Kasparov Chess Engine v1 : Un moteur de jeu d’échecs

Par P. Beuran.

Table des matières

I. Introduction :

Le jeu d'échec est un jeu de plateau au tour par tour, opposant deux joueurs ayant chacun un ensemble de pièces sur ce plateau, appelé échiquier, avec pour but étant de mettre en « échec et mat » le roi adverse. Ce jeu est une icône des jeux de stratégie, tant par son coté historique, ayant traversé les âges du Vème siècle jusqu'à nos jours, que par son coté compétitif, faisant toujours l'objet de nombreux tournois à différent niveaux et des « Grands Maîtres » qui sont mondialement connus (Kasparov, Fisher etc…).

Les échecs ont aussi été un intérêt pour le domaine de l'informatique, plus particulièrement de l'intelligence artificielle. En effet, le jeu d'échec possède des règles assez simples pour être représenter facilement dans un programme informatique, tout en possédant une richesse en termes de jeu et de stratégie, dû aux nombreuses combinaisons de ces règles utilisées pour être amené à une victoire. De nombreuses entreprises et organismes se sont tentés à la création d'algorithmes pouvant jouer aux échecs, et tenter de rivaliser les plus grands joueurs de leur époque. Parmi ces programmes notables, Deep Blue, Stockfish ou Alpha Zero peuvent être cités.

II. Règles :

Le jeu d'échec se joue entre deux joueurs, respectivement le camp des « blancs » et des « noirs » (nommées ainsi dû à la couleur des pièces de chaque camp), sur un plateau de 8x8 cases. Le jeu se déroule au tour-par-tour, chaque joueur devant déplacer une de leurs pièces selon des règles strictes dépendantes du type de pièce, de la position de la pièce et de la position des autres pièces dans l'état donné d'une partie. L'ensemble de ses actions sont appelées mouvements légaux.

Ces différentes pièces et leurs règles associées sont :

- Les pions ♙ / ♟ :
    - Peuvent se déplacer d'une case en avant si la case en question est vide.
    - Peuvent se déplacer d'une case sur leurs diagonales avants si cette case est occupée par une pièce ennemie.
    - Peuvent se déplacer de 2 cases vers l'avant s'ils sont à leur positions d'origines (voir fig 1 pour les positions d'origines).
    - Peuvent être interchanger par une autre pièce de même couleur autre qu'un autre pion et un roi si un de ces pions atteint la dernière ligne du camp adverse (c'est-à-dire la ligne la plus éloigné du point du vue du joueur possédant ce pion).

- Les cavaliers ♘ / ♞ :
    - Peuvent se déplacer en « L », c'est-à-dire sur une des cases résultantes des combinaisons de 2 cases verticalement et d'une case horizontalement, et de 2 cases horizontalement et d'une verticalement, si la case en question est vide ou occupée par une pièce ennemie.

- Les fous ♗/♝ :
    - Peuvent se déplacer sur une des cases situées sur leurs diagonales, tant que cette case est vide ou occupée par une pièce ennemie, et que les cases comprises entre la case de départ et d'arrivée exclues ne contiennent aucune pièce.

- Les tours ♖/♜ :
    - Peuvent se déplacer sur une des cases situées sur leurs horizontales ou leurs verticales, tant que cette case est vide ou occupée par une pièce ennemie, et que les cases comprises entre la case de départ et d'arrivée exclues ne contiennent aucune pièce.

- Les reines ♕/♛ :
    - Peuvent se déplacer sur une des cases situées sur leurs horizontales, verticales ou diagonales, tant que cette case est vide ou occupée par une pièce ennemie et que les cases comprises entre la case de départ et d'arrivée exclues ne contiennent aucune pièce.

- Les rois ♔/♚ :
    - Peuvent se déplacer sur une des cases situées directement sur leurs horizontales, verticales ou diagonales (c'est-à-dire sur les cases qui leurs sont directement adjacentes.)
    - Peuvent effectuer 2 mouvements spéciaux :
        - Le petit roque : si le roi et la tour la plus proche du roi sont à leur positions d'origine respectives, et que les cases entre les deux pièces sont vides, alors le roi peut bouger sur la case d'origine du cavalier le plus proche de lui et la tour en question sur la case d'origine du fou le plus proche de la case d'origine du roi.
        - Le grand roque : si le roi et la tour la plus éloignée du roi sont à leurs positions d'origine respectives, et que les cases entre les deux pièces sont vides, alors le roi peut bouger sur la case d'origine du fou le plus éloignée de lui, et la tour en question sur la case d'origine de la reine.

On parlera de « saut » pour les mouvements des cavaliers (du fait qu'ils ignorent les pièces se trouvant sur leur trajectoire) et de « coulissement » pour les mouvements des fous, des tours et des reines (du fait qu'ils doivent prendre en compte les pièces se trouvant sur leur trajectoire.)

Tous ces mouvements (appelées mouvement pseudo-légaux) sont assujettis à deux règles communes avant d'être considéré comme légaux :

- Si la case d'arrivée d'un ou plusieurs mouvements pseudo-légaux contient le roi ennemi, ce ou ces mouvements ne sera pas considéré comme légaux, mais le roi du joueur ennemi sera considéré en échec.
- Un mouvement légal d'un joueur ne peut induire son roi en échec.
- Par extension, si le roi d'un joueur est en échec, les seuls mouvements légaux seront ceux lui permettant de sortir son roi de cette situation.

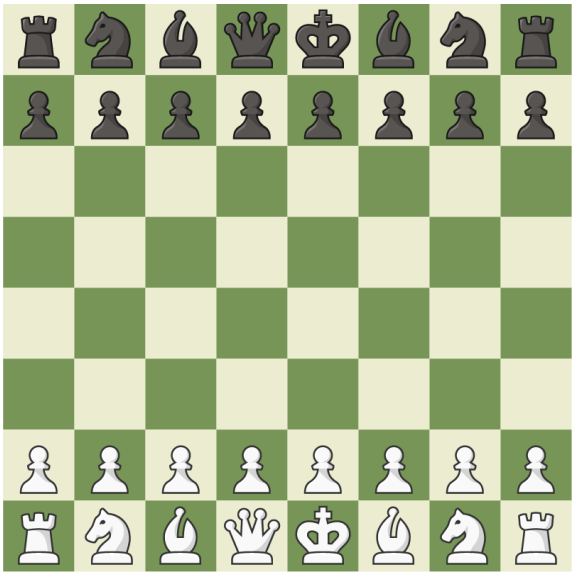
Dans le cas où un joueur se retrouve sans aucun mouvement légal possible :

- Si son roi est en échec, alors la partie est considérée terminée par échec et mat et le joueur ennemi ressort vainqueur de la partie.
- Sinon, la partie est considéré comme nulle (c'est-à-dire sans vainqueur.)

A cela s'ajoute deux règles de fin de parties :

- Si sur 50 mouvements respectifs des 2 camps, il n'y a eu ni captures de pièces, ni de déplacements de pions, la partie est considéré comme nulle.
- Si une combinaison de positions de pièces se répète 3 fois, la partie est considérée comme nulle.

Toute partie d'échec débute avec la même disposition de pièces sur le plateau, tel que montré sur la figure ci-dessous, et le joueur contrôlant les pièces blanches est toujours le premier à jouer. Tant qu'une des conditions de fin de partie n'a pas été rencontrée, le tour de jeu alterne avec le joueur ennemi du joueur actuel.



(fig. 1 : disposition des pièces en début de partie.)

III. Objectifs :

Le « Kasparov Chess Engine », ou KCE, a pour buts de :

- Recréer le jeu d'échec et toutes ces règles.
- Le rendre adaptable pour que tout types de joueurs puissent l'utiliser (que ce soit humain ou artificielles)
- Développeur plusieurs types d'intelligence artificielles faisant office de joueur.
- Créer des méthodes d'apprentissage pour les intelligences artificielles qui en requièrent.
- Analyser les résultats de parties pour en extraire des données.
- Optimiser le déroulement des parties et le traitement des intelligence artificielles pour accélérer les processus de décisions et d'apprentissage.

IV. Aspect technique :

KCE a été programmé en C++ 17, sous l'environnement de développement Visual Studio. Il est compatible pour les systèmes d'exploitation Windows, Mac OS et Linux. Il possède deux exécutables :

- Une version console, ne permettant qu'à des IA de jouer, et ainsi utiliser principalement pour l'analyse et l'apprentissage des différentes IA.
- Une version graphique, servant d'interface de jeu pour un humain contre un autre humain ou une IA, utilisé principalement pour jouer ou tester les capacités des différentes IA.

A cela s'ajoute différents fichiers, contenant :

- Les analyses produites par la version console, si demandées.
- Les fichiers nécessaires à certaines IA, contenant leurs apprentissages de bases.

Le choix du C++ en tant que langage de programmation s'est fait pour 2 raisons :

- La possibilité d'utiliser des opérateurs binaires sur des entiers non-signé de 64 bits.
- La rapidité d'exécution d'un programme en C++, étant un langage plus proche des langages de bas niveau que peut être le Java ou le C#, tout en gardant des notions de langages de haut niveau tel que les classes, ce que le C, son prédécesseur, ne possède pas.
- L'optimisation des fonctions les plus simples par le compilateur, supprimant le prologue et l'épilogue de ces fonctions.

Du fait de l'utilisation d'entier en 64 bits et de différentes méthodes spécifiques au processeur 64 bits, les 2 programmes ci-dessus ne peuvent fonctionner que sur des systèmes 64-bits.

V. Aspect algorithmique et théorique :

a. Représentation d'une partie d'échecs et ses règles :

NB : Ce chapitre se basant énormément sur les opérateurs unaires et binaires sur bits, voir les tables de vérité présentées en annexe 1 pour le résultat de chacune de ces opérations.

Un jeu d'échec est tout à fait possible de représenter de la manière la plus simple possible, c'est-à-dire par un tableau à 2 dimensions de 8x8 cases, contenant chacune l'état de la case (vide ou occupé par une pièce d'un certain type et d'une certaine couleur). Les règles du jeu d'échecs peuvent être ainsi codé de manière très simple, par des combinaisons de conditions pour le calcul des déplacements de pièces, des conditions d'échecs et de fin de partie.

Cependant cette approche est lourde en termes de structure de données, et rendra l'exécution d'une partie d'échec lente, ce qui est indésirable, contenu du fait qu'il sera nécessaire pour certaines IA de jouer des centaines de milliers de tours ou parties, voir des millions, afin de réaliser son apprentissage dans le cas des IA à apprentissage, ou de prendre une décision pour les IA de recherche. De plus, cette approche se verra souvent exécuter de façon monolithique, et le gain de temps pouvant être apporté par la division des tâches par multithreading serait minime.

Une approche intéressante pour optimiser le traitement d'une partie est d'utiliser des structures de données très simples pour représenter la position des pièces, ainsi que des opérations primitives sur ces structures, afin de faciliter le traitement des instructions pour le processeur.

Sachant qu'un échiquier est composé de 8x8 cases, c'est-à-dire 64 cases, la position d'une pièce peut être représenté par un entier de 64 bits (voir fig.2), lu de manière binaire tel que 0 indique une case vide et 1 la position de la pièce. On peut ainsi généraliser la position des pièces d'une certaine couleur ou d'un certain type par un entier de la même manière, et ainsi représenter la position de toutes les pièces par un tableau de 8 entiers en 64 bits tel que :

0. Entier représentant la position des pièces blanches
1. Entier représentant la position des pièces noires
2. Entier représentant la position des pions
3. Entier représentant la position des cavaliers
4. Entier représentant la position des fous
5. Entier représentant la position des tours
6. Entier représentant la position des reines
7. Entier représentant la position des rois

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

(fig 2. Représentation d'un échiquier en entier 64 bits)

La position de chaque type de pièce d'une certaine couleur peut être obtenue facilement en utilisant l'opérateur binaire ET(&) (ex : la position des pions noirs équivaut à dire tab[NOIR] & tab[PION]).

Il est possible de récupérer une pièce en particulier en connaissant sa position en terme d'index, ou de vérifier son existence à cet index, grâce à l'opérateur ET(&) et SHIFT(<<) (ex : l'existence du fou blanc à la position X peut être vérifié en effectuant l'opération tab[BLANC] & tab[FOU] & (1 << X), où (1 << X) équivaut à 2^X).

Pour récupérer l'index de la position d'un certain type de pièce et/ou d'une certaine couleur, on utilisera l'instruction de processeur BIT_SCAN_FORWARD(BSF), qui scanne un entier et renvoie l'index du premier bit égal à 1 en partant du bit le moins significatif.

Pour soustraire une pièce en connaissant son index, on utilisera les opérateurs XOR(^) et SHIFT(<<) (ex : la soustraction du cavalier se trouvant à l'index X se fait en effectuant l'opération tab[CAVALIER] = tab[CAVALIER] ^ (1 << X)) (NB : cela peut aussi se réaliser avec les opérateurs ET(&) et NON(~) au lieu de XOR(^) tel que tab[CAVALIER] = tab[CAVALIER] & ~(1 << index)).

En prenant comme référence la fig.2, on peut déterminer que pour un index X donné :

- La case en haut de cet index a pour valeur (1 << X) << 8.
- La case en bas de cet index a pour valeur (1 << X) >> 8
- La case à droite de cet index a pour valeur (1 << X) << 1
- La case à gauche de cet index a pour valeur (1 << X) >> 1

Dans le cas des bordures de l'échiquier, il est impossible pour une pièce de passer de la bordure supérieure à inférieure, et inversement, l'opération SHIFT(<<) rendant le résultat nul si la capacité pour contenir le résultat dépasse 64 bits, de même pour l'opération NSHIFT(>>) si la capacité pour contenir le résultat est égale à 0 (elle ne peut être inférieur).

Ce n'est pas le cas pour les bordures de droite et de gauche, car si visuellement cette opération semble impossible, elle reste techniquement possible, la machine ne considérant pas l'entier utilisé comme un tableau à 2 dimensions. Pour résoudre ce problème, une comparaison supplémentaire est nécessaire pour tout déplacement sur l'horizontale, vérifiant que le déplacement vers la droite ou la gauche n'a pas pour valeur respective une case se situant la bordure gauche ou la bordure droite. (ex : pour un roi avec une position X voulant se déplacer vers la droite, on vérifie la valeur ((1 << X) << 1) & ~bordure_gauche)

Il est ainsi possible de définir les mouvements pour les pions et les rois de tel sorte, les mouvements sur les diagonales n'étant qu'une combinaison de déplacements horizontaux et verticaux. Pour les cavaliers, les mouvements contenant 2 déplacements sur une direction horizontale devront vérifier s'il n'atteigne pas la bordure et l'avant dernière ligne horizontale opposée à la direction horizontale choisie.

Pour les fous, les tours, et les reines, leurs déplacements sont plus complexes à réaliser sur des bits, sachant leurs déplacements par coulissement (voir Règles) prend en compte l'occupation de l'échiquier même pour les mouvements théoriques. Il est possible de déterminer ce mouvement pour un déplacement horizontal positif (c'est-à-dire en direction des bits de poids les plus forts, donc vers la droite) grâce à la technique du « o^(o – 2r) » où :

- « o » représente l'occupation des pièces sur l'échiquier.
- « r » représente la position de la pièce sur l'échiquier.

Cette technique utilise la propriété de la soustraction sur les bits pour représenter ce mouvement de coulissement, de tel sorte :

Soit o = 01000100, r = 01000000.

On a :

- o – r = 00000100 : on enlève le bit de la position de la pièce.
- o – 2r = 01111000 : la propriété de la soustraction sur les bits fait que tous les bits compris entre la position r compris et la position de la première pièce bloquante deviennent égale à 1, et la position de la première pièce bloquante devient égale à 0 (voir annexe 1).
- o^(o-2r) = 00111100 : du fait de la propriété de l'opération XOR(^), on soustrait toutes les positions des pièces encore présentes (dans notre cas, uniquement r était présent) et on ajoute la pièce bloquante, donnant ainsi le mouvement théorique de la pièce.

Il est possible d'utiliser cette technique pour les mouvements coulissant en colonne, en diagonale et anti-diagonal positif aussi, par cette formule : o^((o&m) – 2r)&m, où m est le masque de la colonne, la diagonale ou l'anti-diagonale contenant la position de la pièce r.

NB : Il faut faire la différence entre la diagonale et l'anti-diagonale dans ce cas précis, la diagonale étant l'addition de deux mouvements de même signe (ex : la diagonale positive est l'addition entre le mouvement menant vers le haut et vers la droite) et l'anti-diagonale étant l'addition de mouvements de signes opposées (ex : l'anti-diagonal positive est l'addition entre le mouvement menant vers le haut et vers la gauche).

Cette technique doit aussi être appliqué pour les mouvements coulissant horizontaux, car comme la soustraction de 2r active tous les bits jusqu'à la position de la première pièce bloquante, s'il n'existe aucune pièce bloquante sur la ligne de la pièce, la soustraction continue d'activer les bits suivants qui sont en dehors de la ligne tant que cette condition n'est pas respectée.

Pour les mouvements horizontaux négatifs (c'est-à-dire vers la gauche), on utilise la même technique, en inversant les bits (c'est à dire que si X est l'index d'un bit donné, son index inversé sera de 63 – X) de o, r et du résultat global, nous donnant cette formule : (o'^(o'-2r'))' où ' est l'opération d'inversion de bits. Si on combine le calcul des mouvements horizontaux positifs et négatifs, on obtient ainsi : o^(o – 2r) ^ (o'^(o'-2r))' ⇔ o^(o-2r) ^ o^(o'-2r')' ⇔ (o-2r) ^ (o'-2r')', car X'' = X et X^X = 0. Cette technique s'explique par le fait que la soustraction fonctionne de manière positive (c'est-à-dire en activant les bits les plus forts à partir de l'index X jusqu'à atteindre le premier bit activé, qu'il désactive et s'arrête ici) même dans la situation inversée, ce qui fait qu'en la ré-inversant, on obtient une soustraction « négative » dans une situation normale, et donc le mouvement de coulissement vers la gauche. En réintégrant le masque de la ligne contenant la position de la pièce, on obtient : ((o&m)-2r) ^ ((o&m)'-2r')' & m (sachant que X & X = X).

Pour les mouvements verticaux, diagonaux et anti-diagonaux négatifs, cette même technique est utilisée. Cependant, pour des raisons techniques, l'opération d'inversion des bits est remplacée par l'opération d'inversion d'octets (c'est-à-dire que si X est l'index d'un octet donné, 7–X est son index inversé), étant plus rapide du fait qu'il est une instruction natif du processeur, et donnant le même résultat que si l'opération d'inversions de bit est utilisé.

NB : Ce remplacement ne fonctionne pas pour les déplacements horizontaux, étant donné l'inversion d'octet équivaut à une inversion par symétrie horizontale, et donc n'inverse pas les directions horizontales, ce qui ne permet pas l'obtention de cette soustraction « négative » recherché. Cela ne fonctionne ainsi pour les déplacements impliquant une composante verticale, car cette soustraction négative n'est utilisée que pour inverser le sens de coulissement horizontal et vertical, ce qui est inutile dans le cas des colonnes, des diagonales et anti-diagonales, qui n'a besoin d'inverser le sens de la soustraction que de manière verticale, ignorant complétement les pièces pouvant se trouver sur la même ligne que le référentiel donné.

La définition des mouvements pseudo-légaux se fait tout simplement en effectuant la conjonction entre les mouvements théoriques et l'union des cellules vides et occupées par les pièces ennemies, excepté les rois.

La définition des mouvements légaux se fait en exécutant chaque mouvement pseudo-légal et en vérifiant qu'il n'y a pas échec sur le roi du joueur actuel, c'est-à-dire que la conjoncture entre les mouvements théoriques ennemi après l'exécution du mouvement pseudo légal et la position du roi est vide (c'est-à-dire égale à 0).

Il est possible de calculer les mouvements théoriques et pseudo-légaux de toutes les pièces au cas par cas, ce qui est nécessaire pour définir les mouvements légaux, utilisées ensuite par les joueurs pour déplacer leurs pièces, du fait de la nécessité de connaître la case d'origine d'une pièce en plus de sa case d'arrivée. Il est aussi possible de calculer les mouvements théoriques et pseudo-légaux des pions et des cavaliers en même temps, utilisées pour les mouvements théoriques dans le cas de la vérification de l'échec d'un roi, dont la question de connaître la case de départ d'une pièce inutile.

L'avantage d'utilisé ces techniques est l'exploitation des opérations sur bits, dont, pour un entier de 64 bits, effectue une opération donnée sur les 64 bits en parallèle ; et des instructions natives du processeur (hormis l'inversion de bits), permettant ainsi une vitesse d'exécution optimale en ce qui concerne la recherche des mouvements et des échecs

Le processus d'exécution d'une partie se fait de manière plus classique, l'optimisation de ce dernier étant non-seulement complexe mais aussi futile, la recherche de mouvements étant le facteur le plus important dans l'optimisation d'une partie. Il en va de même pour la recherche de fin de partie. Tous ces procédés suivent ce qui est à été décrit dans les règles plus haut.

Les promotions, qui sont simplement le changement d'un type de pièce à un autre, sont gérées de la même manière qu'indiquées dans les règles. Cependant, pour simplifier l'exécution des mouvements, tous mouvements qui ne conduit normalement pas à une promotion, conduira à la promotion de la pièce d'un type donné à ce même type.

Du fait de la représentation différente de l'échiquier dans ce projet, l'exécution des mouvements se fait en désactivant le bit se trouvant à l'index de la case de départ pour la couleur et le type de pièce donnée, l'index de la case d'arrivée pour toutes les valeurs du tableau, et en activant le bit de l'index de la case d'arrivée pour la couleur de la pièce et le type de pièce promu donnée.

b. IA

Avant de pouvoir parler des IA utilisées, il faut définir certains termes pour la partie suivante :

- Un état de jeu représente toutes les informations connues à un moment donné d'une partie de ce jeu. Dans le cas du jeu d'échec, un état est représenté par :
    - La position des pièces sur l'échiquier.
    - Le compteur de mouvements non-déterminants.
    - Le compteur de répétitions des positions sur l'échiquier.

  Cependant, ces deux derniers éléments ne rajoutent que de la complexité au domaine des états, sans pour autant apporter de l'information utile. Ainsi, ces deux éléments seront omis implicitement dans la plupart des IA, réduisant l'état à la position des pièces sur l'échiquier.

- Une action représente la capacité d'agir sur un état pour le modifier et obtenir ainsi un nouvel état de jeu. Dans le cas du jeu d'échec, ces actions sont les mouvements légaux.

- Une récompense est une valeur numérique décrivant l'utilité d'un état donné et permettant d'influencer le processus de décision d'une IA. Cette récompense dans le jeu d'échec peut être différente selon les IA mais une victoire représente toujours la plus grande récompense positive possible, une défaite la plus grande récompense négative, et un nul une récompense nulle.

c. Negamax

Negamax est un algorithme d'IA de recherche, dérivé de l'algorithme Min-Max. L'algorithme Min-Max est un algorithme s'appliquant uniquement pour les jeux à deux joueurs, à somme nulle (dans la théorie des jeux, un jeu à somme nulle est un jeu où la somme des scores de tous les joueurs est égale à 0) et à informations complète (c'est-à-dire que toutes les données concernant le jeu sont connues), et donc pouvant s'appliquer aux échecs.

Le but de l'algorithme Min-Max est de minimiser les pertes maximales d'un joueur lors du choix d'une action. Pour se faire, l'algorithme part de l'état actuel de la partie et effectue une action. Dans cet état suivant il effectue une autre action (dans le rôle du joueur ennemi contenu du fonctionnement du jeu d'échec) et continue ainsi jusqu'à atteindre un état final (c'est-à-dire un état qui ne contient aucune action, dans le cas du jeu d'échec, l'état de fin de partie) ou un état dont la profondeur est égale à la profondeur maximale atteignable (la profondeur d'un état est déterminé par le nombre d'états prédécesseurs par rapport à l'état actuel, ce dernier ayant ainsi une profondeur de 0). Il évalue la valeur de cet état (appelé feuille) grâce à une fonction d'évaluation, qui évalue la récompense pour le joueur de l'état actuel (appelé racine), et renvoie sa valeur à son prédécesseur. Toutes les valeurs des feuilles sont ensuite traitées de façon récursive par l'état prédécesseur (appelé nœud) de cette manière :

- Si le joueur de la racine le même que celui du nœud traité, le nœud traité renvoie à son prédécesseur la valeur maximale parmi les valeurs des états successeurs (c'est-à-dire que le joueur actuel cherche à effectuer l'action maximisant son score). Ce nœud est nommé nœud max.

- Si le joueur de la racine est différent de celui du nœud traité, le nœud traité renvoie la valeur minimale parmi les valeurs des états successeurs (c'est-à-dire que le joueur ennemi cherche à effectuer l'action minimisant le score du joueur actuel). Ce nœud est nommé nœud min.

Arrivé à la racine, l'algorithme déterminera l'action à effectuer en choisissant l'action menant à l'état successeur renvoyant le plus grand score.

Bien qu'efficace et théoriquement capable de résoudre le jeu d'échec (c'est-à-dire déterminer toutes les séries d'actions menant à la fin de partie), il en est pratiquement impossible dû aux grands nombres possibilités du jeu d'échec (il existerait plus de 10^{120} parties d'échecs possibles selon C. Shannon, en partant du fait qu'il existe 1000 combinaisons d'un mouvement d'une pièce blanche suivi d'un mouvement d'une pièce noire en moyenne, et qu'une partie dure en moyenne pour 40 de ces combinaisons). Cela est aussi vérifiable en calculant la complexité de l'algorithme qui, pour un facteur d'embranchement moyen b et une profondeur de recherche d, donne une complexité exponentielle de $O(b^d)$ (à partir de la racine on obtient b nœuds, qui pour chacun d'entre eux ont b nœuds etc…). En prenant des statistiques sur un grand nombre de parties (presque 700.000 parties), on trouve un nombre moyen de 30 mouvements par tour pour un joueur donné et de 40 mouvements par joueur, ce qui donnerai un arbre à explorer de $30^{40*2} \approx 10^{118}$, ce qui se rapproche du nombre de Shannon.

Pour optimiser le traitement de l'algorithme autrement que par l'optimisation de la recherche de mouvements et de l'exécutions d'une partie, on utilise des techniques d'élagages, les plus efficaces étant l'élagage Alpha-Beta, et les tables de transpositions.

L'élagage Alpha-Beta est une première méthode pour chercher à réduire le nombre de nœuds étudiés. Pour cela, deux nouvelles variables sont introduites dans l'algorithme Min-Max :

- Alpha, qui représente le score minimum que le joueur cherchant à maximiser le score est assuré d'avoir. Alpha est initialement négativement infini.
- Beta, qui représente le score maximum que le joueur cherchant à minimiser le score est assuré d'avoir. Beta est initialement positivement infini.

Le but de cet élagage est de créer une fenêtre $[\alpha, \beta]$, initialement large pour représenter le pire score possible pour le joueur actuel dans le cas de α (aussi appelé limite basse), et le meilleur score possible pour le joueur actuel dans le cas de α (aussi appelé limite haute), ce qui revient au pire score possible pour le joueur ennemi, et de la réduire à chaque fois pas de l'algorithme si possible, jusqu'à possiblement obtenir $\alpha \geq \beta$, auquel cas la recherche s'arrête dans l'état actuel, étant face à une contradiction (la valeur minimum du score du joueur cherchant à le maximiser sera supérieur ou égale à la valeur maximal du score du joueur cherchant à le minimiser, ce qui rend inutile toute future recherche), aussi appelé coupure. 2 types de coupures peuvent être distinguer :

- La coupure α, se produisant dans un nœud min lorsque le joueur cherchant à minimiser le score trouve un score inférieur ou égale au score minimal trouvé par le joueur cherchant à maximiser le score, indiquant que quel que ce soit le choix du joueur minimisant, le joueur maximisant choisira toujours l'action qui apportera ce score minimal supérieur déjà trouvé.

- La coupure β, se produisant dans un nœud max lorsque le joueur cherchant à maximiser le score trouve un score supérieur ou égal au score maximal trouvé par le joueur cherchant à minimiser le score, indiquant que quel que ce soit le choix du joueur maximisant, le joueur minimisant choisira toujours l'action qui apportera ce score maximal inférieur déjà trouvé.

L'élagage Alpha-Béta est la technique d'optimisation la plus efficace pour l'algorithme Min-Max. En effet, si on considère un facteur d'embranchement moyen b et une profondeur de recherche d, l'algorithme évaluera ainsi un nombre maximum de nœuds égal à celui de Min-Max sans élagage dans le pire des cas (si les meilleurs de nœuds sont évalués en dernier, ce qui n'occasionne aucune coupure) et de $O\left(b^{\frac{d}{2}}\right) = O\left(\sqrt{b^d}\right)$ si les meilleurs nœuds sont évalués en premier. Cela peut être prouvé en posant $S(d)$ le nombre minimal de nœuds à visiter pour obtenir la valeur exacte d'un nœud à la d-ième profondeur, et $R(d)$ le nombre minimal de nœuds à visiter pour obtenir la valeur limite (α pour les nœuds max, β pour les nœuds min) d'un nœud à la d-ième profondeur.

On obtient ainsi

- $S(d) = S(d-1) + (b-1)\,R(d-1)$, car il suffit de connaître la valeur exacte d'un nœud et la valeur limite de tous les autres nœuds successeurs pour déterminer la valeur exacte du nœud à la profondeur d, si les meilleurs nœuds sont évalués en premier.
- $R(d) = S(d-1)$, car il suffit de connaître la valeur exacte d'un nœud successeur pour déterminer la valeur limite du nœud à la profondeur k
- $S(0) = R(0) = 1$, car il suffit juste de connaître la valeur d'un et un seul nœud pour déterminer ces deux valeurs, c'est-à-dire la valeur de nœud à la profondeur 0 (donc lui-même), à travers la fonction d'évaluation.
- $S(d) = S(d-1) + (d-1)\,R(k-1)$
  $S(d) = S(d-2) + (b-1) + R(d-2) + (b-1)\,S(d-1)$
  $S(k) = bS(d-2) + (d-1)\,S(d-3)$
- $b > 0,\ S(0) = 1 \rightarrow S(1) > S(0)$, et par récurrence $S(d) > S(d-1)$
  $\rightarrow\ S(d) < (2b-1)\,S(d-2) < 2b\,S(d-2)$, ce qui implique le nombre de nœuds à visiter pour obtenir la valeur exacte à la d-ième profondeur, et donc le facteur d'embranchement effectif, sera inférieur à $2b$ tous les deux niveaux, ce qui équivaut à $\sqrt{2b^d}$ pour tous les niveaux, et $\sqrt{2b^d}$ pour tous les d-niveaux.
- Si $d = 2k$ (nombre pair), le nombre de nœud visités est réduit au carré, et donc il est possible de doubler la profondeur et d'atteindre les mêmes performances de temps que dans l'algorithme sans élagage ayant une profondeur d . Si $d = 2k + 1$ (nombre impair), le nombre de nœuds visités est réduit d'un peu moins du carré, supérieur de b fois la complexité de l'algorithme avec élagage avec une profondeur paire.

Ainsi, en reprenant les chiffres avancés plus haut, et pour une complexité de $O(b^{\frac{d}{2}})$, on obtient un arbre à visiter de $30^{20} \approx 10^{29}$, ce qui réduit le nombre de nœuds à explorer de plus du carré du carré par rapport à l'algorithme classique.

L'ordre des meilleurs états pour un état donné peut être approximé en utilisant la fonction d'évaluation sur les états successeurs de cet état donné.

L'élagage par tables de transposition repose sur l'utilisation d'une table de transposition, qui stocke les états précédemment rencontrés et la valeur de ces état déterminé précédemment, dans l'optique de les réutiliser si rencontrer de nouveau à un entre endroit de l'arbre, économisant ainsi du temps de calcul et donc améliorant les performances en termes de temps, en évitant de recalculer l'entièreté d'un sous-arbre déjà calculé. Dans le jeu d'échec, retomber sur un même état est une possibilité, grande théoriquement parlant du fait qu'une combinaison de coup, alternant mouvement d'une pièce blanche et mouvement d'une pièce noir, et une autre combinaison équivalente à une transposition de la première amènera au même état en partant d'un même état donné, si l'on ignore les possibilités d'échec et les compteurs de mouvements non-déterministes et de répétitions des

échiquiers. Ainsi, il est plus compliqué en pratique de retomber sur le même état, mais tout de même possible, et les bénéfices d'utiliser cette table de transposition sont supérieurs à sa non-utilisation, du fait que la table de transposition n'est techniquement qu'une table de hachage avec les états pour clés, et que la recherche et l'insertion dans une table de hachage est toujours d'une complexité constante (de l'ordre de $O(1)$), dépendant uniquement de la fonction de hachage utilisé, dont sa complexité est presque assurément bien inférieur à l'exploration d'un sous-arbre entier.

La table de transposition devra aussi prendre en compte les valeurs limites α et β, la valeur exacte enregistré n'étant pas forcément la réelle valeur exacte du nœud en question, du fait que certains nœuds ne nécessitent que la connaissance des valeurs limites pour évaluer l'état relatif à ce nœud, et donc subisse un élagage α-β.

La fonction d'évaluation est utilisée pour donner une évaluation heuristique de l'état donné en argument. Cette évaluation se fait sur des arguments arbitraires, construite par l'humain, l'algorithme alpha-beta n'incluant aucun aspect d'apprentissage. Pour le jeu d'échec, cette fonction d'évaluation est construite sur :

- L'issue de la partie, qui supplante toute autre valeur du score si cette issue est différente de « non-terminé ».
- La somme des valeurs des pièces de la couleur actuelle présentes sur le plateau moins la somme des valeurs des pièces de la couleur opposée présentes sur le plateau :
    - Un pion vaut 1 point.
    - Un cavalier vaut 3 points.
    - Un fou vaut 4 points.
    - Une tour vaut 5 points.
    - Une reine vaut 9 points.
    - Un roi vaut 200 points, même si théoriquement et techniquement, sa capture est impossible.

La modification Negamax de l'algorithme Min-Max part du principe que le score d'un joueur dans un jeu à somme nulle à 2 joueurs revient à la négation du score du joueur ennemie (c'est-à-dire que $\max(a, b) = -\min(-a, -b)$), ce qui simplifie l'algorithme en appliquant une seule et même fonction Negamax, quelque soit la couleur du joueur, et en utilisant la négation du retour de cette fonction, c'est-à-dire la recherche du pire score du joueur ennemi, ce qui revient donc à la recherche du meilleur score du joueur concerné. Pour les valeurs alpha et beta, il suffit juste d'appliquer leur négation et d'inverser leur utilisation, suivant la même logique que la négation du score.

Tous ces éléments donnent un algorithme final tel que :

$negamax(noeud, profondeur, alpha, beta, joueur)$

- $entrée = tableTransposition[noeud]$ (on recherche s'il existe une entrée au nœud donné)
- $Si\ l'entrée\ existe\ et\ que\ l'entrée\ a\ une\ profondeur\ supérieur$ à $celle\ actuel$ (une profondeur supérieur indique une plus haute position dans l'arbre, impliquant une exploration plus profonde à partir de ce nœud et donc une meilleure précision de sa valeur)
    - $Si\ la\ valeur\ de\ l'entrée\ est\ celle\ de\ la\ valeur\ exacte$
        - $On\ renvoie\ la\ valeur\ de\ l'entrée$ (la valeur exacte du nœud est déjà connue, il est donc inutile de continuer la recherche à ce point)
    - $Sinon\ si\ la\ valeur\ de\ l'entrée\ est\ celle\ de\ la\ limite\ basse$
        - $alpha = \max(alpha, valeur\ de\ l'entrée)$ (on récupère la limite basse la plus haute, représentant la valeur minimum attendu par le joueur cherchant à maximiser cette valeur)
    - $Sinon\ si\ la\ valeur\ de\ l'entrée\ est\ celle\ de\ la\ limite\ haute$
        - $beta = \min(beta, valeur\ de\ l'entrée)$ (on récupère la limite haute la plus basse, représentant la valeur maximum attendu par le joueur cherchant à minimiser cette valeur)
    - $Si\ alpha \geq beta$
        - $On\ retourne\ la\ valeur\ de\ l'entrée$ (on retourne la valeur de l'entrée, la limite basse étant supérieur ou égale à la limite haute, rendant inutile toute recherche à ce point)
- $Si\ la\ profondeur\ est\ de\ 0\ ou\ le\ noeud\ est\ un\ noeud\ terminal$
    - $On\ retourne\ la\ valeur\ de\ fonctionEvaluation(joueur)$
- $noeudsSuccesseurs = noeudsAprèsMouvements(noeud)$
- $noeudsSuccesseurs = triParMeilleursNoeuds(noeudsSuccesseurs)$
- $valeur = -\infty$ (ne sachant pas la valeur exacte du nœud, on l'initialise avec la valeur du pire des cas, qui est la négation de l'infini)
- $Pour\ chaque\ noeudSuccesseur\ dans\ noeudsSuccesseurs$
    - $valeur = \max(valeur, -negamax(valeur, profondeur - 1, -beta, -alpha, joueurEnnemi)$ (on recherche la valeur maximum entre la valeur déjà connue et la négation du score du joueur ennemi)
    - $alpha = \max(alpha, valeur)$ (la limite basse est mise-à-jour, quelque soit le joueur, étant donné que dans Negamax, le but est de toujours maximiser cette limite basse, quelque soit le joueur, du fait de la négation de cette valeur à chaque changement de profondeur)
    - $Si\ alpha \geq beta$
        - $fin\ de\ la\ boucle$
- $valeur\ de\ l'entrée = valeur$
- $Si\ valeur \leq alpha$
    - $la\ valeur\ de\ l'entrée\ est\ la\ limite\ haute$ (une valeur inférieur ou égale à alpha implique que la limite haute a été atteinte)
- $Sinon\ si\ valeur \geq beta$
    - $la\ valeur\ de\ l'entrée\ est\ la\ limite\ basse$ (une valeur supérieur ou égale à beta implique que la limite basse a été atteinte)
- $Sinon$
    - $la\ valeur\ de\ l'entrée\ est\ la\ valeur\ exacte$
- $la\ profondeur\ de\ l'entrée\ est\ égale$ à $la\ profondeur\ actuelle$
- $tableTransposition[noeud] \leftarrow entrée$
- $On\ renvoie\ la\ valeur$

Une variante de Negamax plus efficace applicable est l'algorithme MTD-f, signifiant « Memory-enhanced Test Driver » (pilote de test à mémoire améliorée). Le principe est d'appliquer l'algorithme Negamax avec une fenêtre nulle sur les valeurs limites (c'est-à-dire $[\beta, \beta + 1]$), et de faire évoluer cette fenêtre jusqu'à sa convergence, indiquant que la valeur exacte attendue se trouve dans la fenêtre convergente. Le but est ainsi d'effectuer des recherches plus courtes, dû à la fenêtre des valeurs limites étant la plus étroite possible et donc de la forte probabilité d'élagage). Cette recherche est aussi aidée la table de transposition, réutilisant à une forte proportion les valeurs stockées des nœuds (dû à la répétition de l'algorithme en partant d'un même nœud comme racine) comparé à l'algorithme Negamax classique. Une optimisation de temps peut être aussi faite en sauvegardant la valeur de chaque action pour un état donné, et les ranger par ordre décroissant, permettant ainsi d'évaluer les états à la plus grande valeur, et donc d'engranger plus de coupures lors de la recherche. Enfin, MTD-f peut être appliqué à travers une recherche par profondeur itérative, effectuant ainsi des recherches à des profondeurs moindres, afin d'approximer la fenêtre nulle de recherche pour les futures recherches à une profondeur plus importante.

L'algorithme peut se décrire tel que :

$MTD(noeud, profondeur, f)$
$| g \leftarrow f$
$| limiteHaute \leftarrow +\infty$
$| limiteBasse \leftarrow -\infty$
$| Tant\ que\ limiteBasse < limiteHaute$
$|\quad | \beta \leftarrow \max(g, limiteBasse + 1)$
$|\quad | g \leftarrow negamax(noeud, profondeur, \beta - 1, \beta + 1)$
$|\quad | Si\ g < \beta$
$|\quad |\quad | limiteHaute \leftarrow g$
$|\quad | Sinon$
$|\quad |\quad | limiteBasse \leftarrow g$
$|\quad fin$
$| retourner\ g$

$MTD_{itératif}(noeud, profoneur)$
$| f \leftarrow 0$
$| Pour\ d = 1\ jusqu'à\ d = profondeur$
$|\quad | f \leftarrow MTD(noeud, d, f)$
$|\quad fin$
$| retourner\ f$

d. Processus de décision Markovien

Un processus de décision Markovien est un type de graphe, utilisé principalement dans les problèmes de décisions tel que le jeu d'échec pour représenter justement le processus entraînant le choix d'une action dans un état donné selon certains critères. Un processus de décision Markovien peut être représenté par un ensemble $\{S, t, A, P, R\}$ tel que :

- S est l'ensemble fini des états du processus de décision, avec S_0 l'ensemble fini des états initiaux.

- t est la valeur de temps, incrémenté à chaque changement d'état, avec $t = 0$ pour $s_t \in S_0$

- A est l'ensemble fini des actions possibles du processus de décision, avec A_s l'ensemble fini des actions légales pour un état s

- $P(s, a, s') = \mathbb{P}(s_{t+1} = s' \mid s_t = s,\ a_t = a)$, c'est-à-dire, pour un état s et une action a donné à la valeur de temps t, la probabilité d'atteindre l'état s' à la valeur de temps $t + 1$ (c'est-à-dire l'état successeur direct après l'exécution de l'action). Cette probabilité, appelé probabilité de transition, peut être nulle, auquel cas elle indique que l'état s' ne peut être un état successeur direct de s à la suite de l'exécution de a.

- $R(s, a, s') = r$ est la récompense accordé lorsque l'état s' est atteint en partant de l'état s et après exécution de l'action a, démontrant l'utilité d'une action dans la résolution du problème.

Pour résoudre un processus de décision markovien, il faut trouver une police π permettant de déterminer l'action à jouer selon un état s donné, tel que $\exists a_s \in A_s, \pi(s) = a_s$, ce qui implique $P(s, \pi(s), s') \sim P(s, s') = \mathbb{P}(s_{t+1} = s' \mid s_t = s)$, revenant au calcul d'une matrice de transition Markovienne, et le processus de décision markovien revenant à une chaîne de Markov avec récompense (une chaîne Markov étant un ensemble $\{S, P\}$, avec S défini précédemment et $P(s, s')$ aussi défini précédemment).

L'utilité d'une politique peut être déterminé par un critère d'utilité, que l'agent cherche à maximiser à travers cette politique choisie. Le critère le plus courant est celui de la récompense escomptée à horizon infini tel que $V^{\pi}(s_t) = E(\sum_{t=0}^{\infty} \gamma\, R(s_t, \pi(s), s_{t+1}))$, où $\gamma \in [0,1]$ est le facteur d'actualisation, définissant l'importance donnée au récompenses futures.

Ainsi, en utilisant l'équation de Bellman pour décrire la récompense escomptée dans un état s en suivant une politique π de façon récursive, tel que :

$$V^{\pi}(s) = \sum_{s'} P(s, \pi(s), s')\big(R(s, \pi(s), s') + \gamma\, V^{\pi}(s')\big)$$

Ou alors :

$$V^{\pi}(s) = Q^{\pi}\big(s, \pi(s)\big) = \sum_{s'} P(s, \pi(s), s')\left(R(s, \pi(s), s') + \gamma\, Q^{\pi}\big(s', \pi(s')\big)\right)$$

On peut décrire la récompense escomptée optimale dans un état s en suivant une politique optimale π^* tel que :

$$V^*(s) = \max_a(\sum_{s'} P(s,a,s')(R(s,a,s') + \gamma\, V^*(s')))$$

Ou alors :

$$V^*(s) = \max_a Q^*(s,a) = \sum_{s'} P(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q^*(s',a'))$$

Il est ainsi possible de définir des algorithmes pour apprendre la politique optimale de façon simple tel que :

- L'itération de politique tel que :
  - On calcule, pour un ou plusieurs états s, la fonction de valeur $V_i^\pi(s)$, avec $i > 0$ l'indice d'itération sur cette fonction de valeur (aussi appelé évaluation itérative de politique) en utilisant l'équation de Bellman tel que :

$$V_{i+1}^\pi(s) = \sum_{s'} P(s,\pi(s),s')(R(s,\pi(s),s') \; + \gamma\, V_i^\pi(s'))$$

  - On calcule ensuite, pour un ou plusieurs états s la nouvelle itération de la politique, avec $i > 0$ l'indice d'itération sur cette politique tel que :

$$\pi_{i+1}(s) = argmax_a(\sum_{s'} P(s,\pi(s),s')\big(R(s,\pi(s),s') \; + \gamma\, V_i^\pi(s')\big))$$

  - On répète ces 2 étapes jusqu'à atteindre la convergence de la politique vers la politique optimale (c'est-à-dire que $\forall s \in S, \forall k > 0, \pi_{i+k}(s) = \pi_i(s)$)

- L'itération de valeur tel que :
  - On calcule, pour un ou plusieurs états s, la nouvelle itération de la fonction de valeur $V_i(s)$ avec $i > 0$ l'indice d'itération sur cette politique en utilisant l'équation d'optimalité de Bellman tel que :

$$V_{i+1}(s) = \max_a\,(\sum_{s'} P(s,a,s')(R(s,a,s') \; + \gamma\, V_i^\pi(s')))$$

  - On répète cette étape jusqu'à atteindre la convergence de la fonction de valeur vers la fonction de valeur optimale (c'est-à-dire $\forall s \in S, \forall k > 0, V_{i+k}(s) = V_i(s)$)

Pour prouver que l'évaluation itérative de politique converge vers la fonction de valeur V^π, l'itération de politique et l'itération de valeur vers la fonction de valeur optimale V^*, on utilise le théorème de contraction cartographique indiquant que :

- Soit $\Upsilon = (\mathrm{X}, d)$ un espace métrique tel que :
  - X est un ensemble de valeurs
  - d est une fonction de distance qui, pour $x, y, z \in \mathrm{X}$:
    - $d(x,x) = 0$
    - $d(x,y) = d(y,x)$
    - $d(x,z) \leq d(x,y) + d(y,z)$

- Pour tout espace métrique Υ complet (c'est-à-dire de manière intuitive qu'il n'y aucun point manquant dans X ou à ses limites), alors une carte $\mathrm{T} = \mathrm{X} \to \mathrm{X}$ est une contraction cartographique stricte, ou contraction stricte, sur X si :

$$\forall x, y \in \mathrm{X}, \exists c : 0 \leq c < 1, d\big(\mathrm{T}(x), \mathrm{T}(y)\big) \leq cd(x, y)$$

  Ce qui indique que la distance entre les images des points x et y à travers la carte T est inférieure à la distance entre x et y, indiquant que T est une fonction de contraction de l'espace X.

- Un point $x \in \mathrm{X}$ tel que $\mathrm{T}(x) = x$ est appelé un point fixe, et, dans une contraction cartographique stricte de X, est unique. Un point fixe est donc un point dont la contraction à travers T ne modifie pas sa valeur.

- De manière analogue, si $\mathrm{T} : \mathrm{X} \to \mathrm{X}$ est une contraction d'un espace métrique $\Upsilon = (\mathrm{X}, \mathrm{d})$, alors il n'existe qu'un seul point $x \in \mathrm{X}$ tel que $\mathrm{T}(x) = x$, c'est-à-dire que x est un point fixe.

Il est ainsi possible de définir la fonction de valeur V^π en tant qu'espace vectoriel à $n = Card(S)$ dimensions v^π tel que :

$$v^\pi = p^\pi(r^\pi + \gamma v^\pi)$$

$$\begin{pmatrix} v^\pi_{s_1} \\ \vdots \\ v^\pi_{s_n} \end{pmatrix} = \begin{pmatrix} p^\pi_{s_1 \to s_1} & \cdots & p^\pi_{s_1 \to s_n} \\ \vdots & \ddots & \vdots \\ p^\pi_{s_n \to s_1} & \cdots & p^\pi_{s_n \to s_n} \end{pmatrix} \left(\begin{pmatrix} r^\pi_{s_1} \\ \vdots \\ r^\pi_{s_n} \end{pmatrix} + \gamma \begin{pmatrix} v^\pi_{s_1} \\ \vdots \\ v^\pi_{s_n} \end{pmatrix} \right)$$

La distance entre 2 fonctions de valeurs u et v peut être défini comme la différence entre u et v maximum tel que :

$$d_{max}(u, v) = \max_{0 < k \leq n} |u_k - v_k|$$

Pour prouver que $(\mathbb{R}^n, d_{max})$ est un espace métrique complet, il faut prouver que pour toutes suites $u_k \in \mathbb{R}^n$ tel que $\lim_{i,j \to \infty} d_{max}\big(u_i, u_j\big) = 0$, il existe $u \in \mathbb{R}^n$ tel que $\lim_{k \to \infty} d_{max}(u_k, u) = 0$. On devine ainsi que u_k est une suite de Cauchy, et que la définition précédente équivaut à dire que pour une valeur $\epsilon > 0$ donné, il existe un nombre l tel que, pour $i, j > l$, on a $d_{max}\big(u_i, u_j\big) < \epsilon$.

Pour prouver cela, on utilise le fait que $(\mathbb{R}, d)$ est un espace métrique complet, tel que $\forall x, y \in \mathbb{R}, d(x, y) = |x - y|$, ainsi que le fait que soit $\boldsymbol{u}_k \in \mathbb{R}^n$ une suite satisfaisant la condition $\lim_{i,j \to >\infty} d_{max}(\boldsymbol{u}_i, \boldsymbol{u}_j) = 0$, ce qui revient à prouver qu'il existe $\boldsymbol{u} \in \mathbb{R}^n$ tel que $\lim_{k \to \infty} d_{max}(\boldsymbol{u}_k, \boldsymbol{u}) = 0$.

On déclare que, pour chaque $\delta \in \mathbb{N}^+,\ \delta \leq n$, la suite du composant $u_{\delta,k}$ de dimension i de la suite $\boldsymbol{u}_k$ satisfait la condition $\lim_{i,j \to \infty} \big|u_{\delta,i} - u_{\delta,j}\big| = 0$

Ainsi, $\forall \delta \in \mathbb{N}^+,\ \delta \leq n$, $0 \leq \big|u_{\delta,i} - u_{\delta,j}\big| \leq \max_{0 < k \leq n} \big|u_{k,i} - u_{k,j}\big|$

$$\Rightarrow 0 \leq \lim_{i,j \to \infty} \big|u_{\delta,i} - u_{\delta,j}\big| \leq \lim_{i,j \to \infty} \max_{0 < k \leq n} \big|u_{k,i} - u_{k,i}\big|$$

$$\Leftrightarrow 0 \leq \lim_{i,j \to \infty} d\big(u_{\delta,i}, u_{\delta,j}\big) \leq \lim_{i,j \to \infty} d_{max}\big(\boldsymbol{u_i}, \boldsymbol{u}_j\big) \leq 0$$

Et donc chaque composant $u_{\delta,k}$ est une suite de Cauchy. De plus, comme $(\mathbb{R}, d)$ est complet, il existe donc $u_\delta \in \mathbb{R}$ tel que $\lim_{k\to\infty} |u_{\delta,k} - u_\delta| = 0$. Si on définit le vecteur $\boldsymbol{u} \in \mathbb{R}^n, \boldsymbol{u} = \sum_{i=1}^{n} u_i e_i$, où $u_i \geq 0$ est une constante, et e_i la norme sur la i-ème dimension, alors :

$$\begin{aligned}\lim_{k\to\infty} d_{max}(\boldsymbol{u}_k, \boldsymbol{u}) &= \lim_{k\to\infty} \max_{0<m\leq n} \left|u_{m,k} - u_k\right| \\ &= \max_{0<m\leq n} \lim_{k\to\infty} \left|u_{m,k} - u_k\right| \\ &= 0\end{aligned}$$

Donc $(\mathbb{R}'^n, d_{max})$ est un espace métrique complet par définition.

Intuitivement, une suite de Cauchy est une suite dont les termes converge vers une certaine valeur, appelé limite, à partir d'un certain nombre de termes. Ainsi, la distance entre 2 termes de cette suite, au-delà de ce certain nombre de termes, devient quasiment nulle, les deux termes devenant quasiment égaux entre eux et à la limite (sans pour autant atteindre cette limite), et ceux pour un nombre infini de termes. Ainsi, si toutes les suites de Cauchy d'un ensemble définissant un espace métrique convergent en utilisant la fonction de distance définissant ce même espace métrique, alors l'espace métrique est complet, ce qui indique qu'il ne manque aucun point à cet ensemble. A l'inverse, si une séquence de Cauchy ne converge pas ou converge vers une limite en dehors de l'ensemble définissant l'espace métrique, alors cet espace métrique n'est pas complet, indiquant que certains points se trouvent en dehors de l'ensemble définissant l'espace métrique et donc a besoin d'être agrégé à un autre ensemble pour potentiellement se compléter.

On définira l'opérateur de Bellman T^π: $\mathbb{R}^n \to \mathbb{R}^n$, tel que :

$$T^\pi(v) = p^\pi(r^\pi + \gamma v)$$

Si l'opérateur de Bellman est une contraction, alors :

$$d_{max}\big(T^\pi(u), T^\pi(v)\big) \leq d_{max}(u, v)$$

Or :

$$\begin{aligned}d_{max}\big(T^\pi(u), T^\pi(v)\big) &= \max_{0<k\leq n} |p_k^\pi(r_k^\pi + \gamma u_k) - p_k^\pi(r_k^\pi + \gamma v_k)| \\ &= \max_{0<k\leq n} |\gamma p_k^\pi(u_k - v_k)| \\ &\leq \gamma \max_{0<k\leq n} \left|p_k^\pi \max_{0\leq k'\leq n} |u_{k'} - v_{k'}|\right| \\ &\leq \gamma \max_{0<k\leq n} \left|p_k^\pi |u_k - v_k|\right| \\ &\leq \gamma \max_{0<k\leq n} |u_k - v_k|\end{aligned}$$

Donc l'opérateur de Bellman est bien une contraction, contractant les fonctions de valeurs d'au moins γ, et donc il existe un point fixe unique v_f tel que :

$$T^\pi\big(v_f\big) = p^\pi\big(r^\pi + \gamma v_f\big) = v_f$$

On peut aussi définir l'opérateur d'optimalité de Bellman tel que :

$$T^*(v) = \max_{a\in A} p^a\,(r^a + \gamma v)$$

Si l'opérateur d'optimalité de Bellman est une contraction alors :

$$d_{max}\big(T^*(u), T^*(v)\big) \leq d_{max}(u, v)$$

Or :

$$\begin{aligned} d_{max}\left(T^*(u), T^*(v)\right) &= \max_{0<k\leq n} |\max_{a\in A} p_k^a(r_k^a + \gamma u_k) - \max_{a\in A} p_k^a(r_k^a + \gamma v_k)| \\ &= \max_{0<k\leq n} \left|\gamma \max_{a\in A} p_k^a(u_k - v_k)\right| \\ &\leq \gamma \max_{0<k\leq n,\ a\in A} \left|p_k^a \max_{0\leq k'\leq n} |u_{k'} - v_{k'}|\right| \\ &\leq \gamma \max_{0<k\leq n,\ a\in A} \left|p_k^a |u_k - v_k|\right| \\ &\leq \gamma \max_{0<k\leq n} |u_k - v_k| \end{aligned}$$

Donc l'opérateur de Bellman est bien une contraction, contractant les fonctions de valeur d'au moins γ, et donc il existe un point fixe unique v_f tel que :

$$T^*(v_f) = \max_{a\in A} p^a(r^a + \gamma v_f) = v_f$$

Intuitivement, les 2 opérateurs de Bellman représentent simplement les équations de Bellman tel des suites de Cauchy, et les points fixes de ces opérateurs représentent la valeur de convergence de ces suites, tel que la valeur des fonctions de valeurs ne se modifient quasiment plus après un certain nombre d'itérations, atteignant une limite, et donc convergent.

L'évaluation itérative de politique converge donc vers V^π, l'itération de politique vers la politique optimale π^*, et l'itération de valeur vers V^*, utilisant donc implicitement la politique optimale.

Dans le cas du jeu d'échec :

- Le processus de décision markovien représente le processus de décision pour un joueur seulement, les décisions du joueur ennemi étant représenté par P, c'est-à-dire, à la suite de la décision d'exécuter l'action a dans un état s donné du joueur, la probabilité que le joueur ennemi effectue une action débouchant sur l'état s'. L'état s' représente donc l'état s après un couple de mouvement exécuté par les 2 joueurs.

- S_0 ne contient qu'un seul état pour le joueur blanc (les conditions initiales d'une partie d'échecs) et tous les états possibles après exécution d'un mouvement du joueur blanc, pour chaque mouvement légaux des blancs en partant des conditions initiales (ce qui équivaut à 20 états initiaux).

- On peut déclarer S_f l'ensemble des états finaux, c'est-à-dire les états représentant les fins de parties, tel que :
    - $A_{s\,\epsilon\, S_f} = \emptyset$ (il est impossible d'exécuter une action lorsqu'une fin de partie est déclarée).
    - Si l'exécution d'une action d'un joueur mène directement à la fin de partie, sans que le joueur ennemi puisse effectuer une action, pour un état $s_f \in S_f$ représentant cet état de fin de partie, $P(s, a, s_f) = 1$.

- On peut déclarer les récompenses de manière générale tel que :
    - $\forall s_V \in S_V,\ R(s,a,s_V) = 1$, où S_V est l'ensemble des états finaux menant à une victoire du joueur, et donc $S_V \subset S_f$
    - $\forall s_L \in S_L,\ R(s,a,s_L) = -1$, où S_L est l'ensemble des états finaux menant à une défaite du joueur, et donc $S_L \subset S_f$
    - $\forall s' \notin S_V, S_L,\ R(s,a,s') = 0$, indiquant que l'ensemble des états non-finaux et finaux menant à une partie nulle ne donne aucune récompense.

- $Card(A) = 4672$, en comptant tous les mouvements possibles, même ceux destinées au joueur ennemi, et donc non-légaux. $Card(S) \approx 10^{43}$, étant le nombre de dispositions possibles aux échecs, légales ou non, aussi théorisé par Shannon.

- Il est impossible de déterminer $P(s,a,s')$ pour les états s' autres que les états finaux directs (sans action de la part du joueur ennemi), étant donné qu'il est impossible de déterminer l'action du joueur ennemi en réponse à l'action a exécuté par le joueur. Il n'est possible que de donner un encadrement tel que $0 \leq P(s,a,s') < 1$ et de donne des conditions initiales à ce processus de décision, en vue de son évolution tel que $\forall a_s \in A_s, P(s,a_s,s') = \frac{1}{Card(A_s)}$.

En théorie il est possible d'utiliser l'itération de politique ou de valeur pour trouver la politique optimale pour le jeu d'échec, à condition de rajouter, pour chaque transition (s,a,s'), un conteur de visites c_v, incrémenté à chaque fois que l'état actuel devient s' après avoir effectué l'action a dans l'état s et donc servant de poids, permettant ainsi d'influencer la probabilité de transitions $P(s,a,s')$ tel que $P(s,a,s') = \frac{c_v(a,s')}{\sum_{s''} c_v(a,s'')}$. Cette évolution de la probabilité de transition convergera, et donc n'annule pas la convergence originelle de l'itération de politique et de valeur. Il faudra aussi stocker, pour chaque état, la fonction de valeur dans une table, et la mettre à jour à chaque itération de ces algorithmes.

En pratique cependant, ces algorithmes ne pourront jamais converger, du fait à l'espace immense nécessaire pour contenir l'ensemble des états et leurs valeurs. Si nous simplifions un état à l'unique représentation de l'échiquier la plus irréductible possible, tenant pour une représentation sur 64 octets (un tableau de 8 entiers de 64 bits), il faudrait $64 * 10^{43}$ octets de capacité de mémoire pour stocker l'ensemble des états possibles (légaux et non-légaux), ce qui est tout simplement impossible, compte tenu du matériel actuel. Il est aussi impossible de vouloir calculer la fonction de valeurs pour tous les états en même temps, étant donné la taille du domaine des états, et inutile, du fait des états non-légaux et des états extrêmement rares possibles.

De plus, par l'immensité du nombre d'états possible et du fait que la fonction de récompense n'octroie une récompense non-nulle que lors de l'utilisation, l'algorithme peut sembler converger très vite en pratique, alors qu'en théorie il est loin de la fonction de valeur et de la politique optimale.

La recherche de politique optimale est donc au cœur de la résolution des processus de décisions markoviens. Deux techniques sont utilisées au cours de ce projet pour tenter d'approximer cette politique optimale :

- Le Q-Learning, une technique d'apprentissage par renforcement en ligne (c'est-à-dire un apprentissage à travers l'expérience sur un modèle), par la rétropropagation des récompenses à travers une fonction d'évaluation de qualité d'un état et d'une action, appelé Q.

- XCS, une autre technique d'apprentissage par renforcement en ligne se basant sur un système de classeur, représentant le processus de décision à travers des classeurs, qui modélisent les états à travers ces derniers avec certains degrés de généralité, et qui subissent un processus évolutif.

e. Exploration et exploitation

L'apprentissage par renforcement en ligne se base sur 2 paradigmes :

- L'exploration du modèle donné, afin d'en faire ressortir toutes les possibilités et ainsi élargir le domaine de prise de décision.
- L'exploitation du modèle donné, afin d'affiner les conséquences de choix des possibilités et ainsi préciser et assurer ce choix lors de la prise de décision.

L'utilisation de ces 2 paradigmes doit être équilibré, une exploration trop poussée rendant la politique de décision imprécise et une exploitation trop prononcée ne donnant pas une politique assez polyvalente. Il est ainsi naturel de favoriser l'exploration en début d'apprentissage, afin d'étoffer l'univers des possibilités, et d'augmenter les chances d'exploitation au fur et à mesure de l'apprentissage, précisant les issues de ce large domaine obtenu depuis le début.

f. Q-Learning

Le Q-Learning est une méthode d'apprentissage par renforcement en ligne, dont le but est la rétropropagation des récompenses à travers une fonction d'évaluation de qualité d'un état et d'une action, appelé Q, à travers tout le processus de décision. Pour cela, on utilise une Q-Table qui, pour un état s et une action a donné, nous donne sa valeur de qualité $Q(s,a)$. Cette table est initialisée de façon arbitraire, et est mis à jour tel que :

$$Q(s,a) = Q(s,a) + \alpha(R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

- s est l'état actuel et a l'action choisi.
- $\alpha \in [0,1]$ est le taux d'apprentissage, c'est-à-dire la proportion de nouvelle information qui est enregistré dans la nouvelle valeur de Q.
- $R(s,a,s')$ est la récompense obtenue après exécution de l'action a dans l'état s, transitant vers l'état successeur s'.
- $\gamma \in [0,1]$ est le facteur d'actualisation, c'est-à-dire l'importance donnée aux valeurs obtenues dans le futur.
- $\max_{a'} Q(s',a')$ est l'estimation de la future valeur optimale.

Cette équation est obtenue à partir de l'équation d'optimalité de Bellman pour la fonction de qualité Q, avec l'état s' déterminé à l'avance (et donc enlevant la nécessité d'utiliser la fonction de probabilité de transition P), tel que :

$$Q_{t+1}(s,a) = (1-\alpha)Q_t(s,a) + \alpha\left(R(s,a,s') + \max_{a'} Q(s',a')\right)$$

La mise-à-jour est ainsi faite tel qu'une proportion $1 - \alpha$ de l'ancienne valeur de la fonction de qualité Q est gardé, et une proportion α de la nouvelle valeur.

Cette mise-à-jour est appliqué tel que :

$QLearning(\alpha, \gamma, \epsilon)$

| $Initialisation\ de\ la\ QTable$

| $Tant\ que\ \neg(\pi_Q \approx \pi_*)$

| | $s \leftarrow s_0$

| | $Tant\ que\ s \notin S_f$

| | | $a \leftarrow \pi_Q(s, \epsilon)$

| | | $s' \leftarrow Appliquer\ a\ dans\ s$

| | | $Q(s,a) \leftarrow Q(s,a) + \alpha(R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a))$

| | | $s \leftarrow s'$

| | fin

| fin

$\pi_Q(s, \epsilon)$

| $Si\ Rand(0,1) < \epsilon$

| | $a \leftarrow Rand(a_{rand} \mid a_{rand} \in A_s)$

| $Sinon$

| | $a \leftarrow a_m \mid Q(s,a_m) = \max_{a_s} Q(s,a_s),\ \ a_m, a_s \in A_s$

| $Retourner\ a$

Le Q-Learning s'inspirant de l'itération de valeur, les conditions de convergences sont semblables, c'est-à-dire qu'à partir une certaine valeur $M \in \mathbb{N}^+$, pour $i, j > M$, $Q_i(s,a) = Q_j(s,a)$, indiquant que la fonction de qualité Q converge vers la fonction de qualité optimale Q^*, et donc implicitement vers la politique optimale π^*. La complexité du Q-Learning est de $O(mn)$, avec m le nombre moyen d'itération pour atteindre l'état final et n le nombre moyen d'itération pour atteindre la convergence, faisant du Q-Learning un algorithme très rapide pour atteindre la solution optimale.

Tout comme l'itération de valeur, cette solution, bien que théoriquement capable de résoudre le jeu d'échec, est impraticable pour le jeu d'échec, dû au trop grand nombre d'états (10^{43}). De plus, étant donné ce grand nombre d'états, la convergence peut être atteindre sans pour autant converger vers la solution optimale. Il est ainsi nécessaire de déterminer un nombre minimal de mise-à-jour pour éviter une convergence prématurée non-optimale.

g. XCS

Un système de classeurs est un système d'apprentissage en ligne, possiblement par renforcement, avec pour méthode de constituer un ensemble de règles, appelés classeurs, déterminant la politique optimale. Dans un système de classeurs, l'élément d'exploration se base principalement sur un algorithme génétique, permettant de créer de nouveaux classeurs à partir de ceux préexistant. L'élément d'exploitation, dans un système d'apprentissage par renforcement, est une mise-à-jour des paramètres de prédiction et de précision de cette prédiction.

XCS est un système de classeurs d'apprentissage par renforcement, générant un ensemble de règles qui se veut précis dans leur prédiction respective. Un classeur dans XCS possède plusieurs variables :

- C est la condition de correspondance, permettant de vérifier si la règle définit par ce classeur correspond à une situation donné σ, extraite d'un état s donné. Cette condition est un tableau de n variables, pouvant chacune prendre k valeurs. Ces variables peuvent aussi prendre pour valeur λ, cette valeur pouvant correspondre à toutes les valeurs de la variable donné.

- A est l'action à accomplir dans la règle, si la condition C correspond à une situation σ donné.

- p est la prédiction de récompense attendue de la règle.

- ϵ est la marge d'erreur fait sur la prédiction p du classeur.

- f est l'aptitude d'un classeur, utilisé dans une logique évolutive pour calculer son aptitude de survie dans les processus de sélection des algorithmes génétiques.

- exp est l'expérience d'un classeur, incrémenté à chaque fois que le classeur voit sa condition C correspondant à la situation σ et que l'action A correspond à l'action a choisie.

- ts est le pas de temps de la dernière occurrence de l'algorithme génétique sur ce classeur.

- num est la numérosité, en considérant ce classeur comme un macro-classeur, cela représente le nombre de micro-classeur que ce macro-classeur généralise.

XCS possède 4 principales variables :

- t est le pas de temps, incrémenté à chaque pas de temps de l'algorithme.

- $[P]$ est l'ensemble représentant la population de classeurs à un instant t donné. La taille de la population est limitée par la constante $N \in \mathbb{N}^+$.

- $[M]$ est l'ensemble représentant la partie de la population dont leurs conditions respectives C correspondent à la situation σ, extraite de l'état s donné à l'argument.

- $[A]$ est l'ensemble représentant la partie de $[M]$ dont leurs actions respectives A correspondent à l'action choisie a.

Le déroulement simplifié d'une étape XCS est tel que :

- L'état s actuel est récupéré par la variable de situation σ, sous la forme d'un tableau de n variables pouvant prendre chacune k valeurs.

- Les classeurs dans $[P]$ correspondant à la situation σ sont ajoutées à $[M]$.
    - Un classeur correspond à une situation σ si, pour tout attribut $cl.C_i \in cl.C, cl \in [P]$, $1 \leq i \leq k$, si $cl.C_i = \sigma_i$ ou $cl.C_i = \lambda$.

- Tant qu'il n'existe pas assez de classeurs dans $[M]$ dont le nombre total de leurs actions distinctes est inférieure au nombre total d'actions possibles, un classeur est généré pour chaque action possible non couverte et ajouté à $[M]$.
    - La condition C d'un classeur généré cl est créé tel que pour $1 \leq i \leq k$, $cl.C_i \leftarrow \begin{cases} \sigma_i \; si \; Rand([0,1[) > P_{\#} \\ \lambda \; sinon \end{cases}$, avec $P_{\#} \in [0,1]$ la probabilité de généraliser une variable.
    - L'action A est choisie parmi les actions non-couvertes.
    - Les autres paramètres sont initialisés par leur valeur initiales (p_i, ϵ_i, f_i respectivement pour p, ϵ, f, 0 pour exp, t pour ts, 1 pour as, 1 pour num).
    - On procède à la suppression dans la population pour chaque classeur généré.

- La prédiction de chaque action est calculée, à partir de la prédiction p pondéré et moyenné par l'aptitude f respective de chaque classeur cl dans $[M]$ couvrant cette action, obtenant ainsi un tableau de prédiction PA.

- L'action à effectuer a est choisie tel que $a \leftarrow \begin{cases} argmax_A\big(PA(A)\big) \; si \; Rand([0,1[) > P_{exp} \\ Rand_A\big(PA(A)\big) \end{cases}$

- Les classeurs cl dans $[M]$ dont leur action $cl.A$ correspondent à l'action choisie a sont ajoutées à $[A]$.

- L'action a est effectué, la récompense r est récupéré, menant ainsi à l'état successeur s'.

- La prédiction générale P et l'ensemble d'action $[A]$ sont mis-à-jour tel que :
    - Si $[A]_{-1} \neq \emptyset$, avec $[A]_{-1}$ l'ensemble d'action au moment $t-1$
        - $P \leftarrow r + \gamma \max(\mathrm{PA})$, avec r_{-1} la récompense au moment $t-1$.
        - Les classeurs dans $[A]_{-1}$ sont mis-à-jour avec P.
        - L'algorithme génétique est utilisé sur $[A]_{-1}$ en considérant σ_{-1}, la situation au moment $t-1$.
    - Si l'état successeur s' est un état final
        - $P = r$
        - Les classeurs dans $[A]$ sont mis-à-jour.
        - L'algorithme génétique est utilisé sur $[A]$ en considérant σ.

- Les classeurs cl dans $[A]$ sont mis-à-jour avec P et $[A]$ donné en argument tel que :
    - $cl.exp \leftarrow cl.exp + 1$
    - Si $cl.exp < 1/\beta$, avec $\beta \in [0,1]$ le facteur d'apprentissage.
        - $cl.p \leftarrow cl.p + (P - cl.p) \ / \ cl.exp$
        - $cl.\epsilon \leftarrow cl.\epsilon + (|P - cl.p| - cl.\epsilon) \ / \ cl.exp$
        - $cl.as \leftarrow cl.as + (\sum_{c \in [A]} c.num - c.as) \ / \ cl.exp$
    - Sinon
        - $cl.p \leftarrow cl.p + \beta(P - cl.p)$
        - $cl.\epsilon \leftarrow cl.\epsilon + \beta(|P - cl.p| - cl.\epsilon)$
        - $cl.as \leftarrow cl.as + \beta(\sum_{c \in [A]} c.num - c.as)$
    - L'aptitude f des classeurs est mise-à-jour avec $[A]$ donné en argument tel que :
        - Soit κ la valeur de précision selon un classeur.
        - $\forall cl \in [A]$,
            - $\kappa(cl) \leftarrow \begin{cases} 1 \ si \ cl.\epsilon < \epsilon_0 \\ \alpha(cl.\epsilon \ / \ \epsilon_0)^{-v} \ sinon \end{cases}$, tel que ϵ_0 est la marge d'erreur en dessous de laquelle toute marge d'erreur est considéré avoir une précision égale, α le facteur d'apprentissage pour la marge d'erreur
            - $\mathrm{K} \leftarrow \mathrm{K} + \kappa(cl) * cl.num$
        - $\forall cl \in [A]$
            - $cl.f \leftarrow cl.f + \beta(\kappa(cl) * cl.num \ / \ \mathrm{K} - cl.f)$
    - Si $subsumption_A = vrai$, la vérification de la subsumption des classeurs dans $[A]$ entre eux est faite.

- L'algorithme génétique est appliqué sur $[A]$ avec $[A]$ et σ donné en argument tel que :
    - Si $t - \sum_{cl \in [A]} cl.ts * cl.num \ / \ \sum_{cl \in [A]} cl.num > \theta_{GA}$, où θ_{GA} est le seuil d'activation de l'algorithme génétique.
        - $\forall cl \in [A], cl.ts = t$
        - $parent_1, \ parent_2 \leftarrow$ un classeur aléatoire dans $[A]$, dont leurs chances sont pondérés positivement par leurs aptitudes respectives $cl.f$.
        - $fils_1, fils_2 \leftarrow$ enfant respective de $parent1, parent2$
        - $fils_1.num \leftarrow fils_2.num \leftarrow 1$
        - $fils_1.exp \leftarrow fils_2.exp \leftarrow 0$
        - Si $Rand([0,1[) < \chi$, où χ est la probabilité de croisement.
            - Un croisement est effectué entre les 2 fils, interchangeant les attributs de $fils_1$ et $fils_2$ entre une plage x et y aléatoire.
            - $fils_1.p \leftarrow fils_2.p \leftarrow (parent_1.p + parent_2.p) \ / \ 2$
            - $fils_1.\epsilon \leftarrow fils_2.\epsilon \leftarrow 0.25 * (parent_1.\epsilon + parent_2.\epsilon) \ / \ 2$
            - $fils_1.f \leftarrow fils_2.f \leftarrow 0.1 * (parent_1.f + parent_2.f) \ / \ 2$
        - $\forall enfant \in \{enfant_1, enfant_2\}$
            - Le processus de mutation modifie, pour chaque $cl \in [A]$, pour $1 \leq i \leq n$, si $Rand([0,1[) < \mu$, où μ est la probabilité de mutation, l'attribut $cl.C_i$ tel que $cl.C_i \leftarrow \begin{cases} \sigma_i \ si \ cl.C_i = \lambda \\ \lambda \ sinon \end{cases}$, ainsi que $cl.A \leftarrow Rand_{a \neq cl.A}(A_s)$ si $Rand([0,1[) < \mu$.

- Si $subsumption_{GA} = vrai$
  - Si $parent_1$ subsume $enfant$, $parent_1.num \leftarrow parent_1.num + 1$
  - Sinon si $parent_2$ subsume $enfant$, $parent_2.num \leftarrow parent_1.num + 1$
  - Sinon on insère $enfant$ dans la population $[P]$.
- Sinon on insère $enfant$ dans la population $[P]$.
- On procède à la suppression dans la population.

L'insertion dans la population $[P]$ vérifie que le classeur cl donné en argument correspond en termes de condition et d'action à un classeur $c \in [P]$. Si c'est le cas, $c.num \leftarrow c.num + 1$, sinon cl est ajouté dans $[P]$.

La suppression dans la population $[P]$ se fait si $\sum_{c\in[P]} c.num \geq N$. Cette suppression se fait tel que :

- $vote = \sum_{c\in[P]} voteDeSuppression(c)$, où $voteDeSuppression$ est une fonction calculant le score de suppression, corrélé négativement avec l'aptitude d'un classeur tel que :
  - $vote_c \leftarrow cl.as * cl.num$
  - $aptitudeMoyenne \leftarrow \sum_{c\in[P]} c.p \ / \ \sum_{c\in[P]} c.num$
  - Si $cl.exp > \theta_{del}$ et $cl.f \ / \ cl.num \ < \delta * aptitudeMoyenne$, où δ est la fraction de l'aptitude moyenne en dessous de laquelle l'aptitude d'un classeur peut être considéré dans sa probabilité de suppression.
    - $vote_c \leftarrow vote_c * aptitudeMoyenne \ / \ cl.f \ / \ cl.num$
- $pointDeChoix = Rand([0,1[) * vote$
- $vote = 0$
- $\forall c \in [P]$
  - $vote = vote + voteDeSuppression(c)$
  - Si $vote > pointDeChoix$
    - Si $c.num > 1$, $c.num \leftarrow c.num - 1$
  - Sinon
    - $[P].supprimer(c)$
  - On sort de l'algorithme.

La subsumption est le processus permettant de regrouper les classeurs, appelés micro-classeurs, dont leurs conditions correspondent à la condition d'un autre classeur, appelé macro-classeur. Ce processus permet de compacter la population avec une grande efficacité, et d'empêcher des classeurs contradictoires dans la population. Cependant, sa non-utilisation n'empêche pas le système de converger vers une politique optimale, et peut même être plus bénéfique en ayant des classeurs plus spécifiques décrivant la situation sous différents angles.

La subsumption dans l'ensemble d'action $[A]$ se déroule tel que :

- Soit cl un classeur tel que $cl \leftarrow \emptyset$
- $\forall c \in [A]$,
    - Si c peut subsumer un autre classeur, si $cl = \emptyset$ ou c est plus général que cl, $cl \leftarrow c$
- Si $cl \neq \emptyset$
    - $\forall c \in [A]$, si cl est plus général que c
        - $cl.num = cl.num + c.num$
        - $[A].supprimer(c)$
        - $[P].supprimer(c)$

La subsumption de cl_{tos} par cl_{sub} dans l'algorithme génétique se vérifie tel que si $cl_{sub}.A = cl_{tos}.A$, $cl.sub$ peut subsumer et cl_{sub} est plus général que cl_{tos}, alors cl_{sub} subsume cl_{tos}.

La condition de subsumption est définie tel que si $cl.exp > \theta_{sub}$ et si $cl.\epsilon < \epsilon_0$, alors cl peut subsumer, avec θ_{sub} étant le seuil d'expérience au-delà duquel un classeur peut être considéré pour une subsumption.

Un classeur cl_{gen} est plus général que cl_{spec} si $Card_\lambda(cl_{gen}.C) > Card_c(cl_{spec}.C)$, et que pour $1 \leq i \leq n$, si $cl_{gen}.C_i = \lambda$ ou $cl_{gen}.C_i = cl_{spec}.C_i$.

L'utilisation de la valeur généralisant λ permet ainsi de réduire, et même d'annuler, le problème de stockage posé dans le Q-Learning. En effet, si on considère qu'une condition C est constitué de n variables pouvant prendre k valeurs chacune, le nombre d'états théorique est de k^n (étant donné que toutes les valeurs possibles de C est la combinaison de toutes les valeurs k possibles pour chaque variable n). Si g est le nombre moyen de λ dans une condition, alors le nombre de conditions totales possible est de $\binom{n}{g} k^{n-g}$, étant donné que si il y a un nombre de λ égale à g, alors le codage des valeurs possibles de C sur les autres variables non généralisé se fait sur $n - g$ variables, et donc il existera k^{n-g} valeurs possibles sur cette plage de variable, étant donné une combinaison de λ sur n variables. De ce fait, le nombre de combinaisons possibles uniques de d'un nombre de λ égale à g sur n variables revient donc à $\binom{n}{g}$, obtenant ainsi cette formule. Ainsi, une condition comportant g variables égale à λ correspond à k^g états, étant donné le nombre de valeur possible qui peut être codé sur ces g variables.

La convergence de XCS est atteinte lorsque, à partir d'un certain moment M avec $i, j > M$ et $i > j$:

- $Card([P]_i) = Card([P]_j)$, où $[P]_x$ est la population au moment x donné.
- Pour $1 \leq k \leq N$, $[P]_{i,k} = [P]_{j,k}$, où $[P]_{x,y}$ est le classeur $cl \in [P]$ à l'index y au moment x donné.

Tout comme pour le Q-Learning, un minimum de pas de temps passé peut-être appliquer dans le critère de terminaison de l'algorithme, afin d'éviter une convergence prématurée.

XCS est bien plus lourd en termes de complexité, étant donné le plus grand nombre d'opérations appliqué. La complexité de ces opérations est telle que :

- La récupération de la situation σ à partir de l'état s actuel a une complexité insignifiante.

- La vérification des classeurs dans $[P]$ correspondant la situation σ a une complexité de $O(p)$, où $p = Card([P])$ à un instant donné t.

- La création de classeurs couvrant les actions restantes est d'une complexité de $O(cC_{sup})$ où c est le nombre d'actions restantes à couvrir, et C_{sup} est la complexité de l'opération de suppression dans la population.

- L'opération de génération du tableau de prédiction est d'une complexité de $O(m + a_s)$, où $m = Card([M])$

- L'opération de sélection d'action a une complexité insignifiante.

- L'opération de génération de l'ensemble d'action $[A]$ a une complexité de $O(m)$, où $m = Card([M])$.

- L'opération de mise-à-jour de l'ensemble d'action $[A]$ donné en argument a une complexité de $O(a + C_f + C_{subA})$ dans le pire des cas, où $a = Card([A])$ et C_f la complexité de la mise-à-jour de l'aptitude de $[A]$ et C_{subA} la complexité de l'opération de subsumption sur $[A]$, et $O(a + C_f)$ dans le meilleur des cas.

- $C_f = O(a)$

- L'opération de mise-à-jour de l'aptitude a une complexité de $O(2a) = O(a)$.

- L'algorithme génétique a une complexité de dans le pire des cas de $O(a + C_{des} + C_{cro} + C_{mut} + C_{subGA} + C_{ins} + C_{sup})$, où C_{des} est la complexité de sélection de la descendance, C_{cro} est la complexité de l'opération de croisement, C_{mut} est la complexité de l'opération de mutation, C_{subGA} est la complexité de l'opération de subsumption dans l'algorithme génétique et C_{ins} est la complexité de l'opération d'insertion, et de $O(a)$ dans le meilleur des cas.

- $C_{des} = O(a)$

- C_{cro} est insignifiante.

- C_{mut} est insignifiante.

- $C_{ins} = O(p)$ dans le pire des cas.

- $C_{sup} = O(p + pC_{supV})$ dans le pire des cas, où C_{supV} est la complexité de l'opération de vote de suppression.

- $C_{supV} = O(p)$

Dans le cas du jeu d'échec, la condition et la situation est constitué de $n = 64$ variables pouvant chacune prendre chacune $k = 13$ valeurs, 12 pour la totalité des pièces de 2 couleurs, et une pour la case vide. On a ainsi un nombre d'états théorique de $13^{64} \approx 10^{71}$ (bien que cela est impossible, vu que certaines combinaisons de valeurs que cette formule permet sont impossibles dans le jeu d'échec, comme un échiquier rempli de pion blanc par exemple). Si j est le nombre moyen de λ dans la condition d'un classeur, on obtient un nombre de conditions possible de $\binom{64}{j} 13^{64-j}$, chacune pouvant correspondre à 13^j états. De plus, en sachant la condition d'un classeur peut être défini comme un tableau de 64 entiers d'un octet chacun, cela fait que la taille d'une condition est de 64 octets, et donc la taille maximale d'une population est de $\binom{64}{j} 13^{64-j} * 64$ octets. Ainsi, si on définit une taille maximale de 16 Go, la résolution de l'inéquation $\binom{64}{j} 13^{64-j} * 64 \leq 1{,}6 * 10^{10}$ nous donne le nombre moyen j de λ dans la condition d'un classeur pour respecter cette condition. Cette inéquation étant impossible à résoudre par des moyens conventionnels, elle peut être résolu une itération de j, étant donné que $0 \leq j \leq n$ et donc dans notre cas $0 \leq j \leq 64$, incrémentant j jusqu'à obtenir une contradiction dans l'inéquation. Ainsi, on obtient $j \geq 59$, et donc que $P_\lambda \geq \frac{j}{n} = \frac{59}{64} = 0.921875$, et que le nombre moyen de conditions possible est. En pratique, ce nombre est considérablement réduit, du fait qu'il ne peut y avoir qu'un certain nombre de type de pièce d'une certaine couleur sur l'échiquier en même temps (à 10^{43} états possibles comme indiqué plus haut).

VI. Résultats

Le tableau ci-dessous indique les résultats de chaque scénario de test sur 100 parties, tel que :

- Les IAs en abscisse indiquent l'IA utilisé pour le joueur blanc.
- Les IAs en ordonnée indiquent l'IA utilisé pour le joueur noir.
- Le croisement d'une ligne X et d'une colonne Y contient les résultats des 100 parties entre l'IA indiqué en X et l'IA indiqué en Y, indiqué comme W/B/D, où W est le nombre de parties gagnées par les blancs, B le nombre de parties gagnées par les noirs, et D le nombre de parties nulles.

Les IAs utilisées pour les tests sont :

- NM-4 : un Negamax avec élagage et table de transposition à profondeur 4.
- NM-6 : un Negamax avec élagage et table de transposition à profondeur 6.
- MTD-4 : un MTD-f à profondeur 4.
- IMTD-4 : un MTD-f itérative à profondeur 4.
- XCS-1000 : un XCS entrainé sur 1000 parties avec les paramètres suivant :
    - $N = 2147483647$
    - $\beta = 0.1$
    - $\alpha = 0.1$
    - $\epsilon_0 = 0.01$
    - $v = 5$
    - $\gamma = 0.71$
    - $\theta_{GA} = 25$
    - $\chi = 1$
    - $\mu = 0.05$
    - $\theta_{del} = 20$
    - $\delta = 0.1$
    - $\theta_{exp} = 20$
    - $P_\lambda = 0.921875$
    - $p_i = 0$
    - $\epsilon_i = 0.01$
    - $f_i = 0$
    - $p_{exp} = 0.5$
    - $subsumption_{GA} = vrai$
    - $subsumption_{[A]} = vrai$
- XCS-10000 : un XCS entrainé sur 10000 parties avec les paramètres décrits ci-dessus.

Blanc/Noir	NM-4	MTD-4	IMTD-4	XCS-1000	XCS-10000
NM-4	72/20/8	4/56/20	0/96/4	100/0/0	98/0/2
MTD-4	80/2/18	14/12/74	2/30/68	100/0/0	100/0/0
IMTD-4	94/0/6	38/6/56	38/20/42	100/0/0	100/0/0
XCS-1000	0/100/0	0/100/0	0/100/0	0/0/100	0/0/100
XCS-10000	0/94/6	0/100/0	0/100/0	0/0/100	0/0/100

Les résultats confirment que :

- MTD-4 a de meilleures performances que NM-4 et IMTD-4 a de meilleures performances que MTD-4, quelque que soit la couleur. Cela prouve que la recherche par fenêtre nulle est plus efficace que la recherche par fenêtre infini, et que l'itération de profondeur de recherche apporte lui aussi un avantage quant à la recherche du meilleur coup.

- XCS-1000 et XCS-10000 n'ont réussi à battre aucune des IAs de recherche fonctionnant avec Negamax, prouvant que XCS n'est pas efficace en dessous de 1000 parties apprises, même contre Negamax ayant une profondeur de 1 (résultat non présent sur le tableau).

- L'opposition de 2 algorithmes de recherche de même type avec une profondeur égale à 4 pour les 2 algorithmes n'implique pas une répétition de partie nulle, impliquant que la profondeur de recherche n'est pas assez grande pour éviter toute défaite possible au jeu d'échec, et donc qu'une série de mouvements permettant d'arriver à une situation d'échec et mat peut être plus grande que 4.

- L'opposition de XCS-1000 et XCS-10000 contre eux-mêmes ou entre chacun d'eux n'implique que des parties nulles, rendues nulles par répétition d'échiquier, ce qui indique qu'à un certain point, XCS ne fait que répéter les 2 mêmes mouvements contre lui-même, et donc que le meilleur cours d'action est celui menant vers une partie nulle, étant donné que les classeurs choisissent des deux camps pour déterminer la meilleure action seront toujours les mêmes. Un apprentissage plus poussé est cependant nécessaire pour confirmer cette tendance.

- L'opposition de XCS-10000 et de NM-4 montre que XCS-10000 peut arriver à un scénario de nulle avec une faible probabilité, par répétition d'échiquier, indiquant que NM-4 ne fait que répéter le même mouvement tout comme XCS-10000, et ainsi que NM-4 se trouve dans une position dans l'arbre des possibilités où sa profondeur limitée de 4 ne lui permet pas de retrouver un chemin menant vers un meilleur score autre que celui de la répétition.

- L'opposition de 2 algorithmes de recherche de même type avec une profondeur égale à 4 fait ressortir un plus grand nombre de victoires pour les blancs que pour les noirs, impliquant que le fait de jouer en premier est un avantage dans le jeu d'échec.

VII. Conclusion

Le projet a été un succès sur divers points, notamment en termes de constructions des règles et en intelligence artificielle. Bien que plusieurs IAs n'ont pas pu démontrer leur efficacité du point de vue pratique (que ce soit le Q-Learning pour des raisons de stockage ou XCS pour des raisons de rapidité d'exécution), elles ont été démontrées comme efficace d'un point de vue théorique, et d'autres approches pourrait être utilisées afin d'obtenir des IA d'apprentissage au moins aussi efficace que des IA de recherche. Néanmoins, dans le contexte d'un jeu d'échec sur ordinateur personnelle, la meilleure solution pour une IA reste un algorithme de recherche tel que Negamax et ses variantes, dont la puissance de réflexion et donc de décision peut être renforcé en optimisant au mieux possible le déroulé d'une partie et le traitement des nœuds dans l'algorithme lui-même.

Ce projet n'étant qu'à sa première version, plusieurs ajout et modifications de fonctionnalités sont déjà prévues, parmi lesquelles :

- L'optimisation des règles et du traitement d'une partie.
- L'optimisation de Negamax et de ses variantes.
- L'amélioration et la précision de la fonction d'évaluation de Negamax et de ses variantes.
- L'optimisation de XCS.
- L'amélioration de l'algorithme génétique et des fonctions de mise-à-jour de XCS.
- Le changement de représentation de XCS (passé d'un environnement à valeur intégrale à un environnement à valeur booléenne).
- La modification de la structure de données permettant de faire fonctionner le Q-Learning (passé d'une table à 2 entrées à un réseaux de neurones).
- La création d'une interface graphique au lieu de la version console.
- L'ajout des contrôles nécessaires à un humain de jouer sur l'interface graphique.
- L'ajout d'outils d'analyses plus précis, performant et graphique.
- La résolution de diverses fuites de mémoires et bugs non-critiques.

VIII. Références

- Introduction, règles :
  - https://en.wikipedia.org/wiki/Chess
- Représentation des règles :
  - https://www.chessprogramming.org/Bitboard_Board-Definition
  - https://www.chessprogramming.org/Bitboard_Serialization
  - https://www.chessprogramming.org/Pawn_Attacks_(Bitboards)
  - https://www.chessprogramming.org/Knight_Pattern
  - https://www.chessprogramming.org/King_Pattern
  - https://www.chessprogramming.org/Subtracting_a_Rook_from_a_Blocking_Piece
  - https://www.chessprogramming.org/Hyperbola_Quintessence
  - https://www.chessprogramming.org/Flipping_Mirroring_and_Rotating
- Negamax :
  - https://en.wikipedia.org/wiki/Minimax
  - https://en.wikipedia.org/wiki/Negamax
  - http://www.cs.utsa.edu/~bylander/cs5233/a-b-analysis.pdf (preuve d'optimisation de l'élagage alpha-beta)
  - https://en.wikipedia.org/wiki/MTD-f
- Processus de décision markovien :
  - https://en.wikipedia.org/wiki/Markov_decision_process
  - https://en.wikipedia.org/wiki/Metric_space
  - https://en.wikipedia.org/wiki/Complete_metric_space
  - https://en.wikipedia.org/wiki/Cauchy_sequence
  - https://en.wikipedia.org/wiki/Banach_fixed-point_theorem
  - http://chercheurs.lille.inria.fr/~ghavamza/RL-EC-Lille/Lecture3.pdf (preuve de convergence)
  - https://web.archive.org/web/20181127083007/http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/DP.pdf (preuve de convergence)
- Q-Learning :
  - https://en.wikipedia.org/wiki/Q-learning
- XCS :
  - https://en.wikipedia.org/wiki/Learning_classifier_system
  - https://www.researchgate.net/publication/2937494_An_Algorithmic_Description_of_XCS