ENSTA

SIM202

Simulation d'un jeu d'échecs

Axelle DE BRITO Louise LALLEMAND Grégoire GUILLOT

Promotion 2023

Contents

Introduction

Dans le cadre du projet de simulation numérique, nous avons réalisé un jeu d'échecs permettant de jouer contre l'ordinateur, à l'aide d'un algorithme de type min-max. Pour cela, nous avons séparé le travail en deux branches : la modélisation du jeu d'échecs, et la partie algorithme MinMax.

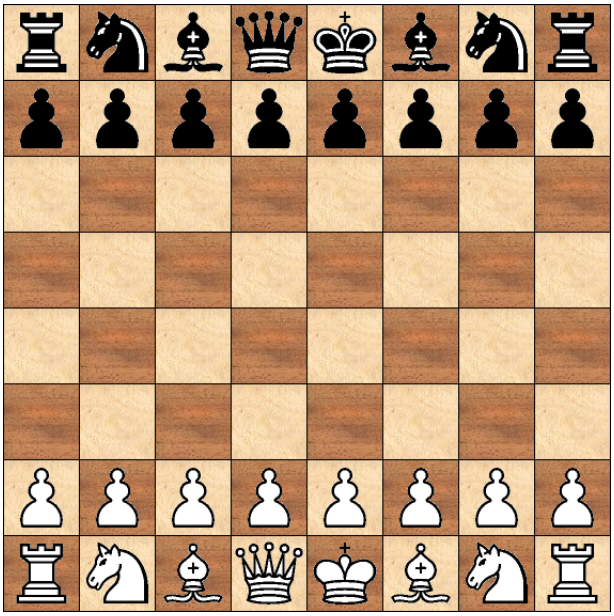
Il s'agissait de coder le jeu d'échecs et toutes ses subtilités, les déplacements des pièces, coups spéciaux et fin de jeu, ainsi que d'implémenter une méthode permettant à la fois de calculer l'ensemble des coups possibles à chaque tour selon une certaine profondeur, et de sélectionner le meilleur selon le critère MinMax.

Nous présentons ici d'abord l'objectif de notre projet et la définition du problème, puis nous nous intéressons à la modélisation du jeu d'échecs, à l'algorithme MinMax, avant de fournir les résultats de notre étude, appliquée un jeu de morpion puis au jeu d'échecs.

Chapter 1

Présentation du problème

Le jeu d'échecs, bien connu, est un jeu de plateau 8x8 qui se joue à deux. Les deux joueurs peuvent être soit deux humains, soit deux ordinateurs (avec une certaine stratégie chacun) soit un humain contre un ordinateur. Notre simulation permettra d'exécuter ces différents cas. Concernant l'ordinateur, tout son intérêt réside dans la stratégie adoptée. Nous essayerons d'obtenir un ordinateur avec une stratégie correcte, en tout cas meilleure qu'un ordinateur jouant des coups aléatoires.



L'ordinateur dispose donc au départ de 16 pièces qu'il doit déplacer sur l'échiquier. Il doit savoir spécifier les pièces, connaître leurs déplacements possibles, ainsi que les règles de prises de pièces. Un outil permettant de faire un coup doit aussi est mis en place, ainsi qu'une règle pour choisir le meilleur coup possible.

Nous avons donc créé autant de classes d'objets nécessaires à une partie, que nous détaillerons par la suite :

- `TypePiece`

  Qui renseigne le type de la pièce (roi, dame, cavalier, fou, tour ou pion), ainsi que ses déplacements relatifs , et la valeur de la pièce

- `Piece` issue de `TypePiece`

  Qui rajoute la couleur de la pièce ainsi que sa place sur l'échiquier

- `Deplac_rel`

  Qui renseigne un déplacement sous la forme d'une pair de deux coordonnées (entières)

- `Coup`

  Qui contient l'entièreté des informations concernant un coup joué par un joueur : la pièce jouée, sa position d'arrivée, ...

- `CoupSpecial`

  Qui est un énumérateur valant NonSpecial ou le nom du coup special (Roque, promotion ou prise en passant)

- `Echiquier`

  Qui contient notamment le tableau 8x8 avec comme chaque case un pointeur vers une Piece (valant NULL si la case est vide)

- `Position`

  Qui contient la configuration du jeu, et d'autres objets nécessaires à l'algorithme MinMax

Chapter 2

Modélisation des échecs

Afin de commencer notre résolution du problème, nous avons du réfléchir à l'architecture globale de notre jeu. Comme le but final de notre projet est d'obtenir un min max fonctionnel, nous avons voulu créer un jeu de morpion en même temps que notre jeu d'échecs, car la simplicité de ce jeu nous permet de tester beaucoup plus simplement notre ordinateur utilisant MinMax. Nous avons donc commencé par créer l'entièreté des outils utiles permettant aux deux différents jeux d'exister. Afin de nous faciliter la tâche, la plupart des fonctions et des classes sont communes aux deux jeux. Nous avons pu faire cela grâce à l'utilisation de "if" sur la taille de l'échiquier (3 ou 8) ou a des fonctions de conversions adaptées (par exemple, une fonction prenant une pair d'entiers (ligne et colonne) allant de 1 à 3 ou de 1 à 8 et renvoyant une place entière dans un tableau unidimensionnelle 3*3 ou 8*8 (cette dite place variant entre 0 et 8 ou 0 et 63)) ou en utilisant des arguments par défaut lorsque les attributs n'étaient pertinents que pour le jeu d'échecs. Afin de présenter le fonctionnement global du jeu, nous allons détaillons pas à pas les différentes classes et fonctions implémentées.

2.1 Modélisation des objets utiles à une partie

Commençons par définir les différents objets utiles à une partie.

2.1.1 Échiquier

Pièce centrale de notre jeu, la classe `Echiquier` modélise le plateau de jeu et ses pièces et comprend :

- `taille`

  Un entier désignant la taille du plateau (8 pour les échecs, 3 pour le morpion)

- `plateau`

  Un tableau de pointeurs sur des objets de type `Piece`

- `roi_noir`

  Un pointeur sur le roi noir. Cela nous évite d'avoir à le rechercher sur l'entièreté du plateau lorsqu'on en a besoin et il est utiles dans de nombreuses fonctions (pour calculer si un coup est légal, s'il met en échecs l'adversaire...)

- `roi_blanc`

  Un pointeur sur le roi blanc, qui a le même intérêt que pour le roi noir.

- `L_coup_depuis_dep`

  Une `list<Coup>` stockant tous les coups depuis le début de la partie, pour arriver à la configuration actuelle. Cela permet d'avoir un historique des coups joués.

Les fonctions internes de la classe sont

- `Echiquier(int n)`

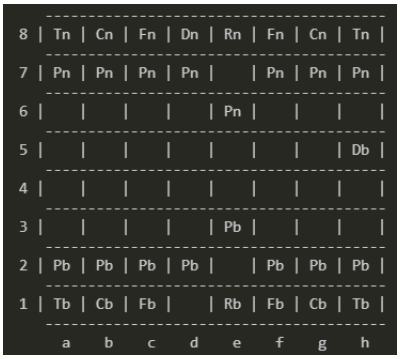
  Le constructeur de l'Echiquier prenant en entrée sa taille et allouant la mémoire nécessaire à son utilisation.

- `~Echiquier()`

  Le destructeur de la classe permettant de supprimer l'échiquier en fin de partie.

- `affiche()`

  Une fonction qui affiche l'échiquier ainsi que les pièces qu'il contient dans le terminal, sous le format suivant :



On dispose aussi de la fonction

- `mise_en_place_echec_piece(Echiquier & Echi)`

  Qui initialise l'échiquier avec toutes les pièces du jeu d'échecs, à leur position initiale (4 lignes de pièces).

2.1.2 Pièce

Puisque nous évoquions les pièces voici leur structure. Pour modéliser une pièce, on utilise une classe `TypePiece` qui renseigne le type de la pièce, et sa classe héritière `Piece` qui ajoute la couleur et ses coordonnées.
Elle est composée de

- `type`

  Un objet de type string qui contient le nom de la pièce : "Pion","Dame",...

- `deplac_relatif`

  Une `list` d'objet de la classe `Deplac_rel` que nous détaillons ensuite, qui contient les déplacement relatifs possibles de la pièce sous forme de paire de coordonnées

- `valeur`

  Un entier qui contient la valeur de la pièce (1 pour un pion, 3 pour un cavalier ou un fou, 5 pour une tour et 9 pour la reine, 100 pour le roi)

Elle contient les fonctions internes

- `TypePiece(string type)`

  Un constructeur de TypePiece à partir du string Type qui crée en plus la liste des déplacements relatifs autorisées

- `~TypePiece()`

  Un destructeur

Ensuite, la pièce hérite des attributs de type pièce et possède en plus :

- `isWhite`

  La couleur de la pièce

- `position_coor` La position de la pièce sous forme d'une paire d'entiers

- `a_bouge` Le nombre de déplacements déjà effectué par la pièce que nous utiliserons pour savoir si roquer est autorisé

Cette classe Piece contient un constructeur par copie et un constructeur par arguments ainsi qu'un destructeur.

2.1.3 Déplacement relatif

Nous avons créé une classe `Deplac_rel` pour formaliser les déplacements relatifs des pièces, afin de faciliter leur usage.
Elle contient

- Un objet `coor`

  Une `pair<int,int>` qui représente le déplacement relatif comme des coordonnées

- Un constructeur `Deplac_rel(pair<int, int> coord)`

- Un deuxième constructeur `Deplac_rel(const Deplac_rel &dep_a_copier)`

2.1.4 Coup

Afin de faciliter notre gestion de l'échiquier et comme nous souhaitons pouvoir revenir en arrière sur l'échiquier, nous avons décidé de créer une classe coup très complète contenant toutes les informations relatives à un coup joué. Il est à noter que ce coup n'est créé que si le déplacement est légal. Elle comprend les attributs

- `isWhite`

  Un booléen qui indique si le joueur joue les blancs

- `pieceJouee`

  Un objet de la classe `Piece` qui représente la pièce jouée par le coup

- `oldPosition`

  Une `pair<int,int>` qui représente l'ancien emplacement de la pièce

- `newPosition`

  Qui représente, elle, la nouvelle position

- `Taken`

  Un pointeur sur la pièce prise, s'il y a prise; un pointeur nul sinon

- `Coup_special`

  Un objet de la classe enum `CoupSpecial` valant soit non special, soit prise en passant, soit promotion, soit petit roque, soit grand roque

- `Type_Promu`

  Un `string` utilisé dans le cas où un pion atteint l'extrémité adverse et valant soit "Pion" si le coup n'est pas une promotion soit le type de la pièce en laquelle le pion se transforme

- `is_echec`

  Un booléen qui indique si le coup joué met en échec le roi ennemi

- `is_mat`

  Un booléen qui indique s'il y a échec et mat sur le roi ennemi du joueur

- `num_tour_de_jeu`

  Un `int` qui indique le numéro du tour

Elle comprend aussi les fonctions internes

- Le constructeur `Coup(bool isW, const Piece &pieceJ, pair<int, int> newP, pair<int, int> oldP, int num_tour_de_jeu, Piece *taken, CoupSpecial coup_special, string type_promu, bool is_echec, bool is_mat)`
- Le second constructeur `Coup(const Coup &coup)`
- Le destructeur `~Coup()`

2.1.5 Coups spéciaux

Aux échecs, il existe des coups spéciaux qui sont des exceptions et doivent être traitées à part. Les différents coups spéciaux traités dans le code sont contenus dans un `enum` : `non_special, petit_roque, grand_roque, promotion, prise_en_passant`. Pour le roque par exemple, le coup n'est autorisé que si ni le roi ni la tour concernée n'ont bougé. Cela justifie la présence d'un attribut booléen `a_bouge` relatif aux Pièces. Il faut de plus vérifier qu'au cours de l'échange, le roi ne se met pas en échec, grâce à la fonction `is_Echec`. Nous avons développé ces coups spéciaux après avoir un jeu d'échecs fonctionnel. Ils ont nécessité une révision presqu'entière du code puisque nous avons dû ajouter des attributs à certains objets, complexifier des fonction existantes comme celle permettant de mettre à jour le plateau de jeu, etc ...

2.2 Déroulement d'une partie

Maintenant que les objets sont définies passons au déroulement d'une partie.

2.2.1 Interface

Afin de nous permettre de tester notre algorithme MinMax, nous avons d'abord développé un joueur aléatoire jouant un coup aléatoire parmi tous les coups possibles. Le développement de ce joueur aléatoire nous a d'abord permis de créer une fonction qui trouve l'entiereté des coups possibles à un instant donné, fonction utile pour le MinMax. Il nous a également permis de tester la robustesse de notre code sans avoir à tester à la main toutes les configurations possibles.

Si une partie d'échecs est lancée, l'utilisateur a à disposition 3 types de joueur : humain, aléatoire et algorithme MinMax. Il peut donc choisir la configuration dans laquelle l'utilisateur souhaite jouer : humain VS humain, humain VS MinMax etc... Il peut également choisir la profondeur de la recherche MinMax. Il peut ensuite choisir d'effectuer un tirage au sort du premier joueur ou non. La partie peut démarrer.

Une partie se déroule dans le terminal, où une interface graphique est implémentée à l'aide de caractères simples ("-" et "|" pour bordure, "Pb" pour pion blanc, "Cn" pour cavalier noir...). Elle permet de visualiser la partie et est donc un élément clé pour tester nos fonctions. Elle permet également au joueur humain d'avoir les informations nécessaires sur le déroulé du jeu.

2.2.2 Déroulé

A chaque fois que c'est au tour de l'humain, il doit choisir la case de départ en la notant dans le terminal, puis la case d'arrivée.

Le programme vérifie ensuite si ce coup est un coup spécial puis s'il est légal. Ces deux fonctions sont assez simples dans leur concept mais leur codage est long puisqu'il faut vérifier de nombreux faits (est-ce que le déplacement est bien dans les déplacements de la pièce jouée,

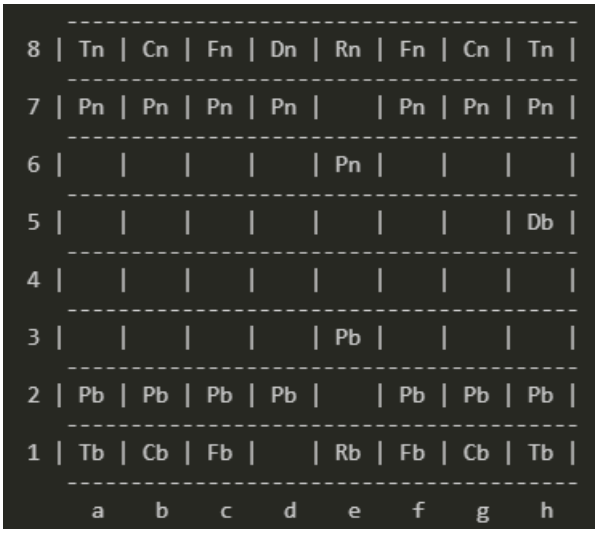


Figure 2.1: Etat de la partie visualisé dans le terminal

est-ce que le roi qui veut roquer n'est pas en échec et est ce qu'il est bien resté immobile tout le début de partie, etc...). La légalité des coups spéciaux (sauf la promotion) se fait dans la fonction coup spécial tandis que la légalité d'un coup "normal" se fait dans la fonction de légalité. Cela signifie que la fonction de légalité est appelé après la fonction coup spécial, prend en argument ce coup spécial et renvoie toujours vrai si le coup spécial vaut autre chose que non spécial ou promotion. Il est à noté que la fonction de légalité ne vérifie pas si un déplacement place le joueur qui l'effectue dans une position d'échec (ce qui est interdit) ; puisque cela nous conduisait à réaliser des boucles sans fin entre la fonction is_échecs et la fonction de légalité. Une fois vérifié la légalité d'un déplacement, nous vérifions donc que le joueur ne place pas son propre roi en échec. Si ces deux conditions sont vérifiées, on passe à la suite, sinon, le joueur doit entrer d'autres cases.

Arrivée à cette étape, nous créons le coup correspondant au déplacement en ajoutant le fait qu'il mette en échec ou en mat le roi ennemi et la pièce qu'il capture s'il en capture une. Nous actualisons alors le plateau et tous ces attributs avec le nouveau coup joué et nous passons à la vérification de fin de partie. Il existe plusieurs cas dans lesquels la partie doit se terminer et nous les développerons dans le paragraphe suivant. Si aucun de ces cas n'est atteint, nous changeons de joueur et relançons le processus si c'est un joueur humain.

Si le joueur est un ordinateur, il suffit d'appeler la fonction coup_min_max ou coup_aleatoire pour obtenir le coup qu'il joue et actualiser le plateau en conséquent.

2.2.3 Fin de partie

Une partie se termine si l'un des deux joueurs a gagné ou s'il y a égalité. Le vérifier demande peu d'efforts pour une partie de TikTakToe, mais c'est une autre histoire pour les échecs...

Un joueur a gagné dès que l'adversaire est mat. Il ne suffit donc pas de vérifier que le roi adversaire est mangé : on appelle la fonction `is_mat`.

Il peut y avoir égalité dans plusieurs cas. Si le joueur ne dispose d'aucun coup légal, et que le roi n'est pas en échec, alors il y a pat et la partie se termine sans vainqueur. De même, s'il y a 50 coups d'affilée sans prise ni pion déplacé, la partie se finit. Enfin, s'il y a une insuffisance matérielle qui empêche un joueur de faire échec et mat (configuration roi VS roi, roi et fou VS roi par exemple).
En revanche, l'égalité d'échec perpétuel n'a pas été codée, car elle est trop rare pour être intéressante à notre échelle, et longue à mettre en oeuvre. Cette égalité a normalement lieu lorsque qu'il y a échec à chaque coup, mais qu'il ne peut y avoir échec et mat.

2.3 Conclusion partielle

Toutes ces modélisations nous ont permis d'obtenir un jeu d'échecs fonctionnel contenant l'entièreté des subtilités de ce jeu ancestral. En parallèle du jeu d'échecs, nous avons codé le jeu du morpion. Sur ces deux jeux, nous avons codé un joueur aléatoire permettant au joueur humain d'affronter un ordinateur (même si ces résultats étaient complètement nuls). Il était désormais temps d'obtenir un ordinateur performant et nous allons utiliser pour cela l'algorithme MinMax.

Chapter 3

Algorithme MinMax

A chaque tour de l'ordinateur, il s'agit maintenant de générer toutes les positions possibles de l'échiquier jusqu'à une certaine profondeur choisie, auxquelles on associe une valeur selon la règle du MinMax.

3.1 Principe de l'algorithme

C'est au tour de l'ordinateur de jouer. Nous avons à disposition la configuration actuelle de l'échiquier, issue d'une classe appelée `Position` qui contient notamment

- L'échiquier à un instant donné
- La liste des coups précédents faisant le lien entre l'échiquier stocké et l'échiquier à jour
- Un pointeur vers la position fille
- Un pointeur vers la position soeur
- La valeur de la position

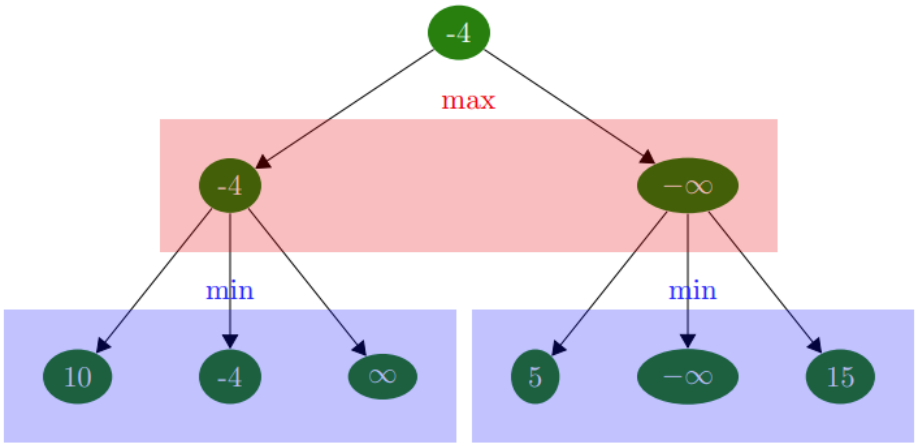
Pour sélectionner un coup à jouer, l'ordinateur génère d'abord l'ensemble des coups possibles, auxquels sont attribués une valeur. Un coup gagnant (dans notre cas, un échec et mat), correspond à ∞, et un coup perdant à $-\infty$. L'ordinateur va jouer le coup qui maximise ses positions et minimise les positions de l'adversaire de la façon suivante :

Si on note $\mathbf{P}$ la position en question, on a alors

$$\texttt{MinMax}(\mathbf{P}) = \begin{cases} \text{Valeur}(\mathbf{P}) & \text{si P est une position terminale} \\ \min(\texttt{MinMax}(\mathbf{P}_1), ..., \texttt{MinMax}(\mathbf{P}_k)) & \text{si P est une position de l'adversaire} \\ \max(\texttt{MinMax}(\mathbf{P}_1), ..., \texttt{MinMax}(\mathbf{P}_k)) & \text{si P est une position de l'ordinateur} \end{cases}$$

Avec $\mathbf{P}_1, ..., \mathbf{P}_k$ les positions filles de $\mathbf{P}$

Si l'emplacement de -4 représente la position initiale de l'ordinateur, on peut représenter l'arbre de profondeur 2 :



Le coup choisi par l'ordinateur sera alors le coup maximisant `MinMax`(P_i) parmi tous les coups possibles (donc toutes les filles directes de la position initiale).

3.2 Classe Position

Nous détaillons dans cette section la classe `Position` en détails. Une position contient des informations diverses comme la configuration de l'échiquier à ce moment, le joueur associé au prochain coup, ou encore la valeur associée à cette configuration.

3.2.1 Attributs

La classe Position contient

- `plateauRef`

  Un pointeur sur un objet de la classe `Echiquier`, qui correspond à la configuration du jeu au début du tour

- `CoupsPrecedents`

  Une liste `list<Coup>` d'objets de la classe `Coup`, qui stocke les coups permettant d'amener l'échiquier de la configuration de référence à la configuration actuelle et inversement grâce aux fonctions `actualisePlateau` et `resetPlateau`. Cela permet de limiter le stockage des échiquiers lors de la génération de l'arbre des possibles.

- `Soeur`

  Un pointeur sur la soeur de la position, de la classe `Position`

- `Fille`

  Un pointeur sur la fille de la position, de la classe `Position`

- `Joueur_current`

  Un booléen qui indique qui est le joueur devant jouer la position

- `Joueur`

  Un booléen qui indique qui est le joueur en cours dans la partie. Il est égal à `Joueur_current` au moment de l'appel de la fonction `generateur`, et reste fixe au cours du calcul du coup à jouer.

- `Num_tour_de_jeu`

  Un compteur du nombre de tours effectués. Il sert notamment au tracé de figures.

3.2.2 Fonctions internes

La classe Position contient les fonctions internes

- `Position (Echiquier *plateau,list<Coup> coups, Position *positionSoeur, Position *positionFille, bool joueurCoup, int num_tour)`

  Un constructeur

- `set_valeur`

  La fonction qui calcule la valeur propre de la position. Elle est appelée sur les feulles de l'arbre des possibles. Cette fonction est détaillée par la suite.

- `generateur`

  La fonction qui crée l'arbre des possibles, c'est-à-dire les positions filles et soeurs accessibles depuis la position de départ par des coups légaux. En les créant, `generateur` attribue les valeurs MinMax aux positions. Cette fonction est détaillée par la suite.

- `Position`

  Un destructeur

3.2.3 Fonctions externes

Nous avons aussi créé des fonctions qui permettent de ne pas stocker un échiquier entier systématiquement, mais seulement une liste de coups. Ainsi

- `actualisePlateau(Echiquier &plateau, const list<Coup> &coupsPrecedents`

  actualise le plateau de référence en parcourant les coups précédents

- `actualisePlateau(Echiquier &plateau, const Coup &coupjoue)`

  actualise le plateau de référence selon le dernier coup : il "joue" le coup et met à jour l'échiquier.

- `resetPlateau(Echiquier &plateau, const list<Coup> &coupsPrecedents)`

  Reviens à un plateau antérieur grâce à une liste de coups

- `resetPlateau(Echiquier &plateau, const Coup &coup_joue)`

  Reviens à un plateau antérieur grâce au dernier coup : il "joue" le coup à l'envers et met à jour l'échiquier.

3.3 Quelques étapes détaillées

3.3.1 Etape 1 - génération des états possibles

Soit P_0 l'état dans lequel se trouve l'ordinateur au début de son tour. Cet état est le sommet à partir duquel est construit l'arbre des états possibles. Cet arbre a une profondeur limite k, fixée à chaque partie, correspondant au nombre de coups qu'anticipera d'ordinateur.

Chaque génération enfant est l'ensemble des états atteignables par l'état mère. Pour les construire, nous utilisons une fonction `coupsPossibles` retournant la liste de tous les coups légaux à partir d'un état.

Dans le cas du TicTacToe, il existe autant de coups légaux que de cases vides. Dans le cas des Echecs, un coup pour être légal doit : concerner une pièce possédée par le joueur, rester dans l'échiquier, arriver sur une case vide ou occupée par l'adversaire, respecter les règles de déplacement de la pièce, ne pas mettre le roi en échec, ne trouver aucune pièce le long du déplacement. Pour le vérifier, on appelle `is_legal` sur chaque coup candidat.

La génération s'arrête verticalement à un noeud P_i si l'une des conditions suivantes est remplie :

- La profondeur limite k est atteinte.
- Il n'existe pas de coup possible partant de la position P_i.
- P_i est une position gagnante ou perdante.

Une fois une feuille atteinte, on lui attribue sa `valeurMinMax` grâce à `set_valeur`.

3.3.2 Etape 2 - évaluation d'un état

Une fois l'arbre construit, il faut attribuer une valeur à chaque noeud. Par souci d'efficacité, cette étape est comprise dans la fonction `generateur`.

Une configuration de la classe `Position` est évaluée en établissant un arbre de coups possibles avec la fonction `generateur`. . A chaque position est associée de manière récursive sa `valeurMinMax`, calculée dans la même fonction. Par défaut, si la position n'a pas de filles, la valeur est calculée dans la fonction `set_valeur` selon le calcul :

$$\gamma_p(\text{valeur(ordi)} - \text{valeur(humain)}) + \gamma_c(\text{cont(ordi)} - \text{cont(humain)})$$

où valeur() est la somme des valeurs des pièces présentes sur l'échiquier du joueur en question, et cont() le nombre de cases contrôlées. γ_p et γ_d sont des entiers choisis au départ. Cas particuliers, elle est égale à ∞ si elle est gagnante et $-\infty$ si elle est perdante.

Dans le cas du TikTakToe, elle vaut simplement $+\infty$ dans le cas d'une position gagnante, $-\infty$ pour une position perdante et 0 sinon. La fonction `set_valeur()` discrimine le TikTakToe des échecs selon la taille de l'échiquier.

Dans la fonction `generation` de la classe `Position`, on évalue ainsi la `valeurMinMax` = `MinMax(P)` de la façon récursive suivante :

- Si la position **P** est au plus bas niveau de profondeur, on retourne sa valeur propre avec `set_valeur` selon le calcul précédent
- Sinon, on retourne le maximum de ses valeurs filles s'il s'agit d'un coup de l'ordinateur, et le minimum s'il s'agit d'un coup du joueur humain

Ainsi, on parcourt l'ensemble des positions possibles de bas en haut et de gauche à droite de façon récursive, auxquelles on associe une valeur : la valeur maximale ou minimale de ses filles. La fonction `coup_min_max` de la classe `Coup` renvoie alors le coup associé à la meilleure position fille.

3.3.3 Etape 3 - décision

Maintenant que nous avons les valeurs de toutes les filles de P_0, il ne reste plus qu'à choisir le coup menant à la fille de valeur maximale. C'est le rôle de `coup_min_max`.

3.3.4 Conclusion partielle

A cause d'un manque de vision globale sur le projet ou par souci d'amélioration, nous avons été confrontés à plusieurs changements de direction en cours de projet.

Pour représenter des listes de coups, nous avions d'abord opté pour des listes chaînées de classe `ListeCoups` comportant un pointeur vers le premier élément, le dernier élément et le nombre d'élément. Chaque Coup possédait de fait en attribut un pointeur `Coup *Next` vers le Coup suivant s'il existe et un pointeur `Coup *Prev` vers le précédent, afin de pouvoir parcourir les liste dans les deux sens. Malheureusement, le code perdait beaucoup en clarté, notamment lorsque l'on devait ajouter ou retirer un élément à une liste. Nous avons finalement abandonné la classe ListeCoups au profit du type `list<Coup>`.

Nous avions à l'origine une méthode `void MinMax()` de `Position`. Elle parcourait récursivement son arbre des possibles (créé au préalable par `generateur`) pour attribuer à chaque sommet sa `valeurMinMax`. En travaillant le code, nous avons su fusionner cette méthode à `generateur` qui opérait déjà ce parcours récursif.

Chapter 4

Résultats

Maintenant que nous avons détaillé le cheminement de notre projet, étudions ce que nous obtenons comme résultat.

4.1 Interface globale

Nous avons ainsi créé un main global permettant d'explorer toutes les possibilités que nous avons créé. Avant le lancement d'une partie, l'utilisateur est soumis à plusieurs choix via des invites de commande : il peut jouer au TikTakToe, jouer aux échecs ou bien lancer des parties ordinateur qui permettent de relever les performances de l'algorithme pour différents paramètres (temps ou nombre de victoires) (voir partie Résultats).

4.2 Morpion

Commençons par étudier les performances en terme de victoires sur 100 parties jouées des différents algorithmes.
Nous observons qu'à ce jeu, commencer est un avantage certain ! Même notre joueur aléatoire arrive à rivaliser contre un mim max de profondeur 4 sur un nombre non négligeable de partie. Cela s'explique en partie par le fait que le MinMax postule sur le fait que le joueur adverse jouera un coup intelligent, ce qui n'est pas le cas du joueur aléatoire...
Ensuite, on observe que seul le joueur 4 peut rivaliser contre un MinMax d'un niveau inférieur qui commence.

	Aléatore	MinMax *k=1*	MinMax *k=2*	MinMax *k=3*	MinMax *k=4*
Aléatore	11	42	72	72	68
	0	0	12	17	13
	89	58	16	13	19
MinMax k=1	0	0	0	0	100
	0	0	0	0	0
	100	100	100	100	0
MinMax k=2	1	0	0	0	100
	8	0	0	0	0
	91	100	100	100	0
MinMax k=3	1	0	0	0	0
	5	0	0	0	100
	94	100	100	100	0
MinMax k=4	4	0	0	0	0
	0	0	0	0	100
	96	100	100	100	0

Figure 4.1: Nombre de victoires (rouge), égalités (blanc) et défaites (vert). Le joueur rouge commence la partie.

Le temps de calcul explose si la profondeur augmente, et c'est normal au vu du fonctionnement du MinMax. Si on veut aller presqu'au bout de toutes les fins de parties (profondeur 8), on atteint un temps de calcul de 998s par coup ce qui est bien trop élevé. Cependant, on peut obtenir un algorithme qui ne perd jamais, prouvant la simplicité de ce jeu ...

Figure 4.2: Temps (ms) de calcul aux 1er, 2ème et 3ème coup joué du TikTakToe pour différents algorithmes

4.3 Echecs

Pour 10 parties lancées opposant l'ordi aléatoire au MinMax, on a bien 10 victoires pour MinMax, et ce même si l'ordi aléatoire commence. En effet, le fait de commencer constitue pour l'ordi aléatoire un avantage stratégique : pour 10 parties aléatoire VS aléatoire, le joueur qui commence gagne 2 fois pour 8 égalités. Le test montre donc une performance satisfaisante de l'algorithme MinMax en terme de jeu. Un joueur MinMax2 gagne également contre un joueur MinMax 1.

Figure 4.3: Temps de calcul (ms) en fonction du tour joué aux échecs

Cependant, le temps de calcul n'est plus du tout raisonnable pour ce jeu ! Avec une profondeur de 2 chaque coup prend près d'une minute avant d'être retourné ce qui est déjà très grand et nous empêche d'aller plus loin dans l'exploration...

Conclusion

Ce projet nous aura permis de tirer de nombreuses enseignements. Tout d'abord, le codage d'un jeu d'échecs (même sans faire intervenir de joueur ordinateur) et des ses subtilités est un projet de grande envergure qui nous a permis d'explorer de nombreux aspects du codage en C++. Nous avons eu à élaborer l'architecture d'un projet de groupe, utiliser GIT afin de communiquer, travailler sur une architecture globale avant de coder, ...

Nous avons été freinés par des problèmes techniques : l'ordinateur d'un membre de l'équipe a rencontré des dysfonctionnements graves et a dû partir en réparation, tandis qu'un autre membre a rencontré des problèmes mystérieux avec les .json de VScode. Cela a rendu plus complexe le partage des tâches.

En terme de résultat final, nous obtenons un jeu d'échecs et de morpion fonctionnel, ainsi qu'un ordinateur aléatoire et un ordinateur utilisant MinMax. Nous obtenons des premiers résultats de performance mais notre ordinateur MinMAx est loin d'être parfait. Sur le jeu d'échecs, l'algorithme MinMax est bien trop lent et aurait pu être amélioré. Nous aurions pu pratiquer un élagage Alpha-Beta des branches, ou améliorer la structure interne pour éviter de devoir actualiser le plateau deux fois (calcul des coups possibles et calcul de la valeur si on arrive à une feuille).