Filière MP - ENS de Paris-Saclay, Lyon, Rennes et Paris - Session 2023

Page de garde du rapport de TIPE

Attention : Le rapport doit être réalisé par l'étudiant(e).

Si le rapport résulte d'une collaboration, elle doit être clairement annoncée.

NOM :	CUINGNET	Prénoms :	MARTIN
Classe :	MP*		
Lycée :	Janson de Sailly	Numéro de candidat :	22268
Ville :	PARIS		

Concours auxquels vous êtes admissible, dans la banque MP inter-ENS (les indiquer par une croix) :

ENS P.-Saclay	MP - Option MP		MP - Option MI	
	Informatique MP	X		
ENS Lyon	MP - Option MP		MP - Option MI	
	Informatique MP	X		
ENS Rennes	MP - Option MP		MP - Option MI	X
ENS Paris	MP - Option P		MP - Option I	
	Informatique MP			

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique	X	Mathématiques		Physique	

Titre du TIPE :

Construction d'un moteur d'échecs

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	16	Illustration	27	Bibliographie	1

Attention, les illustrations doivent figurer dans le corps du texte et non en fin du document !

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

Je vais ici réaliser un moteur d'échecs : un programme comprenant les règles des échecs en plus de pouvoir, à partir d'une position quelconque, donner le meilleur coup à jouer afin de maximiser les chances de victoire du camp à jouer ; lui permettant de jouer de manière autonome.

À Paris

Le 06/06/23

Signature du (de la) candidat(e)

Signature du professeur responsable de la classe préparatoire dans la discipline

Cachet de l'établissement

LYCEE JANSON DE SAILLY
106, rue de la Pompe
75775 PARIS Cedex 16
Tél. 01.55.73.28.50 - Fax 01.55.73.28.50
075 0699 C

La signature du professeur responsable et le tampon de l'établissement ne sont pas indispensables pour les candidats libres (hors CPGE).

TIPE : Construction d'un moteur d'échec

Martin CUINGNET
Numéro d'inscription : 22268

Résumé

J'ai réalisé le programme en C++ en utilisant la bibliothèque graphique SFML pour réaliser l'interface utilisateur. Le code est disponible en annexe, le projet en son ensemble peut être retrouvé sur ce répertoire github et le profil en ligne du moteur peut être trouvé sur `https://lichess.org/@/mcngnt`.

Table des matières

1 Implémentation des règles

Pour représenter une position du jeu d'échec, il faut savoir la position de chaque pièce sur le plateau ainsi que des informations comme la couleur qui doit jouer ou encore les possibilités de roque. Pour représenter le plateau d'échecs en lui-même, j'utilise une array de 8×8 cases contenant chaque pièce présente. Une pièce sera représentée par un entier de 8 bits dont les trois premiers bits donneront le type de pièce (Pion, Tour, Fou, Cavalier, Dame, Roi) et les deux derniers la couleur de la pièce. Ainsi la Dame Noire sera représentée par :

$$\overbrace{10}^{\text{Noir}}\overbrace{101}^{\text{Dame}}$$

Cette représentation permet un accès efficace à la couleur d'une pièce au moyen d'un masque binaire, de l'opération ET binaire et du décalage de bits.
À ce tableau, j'adjoins les informations nécessaires à la représentation d'une position à l'aide de cette structure :

```
struct GameState
{
  int capturedPiece;
  bool canWhiteKingCastle;
  bool canWhiteQueenCastle;
  bool canBlackKingCastle;
  bool canBlackQueenCastle;
  int doublePushFile;
  int moveCount;
  int whiteKingPos;
  int blackKingPos;
  uint64_t zobristKey;
};
```

Celle-ci permet notamment de gérer le roque ou la prise en passant.
Une partie sera représentée par une suite de coups, le plateau étant modifié de manière dynamique à chaque coup, tandis que le `GameState` sera ajouté à une pile.
On va également garder à jour une liste contenant la position de chaque pièce sur l'échiquier afin de ne pas à avoir à itérer sur l'entièreté des cases (potentiellement vides) de l'échiquier.
Un coup va être représenté par un entier contenant la position d'arrivée et de départ de la pièce effectuant le coup ainsi que le type de coup réalisé (un drapeau représenté par un entier de 4 bits). Positions de départ et d'arrivée sont ainsi représentées par un entier de 5 bits numérotant le plateau de la manière suivante :

FIGURE 1 – Identification des cases du plateau par des entiers

2 Génération des mouvements

Pour pouvoir jouer et évaluer le meilleur coup possible à partir d'une position, il faut pouvoir générer tous les coups légaux depuis ladite position. Pour cela, je génère d'abord tous les coups pseudo-légaux, c'est-à-dire les mouvements pouvant être réalisés par les pièces, indépendamment du fait qu'un roi soit en échec ou non. La fonction `generatePseudoMoves` va ainsi prendre en paramètre le plateau représenté par une classe `BoardManager` et parcourir les cases du plateau en rajoutant pour chaque pièce rencontrée les mouvements possibles de la pièce dans une liste chainée (ici un `std::vector<int>` en C++). Ensuite, pour conserver seulement les coups légaux, j'utilise la fonction `controlledSquares` déterminant les cases de l'échiquier contrôlées par l'adversaire : c'est-à-dire les cases où une pièce ennemie peut se déplacer.À l'aide de ces informations, je vérifie pour chacun des mouvements possibles si le roi est en échec après l'avoir effectué. La fonction `generateMoves` va donc s'écrire :

```
fonction generateMoves():
    pseudoCoups = generatePseudoMoves()
    initialiser la liste coupsLégaux
    pour chaque coup dans pseudoCoups:
        faire coup
        si notre roi est sur une case de controlledSquares():
            passer à l'itération suivante
        sinon:
            défaire coup
            ajouter coup à coupsLégaux
    retourner coupsLégaux
```

On va ainsi avoir besoin de deux fonctions pour générer des coups légaux : une fonction pour jouer des coups (`makeMove`) et une pour les annuler (`unmakeMove`).

Dans ces deux fonctions, on va modifier l'état du plateau en accord avec le coup passé en paramètre ainsi que l'état du jeu représenté par la structure `GameState`. Pour plus de facilités dans l'implémentation de `unmakeMove`, je vais utiliser une pile de `GameState` où un appel de `makeMove` va empiler un nouveau `GameState` là où `unmakeMove` va le dépiler. Mais contrairement à l'état du jeu, le

plateau ne sera pas, lui, gardé en mémoire, mais modifié de manière dynamique de manière à diminuer le coup temporel et en mémoire de `makeMove` et `unmakeMove`.
Ainsi, à partir d'une position, on peut générer l'ensemble des coups légaux pouvant être joués.

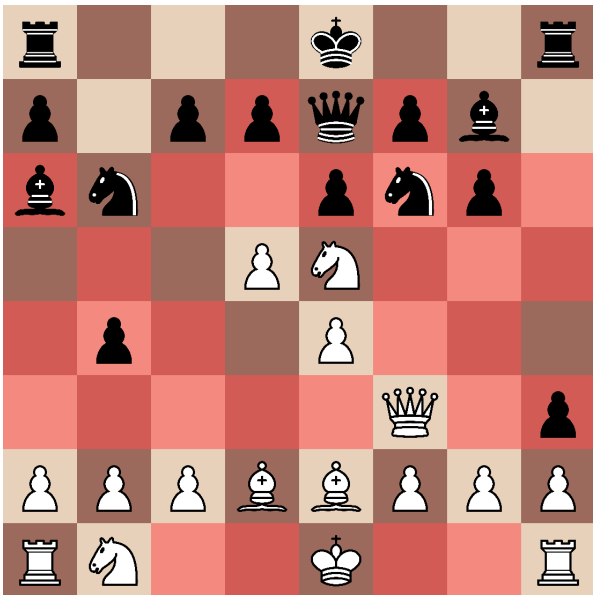


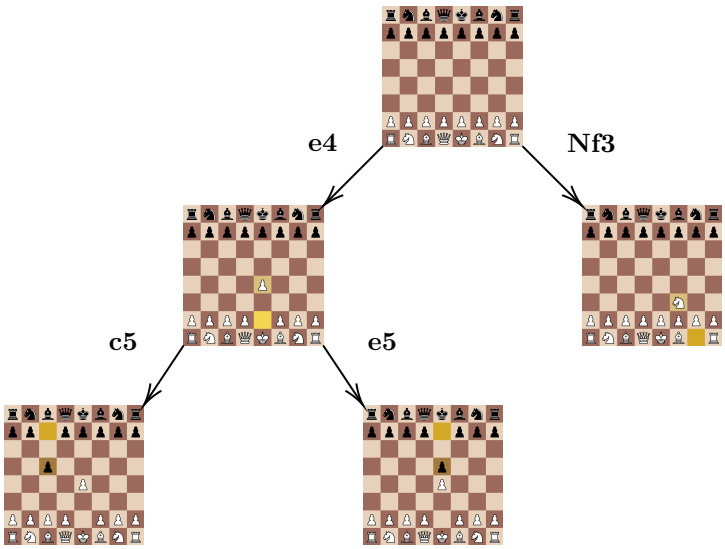
FIGURE 2 – En rouge, les cases pouvant être atteintes en 1 coup par une pièce blanche

3 Test

3.1 Le test de performance (perft)

Pour vérifier la correction de la fonction `generateMoves`, j'ai réalisé des tests de performances (PERFormance Test) en m'aidant du moteur d'échec StockFish.

On va ici associer au jeu des échecs un arbre de jeu. Chaque position va représenter un ou plusieurs nœuds de l'arbre et un nœud sera parent d'un autre s'il existe un coup légal depuis la position parente amenant à la position enfant :



Ainsi, une position pourra être représentée par plusieurs nœuds dans l'arbre, car une même position peut être atteinte par des séries de coups différentes.

De la même manière que dans un arbre classique, on définit ici la profondeur d'un nœud par le nombre de coups permettant de l'atteindre depuis la racine, c'est-à-dire la position initiale.

On va ainsi chercher le nombre de nœuds dans l'arbre à partir d'une certaine position initiale avec une profondeur fixée et le comparer à la valeur fournie par Stockfish pour un même test. Pour cela, on va utiliser la fonction récursive `perft` de pseudo-code suivant :

```
fonction perft(profondeur):
    si profondeur = 0:
        retourner 1
    nbNoeuds = 0
    coups = generateMoves()
    pour chaque coup de coups:
        makeMove(coup)
        nbNoeuds += perft(profondeur - 1)
        unmakeMove(coup)
    retourner nbNoeuds
```

3.2 Résultats de perft

En partant de la position de départ d'une partie d'échec, voici le nombre de noeuds obtenu avec `perft` pour différentes profondeurs :

Profondeur	Nombre de noeuds
0	1
1	20
2	400
3	8902
4	197281
5	4865609
6	119060324

TABLE 1 – Test de performance réalisé depuis la position de départ d'une partie d'échec en fonction de la profondeur

Ces valeurs concordent avec celles générées par Stockfish et les valeurs admises par la communauté échiquienne. D'autres perft réalisés sur d'autres positions initiales ont ainsi permis de confirmer la validité de `generateMoves`.

4 Recherche du meilleur coup

4.1 Méthode de fonctionnement

Maintenant que les règles des échecs sont implémentées, on peut tenter de trouver, à partir d'une position, le meilleur coup à jouer. Pour cela, on va faire une recherche en profondeur dans l'arbre de jeu présenté dans la partie précédente pour trouver le coup maximisant une certaine métrique. Pour définir cette métrique, on va utiliser une fonction d'évaluation assignant à chaque position un score quantifiant la qualité d'une position (plus une position rapproche de la victoire, plus elle est de qualité). On va ici se concentrer sur la réalisation de deux fonctions : une fonction `search` et une fonction `evaluate`.

4.2 Evaluation

Étant donné la taille de l'arbre de jeu (une taille d'environ 10^{120} d'après [5]), il n'est pas envisageable de faire une recherche exhaustive pour trouver le meilleur coup comme il serait possible de le faire au morpion par exemple (L'arbre de jeu ayant pour taille $9! = 362880$). On va donc se servir

d'une heuristique pour pouvoir limiter la recherche à une profondeur fixée.
Une première appoche simpliste pour l'heuristique consiste à évaluer la valeur d'une position en considérant seulement quelles pièces sont présentes sur l'échiquier, indépendamment de leur positionnement sur ce dernier. Je vais ainsi attribuer à chaque pièce une valeur définie et faire la somme de ces valeurs pour déterminer la qualité d'une position. En utilisant les valeurs suivantes :

Pièce	Valeur
Pion	100
Cavalier	320
Fou	330
Tour	500
Dame	900
Roi	20000

TABLE 2 – Valeur des pièces proposée dans [4]

On détermine alors l'évaluation d'une position avec le pseudo-code suivant :

```
eval = 0
pour chaque case de l'échequier:
    si la case est non vide:
        valeur = valeur de la piece
        si la piece est de ma couleur:
            eval += valeur
        sinon
            eval -= valeur
retourner eval
```

4.3 Recherche

4.3.1 L'algorithme du négamax

Comme précisé dans l'introduction de cette partie, on va assigner à chaque position une métrique que l'on va chercher à maximiser.

Cette métrique à laquelle on va se référer par le terme score va dans un premier temps indiquer les chances du camp blanc de gagner à partir de cette position. Ce score va être défini inductivement de la manières suivante : Pour une position P et l'ensemble $E(P)$ des positions accessibles depuis P (les enfants de P dans l'arbre de jeu), on définit le score de P, $s(P)$ par :

$$s(P) = \begin{cases} +\infty & \text{si le roi noir est échec et mat sur } P \\ -\infty & \text{si le roi noir est échec et mat sur } P \\ \max\limits_{P' \in E(P)} s(P') & \text{si c'est le tour des blancs sur } P \\ \min\limits_{P' \in E(P)} s(P') & \text{si c'est le tour des noirs sur } P \end{cases}$$

Cette métrique a du sens du fait que la partie se finisse quand un des rois est en échec et mat et que le meilleur score que l'on puisse atteindre depuis une position correspond au maximum des scores pour les blancs (on cherche le mouvement maximisant la victoire des blancs) et au minimum pour les noirs (on cherche le mouvement minimisant la victoire des blancs).
Le jeu d'échec étant à somme nulle (les blancs ont autant de chance de gagner depuis une position que les noirs risquent de perdre), on peut changer cette métrique pour simplifier la définition du score. On ne va cette fois-ci plus définir le score en fonction des chances de victoire des blancs, mais relativement à la couleur qui doit jouer sur la position. Ainsi, un score positif sur une position où les noirs doivent jouer signifie que les noirs sont sur une position gagnante.
Finalement, en remarquant que $\forall a, b \in \mathbb{R}, \min(a, b) = -\max(-a, -b)$, on peut simplifier le classique algorithme du minimax en l'algorithme du négamax en utilisant le fait que le score des blancs est l'opposé du score des noirs avec cette nouvelle définition du score :

```
initialiser meilleurcoup

fonction negamax():
    si echec et mat:
        retourner -20000
    bestEval = -∞
    coups = genereateMoves()
    pour chaque coup dans coups:
        makeMove(coup)
        eval = -negamax()
        unmakeMove(coup)
    retourner bestEval
```

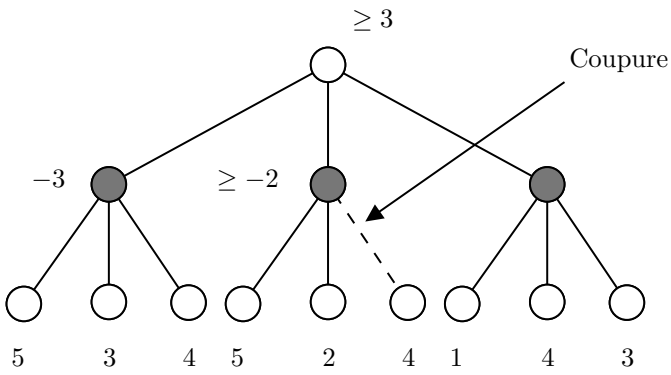
On voit ainsi que pour obtenir le score, on cherche le maximum de l'opposé des scores des enfants de la position dans l'arbre, ce qui correspond bien au minimum des scores d'après la formule précédente. Ce score quantifie donc les chances de victoires de la position qui doit jouer, car maximiser le score revient à minimiser le score de l'ennemi.
Un problème avec l'algorithme précédent est qu'il parcourt l'entièreté de l'arbre de jeu, ce dernier pouvant atteindre des tailles astronomiques. On va donc limiter la recherche à une profondeur fixée et estimer le score des feuilles par la fonction d'évaluation introduite précédemment :

```
initialiser meilleurcoup

fonction negamax(profondeur):
    si profondeur = 0:
        retourner evaluate()
    si echec et mat:
        retourner -20000
    bestEval = -∞
    coups = genereateMoves()
    pour chaque coup dans coups:
        makeMove(coup)
        eval = -negamax(profondeur)
        unmakeMove(coup)
        si eval > bestEval:
            bestEval = eval
            si profondeur = profondeur_maximale:
                // On appelle minimax initalement sur profondeur_maximale
                meilleurCoup = coup

    retourner bestEval
```

4.3.2 Élagage $\alpha - \beta$

Dans le parcours de l'arbre de jeu par l'algorithme du négamax, parcourir certaines parties de l'arbre est inutile en connaissant les scores des nœuds explorés au préalable.
Pour éviter ces nœuds inutiles, on va se donner une ''fenêtre" des scores envisageables que l'on va mettre à jour au fur et à mesure du parcours. Si on rencontre une position dont le score est en dehors de la fenêtre, on va arrêter d'explorer l'arbre à partir de cette position : on élague la branche. Notons α la borne inférieure de cette fenêtre et β la borne supérieure.

Dans l'exemple suivant, la fenêtre nous permet de réaliser une coupure car le score des noirs sera trop élevé pour pouvoir être une position choisie par les blancs :



Cette fenêtre sera implémentée dans la fonction `search` en l'appelant initialement avec $\alpha = -\infty$ et $\beta = \infty$ et en modifiant récursivement l'intervalle selon les évaluations rencontrées :

```
fonction search(profondeur, alpha, beta):
    si profondeur = 0:
        retourner evaluate()
    si echec et mat:
        retourner -20000
    bestEval = -∞
    coups = genereateMoves()
    pour chaque coup dans coups:
        makeMove(coup)
        eval = -search(profondeur, -beta, -alpha)
        unmakeMove(coup)
        si eval >= beta: // Coupure
            retourner beta
        si eval > alpha:
            alpha = eval

    retourner alpha
```

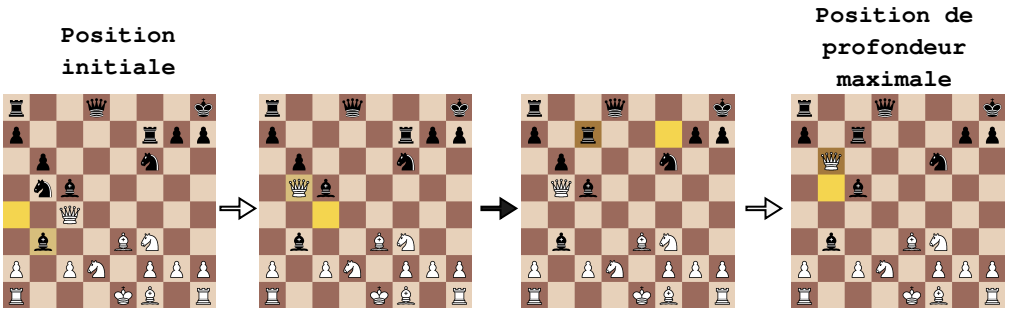
4.3.3 Tri des mouvements

Grâce à l'élagage $\alpha - \beta$, l'algorithme de recherche ne parcourt qu'une partie de l'arbre de jeu. Ainsi, l'algorithme devient d'autant plus efficace que les mouvements sont classés du meilleur au pire. En effet, dans le pire des cas, les mouvements seraient ordonnés de telle sorte qu'aucun élagage ne soit possible et l'on se ramènerait alors à l'algorithme du négamax naïf. Pour améliorer les performances de `search`, je vais donc tenter de classer les mouvements selon une heuristique.
L'idée derrière l'heuristique sera d'assigner à chaque mouvement légal une valeur, valeur d'autant plus élevée que le mouvement semble à priori de qualité comme par exemple : les captures d'une pièce de grande valeur par une pièce de petite valeur, les promotions ou encore les mouvements limitant le nombre de cases contrôlées ennemies. Je vais ensuite classer les mouvements selon ces valeurs, ce classement se faisant à l'aide d'un tri, donc en $\mathcal{O}(n \log(n))$ avec n le nombre de mouvements légaux disponibles depuis la position. L'arité de 35 des positions permet de supposer $n \simeq 35$ et donc que le tri s'effectue en un coût constant. Ce coût en temps supplémetaire est compensé par l'élagage en découlant, rendant la fonction `search` plus efficace qu'elle ne l'était auparavant (Voir le comparatif de la table 3).

4.4 L'effet d'horizon

Si le moteur arrive désormais à donner des coups sensés pour la plupart des positions, ce dernier peut parfois donner de très mauvais coups, et ce, à cause de l'effet d'horizon.

Considérons la situation suivante :



On se place ici à la profondeur 3. Le meilleur coup trouvé par l'algorithme découle d'une recherche commençant par une première capture d'un cavalier, puis celui d'un pion. L'évaluation du premier coup sera donc très élevée, car deux captures ont été réalisées. Or, dans la dernière position explorée, un pion noir menace la reine, qui pourra être capturée au coup suivant. Le coup que `search` aura considéré comme très bon est en fait très mauvais, car il résultera de la perte de la reine.

On ne peut donc pas simplement évaluer la position une fois la profondeur maximale atteinte, dû à la possible instabilité de la position finale : c'est l'effet d'horizon. On va donc lancer une nouvelle recherche une fois la profondeur maximale atteinte pour pallier cette instabilité. Cette recherche, que l'on va appeler recherche silencieuse (QuiescenceSearch en anglais), va continuer d'explorer l'arbre de jeu en ne considérant que des coups pouvant changer radicalement l'évaluation d'une position (par exemple la capture de la reine par le pion noir).

La recherche silencieuse va uniquement considérer les mouvements de capture : mouvements prompts à changer la valeur de la position, et ce, sans profondeur maximale, au risque d'explorer une grande partie de l'arbre de jeu. Mais cette dernière possibilité a peu de chances de se produire, car le nombre de captures a tendance à diminuer à rapidement au fur et à mesure qu'on les joue. En effet, si le nombre de coups moyens (l'arité moyenne d'une position dans l'arbre de jeu) est de 35, le nombre de captures seulement est bien moindre, ce qui permet à la recherche silencieuse, malgré l'absence de profondeur maximale, de terminer en un temps raisonnable.
Cette recherche silencieuse, malgré son coût temporel (environ 70% des nœuds visités sont explorés pendant la recherche silencieuse), améliore grandement la qualité de jeu du moteur.

5 Amélioration du moteur : Le début de jeu

Dû au nombre important de coups pouvant être joués depuis une position de début de jeu, la fonction `search` ne permet pas d'atteindre des profondeurs suffisantes pour obtenir une compréhension correcte d'une position. Ainsi, le moteur va simplement bouger des pièces de manière aléatoire, sans tenter de développer ses pièces ou d'occuper l'échiquier.
Pour palier à ces faiblesses, on va modifier la fonction `evaluate` pour encourager les pièces à se développer tout en donnant une connaissance plus académique des ouvertures au moteur.

5.1 Modification de la fonction `evaluate`

Pour encourager le développement des pièces, je vais, pour chaque pièce, attribuer à chaque case un score en fonction de l'intérêt stratégique d'avoir cette pièce sur cette case. J'utilise pour cela des tables de valeurs pour chaque pièces proposées par Ronald Friederich pour son moteur d'échecs RofChade comme présenté dans [3].
Voici par exemple les valeurs attribuées pour le cavalier :

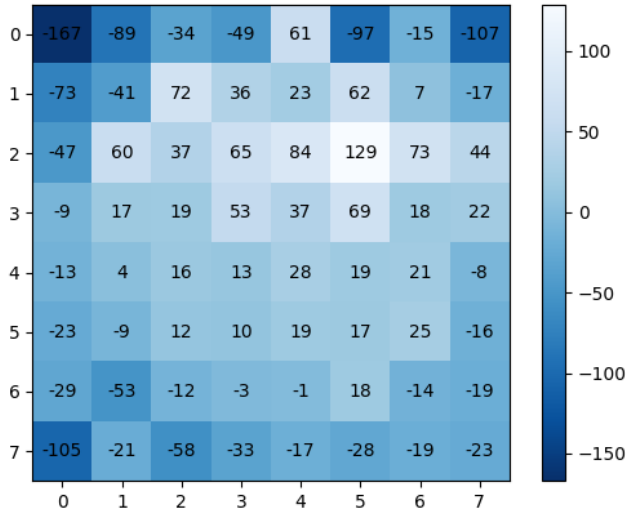


FIGURE 3 – Table de valeurs pour le cavalier

On voit ici qu'on favorise les cases centrales pour le cavalier, celles lui donnant plus de mobilité et de contrôle sur l'échiquier. De même pour toutes les autres pièces, on utilise des tables de même nature.

Le code à ajouter à la fonction d'évaluation sera donc :

```
pour chaque case de l'échequier:
    si la case est non vide:
        valeurCase = valeur de la case pour la pièce occupant la case
        si la piece est de ma couleur:
            eval += valeurCase
        sinon
            eval -= valeurCase
```

En plus d'être d'avoir un coût temporel peu élevé (une addition pour chaque pièce de l'échiquier), cette méthode permet au moteur d'acquérir des notions simplistes de positionnement des pièces.

5.2 Ouvertures

Pour compenser le manque de savoir académique du moteur, on va mettre à sa disposition un livre d'ouvertures lui permettant de se baser sur des réponses connues pour les quelques premiers coups. En effet, en dépit de `search` et `evaluate`, le moteur ne pourra jamais atteindre les profondeurs nécessaires pour savoir si tel ou telle ouverture lui permettra d'acquérir un avantage stratégique au long terme. Ce livre prendra la forme d'une compilation de débuts de parties de grands maîtres où, si le programme détecte la position actuelle dans le livre, choisira aléatoirement une partie où elle apparait et jouera le coup suivant. En plus de son impact positif sur la qualité de jeu, cette méthode a également l'avantage de se faire en un coût constant faible.

6 Amélioration du moteur : La fin de jeu

Comme pour le début de jeu, le moteur a des difficultés en fin de jeu, notamment pour conclure une partie. Comme il n'y a plus (ou presque) de captures en fin de jeu, la fonction `evaluate` sera peu pertinente en ces conditions et la profondeur ne sera pas assez élevée pour directement voir l'échec et mat.

6.1 Heuristique de fin de jeu

Pour permettre l'échec et mat, il faut limiter le plus possible les possibilités de mouvements du roi adverse. On va donc valoriser les positions où le roi adverse est proche du bord au sens de la distance de Manhatan (cette distance correspondant à la distance induite par $\|.\|_1$) :

```
distCentreBlanc = abs((positionRoiBlanc % 8) - 3) + abs((positionRoiBlanc / 8) - 3)
distCentreBlanc = abs((positionRoiBlanc % 8) - 3) + abs((positionRoiBlanc / 8) - 3)
si aux blancs de jouer:
    eval += distCentreNoir - distCentreBlanc
sinon:
    eval += distCentreBlanc - distCentreNoir
```

En fin de partie, les pièces étant en nombre bien plus faible, le roi peut devenir une pièce d'attaque, je vais donc valoriser des positions où les rois sont proches afin d'augmenter les chances d'échec et mat :

```
eval += abs((myKingPos%8) - (opponentKingPos%8)) + abs((myKingPos/8) - (opponentKingPos/8))
```

6.2 Les transpositions

En fin de partie, les captures étant plus rares, la majorité des coups possibles sont des déplacements de pièces. Or, les déplacements sont les coups les plus prompts à faire émerger à ce que l'on appelle des transpositions. Une transposition est une position atteinte par deux séries de coups différentes.

Or, dans l'étape de recherche, rencontrer une transposition signifie refaire des calculs ayant déjà été effectué précédemment, la position étant la même. Pour rendre la fonction de recherche plus efficace, je vais donc implémenter une technique de mémoïsation au travers d'une table de hachage. On va associer à chaque position un hash et sauvegarder l'évaluation d'une position ainsi que le meilleur coup lui étant associé dans une table indicée par ce hash (modulo la taille de la table).

6.2.1 Utilisation des informations stockées

On va implémenter une transposition sous la forme suivante :

```
struct Transposition
{
    uint64_t key;
    char depth;
    int value;
    int bestMove;
    char nodeType;
    bool isValid = false;
};
```

Ici, `key` représente le hash de la transposition, `value` son évaluation, `depth` la profondeur à laquelle elle a été rencontrée et `nodeType` le type de nœud auquel appartient la transposition : α, β ou exact. Le type de nœud va dépendre du contexte où est rencontré le nœud : lors d'une coupure α ou β ou lorsque le nœud n'est pas élagué.
Pendant l'exécution de `search`, on va ajouter une position à la table des transpositions dès que le nœud correspondant est élagué ou que `search` a fini de s'exécuter récursivement sur tous les fils de la position.

Lors d'une coupure β, on rajoute la transposition avec un nœud β, si une nouvelle meilleure évaluation est trouvée dans `search`, le nœud sera exact et sinon, le nœud sera de type α (coupure α).

Pour pouvoir utiliser les résultats de la table de transposition, à chaque position rencontrée, on va tester si son hash appartient à la table puis, en fonction du type de nœud de la transposition et sa profondeur, renvoyer ou non la valeur de cette dernière. En effet, si la transposition a été rencontrée à une profondeur plus faible que la profondeur actuelle, celle-ci ne doit pas être utilisée, car découlant d'une vision plus limitée de la position. De même, un nœud α (resp. β) ne donne qu'un majorant (resp.

minorant) de la véritable valeur de la position et ne doit être utilisé que si sa valeur est plus petite (resp. plus grande) que la valeur de α (resp. β) actuelle.

Le pseudo code de la fonction `search` devient alors :

```
fonction search(profondeur, alpha, beta):
    type_noeud = α
    si profondeur = 0:
        retourner evaluate()
    si echec et mat:
        retourner -20000
    si position dans table des transposition et transposition utile:
        retourner transposition.value
    bestEval = -∞
    coups = genereateMoves()
    pour chaque coup dans coups:
        makeMove(coup)
        eval = -search(profondeur, -beta, -alpha)
        unmakeMove(coup)
        si eval >= beta: // Coupure
            ajouter transposition β beta profondeur
            retourner beta
        si eval > alpha:
            type_noeud = exact
            alpha = eval

    ajouter transposition type_noeud alpha profondeur
    retourner alpha
```

6.2.2 Génération d'un hash

Pour accéder rapidement aux positions stockées dans la table de transposition, je vais assigner à chaque position un hash sous la forme d'un entier de 64 bits. Ce dernier devra être rapide à calculer et uniformément distribué sur l'ensemble des mots de 64 bits. Pour calculer ce hash, je vais utiliser la méthode du hash de Zobrist introduite dans [6] par Albert L. Zobrist.

J'associe d'abord un entier pseudo aléatoire à chaque élément possible présent sur le plateau permettant de le définir de manière unique. Ici, un élément sera une caractéristique du plateau dont l'ensemble permettra de définir ce même plateu de manière unique. Je vais donc générer un nombre pour chaque type de pièce sur chaque case, mais également pour les possilités de roque ainsi que la colonne de prise en passant et finalement la couleur qui doit jouer. Il faut donc générer $64 \cdot (6+6) + 4 + 8 + 1 = 781$ nombres pseudo aléatoires de 64 bits.

Pour calculer le hash de Zobrist d'une position, on réalise l'opération ou exclusif binaire (XOR binaire) de tous les éléments présents sur le plateau. En notant $\wedge$ le XOR binaire et $a = a_0...a_n$ et $b = b_0..b_n$ des mots binaires : $a \wedge b = (a_0 \oplus b_0)...(a_n \oplus b_n)$ avec $\oplus$ le ou exclusif.

Ainsi, le calcul du hash de Zobrist peut être réalisé rapidement dans la fonction `makeMove` en ''XO-Rant" le hash actuel avec l'entier pseudo aléatoire du nouvel élément découlant de `makeMove`. Retirer un élément du plateau du hash est également aisé du fait que l'opération XOR binaire est involutive et commutative. Ainsi, pour retirer un élément du hash, il suffit de ''XORer" de nouveau l'entier pseudo aléatoire de l'élément au hash.
Le hash ne doit donc pas être recalculé entièrement à chaque mouvement, il suffit de réaliser quelques opérations XOR binaires dans `makeMove` et stocker le hash dans `GameState` afin de réaliser facilement l'opération `unmakeMove`.

6.2.3 Remarques sur la mémoïsation des transpositions

Un problème découlant du hachage précédent est l'espace des possibilités du hash. Comme entier de 64 bits, il y a $2^{64} \simeq 10^{19}$ hash possibles contre les 10^{43} positions d'échecs possibles calculées par Claude Shannon dans son article fondateur de la programmation échiquéenne [5] : il y a donc des risques de collisions , plusieurs positions d'échec peuvent avoir le même hash de Zobrist. Or le nombre de nœuds explorés pendant la recherche est de l'ordre de plusieurs millions, un nombre très petit devant le nombre de hachage possibles, ce qui garantit un risque de collision faible en supposant le hash uniformément distribué sur l'ensemble des entiers de 64 bits, ce que l'on tente de mettre en œuvre en choisissant des entiers pseudo-aléatoires.

En réalité, malgré les risques de collisions, le gain temporel octroyé par cette méthode rend l'impact des collisions négligeables sur la complexité totale.

Cette technique de mémoïsation permet ainsi d'explorer à des profondeurs bien plus grandes en fin de partie, passant d'une profondeur de 13 à 25 dans certaines positions.

7 Jouer contre des joueurs : La contrainte de temps

Pour pouvoir évaluer la qualité du moteur, il faut le faire jouer contre d'autres joueurs / moteurs afin de pouvoir lui attribuer un score dans le système d'évaluation ELO. Or, pour pouvoir jouer, il faut que le moteur puisse décider du meilleur coup en temps limité. Ce dernier ne peut pas seulement jouer en explorant à une profondeur fixée : une recherche à même profondeur peut mettre un temps très variable en fonction de la position de départ. Ainsi, une recherche à profondeur 10 s'exécutera en une poignée de millisecondes pour une position de fin de jeu et plus de 5 minutes pour une position de début de jeu.

7.1 L'approfondissement itératif

Pour permettre au moteur de jouer en temps limité, on va utiliser la technique de l'approfondissement itératif (iterative deepening en anglais). Au lieu de chercher à une profondeur fixée, on va lancer des recherches à des profondeurs de plus en plus élevées en gardant en mémoire les meilleurs coups à chaque profondeur jusqu'à atteindre le temps imparti. Si une recherche n'a pas eu le temps de se terminer à temps, on renvoie le meilleur coup de la recherche précédente. Ainsi, le moteur renverra le meilleur coup avec la plus grande profondeur permise par le temps imparti et la position initiale
Cette méthode nécessite d'effectuer des recherches de profondeurs croissantes. Or, on ne peut pas utiliser des résultats des recherches précédentes pour les recherches suivantes, ces dernières ayant été faites à des profondeurs moindres. On doit donc refaire tout le travail effectué aux recherches précédentes. Cette méthode semble donc avoir un coût temporel important.

Or, en réalité, ce coût n'est pas si important.

7.1.1 Nature exponentielle de la recherche

Premièrement, le nombre de positions évaluées à chaque profondeur augmente de façon exponentielle avec la profondeur. En supposant toujours que l'arité moyenne dans l'arbre de jeu est de 35, en notant p la profondeur, le nombre de nœuds à la profondeur p est un $\mathcal{O}(35^p)$. Ainsi, en supposant que la profondeur maximale atteinte lors d'une recherche par approfondissement itératif soit de p, le nombre de nœuds explorés aux profondeurs $1, 2, ..., p-1$ représentera seulement environ $\frac{1}{35}$ ème du nombre total de nœuds explorés, montrant que le coût des recherches précédentes est faible par rapport à la dernière recherche.

7.1.2 Amélioration du tri des mouvements

Pour rendre l'approfondissement plus rapide, on peut de plus exploiter les résultats des recherches aux profondeurs moindres pour rendre plus rapide la recherche suivante. Ce gain de temps est réalisé en explorant en premier les meilleurs coups trouvés à la profondeur précédente dans la recherche de profondeur suivante.

Pour chaque recherche, on va calculer la variation principale (Principal Variation) de la recherche : c'est-à-dire la suite des meilleurs coups calculés à partir de chaque position. Cette variation principale peut être calculée très simplement en utilisant le calcul de transpositions mis en place précédemment. Chaque transposition stockant le meilleur mouvement à partir de la position, il est possible de trouver la principale variation avec le pseudo-code suivant :

```
initialiser la liste coups

pour chaque entier i de 1 à la profoneur maximale atteinte lors de la recherche:
    t = obtenir la transposition associée au hash de la position actuelle
    PV[i] = t.bestMove // On stocke le ième coup de la variation principale
    ajouter t.bestMove à coups
    makeMove t.bestMove


pour chaque coup dans coups:
    unmakeMove coup
```

À cette variation principale, on va assigner un score infini dans le tri des mouvements afin que ces dernières soient évaluées en premier, amortissant le coût temporel de l'approfondissement itératif. En effet, les meilleurs coups de la recherche précédente ont des chances de rester bons, malgré le fait que l'on cherche plus profondément.

8 Résultats

Pour déterminer le niveau final du moteur, je me base sur le système ELO, un modèle mathématique assignant un score aux joueurs dépendant de résultats de parties contre d'autres joueurs détaillé en annexe.
Une estimation rapide de ce niveau peut être réalisée en faisant passer au moteur le STS (Strategic Test Suite) [1], un ensemble de 1500 positions regroupées dans 15 thèmes évaluant les compétences stratégiques du moteur dans diverses situations. À chaque position est associé un ensemble de mouvements ''solutions" possédant chacun un score de 1 à 10. Le score du moteur est augmenté quand il trouve un mouvement solution et son score total permet de calculer une estimation de son ELO.
Voici les résultats du moteur final avec un temps de 5 secondes passé par position :

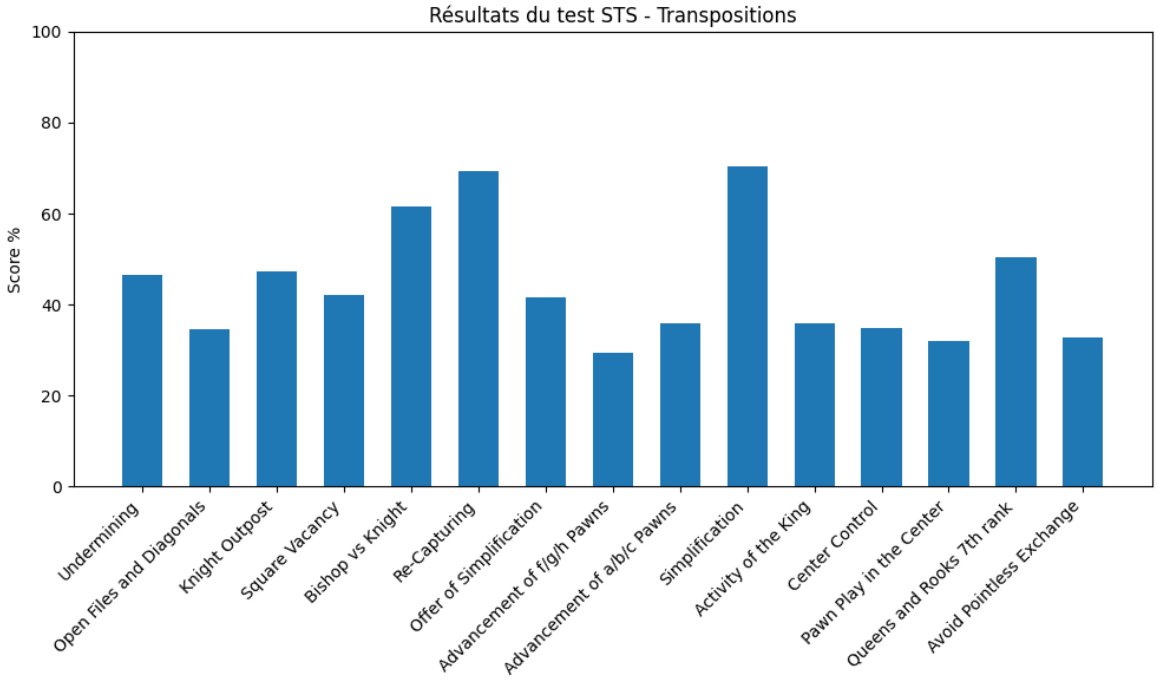


FIGURE 4 – Résultats du test STS

On en déduit alors que le moteur possède un ELO approché de 1730, ce qui correspond au niveau d'un bon joueur d'échec.

Pour confirmer ou infirmer cette valeur, le moteur doit jouer un grand nombre de parties contre des adversaires, eux aussi, notés dans le système ELO. Pour réaliser un grand nombre de parties, j'ai placé mon moteur sur le serveur d'échecs Lichess (lichess.com) sous le nom de mcngnt et organisé des parties contre d'autres robots du serveur. Ces parties se sont effectuées en temps limité, le serveur de Lichess communiquant avec le moteur en utilisant le protocole UCI détaillé en annexe, l'issue de la partie augmentant ou diminuant l'ELO du moteur en fonction de sa différence d'ELO avec son adversaire. Après environ 150 parties, l'ELO du moteur s'est stabilisé aux alentours de 1725 ELO, concordant avec l'estimation précédente.

9 Conclusion

9.1 Comparaison aux moteurs modernes

En dépit des résultats concluants du moteur (il bat la plupart des compétiteurs humains amateurs d'échecs non professionnels), il reste tout de même très en dessous du niveau moyen des moteurs d'échecs modernes avec une différence d'environ 1000 ELO.

Pour comparer, le moteur d'échec le plus reconnu, StockFish, possède un ELO de 3500 soit une différence de plus de 2000 points ELO.

9.2 Défauts et améliorations possibles

Comme on peut le voir avec les résultats des STS dans la partie précédente, le moteur a des difficultés dans la gestion de la structure des pions ainsi que dans le contrôle du plateau, le poussant à parfois faire des erreurs stratégiques.

Pour améliorer ses performances, on pourrait améliorer la fonction d'évaluation, par exemple en encourageant les pions à se développer et considérer les pions menacés, doublés ou isolés. On pourrait

de plus valoriser les coups augmentant le contrôle de l'échiquier ou protégeant le roi.

La recherche pourrait être également améliorées en le rendant plus rapide : par exemple en optimisant la représentation de l'échiquier ainsi que la génération des coups légaux. D'autres formes d'élagages peuvent aussi être considérés, par exemple l'élagage du mouvement nul [2].

10 Bibliographie

Références

[1] Dann Corbit and Swaminathan Natarajan. Strategig test suite. https://sites.google.com/site/strategictestsuite/about, 2010.

[2] Omid David-Tabibi and Nathan S. Netanyahu. Verified null-move pruning. https://arxiv.org/pdf/0808.1125.pdf, 2008.

[3] Ronald Friederich. Pesto's evaluation function. https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function, 2021.

[4] Tomasz Michniewski. Simplified evaluation function. https://www.chessprogramming.org/Simplified_Evaluation_Function, 2021.

[5] CLAUDE E. SHANNON. Programming a computer for playing chess. https://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf, 1949.

[6] Albert L. Zobrist. A new hashing method with applicaion for game playing. https://research.cs.wisc.edu/techreports/1970/TR88.pdf, 1970.

11 Annexe

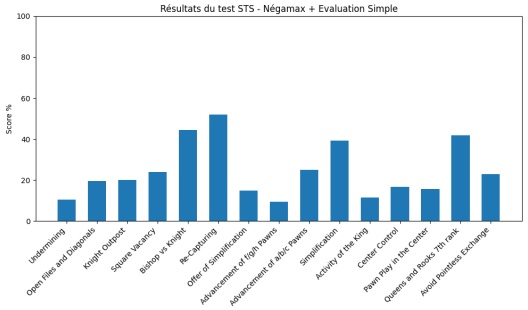
11.1 Niveau du moteur après chaque amélioration

Ci-dessous une table donnant le score ELO approché du moteur en utilisant le STS [1] après chaque amélioration présentée dans ce rapport avec un temps de 100 ms par position en utilisant une recherche itérative :

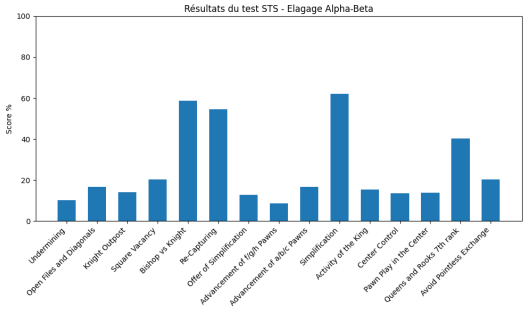
Amélioration	Score ELO
Négamax + Evaluation Simple	850
Elagage Alpha-Beta	880
Tri des mouvements	893
Recherche Silencieuse	927
Table de valeurs pour les pièces	1377
Heuristique de fin de jeu	1392
Transpositions	1447

TABLE 3 – Score ELO aprés chaque amélioration

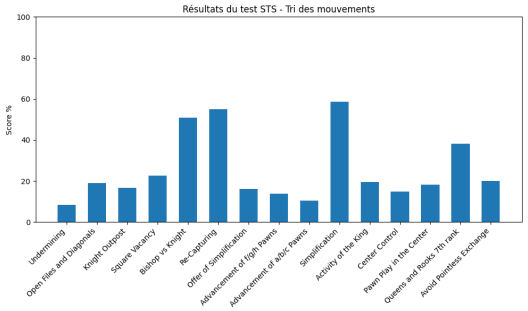
Je fournis également les résultats des tests STS effectués aprés l'implémentation de chaque amélioration afin de voir leur impact stratégique :



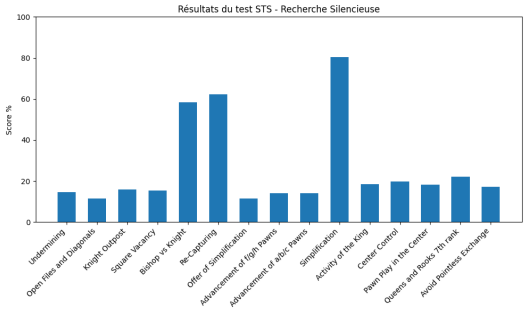
(a) Négamax + Evaluation Simple - ELO 850



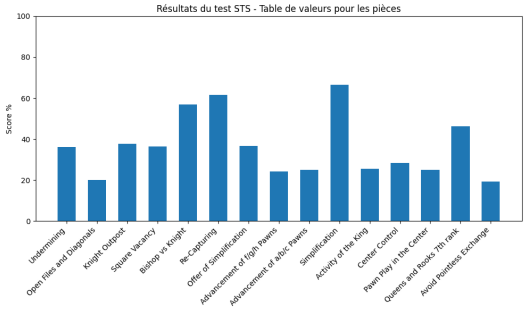
(b) Elagage Alpha-Beta - ELO 880



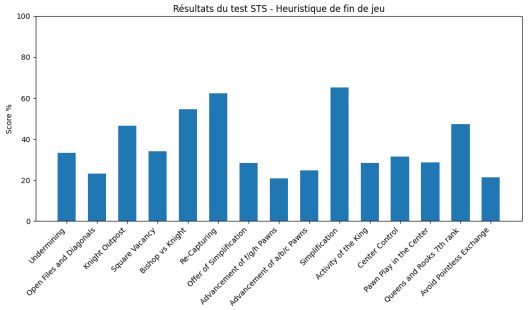
(c) Tri des mouvements - ELO 893



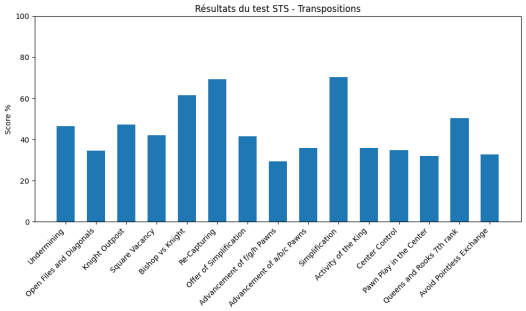
(d) Recherche Silencieuse - ELO 927



(e) Table de valeurs pour les pièces - ELO 1377



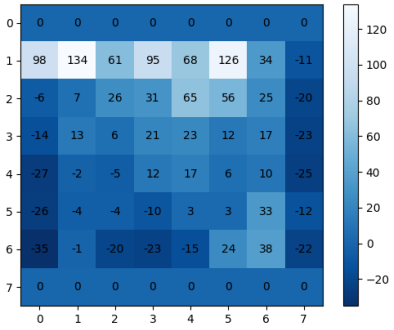
(f) Heuristique de fin de jeu - ELO 1392



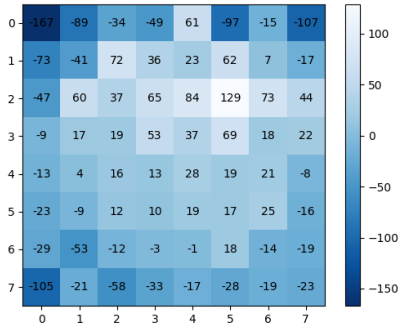
(g) Transpositions - ELO 1447

FIGURE 5 – Résultats du STS aprés chaque amélioration

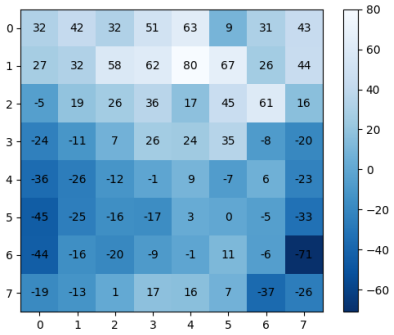
11.2 Tables de valeurs pour chaque pièce



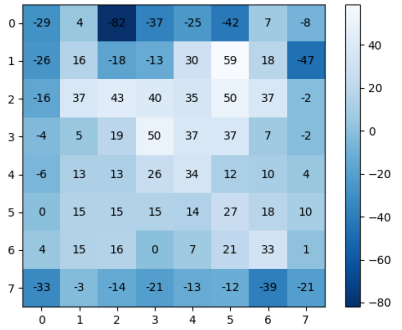
(a) Pion



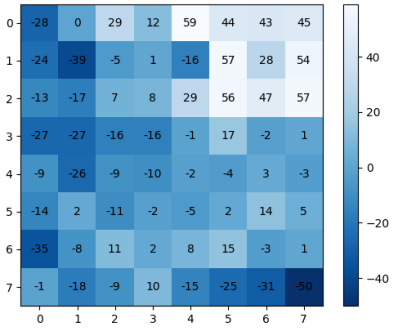
(b) Cavalier



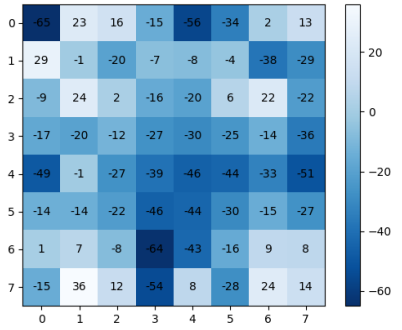
(c) Tour



(d) Fou



(e) Dame



(f) Roi

FIGURE 6 – Tables de valeurs proposées dans [3]

11.3 Le système d'évaluation ELO

Le système d'évaluation ELO est un modèle mathématique utilisé pour noter des joueurs sur la base des résultats de matchs contre d'autres joueurs. Le système garantit qu'un joueur plus fort recevra une note plus élevée qu'un joueur plus faible, la différence de points ELO renseignant sur la probabilité de victoire de chacun des joueurs. Si un joueur réalise une performance supérieure (resp. inférieure) à son score ELO estimé, il gagne (resp. perd) des points ELO.

Un joueur de tournoi sérieux de bas niveau aurait un score d'environ 1800. Magnus Carlsen, le champion du monde d'échecs en titre, a un score ELO de 2853. Les meilleurs moteurs d'échecs (Comme StockFish) ont des scores supérieurs à 3200.

11.4 Le protocole UCI

Le protocole UCI (Universal Chess Interface) est un protocole utilisé par les moteurs d'échecs afin de communiquer avec d'autres systèmes (par exemple une interface graphique ou des plateformes d'échecs pour jouer des parties). La communication se fait au travers de l'entrée et la sortie standard, le moteur recevant et envoyant des commandes.

11.5 Code source

J'inclus ci-dessous les différentes classes composant mon programme, à l'exception du code gérant l'interface graphique (utilisée uniquement pour la production des images de ce TIPE et pour le débogage) :

11.5.1 Représentation des pièces

piece.h

```
enum Pieces
{
  None = 0,
  Pawn = 1,
  Rook = 2,
  Knight = 3,
  Bishop = 4,
  Queen = 5,
  King = 6,

  White = 8,
  Black = 16
};

bool isPieceWhite(int piece);
int pieceType(int piece);
int pid(int i, int j);
```

piece.cpp

```
#include "piece.h"

bool isPieceWhite(int piece){
  return (piece >> 3) == 1;
}
int pieceType(int piece){
  return piece & 7;
}
int pid(int i, int j){
  return i + 8 * j;
}
```

11.5.2 Représentation des coups

move.h

```h
#define positionsMoveMask 4095
#define startPosMask 63
#define endPosMask 4032
#define tagMask 61440

enum Tag{
  QuietMove = 0,
  DoublePawnPush = 1,
  KingCastle = 2,
  QueenCastle = 3,
  Capture = 4,
  EPCapture = 5,
  KnightProm = 8,
  BishopProm = 9,
  RookProm = 10,
  QueenProm = 11,
  KnightPromCapture = 12,
  BishopPromCapture = 13,
  RookPromCapture = 14,
  QueenPromCapture = 15
};

int genMove(int startPos, int endPos, int tag);
int genMove(int startPosx, int startPosy, int endPosx, int endPosy, int tag);
int discardTag(int move);
int endPos(int move);
int startPos(int move);
int tag(int move);
std::string standardNotation(int move);
bool isCapturingTag(int tag);
int standPosToInt(char c1, char c2);
int standNotToMove(std::string standNot);
```

move.cpp

```cpp
#include <move.h>

int genMove(int startPos, int endPos, int tag){
  if (startPos >= 0 && startPos < 64 && endPos >= 0 && endPos < 64){
    return startPos | endPos << 6 | tag << 12;
  }
  else{
    return 0;
  }
}

int genMove(int startPosx, int startPosy, int endPosx, int endPosy, int tag){
  return (startPosx + 8 * startPosy) | (endPosx + 8 * endPosy) << 6 | tag << 12;
}
int discardTag(int move){
  return move & positionsMoveMask;
}
int endPos(int move){
  return (move >> 6) & 63;
}
int startPos(int move){
  return move & 63;
}
int tag(int move){
  return (move & tagMask) >> 12;
}
std::string standardPos(int pos){
  std::string res;
  res.push_back('a' + pos%8);
  res.push_back('0'+ 7 - pos/8 + 1);
  return res;
}
std::string standardNotation(int move){
  int t = tag(move);
  char c = ' ';
  switch (t)
 {
    case QueenProm:
      c = 'q';
      break;
    case QueenPromCapture:
```

```
      c = 'q';
      break;
    case KnightProm:
      c = 'k';
      break;
    case KnightPromCapture:
      c = 'k';
      break;
    case RookProm:
      c = 'r';
      break;
    case RookPromCapture:
      c = 'r';
      break;
    case BishopProm:
      c = 'b';
      break;
    case BishopPromCapture:
      c = 'b';
      break;
  }
  return standardPos(startPos(move)) + standardPos(endPos(move)) + c;
}
bool isCapturingTag(int tag){
  if ( (tag <= 3) || ( (8 <= tag) && (tag <= 11) )){
    return false;
  }
  else{
    return true;
  }
}
int standPosToInt(char c1, char c2){

  return (c1 - 'a') + 8 * (7 - (c2 - '1'));
}
int standNotToMove(std::string standNot){

  return genMove(standPosToInt(standNot[0],standNot[1]), standPosToInt(standNot[2],standNot[3]), 0);
}
```

11.5.3 Représentation de l'échiquier et génération des coups possibles

boardManager.h

```
#include "move.h"
#include "piece.h"

#define northMask 7
#define southMask 56
#define eastMask 448
#define westMask 3584

struct GameState{
  int capturedPiece;
  bool canWhiteKingCastle;
  bool canWhiteQueenCastle;
  bool canBlackKingCastle;
  bool canBlackQueenCastle;
  int doublePushFile;
  int moveCount;
  int whiteKingPos;
  int blackKingPos;
  uint64_t zobristKey;
  bool hasWhiteCastled;
  bool hasBlackCastled;
};
enum Directions{
  South = 8,
  North = -8,
  East = 1,
  West = -1,
  SouthWest = 7,
  NorthWest = -9,
  SouthEast = 9,
  NorthEast = -7
};
enum DirectionsID{
  NorthID = 0,
  SouthID = 1,
```

```
  EastID = 2,
  WestID = 3,
  NorthEastID = 4,
  NorthWestID = 5,
  SouthEastID = 6,
  SouthWestID = 7
};

class BoardManager{
public:
  BoardManager();

  void makeMove(int move);
  void unmakeMove(int move);

  std::vector<int> generatePseudoMoves();
  std::vector<int> generateMoves(bool onlyCaptures);

  int get(int pos);
  int get(int x, int y);
  bool isSquareEmpty(int i, int j);
  bool isSquareEmpty(int pid);
  bool isSquareFree(int i, int j);
  bool isSquareFree(int pid);
  bool isSquareEnemy(int pid);
  bool isSquareEnemy(int i, int j);
  bool isSquareFriendly(int pid);

  std::string startingFen = "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
  void loadFen(std::string fen);

  uint64_t piecesZobrist[8][2][64];
  uint64_t doublePushFileZobrist[9];
  uint64_t whiteToMoveZobrist;
  uint64_t castlingRightZobrist[4];

  void initZobrist();
  uint64_t computeZobrist();
  uint64_t zobristKey;
  std::vector<uint64_t> zobristHistory;

  void controlledSquares();
  void assign(int i, int j);
  void resetControl();

  bool isChecked();
  bool controlled[8][8];

  bool whiteToMove = true;
  int board[8][8];
  GameState currentGameState;
  std::stack<GameState> gameStateHistory;
};
```

boardManager.cpp

```
#include "boardManager.h"
{
int numSquares[64][8];

BoardManager::BoardManager(){
  for (int i = 0 ; i < 8 ; ++i){
    for (int j = 0 ; j < 8 ; ++j){
      int numNorth = j;
      int numSouth = 7 - j;
      int numEast = 7 - i;
      int numWest = i;
      numSquares[i + j*8][0] = numNorth;
      numSquares[i + j*8][1] = numSouth;
      numSquares[i + j*8][2] = numEast;
      numSquares[i + j*8][3] = numWest;
      numSquares[i + j*8][4] = min(numNorth, numEast);
      numSquares[i + j*8][5] = min(numNorth, numWest);
      numSquares[i + j*8][6] = min(numSouth, numEast);
      numSquares[i + j*8][7] = min(numSouth, numWest);
    }
  }
  currentGameState.capturedPiece = 0;
  currentGameState.canWhiteKingCastle = true;
  currentGameState.canWhiteQueenCastle = true;
```

```
  currentGameState.canBlackKingCastle = true;
  currentGameState.canBlackQueenCastle = true;
  currentGameState.hasWhiteCastled = false;
  currentGameState.hasBlackCastled = false;
  currentGameState.doublePushFile = 0;
  currentGameState.moveCount = 0;

  loadFen(startingFen);
  initZobrist();
  computeZobrist();

  currentGameState.zobristKey = zobristKey;
}

bool BoardManager::isChecked(){
  whiteToMove = !whiteToMove;
    controlledSquares();
    whiteToMove = !whiteToMove;
    if (whiteToMove){
      return controlled[currentGameState.whiteKingPos / 8][currentGameState.whiteKingPos % 8];
    }
    else{
      return controlled[currentGameState.blackKingPos / 8][currentGameState.blackKingPos % 8];
    }
}

bool BoardManager::isSquareEmpty(int i, int j){
  return get(i,j) == 0;
}
bool BoardManager::isSquareEmpty(int pid){
  return get(pid) == 0;
}
bool BoardManager::isSquareEnemy(int i, int j){
  return isPieceWhite(get(i,j)) != whiteToMove && (get(i,j) > 0);
}
bool BoardManager::isSquareEnemy(int pid){
  return isPieceWhite(get(pid)) != whiteToMove && (get(pid) > 0);
}
bool BoardManager::isSquareFree(int i, int j){
  return isPieceWhite(get(i,j)) != whiteToMove || (get(i,j) == 0);
}
bool BoardManager::isSquareFree(int pid){
  return isPieceWhite(get(pid)) != whiteToMove || (get(pid) == 0);
}
bool BoardManager::isSquareFriendly(int pid){
  return isPieceWhite(get(pid)) == whiteToMove && (get(pid) > 0);
}

void BoardManager::assign(int i, int j){
  if (i >= 0 && i < 8 && j >= 0 && j < 8){
    controlled[i][j] = true;
  }
}

void BoardManager::controlledSquares(){

  resetControl();

  for (int i = 0 ; i < 8 ; ++i){
    for (int j = 0 ; j < 8 ; ++j){
      int piece = get(i,j);
      int currentPID = pid(i,j);

      if(piece > 0 && isPieceWhite(piece) == whiteToMove){

        if (pieceType(piece) == Pawn){

          if (isPieceWhite(piece)){
            if (numSquares[currentPID][NorthEastID] >= 1 && isSquareFree(i+1,j-1)){
              assign(j-1,i+1);
            }
            if (numSquares[currentPID][NorthWestID] >= 1 && isSquareFree(i-1,j-1)){
              assign(j-1,i-1);
            }

          }
          else{
            if (numSquares[currentPID][SouthEastID] >= 1 && isSquareFree(i+1,j+1)){
              assign(j+1, i+1);
            }
            if (numSquares[currentPID][SouthWestID] >= 1 && isSquareFree(i-1,j+1)){
              assign(j+1,i-1);
```

```
            }
          }

        }

        if (pieceType(piece) == King){
          for (int dirID = 0 ; dirID <= 7 ; ++dirID){
            int targetPos = currentPID + directions[dirID];

            if (numSquares[currentPID][dirID] >= 1 && isSquareFree(targetPos)){
              assign(targetPos/8, targetPos % 8);
            }
          }
        }

        if (pieceType(piece) == Knight){
          if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][EastID] >= 1 &&
          ↪  isSquareFree(i+1,j-2)){
            assign(j-2,i+1);
          }
          if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][WestID] >= 1 &&
          ↪  isSquareFree(i-1,j-2)){
            assign(j-2,i-1);
          }
          if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][EastID] >= 1 &&
          ↪  isSquareFree(i+1,j+2)){
            assign(j+2,i+1);
          }
          if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][WestID] >= 1 &&
          ↪  isSquareFree(i-1,j+2)){
            assign(j+2,i-1);
          }
          if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][NorthID] >= 1 &&
          ↪  isSquareFree(i+2,j-1)){
            assign(j-1,i+2);
          }
          if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][SouthID] >= 1 &&
          ↪  isSquareFree(i+2,j+1)){
            assign(j+1,i+2);
          }
          if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][NorthID] >= 1 &&
          ↪  isSquareFree(i-2,j-1)){
            assign(j-1,i-2);
          }
          if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][SouthID] >= 1 &&
          ↪  isSquareFree(i-2,j+1)){
            assign(j+1,i - 2);
          }
        }

        if (pieceType(piece) == Rook || pieceType(piece) == Bishop || pieceType(piece) == Queen){

          int startDir = (pieceType(piece) == Bishop) ? 4 : 0;
          int endDir = (pieceType(piece) == Rook) ? 3 : 7;

          for (int dirID = startDir ; dirID <= endDir ; ++dirID){
            for (int i = 0 ; i < numSquares[currentPID][dirID]; ++i){
              int targetPos = currentPID + directions[dirID] * (i+1);

              if (isSquareFriendly(targetPos)){
                break;
              }

              assign(targetPos / 8, targetPos % 8);

              if (isSquareEnemy(targetPos)){
                break;
              }
            }
          }
        }
      }
    }
  }

}

void BoardManager::resetControl(){
  for (int i = 0 ; i < 64; ++i){
    controlled[i % 8][i / 8] = false;
  }
}
```

```
181
182 std::vector<int> BoardManager::generateMoves(bool onlyCaptures){
183   std::vector<int> pseudoMoves = generatePseudoMoves();
184   std::vector<int> legalMoves;
185
186   for (int pseudoMove : pseudoMoves){
187     if (onlyCaptures && !isCapturingTag(tag(pseudoMove))){
188       continue;
189     }
190     makeMove(pseudoMove);
191     whiteToMove = !whiteToMove;
192     if (!isChecked()){
193       legalMoves.push_back(pseudoMove);
194     }
195     whiteToMove = !whiteToMove;
196     unmakeMove(pseudoMove);
197   }
198   return legalMoves;
199 }
200
201
202 std::vector<int> BoardManager::generatePseudoMoves(){
203   std::vector<int> moves;
204
205   whiteToMove = !whiteToMove;
206   controlledSquares();
207   whiteToMove = !whiteToMove;
208
209   for (int i = 0 ; i < 8 ; ++i){
210     for (int j = 0 ; j < 8 ; ++j){
211       int piece = get(i,j);
212       int currentPID = pid(i,j);
213
214       if(piece > 0 && isPieceWhite(piece) == whiteToMove){
215         if (pieceType(piece) == Pawn){
216
217           if (isPieceWhite(piece)){
218             if (numSquares[currentPID][NorthID] >= 1 && isSquareEmpty(i,j-1)){
219               if (j > 1){
220                 moves.push_back(genMove(i,j,i,j-1, QuietMove));
221               }
222               else{
223                 moves.push_back(genMove(i,j,i,j-1, KnightProm));
224                 moves.push_back(genMove(i,j,i,j-1, BishopProm));
225                 moves.push_back(genMove(i,j,i,j-1, RookProm));
226                 moves.push_back(genMove(i,j,i,j-1, QueenProm));
227               }
228             }
229             if (j == 6 && isSquareEmpty(i,j-2) && isSquareEmpty(i, j-1)){
230               moves.push_back(genMove(i,j,i,j-2, DoublePawnPush));
231             }
232             if (numSquares[currentPID][NorthEastID] >= 1 && isSquareEnemy(i+1,j-1)){
233               if (j - 1 == 0){
234                 moves.push_back(genMove(i,j,i+1,j-1, KnightPromCapture));
235                 moves.push_back(genMove(i,j,i+1,j-1, BishopPromCapture));
236                 moves.push_back(genMove(i,j,i+1,j-1, RookPromCapture));
237                 moves.push_back(genMove(i,j,i+1,j-1, QueenPromCapture));
238               }
239               else{
240                 moves.push_back(genMove(i,j,i+1,j-1, Capture));
241               }
242             }
243             if (numSquares[currentPID][NorthWestID] >= 1 && isSquareEnemy(i-1,j-1)){
244               if (j - 1 == 0){
245                 moves.push_back(genMove(i,j,i-1,j-1, KnightPromCapture));
246                 moves.push_back(genMove(i,j,i-1,j-1, BishopPromCapture));
247                 moves.push_back(genMove(i,j,i-1,j-1, RookPromCapture));
248                 moves.push_back(genMove(i,j,i-1,j-1, QueenPromCapture));
249               }
250               else{
251                 moves.push_back(genMove(i,j,i-1,j-1, Capture));
252               }
253             }
254
255             if (numSquares[currentPID][NorthEastID] >= 1 && j == 3 && isSquareEnemy(i+1,j) &&
             ↪  pieceType(get(i+1,j)) == Pawn && isSquareEmpty(i+1, j-1) && currentGameState.doublePushFile - 1
             ↪  == i+1){
256               moves.push_back(genMove(i,j,i+1,j-1, EPCapture));
257             }
258             if (numSquares[currentPID][NorthWestID] >= 1 && j == 3 && isSquareEnemy(i-1,j) &&
             ↪  pieceType(get(i-1,j)) == Pawn && isSquareEmpty(i-1, j-1) && currentGameState.doublePushFile - 1
             ↪  == i-1){
```

```
259                 moves.push_back(genMove(i,j,i-1,j-1, EPCapture));
260               }
261             }
262             else{
263               if (numSquares[currentPID][NorthID] >= 1 && isSquareEmpty(i,j+1)){
264                 if (j < 6){
265                   moves.push_back(genMove(i,j,i,j+1, QuietMove));
266                 }
267                 else{
268                   moves.push_back(genMove(i,j,i,j+1, KnightProm));
269                   moves.push_back(genMove(i,j,i,j+1, BishopProm));
270                   moves.push_back(genMove(i,j,i,j+1, RookProm));
271                   moves.push_back(genMove(i,j,i,j+1, QueenProm));
272                 }
273               }
274               if (j == 1 && isSquareEmpty(i,j+2) && isSquareEmpty(i, j+1)){
275                 moves.push_back(genMove(i,j,i,j+2, DoublePawnPush));
276               }
277               if (numSquares[currentPID][SouthEastID] >= 1 && isSquareEnemy(i+1,j+1)){
278                 if (j + 1 == 7){
279                   moves.push_back(genMove(i,j,i+1,j+1, KnightPromCapture));
280                   moves.push_back(genMove(i,j,i+1,j+1, BishopPromCapture));
281                   moves.push_back(genMove(i,j,i+1,j+1, RookPromCapture));
282                   moves.push_back(genMove(i,j,i+1,j+1, QueenPromCapture));
283                 }
284                 else{
285                   moves.push_back(genMove(i,j,i+1,j+1, Capture));
286                 }
287               }
288               if (numSquares[currentPID][SouthWestID] >= 1 && isSquareEnemy(i-1,j+1)){
289                 if (j + 1 == 7){
290                   moves.push_back(genMove(i,j,i-1,j+1, KnightPromCapture));
291                   moves.push_back(genMove(i,j,i-1,j+1, BishopPromCapture));
292                   moves.push_back(genMove(i,j,i-1,j+1, RookPromCapture));
293                   moves.push_back(genMove(i,j,i-1,j+1, QueenPromCapture));
294                 }
295                 else{
296                   moves.push_back(genMove(i,j,i-1,j+1, Capture));
297                 }
298               }
299
300               if (numSquares[currentPID][SouthEastID] >= 1 && j == 4 && isSquareEnemy(i+1,j)&&
                  ↪  pieceType(get(i+1,j)) == Pawn && isSquareEmpty(i+1, j+1) && (currentGameState.doublePushFile - 1
                  ↪  == i+1)){
301                 moves.push_back(genMove(i,j,i+1,j+1, EPCapture));
302               }
303               if (numSquares[currentPID][SouthWestID] >= 1 && j == 4 && isSquareEnemy(i-1,j) &&
                  ↪  pieceType(get(i-1,j)) == Pawn && isSquareEmpty(i-1, j+1) && (currentGameState.doublePushFile - 1
                  ↪  == i-1)){
304                 moves.push_back(genMove(i,j,i-1,j+1, EPCapture));
305               }
306             }
307           }
308
309           if (pieceType(piece) == King){
310
311             for (int dirID = 0 ; dirID <= 7 ; ++dirID){
312               int targetPos = currentPID + directions[dirID];
313
314               if (numSquares[currentPID][dirID] >= 1 && isSquareFree(targetPos)){
315                 moves.push_back(genMove(currentPID, targetPos, isSquareEnemy(targetPos) * Capture));
316               }
317             }
318             if (!controlled[j][i]){
319               if ((currentGameState.canWhiteQueenCastle && whiteToMove) ||  (currentGameState.canBlackQueenCastle
                  ↪  && !whiteToMove)){
320                 bool isOK = true;
321                 if (!isSquareEmpty(1,j)){
322                   isOK = false;
323                 }
324                 for (int p = 2; p <= 3; ++p ){
325                   if (!isSquareEmpty(p, j) || controlled[j][p]){
326                     isOK = false;
327                   }
328                 }
329                 if (isOK){
330                   moves.push_back(genMove(currentPID, currentPID - 2, QueenCastle));
331                 }
332               }
333               if ((currentGameState.canWhiteKingCastle && whiteToMove) ||  (currentGameState.canBlackKingCastle &&
                  ↪  !whiteToMove)){
334                 bool isOK = true;
```

```
335                 for (int p = 5; p <= 6; ++p ){
336                   if (!isSquareEmpty(p, j) || controlled[j][p]){
337                     isOK = false;
338                   }
339                 }
340                 if (isOK){
341                   moves.push_back(genMove(currentPID, currentPID + 2, KingCastle));
342                 }
343               }
344             }
345           }
346
347           if (pieceType(piece) == Knight){
348
349             if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][EastID] >= 1 &&
               ↪  isSquareFree(i+1,j-2)){
350               moves.push_back(genMove(i,j,i+1,j-2, isSquareEnemy(i+1,j-2) * Capture));
351             }
352             if (numSquares[currentPID][NorthID] >= 2 && numSquares[currentPID][WestID] >= 1 &&
               ↪  isSquareFree(i-1,j-2)){
353               moves.push_back(genMove(i,j,i-1,j-2, isSquareEnemy(i-1,j-2) * Capture));
354             }
355             if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][EastID] >= 1 &&
               ↪  isSquareFree(i+1,j+2)){
356               moves.push_back(genMove(i,j,i+1,j+2, isSquareEnemy(i+1,j+2) * Capture));
357             }
358             if (numSquares[currentPID][SouthID] >= 2 && numSquares[currentPID][WestID] >= 1 &&
               ↪  isSquareFree(i-1,j+2)){
359               moves.push_back(genMove(i,j,i-1,j+2, isSquareEnemy(i-1,j+2) * Capture));
360             }
361
362             if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][NorthID] >= 1 &&
               ↪  isSquareFree(i+2,j-1)){
363               moves.push_back(genMove(i,j,i+2,j-1, isSquareEnemy(i+2,j-1) * Capture));
364             }
365             if (numSquares[currentPID][EastID] >= 2 && numSquares[currentPID][SouthID] >= 1 &&
               ↪  isSquareFree(i+2,j+1)){
366               moves.push_back(genMove(i,j,i+2,j+1, isSquareEnemy(i+2,j+1) * Capture));
367             }
368             if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][NorthID] >= 1 &&
               ↪  isSquareFree(i-2,j-1)){
369               moves.push_back(genMove(i,j,i-2,j-1, isSquareEnemy(i-2,j-1) * Capture));
370             }
371             if (numSquares[currentPID][WestID] >= 2 && numSquares[currentPID][SouthID] >= 1 &&
               ↪  isSquareFree(i-2,j+1)){
372               moves.push_back(genMove(i,j,i-2,j+1, isSquareEnemy(i-2,j+1) * Capture));
373             }
374
375           }
376
377           if (pieceType(piece) == Rook || pieceType(piece) == Bishop || pieceType(piece) == Queen){
378
379             int startDir = (pieceType(piece) == Bishop) ? 4 : 0;
380             int endDir = (pieceType(piece) == Rook) ? 3 : 7;
381
382             for (int dirID = startDir ; dirID <= endDir ; ++dirID){
383               for (int i = 0 ; i < numSquares[currentPID][dirID]; ++i){
384                 int targetPos = currentPID + directions[dirID] * (i+1);
385
386                 if (isSquareFriendly(targetPos)){
387                   break;
388                 }
389                 moves.push_back(genMove(currentPID, targetPos, isSquareEnemy(targetPos) * Capture));
390
391                 if (isSquareEnemy(targetPos)){
392                   break;
393                 }
394               }
395             }
396           }
397         }
398       }
399     }
400
401     return moves;
402   }
403
404   int BoardManager::isLegal(std::vector<int> moves, int move){
405     for (int movei : moves){
406       if (discardTag(movei) == move){
407         return movei;
408       }
```

```
409     }
410     return 0;
411   }
412
413   void BoardManager::loadFen(std::string fen){
414     gameStateHistory = std::stack<GameState>();
415
416     currentGameState.canBlackQueenCastle = false;
417     currentGameState.canWhiteQueenCastle = false;
418     currentGameState.canWhiteKingCastle = false;
419     currentGameState.canBlackKingCastle = false;
420
421     for (int ii = 0 ; ii <= 7; ++ii){
422       for (int jj = 0; jj <= 7; ++jj){
423         board[jj][ii] = None;
424       }
425     }
426
427     int state = 0;
428     int i = 0;
429     int j = 0;
430
431     bool hasReadNb = false;
432     int nb = 0;
433
434     for(char c : fen){
435       if (c == ' '){
436         state +=1;
437         continue;
438       }
439       if (state == 0){
440         switch (c){
441           case '/':
442              j +=1;
443              i = 0;
444              break;
445           case '1' : case '2' : case '3' : case '4' : case '5' : case '6' : case '7' : case '8' :
446             i += c - '0';
447             break;
448           case 'r':
449             board[j][i] = Black | Rook;
450             i += 1;
451             break;
452           case 'n':
453             board[j][i] = Black | Knight;
454             i += 1;
455             break;
456           case 'b':
457             board[j][i] = Black | Bishop;
458             i += 1;
459             break;
460           case 'q':
461             board[j][i] = Black | Queen;
462             i += 1;
463             break;
464           case 'k':
465             board[j][i] = Black | King;
466             currentGameState.blackKingPos = i + j*8;
467             i += 1;
468             break;
469           case 'p':
470             board[j][i] = Black | Pawn;
471             i += 1;
472             break;
473           case 'R':
474             board[j][i] = White | Rook;
475             i += 1;
476             break;
477           case 'N':
478             board[j][i] = White | Knight;
479             i += 1;
480             break;
481           case 'B':
482             board[j][i] = White | Bishop;
483             i += 1;
484             break;
485           case 'Q':
486             board[j][i] = White | Queen;
487             i += 1;
488             break;
489           case 'K':
490             board[j][i] = White | King;
```

```
491             currentGameState.whiteKingPos = i + j*8;
492             i += 1;
493             break;
494           case 'P':
495             board[j][i] = White | Pawn;
496             i += 1;
497             break;
498 
499       }
500     }
501     if (state == 1){
502       whiteToMove = (c == 'w');
503     }
504     if (state == 2){
505       switch (c){
506         case 'Q':
507           currentGameState.canWhiteQueenCastle = true;
508           break;
509         case 'K':
510           currentGameState.canWhiteKingCastle = true;
511           break;
512         case 'q':
513           currentGameState.canBlackQueenCastle = true;
514           break;
515         case 'k':
516           currentGameState.canBlackKingCastle = true;
517           break;
518       }
519     }
520     if (state == 3){
521       switch (c){
522         case '1' : case '2' : case '3' : case '4' : case '5' : case '6' : case '7' : case '8' :
523           currentGameState.doublePushFile = (c - '0');
524           break;
525       }
526     }
527     if (state == 4){
528       if (!hasReadNb){
529         nb = c - '0';
530         hasReadNb = true;
531       }
532       else{
533         nb *= 10;
534         nb += c - '0';
535       }
536     }
537   }
538   currentGameState.moveCount = nb;
539 }
540 
541 void BoardManager::makeMove(int move){
542   int startPosi = startPos(move);
543   int endPosi = endPos(move);
544   int tag = (move & tagMask) >> 12;
545 
546   GameState newGameState;
547   newGameState.canBlackQueenCastle = currentGameState.canBlackQueenCastle;
548   newGameState.canWhiteQueenCastle = currentGameState.canWhiteQueenCastle;
549   newGameState.canWhiteKingCastle = currentGameState.canWhiteKingCastle;
550   newGameState.canBlackKingCastle = currentGameState.canBlackKingCastle;
551   newGameState.doublePushFile = 0;
552   newGameState.whiteKingPos = currentGameState.whiteKingPos;
553   newGameState.blackKingPos = currentGameState.blackKingPos;
554   newGameState.hasWhiteCastled = currentGameState.hasWhiteCastled;
555   newGameState.hasBlackCastled = currentGameState.hasBlackCastled;
556 
557   sf::Vector2i startPos = posIntTo2D(startPosi);
558   sf::Vector2i endPos = posIntTo2D(endPosi);
559 
560   if (startPosi == endPosi){
561     return;
562   }
563 
564   zobristKey ^= piecesZobrist[pieceType(get(startPosi))][whiteToMove][startPosi];
565   if (tag == Capture || tag == KnightPromCapture || tag == RookPromCapture || tag == BishopPromCapture || tag ==
      ↪  QueenPromCapture){
566     zobristKey ^= piecesZobrist[pieceType(get(endPosi))][!whiteToMove][endPosi];
567   }
568   zobristKey ^= piecesZobrist[pieceType(get(startPosi))][whiteToMove][endPosi];
569 
570   if (tag == Capture || tag == KnightPromCapture || tag == RookPromCapture || tag == BishopPromCapture || tag ==
      ↪  QueenPromCapture){
```

```
571       newGameState.capturedPiece = get(endPosi);
572     }
573     if (tag == QuietMove || tag == Capture){
574       if ( (startPos.x == 0 && pieceType(get(startPosi)) == Rook) || (endPos.x == 0 && pieceType(get(endPosi)) ==
          ↪  Rook)  ){
575         if (whiteToMove){
576           newGameState.canWhiteQueenCastle = false;
577         }
578         else{
579           newGameState.canBlackQueenCastle = false;
580         }
581       }
582       if (  (startPos.x == 7 && pieceType(get(startPosi)) == Rook) || (endPos.x == 7 && pieceType(get(endPosi)) ==
          ↪  Rook)  ){
583         if (whiteToMove){
584           newGameState.canWhiteKingCastle = false;
585         }
586         else{
587           newGameState.canBlackKingCastle = false;
588         }
589       }
590       if (pieceType(get(startPosi)) == King){
591         if (whiteToMove){
592           newGameState.canWhiteQueenCastle = false;
593           newGameState.canWhiteKingCastle = false;
594           newGameState.whiteKingPos = endPosi;
595         }
596         else{
597           newGameState.canBlackQueenCastle = false;
598           newGameState.canBlackKingCastle = false;
599           newGameState.blackKingPos = endPosi;
600         }
601       }
602       board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
603       board[startPos.y][startPos.x] = None;
604     }
605
606     if (tag == KnightPromCapture || tag == RookPromCapture || tag == BishopPromCapture || tag == QueenPromCapture){
607       int piecePromoType = Queen;
608       switch (tag){
609         case KnightPromCapture:
610           piecePromoType = Knight;
611           break;
612         case BishopPromCapture:
613           piecePromoType = Bishop;
614           break;
615         case RookPromCapture:
616           piecePromoType = Rook;
617           break;
618         default:
619           piecePromoType = Queen;
620           break;
621       }
622
623
624       if (isPieceWhite(get(startPosi))){
625         board[endPos.y][endPos.x] = White | piecePromoType;
626         zobristKey ^= piecesZobrist[Pawn][1][endPosi];
627         zobristKey ^= piecesZobrist[piecePromoType][1][endPosi];
628       }
629       else{
630         board[endPos.y][endPos.x] = Black | piecePromoType;
631         zobristKey ^= piecesZobrist[Pawn][0][endPosi];
632         zobristKey ^= piecesZobrist[piecePromoType][0][endPosi];
633       }
634       board[startPos.y][startPos.x] = None;
635     }
636
637     if (tag == DoublePawnPush){
638       board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
639       board[startPos.y][startPos.x] = None;
640       newGameState.doublePushFile = startPos.x + 1;
641     }
642     if (tag == EPCapture){
643       board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
644       board[startPos.y][startPos.x] = None;
645       board[startPos.y][endPos.x] = None;
646       zobristKey ^= piecesZobrist[Pawn][!whiteToMove][endPos.x + 8 * startPos.y];
647     }
648     if (tag == KingCastle || tag == QueenCastle){
649       board[endPos.y][endPos.x] = board[startPos.y][startPos.x];
650       board[startPos.y][startPos.x] = None;
```

```
651       if (whiteToMove){
652         newGameState.canWhiteKingCastle = false;
653         newGameState.canWhiteQueenCastle = false;
654         newGameState.whiteKingPos = endPosi;
655         newGameState.hasWhiteCastled = true;
656       }
657       else{
658         newGameState.canBlackKingCastle = false;
659         newGameState.canBlackQueenCastle = false;
660         newGameState.blackKingPos = endPosi;
661         newGameState.hasBlackCastled = true;
662       }
663       if (tag == KingCastle){
664         board[startPos.y][5] = board[startPos.y][7];
665         board[startPos.y][7] = None;
666         zobristKey ^= piecesZobrist[Rook][whiteToMove][7 + startPos.y * 8];
667         zobristKey ^= piecesZobrist[Rook][whiteToMove][5 + startPos.y * 8];
668       }
669       else{
670         board[startPos.y][3] = board[startPos.y][0];
671         board[startPos.y][0] = None;
672         zobristKey ^= piecesZobrist[Rook][whiteToMove][startPos.y * 8];
673         zobristKey ^= piecesZobrist[Rook][whiteToMove][3 + startPos.y * 8];
674       }
675     }
676
677     if (tag == KnightProm || tag == RookProm || tag == BishopProm || tag == QueenProm){
678       int piecePromoType = Queen;
679       switch (tag){
680         case KnightProm:
681           piecePromoType = Knight;
682           break;
683         case BishopProm:
684           piecePromoType = Bishop;
685           break;
686         case RookProm:
687           piecePromoType = Rook;
688           break;
689         default:
690           piecePromoType = Queen;
691           break;
692       }
693
694       if (isPieceWhite(get(startPosi))){
695         board[endPos.y][endPos.x] = White | piecePromoType;
696         zobristKey ^= piecesZobrist[Pawn][1][endPosi];
697         zobristKey ^= piecesZobrist[piecePromoType][1][endPosi];
698       }
699       else{
700         board[endPos.y][endPos.x] = Black | piecePromoType;
701         zobristKey ^= piecesZobrist[Pawn][0][endPosi];
702         zobristKey ^= piecesZobrist[piecePromoType][0][endPosi];
703       }
704       board[startPos.y][startPos.x] = None;
705     }
706
707     if (newGameState.canWhiteKingCastle != currentGameState.canWhiteKingCastle){
708       zobristKey ^= castlingRightZobrist[0];
709     }
710     if (newGameState.canWhiteQueenCastle != currentGameState.canWhiteQueenCastle){
711       zobristKey ^= castlingRightZobrist[1];
712     }
713     if (newGameState.canBlackKingCastle != currentGameState.canBlackKingCastle){
714       zobristKey ^= castlingRightZobrist[2];
715     }
716     if (newGameState.canBlackQueenCastle != currentGameState.canBlackQueenCastle){
717       zobristKey ^= castlingRightZobrist[3];
718     }
719     zobristKey ^= doublePushFileZobrist[currentGameState.doublePushFile];
720     zobristKey ^= doublePushFileZobrist[newGameState.doublePushFile];
721     zobristKey ^= whiteToMoveZobrist;
722
723     newGameState.zobristKey = zobristKey;
724     whiteToMove = !whiteToMove;
725     gameStateHistory.push(currentGameState);
726     currentGameState = newGameState;
727
728   }
729
730   void BoardManager::unmakeMove(int move){
731     int startPosi = startPos(move);
732     int endPosi = endPos(move);
```

```
  int tag = (move & tagMask) >> 12;
  sf::Vector2i startPos = posIntTo2D(startPosi);
  sf::Vector2i endPos = posIntTo2D(endPosi);

  if (startPosi == endPosi){
    return;
  }
  if (tag == QuietMove || tag == DoublePawnPush){
    board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
    board[endPos.y][endPos.x] = None;
  }
  if (tag == Capture){
    board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
    board[endPos.y][endPos.x] = currentGameState.capturedPiece;
  }
  if (tag == EPCapture){
    board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
    board[endPos.y][endPos.x] = None;
    if (!whiteToMove){
      board[endPos.y + 1][endPos.x] = Black | Pawn;
    }
    else{
      board[endPos.y - 1][endPos.x] = White | Pawn;
    }
  }
  if (tag == KingCastle || tag == QueenCastle){
    board[startPos.y][startPos.x] = board[endPos.y][endPos.x];
    board[endPos.y][endPos.x] = None;
    if (tag == KingCastle){
      board[startPos.y][7] = board[startPos.y][5];
      board[startPos.y][5] = None;
    }
    else{
      board[startPos.y][0] = board[startPos.y][3];
      board[startPos.y][3] = None;
    }
  }

  if (tag == KnightProm || tag == RookProm || tag == BishopProm || tag == QueenProm){
    if (!whiteToMove){
      board[startPos.y][startPos.x] = White | Pawn;
    }
    else{
      board[startPos.y][startPos.x] = Black | Pawn;
    }
    board[endPos.y][endPos.x] = None;
  }

  if (tag == KnightPromCapture || tag == RookPromCapture || tag == BishopPromCapture || tag == QueenPromCapture){
    if (!whiteToMove){
      board[startPos.y][startPos.x] = White | Pawn;
    }
    else{
      board[startPos.y][startPos.x] = Black | Pawn;
    }
    board[endPos.y][endPos.x] = currentGameState.capturedPiece;
  }

  whiteToMove = !whiteToMove;
  currentGameState = gameStateHistory.top();
  zobristKey = currentGameState.zobristKey;
  gameStateHistory.pop();
}

int BoardManager::get(int pos){
  if (pos >= 0 && pos < 64){
    return board[pos / 8][pos % 8];
  }
  else{
    return 0;
  }
}
int BoardManager::get(int x, int y){
  if (x >= 0 && x < 8 && y  >= 0 && y < 8){
    return board[y][x];
  }
  else{
    return 0;
  }
}

uint64_t BoardManager::computeZobrist(){
```

```
  uint64_t newZobristKey = (uint64_t)0;

  for (int i = 0; i < 64; ++i){
    int piece = get(i);
    if (piece > 0){
      newZobristKey ^= piecesZobrist[pieceType(piece)][isPieceWhite(piece)][i];
    }
  }
  newZobristKey ^= doublePushFileZobrist[currentGameState.doublePushFile];
  if (!whiteToMove){
    newZobristKey ^= whiteToMoveZobrist;
  }

  if (currentGameState.canWhiteKingCastle){
    newZobristKey ^= castlingRightZobrist[0];
  }
  if (currentGameState.canWhiteQueenCastle){
    newZobristKey ^= castlingRightZobrist[1];
  }
  if (currentGameState.canBlackKingCastle){
    newZobristKey ^= castlingRightZobrist[2];
  }
  if (currentGameState.canBlackQueenCastle){
    newZobristKey ^= castlingRightZobrist[3];
  }

  return newZobristKey;
}

void BoardManager::initZobrist(){
    std::mt19937_64 gen(time(NULL));
    std::uniform_int_distribution<uint64_t> dis(0, std::numeric_limits<uint64_t>::max());

    for (int pType = 1; pType <= 6; ++pType){
      for (int color = 0 ; color <= 1; ++color){
        for (int squareID = 0; squareID < 64; ++squareID){
          piecesZobrist[pType][color][squareID] = dis(gen);
        }
      }
    }

    for (int i = 0 ; i < 9; ++i){
      doublePushFileZobrist[i] = dis(gen);
    }
    whiteToMoveZobrist = dis(gen);
    for (int i = 0; i < 4; ++i){
      castlingRightZobrist[i] = dis(gen);
    }

    zobristKey = computeZobrist();
}
```

11.5.4 Trouver le meilleur coup à partir d'une position

bot.h

```
#include "boardManager.h"
#include "transposition.h"

#define maxBotDepth 50

#define pawnValue 100
#define knightValue 280
#define bishopValue 320
#define rookValue 479
#define queenValue 929
#define kingValue 20000

enum BotType
{
  NotBot = 0,
  Random = 1,
  Good = 2

};
struct MoveScore
{
    int move;
    int score;
```

```
};

class Bot
{
public:
  Bot();
  int getBestMove(BoardManager* board);

  int quietSearch(BoardManager* board, int alpha, int beta);
  int search(BoardManager* board, char depth, int alpha, int beta);
  int evaluate(BoardManager* board);
  int scoreMove(BoardManager* board, int move);
  std::vector<int> orderMoves(BoardManager* board, std::vector<int> moves, char depth);
  int accessHeatMapMG(int pType,int i, bool whitePlaying);
  int accessHeatMapEG(int pType,int i, bool whitePlaying);

  int nbQMoves;
  int nbMoves = 0;
  char currentDepth;

  int maxTime;
  std::chrono::high_resolution_clock::time_point startTime;
  bool reachedTime;

  TranspositionTable transpositionTable;
  int nbTranspo = 0;
  int PVmoves[maxBotDepth];


  int mg_pawn_table[64] = {
        0,   0,   0,   0,   0,   0,  0,   0,
       98, 134,  61,  95,  68, 126, 34, -11,
       -6,   7,  26,  31,  65,  56, 25, -20,
      -14,  13,   6,  21,  23,  12, 17, -23,
      -27,  -2,  -5,  12,  17,   6, 10, -25,
      -26,  -4,  -4, -10,   3,   3, 33, -12,
      -35,  -1, -20, -23, -15,  24, 38, -22,
        0,   0,   0,   0,   0,   0,  0,   0,
  };
  int eg_pawn_table[64] = {
        0,   0,   0,   0,   0,   0,   0,   0,
      178, 173, 158, 134, 147, 132, 165, 187,
       94, 100,  85,  67,  56,  53,  82,  84,
       32,  24,  13,   5,  -2,   4,  17,  17,
       13,   9,  -3,  -7,  -7,  -8,   3,  -1,
        4,   7,  -6,   1,   0,  -5,  -1,  -8,
       13,   8,   8,  10,  13,   0,   2,  -7,
        0,   0,   0,   0,   0,   0,   0,   0,
  };
  int mg_knight_table[64] = {
      -167, -89, -34, -49,  61, -97, -15, -107,
       -73, -41,  72,  36,  23,  62,   7,  -17,
       -47,  60,  37,  65,  84, 129,  73,   44,
        -9,  17,  19,  53,  37,  69,  18,   22,
       -13,   4,  16,  13,  28,  19,  21,   -8,
       -23,  -9,  12,  10,  19,  17,  25,  -16,
       -29, -53, -12,  -3,  -1,  18, -14,  -19,
      -105, -21, -58, -33, -17, -28, -19,  -23,
  };
  int eg_knight_table[64] = {
      -58, -38, -13, -28, -31, -27, -63, -99,
      -25,  -8, -25,  -2,  -9, -25, -24, -52,
      -24, -20,  10,   9,  -1,  -9, -19, -41,
      -17,   3,  22,  22,  22,  11,   8, -18,
      -18,  -6,  16,  25,  16,  17,   4, -18,
      -23,  -3,  -1,  15,  10,  -3, -20, -22,
      -42, -20, -10,  -5,  -2, -20, -23, -44,
      -29, -51, -23, -15, -22, -18, -50, -64,
  };
  int mg_bishop_table[64] = {
      -29,   4, -82, -37, -25, -42,   7,  -8,
      -26,  16, -18, -13,  30,  59,  18, -47,
      -16,  37,  43,  40,  35,  50,  37,  -2,
       -4,   5,  19,  50,  37,  37,   7,  -2,
       -6,  13,  13,  26,  34,  12,  10,   4,
        0,  15,  15,  15,  14,  27,  18,  10,
        4,  15,  16,   0,   7,  21,  33,   1,
      -33,  -3, -14, -21, -13, -12, -39, -21,
  };
  int eg_bishop_table[64] = {
      -14, -21, -11,  -8, -7,  -9, -17, -24,
       -8,  -4,   7, -12, -3, -13,  -4, -14,
```

```
        2,  -8,   0,  -1, -2,   6,   0,   4,
       -3,   9,  12,   9, 14,  10,   3,   2,
       -6,   3,  13,  19,  7,  10,  -3,  -9,
      -12,  -3,   8,  10, 13,   3,  -7, -15,
      -14, -18,  -7,  -1,  4,  -9, -15, -27,
      -23,  -9, -23,  -5, -9, -16,  -5, -17,
  };
  int mg_rook_table[64] = {
       32,  42,  32,  51, 63,  9,  31,  43,
       27,  32,  58,  62, 80, 67,  26,  44,
       -5,  19,  26,  36, 17, 45,  61,  16,
      -24, -11,   7,  26, 24, 35,  -8, -20,
      -36, -26, -12,  -1,  9, -7,   6, -23,
      -45, -25, -16, -17,  3,  0,  -5, -33,
      -44, -16, -20,  -9, -1, 11,  -6, -71,
      -19, -13,   1,  17, 16,  7, -37, -26,
  };
  int eg_rook_table[64] = {
      13, 10, 18, 15, 12,  12,   8,   5,
      11, 13, 13, 11, -3,   3,   8,   3,
       7,  7,  7,  5,  4,  -3,  -5,  -3,
       4,  3, 13,  1,  2,   1,  -1,   2,
       3,  5,  8,  4, -5,  -6,  -8, -11,
      -4,  0, -5, -1, -7, -12,  -8, -16,
      -6, -6,  0,  2, -9,  -9, -11,  -3,
      -9,  2,  3, -1, -5, -13,   4, -20,
  };
  int mg_queen_table[64] = {
      -28,   0,  29,  12,  59,  44,  43,  45,
      -24, -39,  -5,   1, -16,  57,  28,  54,
      -13, -17,   7,   8,  29,  56,  47,  57,
      -27, -27, -16, -16,  -1,  17,  -2,   1,
       -9, -26,  -9, -10,  -2,  -4,   3,  -3,
      -14,   2, -11,  -2,  -5,   2,  14,   5,
      -35,  -8,  11,   2,   8,  15,  -3,   1,
       -1, -18,  -9,  10, -15, -25, -31, -50,
  };
  int eg_queen_table[64] = {
       -9,  22,  22,  27,  27,  19,  10,  20,
      -17,  20,  32,  41,  58,  25,  30,   0,
      -20,   6,   9,  49,  47,  35,  19,   9,
        3,  22,  24,  45,  57,  40,  57,  36,
      -18,  28,  19,  47,  31,  34,  39,  23,
      -16, -27,  15,   6,   9,  17,  10,   5,
      -22, -23, -30, -16, -16, -23, -36, -32,
      -33, -28, -22, -43,  -5, -32, -20, -41,
  };
  int mg_king_table[64] = {
      -65,  23,  16, -15, -56, -34,   2,  13,
       29,  -1, -20,  -7,  -8,  -4, -38, -29,
       -9,  24,   2, -16, -20,   6,  22, -22,
      -17, -20, -12, -27, -30, -25, -14, -36,
      -49,  -1, -27, -39, -46, -44, -33, -51,
      -14, -14, -22, -46, -44, -30, -15, -27,
        1,   7,  -8, -64, -43, -16,   9,   8,
      -15,  36,  12, -54,   8, -28,  24,  14,
  };
  int eg_king_table[64] = {
      -74, -35, -18, -18, -11,  15,   4, -17,
      -12,  17,  14,  17,  17,  38,  23,  11,
       10,  17,  23,  15,  20,  45,  44,  13,
       -8,  22,  24,  27,  26,  33,  26,   3,
      -18,  -4,  21,  24,  27,  23,   9, -11,
      -19,  -3,  11,  21,  23,  16,   7,  -9,
      -27, -11,   4,  13,  14,   4,  -5, -17,
      -53, -34, -21, -11, -28, -14, -24, -43
  };

};
```

bot.cpp

```
#include "bot.h"
{

int pieceValue(int ptype){
  switch (ptype){
    case Pawn:
      return pawnValue;
      break;
```

```
    case Knight:
      return knightValue;
      break;
    case Rook:
      return rookValue;
      break;
    case Bishop:
      return bishopValue;
      break;
    case Queen:
      return queenValue;
      break;
    case King:
      return kingValue;
      break;
  }
  return 0;
}

int Bot::scoreMove(BoardManager* board, int move){
  int score = 0;

  int movingPieceType = pieceType(board->get(startPos(move)));
  int capturedPieceType = pieceType(board->get(endPos(move)));
  int moveTag = tag(move);

  if (capturedPieceType > 0){
    score += 10 * pieceValue(capturedPieceType) - pieceValue(movingPieceType);
  }

  if (moveTag >= 8) // Promotion{
    int promPieceType = Queen;
    switch (moveTag){
      case KnightProm : case KnightPromCapture:
        promPieceType = Knight;
        break;
      case BishopProm : case BishopPromCapture:
        promPieceType = Bishop;
        break;
      case RookProm: case RookPromCapture:
        promPieceType = Rook;
        break;
    }
    score += pieceValue(promPieceType);
  }
  board->whiteToMove = !board->whiteToMove;
  board->controlledSquares();
  if (board->controlled[endPos(move) / 8][endPos(move) % 8]){
    score -= pieceValue(movingPieceType);
  }
  board->whiteToMove = !board->whiteToMove;
  return score;
}

bool compMove(MoveScore mscore1, MoveScore mscore2){
  return mscore1.score > mscore2.score;
}

std::vector<int> Bot::orderMoves(BoardManager* board, std::vector<int> moves, char depth){
  std::vector<MoveScore> movesScore;
  for (int move : moves){
    MoveScore mscore;
    mscore.move = move;

    if (move == PVmoves[currentDepth - depth]){
      mscore.score = infinity;
    }
    else{
      mscore.score = scoreMove(board, move);
    }

    movesScore.push_back(mscore);
  }

  std::sort(movesScore.begin(), movesScore.end(), compMove);
  std::vector<int> sortedMoves;

  for(MoveScore mscore : movesScore){
    sortedMoves.push_back(mscore.move);
  }
  return sortedMoves;
}
```

```
91
92
93
94  int rotate(int i, bool whitePlaying){
95    return whitePlaying ? i : (7 - (i%8)) + (i/8) * 7;
96  }
97
98  int Bot::accessHeatMapMG(int pType,int i, bool whitePlaying){
99    switch(pType){
100     case Pawn:
101       return mg_pawn_table[rotate(i, whitePlaying)];
102       break;
103     case Knight:
104       return mg_knight_table[rotate(i, whitePlaying)];
105       break;
106     case Bishop:
107       return mg_bishop_table[rotate(i, whitePlaying)];
108       break;
109     case Rook:
110       return mg_rook_table[rotate(i, whitePlaying)];
111       break;
112     case Queen:
113       return mg_queen_table[rotate(i, whitePlaying)];
114       break;
115     case King:
116       return mg_king_table[rotate(i, whitePlaying)];
117       break;
118   }
119   return 0;
120 }
121
122 int Bot::accessHeatMapEG(int pType,int i, bool whitePlaying){
123   switch(pType){
124     case Pawn:
125       return eg_pawn_table[rotate(i, whitePlaying)];
126       break;
127     case Knight:
128       return eg_knight_table[rotate(i, whitePlaying)];
129       break;
130     case Bishop:
131       return eg_bishop_table[rotate(i, whitePlaying)];
132       break;
133     case Rook:
134       return eg_rook_table[rotate(i, whitePlaying)];
135       break;
136     case Queen:
137       return eg_queen_table[rotate(i, whitePlaying)];
138       break;
139     case King:
140       return eg_king_table[rotate(i, whitePlaying)];
141       break;
142   }
143   return 0;
144 }
145
146
147 int restrainKingEndGame(BoardManager* board, int myKingPos, int opponentKingPos){
148   int newScore = 0;
149   int distCenterOpp = abs((opponentKingPos % 8) - 3) + abs((opponentKingPos / 8) - 3);
150   int distCenterFriend = abs((myKingPos % 8) - 3) + abs((myKingPos / 8) - 3);
151   int distKings = abs((myKingPos % 8) - (opponentKingPos % 8)) + abs((myKingPos / 8) - (opponentKingPos / 8));
152
153   newScore += distCenterOpp - distCenterFriend;
154   newScore += 2 * (14 - distKings);
155   return newScore;
156
157 }
158
159 int Bot::evaluate(BoardManager* board){
160   int score = 0;
161   int whiteScoreValue = 0;
162   int blackScoreValue = 0;
163   int heatMapScoreMGWhite = 0;
164   int heatMapScoreMGBlack = 0;
165   int heatMapScoreEGWhite = 0;
166   int heatMapScoreEGBlack = 0;
167   int pieceNumber = 0;
168
169   for (int i = 0 ; i < 64 ; ++i){
170     int piece = board->get(i);
171     int pType = pieceType(piece);
172     if (pType != None){
```

```
      pieceNumber += 1;
    }
    if (isPieceWhite(piece)){
      whiteScoreValue += pieceValue(pType);
      heatMapScoreMGWhite += accessHeatMapMG(pType,i, board->whiteToMove);
      heatMapScoreEGWhite += accessHeatMapEG(pType,i, board->whiteToMove);
    }
    else{
      blackScoreValue += pieceValue(pType);
      heatMapScoreMGBlack += accessHeatMapMG(pType,i, board->whiteToMove);
      heatMapScoreEGBlack += accessHeatMapEG(pType,i, board->whiteToMove);
    }

  }

  float endGameWeight = 1.0 - (float(pieceNumber) / 32.0);

  score += int((1.0 - endGameWeight) * ( (board->currentGameState.hasWhiteCastled) - (board->currentGameState.hasBlackCastled)) * (board->whiteToMove ? 1 : -1) * 50);
  score += int( 0.5 * ( (1.0 - endGameWeight) * (heatMapScoreMGWhite - heatMapScoreMGBlack) + endGameWeight * (heatMapScoreEGWhite - heatMapScoreEGBlack) )* (board->whiteToMove ? 1 : -1));
  score -= board->isChecked() * 50;

  if (board->whiteToMove){
    score += int(restrainKingEndGame(board, board->currentGameState.whiteKingPos, board->currentGameState.blackKingPos) * 10 * endGameWeight);
  }
  else{
    score += int(restrainKingEndGame(board, board->currentGameState.blackKingPos, board->currentGameState.whiteKingPos) * 10 * endGameWeight);
  }

  score += (whiteScoreValue - blackScoreValue) * (board->whiteToMove ? 1 : -1);
  return score;
}

int Bot::quietSearch(BoardManager* board, int alpha, int beta){
  int eval = evaluate(board);
  if (eval >= beta){
    return beta;
  }
  if (eval > alpha){
    alpha = eval;
  }

  std::vector<int> moves = board->generateMoves(true);
  std::vector<int> sortedMoves = orderMoves(board, moves, 0);

  for (int move : sortedMoves){
    board->makeMove(move);
    int eval = -quietSearch(board, -beta, -alpha);
    board->unmakeMove(move);

    nbQMoves += 1;

    if (eval >= beta){
      return beta;
    }
    if (eval > alpha){
      alpha = eval;
    }
  }
  return alpha;

}

int Bot::search(BoardManager* board, char depth, int alpha, int beta){
  char nodeType = AlphaNode;
  Transposition t = transpositionTable.get(board->zobristKey, depth, alpha, beta);
  if (t.isValid){
    nbTranspo += 1;
    return t.value;
  }

  std::chrono::high_resolution_clock::time_point endTime = std::chrono::high_resolution_clock::now();
  std::chrono::milliseconds duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);

  if (depth == 0){
    nbMoves += 1;
    return quietSearch(board, alpha, beta);
  }
```

```
250     if (duration.count() > maxTime){
251       reachedTime = true;
252       return 0;
253     }
254
255     std::vector<int> moves = board->generateMoves(false);
256     std::vector<int> sortedMoves = orderMoves(board, moves, depth);
257     if (sortedMoves.size() == 0){
258       if (board->isChecked()){
259         return -3*kingValue;
260       }
261       return 0;
262     }
263
264     int bestPositionMove = 0;
265
266     for (int move : sortedMoves){
267
268       board->makeMove(move);
269       int eval = -search(board, depth - 1, -beta, -alpha);
270       board->unmakeMove(move);
271
272
273       if (eval >= beta){
274         transpositionTable.set(board->zobristKey, depth, beta, BetaNode, move);
275         return beta;
276       }
277       if (eval > alpha){
278         nodeType = ExactNode;
279         bestPositionMove = move;
280         alpha = eval;
281       }
282
283     }
284     transpositionTable.set(board->zobristKey, depth, alpha, nodeType, bestPositionMove);
285     return alpha;
286
287   }
288
289
290
291   int Bot::getBestMove(BoardManager* board){
292     nbMoves = 0;
293     nbTranspo = 0;
294     reachedTime = false;
295     nbQMoves = 0;
296     startTime = std::chrono::high_resolution_clock::now();
297     int eval = 0;
298     int finalEval = 0;
299
300     for (char i = 1; i <= maxBotDepth; ++i){
301       currentDepth = i;
302       eval = search(board, i, -infinity, infinity);
303       if (!reachedTime){
304         finalEval = eval;
305         std::stack<int> moves;
306         for (int j = 0; j < 1; ++j){
307           Transposition t = transpositionTable.get(board->zobristKey, 0, 0, 0);
308           if (t.isValid){
309             PVmoves[j] = t.bestMove;
310             board->makeMove(t.bestMove);
311             moves.push(t.bestMove);
312           }
313           else{
314             break;
315           }
316         }
317         while (!moves.empty()){
318           board->unmakeMove(moves.top());
319           moves.pop();
320         }
321       }
322       else{
323         break;
324       }
325
326     }
327
328     printf("info Profondeur : %d\n", currentDepth - 1);
329     printf("info Nombre positions evaluees : %d\n", nbMoves);
330     printf("info Nombre transpositions rencontrees : %d\n", nbTranspo);
331     printf("info Nombre d'entrees dans la table de transposition : %d\n", transpositionTable.count);
```

```
  printf("info Nombre de positions silencieuses evaluees : %d\n", nbQMoves);
  printf("info Mouvement --> %s  : eval =  %d\n", standardNotation(PVmoves[0]).c_str(), finalEval);
  printf("info ----------\n");

  transpositionTable.clear();
  return PVmoves[0];
}
```

11.5.5 Gérer les transpositions

transposition.h

```
#define tableSize 35000

enum NodeType{
  ExactNode = 0,
  AlphaNode = 1,
  BetaNode = 2
};
struct Transposition{
  uint64_t key;
  char depth;
  int value;
  int bestMove;
  char nodeType;
  bool isValid = false;
};

class TranspositionTable{
public:
  TranspositionTable();
  Transposition get(uint64_t key, char depth, int alpha, int beta);
  void set(uint64_t key, char depth, int value, char nodeType, int bestMove);
  void clear();

  int count;
  Transposition table[tableSize];

};
```

transposition.cpp

```
#include "transposition.h"

TranspositionTable::TranspositionTable(){
  clear();
}

Transposition TranspositionTable::get(uint64_t key, char depth, int alpha, int beta){
  int index = key % tableSize;
  Transposition t = table[index];
  if (t.key == key){
    if (t.depth >= depth){
      if (t.nodeType == ExactNode){
        return t;
      }
      if (t.nodeType == AlphaNode && t.value <= alpha){
        return t;
      }
      if (t.nodeType == BetaNode && t.value >= beta){
        return t;
      }
    }
  }
  t.isValid = false;
  return t;
}

void TranspositionTable::set(uint64_t key, char depth, int value, char nodeType, int bestMove){
  Transposition t;
  t.isValid = true;
  t.key = key;
  t.depth = depth;
  t.value = value;
  t.nodeType = nodeType;
  t.bestMove = bestMove;
```

```
  table[key % tableSize] = t;
  count += 1;

}

void TranspositionTable::clear(){
  for (int i = 0 ; i < tableSize; ++i){
    table[i].isValid = false;
  }
  count = 0;
}
```

11.5.6 Le protocole UCI

uci.cpp

```
#include "engine.h"

#define botBaseTime 3000

int time_to_alloc(int baseTime, int inc){
    if (baseTime == 0){
        return botBaseTime;
    }
    if (baseTime < 5 * inc){
        return inc;
    }
    return (baseTime/30) + inc;
}



int main(){
    bool useUCI = false;

    Bot whiteBot();
    Bot blackBot();

    BoardMaager board();

    engine.whiteBot.maxTime = botBaseTime;
    engine.blackBot.maxTime = botBaseTime;


    while(window.isOpen()){
        if (useUCI){
            std::string command;
            std::getline(std::cin, command);
{
                file << command << std::endl;
                file.close();
            }

            std::vector<std::string> words;
            std::stringstream ss(command);
            std::string str;
            while (std::getline(ss, str, ' ')) {
                words.push_back(str);
            }

            bool readPos = false;

            // Pos mode :
            // 0 : None
            // 1 : Fen
            // 2 : startpos
            int posMode = 0;
            bool readMoves = false;

            bool readBlackTime = false;
            bool readWhiteTime = false;
            bool readBlackInc = false;
            bool readWhiteInc = false;
            bool readMoveTime = false;

            bool needToGo = false;

            int whiteBaseTime = 0;
            int whiteInc = 0;
```

```
64              int blackBaseTime = 0;
65              int blackInc = 0;
66
67              int moveTime = 0;
68
69              std::string word;
70
71              std::string fen;
72
73              for (std::string word : words){
74                  if (word == "quit"){
75                      return 0;
76                  }
77                  if (word == "uci"){
78                      std::cout << "uciok" << std::endl;
79                      break;
80                  }
81                  if (word == "isready"){
82                      std::cout << "readyok" << std::endl;
83                      break;
84                  }
85                  if (word == "movetime"){
86                      readMoveTime = true;
87                      continue;
88                  }
89                  if (readMoveTime){
90                      moveTime = atoi(word.c_str());
91                      readMoveTime = false;
92                      continue;
93                  }
94                  if (word == "go"){
95                      needToGo = true;
96                  }
97                  if (word == "wtime"){
98                      readWhiteTime = true;
99                      continue;
100                 }
101                 if (readWhiteTime){
102                     whiteBaseTime = atoi(word.c_str());
103                     readWhiteTime = false;
104                     continue;
105                 }
106                 if (word == "btime"){
107                     readBlackTime = true;
108                     continue;
109                 }
110                 if (readBlackTime){
111                     blackBaseTime = atoi(word.c_str());
112                     readBlackTime = false;
113                     continue;
114                 }
115                 if (word == "winc"){
116                     readWhiteInc = true;
117                     continue;
118                 }
119                 if (readWhiteInc){
120                     whiteInc = atoi(word.c_str());
121                     readWhiteInc = false;
122                     continue;
123                 }
124                 if (word == "binc"){
125                     readBlackInc = true;
126                     continue;
127                 }
128                 if (readBlackInc){
129                     blackInc = atoi(word.c_str());
130                     readBlackInc = false;
131                     continue;
132                 }
133                 if (word == "position"){
134                     readPos = true;
135                     continue;
136                 }
137                 if (word == "startpos" && readPos){
138                     posMode = 2;
139                     engine.board.loadFen(engine.board.startingFen);
140                     engine.currentMoves = engine.board.generateMoves(false);
141                     engine.movesHistory = std::stack<int>();
142                     continue;
143                 }
144                 if (word == "fen" && readPos){
145                     posMode = 1;
```

```
                    continue;
                }
                if (readPos && posMode == 1){
                    fen += word + " ";
                    continue;
                }
                if (posMode == 2 && word == "moves"){
                    engine.board.whiteToMove = true;
                    readMoves = true;
                    continue;
                }
                if(posMode == 2 && readMoves){
                    if (word.size() == 4){
                        engine.tryMove(standNotToMove(word), ' ');
                    }
                    else{
                        char c = word[4];
                        word.pop_back();
                        engine.tryMove(standNotToMove(word), c);
                    }
                    continue;
                }
            }

            if (readPos && posMode == 1){
                board.loadFen(fen);
            }

            if (needToGo){
                whiteBot.maxTime = time_to_alloc(whiteBaseTime, whiteInc);
                blackBot.maxTime = time_to_alloc(blackBaseTime, blackInc);

                if (moveTime > 0){
                    whiteBot.maxTime = moveTime;
                    blackBot.maxTime = moveTime;
                }

                int bestMove = engine.getBestMove();
                std::string bestMoveString;
                bestMoveString  = standardNotation(bestMove);
                std::cout << "bestmove ";
                std:: cout << bestMoveString << std:: endl;
            }

        }

    }
        return 0;
}
```