

Licence MIASHS deuxième année

Rapport de projet informatique

Jeu d'échecs et intelligence artificielle

Projet réalisé du octobre 2022 au 16 janvier 2023

Membres du groupe

**Badr Agrad
Naim Chefirat**

Table des matières

1	Présentation du projet	4
2	Jeu d'échecs et intelligence artificielle	4
2.1	Échecs	4
2.1.1	Règles du Jeu	4
2.1.2	La notation FEN	7
2.2	Intelligence artificielle	8
2.2.1	L'algorithme Minimax	8
2.2.2	L'élagage alpha-bêta	9
2.3	Les enjeux de l'utilisation du langage C	10
3	Le cahier des charges	10
3.1	Les besoins et problématiques	10
3.1.1	L'efficacité de l'algorithme	10
3.1.2	Fournir des données à notre IA	11
3.2	Nos solutions	11
3.3	Le résultat attendu	11
4	Outils	12
4.1	Github et Git	12
4.2	Docker	12
5	Le code	13
5.1	Structures et définitions	13
5.1.1	Échiquier et types de données	13
5.1.2	Liste chaînée	13
5.1.3	Informations sur la partie	14
5.1.4	Description d'un déplacement	15
5.1.5	Table de hachage	15
5.2	Fonctions principales	15
5.2.1	Jeu d'échecs	16
5.2.2	IA	18
5.3	Optimisation	24
5.3.1	Élagage Alpha Beta	24
5.3.2	Opérations de copie du plateau	25
5.3.3	Optimisations du compilateur	26
5.3.4	Optimisations mineures	26
5.4	Fonctions n'ayant pas pu être réalisées	27
5.4.1	Protocole UCI	27
5.4.2	Tri des mouvements	28
5.4.3	Approfondissement itératif	28
6	Répartition du travail	28

7	Difficultés rencontrées	29
7.1	Difficultés rencontrées par Naim Chefirat	29
7.2	Difficultés rencontrées par Badr Agrad	29
8	Bilan	30
8.1	Le Bilan de Naim Chefirat	30
8.2	Le Bilan de Badr Agrad	30
8.3	Conclusion et perspectives	31
A	Sources	31
A.1	Webographie	31
B	Manuel Utilisateur	32
B.1	Mise en place de l’environnement et compilation	32
B.2	Utilisation-type	32
B.3	Utiliser la table de hachage	33
B.4	Erreurs courantes	33

1 Présentation du projet

Depuis l'avènement des Big Datas dans les années 2000, les systèmes informatiques ne cessent de se développer à une vitesse quasi exponentielle. Ce fut le cas des intelligences artificielles, programme informatique capable de reproduire des comportements liés aux humains, tels que le raisonnement, la planification ou encore la créativité. Le développement des réseaux de neurones artificielles a permis de créer des machines capable d'apprendre par elles-même et de s'améliorer au fil du temps. C'est le cas du logiciel Alpha Go, développé par l'entreprise DeepMind en 2014, qui a vaincu le champion du monde de jeu de Go Ke Jie en 2017.

Aujourd'hui, il existe énormément d'algorithmes différents permettant de développer des intelligences artificielles selon le besoin.

C'est ainsi que nous avons décidé de développer un jeu d'échec ainsi qu'une intelligence artificielle capable d'y jouer à un niveau assez élevé.

2 Jeu d'échecs et intelligence artificielle

2.1 Échecs

2.1.1 Règles du Jeu

Le jeu d'échec se joue sur un échiquier à 64 cases, comportant une moitié de cases noire et l'autre de cases blanches. Chaque joueur possède 16 pièces disposées de manière ordonnée sur l'échiquier.

En début de partie, chaque joueur possède :

- 1 roi
- 1 dame
- 2 fous
- 2 cavaliers
- 2 tours
- 8 pions

Ainsi, voici à quoi ressemble l'échiquier en début de partie :

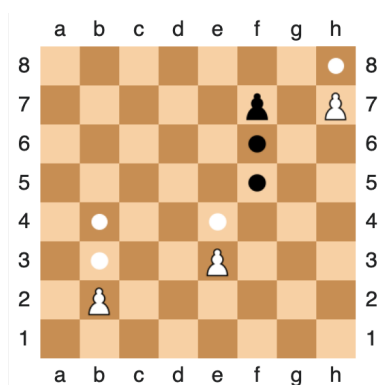


L'objectif de chaque joueur est de capturer le roi adverse. Ainsi on parlera d'échec et mat, de l'arabe "Al cheikh mat" (le roi est mort), pour désigner la situation dans laquelle le roi ne peut pas s'échapper de la capture de son adversaire. Dans le cas où aucun joueur n'arrive à capturer le roi de l'autre, la partie s'arrête au bout de 50 coups joués.

Voici les différents déplacements autorisés aux échecs, selon la pièce choisie :

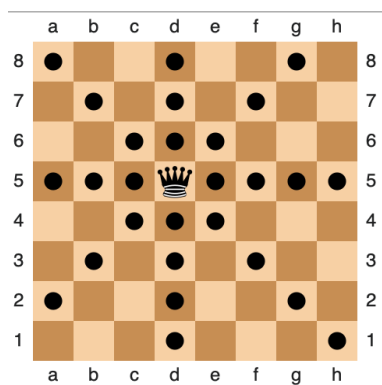
Les pions :

Ce sont les pièces les plus simples. Ils ne peuvent se déplacer que d'une case vers l'avant, sauf pour les captures. Dans le cas où un pion n'a jamais bougé, il peut se déplacer de deux case vers l'avant.



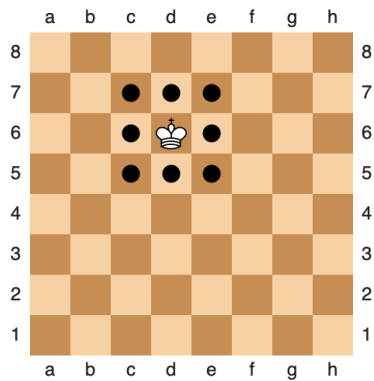
La Reine :

C'est la pièce la plus puissante. Elle peut se déplacer dans n'importe quelle direction, soit en diagonale, soit en ligne droite (vers l'avant comme vers l'arrière).



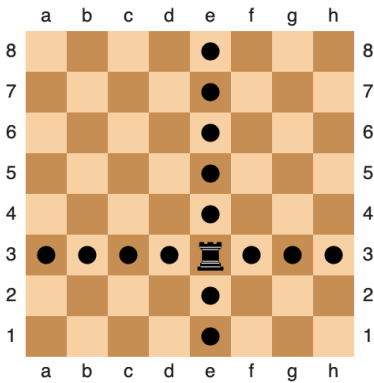
Le Roi :

C'est la pièce la plus importante. Il peut se déplacer d'une case dans n'importe quelle direction.



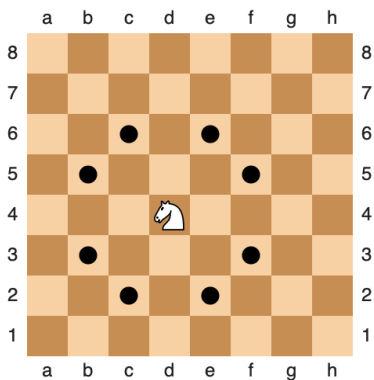
La Tour :

c'est une pièce qui se déplace en ligne droite. Elle peut se déplacer autant de cases qu'elle le souhaite dans cette direction.



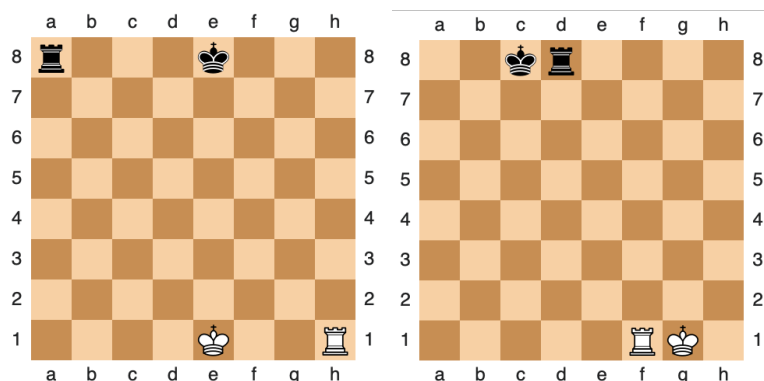
Le cavalier :

C'est une pièce qui peut passer par dessus les autres pièces. Il se déplace en forme de L vers l'avant ou l'arrière.



Le Roque :

Le roque est un mouvement qui permet de protéger le Roi et de renforcer la position de la Tour. Si le Roi et la Tour sont tous deux situés sur leurs emplacements d'origine et qu'il n'y a aucune pièce entre elles, il est possible d'effectuer un roque. Voici un schéma des deux roques possibles (les blancs réalisent un petit roque tandis que les noirs réalisent un grand).



2.1.2 La notation FEN

La notation Forsyth-Edwards, ou FEN, permet de rendre compte de la position des pièces sur l'échiquier et donc du statut de la partie. Créée par le journaliste David Forsyth, puis développée par Steven J Edwards qui l'a rendu compréhensible pour les ordinateurs, la notation FEN est devenue essentielle pour traiter des parties d'échecs. Cette notation est composée uniquement d'une chaîne de caractères ASCII composée de différents champs séparés par des espaces.

Le premier champ représente les huit lignes de l'échiquier. Il est composé de huit chaînes de caractères séparés par un slash "/". Les lettres R,B,K,P,N,Q correspondent respectivement aux tours, aux fous, aux rois, aux pions, aux cavaliers et aux reines. Les minuscules correspondent aux pièces noires tandis que les majuscules aux blanches. Les cases vides sont comptées une à une, jusqu'à la fin de la ligne ou l'apparition d'une pièce.

Le deuxième champ indique qui joue au prochain tour. La lettre "w" concerne les blancs tandis que la lettre "b" les noirs.

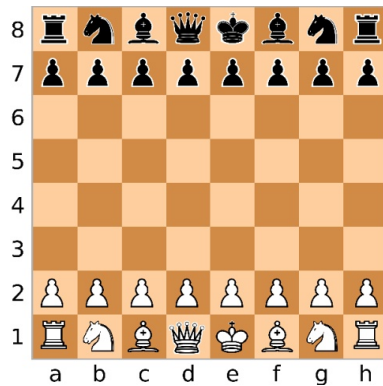
Le troisième champ concerne les informations sur les probables roques pouvant être effectués par les joueurs. D'une part, les lettres "k" et "q" désignent, respectivement, les roques côté Roi et Reine. D'autre part, l'absence de possibilité de roque est désigné par le caractère "-".

Le quatrième champ concerne la capture "en passant". C'est une règle spéciale permettant à un pion de capturer un autre pion qui vient de se déplacer de deux cases. Cela ne peut se produire que lorsque le pion qui effectue la capture était à côté de l'autre pion (avant que celui-ci ne se déplace de deux cases). Ainsi, cela permet d'éviter que les pions puissent se déplacer deux cases en avant sans risque d'être capturés.

Le cinquième champ concerne le compteur halfmove, qui est incrémenté dès lors qu'un joueur effectue un mouvement. Cependant, le compteur est remis à zéro dès lors qu'une pièce est capturée. Ainsi, au bout de 100 half-moves la partie s'arrête et on déclare un nul.

Le dernier champ concerne le compteur fullmove, qui est incrémenté dès lors qu'une pièce noire est déplacée.

Voici un exemple de notation FEN :



Ici, on notera :

"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"

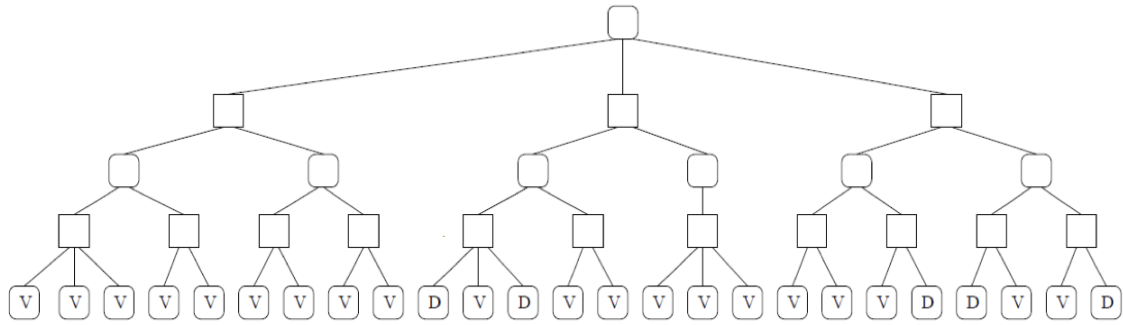
2.2 Intelligence artificielle

2.2.1 L'algorithme Minimax

L'algorithme Minimax est un algorithme qui permet à un ordinateur de jouer à un jeu qui oppose deux joueurs et où le gain de l'un correspond inévitablement à la perte de l'autre. On dit de ces jeux qu'ils sont de "somme nulle" car la somme des gains et des pertes des joueurs est nulle. L'objectif de cet algorithme est de minimiser la perte maximum. En effet, le programme va évaluer une valeur, un score, à chaque coup qui sera joué par lui ou le joueur. Ainsi, il passera en revue toutes les possibilités de jeu et choisira le meilleur coup possible (ayant la meilleure incidence pour lui). Il existe un aspect de profondeur, intrinsèque à cet algorithme, permettant de gérer le nombre d'appels récurrents réalisés Minimax. L'algorithme va effectivement fonder un arbre comportant les scores des différents échiquiers obtenus par coups. De ce fait, plus la profondeur est élevée, plus Minimax évaluera les conséquences de ses coups et de ceux de l'adversaire et plus l'arbre aura de branches.

Voici comment se présente l'arbre :

1. La racine correspond au jeu à l'état initial
2. Les noeuds correspondent aux différentes situations de jeu
3. Les fils correspondent aux possibilités de coup joué par l'autre joueur à partir de la nouvelle position de jeu
4. les branches correspondent à des séquences de jeu



Par conséquent, l'algorithme Minimax oppose deux joueurs : le joueur MAX qui cherche à maximiser ses gains à tous les coups, et le joueur MIN qui cherche à minimiser les gains du joueur MAX.

Minimax nous permet donc d'explorer les possibilités de jeu et de choisir la meilleure option possible.

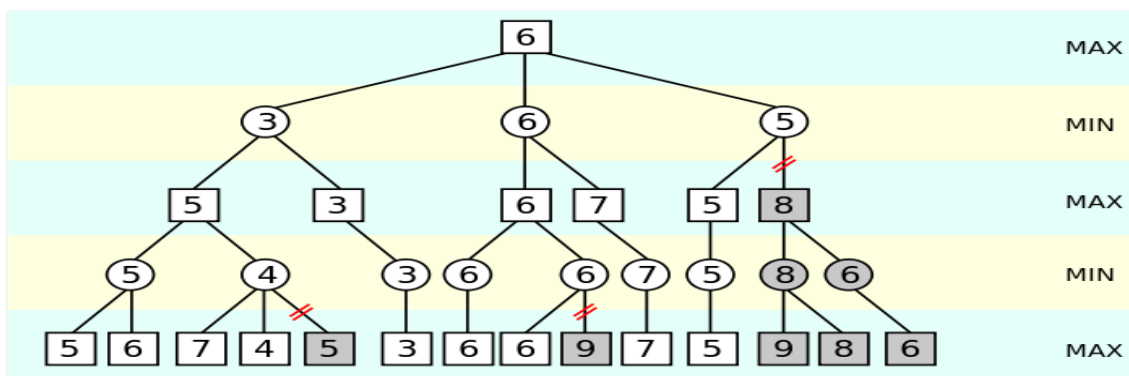
2.2.2 L'élagage alpha-bêta

L'élagage alpha-bêta est une méthode permettant d'optimiser la recherche dans l'arbre généré par mini-max. En effet, il permet de déterminer plus facilement quelle est la meilleure série de coups.

Deux valeurs, alpha et bêta, sont utilisées pour supprimer les branches d'arbre qui ne seront pas étudiées (car impertinentes). Ainsi, moins il y a de noeuds à étudier, plus l'algorithme sera performant. Alpha représente la meilleure option actuelle pour le premier joueur tandis que bêta représente la meilleure option pour le deuxième joueur. Au début de la recherche, alpha est initialisé à $-\infty$ et bêta à $+\infty$. Chaque fois que l'algorithme examine une branche de l'arbre, celui-ci va comparer les valeurs de la branche actuelle avec alpha et bêta. Si la valeur de la branche est supérieure à alpha, alpha sera mis à jour avec cette valeur. Tandis que si la valeur de la branche est inférieure à bêta, bêta sera mis à jour avec cette valeur.

Dès lors que la valeur d'alpha est supérieure ou égale à bêta, on réalise une "coupure" des branches correspondantes de l'arbre, car le résultat d'aucun des joueurs ne peut être amélioré.

Voici un exemple pour un jeu à somme nulle :



2.3 Les enjeux de l'utilisation du langage C

L'objet de notre projet est le développement d'une intelligence artificielle pour le jeu d'échec. Ainsi, notre objectif est de trouver les solutions les plus optimales afin d'atteindre ce but. D'une part, le travail réside dans le développement du jeu d'échec. Le langage C n'étant pas un langage de programmation orienté objet, il n'existe pas de classes nous permettant de traiter chaque pièce une à une, selon ses propriétés. Par conséquent, de par la création de structures ou encore de fonctions, il faut choisir la façon la plus efficace de coder. L'enjeu est de taille car notre jeu d'échec doit être à la fois efficace et interactif, mais doit aussi permettre la meilleure implémentation de l'algorithme Mini max. De ce fait, nous avons modifié à de nombreuses reprises notre façon de faire car notre travail de recherche nous permettait de trouver des solutions plus efficaces pour traiter les problématiques rencontrées.

L'enjeu principal de l'utilisation du langage C est l'optimisation du code. En effet, ce langage de programmation est le plus performant en termes d'exécution et d'exploitation des ressources (CPU et RAM). De plus, le C nous permet de manipuler la mémoire ce qui offre davantage de solutions concernant l'optimisation.

3 Le cahier des charges

3.1 Les besoins et problématiques

Dans ce chapitre, nous allons détailler nos attentes et objectifs pour ce projet. Premièrement, il est important de préciser que notre sujet est la programmation d'une intelligence artificielle jouant aux échecs, et non pas d'un jeu d'échecs. L'implémentation de ce dernier n'est supposée être qu'une étape pour parvenir à nos fins, et l'enjeu était donc de ne pas y passer trop de temps afin d'avoir suffisamment de temps à consacrer à l'IA. Néanmoins, comme nous allons le développer dans la partie suivante, nous avons dû écrire toutes les fonctions fondamentales du jeu d'échec (génération des mouvements légaux, détections des échecs et des déplacements qui vont entraîner un échec, etc) en portant un grand intérêt à l'optimisation.

3.1.1 L'efficacité de l'algorithme

Durant tout le développement de notre programme, l'efficacité de l'algorithme a été l'enjeu principal. Un algorithme efficace signifie un algorithme rapide, ce qui signifie une capacité à rechercher davantage de mouvements possibles dans la limite de temps. Notre objectif était de créer une IA capable de nous battre, il était donc crucial d'élaborer l'algorithme le plus efficace possible.

Or, cela ne dépend pas uniquement de la fonction de l'algorithme de l'IA mais de la quasi-totalité des fonctions fondamentales de notre jeu d'échecs (génération des mouvements légaux pour chaque pièce, vérification d'échec, énumération de toutes les pièces pouvant bouger, etc). En effet, la recherche récursive que nous utilisons fait appel à toutes ces fonctions des milliers de fois par seconde. Chaque milliseconde économisée aurait donc un impact positif sur la puissance de notre IA.

3.1.2 Fournir des données à notre IA

Un algorithme rapide et optimisé n'est pas le seul critère quand il s'agit de jouer aux échecs. Il est inutile de parser des milliers de combinaisons de plateaux par seconde si l'on n'est pas capable d'évaluer correctement si ces positions sont viables. Nous avons donc dû trouver des moyens de faire comprendre à notre algorithme que les cavaliers étaient plus puissants au centre de l'échiquier, ou encore qu'il vallait mieux protéger son roi derrière sa tour.

3.2 Nos solutions

Afin d'optimiser notre code, nous avons mesuré le temps d'exécution des fonctions-clés pour détecter les moins efficaces et chercher des solutions.

Pour faire en sorte que notre programme puisse quantifier correctement "la valeur" d'un coup, nous avons choisi de nous baser sur des méthodes d'évaluations heuristiques disponibles sur internet et utilisées par d'autres IAs, car ces dernières ont été conçues par des experts des échecs et prennent en considération des paramètres qui nous échappent. Pour évaluer un coup, notre algorithme utiliserait donc des valeurs matérielles, différentes pour chaque type de pièces, et plus ou moins élevées en fonction de leur importance, mais attribuerait également un bonus ou malus à chaque pièce en fonction de la case où elle se trouverait. Pour cela, nous aurions recours à des tables associant une valeur (spécifique à chaque pièce), pour chaque case de l'échiquier. Ainsi, nous pourrions inciter les cavaliers à contrôler le centre en attribuant à ces cases une valeur plus élevée que les autres.

Afin de fournir davantage de données à notre IA, nous avons également choisi d'implémenter une table contenant une grande quantité de plateaux, tous associés à un score. Nous pourrions ainsi forcer l'IA à jouer certaines ouvertures en leur donnant un score élevé.

3.3 Le résultat attendu

En appliquant toutes ces méthodes, nous nous attendions donc à une IA solide, capable de voir bien plus loin dans l'avenir que nous, à environ 5 coups, et qui pourrait constamment déterminer le meilleur coup possible dans divers contextes. Concrètement, nous voulions qu'elle puisse nous battre et jouer à un niveau d'environ 1500 Elo (bien plus que nous, mais loin des champions).

4 Outils

Réalisant notre projet en binôme, il est indispensable d'utiliser des outils nous facilitant le travail en groupe.

4.1 Github et Git

Github est une plateforme en ligne permettant de stocker, partager et collaborer sur des projets informatique. En effet, ce service dispose de nombreux outils comme la collaboration en temps réel ou encore la gestion des bugs. Il est basé sur Git, un système permettant de gérer des projets informatique. En effet, Git permet de suivre les modifications apportées au code, de créer des branches de développement, de fusionner notre travail ou encore de collaborer en temps réel.

Nous avons donc choisi de créer un projet sur Github et de créer un dépôt Git, manipulable sur nos IDE (Vim et Visual Studio). Ainsi cela nous permettait de coder chacun de notre côté tout en alimentant les branches du projet. Dès lors que l'un de nous deux réalisait une sauvegarde des modifications apportées au code, ou commit, celles-ci étaient présentes sur nos machines. De plus, nous avons pu bénéficier d'un historique listant nos modifications par branche de développement, ce qui permis un suivi de notre progression.

Cependant, nous avons rencontré des problématiques de version de compilateur ou encore d'architecture processeur, pouvant changer l'efficacité voire même freiner la compilation code. Ainsi, nous avons dû utiliser un outil, semblant être le prolongement de Git.

4.2 Docker

Durant la conception de notre projet, le code était essentiellement développé dans des systèmes d'exploitation de la famille Unix. Cependant, nous avons rencontré différents problèmes de compilation du code. En effet, le compilateur "clang" de Mac Os ne compilait pas de la même manière que "gcc" de Linux. De ce fait, nous avons donc téléchargé tous deux la même version du compilateur gcc et créé un makefile afin d'éviter toute ambiguïté. Néanmoins, les problèmes de compatibilité persistaient. En effet, sur MacOS il était impossible de compiler tous les fichiers du projet, ce qui n'était pas le cas sur linux. Par conséquent, nous avons décidé de créer un conteneur Docker afin de pouvoir travailler sur le même environnement.

Docker est un logiciel permettant de créer, déployer et d'exécuter un projet dans un conteneur. Un conteneur est semblable à une boîte qui comporte tous les composants intrinsèques à un projet et qui les regroupe dans un unique paquet pouvant être exécuté sur n'importe quel environnement. En effet, un conteneur docker permet de faire fonctionner un projet de la même manière quel que soit l'appareil.

Ainsi, Docker nous a permis de travailler sur un environnement ayant les mêmes propriétés sans devoir utiliser un système d'exploitation virtuel.

5 Le code

Ce chapitre sera dédié au fonctionnement détaillé de notre code, et aux raisons qui ont déterminé nos choix.

5.1 Structures et définitions

Pour commencer, nous allons lister les différents types de données et structure utilisées, ainsi que leur utilité. Toutes ces déclarations se trouvent dans le fichier d'en-tête de notre programme : "chess.h".

5.1.1 Échiquier et types de données

Le plus important est la méthode choisie pour représenter le plateau et ses pièces : un tableau de caractères non signés. On peut se dire au premier abord qu'un tableau à deux dimensions serait nécessaire mais nous avons préféré faire simple, un tableau standard étant largement suffisant une fois les fonctions permettant de calculer une ligne et une colonne à partir de l'index du tableau déterminées. De plus, un tableau à deux dimensions est plus lent car les données ne sont pas stockées de manière contiguë en mémoire.

Pour représenter les pièces, nous avons associé chacune d'entre elles à un nombre, ce qui nous en donne 12 au total (deux autres seront rajoutées par la suite). Intuitivement, nous avons alors attribué des valeurs entières aux pièces, avant de revenir plus tard sur notre décision. Pour optimiser notre code au maximum, nous avons décidé d'attribuer un type de données plus adapté à ces variables. Nous savions en effet qu'elles n'auraient pas besoin de prendre beaucoup de valeurs différentes, et utiliser un entier codé sur 32 bits sur nos machines était inutile : nous avions besoin de 15 valeurs possibles (vide + 14 pièces), pas 2147483647. Nous avons donc choisi le type caractère non signé qui occupe qu'un seul octet en mémoire, et permet 256 valeurs différentes. Nous avons par la suite remplacé ce type par "uint8_t", qui correspond à un entier non signé codé sur 8 bits, quelle que soit l'architecture du processeur.

Pour distinguer les pièces noires des pièces blanches, nous avons créé une constante "PIECE_NOIRE" (de valeur 128) à ajouter à chaque pièce, ce qui nous permettait facilement de distinguer la couleur des pièces sans faire de cas par cas. Une deuxième constante "PIECE_SPECIALE" avec le même fonctionnement a été ajoutée plus tard pour distinguer les rois ayant déjà bougé depuis le début de la partie afin d'implémenter la règle du roque.

L'utilisation de variables de type approprié en fonction des différentes valeurs qui allaient leur être attribuée est un principe que nous avons appliqué dans tout notre programme. Ainsi, de nombreuses fonctions retournent des valeurs de type "uint8_t" et non int.

5.1.2 Liste chaînée

Dans notre programme, beaucoup de fonctions doivent nous retourner des listes (liste des pièces pouvant bouger, ou liste des déplacements possibles d'une pièce par exemple). Nous avons d'abord choisi d'utiliser des tableaux, car ces derniers sont les

plus rapides et leurs opérations de lecture et écriture sont en temps constant, mais cela nous a causé plus de problèmes que prévu.

En effet, les listes que nous utilisons n'étaient pas toujours de même taille : selon la situation, une reine peut avoir entre 0 et 27 déplacements possibles, mais allouer un tableau de taille 27 pour la reine n'était pas efficace, car il serait majoritairement rempli de -1, la valeur utilisée pour signifier qu'aucun déplacement n'est possible. Le même problème s'applique pour tous les autres tableaux, comme celui des pièces pouvant bouger, qui ne contiendra jamais 16 valeurs.

Après avoir réalisé à quel point cette approche était inefficace, nous avons décidé d'utiliser des listes chaînées, pour leur taille variable et leur flexibilité. Malgré le fait que la vitesse des opérations de recherche dépend de la position de la valeur dans la liste, l'impact sur les performances est moindre car ces listes ne dépassent que très rarement les 15 éléments.

```
48  typedef struct liste
49  {
50      uint_fast8_t valeur;
51      struct liste *next;
52  } liste;
53
```

FIGURE 1 – Notre implémentation de la liste chaînée pour ce projet

5.1.3 Informations sur la partie

Pour stocker les informations importantes sur la partie en cours (nombre de pièces pour chaque couleur, nombre de tours écoulés, échec...), nous avons créé une structure de données appelée "FEN", car initialement inspirée de la notation FEN. Au cours du développement, cette structure a évolué pour répondre à nos nouveaux besoins et contient à présent des données différentes de celles du FEN standards, mais nous avons choisi de conserver ce nom.

```
typedef struct FEN
{
    int tour;
    int nb_pcs_w;
    int nb_pcs_b;
    int nb_tours;
    int half_move;
    int_fast8_t full_move;
    int echec;
    int echec_et_mat;
    int_fast8_t capture;
    int_fast8_t endgame;
} FEN;
```

FIGURE 2 – La structure d'informations sur la partie : "FEN"

5.1.4 Description d'un déplacement

Afin de nous retourner le résultat de sa recherche récursive, notre algorithme minimax remplit une structure de données appelée "best_move", composée de la position de la pièce à bouger, la position où nous devons la déplacer, et du score engendré par ce déplacement.

```
77 typedef struct best_move
78 {
79     uint_fast8_t piece;
80     int_fast8_t move;
81     int score;
82 } best_move;
```

FIGURE 3 – La structure "best_move", utilisée pour décrire un déplacement

5.1.5 Table de hachage

Nous avons aussi décidé d'implémenter une table de hachage dans notre programme, pour stocker des configurations de plateaux (appelées positions) associées à leur valeur (score). Pour ce faire, nous avons utilisé la technique du hachage de Zobrist, que nous détaillerons dans la partie suivante, afin d'obtenir une clé unique pour chaque position. Chaque clé est donc stockée avec son score associé.

Notre implémentation de la table de hachage est une structure nommée "Hash_table" composée d'un entier correspondant au nombre d'entrées dans la table, et d'un tableau d'entrées. Chaque entrée est une structure nommée "Entry" composée d'un entier correspondant au score et d'un "unsigned long long" correspondant à la clé unique de cette position.

```
84 typedef struct Entry
85 {
86     U64 posKey; // position
87     int score; // score associé
88 } Entry;
89
90 typedef struct Hash_table
91 {
92     Entry **entries; // tableau d'entries (poskey + move)
93     int nb_entries;
94 } Hash_table;
95
```

FIGURE 4 – Les structures "Hash_table" et "Entry" représentant la table de hachage et les éléments de la table

5.2 Fonctions principales

Cette section sera dédiée à la présentation de plusieurs fonctions importantes de notre jeu, qui ont été subites à diverses optimisations au fil du développement. Nous nous concentrerons d'abord sur les fonctions fondamentales du jeu d'échecs, puis sur celles utilisées par l'IA.

5.2.1 Jeu d'échecs

Liste des pièces pouvant bouger L'énumération des pièces du plateau pouvant se déplacer est essentielle au fonctionnement du jeu. Elle est par exemple utilisée pour empêcher l'utilisateur de choisir une pièce qui ne peut pas être déplacée, pour vérifier si il y a échec et mat (si aucune pièce d'une couleur ne peut bouger, alors elle a perdu la partie), ou encore pour évaluer chaque déplacement de chaque pièce, notre IA a besoin de savoir quelles pièces analyser.

La fonction "liste_moves" du fichier "chess.c" prend en argument une liste chaînée à remplir, la couleur dont on veut lister les pièces un plateau. Voici l'algorithme utilisé :

Nous récupérons le nombre de pièces de la couleur choisie avec un appel à la fonction "compter_pieces".

Nous parcourons ensuite l'ensemble de l'échiquier tant que nous n'avons pas analysé toutes les pièces de la couleur spécifiée. Pour ce faire nous incrémentons un compteur à chaque fois que nous analysons une pièce de la bonne couleur, que nous comparons au nombre de pièces calculé à l'étape précédente.

A chaque pièce de la bonne couleur trouvée (grâce à la fonction get_color), nous récupérons la liste des déplacements possibles de cette pièce avec un appel à la fonction "get_legal_all".

Si la liste des mouvements de la pièce n'est pas vide, nous ajoutons la position de la pièce à la liste des pièces pouvant bouger.

Une fois toutes les pièces de la couleur vérifiées, nous n'avons plus qu'à retourner la liste des pièces pouvant bouger qui est maintenant complète.

```
// liste des pieces de couleur qui peuvent bouger
liste *liste_moves(int_fast8_t couleur, liste *liste_pieces, uint_fast8_t *plateau)
{
    int_fast8_t i, j = 0;
    liste *moves = (liste *)malloc(sizeof(liste));
    moves = NULL;
    liste_pieces = NULL;

    int_fast8_t nb_pieces = compter_pieces(couleur, plateau); // nombre de pieces de couleur
    while (i < TAILLE_ECHEQUIER && j < nb_pieces)
    {
        // on regarde les moves dispo pour chaque piece alliée qu'on trouve
        if (get_color(plateau[i]) == couleur)
        {
            ++j;
            moves = get_legal_all(i, moves, plateau);

            // on regarde si il y a au moins 1 move dispo:
            if (moves != NULL)
            {
                liste_pieces = ajout_tete(liste_pieces, creation_maillon(i));
                liste *t = moves;
                liberation(moves);
            }
            ++i;
        }
    }
    return liste_pieces;
}
```

FIGURE 5 – La fonction "liste_moves"

Vérification des échecs et échecs et mat La vérification des échecs a lieu après chaque tour, pour mettre à jour notre structure "FEN", qui contient un champ "echec", mais également à chaque fois que nous voulons lister les mouvements possibles d'une pièce, car il est impossible d'effectuer un déplacement qui va nous mettre en échec. Pour obtenir cette liste, la fonction à utiliser est "get_legal_all", qui récupère une liste de toutes

les cases de l'échiquier où une pièce peut potentiellement aller, puis appelle la fonction "retirer_echec", qui pour chacune des cases contenues dans cette liste, va effectuer le déplacement puis appeler la fonction "vérifier_echec" sur le plateau engendré.

La fonction "vérifier_echec" est donc appelée plusieurs milliers de fois à chaque itération de l'algorithme minimax, et il était crucial pour nous de l'optimiser afin d'améliorer la performance de l'IA. Nous allons maintenant expliquer cet algorithme. Pour éviter de rechercher tout l'échiquier, la fonction prend une couleur en paramètres : "NOIR" ou "BLANC" (respectivement 1 et 2), et recherche les échecs en parcourant les mouvements possibles de toutes les pièces ennemies à la recherche d'une case contenant le roi allié.

Pour ce faire, si l'on veut vérifier si les noirs sont en échec par exemple, on commence par obtenir le nombre de pièces blanches présentes sur le plateau, grâce à la fonction "compter_pieces".

On parcourt ensuite l'échiquier tant que nous n'avons pas analysé toutes les pièces de couleur blanche (en utilisant un compteur comme dans la fonction précédente).

A chaque pièce de couleur blanche trouvée, on incrémente le compteur, puis on récupère la liste de toutes les cases où la pièce peut se déplacer (y compris celles causant un échec), avec la fonction "get_legal_any".

Pour chacune de ces cases, nous allons ensuite regarder si elle contient le roi noir. Si une pièce blanche peut se rendre dans une case contenant un roi noir, alors les noirs sont en situation d'échec. Dans le cas où le roi noir est présent dans une des listes, on peut directement retourner la valeur "NOIR", indiquant que les noirs sont en échec. Si jamais l'algorithme continue jusqu'au bout sans trouver d'échec, la valeur retournée sera -1.

```

liste *moves = (liste *)malloc(sizeof(liste));
moves = NULL;
liste *tmp, *t;
int_fast8_t i, j = 0, nb_pieces, echec = -1;

if (couleur == NOIR) // on va chercher dans les moves des blancs
{
    nb_pieces = compter_pieces(BLANC, plateau);
    i = TAILLE_ECHEQUIER - 1; // car les blancs commencent en bas
    while (j < nb_pieces && i >= 0)
    {
        if (get_color(plateau[i]) == BLANC)
        {
            j++; // on est sur une piece blanche
            moves = get_legal_any(i, moves, plateau);
            tmp = moves;
            while (tmp != NULL)
            {
                if (plateau[tmp->valeur] == ROI+PIECE_NOIRE || plateau[tmp->valeur] == ROI+PIECE_NOIRE+PIECE_SPECIAL)
                {
                    liberation(moves);
                    return NOIR;
                }
                tmp = tmp->next;
            }
            liberation(moves);
        }
        i--;
    }
}
return echec;
}

```

FIGURE 6 – La partie de la fonction "verifier_echec_fast" qui vérifie si les noirs sont en échec

Réalisation des déplacements et roque Pour effectuer un déplacement, notre approche initiale était très minimaliste : il suffit de connaître la case de la pièce à bouger et la case où elle doit aller (indices du tableau échiquier), pour attribuer à la case destination la valeur présente dans la case source, puis d'attribuer la valeur "VIDE" à

la case source. Cependant cette approche rudimentaire a atteint sa limite lorsque nous avons implémenté des déplacements plus complexes comme le roque ou la promotion. C'est pourquoi nous avons dû créer la fonction "effectuer_move", qui prend en argument l'indice d'une case source, l'indice d'une case destination, et un pointeur vers un tableau d'uint8_t correspondant au plateau dans lequel effectuer le déplacement.

La fonction vérifie si les conditions d'un déplacement spécial sont remplies, avant d'appliquer la méthode standard décrite ci-dessus.

Dans l'exemple du roque, elle vérifie d'abord que la pièce à déplacer soit un roi n'ayant jamais bougé (différencié des rois ayant bougé grâce à la constante "PIECE_SPECIALE" ajoutée à leur valeur). Elle vérifie ensuite que la case où le roi veut se rendre correspond à celle du grand ou du petit roque, que la tour du côté correspondant soit en place, et appelle la fonction "vide" pour déterminer si la valeur des cases séparant le roi de sa tour est bien "VIDE". Dans le cas où toutes ces conditions sont remplies, le déplacement spécial des deux pièces à la fois peut être effectué.

```

1430     if (plateau[position_piece] == ROI && position_move == 62) // castle blanc coté roi
1431     {
1432         plateau[62] = ROI + PIECE_SPECIAL;
1433         plateau[position_piece] = VIDE;
1434         plateau[61] = plateau[63];
1435         plateau[63] = VIDE;
1436     }
1437
1438     else if (plateau[position_piece] == ROI && position_move == 58) // blanc coté reine
1439     {
1440         plateau[58] = ROI + PIECE_SPECIAL;
1441         plateau[position_piece] = VIDE;
1442         plateau[59] = plateau[56];
1443         plateau[56] = VIDE;
1444     }
1445
1446     else if (plateau[position_piece] == ROI + PIECE_NOIRE && position_move == 6) // castle noir roi
1447     {
1448         plateau[6] = ROI + PIECE_NOIRE + PIECE_SPECIAL;
1449         plateau[position_piece] = VIDE;
1450         plateau[5] = plateau[7];
1451         plateau[7] = VIDE;
1452     }
1453
1454     else if (plateau[position_piece] == ROI + PIECE_NOIRE && position_move == 2) // castle noir coté reine
1455     {
1456         plateau[2] = ROI + PIECE_NOIRE + PIECE_SPECIAL;
1457         plateau[position_piece] = VIDE;
1458         plateau[3] = plateau[0];
1459         plateau[0] = VIDE;
1460     }
1461

```

FIGURE 7 – La partie de la fonction "effectuer_move" qui vérifie si le déplacement à effectuer est un roque

5.2.2 IA

Nous allons à présent expliquer les algorithmes et fonctions clés au fonctionnement de l'IA. Ces fonctions se trouvent dans les fichiers "minimax.c" et "hashtable.c".

Calcul du score Une bonne implémentation de l'algorithme minimax ne vaut rien si la fonction d'évaluation n'est pas au point, car c'est la valeur de retour de cette dernière qui est utilisée pour départager les bonnes positions des mauvaises. Comme expliqué dans le cahier des charges, notre fonction d'évaluation du score s'appuie sur deux facteurs : la valeur matérielle de chaque pièce, ainsi que les bonus/malus attribués en fonctions des cases qu'elles occupent.

La fonction "get_score" prend en argument une couleur et un tableau correspondant au plateau que l'on veut évaluer, et retourne un score pour la couleur choisie, obtenu en calculant la somme des valeurs matérielles et des bonus de placements de toutes les pièces de la couleur.

La valeur d'un plateau est toujours centrée par rapport au côté noir, ce qui signifie

qu'elle est positive si la fonction estime qu'ils sont dans une position avantageuse, et négative sinon.

Pour l'évaluation matérielle, nous avons défini les valeurs de chaque pièce dans le fichier "chess.h", la somme des valeurs de chaque pièce d'une couleur est calculée par la fonction "get_valeur_materielle_totale", qui itère sur tout l'échiquier et incrémente la variable retournée "materiel_total" avec la valeur de la pièce analysée.

Pour décider de quelles valeurs attribuer à chaque pièce, nous avons fait beaucoup de recherches en ligne, puis fait jouer l'IA contre elle-même avec des fonctions d'évaluation différentes. Les deux méthodes d'évaluations qui ont le mieux fonctionné avec notre IA sont celle de l'IA Sunfish, dont le code est Open Source, et la "Simplified Evaluation Function", créée par Tomasz Michniewski, sur laquelle nous avons lu l'article du Chess Programming Wiki. Nous avons décidé d'utiliser cette méthode car elle nous paraissait être la plus stable, que nous avions un article à notre disposition expliquant les choix derrière toutes ces valeurs, mais surtout car l'IA qui l'utilisait battait celle dont l'évaluation était basée sur la méthode Sunfish.

```
#define VALEUR_PION 10
#define VALEUR_CAVALIER 32
#define VALEUR_FOU 33
#define VALEUR_TOUR 50
#define VALEUR_REINE 90
#define VALEUR_ROI 2000
```

FIGURE 8 – Les valeurs matérielle des pièces utilisées par notre programme

Pour ajouter une valeur à une pièce selon la case où elle se trouve, nous avons défini un tableau de 64 cases pour chaque pièce contenant les bonus à attribuer pour chaque case. Par exemple si un pion se trouve dans la 24ème case de l'échiquier, il faut ajouter au pion la valeur contenue dans la 24ème case du tableau pour les pions. Ces tables sont appelés "Piece Square Tables", et les valeurs qu'elles contiennent viennent également de la "Simplified Evaluation Function" mentionnée précédemment.

```
const int table_cavalier[TAILLE_ECHEQUIER] = {
    -50, -40, -30, -30, -30, -30, -40, -50,
    -40, -20, 0, 0, 0, 0, -20, -40,
    -30, 0, 10, 15, 15, 10, 0, -30,
    -30, 5, 15, 20, 20, 15, 5, -30,
    -30, 0, 15, 20, 20, 15, 0, -30,
    -30, 5, 10, 15, 15, 10, 5, -30,
    -40, -20, 0, 5, 5, 0, -20, -40,
    -50, -40, -30, -30, -30, -30, -40, -50};
```

FIGURE 9 – Exemple de "Pièce Square Table", il s'agit ici de celle du cavalier

Un problème se pose : nous possédons 6 tables, une pour chaque pièce, mais leurs valeurs doivent être différentes en fonction de la couleur de la pièce. Les 6 tables que nous avons sont adaptées aux blancs, il nous a donc de déterminer une solution rapide pour inverser ces valeurs. Nous avons donc créé un 7ème tableau, appelé "table_miroir",

qui associe à chaque case l'indice de la case symétrique par rapport au centre du plateau. Pour obtenir la valeur à attribuer à une pièce noire, il suffit donc de récupérer la valeur à l'indice "table_miroir[indice]" de la table appropriée.

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

FIGURE 10 – La table miroir utilisée pour appliquer les valeurs aux pièces noires

Implémentation de l'algorithme minimax Pour développer une IA dans un jeu à deux joueurs à information complète (comme les échecs), l'algorithme à utiliser est Minimax. Il s'agit d'un algorithme de recherche récursive dont le but est de maximiser le gain d'un joueur en partant du principe que l'adversaire cherche également à maximiser son propre gain. Voici les étapes que ce dernier suit dans notre implémentation.

Notre fonction "minimax" retourne une valeur entière : un score, et prend en argument la valeur 0 ou 1 pour indiquer si l'on cherche à maximiser le score (calculé par rapport aux noirs) ou à le minimiser, la couleur du joueur dont on cherche à maximiser ou minimiser le score, le plateau (tableau de 64 cases) sur lequel on souhaite effectuer la recherche, et la profondeur de la recherche que l'on souhaite effectuer, qui correspond au nombre de récursions, et donc à la distance à laquelle l'algorithme va regarder dans le futur.

La première étape de notre fonction est donc de retourner le score du plateau qui lui a été fourni, avec un appel à "get_score" si la profondeur est égale à 0.

Sinon, dans le cas où l'on veut maximiser le score, on va récupérer la liste de chaque pièce de pouvant bouger (de la couleur du joueur appelant, et pour chacune d'entre elles, récupérer la liste de ses déplacements possibles ("liste_moves" et "get_legal_all"). On va ensuite effectuer chaque déplacement contenu dans la liste et stocker dans une variable appelée "eval" le résultat d'un appel à minimax avec pour paramètre la couleur ennemie, 0 pour indiquer qu'on veut minimiser, le plateau engendré par le déplacement et profondeur-1. L'algorithme va donc évaluer le plateau de la perspective de l'ennemi (qui cherche à minimiser le score), en regardant le résultat de l'évaluation de chacun de ses déplacements possibles. Cela va se répéter jusqu'à ce que la profondeur atteigne 0, et on récupérera ainsi dans "eval" le score de notre couleur après un échange d'un nombre de coup égal à la profondeur, ou chaque joueur joue son meilleur coup possible. La valeur "eval" est ensuite comparée à "best_eval" : l'évaluation la plus haute trouvée jusqu'à présent (initialisée à INT_MIN pour être certain de trouver mieux).

Si l'on a trouvé une nouvelle meilleure évaluation, on peut alors remplir la structure "best_move" avec le déplacement qu'on a effectué, et le score engendré (la valeur d'"eval"). Il nous suffit ensuite d'incrémenter les compteurs de boucles et de réinitialiser le plateau à sa position originale, grâce à un tableau appelé "plateau_backup" alloué précédemment, et de recommencer l'opération tant qu'il reste des pièces à analyser. Lorsqu'on a fini, on retourne "best_eval" : le meilleur score possible à l'issue de profondeur coups.

Dans le cas où on veut minimiser le score, les étapes sont les mêmes à l'exception de l'appel à minimax sur le plateau engendré par le déplacement testé, ou l'on va cette fois-ci chercher à maximiser le score pour se mettre dans la perspective de l'ennemi. La variable "best_eval" sera à "INT_MAX", car on cherche le pire score possible. Il faudra donc remplacer "best_eval" par "eval" si cette dernière lui est inférieure.

Utilisation de la table de hachage Dans une table de hachage, il est essentiel pour toutes les valeurs d'avoir un index unique. Dans le cas des échecs, nous avons utilisé la méthode de Zobrist pour obtenir une clé unique pour chaque configuration d'échiquier (position). Elle consiste à remplir un tableau de la taille de l'échiquier (64) et un tableau de la taille du nombre de pièces (14) avec des valeurs aléatoires, puis à effectuer l'opération logique "XOR" entre ces deux tableaux et le plateau sur lequel nous voulons appliquer le hachage. Le résultat est stocké dans une variable de type "unsigned long long".

```

48 // Genere un hash unique pour chaque échiquier
49 U64 generate_posKey(uint_fast8_t *plateau, int_fast8_t tour)
50 {
51     int i = 0;
52     U64 FinalKey = 0;
53     uint_fast8_t piece = VIDE;
54
55     // pieces
56     for (i = 0; i < TAILLE_ECHEQUIER; ++i)
57     {
58         piece = get_piece_index(plateau[i]); //change les valeurs des pièces pour [1-14]
59         if (piece != VIDE)
60         {
61             if (piece >= PION && piece <= ROI + PIECE_NOIRE + PIECE_SPECIAL)
62             {
63                 FinalKey ^= PieceKeys[piece][i];
64             }
65             else
66             {
67                 fprintf(stderr, "\nERREUR KEY\n");
68                 return 0;
69             }
70         }
71     }
72
73     if (tour == BLANC)
74     {
75         FinalKey ^= SideKey;
76     }
77
78     return FinalKey;
79 }

```

FIGURE 11 – La fonction "generate_poskey", qui permet d'obtenir la clé unique pour chaque position

Le but initial lors de l'implémentation de la table de hachage était de gagner du temps de recherche en stockant les évaluations des positions que l'algorithme rencontrait, pour ne pas avoir à les réévaluer récursivement à nouveau si il les retrouvait plus tard dans ses recherches. Ce concept est sensé être efficace car aux échecs, quel que soit la série d'échanges qui a donné lieu à une position, l'évaluation sera toujours la même. Nous avons donc rédigé une fonction "search_table", qui retourne le score de la position recherchée si elle est trouvée dans la table ("INT_MIN" sinon) , et une fonction "add_entry" qui ajoute à la table la position spécifiée. Pour décrire la position, les fonctions utilisent la clé "posKey" unique à chaque position générée par la fonction "generate_posKey".

Dans la fonction minimax, après avoir effectué un déplacement, nous pouvons donc

appeler la fonction "seach_table" sur la position engendrée par le déplacement. Si la fonction retourne un score, nous pouvons directement attribuer cette valeur à "eval", sinon ("INT_MIN"), nous évaluons la position avec la fonction "minimax" puis nous ajoutons l'évaluation à la table avec la fonction "add_entry".

```
void add_entry(Hash_table *hashtable, U64 posKey, int score)
{
    if (hashtable->nb_entries == MAX_TABLE_SIZE - 50)
    {
        return;
    }
    Entry *new_entry = (Entry *)malloc(sizeof(Entry));
    if (new_entry == NULL)
    {
        fprintf(stderr, "\nERREUR MALLOC NEWENTRY\n");
        return;
    }
    new_entry->posKey = posKey;
    new_entry->score = score;

    hashtable->entries[hashtable->nb_entries] = new_entry;
    hashtable->nb_entries++;
}

146 // parcours la hashtable, retourne un score si il existe une entrée avec la même clé,
147 // INT_MIN sinon
148 int search_table(Hash_table *hashtable, U64 posKey)
149 {
150     for (int i = 0; i < hashtable->nb_entries; ++i)
151     {
152         if (hashtable->entries[i]->posKey == posKey)
153         {
154             return hashtable->entries[i]->score;
155         }
156     }
157     return INT_MIN; // la clé n'est pas dans la hashtable
158 }
```

FIGURE 12 – Les fonctions "add_entry" et "search_table"

Cette approche s'est malheureusement avérée inefficace car plus lente que la méthode normale, sûrement car les opérations de lecture/écriture dans notre table étaient trop lentes. En effet notre table étant un tableau d'entrées (score et clé), plus la taille de ce dernier est grande, plus la recherche va être lente. Nous avons essayé plusieurs tailles de tableau différentes sans succès.

Il nous a donc fallu innover pour chercher à rendre tout ce travail rentable. Nous avons alors choisi d'utiliser la table en lecture seulement dans minimax, en remplissant la table avant l'exécution de l'algorithme. Pour cela, nous avons ajouté un fichier texte "hashtable.txt" dans le dossier de notre projet, et nous avons rempli ce fichier de clés associées à leur score en remplaçant dans la fonction "minimax" l'appel à "add_entry" par une écriture dans le fichier avec "fprintf". Le fichier a atteint une taille de plus de 5 millions de lignes en à peine 30 secondes de partie IA contre IA avec une profondeur de 4.

489683	4953981477664717150	-50
489684	2484668829916472571	-45
489685	3984397546903151919	-40
489686	11564352355876230952	-50
489687	4167398639939891784	-35
489688	4448524141817686971	-30
489689	2354807575321380010	-30
489690	1294029989944556834	-40
489691	15582299279634212030	0
489692	18310197597104848709	-40
489693	11416449319864457215	-5
489694	6981073086438344548	-70
489695	8666876706576947972	-35
489696	17290046626697358258	-65
489697	12480914828356346522	-60
489698	16400939592229080948	-75
489699	3134509781006836195	-70
489700	11904303846301276713	-80
489701	10177886658635643281	-85
489702	15284011495379540844	-30
489703	1096505295955528771	-50

FIGURE 13 – Aperçu du fichier "hashtable.txt"

Nous avons créé une nouvelle fonction, "fill_from_file", qui remplit la table avec les données présentes dans le fichier. Au lancement du programme, si l'utilisateur souhaitait utiliser l'IA, nous appelions donc cette fonction pour remplir la table avec toutes les données collectées auparavant. Néanmoins, même en réduisant le nombre de lignes présentes dans le fichier, cette méthode ne nous a pas permis de gagner en efficacité de manière consistante car la durée de recherche dépendait de la position de la valeur dans la table. Nous n'avons donc pas pu améliorer la performance de notre algorithme comme nous le souhaitions.

```

192 void fill_from_file(hash_table *hashtable)
193 {
194     FILE *fp = fopen("aaa.txt", "r");
195     if (fp == NULL)
196     {
197         fprintf(stderr, "ERR");
198         return;
199     }
200     int i = 0, int_buf;
201     U64 llu_buf;
202     int nb_lignes = 0;
203
204     nb_lignes = 489703;
205
206     while (i < nb_lignes)
207     {
208         fscanf(fp, "%llu", &llu_buf);
209         fseek(fp, 3, SEEK_CUR); // on décale le pointeur de 3 pour sauter " | "
210         fscanf(fp, "%d", &int_buf);
211         fseek(fp, 1, SEEK_CUR); // on décale le pointeur de 1 pour sauter le "\n"
212         add_entry(hashtable, llu_buf, int_buf);
213         i++;
214     }
215 }

```

FIGURE 14 – La fonction "fill_from_file", permettant de charger le contenu du fichier "hashtable.txt" dans la table.

Notre dernière idée pour la table était de l'utiliser pour compenser un défaut de l'IA plutôt que d'essayer de la rendre plus rapide, en lui "apprenant" des ouvertures. En effet, notre algorithme ne peut voir que 4 ou 5 coups dans l'avenir, ce qui le rend très faible à l'ouverture et en fin de partie, car c'est typiquement les phases de la partie où des stratégies connues permettent même aux humains de jouer sur le long terme. Il existe un nombre bien trop grand de combinaisons de positions en fin de partie, même

lorsqu'il ne reste que 5 pièces en jeu. Nous nous sommes donc rabattus sur l'idée d'ajouter des ouvertures (et leurs contre-attaques) communes dans notre table de hachage. Pour ce faire, il nous suffisait de collecter les clés des positions communes (pion e4 par exemple), et de "mentir" à notre IA en leur attribuant un score plus élevé ou bas (selon si il s'agit d'une défense ou d'une attaque) que ce que notre fonction d'évaluation aurait déterminé. Nous avons ainsi ajouté les positions-types des débuts de partie les plus courants : Ruy Lopez, partie Italienne, partie Ecossaise, Gambit du roi, partie Viennoise, défenses Sicilienne, Française, Scandinave. Le but était de rendre notre algorithme plus solide en lui apprenant la façon formelle de réagir face à ces ouvertures. Nous n'avons néanmoins pas obtenu le résultat attendu : notre IA jouait de manière encore plus imprévisible qu'avant, et se résolvait souvent à son mouvement par défaut lorsqu'elle ne sait pas quoi faire : bouger une tour d'une case. Nous supposons que cela est dû au fait que toutes ces ouvertures partagent des positions en commun, et que lors de la recherche récursive, l'algorithme est incapable d'en choisir une qui ne mènera pas l'ennemi à un score trop avantageux, à causes d'évaluations introduites artificiellement dans la table.

Tous les usages que nous avons trouvé à cette table de hachage n'ont malheureusement pas fonctionné comme prévu, et nous nous sommes donc résolus à ne pas l'utiliser dans notre programme final. Néanmoins, les fonctions sont quand même disponibles dans le code, car elles représentent une grande partie de notre travail sur la phase finale de ce projet, et font partie des fonctionnalités les plus complexes et réfléchies de notre programme.

5.3 Optimisation

5.3.1 Élagage Alpha Beta

L'élagage alpha beta est une technique d'optimisation de l'algorithme minimax très efficace car elle permet de réduire drastiquement le temps d'exécution de l'algorithme, et d'effectuer des recherches avec une profondeur plus élevée. Il permet à l'algorithme d'arriver à un résultat concluant en n'explorant qu'une partie de l'arbre des combinaisons. Pour cela, l'algorithme part du principe que l'adversaire choisira toujours le meilleur coup possible pour minimiser notre profit. Par exemple, dans le cas où l'on veut maximiser le score noir, l'adversaire jouera le coup qui nous accorde le score le plus faible possible à l'issue de l'échange. Ainsi, si l'on évalue un déplacement qui engendrera forcément un score plus faible que le meilleur score possible ("best_eval") actuel, on peut directement l'ignorer.

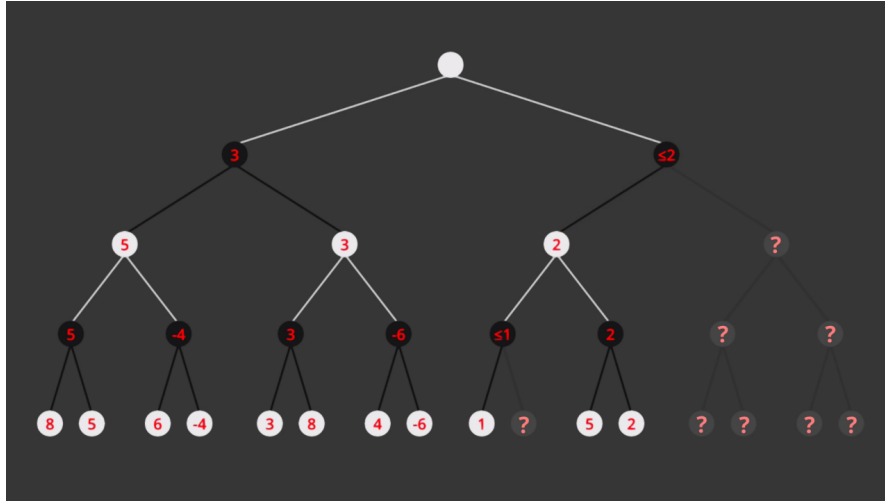


FIGURE 15 – Schéma représentant l'arbre des positions analysées par minimax

Dans le schéma ci-dessus, chaque joueur ne possède que deux options différentes à chaque tour. Aux échecs, il y en a des dizaines, mais le fonctionnement est le même. Le score est calculé par rapport aux blancs, ce qui signifie que les blancs vont toujours choisir le déplacement leur permettant de permettre le score le plus élevé, tandis que les noirs vont chercher le score le plus bas possible.

La partie gauche de l'arbre a été évaluée normalement en inspectant à chaque niveau de profondeur les deux options possibles. L'évaluation de la partie droite de l'arbre s'effectue après la gauche, l'algorithme sait donc déjà qu'il peut obtenir un score de 3 en jouant le coup de gauche. Il est donc inutile de poursuivre l'évaluation des déplacements marqués "?" car on sait que les noirs ont la possibilité de choisir un déplacement entraînant un score inférieur à 3. Comme les noirs cherchent à minimiser le score le plus bas, même si la branche de droite contient la valeur 50, les blancs ne pourront jamais l'atteindre car les noirs ne la choisiront pas. Ainsi, à ce stade de la recherche, les blancs sont déjà sûrs du coup à jouer. On peut ainsi gagner énormément de temps en ne prenant pas la peine d'évaluer ces positions qui n'arriveront jamais.

Pour implémenter cette optimisation dans le code, il faut ajouter deux paramètres à notre fonction : "alpha" et "beta", respectivement initialisés à "INT_MIN" et "INT_MAX". Dans le cas où on veut maximiser le score, après chaque évaluation, on compare le score trouvé par l'appel à "minimax" ("eval") avec alpha. Si "eval" est plus élevée, alpha prend la valeur de "eval". On compare ensuite "alpha" avec "beta", et on arrête l'algorithme dans le cas où "alpha" est supérieur ou égal à "beta". Dans le cas contraire, "beta" prend la valeur de "eval" dans le cas où "beta" est plus petite (au lieu de "alpha" si elle est plus grande). Encore une fois, on peut stopper la recherche si "alpha" est plus grand que beta.

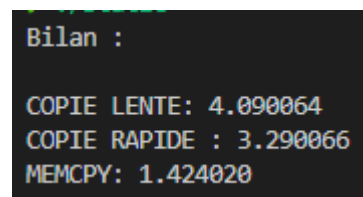
5.3.2 Opérations de copie du plateau

La copie du plateau actuel vers un nouveau est une opération effectuée par la fonction minimax, qui doit garder un backup afin de réinitialiser le plateau quand on a fini de tester les conséquences d'un déplacement, mais également par la fonction "retirer_echec", qui va réinitialiser le plateau après avoir essayé un mouvement et avant de passer au suivant. il a donc été nécessaire de chercher à optimiser cette copie.

A l'origine nous utilisons une fonction que nous avons rédigé pour simplement utiliser

une boucle for pour assigner à chaque case du tableau destination la valeur contenue dans celle du tableau source. Or, lors de nos recherches pour une méthode plus poussée, nous avons découvert la fonction "memcpy", présente dans la bibliothèque "string.h". Elle permet de copier un nombre spécifié d'octets d'une zone de mémoire vers une autre. Le nom d'un tableau étant un pointeur vers son premier élément, il nous suffit d'appeler cette fonction avec les noms de nos tableaux et la taille $64 * \text{sizeof}(\text{uint8_t})$ (= 64 car le type uint8_t occupe toujours 8 bits) pour copier le contenu de l'un dans l'autre.

La fonction "memcpy" est en général plus rapide qu'une copie "à la main" comme nous utilisons, et pour vérifier cela, nous avons mesuré le temps d'exécution des deux fonctions en utilisant le type "clock_t", la fonction "clock" et la macro "CLOCKS_PER_SEC" inclus dans la bibliothèque "time.h". Nous avons donc écrit un programme qui comparait l'efficacité des deux fonctions en copiant 10 millions de tableaux de "uint8_t" de taille 64 avec chacune d'entre elles, puis affichait le temps d'exécution.



```
Bilan :  
  
COPIE LENTE: 4.090064  
COPIE RAPIDE : 3.290066  
MEMCPY: 1.424020
```

FIGURE 16 – Résultat d'un programme comparant le temps d'exécution de trois fonctions de copie de tableaux

La seule différence entre les fonctions "copie_lente" et "copie_rapide" est l'incrément de la variable d'index ("i++" pour la lente et "++i" pour la rapide). On remarque ainsi une forte supériorité de "memcpy". L'utilisation de cette fonction dans le programme a permis de réduire le temps de recherche de minimax durant toute une partie IA contre IA avec une profondeur de 4 de 4 secondes en moyenne sur nos machines (132,505 secondes en moyenne avec notre fonction originale, puis 128,574 avec "memcpy").

5.3.3 Optimisations du compilateur

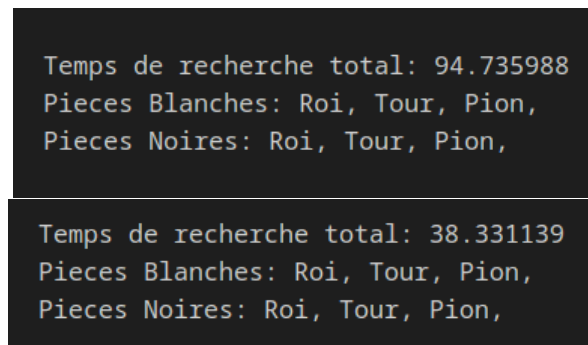
La deuxième optimisation ayant eu le plus d'impact après l'implémentation de l'alpha beta pruning est de loin la plus simple. Il nous a suffi d'ajouter un simple flag dans la commande gcc de notre makefile pour diviser le temps d'exécution par pratiquement 2, à un stade où les optimisations précédentes étaient déjà toutes appliquées.

Le flag "-O3" indique à gcc d'optimiser l'exécutable créé pour la meilleure vitesse d'exécution possible. L'utilisation de ce flag n'est pas toujours recommandée car elle peut avoir des effets indésirables, mais nous n'en avons trouvé aucun avec notre programme. En utilisant les optimisations du compilateur, nous avons pu passer d'un temps d'exécution d'en moyenne 128,574 secondes à 64,841 pour une partie IA contre IA avec une profondeur de 4.

5.3.4 Optimisations mineures

Nous avons également apporté beaucoup d'optimisations à nos fonctions au cours du développement, dont l'impact est plus faible et n'a pas été forcément mesuré. La

plupart du temps, il s'agissait d'incrémenter les boucles avec `++i` au lieu de `i++` par exemple. On peut aussi citer le fait d'ajouter une couleur en paramètre aux fonctions qui analysent toutes les cases de l'échiquier, comme `echec_et_mat`, `verifier_echec` et `liste_moves`, on peut ainsi stopper l'exécution dès qu'on a inspecté toutes les pièces de la couleur. Une autre optimisation consiste à commencer en partant du haut de l'échiquier (indice 0) lorsque la couleur qu'on analyse est noire, et du bas (indice 63) si la couleur est blanche. Nous gagnons ainsi du temps car les pièces blanches se trouvent en général dans la partie inférieure de l'échiquier, et les noires dans la partie supérieure. Comme expliqué dans la partie 2.1.1, nous utilisons un maximum de variables de type `uint8_t` et `int8_t`, mais nous avons remplacé ces types par `uint_fast8_t` et `int_fast8_t`. Ces types correspondent à des entiers dont la taille peut être supérieure à 8 bits si cela les rend plus rapide. Nous n'avons remarqué aucun changement sur nos machines, mais nous les avons laissé dans le code car ils pourraient faire la différence sur d'autres processeurs.



The figure consists of two screenshots of a terminal window with a dark background and light-colored text. The top screenshot shows the results of a search without major optimizations, and the bottom screenshot shows the results with all major optimizations.

Configuration	Temps de recherche total	Pieces Blanches	Pieces Noires
Sans optimisations majeures	94.735988	Roi, Tour, Pion,	Roi, Tour, Pion,
Avec toutes les optimisations majeures	38.331139	Roi, Tour, Pion,	Roi, Tour, Pion,

FIGURE 17 – Temps de recherche total lors d'une partie sans aucune des optimisations majeures, puis avec toutes les optimisations majeures (différents de ceux mentionnés plus hauts car mesurés sur une autre machine, les les optimisations mineures étaient déjà implémentées)

5.4 Fonctions n'ayant pas pu être réalisées

Avec plus de temps, nous aurions aimé ajouter d'avantage de fonctionnalités qui auraient permis à notre algorithme d'atteindre un meilleur niveau. Nous allons les expliquer brièvement dans cette section.

5.4.1 Protocole UCI

Le protocole UCI (pour Universal Chess Interface) est un protocole de communication entre un moteur d'échecs (comme notre programme) et un programme d'échecs avec une interface utilisateur. Nous n'avons pas accordé de temps à l'implémentation de ce protocole dans notre programme car une interface graphique n'était pas notre priorité, ni le sujet du projet. Cependant, nous avons découvert assez tard dans le développement qu'il aurait pu être très utile pour le niveau de notre IA d'implémenter ce protocole. En effet, lors de nos recherches de méthodes pour améliorer notre IA en ouverture et fin de partie, nous avons trouvé plusieurs bases de données contenant énormément de positions et permettant de déterminer le meilleur coup possible sur une profondeur bien plus grande que 4 ou 5. Il existe des programmes permettant d'implémenter ces bases de données directement dans un moteur en passant par le protocole UCI, comme le programme Polyglot.

Le fait que le protocole UCI soit standardisé signifie qu'il existe de nombreux outils qui auraient pu nous faire gagner beaucoup de temps, et procurer un résultat plus satisfaisant que notre table de hachage, mais également nous fournir d'avantages d'informations de débogage et de statistiques sur notre programme.

5.4.2 Tri des mouvements

Dans notre programme, les liste des pièces pouvant bouger, ainsi que les listes des mouvements possibles d'une pièce (retournées respectivement par les fonctions "liste_moves" et "get_legal_all") stockent leurs valeurs dans un ordre qui n'est pas cohérent. Dès qu'une des fonctions trouve un résultat concluant, elle l'ajoute en tête de liste. Avec plus de temps, nous aurions voulu nous concentrer d'avantage sur ce point, car l'ordre dans lequel la fonction "minimax" évalue chaque mouvement de chaque pièce dépend de l'ordre dans lequel ces éléments sont stockés dans la liste. Or, avec l'élagage alpha beta, plus on trouve le meilleur coup possible tôt, moins on a besoin d'explorer l'arbre, ce qui permet un gain de temps important. Il est donc recommandé d'évaluer tous les déplacements possibles en commençant par ceux étant les plus susceptibles d'engendrer un bon score, comme les captures, ou les déplacements des pièces se trouvant au centre du plateau.

Il existe des méthodes pour trier efficacement les listes de mouvements comme la "Killer Heuristic", mais leur implémentation dans notre jeu aurait nécessité trop de modifications à la façon dont les déplacements sont notés et générés, et nous n'avons pas trouvé assez de temps à accorder à cette technique.

5.4.3 Approfondissement itératif

L'approfondissement itératif est une autre technique d'optimisation que nous aurions voulu ajouter à notre programme. La méthode peut paraître contre-intuitive au premier abord mais elle permet bien de gagner du temps d'exécution si elle est couplée avec des techniques complexes de tri des mouvements et d'un moyen de stocker les évaluations, comme un table de hachage.

Le principe est le suivant : au lieu d'effectuer une recherche minimax avec une profondeur donnée fixe (par exemple 4), on effectue une première recherche avec une profondeur de 1, puis une seconde avec une profondeur de 2, et ainsi de suite jusqu'à ce que le temps de recherche maximal (constante définie par le programmeur) soit atteint. Une fois le temps écoulé, l'algorithme retourne le résultat de la dernière recherche terminée. Cette méthode était trop complexe à implémenter pour nous car elle nécessitait l'utilisation d'une table de hachage, qui nous a pris du temps à implémenter, mais également de techniques avancées de tri des mouvements qui nous auraient requis de réécrire la plusieurs fonctions fondamentales de notre jeu.

6 Répartition du travail

L'utilisation d'un repository Github a rendu le travail à distance plus simple et nous a permis d'avancer indépendamment l'un de l'autre en fonctions de nos disponibilités. Cette organisation nous a rendus plus efficace que de travailler ensemble, que ce soit pour coder, s'informer ou discuter. Nous communiquons régulièrement pour nous partager les résultats de nos recherches et nos nouvelles idées.

L'inconvénient principal de cette approche est que nous ne pouvions pas coder en

même temps, au risque de créer des conflits de version au moment de publier le code sur Github. Pour résoudre ce problème, nous avons créé plusieurs branches dans notre repository. Les heures de TD et de CM nous ont cependant été utiles pour discuter de la direction générale du projet et pour poser des questions.

La répartition des tâches s’est fait assez naturellement dans notre binôme, nous nous impliquions chacun dans tous les domaines du projet et la quantité de travail fournie était équitable. Nous n’avons donc jamais vraiment pris la peine de nous ”organiser” de manière précise.

Il est difficile de quantifier le travail fourni par chacun, la quantité exacte n’est sûrement pas de 50% chacun, mais elle reste suffisamment équitable à nos yeux. Nous percevons ce projet comme un effort collaboratif qui n’aurait pas du tout eu la même tournure sans l’implication des deux membres du groupe.

7 Difficultés rencontrées

7.1 Difficultés rencontrées par Naim Chefirat

Pour Naïm, la plus grande difficulté à surmonter pour ce projet a été de se projeter dans l’avenir lorsqu’il s’agissait de faire un choix. Par exemple lors de la création d’une nouvelle fonction, en plus de réfléchir à comment l’intégrer au reste du programme, il était difficile de penser à la suite pour savoir comment faire en sorte qu’elle ne nous empiète pas par la suite. Cet effort s’est avéré assez difficile car nous avons commencé le développement du jeu très vite, alors que nous étions encore novices sur le sujet de la programmation de jeux incorporant une IA. Il était alors parfois difficile de prendre des décisions sur la façon dont les choses devaient être faites.

Le même problème s’est manifesté lorsqu’il était question de choisir parmi plusieurs fonctionnalités et optimisations à implémenter, car nous savions que le temps était compté et que nous ne pourrions pas nous permettre de tout faire. Par exemple, après avoir implémenté l’algorithme minimax et l’élitage alpha beta, était-ce plus rentable de consacrer notre temps à jouer contre l’IA pour cerner ses défauts, ou bien à chercher à mettre en place des optimisations comme la table de hachage ou au tri des listes de mouvements ? Le fait que nous ne maîtrisions pas ces sujets car nous avions seulement lu des articles dessus n’a pas rendu le processus de décision plus facile.

7.2 Difficultés rencontrées par Badr Agrad

Pour Badr, la difficulté résidait dans la compréhension de certains algorithmes. Cet aspect englobe de nombreux points notamment la manière d’implémenter les algorithmes, selon notre façon de coder. Ayant un raisonnement assez empirique, il fut nécessaire pour lui de tester énormément d’idées, peu importe leur pertinence, pour pouvoir continuer à coder. Il était clairement complexe de prendre des décisions concises tant le nombre de possibilités était large. Par exemple, Badr avait décidé de coder tout le jeu en définissant l’échiquier comme une matrice, ne comprenant pas l’intérêt d’utiliser un tableau. C’est en testant cette possibilité qu’il se rendit compte qu’un tableau était plus simple à gérer et à manipuler. Néanmoins, cela lui a permis d’apprendre à aborder différemment les problématiques. Le questionnement n’était pas de savoir comment réaliser, mais plutôt de trouver quelles étaient les solutions les plus optimales pour répondre à nos besoins. De plus, cela impliqua un long travail de recherche, le

jeu d'échec étant un vaste sujet d'étude. De surcroît, la projection fut assez complexe car nous ne savions pas comment l'IA allait être implémentée et quels allaient être les nouveaux enjeux que nous allions rencontrer.

8 Bilan

8.1 Le Bilan de Naim Chefirat

Le développement de ce projet au cours d'un semestre entier a permis à Naïm d'améliorer ses compétences en algorithmique, en ayant rédigé plusieurs des fonctions les plus complexes comme "minimax", "vérifier_echec", ou encore l'implémentation de la table de hachage.

De plus, l'élaboration d'un projet de cette envergure l'a forcé à développer de nouvelles compétences essentielles comme la mise en place d'un environnement de travail stable et partagé, avec Git, Github et Docker, la gestion d'un programme composé de plusieurs fichiers dont un fichier d'en-tête et une makefile, et de nombreuses techniques d'optimisations, notamment l'utilisation adéquate de types des données présents dans la bibliothèque "stdint.h", ou encore les optimisations de compilateur

C'est ce dernier type de compétences que Naïm est le plus satisfait d'avoir acquis, car elles ne viennent pas simplement en s'exerçant comme on le ferait en TD, mais sont le fruit d'un travail de tout un semestre pour répondre à une seule problématique majeure.

8.2 Le Bilan de Badr Agrad

Ce projet a permis à Badr d'améliorer ses compétences de développeur. En effet, cela enveloppe de nombreux aspects comme les compétences algorithmique, l'utilisation et la découverte de nouveaux outils ou encore le travail de groupe. Étant novice au jeu d'échec, le projet fut d'autant plus stimulant au vu du travail de compréhension et de recherche derrière chaque aspect du projet allant de la manière de bouger une pièce jusqu'à l'optimisation de Minimax pour notre jeu. Ayant codé des fonctions et des structures essentielles comme "FEN" ou encore l'implémentation de l'élagage alpha beta, Badr se confronta aux difficultés de devoir construire un travail cohérent et compréhensible par l'ensemble du groupe. D'autre part, cela lui a permis d'apprendre à utiliser des applications comme Git ou encore Docker, essentielles au développement d'un projet aussi complexe. Enfin, les différentes contraintes qu'il rencontra notamment les problèmes d'architecture processeur ou encore de compatibilité lui ont permis d'en apprendre davantage sur les environnements et systèmes d'exploitations. Le développement du projet lui a ainsi offert une progression assez globale de ses compétences. L'aspect qu'il a préféré fut la découverte des méthodes d'optimisation et des algorithmes d'intelligences artificielles.

8.3 Conclusion et perspectives

Pour conclure, travailler sur ce projet durant l'ensemble du semestre nous a permis d'améliorer grandement nos compétences algorithmiques, de travail en groupe et de découvrir de nouvelles technologies. Nous sommes particulièrement satisfait de nos progrès, mais aussi du résultat final. Finalement, notre programme a su battre une intelligence artificielle du célèbre site "chess.com" ayant un niveau de 1200 Elo. L'objectif d'atteindre un niveau supérieur à 1500 Elo n'est donc pas pleinement rempli. Nous sommes cependant tout à fait satisfait de ce résultat. Néanmoins, notre algorithme présente quelques défauts durant l'ouverture et en fin de partie. En effet, l'interface de chess.com a jugé certains mouvements exécutés, durant ces périodes de la partie, assez impertinents. De plus, l'optimisation par la table de hachage, permettant d'éviter de recalculer des scores de positions déjà évaluées, n'a pas eu énormément d'impact sur les performances de notre programme. En somme, tous ces éléments nous ont donné l'occasion de développer notre esprit d'ingénieur en nous poussant à être créatif afin de trouver les meilleures solutions possibles. De ce fait, nous souhaitons toujours développer le projet de notre côté au vu du nombre d'algorithmes d'optimisation existant. Au cours du semestre, nous avons découvert les intelligences artificielles sous un angle que nous ne connaissions pas. Entre réseau de neurones et arbre binaire de recherche, nous avons su constater le panel large de méthodes informatiques visant à reproduire les comportements humains. L'intelligence artificielle ne cesse de prendre de la place dans notre société. En effet, le monde tend à devenir de plus en plus automatisé et le rôle des ingénieurs est en jeu. Nous souhaitons, à l'avenir, être acteur de cette transition technologique en nous investissant dans ce domaine d'étude.

A Sources

A.1 Webographie

1. Chess Programming Wiki
 - Table de transposition
 - Élagage alpha beta
 - Hachage de Zobrist
 - Protocole UCI
 - Approfondissement itératif
 - Polyglot
2. Vidéo pour comprendre l'algorithme Minimax et l'élagage alpha beta
3. Améliorations et optimisations de l'algorithme
4. Série de vidéos sur le développement d'un jeu d'échecs avec IA en C

B Manuel Utilisateur

B.1 Mise en place de l'environnement et compilation

Notre projet a été conçu pour fonctionner sur linux, et pour être compilé avec GCC 10.2.1. Bien qu'il soit sûrement possible de le compiler avec d'autres versions de GCC ou même avec Clang, nous ne pouvons pas garantir que le programme fonctionnera correctement. Pour répondre à ce problème, vous pouvez utiliser la Dockerfile présente dans le dossier du projet pour créer un conteneur Docker pour disposer d'un environnement identique à celui dans lequel nous avons développé et testé le code. Pour ce faire, après avoir installé Docker sur votre machine, placez-vous dans le répertoire du projet et saisissez la commande suivante :

```
$ docker build -t image .
```

Cela aura pour effet de créer une nouvelle image Debian (appelée "image") à partir de la Dockerfile. Vous pouvez maintenant créer le conteneur et l'exécuter avec cette commande (cela donnera le nom de "conteneur" au nouveau conteneur) :

```
$ docker run -it --name conteneur image /bin/bash"
```

La prochaine fois que vous voudrez utiliser le conteneur, vous pourrez utiliser ces deux commandes :

```
$ docker start conteneur  
$ docker exec -it conteneur /bin/bash
```

Lorsque vous voulez quitter le conteneur, utilisez ces deux commandes :

```
$ exit  
$ docker stop conteneur
```

Vous vous trouvez à présent dans le dossier "/"projet" du conteneur, qui contient une copie de tout ce qui se trouvait dans le répertoire du projet. Vous pouvez à présent compiler les fichiers avec la makefile :

Cela créera un fichier exécutable nommé "main".

B.2 Utilisation-type

Le programme s'ouvre sur un menu qui vous permet de configurer la partie. Vous choisir de jouer contre vous-même, contre notre IA, ou d'assister à une partie IA contre IA. Si l'un des joueurs est une IA, il vous sera demandé de choisir une profondeur, nous vous recommandons 4 pour l'expérience la plus fluide, et 5 si vous voulez voir l'IA à son meilleur niveau (la durée de recherche peut être plus ou moins longue en fonction de la configuration de l'échiquier, entre 5 et 40 secondes sur nos machines). Les informations de debug affichent diverses statistiques à chaque tour comme le score actuel, le nombre de tours écoulés, etc. Elles peuvent être utiles pour voir quel déplacement l'ia vient d'effectuer, ou pour connaître son temps précis de recherche.

B.3 Utiliser la table de hachage

Comme expliqué dans la partie 5.2.2 de ce document, nous avons décidé de ne pas utiliser la table de hachage que nous avons développé pour notre IA. Il existe cependant une fonction appelée "minimax_ht" présente dans le fichier "hashtable.c" qui utilise ses fonctionnalités.

Si vous souhaitez utiliser la table de hachage en lecture et écriture, vous devez commenter l'appel à "fill_from_file" dans la fonction main (fichier "main.c" et remplacer les appels à "minimax" aux lignes 1601 et 1609 du fichier "chess.c" par "minimax_ht", en ajoutant le pointeur "fp" aux paramètres de la fonction.

Si vous souhaitez utiliser la table en lecture seule à partir du fichier "hashtable.txt", décommentez l'appel à "fill_from_file" dans la fonction main, et commentez les appels à "add_entry" aux lignes 270 et 306 de "hashtable.c". Les valeurs présentes dans le fichier texte ont été générées lors de parties IA contre IA avec l'appel à "fprintf" non commenté dans "minimax_ht".

B.4 Erreurs courantes

Lorsque vous écrivez un mot ou certains caractères alors que le programme vous demande un nombre, une boucle infinie de messages d'erreurs va alors avoir lieu. Le seul moyen de stopper ce bug est d'arrêter l'exécution du programme avec ctrl+C.