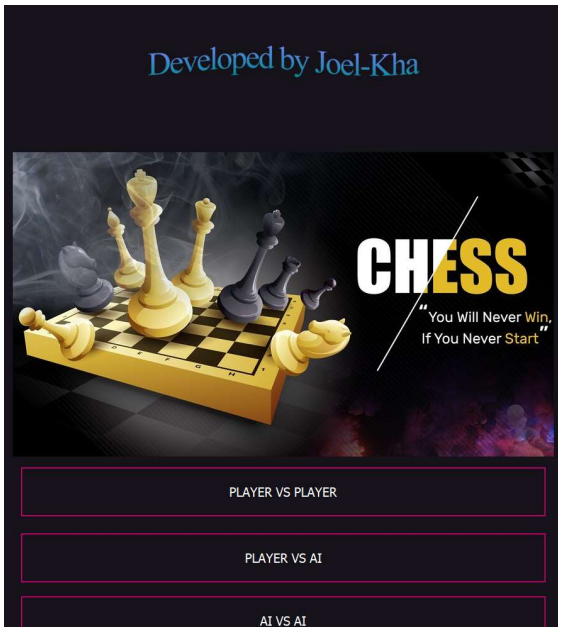

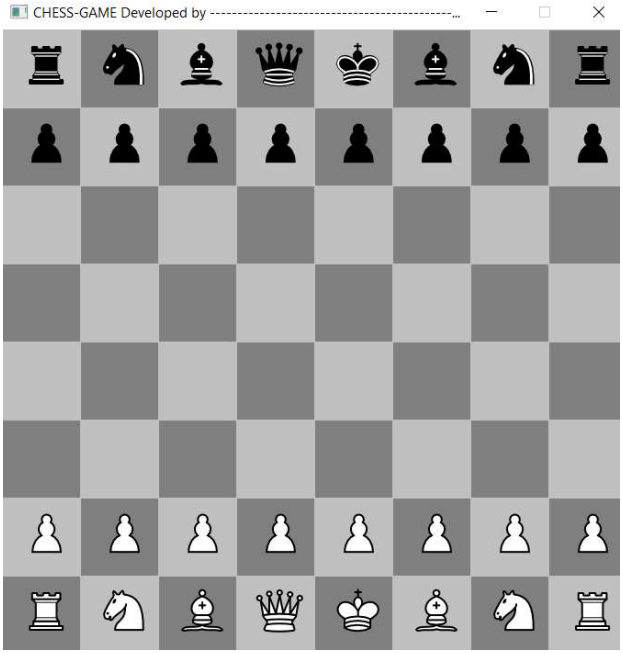

Programmation d'un jeu d'échecs en Python

ENSTA Bretagne	ENSTA Bretagne 2 rue F. Verny 29806 Brest Cedex 9, France	FOYANG, Joël, joel.foyang@ensta-bretagne.org KHA, Alexandre, alexandre.kha@ensta-bretagne.org

Table des matières

Introduction 4
1 Contenu des fichiers et notice d'utilisation 5
2 Construction des pièces 6
2.1 Méthode de R. Hyatt 6
2.2 La classe Piece 7
2.2.1 Le constructeur 7
2.2.2 Méthode *move* 8
2.2.3 Méthode *another_move* 8
2.2.4 Méthode filter_from_color_pieces. 8
2.2.5 Méthode simulation_save_king_move 8
2.2.6 Méthodes à surcharger 9
2.3 Les classes filles 9
2.3.1 Constructeurs 9
2.3.2 Méthodes en commun et surcharge 9
2.3.3 Coups spéciaux 10
3 L'échiquier 11
3.1 Constructeur de la classe Echiquier 12
3.2 Rendre effectif le déplacement des pièces et le tour par tour 12
3.2.1 Alterner les tours : méthode *switch_current_player_color* 12
3.2.2 Signifier l'échec : méthode *check_echec* 12
3.2.3 Mieux se repérer dans l'échiquier 12
3.2.4 Déplacements 13
3.2.5 Outils de simulation : sauvegarder et revenir à l'étape précédente 13
3.2.6 Menaces et échec 13
3.2.7 Fin de partie 13
3.3 Intelligence artificielle 14
4 Interface Homme-Machine 14
4.1 Méthode execute_first_click 15
4.2 Méthode execute_second_click 15
5 Intelligence artificielle 16
5.1 Constructeur de la classe AIEngine 16
5.2 BasicAI 16
5.3 MiniMax 17
5.4 NegaMax 18

5.4.1 Elagage alpha-beta 18
5.4.2 Fonction d'évaluation 20
5.5 Mémorisation 20
5.6 Bibliothèque d'ouverture 21
6 Tests Unitaires 23
Conclusion 23
Contributions 24
Annexe : diagrammes de classes 24

Introduction

Le jeu d'échecs, est un jeu ancien, introduit au Xe siècle en Europe par les arabes, qui malgré des règles simples génère un nombre très important de positions. Cela en fait d'ailleurs un jeu encore non résolu à notre époque : c'est-à-dire qu'il n'existe pas de méthode réaliste qui permettrait de gagner à coup sûr comme au Puissance 4 (Connect 4 en anglais) par exemple.
Le jeu standard oppose deux joueurs sur un échiquier de 64 cases. Chacun dispose de 16 pièces, dont la couleur, noir ou blanc, distingue les deux joueurs. Ces pièces sont composées d'un roi, une dame, de deux tours, deux cavaliers, deux fous et de huit pions. Chacun de ces huit types de pièces possède des mouvements particuliers, et peuvent capturer (càd éliminer) les pièces adverses selon leur positionnement. Le joueur qui gagne est celui qui neutralise le roi adverse, c'est-à-dire qu'il le menace sans que le roi adverse puisse être défendu.
L'objectif de notre projet est d'implémenter ce jeu d'échecs en Python.
Pour ce faire, il sera nécessaire de construire des objets adéquats, qui permettront de créer les pièces, l'échiquier et d'implémenter les règles du jeu. De plus, il sera primordial de réaliser une Interface Homme-Machine (IHM) afin d'avoir une représentation visuelle du déroulement du jeu.
Au final, le programme devra permettre à deux joueurs physiques de jouer l'un contre l'autre, ainsi que proposer des parties Joueur vs Intelligence Artificielle (I.A.).
Le jeu d'échecs n'étant pas résolu, on proposera une approche pour construire le comportement de l'intelligence artificielle basée sur les algorithmes MiniMax et NegaMax.

1 Contenu des fichiers et notice d'utilisation

Sept fichiers python sont utilisés dans ce projet.
Un premier, nommé « pieces.py » permet de construire la classe « Piece » ainsi que les classes filles correspondant à chacun des huit types de pièces. Si chacune des pièces possède des mouvements très généraux (ex. les fous se déplacent en diagonale), il faut déterminer si ces coups sont toujours possibles de par la position des autres pièces. On implémente alors dans chacune de ces classes les mouvements légaux respectifs, prenant en compte la disposition de l'échiquier.
Un deuxième fichier, intitulé « board.py » sert à construire l'échiquier ainsi que de le mettre à jour au cours de la partie. Nous y implémentons la classe Echiquier, qui contient les méthodes permettant de rendre effectifs les déplacements possibles de chaque pièce.
Dans le fichier « AIEngine.py » nous implémentons les intelligences artificielles MiniMax et Negamax. Ces programmes simulent des situations de jeu sur une profondeur donnée et les évaluent selon des critères spécifiques, choisis personnellement ou non, puis jouent les coups adaptés. Ce fichier va de pair avec le fichier « dict.py » qui enregistre dans un dictionnaire les coups déjà calculés par les IA, leur évitant les calculs redondants.
Un fichier est dédié aux tests unitaires, « test_chess.py ».
Le fichier « gui.py » est celui qui permet de créer l'IHM, et finalement, le programme final, c'est-à-dire le jeu, est lancé depuis le fichier « main.py ».

Pour lancer le jeu correctement, il suffit d'enregistrer le dossier fourni « Chess_final » dans un répertoire pycharm et de lancer le « main ». Au préalable il faudra installer le module *playsound* si ce n'est pas déjà fait. Pour des raisons qui nous sommes inconnues le mode Player vs AI et AI vs AI ne joue qu'en mode debugg .

2 Construction des pièces

2.1 Méthode de R. Hyatt

L'échiquier, réellement déclaré dans la classe *Echiquier*, est perçu comme une liste de 64 éléments contenant le nom des cases comme suit :

```
coord =
['a8', 'b8', 'c8', 'd8', 'e8', 'f8', 'g8', 'h8',
'a7', 'b7', 'c7', 'd7', 'e7', 'f7', 'g7', 'h7',
'a6', 'b6', 'c6', 'd6', 'e6', 'f6', 'g6', 'h6',
'a5', 'b5', 'c5', 'd5', 'e5', 'f5', 'g5', 'h5',
'a4', 'b4', 'c4', 'd4', 'e4', 'f4', 'g4', 'h4',
'a3', 'b3', 'c3', 'd3', 'e3', 'f3', 'g3', 'h3',
'a2', 'b2', 'c2', 'd2', 'e2', 'f2', 'g2', 'h2',
'a1', 'b1', 'c1', 'd1', 'e1', 'f1', 'g1', 'h1']
```

Pour pouvoir rendre compatible la représentation précédente avec le mouvement des pièces, nous avons décidé d'exploiter la méthode de Robert HYATT. Celle-ci vise à empêcher les sorties de l'échiquier.
En effet, disons qu'une tour se trouve sur la case " a1", donc l'élément 56 de notre liste. Un de ses coups usuels consiste à se déplacer vers la gauche de 1, ce qui correspond donc au passage à l'élément d'indice précédent (55) dans notre liste. Or ce coup ne peut pas avoir lieu (bord de l'échiquier) et admettons que l'on autorise la décrémentation de 1 de son indice, la case 55 correspond à la case du bord droit dans la ligne supérieure, de nom "h2", ce qui n'est pas du tout autorisé.
Pour cela il est utile de considérer un tuple de 120 éléments qui correspond à notre échiquier agrandi avec des cases « -1 » en plus, qui sont des cases interdites :

```
tab120 = (
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1,  0,  1,  2,  3,  4,  5,  6,  7, -1,
-1,  8,  9, 10, 11, 12, 13, 14, 15, -1,
-1, 16, 17, 18, 19, 20, 21, 22, 23, -1,
-1, 24, 25, 26, 27, 28, 29, 30, 31, -1,
-1, 32, 33, 34, 35, 36, 37, 38, 39, -1,
-1, 40, 41, 42, 43, 44, 45, 46, 47, -1,
-1, 48, 49, 50, 51, 52, 53, 54, 55, -1,
-1, 56, 57, 58, 59, 60, 61, 62, 63, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1)
```

On considère un deuxième tuple de 64 éléments qui prend en argument l'indice des cases autorisées dans tab120 (de valeurs 0 à 63) :

```
tab64 =
 (  21, 22, 23, 24, 25, 26, 27, 28,

    31, 32, 33, 34, 35, 36, 37, 38,

    41, 42, 43, 44, 45, 46, 47, 48,

    51, 52, 53, 54, 55, 56, 57, 58,

    61, 62, 63, 64, 65, 66, 67, 68,

    71, 72, 73, 74, 75, 76, 77, 78,

    81, 82, 83, 84, 85, 86, 87, 88,

    91, 92, 93, 94, 95, 96, 97, 98)
```

L'idée de la méthode est que si un coup nous fait positionner dans une case « -1 » dans tab120, le coup sera détecté comme illégal. Prenons l'exemple précédent avec la tour en « a1 » (élément 56 de *coord*) qui veut se déplacer vers la gauche de 1.
On se réfère à tab64 : le 56^{e} élément de tab64 est « 91 ». Le mouvement de 1 vers la gauche correspond à une décrémentation de 1 donc transforme 91 en 90. Or dans tab120, le 90^{e} élément est « -1 », ce qui nous fait conclure que le coup ne peut pas être autorisé.

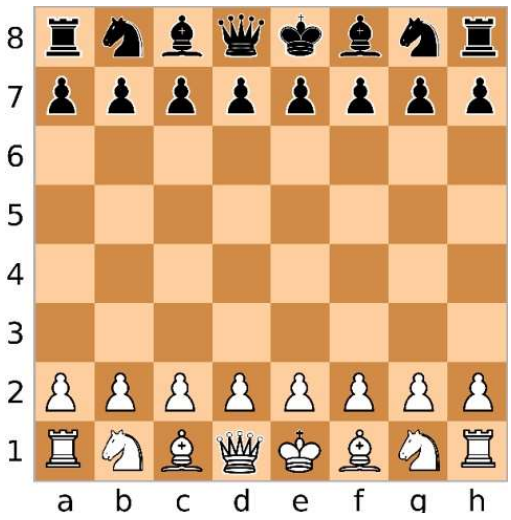

Figure 1 : échiquier et système de coordonnées

2.2 La classe *Piece*

2.2.1 Le constructeur

La classe *Piece* est construite avec les variables d'instances suivantes : nom, couleur qui caractérisent intuitivement chaque pièce, puis on choisit également de leur attribuer les variables d'instance image, qui permettra de reconnaître les pièces par une image png, une fois l'IHM implémentée, et code qui servira pour l'IA.

2.2.2 Méthode *move*

Cette méthode permet d'obtenir les possibilités de coups des pièces à mouvement continu (i.e. la dame, la tour et le fou). Elle prend notamment en arguments des entiers *north, east, west, south,* qui caractérisent les directions des pièces (ex. Le fou se déplace en diagonale), et fournies comme on le verra plus tard dans les classes filles dédiées. Selon ces directions caractéristiques et la position actuelle, la méthode parcourt l'échiquier de façon à ne pas en sortir, ce qui est permis par une vérification dans *tab120* : il ne faut pas que l'on se trouve sur une case de valeur –1. De plus, la méthode cesse d'explorer une direction si elle pointe sur une case occupée. Les cases d'après ne sont donc pas prises comme coups possibles. Cependant, si la méthode pointe sur une case occupée par une pièce alliée (de même couleur), il faut aussi veiller à retirer ce coup des possibilités, car il n'existe pas de coup qui permette momentanément à deux pièces alliées d'être sur la même case. Cela est permis par la méthode filtre *filter_from_color_pieces.*

2.2.3 Méthode *another_move*

Cette méthode retourne cette fois les coups possibles des pièces autres que la reine, la tour et le fou. Pour ce faire, à partir de la liste des coups généraux d'une pièce, *possible_move,* la méthode vérifie que chaque coup ne sortira pas de l'échiquier et que de plus il ne peut pas y avoir de collision avec des pièces de même couleur grâce au filtre *filter_from_color_pieces.*

2.2.4 Méthode filter_from_color_pieces.

Comme rappelé précédemment, le filtre supprime des coups possibles, ceux qui induisent un déplacement sur une case occupée par une pièce alliée. Pour ce faire il suffit juste de vérifier que si la case du coup en question n'est pas vide (« *is not None* » en syntaxe python), la couleur de la pièce occupante est de notre couleur. Si c'est le cas, le coup est supprimé.

2.2.5 Méthode simulation_save_king_move

Cette fonction est clé dans le bon fonctionnement de notre jeu. En effet, même si de façon théorique, un coup respecte le fait de rester sur le terrain, ne passe pas au travers d'une autre pièce (pour les pièces à mouvement continu) et ne collisionne pas une pièce alliée, il existe une contrainte supplémentaire qui est au cœur du jeu : ne pas laisser le roi en échec. En effet, il est illégal de ne pas protéger son roi lorsqu'il est menacé, et si c'est impossible, le jeu se conclut par échec et mat.
Donc si notre roi est menacé, il faut soit mouvoir une autre pièce de façon à le protéger, soit le déplacer directement. La méthode *simulation_save_king_move* veille justement à ce que l'on ne puisse pas jouer un coup qui mettrait en péril le roi, soit en le gardant en échec, soit en ouvrant la voie à un échec au prochain coup. Pour ce faire, grâce à la méthode *deplacer* d'*Echiquier*, cette fonction simule naïvement des coups pris en paramètres sur l'échiquier et vérifie si notre roi est en échec, grâce à la fonction auxiliaire *check_echec* explicitée dans la classe Echiquier. Si c'est le cas, ces coups sont stockés dans une liste des coups illégaux. On fait ensuite appel à une autre fonction, *back_last_move*, pour revenir à l'état précédent de l'échiquier pour ne pas fausser le déroulement du jeu. Au final, tous ces coups illégaux sont éliminés de la liste des coups possibles.

2.2.6 Méthodes à surcharger

moves et *simple moves* sont des méthodes qui seront surchargées dans les classes filles.

2.3 Les classes filles

Nous créons des classes qui correspondent aux six types de pièces que sont le roi, la reine, la tour, le fou, le cavalier et le pion. Elles héritent toutes de la classe mère *Piece*.

2.3.1 Constructeurs

2.3.1.1 Eléments communs

En plus des variables d'instances héritées de *Piece,* chaque classe fille possède d'autres variables en commun. Toutes possèdent la variable *evaluation,* qui quantifie son poids. Par exemple, *evaluation* pour *Pion* est de 10, mais celle pour *Tour* est de 60, ce qui veut dire que matériellement la tour est plus importante que le pion.
De plus, selon sa couleur, chaque type de pièce possède un code.

2.3.1.2 Eléments spécifiques

La classe *Pion* dispose des variables d'instance *test_deplace_pN, test_deplace_pB* initialisées à *False.* Ces variables renseignent si le pion a avancé de deux cases pour son premier coup. Cela sera important pour implémenter la prise en passant.
Tour possède la variable d'instance *has_made_first_move_t* initialisée à *False* et qui renseigne si la tour a déjà bougé ou non. La variable devient *True* si la tour en question a bougé, et cela interdira par la suite la possibilité de roquer, un coup particulier impliquant une tour et le roi, que l'on détaillera après. De même *Roi* possède une variable d'instance similaire *has_made_first_move_r* initialisée à *False,* dont le passage à *True* permettra d'interdire les roques.
De plus, *Roi* possède les variables d'instance booléennes *echec, mat et pat* qui renseignent respectivement sur l'état d'échec, de mat (s'il est en mat, l'adversaire gagne) et de pat ou non du roi. Cette dernière situation se produit lorsqu'il y a impasse : le joueur n'est pas en échec mais ne peut plus jouer de coups légaux. C'est match nul.

2.3.2 Méthodes en commun et surcharge

2.3.2.1 Méthode simple_moves

Dans chaque classe fille de *Piece,* on surcharge de manière spécifique la méthode *simple_moves.* Cette dernière donne la liste des coups "naïfs" de chaque pièce, c.-à-d. qu'ils ne prennent pas en compte l'échec de son roi. Pour la plupart des pièces, il s'agit d'appeler la méthode *move* décrite précédemment en spécifiant un vecteur déplacement ou les directions de déplacement. Notons que la dame combine les mouvements du fou et de la tour, et que le pion nécessite un traitement particulier. En effet, le mouvement du pion est très variable : au début de la partie seulement, il a le droit de se déplacer de deux cases vers l'avant, puis face à une autre pièce il ne peut se déplacer mais peut éventuellement capturer une pièce adverse directement en diagonale. Finalement, le pion peut effectuer le coup spécial de la prise en passant, que l'on détaille plus tard.

2.3.2.2 Méthode *moves*

La méthode moves prend cette fois en compte l'échec au roi. Il simule les coups de *simple_moves,* vérifie si le roi est en échec en appelant *simulation_save_king_move* tout en annulant à la fin les coups simulés et en supprimant les coups qui mettent en péril le roi.

2.3.2.3 Méthode *evaluer*

Chaque pièce est dotée de la méthode *evaluer*, qui permet d'évaluer leur poids relativement à une pièce adverse. Si la valeur retournée est positive, la pièce adverse a plus de poids. Cela sera utile pour l'IA.

2.3.3 Coups spéciaux

2.3.3.1 Prise en passant

Ce coup permet de capturer un pion adverse dans des conditions non standards. L'idée de ce coup est de pouvoir capturer un pion qui, en se déplaçant de deux cases (c'est donc son premier mouvement), évite une capture : nous faisons comme si elle ne s'était déplacée que d'une case.
Comme nous pouvons le voir sur l'illustration ci-dessous, le pion blanc peut ici capturer le pion noir et se retrouver en e6.

Figure 2 : Prise en passant

Pour implémenter ce coup, on choisit de considérer deux méthodes de la classe *Pion*, *prise_en_passant_g* pour la prise à gauche puis *prise_en_passant_d* pour la droite. Le principe des deux fonctions est de repérer les pions blancs sur la rangée 5 (et par symétrie, les pions noirs sur la rangée 4). Puis si un pion de couleur opposée se trouve directement latéralement et s'est déplacé de deux cases, ce que l'on sait grâce à *test_deplacer_pN* et *test_deplacer_pB*, le coup en passant est inclus dans les coups possibles.

2.3.3.2 Promotion

La promotion est un évènement qui se produit lorsqu'un pion arrive sur la dernière rangée en face. Basiquement, elle transforme le pion en une pièce du choix du joueur (à part le roi). Pour simplifier notre programme, la promotion transformera toujours le pion en une dame, ce qui est une transformation populaire du fait de la puissance de celle-ci. Au niveau du programme, la méthode *promotion* de *Pion* enregistre une dame dans la case concernée de l'échiquier à la place du pion.

2.3.3.3 Roques

Le grand et petit roque sont des coups qui peuvent être utilisés pour protéger le roi. Ils mobilisent une tour et le roi en même temps. Trois conditions doivent être remplies pour exécuter le roque : le roi et la tour ne doivent pas avoir joué de coup, aucune pièce ne se trouve entre le roi et la tour et finalement aucune case de transit (dont arrivée et départ) du roi n'est menacée par l'adversaire.

Figure 3 : En vert le petit roque et en rouge le grand roque

Les coups du roque sont implémentés par les méthodes *petit_roque* et *grand_roque* de la classe Roi, ce qui fait que ces coups seront vus comme des coups disponibles du roi mais pas des tours.
Elles vérifient exactement les critères cités précédemment, en analysant l'échiquier et en prenant en compte les variables d'instance *has_do_first_move_*r et *has_do_first_move_t*, qui indiquent le déplacement ou non du roi et de la tour.

3 L'échiquier

La classe *Echiquier* est munie de la variable de classe *coord,* explicitée auparavant et qui permet de faire correspondre un indice de liste au nom de la case en question. Elle permettra de mieux repérer les pièces.

3.1 Constructeur de la classe *Echiquier*

cases modélise l'échiquier par une liste qui contient les pièces présentes. Elle est initialisée par

```
self.cases =
[Tour('noir'), Cavalier('noir'), Fou('noir'), Reine('noir'), Roi('noir'),
Fou('noir'),
Cavalier('noir'), Tour('noir'),
Pion('noir'), Pion('noir'), Pion('noir'), Pion('noir'), Pion('noir'),
Pion('noir'), Pion('noir'),
Pion('noir'),
None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None,
None, None, None, None, None, None, None, None,
Pion('blanc'), Pion('blanc'), Pion('blanc'), Pion('blanc'), Pion('blanc'),
Pion('blanc'),
Pion('blanc'), Pion('blanc'),
Tour('blanc'), Cavalier('blanc'), Fou('blanc'), Reine('blanc'), Roi('blanc'),
Fou('blanc'),
Cavalier('blanc'), Tour('blanc')]
```

Elle est mise à jour continuellement au cours de la partie.
is_echec_blanc et *is_echec_noir* sont des booléens qui renseignent respectivement sur l'échec ou non de chaque joueur. *current_player* et *current_player_color* vont permettre de mettre en place un système de tour par tour afin que l'adversaire puisse jouer à la fin de notre coup. Enfin *last_moves_white* et *last_moves_black* sont des listes qui enregistrent les derniers coups de chacun.

3.2 Rendre effectif le déplacement des pièces et le tour par tour

3.2.1 Alterner les tours : méthode *switch_current_player_color*

La méthode change *current_player_color* en "noir" si ceux qui viennent de jouer sont les blancs et vice versa.

3.2.2 Signifier l'échec : méthode *check_echec*

La méthode *check_echec* prend en paramètre l'une des deux couleurs et retourne *True* si le joueur de cette couleur est en échec et *False* s'il ne l'est pas. On détaille plus tard comment on détermine qu'un joueur est en échec.

3.2.3 Mieux se repérer dans l'échiquier

Pour mieux se repérer dans l'échiquier et manipuler l'identification entre coordonnées alphanumériques et indices, on définit la méthode *get_dest_pos* qui prend en paramètre le nom d'une case (ex. "a2", "h5") puis renvoie l'indice correspondant dans *coord*. Par exemple : `Echiquier.get_dest_pos("a1") = 56`.
La méthode *num* fait l'inverse, elle prend en paramètre une liste de positions en indice (de *coord)* et renvoie la liste de ces positions converties en alphanumérique ("a1", "a2", …)

3.2.4 Déplacements

Les déplacements sont mis en place principalement par deux fonctions.
deplacer_simple rend effectif le coup d'une pièce sur l'échiquier. La méthode prend en paramètres une pièce ainsi que la position finale et en prenant en compte le type de la pièce, modifie *cases* de sorte à changer la case de départ en "None" et d'inscrire la pièce qui bouge dans la case d'arrivée. Le traitement des coups spéciaux est différent mais l'idée reste de mettre à jour les éléments de la liste *coord*. La méthode renvoie également les informations relatives (départ, arrivée …(?)) au coup joué ainsi que qu'un booléen qui caractérise si l'adversaire est en échec ou non.
deplacer complète la méthode précédente en l'appelant, ce qui met à jour l'échiquier, mais elle sauvegarde les informations du coup joué grâce à *save_last_move* puis change la couleur de celui qui joue pour permettre à l'adversaire de jouer. Finalement, la méthode renvoie comme *deplacer_simple* un booléen qui détermine l'échec ou non de l'adversaire.

3.2.5 Outils de simulation : sauvegarder et revenir à l'étape précédente

Pouvoir sauvegarder une position et revenir à l'étape précédente est la clé de la simulation (et notamment de *simulation_save_king_move*).
Deux méthodes le permettent :

- *save_last_moves* prend en argument une couleur et un coup représenté par un tuple qui contient la position initiale, position finale, référence de la pièce à la position initiale et référence de la pièce à la position finale. Ce coup est concaténé dans *last_moves_white* ou *last_moves_black* selon la couleur prise en paramètre.

- *back_last_move* prend en paramètre une couleur et annule le dernier déplacement effectué par le joueur de cette couleur, ce qui se fait en supprimant le dernier terme de *last_moves_white* ou *last_moves_black.* La méthode veille également à réinitialiser les variables *has_do_first_move* à *False* si le roi ou la tour ont joué ou si un roque a été produit. (En passant ?)

3.2.6 Menaces et échec

Pour vérifier l'échec, on a besoin de savoir si le roi est menacé par une pièce adverse.
check_menace prend en paramètre une couleur et parcourt l'échiquier en concaténant dans une liste les coups possibles de l'adversaire.
is_echec prend en paramètre une couleur et vérifie si le roi de cette couleur est en échec ou pas. Pour cela la méthode appelle *check_menace* pour donner la liste des positions menacées par l'adversaire et vérifier si la position du roi en fait partie. Si oui, le roi est en échec, donc la méthode modifie is_echec_blanc ou is_echec_noir en True. L'indice dans cases du roi en échec est aussi retourné par la méthode.

3.2.7 Fin de partie

On définit certaines méthodes qui analyseront l'échiquier et indiqueront si un joueur est en position de mat ou de pat, ce qui signe la fin de partie.
La méthode qui vérifie l'échec et mat du joueur actuel est *check_echec_matt.* Si la liste des coups possibles du joueur actuel -obtenable via *moves*- est vide et qu'en plus le joueur est en échec, ce qui est vérifiable grâce à *is_echec, al*ors la méthode retourne *True,* signifiant le mat et donc la perte du joueur actuel. Sinon la méthode renvoie *False.*

check_pat vérifie de la même manière le pat du joueur actuel, à la différence qu'il n'y a pas nécessité que le joueur soit en échec pour qu'il soit pat. Il suffit juste que sa liste de coups légaux soit vide : il ne peut plus jouer.

3.3 Intelligence artificielle

Dans le fichier *board.py* sont aussi implémentées certaines méthodes liées à une des IA, nommée BasicAI. Puisqu'une partie sera ensuite dédiée aux intelligences artificielles, on explique dans les grandes lignes le rôle de ces méthodes :

- *Pos_piece* renvoie la liste des indices des pièces dont la couleur et le type ont été précisés en argument.
- Prenant en paramètre une liste de pièces pouvant être capturées, *Choice_best_piece* renvoie la position de la pièce à la plus grande valeur (évaluation matérielle).
- *Pos_piece_pouvant_etre_manger* retourne la liste des positions des pièces d'une certaine couleur qui sont menacées et *piece_pouvant_etre_mange* retourne la liste des pièces menacées.
- *Play_ai_move* retourne des informations relatives au coup de l'IA.
- *get_score_by_color* renvoie la somme l'évaluation matérielle de chaque pièce d'une couleur donnée, ce qui détermine l'avantage ou non de cette couleur. *Score_noir* (sans paramètre) donne l'évaluation matérielle des noirs grâce à *get_score_by_color* évaluée en "noir". Idem pour *score_blanc*.
- La méthode *get_move* (sans paramètre) donne tous les coups du joueur actuel représentés comme des couples alphanumériques ("départ", "arrivée") dans une liste, ce qui sera utile pour simuler les coups à la suite.

4 Interface Homme-Machine

Le fichier principal permettant la gestion des interfaces graphiques est le module *gui.py.* Afin de pouvoir visualiser dans l'intégralité le mouvement des pièces dans le tableau (Echiquier) nous avons opté pour la création d'une classe *QtBoard* qui hérite de *QtWidget*. Pour notre jeu nous avons décidé d'attribuer à l'interface différents modes de jeu à savoir : *Player vs Player, AI vs Player et AI vs AI*.
La classe *QtBoard* prendra en paramètres la classe *Echiquier* et le mode, cette classe nous permettra de dessiner l'échiquier dans son ensemble (taille des cases, couleur des cases et autres...), de gérer les évènements, tels que le dessin perpétuel des pièces dans l'échiquier assuré par la fonction *PaintEvent*, la gestuelle des Pieces dans le tableau avec des cliques assurés par la fonction *mousePressEvent*, et enfin la mise à jour automatique du tableau lorsqu'il y a mouvement. Notons que la fonction *PaintEvent*, appelle la méthode *draw_images_pieces* (charger du dessin des pièces dans le tableau)
La gestuelle des pièces n'étant pas très évidente (il faudra faire une différence entre le premier click (sélection de la pièce) et le deuxième click (déplacement de la pièce)), nous avons opté pour deux méthodes, qui sont *execute_first_click* et *execute_second_click*.

4.1 Méthode *execute_first_click*

Cette méthode repère les coordonnées du click dans le tableau si la position du click est une pièce alors elle demande à la fonction *PaintEvent* de peindre en rouge les cases où la pièce peut se déplacer, si la position n'est pas une pièce alors aucun événement ne se produit.

4.2 Méthode execute_second_click

Après le premier click, lors du deuxième click, cette méthode repère si le click se fait sur la même position, dans ce cas, la pièce ne se déplace pas, si le click se fait dans une position autre que les cases peintes en rouge alors le déplacement voulu n'est pas permis, aucun évènement ne se produit. L'utilisateur comprendra qu'il essaye un mauvais déplacement. Notons qu'à chaque click il y a une mise à jour du tableau qui se produit par la méthode intégré *update ()*
Pour des raisons de design nous avons pris l'initiative de créer une classe *Game*, qui sera la première fenêtre d'accueil du jeu, cette fenêtre contient des logo, images et 3 boutons cliquable (correspondant aux modes de jeu cités précédemment).

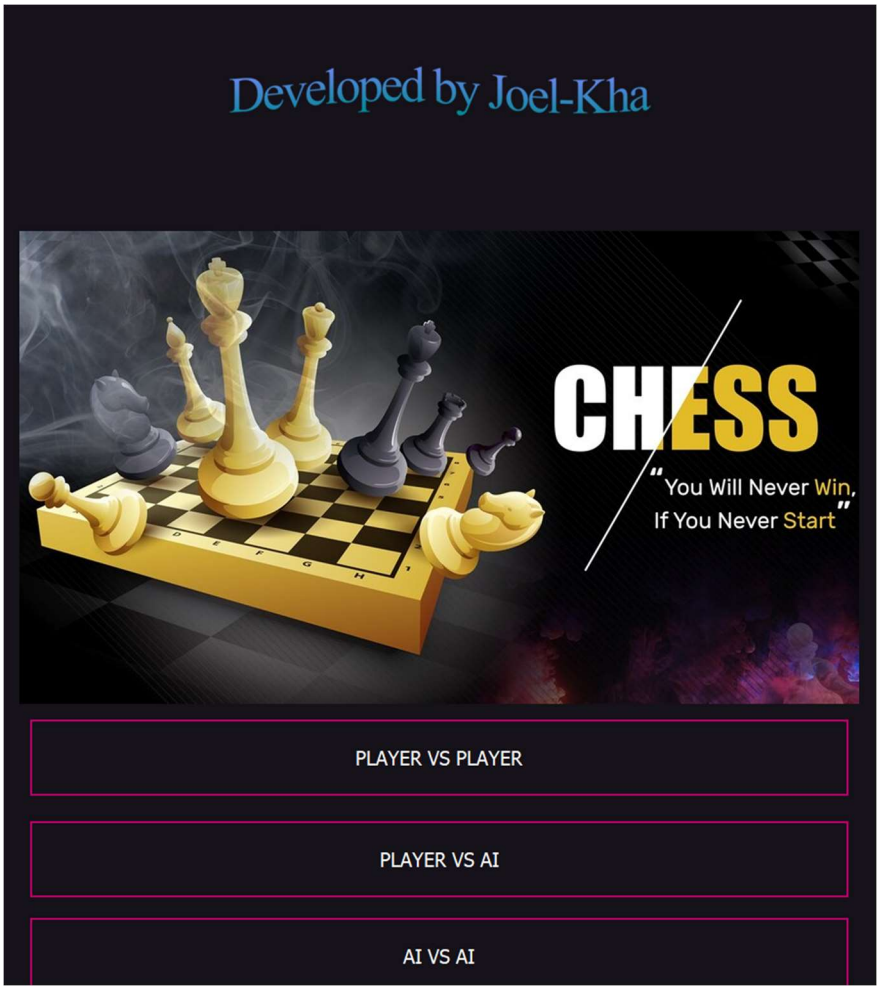

Figure 4 : fenêtre d'accueil du jeu

C'est cette classe qui construira une instance de la classe *QtBoard*. Elle prendra en paramètres le mode sélectionné et l'échiquier dans la classe *Game.* Afin de faciliter la reconnaissance des Player par le programme, on a créé une class *Player,* et deux classes *HMPlayer (HuMan Player)* et *AIPlayer* qui héritent de *Player.* Ces trois classes permettront d'une part de faciliter l'échange de tour dans le mode *Player vs Player,* elles permettront aussi à l'IA de savoir quand elle devra jouer dans le mode *AI vs Player.*
La gestion des évènements étant faite, le challenge sera actuellement de pouvoir lier l'interface graphique aux mouvements de l'IA. Les mouvements de l'IA sont traités dans le fichier *AIEngine.py.* La liaison entre les mouvements de l'IA reçus dans le fichier *AIEngine.py* et l'interface graphique causait quelques problèmes majeurs car la réponse de l'IA n'étant pas instantanée, la fenêtre de jeu était en attente lorsque l'on cliquait sur la fenêtre par hasard. Celle-ci (la fenêtre de jeu) n'ayant rien reçu, la mise à jour du tableau ne sera pas possible, car nous avons vu précédemment qu'à chaque click il y a une mise à jour du tableau qui se produit.
Afin de pallier ce problème nous avons décidé de mettre la réponse de l'IA sous forme de Thread (programme en cours d'exécution par un ordinateur n'influençant pas les autres process) dans l'interface graphique. Nous avons donc créé dans l'interface graphique des classes *NegamaxWorker, MinimaxWorker, BasicAI,* et *OpeningAIWorker (*ces algorithmes seront expliqués dans la partie intelligence artificielle). Ces classes seront le siège des traitements des donnes reçu par l'AI dans l'interface graphique. Ainsi c'est dans ces Threads que sera faite la gestion des mouvements de l'IA dans l'interface graphique par la méthode *start_ai_compute* présente dans la classe *QtBoard*

5 Intelligence artificielle

Le terme « intelligence artificielle », créé par John McCarthy, est souvent abrégé par le sigle « I. A. » (ou « A. I. » en anglais, pour Artificial Intelligence). Il est défini par l'un de ses créateurs, Marvin Lee Minsky, comme « la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains, car elles demandent des processus mentaux de haut niveau tels que : l'apprentissage perceptuel, l'organisation de la mémoire et le raisonnement critique ». La gestion de la recherche du meilleur coup dans notre projet est dans le module *AIEngine.py.* Ce module contient la classe *AIEngine* qui sera le siège du moteur de recherche de notre IA

5.1 Constructeur de la classe AIEngine

La classe *AIEngine* 3 variables d'instance qui sont *zobTable*, transposition, nécessaires pour le *Zobrist Hachage* (sera expliquer ultérieurement) cette classe utilisera plusieurs moteurs de recherche (dont certains sont des moteurs test).

5.2 BasicAI

Le moteur de jeu BasicAI est un simple moteur de recherche de niveau 1, nous l'avons implémenté pour pouvoir visualiser l'effectivité du déplacement d'une pièce recommandé par l'ordinateur. Ce moteur est notamment basé sur du hasard.

5.3 MiniMax

C'est un algorithme de jeu assez puissant sur les jeux à somme nulle (où la somme des gains et des pertes de tous les joueurs est égale à 0. Cela signifie donc que le gain de l'un constitue obligatoirement une perte pour l'autre) consistant à minimiser la perte maximum.

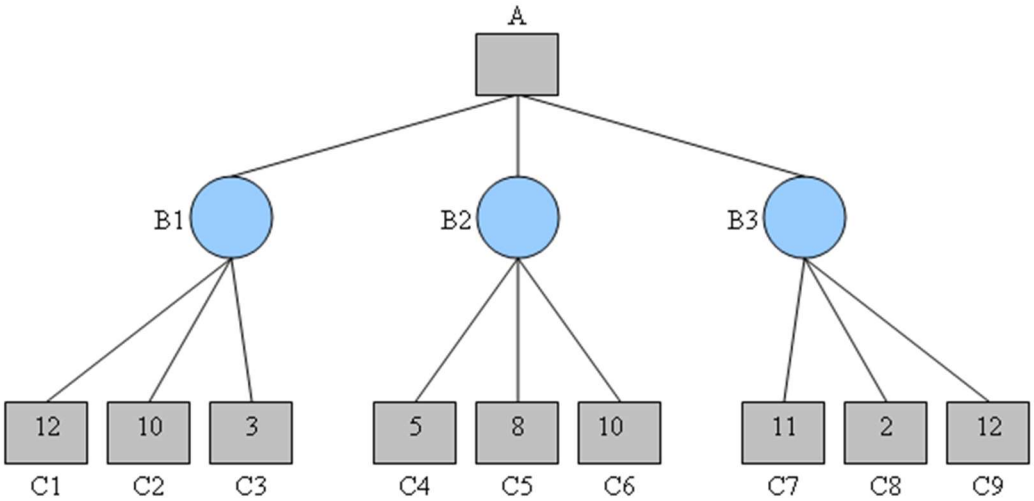

Figure 5 : Arbre de jeu

Dans le schéma ci-dessus, les nœuds gris représentent les nœuds joueurs et les bleus les nœuds opposants. Pour déterminer la valeur du nœud A, on choisit la valeur maximum de l'ensemble des nœuds B (A est un nœud joueur). Il faut donc déterminer les valeurs des nœuds B qui reçoivent chacun la valeur minimum stockée dans leurs fils (les nœuds B sont opposants). Les nœuds C sont des feuilles, leur valeur peut donc être calculée par la fonction d'évaluation.

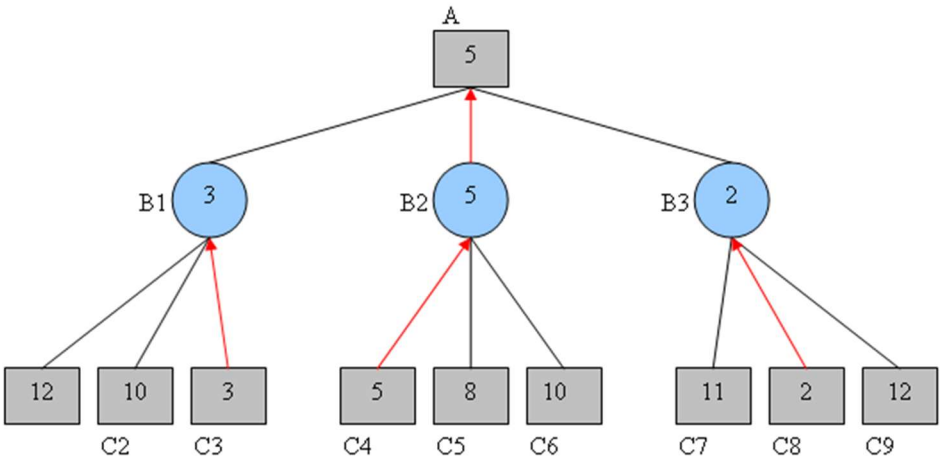

Figure 6 : Recherche MiniMax

Le nœud A prend donc la valeur 5. Le joueur doit donc jouer le coup l'amenant en B2. En observant l'arbre, on comprend bien que l'algorithme considère que l'opposant va jouer de manière optimale : il prend le minimum. Sans ce prédicat, on choisirait le nœud C_1 qui propose le plus grand gain et le prochain coup sélectionné amènerait en B1. Mais alors on prend le risque que l'opposant joue C3 qui propose seulement un gain de 3.

En pratique, la valeur théorique de la position P ne pourra généralement pas être calculée. En conséquence, la fonction d'évaluation sera appliquée sur des positions non terminales. On considérera que plus la fonction d'évaluation est appliquée loin de la racine, meilleur est le résultat du calcul. C'est-à-dire qu'en examinant plus de coups successifs, nous supposons obtenir une meilleure approximation de la valeur théorique donc un meilleur choix de mouvement.

5.4 NegaMax

NegaMax est une variance de l'algorithme Minimax basé sur le fait que :
max (a, b) = -min (-a, -b). Le pseudo-code de cet algorithme peut être :

```
int negaMax( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves)  {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

Figure 7 : Pseudo-code de NegaMax
Source : ChessProgramming Wiki

5.4.1 Elagage alpha-beta

L'élagage alpha-beta est une version modifiée de l'algorithme minimax. Il s'agit d'une technique d'optimisation pour l'algorithme minimax. Le jeu d'échec dénombrant énormément de possibilités, le nombre d'états de jeu que l'algorithme doit examiner est exponentiel. Nous ne pouvant pas éliminer l'exposant mais nous pouvons réduire considérablement le nombre d'état de recherche. Cette réduction se fera par **l'élagage**, technique par laquelle sans vérifier chaque nœud de l'arbre de jeu, nous pouvons calculer la décision minimax correcte. Cela implique deux paramètres de seuil Alpha et bêta pour l'expansion future, il est donc appelé **alpha-bêta élagage**. Il est également appelé **algorithme Alpha-Beta**.

On en détaille le principe :

Étape 1 : À la première étape, le lecteur Max commencera d'abord à se déplacer à partir du nœud A où $\alpha = -\infty$ et $\beta = +\infty$, ces valeurs de alpha et bêta sont passées au nœud B ou encore $\alpha = -\infty$ et $\beta = +\infty$, et le nœud B passe la même valeur à son enfant D.

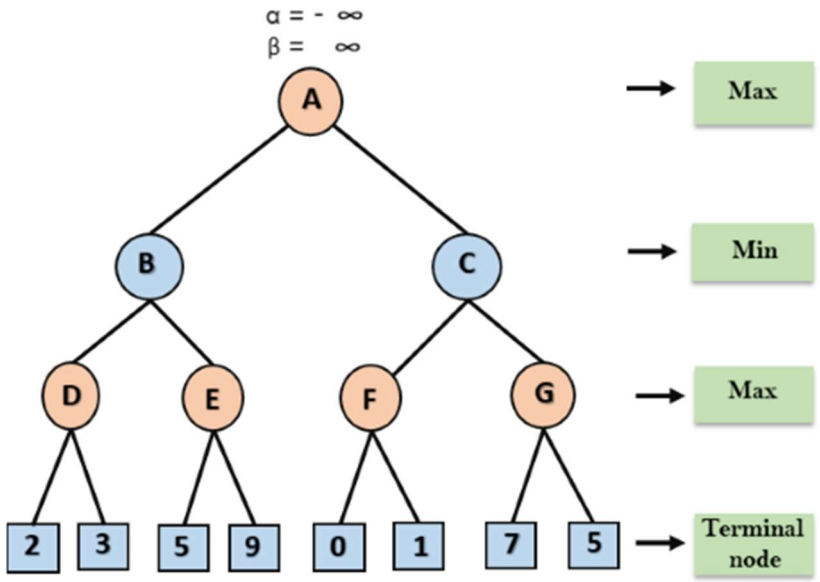

Figure 8 : Etape 1 de la recherche MiniMax

Étape 2 : Au nœud D, la valeur de α sera calculée comme son tour pour Max. La valeur de α est comparée d'abord à 2, puis 3, et le max (2, 3) = 3 sera la valeur de α au nœud D et la valeur du nœud sera également 3.

Étape 3 : Maintenant, l'algorithme revient au nœud B, où la valeur de β changera car il s'agit d'un tour de Min, maintenant β = +∞, comparera avec la valeur des nœuds suivants disponibles, c'est-à-dire min (∞, 3) = 3, donc au nœud B maintenant α = -∞, et β = 3.

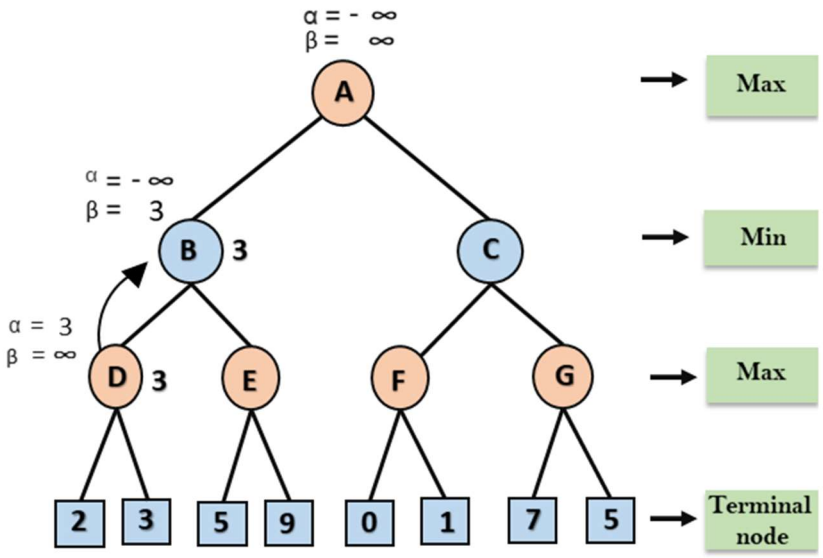

Figure 9 : Etape 3 de la recherche MiniMax

Dans l'étape suivante, l'algorithme traverse le successeur suivant du nœud B qui est le nœud E, et les valeurs de α = -∞, et β = 3 seront également passées.

Étape 4 : Au nœud E, Max prendra son tour et la valeur de alpha changera. La valeur actuelle de alpha sera comparée à 5, donc max (-∞, 5) = 5, donc au nœud E α = 5 et β = 3, où α>= β, donc le bon successeur de E sera élagué, et l'algorithme ne le traversera pas, et la valeur au nœud E sera 5.

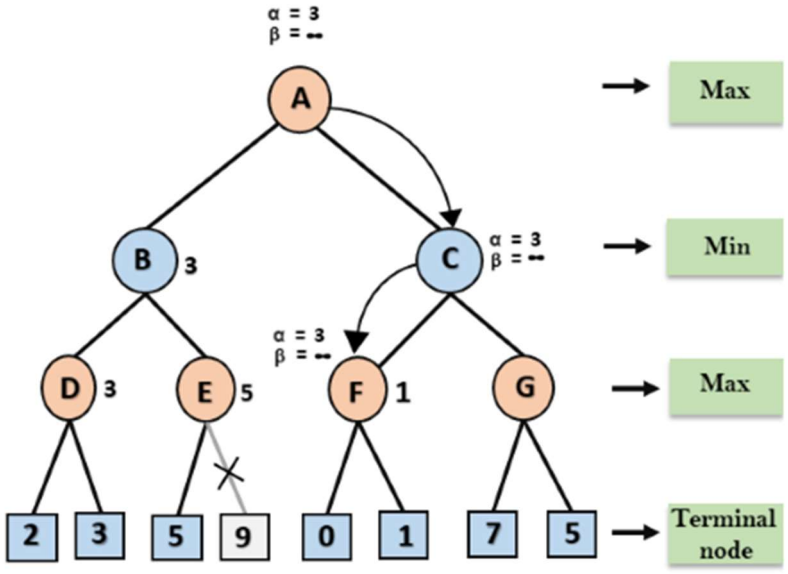

Figure 10 : Etape 4 de la recherche MiniMax

Le nœud F renvoie la valeur du nœud 1 au nœud C, à C α = 3 et β = +∞, ici la valeur de beta sera modifiée, elle sera remplacée par 1 donc min (∞, 1) = 1. Maintenant à C, α = 3 et β = 1, et encore une fois, il satisfait à la condition α>= β, de sorte que le prochain enfant de C qui est G sera élagué, et l'algorithme ne calculera pas l'ensemble du sous-arbre G.

5.4.2 Fonction d'évaluation

Pour « évaluer » une position, c'est-à-dire savoir si elle est bonne ou mauvaise, il faut lui attribuer une valeur. Chaque position de l'échiquier est analysée selon plusieurs critères. Une note est calculée en fonction des pièces présentes et de nombreux bonus/malus.
Nous nous sommes limités à l'évaluation matérielle et évaluation de postions (le cavalier reçoit un bonus lorsqu'il est au centre de l'échiquier par exemple)

5.5 Mémorisation

L'algorithme Negamax, notre moteur de jeux d'échec, n'est pas très intelligent dans la mesure ou s'il rencontre une situation de jeu donnée déjà évaluée, il sera obligé de la recalculer. A l'échelle d'un jeu d'échec ce n'est pas très optimal. Pour pallier ce problème nous allons utiliser :
Zobrist hashing (en outre mentionné comme zobrist keys ou zobrist marks), qui est un développement de fonction de hachage utilisé dans le cadre de programmes informatiques qui jouent à des jeux de société uniques, par exemple, les échecs et le go, pour exécuter des tables de transposition, une sorte exceptionnelle de table de hachage qui est répertoriée par une

position de plateau et utilisé pour s'abstenir d'examiner une position similaire plus d'une fois. Zobrist hashing est nommé d'après son créateur, Albert Lindsey Zobrist.
L'idée derrière Zobrist Hashing est que pour un état de carte donné, s'il y a une pièce sur une cellule donnée, nous utilisons le nombre aléatoire de cette pièce de la cellule correspondante dans le tableau.

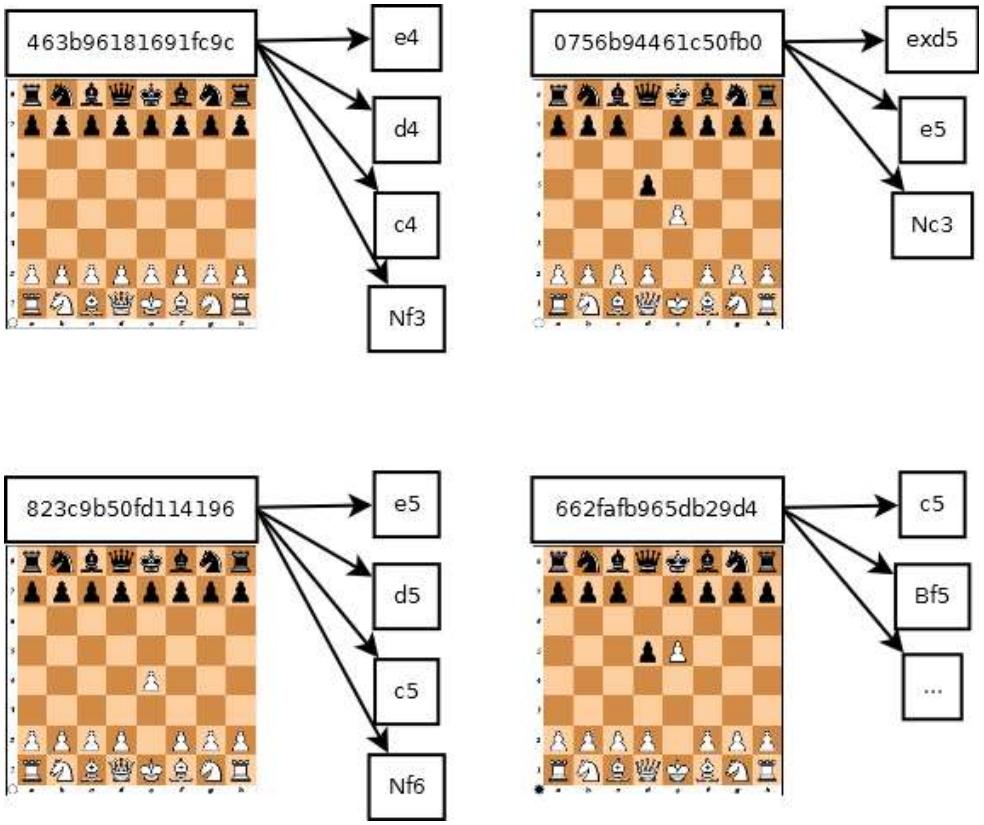

Figure 11 : Hashing

Chaque situation dans l'échiquier est codée par la méthode *compute_hash* du module *AIEngine.py* cette méthode utilise bien évidement la variable *Zobtable* de la classe *AIEngine.*
Les problèmes avec les méthodes de hachage sont les collisions (situation dans laquelle deux données ont un résultat identique avec la même fonction de hachage)
pour éviter cela, on a donc utilisé les nombres de 64 bits, en effet il est très peu probable qu'une collision de hachage se produise avec des nombres aussi grands. Une fois les situations dans l'échiquier codée, la variable *transposition* (dictionnaire ayant pour clé : le hashcode d'une situation et pour valeur la position dans le tableau) permet à AI d'enregistrer au cours du jeu les différentes situations, ainsi lorsque la fonction Negamax sera appelé elle vérifiera tout d'abord si la situation actuelle n'a pas déjà été rencontré une fois que cette vérification a été faite notre moteur peut alors renvoyé le meilleur coup (selon la profondeur) a l'interface graphique par l'intermédiaire des Threading *NegamaxWorker* ou *MinimaxWorker* présent dans le module *gui.py*.

5.6 Bibliothèque d'ouverture

Après avoir utilisé les tables de transposition nous avons optimiser le moteur de recherche Negamax. Cependant en jouant des parties contre notre moteur d'échec on a pu

remarquer qu'il avait des débuts de parties très étranges (mêmes coups joués). Cela n'est pas étonnant : vu que la fonction d'évaluation est uniquement basée sur l'avantage matériel, et que généralement les prises interviennent plus tard dans la partie, tous les coups sont jugés plus ou moins équivalents par l'intelligence artificielle.
La solution est de doter ce programme d'une bibliothèque d'ouvertures (*book.txt*). Il s'agit d'un fichier où sont répertoriées des suites de premiers coups classiques (il y a notamment l'ouverture italienne, l'espagnole, la défense Pirc, la nimzo-indienne, etc.). Tout joueur d'échecs se doit de bien connaître ces ouvertures et l'ordinateur devra faire de même. On va donc procéder ainsi pour améliorer le début de partie :

• On va donner à l'ordinateur quelques centaines d'ouvertures

• Au début, le programme sera en mode « ouverture »

• S'il commence la partie, le programme jouera le premier coup d'une de ces ouvertures prise au hasard

• Le programme ira fouiller dans la liste d'ouvertures pour trouver celle qui est en train de se jouer, et il se contentera de jouer les coups de cette ouverture tant qu'il le pourra

• Quand la liste de coups est terminée ou lorsque l'adversaire ne jouera plus le coup « classique », le programme quittera le mode ouverture et utilisera son I.A.

Cette bibliothèque d'ouverture est ainsi un arbre de coups dont les nœuds sont des coups aux échecs. La génération de cet arbre se fait dans le module *dict.py* grâce à la classe *OpenBookTree* contenant une sous classe *Node* qui représente les nœuds de l'arbre

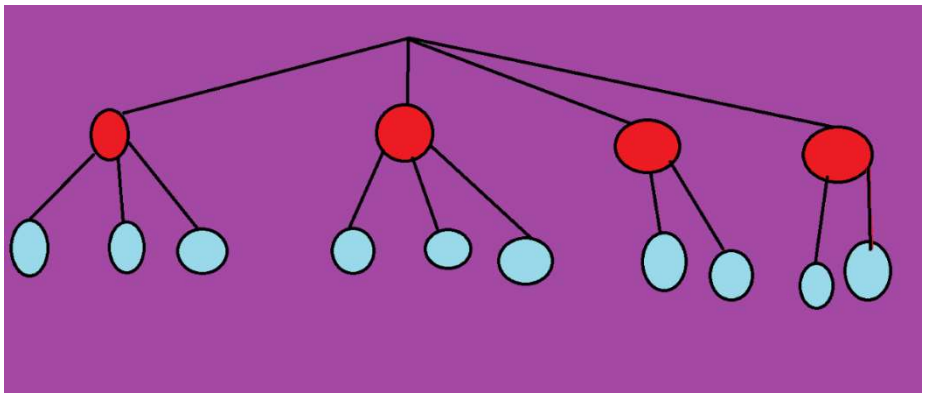

Figure 12 : Représentation de la bibliothèque d'ouvertures comme un arbre de coups

Dans cet arbre les nœuds rouges sont potentiellement des ouvertures très connues. Supposons que ce soit un joueur humain qui le fasse au premier coup, alors l'IA ira chercher la réponse adéquate dans les enfants bleus du nœuds rouge choisi et ainsi de suite.
Le mode ouverture est lié à l'interface graphique par la classe *OpeningAIWorker* a l'aide des fonctions *get_first_move* (premier coup dans l'arbre), *get_next_move_single_AI (*En fonction du premier coup joué la fonction choisie parmi les enfants le coup suivant)
et get_next_move_AI_AI (réservé uniquement au mode *AI vs AI)* on note que ces

fonctions renvoient le coup de l'AI dans le Thread *OpeningAIWorker* présent dans le module *gui.py* qui est ensuite affiché par l'interface.

6 Tests Unitaires

Afin de tester le bon fonctionnement de nos méthodes et du type de nos objets en cours de jeu nous avons doté notre projet d'un module *test_chess.py.* Ce module met en évidence 7 tests à savoir :
Test_pieces : qui permet de vérifier si nos objets sont réellement des pièces.
Test_deplacement : qui permet d'assurer l'effectivité du déplacement c'est-à-dire la pièce qui se déplace, déplace avec lui son instance et laisse derrière lui une case vide.
Test_prise_en_passant : qui permet d'assurer l'effectivité de ce mouvement spécial, c'est-à-dire le pion subissant la prise en passant est mangé sans toutefois avoir été remplacé
Test roque : qui permet d'assurer l'effectivité de ce mouvement spéciale c'est à dire le déplacement instantané de la tour lorsque le roi est en train de roqué
Test_back_move *:* qui permet d'assurer l'annulation d'un déplacement c'est à dire vérifier si après le *back_moves* la pièce se retrouve à sa place précédente, notons que nous avons applique ce test a la prise en passant et le roque du roi
Test_promotion: qui permet de tester la promotion d'un pion
Test_si_deplacement**:** qui permet de tester les variables booléennes *has_do_first_moves_r, has_do_first_moves_t* et *test_deplace_p* c'est à dire vérifie si après un quelconque déplacement ces dernières sont passées à *True*

Conclusion

Si les principaux objectifs du projet qui visaient à permettre des parties Joueur vs Joueur puis Joueur vs IA ont été remplis, plusieurs difficultés ont été rencontrées durant le projet.

On notera une difficulté premièrement à l'implémentation des coups autorisés.
En effet, des contraintes spatiales empêchent les pièces de sortir de l'échiquier, de passer au travers des pièces adverses et de collisionner nos propres pièces, ce qui est à prendre en compte au-delà des coups généraux. Pour remédier à ces problèmes, on a vu l'utilité de la méthode de R. Hyatt pour délimiter l'échiquier puis des fonctions filtres empêchant les collisions alliées.
De plus, il faut que chaque coup joué ne menace pas notre propre roi, ce qui passe par des simulations de coups pour voir si le roi est en échec au tour prochain, puis des retours en arrière.
Egalement, le pion et le roi possèdent des coups spéciaux, qui nécessitent une vérification de conditions supplémentaires sur l'échiquier puis un traitement différent de leur déplacement par rapport aux autres pièces.

Une deuxième difficulté est liée à l'implémentation de l'IHM, mais qui a pu être surmontée par une documentation sur les fonctionnalités techniques de PyQT5.

Finalement, la partie Intelligence Artificielle a pu susciter de nombreux questionnements et a requis non seulement de se documenter sur les algorithmes MiniMax et NegaMax, mais également des prises d'initiatives dans l'évaluation de l'échiquier (position avantageuse, désavantageuse).
On notera que ces algorithmes sont plutôt coûteux en temps, ce qui a motivé l'implémentation d'une mémorisation des coups dans un dictionnaire.

Après implémentation de la mémorisation, les algorithmes sont plutôt rapides en profondeur 2, plus lents en profondeur 3 mais jouables, puis beaucoup trop longs pour les profondeurs supérieures, ce qui constitue une limite de notre projet.

Contributions

Nous récapitulons succinctement les contributions de chacun dans le code.

Joël FOYANG : *board.py*, IHM, *AIEngine.py* (Intelligence Artificielle)
Alexandre KHA : *pieces.py*, board.py, seulement *MiniMax* de *AIEngine.py*
Certains codes ont été travaillés à deux

Annexe : diagrammes de classes

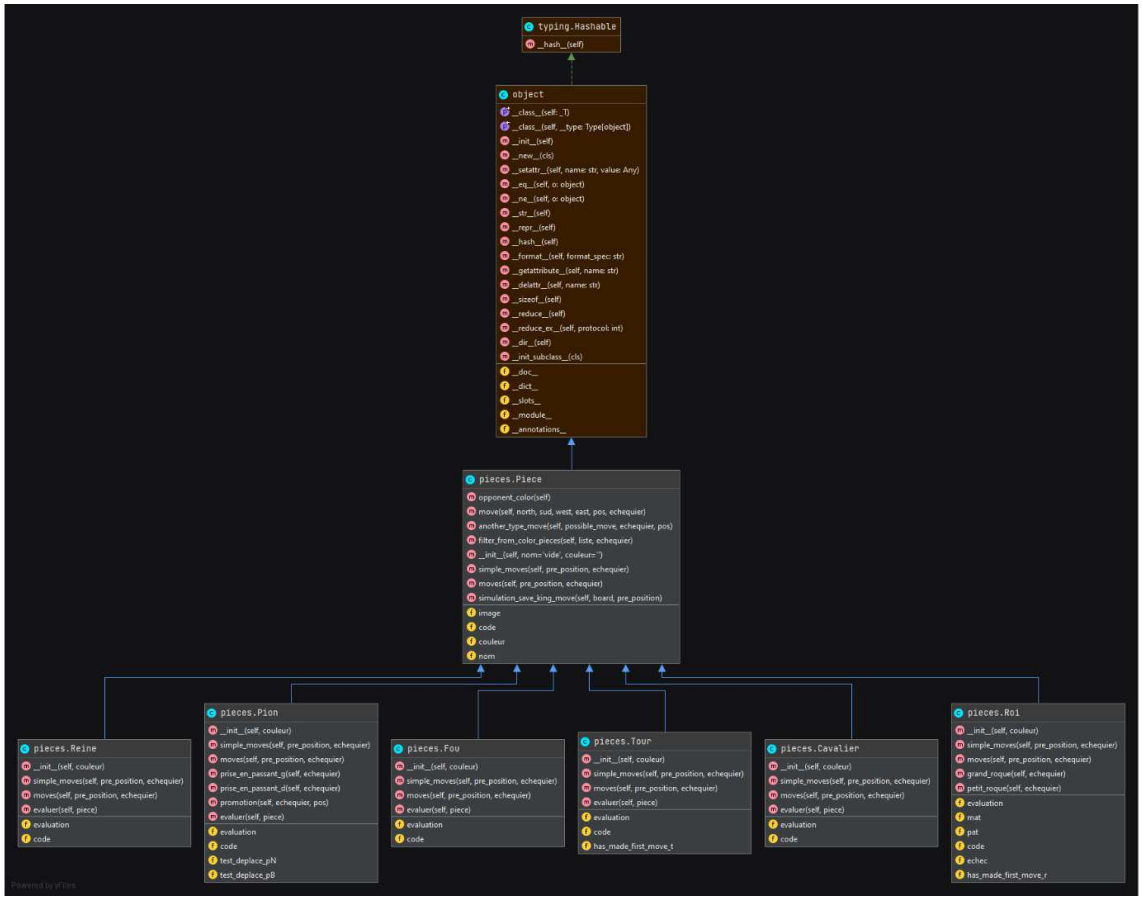

Figure 13 : Diagramme de la classe ***Piece***

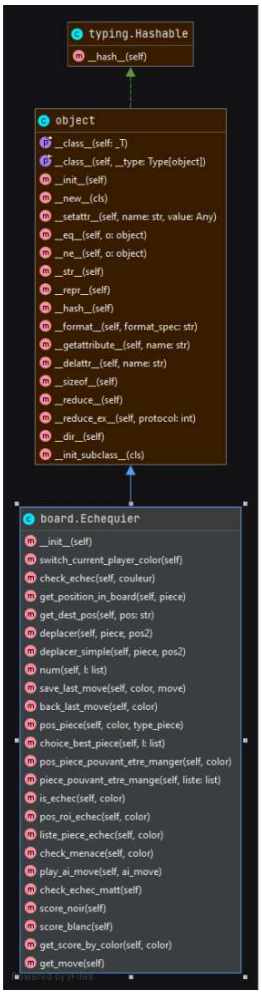

*Figure 14 : Diagramme de la classe **Echiquier***

Note : cette classe a été nommée « Echequier » auparavant, ce qui a été modifié dans la version finale du code

*Figure 15 : Diagramme de la classe **AIEngine***

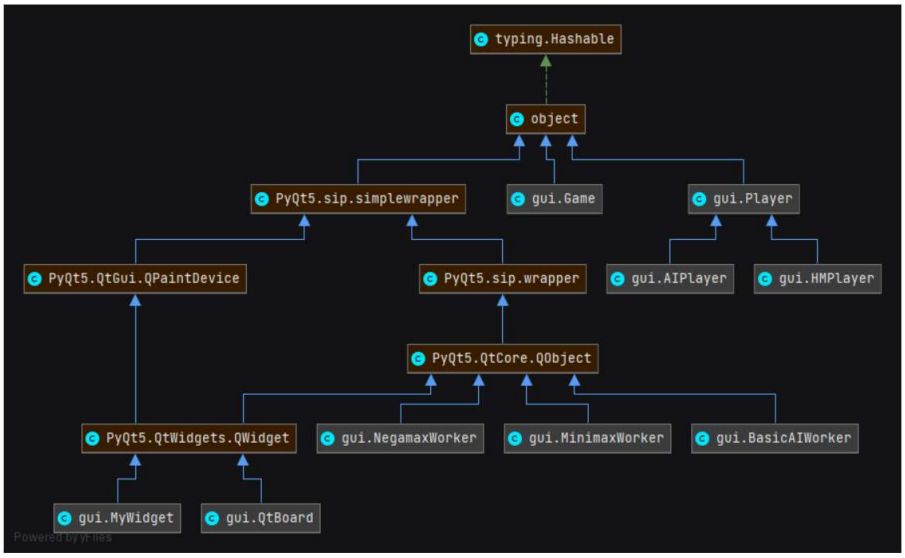

Figure 13 : Diagrammes de classes de l'IHM