons aussi des besoins en termes de rendu de l'application.

b. Besoins application

Nous avons besoins de créer une application du jeu d'Echecs. Pour cela nous avons deux options :

→ Soit nous faisons ce jeu sans mode graphique. C'est à dire que le joueur sera limité au niveau des possibilités de jeu (options, proposition des déplacements possibles). De plus le jeu ne sera pas intuitif.

→ Soit nous faisons un jeu dans une fenêtre graphique qui permettra de visualiser directement le jeu avec des images des différentes pièces du jeu. La possibilité d'ajouter des options telles que l'activation/désactivation de certains paramètres par défaut.

Pour cela nous avons donc utilisé l'environnement de développement Qt. Cela nous permet de générer des interfaces graphiques simplement grâce à des objets graphiques.

Pour gérer les déplacements il nous fallait une méthode qui soit rapide et facilement utilisable dans tout type de contexte on avait plusieurs choix :

→ Soit faire une classe pour chaque pièce et simuler le terrain à partir d'une liste de pièces. Concrètement nous n'avons pas un espace mémoire par case de l'échiquier mais simplement des coordonnées de pièces accessible à travers leur classe respective. Dans ce cas-là, on aura au maximum 32 classes en mémoire chacune d'elle représentant une pièce.

→ On Pouvait aussi utiliser 3 matrices 8*8 qui représente tous les aspects du jeu,

- une pour savoir quelle est le type de pièces qui est sur la case,
- une autre matrice pour savoir à quel groupe appartient la pièce (couleur du joueur),
- et une autre matrice pour savoir si la pièce a déjà été déplacée (utiliser pour les coups spéciaux).

Avec ces matrices on peut, sans créer de classe spécifique pour chaque pièce, réussir à simuler entièrement le jeu d'échecs.

→ Il existe aussi une méthode hybride qui consisterai à créer une matrice « conteneur » ou l'élément à l'intérieur de la matrice serai un parent de chaque classe de pièce, ensuite placer chaque pièce au bon endroit dans le conteneur et déplacer les classes créées sur la grille en profitant de la relation d'héritage.

Pour le jeu nous avons choisis de prendre la solution numéro deux avec les 3 matrice 8*8 car plus simple à visualiser le jeu d'échec, de plus je trouve plus facile de manipuler des matrices pour faire l'IA qui requiers de faire beaucoup de déplacements sur des profondeurs parfois grande.

2. Spécifications techniques

a. Organisation des classes

Definement.h :

Enumération des types de pièces dans l'échiquier.

```
enum piece { Pion,Rois, Reine,Fous,Tour,Cavalier,Rien};
```

terrain.h :

Organisation du terrain en temps qu'objet

```
// Constructeur appelle initialisation
explicit terrain(QWidget *parent = 0);
// Destructeur → supprime toutes les matrices associé a la classe
~terrain();
/* Fixe le nombre maximum de coups par défaut, le numéro du joueur a qui
s'est le tour, c'est une partie à deux joueur physique, initialise l'échiquier
avec les images et les pièces à leur places respectives ainsi que les matrices
utiles pour le jeu.*/
void initialisation();
/* Affiche les possibilités de mouvements de la pièce sélectionnée et
fait le déplacement si il est possible.*/
void caseCliquer(int x, int y);
// Active l'ia
void setIA(bool actif ,int difficulte = 2) ;
// Retourne la case_terrain sélectionnée
case_terrain * getCaseTerrain(int x,int y);
// Simulation du joueur au moyen de l'ia
void faireJouerIA();
// Enlève le halo jaune pour toutes les cases de l'échiquier
void deselectAllCaseTerain();
// Redessine la case sélectionnée (couleur et pièce)
void rafraichirCase(int x,int y);
// Retourne le nombre maximum de tour de la partie
int getNbTourMax();
// Change le nombre de tour maximum de la partie
void setNbTourMax(int nb);
// Retourne vrai si la partie est a un joueur
bool getIAActiver();
// Retourne la valeur de la difficulté de l'ia de la partie en cours.
int getIADifficulter();
/* Fixe le nombre maximum de coups par défaut, le numéro du joueur a qui
s'est le tour, c'est une partie à deux joueur physique, initialise l'échiquier
avec les images et les pièces à leur places respectives ainsi que les matrices
utiles pour le jeu.*/
void reinitialisation()
```

case_terrain.h :

Option possible sur la case choisit en fonction de son type

```
    /*Initialise la taille du bouton et à une position, type, couleur par défaut.*/
    case_terrain(QLabel *parent = 0);
    // Affecte le l'échiquier auquel elle se rapporte
    void setTerrain(terrain * mon_terrain);
    // Emet un signal si le click gauche est fait sur la case.
    void mousePressEvent ( QMouseEvent * event );
    // Redessine la case avec la nouvelle taille de terrain.
    void resizeEvent ( QResizeEvent * event );
    // Défini la couleur de la case.
    void setCouleur(int couleur);
    // Ajoute le background de sélection/coups possibles (sélection jaune)
    void deselect();
    // Enlève le background de sélection /coups possibles
    void select();
    // Met à jour la position de la case en fonction de l'échiquier.
    void setPosition(int x,int y);
    // Met à jour la pièce qui se trouve ou pas sur la case.
    void setPiece(piece ma_piece,int groupe);
    // Redessine la case avec en plus l'image passé en paramètre.
    QPixmap superpositionImage(QPixmap base, QPixmap overlay);
```

deplacements.h :

Ensemble des mouvements possible en fonction du type de pièce + coup spéciaux

```
    // Teste si le mouvement spécial est possible
    static bool petitRoguePossible
    // Teste si le mouvement spécial est possible
    static bool grandRoguePossible
    static QVector<QPoint> deplacementRoi
    static QVector<QPoint> deplacementReine
    static QVector<QPoint> deplacementFou
    static QVector<QPoint> deplacementCavalier
    static QVector<QPoint> deplacementTour
    static QVector<QPoint> attaquePion
    static QVector<QPoint> deplacementPion
    /* Appelle la bonne fonction de deplacement en fonction de la piece sélectionée.*/
    static QVector<QPoint> deplacement
    // teste si le joueur est en echec.
    static bool estEnEchec
```

fenetre option.h :

Organisation de la fenêtre d'option du jeu

ia.h :

Gestion de la partie contre un ordinateur

```
// Initialise l'ia en fonction
IA
QVector<QPoint> calc_echec_et_mat
QVector<QPoint> jouer(int joueur);
void setDifficulter(int diff);
int eval
bool gagnantJeux
int max
int min
```

main.h :

Appelle la fenêtre principale du jeu (MainWindows).

mainwindows.h :

Organisation et gestion de la fenêtre principale du jeu (échiquier + menu)

```
// Créer un nouveau terrain et appelle initialisation.
MainWindow(QWidget *parent = 0);
// Initialise une nouvelle partie par défaut.
void lancerNouvellePartie();
// Affiche la fenêtre d'option.
void afficherOption();
// Création du menu.
void initialisation();
// Met à jour l'affichage de la liste des pièces prises
void updateaffichageall();
// sauvegarde la partie en cours
void SavPartieFichier();
// restaure la partie de son choix
void RestaurerPartieFichier();
```

Nous avons généré une documentation doxygen pour une vision plus approfondit des méthodes et des classes réalisées.
Pour cela il vous suffit ouvrir une page web du dossier html dans un navigateur web (de préférence index.html), cela vous donnera un meilleur rendu des connexions entre les classes.

3. Organisation graphique du jeu

a. Interface graphique

Pour des raisons de simplicité d'utilisation, nous avions choisi de faire une interface graphique pour le joueur. Ainsi, il peut voir sans difficultés l'échiquier et ses pièces ainsi que paramétrer le jeu en fonction de ses besoins.

Événements à réaliser

Pour réaliser l'interface graphique, il nous a fallu créer un tableau de boutons pour l'échiquier. En effet chaque case doit être cliquable pour effectuer ces actions :

→ Si il y a une pièce et c'est mon tour, afficher les possibilités.

→ Si j'ai déjà sélectionné une pièce et que je clique sur une des possibilités de déplacements je me déplace.

→ Si les cases sont vides, ne rien faire.

→ Si les cases ont une pièce qui ne m'appartient pas, ne rien faire.

Chaque bouton est donc connecté avec un signal (le click gauche de la sourie) pour faire l'action qui lui est propre.

Cela est aussi valable pour toutes les actions que l'on peut faire sur l'interface graphique. La même méthode est donc appliquée pour le menu et les items du menu.

Pour chaque case du terrain qui hérite d'un QLabel il y a une méthode virtuelle à implémenter pour pouvoir gérer le clic sur la case :

```
void mousePressEvent ( QMouseEvent * event );
```

A chaque clic sur une case cette méthodes est appeler, vu que ce n'est pas la case qui gère le clic reçu j'ai donc décidé de créer un nouveau signal qui prend deux paramètres x et y pour que quand je recevrai tous les signaux de clics je sache qui l'a envoyé :

```
void clicked(int x,int y);
```

Et donc dans *mousePressEvent* je fais j'envoie le signal que je viens de créer grâce à :

```
Emit clicked(this->x,this->y);
```

Et enfin je récupère les signaux que je connecte à une fonction qui le traitera directement dans le terrain sans mêler les cases aux clics.

Pour pouvoir connecter un signal et un slot Qt dispose de cette fonction :

```
connect(QAction *elementGraphique,SIGNAL(typeSignal()),this,SLOT(action()));
```

→ elementGraphique : soit un items du menu soit un bouton dans notre cas.

→ TypeSignal : triggered ou cliqued dans notre cas.

→ this : fenêtre graphique parent.

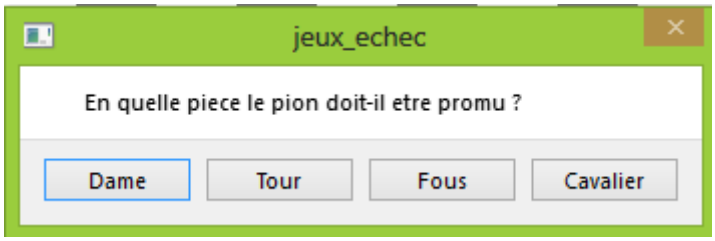
→ Action() : fonction que l'on va appeler lorsque le signa sera émis.

C'est d'ailleurs le même principe qui est appliqué à tous les éléments cliquable car Qt leur a donné un ensemble de signaux par défaut

Promotion

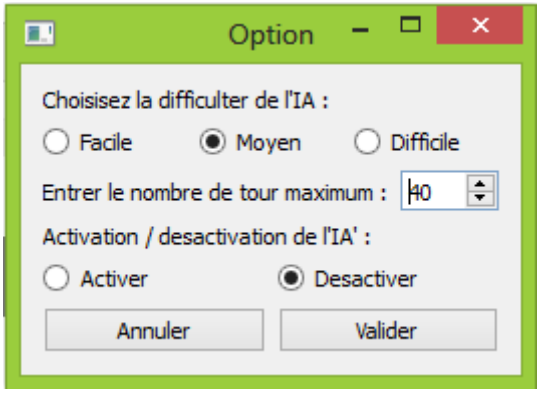
La promotion est un coup spécial que l'on doit faire quand un pion est arrivé au bout de l'échiquier.

Pour cela nous avons choisi de faire en sorte que quand ce cas est vérifié, il y a une boite de dialogue qui apparaisse et demande au joueur en quelle pièce doit être promu son pion.



Fenêtre Option

Cette fenêtre permet au joueur de choisir ses paramètres de jeu de façon simple. Il peut donc choisir si la partie en cours est avec un ou deux joueurs. Une fois validée les paramètres seront actifs pour la partie que l'on va démarrer.



Gestion des Images

Le terrain du jeu d'échec est constitué de QLabel organisé avec un QGridLayout qui leur permet de faire l'effet d'un damier, par défaut un QLabel a été créé pour afficher du texte ou éventuellement une image.

On a donc fait un jeu d'image, des images pour représenter un damier c'est-à-dire une image avec un fond blanc et noir, puis chaque pièce du jeux en noir et blanc avec un fond transparent, on a aussi fait une image qui a un effet orange autour de la case pour faire la sélection d'une pièce et la prévisualisation des coups possibles pour la pièce sélectionnée.

Avec toutes ces images le principe d'affiche est simple, on définit si la case est noir ou blanche et on lui charge l'image correspondante celle-ci restera en mémoire pour pouvoir effacer la case.

Ensuite quand le terrain indique à la case de faire apparaitre une pièce sur une case, la case charge le pion et grâce à une fonction de fusion d'image va afficher l'image du damier avec au-dessus l'image de la pièce.

Le même principe est appliqué si le terrain demande à une case de se sélectionner.

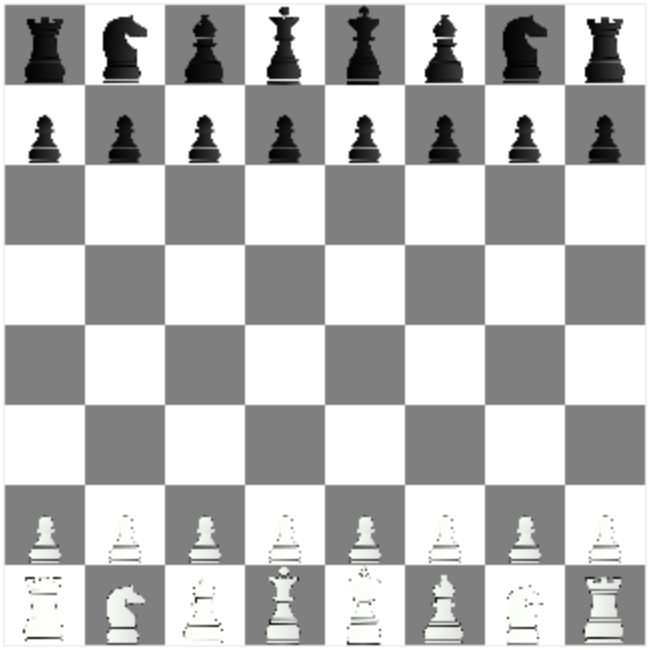
La fonction de fusion d'image fonctionne comme suit :

- → On prend la taille de l'image de base
- → On redimensionne la seconde image pour quelle colle à la taille de la première
- → On crée une nouvelle image de la taille finale
- → On peint la première image puis la seconde par-dessus en forçant la transparence
- → On renvoie l'image fraichement crée

Fonction pour superposer les images :

```
QPixmap case_terrain::superpositionImage(QPixmap base, QPixmap overlay)
{
    int width = base.width();
    int height = base.height();
    if(width != 0 && height != 0 )
    {
        overlay = overlay.scaled(width,height);
        QPixmap result(width,height);
        result.fill(Qt::transparent); // force alpha channel
        {
            QPainter painter(&result);
            painter.drawPixmap(QPoint(0, 0), base);
            painter.drawPixmap(QPoint(0, 0), overlay);
            painter.end();
        }
        return result;
    }
    else
    {
        return QPixmap();
    }
}
```

Résultat Final de l'affichage



b. Fonctionnalités du projet

Voici les fonctionnalités mise en place pour le projet :

→ Jouer contre un ordinateur.

→ Jouer à deux joueurs.

→ Barre de score.

→ Fenêtre option pour configurer les paramètres du jeu.

→ Réinitialisation de la partie.

→ Sauvegarder/Restaurer une partie.

Sauvegarde/restauration de partie

Dans notre jeu d'Echecs, nous pouvons sauvegarder une partie à tout moment. Pour cela il nous a fallu sauvegarder l'ensemble des données importantes pour la restauration de cette dernière à un moment ultérieur.

Pour cela nous avons créé les méthodes suivantes.

```
// Met à jour l'affichage du temps
void update_temp_jeux();
// Sauvegarde la partie dans un fichier au choix du joueur
void SavPartieFichier();
// Restaure une partie sauvegardée
void RestaurerPartieFichier();
// Ajoute la pièce au joueur
void addtoScore(enum piece type, int joueur);
// Mets à jour l'affichage des pieces prises
void updateaffichageall();
// Ajoute le nombre de pièce prises pour le joueur
void setScore(enum piece type,int nb, int joueur);
// Retourne le temps restant pour la partie
int getTempsrestant();
// Met à jour le temps accordé à la partie
void setTempsrestant(int temps);
// Lit le fichier de sauvegarde et remplace son contenu à la partie courante
void loadFile(QString fichier);
// Enregistre dans le fichier les paramètres de la partie en cours
void saveToFile(QString fichier);
```

Elles permettent d'enregistrer dans un fichier :

- Numéro du joueur qui joue
- Nombre maximum de tour de la partie
- Numéro du tour en cours
- Si c'est une partie à un ou deux joueurs
- Temps restant
- Liste des pièces restantes

Lors de la restauration d'une partie, ces variables sont restaurés dans la partie courante. Mais il faut dans un premier temps mettre à jour les variables enregistrées du fichier ainsi que calculer les pièces prises.

Une fois que les variables ont les valeurs d'avant la sauvegarde, il faut mettre à jour l'ensemble de l'affichage. Pour cela il faut redessiner les cases de l'échiquier avec les bonnes pièces au bon endroit. Ensuite il faut mettre à jour l'affichage du temps restant et de la liste des pièces restantes pour chaques joueurs.

Intelligence artificielle

Afin de pouvoir jouer une partie seul contre l'ordinateur, il faut pour cela développer une fonction qui va simuler le comportement d'un joueur physique.

Pour cela nous avons développé plusieurs fonctions qui calculent l'ensemble des possibilités qui peuvent être jouées.

De plus en fonction de la difficulté choisit pour la partie, l'ordinateur simule les coups que pourra jouer le joueur adverse et cela jusqu'à trois coups à l'avance pour la difficulté la plus haute.

C'est l'algorithme MIN-MAX.

Détail de l'algorithme MIN-MAX appliqué au jeu d'Echec :

1ère étape :

La première étape de l'IA qui est la fonction *jouer* dans IA.h cette fonction va commencer par récupérer les déplacements de chaque pièce de son équipe sur le terrain pour savoir où se déplacer.

Ensuite il va faire le premier déplacement (déplacements sur les matrices) après ce premier déplacement il va appeler la fonction *min* pour faire jouer le joueur adverse, la valeur que renverra min correspond au minimum que pourra perdre le joueur ennemi (si il jouait le mieux possible).

2ème étape :

La deuxième étape est exécutée par la fonction min car elle aussi va récupérer tous les déplacements par le joueur ennemi et elle aussi va jouer déplacement après déplacement chaque pièce sauf que elle, après chaque déplacement va exécuter max qui lui permettra de récupérer le score maximum que pourra gagner l'IA si on déplaçait la pièce.

3ème étape

La troisième étape est exécutée par la fonction max car elle aussi va récupérer tous les déplacements par le joueur ennemie et elle aussi va jouer déplacement après déplacement chaque pièce sauf que elle, après chaque déplacement va exécuter min qui lui permettra de récupérer le score minimum que pourra gagner l'IA si on déplaçait la pièce.

4ème étape :

Comme on peut le voir l'étape 2 et 3 s'exécute sans fin, la quatrième étape est la fin d'une de ces deux étape. On considère qu'on renvoie simplement le score quand :

→ On a atteint la profondeur maximum, on ne peut donc pas aller plus loin, on calcule et on renvoie le score.

On a mis un des rois en échec et mat si c'est le roi de l'IA qui gagne le score sera très élevé si il perd il sera très négatif, étant en échec et mat même si on n'est pas à la profondeur maximum, on revoie le score.

5ème étape :

Lorsque l'étape 2 ou 3 reçois un résultat, alors il doit remettre la pièce qu'il était en train de déplacer à sa place et essaye d'en déplacer un autre et redemander le min ou le max en fonction de l'étape.

A chaque déplacement calculer :

Pour min :

On renvoie le plus petit des scores reçus

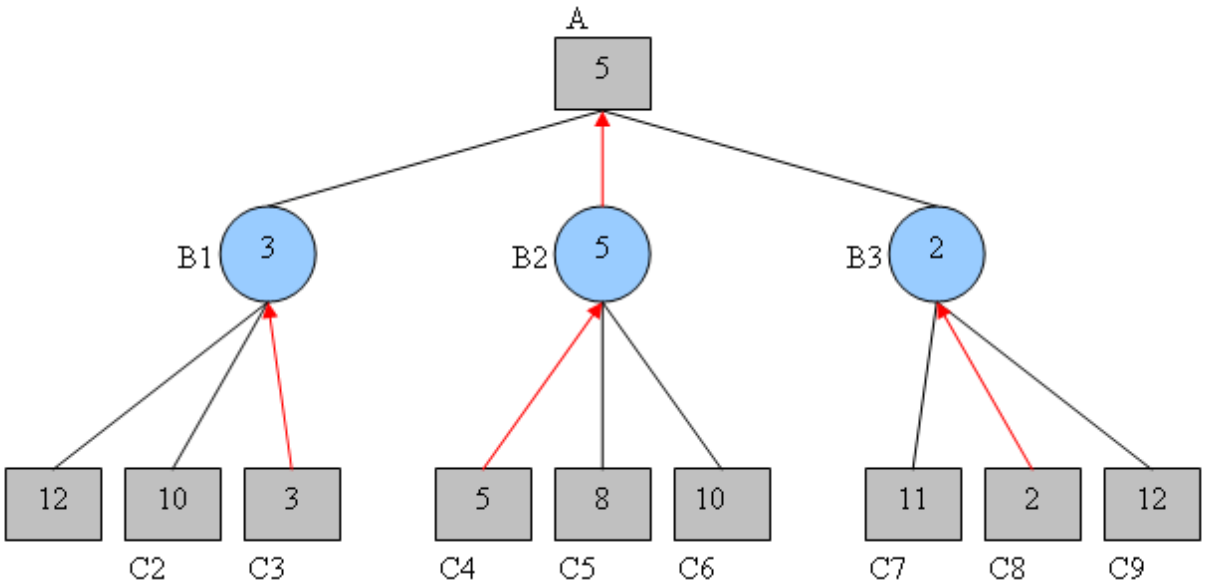
Pour max :

On renvoie le plus grand des scores reçus.

6ème étape :

De retour dans la fonction jouer, on prend les déplacements qui rapporte un maximum de point seulement et si il y en a plusieurs on les place tous dans la liste des meilleurs déplacements qu'on va renvoyer, quand on déplacera vraiment l'ia dans le terrain nous choisirons un déplacement aléatoire parmi les meilleurs.

Représentation graphique de l'algorithme MIN-MAX :



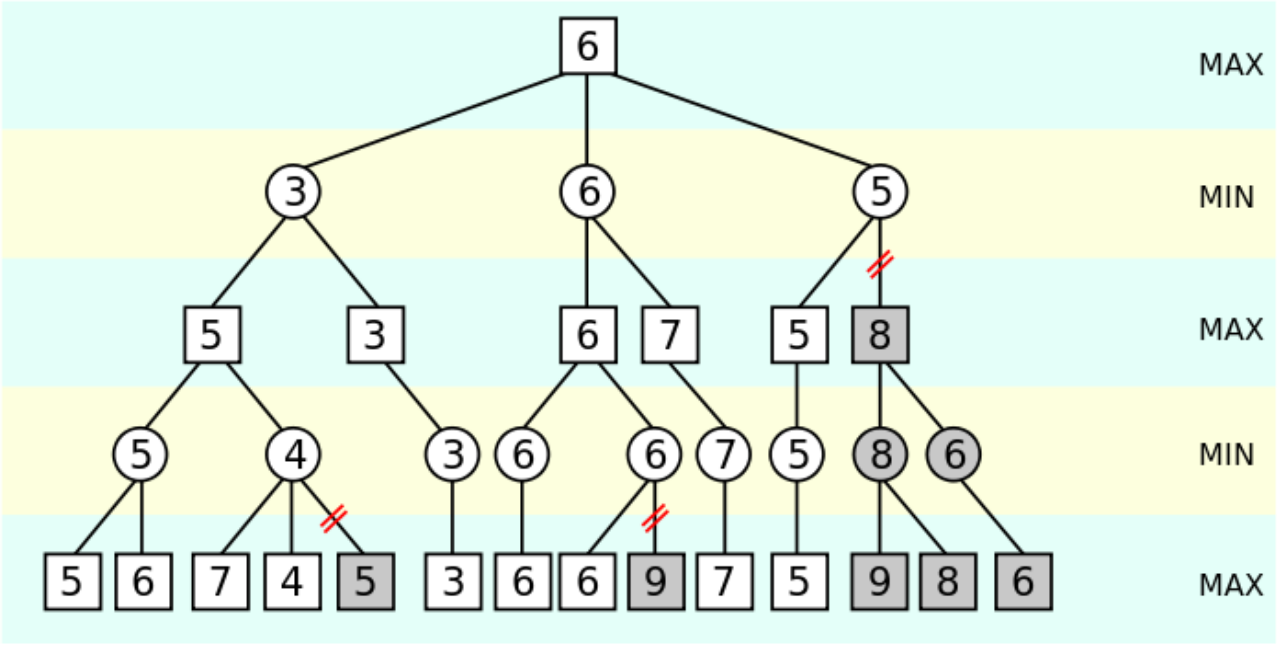
Elagage alpha –beta (optimisation de performance)

L'algorithme MIN-MAX nous fournissait un IA basique mais extrêmement long au vu du nombre de branches, le but de l'élagage alpha beta est de réduire le nombre de branche calculée.

Pour couper les branches de l'arbre il y a deux types de coupure alpha et beta :

- → La coupure alpha se produit quand on calcule le premier fils d'un nœud min, on récupère la valeur du premier déplacement, et le nœud min vaudra au plus cette valeur on peut donc couper toute les branches ou l'on s'aperçoit qu'elle dépasse cette valeur bien sûr on met à jour le min du nœud si un déplacement vaut encore moins.
- → La coupure beta se produit quand on calcule le premier fils d'un nœud max, on récupère la valeur du premier déplacement, et le nœud max vaudra au moins cette valeur on peut donc couper toute les branches ou l'on s'aperçoit qu'elles sont en dessous de cette valeur. Bien sur la valeur max est actualisée à chaque déplacement qui est encore plus le maximum.

Représentation graphique de l'algorithme MIN-MAX avec élagage alpha-beta :



Héritage

Nous avons décidé de ne pas faire de polymorphisme à proprement parler en créant directement des classes mères et filles.

Nous faisons simplement des héritages de classes mères déjà existantes :

→ héritage de QLabel pour ce qui s'agit des case_terrain

→ héritage de QWidget pour la fenêtre d'option et le terrain

→ héritage de QMainWindow pour la fenêtre principal.

→ héritage de QObject pour l'ia.

Ces objets parents sont des interfaces graphiques propres à notre environnement de développement, à savoir Qt.

Nous avons choisi de gérer les pièces seulement avec une énumération.

Pour gérer l'échiquier nous avons choisi d'utiliser des matrices. Ces matrices permettent de savoir où sont les pièces sur l'échiquier et savoir à quelle joueur elles appartiennent.

Lors d'un déplacement, on ne fait que changer la valeur de l'énumération. Alors que si on avait choisi de faire une matrice d'objet de type Piece, il faudrait recharger l'ensemble de l'objet. C'est beaucoup plus lourd dans la mémoire. C'est donc, dans un premier temps, dans un souci de rapidité que nous avons choisi de ne pas faire de polymorphisme pour les pièces du jeu.

c. Gestion des contenus

À tout moment il faut savoir à quel joueur appartient tel ou tel pièces, à quel endroit ces dernières se trouvent.

Pour cela nous avons mis en place des matrices de pièces, ainsi que des matrices de groupe afin de savoir à quel joueur appartient la pièce. De plus nous avons mis en place des matrices afin de savoir quelle pièce a bouger et lesquelles n'ont pas bougé. Cela nous sert notamment dans la gestion des coups spéciaux.

Dans la gestion de l'ia nous avons mis en place des matrices de précédence. Elles servent notamment pour l'ia de niveau supérieur. En effet, elle prévoit des coups à l' avance. Donc il faut être ne mesure de remettre les pièces en place après le traitement.

Template

Quand un joueur sélectionne une pièce dans l'intention il faut qu'il sache quelles sont les possibilités de la pièce sélectionnée. C'est pourquoi nous devront être en mesure de connaître tous les mouvements de la pièce. Pour cela nous avons décidé d'utiliser des QVector<>.

Nous avons choisi d'utiliser des templates QVector car nous n'avons pas à gérer la gestion pour l'ajout de variables. En effet si nous avions choisi de faire un tableau, la méthode ajouter n'existe pas. De plus il faudrait dans le meilleur des cas savoir la taille du tableau que l'on veut faire. Or le nombre de coups possible ne peut pas être connu à l'avance. Enfin, la gestion mémoire est meilleure que pour un tableau.

Les Qvectors sont là aussi des templates propre à Qt, nous aurions pu utiliser des std::vector qui sont dans la librairie standard de C++ mais les méthodes d'ajouts sont assez limités.

Les QVector nous permettent d'ajouter facilement des éléments afin de les traiter ultérieurement. De plus comme il s'agit d'un template, nous pouvons ajouter n'importe quel type de donnée dans le vecteur.

Dans notre cas, il s'agit de faire une liste de toutes les cases ou peut aller la pièce sélectionnée. C'est pourquoi nous avons choisi d'utiliser des QPoint. Ils permettent d'avoir rapidement les coordonnées de la case.

Ainsi, toutes les fonctions qui listent les possibilités de déplacements d'une pièce sont des QVector<QPoint>

4. Évolutions possibles

Nous pouvons améliorer la structure de l'application en faisant par exemple du polymorphisme pour la gestion des pièces.

Pour cela nous feront une classe mère qui aurait une méthode déplacement qui serait implémenté par ses classes filles (qui ne seront rien d'autre que l'ensemble des types de pièces). De ce fait quand on appellera la méthode déplacement, le système sera quelle est la bonne fonction à appeler en fonction de son type.

Nous pourrons aussi permettre de faire un ou des retours en arrière grâce à une liste de déplacement dans la matrice de coups précédent. Cela ne sera possible que pour le mode 1 joueur, car avec deux joueurs, cela n'aura aucun intérêt.

Sur le même principe, nous pourront faire en sorte de pouvoir changer la durée maximum de la partie.

Pour la visualisation de l'application, nous pouvons aussi noté que la gestion des images peut être réalisée avec des images vectorielles et non matricielles. Cela nous offrirai un meilleur rendu au niveau du lissage de l'image des pièces sur l'échiquier.

Sur le même principe, nous pouvons faire en sorte que le compteur de pièces prises sera l'image des pièces sur le côté pour un côté plus attractif du jeu.