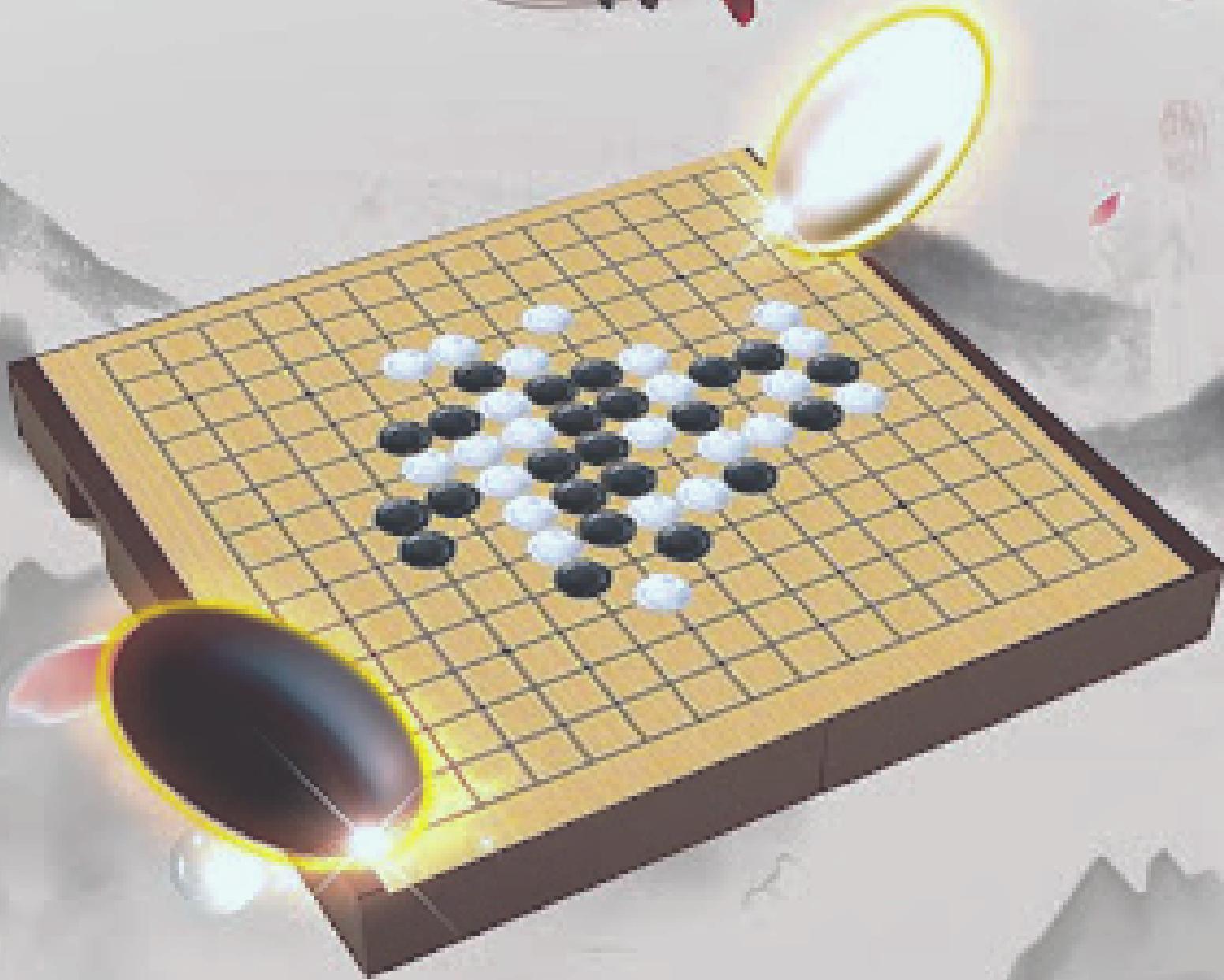


Rapport de fin de Projet: Algorithmique avancée: Gomoku

五子棋



五子棋研究

Prérequis :

Le gestionnaire, étant écrit en C, uniquement avec la librairies standard, aucun téléchargement supplémentaire ne devrait être nécessaire

Le projet a été testé sur 4 machines différentes(2 WSL, 2 linux), aucun n'a rencontré de problème à l'exécution, mais dans l'éventualité où cela peut arriver, voici une des configuration linux n'ayant eu aucun problèmes:

```
→ Downloads neotetch
      .';::::;'.
      .';:cccccccccccccc:;.
      .:cccccccccccccccccccccc:.
      .:cccccccccccccccccccccc:.
      .:cccccccccccccc:ddd:.;ccccccc:.
      .:cccccccccccccc:OWMKOOXMWd;ccccccc:.
      .:cccccccccccccc:KMMc;cc;xMMC:ccccccc:.
      ,cccccccccccccc:MMM..cc;;WW:;ccccccc,
      :cccccccccccccc:MMM..cccccccccccccc:.
      :ccccccc;ox000o;MMM000k.;ccccccccccc:.
      ccccccc:0MMKxdd:;MMMKddc.;ccccccccccc;
      ccccc:XM0';cccc;MMM.;cccccccccccccc' .
      ccccc;MMo;cccccc;MMW.;cccccccccccccc;
      ccccc;0MNc.ccc.xMMd:cccccccccccccc;
      ccccccc;dNMWXXWM0::cccccccccccccc:.
      ccccccc:.;odl:.;cccccccccccccc:;.
      .:cccccccccccccccccccccc:;.
      '':cccccccccccccc:;..
```

```
clem@fedora
-----
OS: Fedora Linux 38 (KDE Plasma) x86_64
Host: N350TW
Kernel: 6.5.8-200.fc38.x86_64
Uptime: 21 mins
Packages: 2409 (rpm)
Shell: zsh 5.9
Resolution: 1920x1080
DE: Plasma 5.27.8
WM: KWin
Theme: [Plasma], Breeze [GTK2/3]
Icons: [Plasma], breeze [GTK2/3]
Terminal: konsole
CPU: Intel i5-9400 (6) @ 4.100GHz
GPU: Intel CoffeeLake-S GT2 [UHD Graphics 630]
Memory: 4528MiB / 31796MiB
```

Guide d'utilisation et règles :

Compilation

- Un Makefile est fourni pour la compilation, utilisant les arguments Wall, Werror et fsanitize.

Exécution

- L'exécutable sera généré sous le nom gomoku, Vous devrez alors utiliser la commande ./gomoku
- un problème au niveau des codes couleurs pour l'affichage dans le terminal peut se présenter, d'après un camarade cela viendrait de %s au lieu de %c dans les printf, je n'ai jamais eu ce souci je ne peux donc pas savoir si cela sera nécessaire.

Pendant l'exécution du programme

- Je n'ai malheureusement pas eu le temps de vous proposer une interface graphique avec la librairie SDL, le programme va donc s'utiliser via le terminal
- Un choix vous sera proposé, il suffira d'entrer le chiffre correspondant pour exécuter la fonction de jeu attribué (1 pour le Joueur contre Joueur, 2 pour le Joueur contre IA ,etc...)

```
make && ./gomoku
```

Règles du jeu

- Le gomoku est un jeu très simple, il se joue sur un plateau quadrillé de 19 lignes horizontales et 19 lignes verticales formant, comme un jeu de go, 361 intersections. Le nombre de pions est toutefois nettement inférieur, puisque les joueurs n'en reçoivent que 60, qu'ils posent un à un et à tour de rôle sur les intersections.
- Le joueur qui a choisi ou obtenu par un tirage au sort les pions noirs (et que l'on appelle Noir par convention) joue toujours le premier en plaçant son premier pion sur l'intersection centrale du damier.
- le but du jeu étant de prendre l'adversaire de vitesse et de réussir le premier à aligner 5 pions de sa couleur, dans les trois directions possibles : vertical, horizontal ou diagonal.
- Si les deux joueurs placent tous leurs pions sans qu'aucun ne parvienne à réaliser un alignement, le jeu est déclaré nul, et ils recommencent. Une partie se dispute généralement en deux manches, afin que celui qui a commencé avec les noirs ait les blancs la seconde fois.

Sources : Wikipédia, <https://fr.wikipedia.org/wiki/Gomoku>

Présentation du projet :

Le but de ce projet est de réaliser un jeu de Gomoku, ainsi que d'implémenter une IA utilisant l'algorithme MinMax, aussi appelé MiniMax, puis son amélioration avec l'élagage alpha-bêta.

La première étape étant donc d'implémenter le jeu en lui même, une version Joueur contre Joueur est donc naturellement proposée.

Une version Joueur contre Environnement utilisant l'ia implémenté est aussi proposée.

De plus, l'algorithme minmax étant prévu pour les jeux à plusieurs joueurs, une version Environnement contre Environnement est aussi proposée, pour permettre l'affrontement de deux IA.

Le projet est scindé en 3 fichiers C et leurs en-tête :

- main.c : Il gère la sélection du mode de jeu et son exécution.
- gomoku.c : Ce fichier contient toutes les fonctions permettant le bon déroulement du jeu et gère les parties.
- minmax.c : Comme son nom l'indique, ce fichier contient



Explications du code :

main.c :

La fonction main n'est que très peu intéressante, elle régit simplement le choix du mode et de la taille du plateau via un switch et une condition. Nous n'avons pas besoin de nous attarder dessus.

Gomoku.c :

Ce fichier gère le déroulement des partie ainsi que les règles du jeu, les fonctions simple comme printGrid qui affiche la grille et IsValid qui vérifie qu'une case n'est pas vide/dans les limites du plateau ou déjà prise n'e mérite pas qu'on s'attarde dessus non plus.

Commençons par les définition en début de code,
EMPTY_CELL définie les case vide représenté par un point
PLAYER1 représente le pion noir par un N et PLAYER2 le pion blanc par un B

La fonction CheckWin :

- Comme son nom l'indique, cette fonction s'occupe de vérifier si les conditions de victoire sont atteintes.

```
bool checkWin(int GRID_SIZE, char grid[GRID_SIZE][GRID_SIZE], int row, int col, char symbol){  
    int directions[4][2] = {{1, 0}, {0, 1}, {1, 1}, {1, -1}};  
    for (int i = 0; i < 4; i++){  
        int count = 1;  
        int dx = directions[i][0], dy = directions[i][1];  
        int x = row + dx, y = col + dy;  
        while (x >= 0 && x < GRID_SIZE && y >= 0 && y < GRID_SIZE && grid[x][y] == symbol){  
            count++;  
            x += dx;  
            y += dy;  
        }  
        x = row - dx;  
        y = col - dy;  
        while (x >= 0 && x < GRID_SIZE && y >= 0 && y < GRID_SIZE && grid[x][y] == symbol){  
            count++;  
            x -= dx;  
            y -= dy;  
        }  
        if (count >= 5)  
            return true;  
    }  
    return false;  
}
```

Elle utilise un tableau contenant les direction cardinal, chaque couple représente un directe vers laquel vérifier si la condition de victoire 5 pions) est réuni, cette manière de faire est, à mon avis, plus intuitive et simplifie le code, et plus efficace, ce tableau est repris à plusieurs reprise dans le code

La fonction renvoie true si la victoire est vérifiée sinon, elle renvoie false et le jeu continue.

Les fonctions de mode de jeu :

- les fonction play_pve,play_pve_play_eve et leurs version avec élage suivent toutes le même principe, en expliquer une revient à toute les expliquer, je vais donc expliquer la fonction play_pve_elage, le fait qu'elle contient à la fois des entrées utilisateur et l'utilisation de l'ia, ce qui représente bien l'ensemble des fonctions.
- La première partie de la fonction se concentre sur le joueur, il s'agit toujours du joueur Noir, il doit donc jouer au centre du plateau, sur la taille classique 19×19 il s'agit de l'emplacement [9,9]
- Si un coup n'est pas valide il doit donc le rejouer, diminuant ainsi son sombre de pion et laissant l'ia (ou le joueur adverse) jouer.

```
while (!game_over){  
    printGrid(GRID_SIZE, grid);  
    if (turn % 2 == 1) {  
        //tour du joueur (Noir)  
        printf("Votre tour (entrer dans cette ordre : Ligne, Colonne ): ");  
        scanf("%d %d", &row, &col);  
  
        if (turn == 1 && (row != GRID_SIZE / 2 || col != GRID_SIZE / 2)){  
            printf("\033[1;31m Le premier pion doit être joué au centre, soit %d,%d\033[1;0m\n",  
            GRID_SIZE / 2, GRID_SIZE / 2);  
            continue;  
        }  
  
        if (!isValid(row, col, GRID_SIZE, grid)){  
            printf("\033[1;31m Vous ne pouvez pas jouer ici.\033[1;0m\n");  
            continue;  
        }  
        grid[row][col] = PLAYER1;  
        nbrPion++;  
        if (checkWin(GRID_SIZE, grid, row, col, PLAYER1)){  
            printf("Félicitations ! Vous avez gagné.\n");  
            game_over = true;  
        }  
    }
```

```

} else {
    // tour de l'ia (Blanc)
    printf("Tour de l'Ordinateur\n");
    int MoveRow, MoveCol;
    minimax(GRID_SIZE, grid, 3, false, PLAYER2, &MoveRow, &MoveCol, turn, INT_MIN, INTMAX);
    grid[MoveRow][MoveCol] = PLAYER2;
    nbrPion++;

    if (checkWin(GRID_SIZE, grid, MoveRow, MoveCol, PLAYER2)){
        printf("L'Ordinateur a gagné.\n");
        game_over = true;
    }
    turn++;
}

if (turn % 2 == 0) ? PionJ1-- : PionJ2--;
if (PionJ1 == 0 && PionJ2 == 0){
    printf("\033[1;32mPlus de pions, aucun joueur n'a gagné, Egalité !\033[1;0ms");
    game_over = true;
}
}
}
}

```

- La seconde partie de la fonction gère le tour de l'IA, il est déclaré deux entier qui serviront de position pour placer le pion.
- Elle appelle ensuite la fonction minimax, que nous étudirons plus tard dans ce rapport, qui va déterminer l'emplacement idéal où placer le pion de l'IA. diminuant ainsi le nombre de pion de l'IA et vérifiant si elle a gagné.

Si aucun joueur n'a gagné, alors la fonction vérifie ensuite si les deux joueurs ont encore des pions à jouer, si non la partie se fini sur un match nul.

Pourquoi utiliser un tableau static plutôt qu'une structure ? (liste chaîné ou autre) :

Étant donné que le plateau n'a pas besoin d'être alloué dynamiquement, le tableau statique représente la solution la plus performante et simple d'utilisation. Utiliser une structure serait en réalité moins performante que le tableau statique dû à toute les opération nécessaire.

J'ai cependant envisagé l'utilisation d'une structure similaire au vecteur de vecteur présente en C++ (`vector<vector<char>`).

minmax.c :

Ce fichier est celui qui nous intéresse principalement, il contient notre IA utilisant l'algorithme MinMax.

Cette partie sera plus longue que les précédentes, les fonctions contenues dans ce fichier nécessitant toutes des explications. (en dehors de la fonction ia_de_fou)

```
bool isNear(int GRID_SIZE, char grid[GRID_SIZE][GRID_SIZE], int row, int col) {  
    for (int i = -3; i <= 3; i++) {  
        for (int j = -1; j <= 1; j++) {  
            if (i == 0 && j == 0) continue;  
            int x = row + i;  
            int y = col + j;  
            if (x >= 0 && x < GRID_SIZE && y >= 0 && y < GRID_SIZE && grid[x][y] !=  
EMPTY_CELL) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

Cette fonction permet de vérifier si un pion se situe dans un rayon de 3 cases d'une case vide, si oui alors la fonction renvoie true et la case sera évaluée, sinon la case sera passée dans l'algorithme minmax.

Cette fonction permet de gagner énormément de performance en temps en réduisant la plage de recherche, les cases vides étant éloignées des pions n'ont que peu d'impact sur le jeu, cependant, plus la partie avance, moins elle à d'effet étant donné que le plateau se remplit, à partir d'un certains points, elle peut même réduire légèrement les performances, en effet quand le plateau sera rempli, cela rajoute une évaluation de plus, mais elle augmente plus les performances qu'elle ne réduit dans son ensemble, c'est donc un très bon ajout.

La fonction eval, il s'agit de la fonction d'évaluation, ont peut la considérer comme la fonction al plus importante du projet, c'est elle qui va évaluer les case et permettre à MinMax de choisir la case au meilleur score.

Elle commence par une boucle qui pendant le début de la partie (ici fixé aux 10 premiers tours) va accorder un score plus élevé au centre du plateau, le centre du plateau étant évidemment un endroit a maîtriser en début de partie pour s'assurer une victoire.

la fonction midgame suit une logique similaire, après le nombre de tours définie dans la condition, l'IA va accorder plus de priorité au blocages des menaces ennemis. La gestion des menaces devient, au cours d'une partie, de plus en plus importante.

La boucle qui suit, parcourt chaque cellule du plateau, en faisant attention aux pions du joueur et de l'adversaire, effectuant une évaluation des alignements de pions. En examinant chaque direction possible à partir de chaque cellule. Pour chaque direction explorée, elle compte le nombre de pions alignés (dans count), les endroits ouver (dans open) et les zones bloqués (blocked). Cela permet d'identifier les opportunités en repérant les menaces adverses. De plus, en limitant la considération aux alignements avec minimum une ouverture ou et maximum un blocage, cela optimise l'évaluation en se concentrant sur les cases stratégiquement viables sur le plateau.

Pour finir, la fonction nuisance (appelé ainsi par le fait que mon IA se focalise énormément sur le blocage, étant ainsi une nuisance pour l'adversaire) Va uniquement s'occuper de la gestion des menaces ennemis, accordant un score selon le nombre de pion ennemis aligné et non bloqué. principalement au ligne de 3 et 4 qui représente le plus grand danger de défaite.

```

if (turn < 10) {
    int centre = GRID_SIZE / 2;
    for (int i = -2; i <= 2; i++){
        for (int j = -2; j <= 2; j++){
            if (grid[centre + i][centre + j] == symbol){
                score += 7;
            }
        }
    }
} else{
    midgame(GRID_SIZE,grid,adversaire,directions,&score); }

for (int i = 0; i < GRID_SIZE; i++) {
    for (int j = 0; j < GRID_SIZE; j++) {
        if (grid[i][j] == symbol || grid[i][j] == adversaire){
            for (int d = 0; d < 4; d++) {
                int dx = directions[d][0];
                int dy = directions[d][1];
                count = 0;
                open = 0;
                blocked = 0;

                for (int k = 0; k < 5; k++) {
                    int x = i + k * dx;
                    int y = j + k * dy;

                    if (!isValid(x, y, GRID_SIZE, grid)) break;
                    if (grid[x][y] == grid[i][j]){
                        count++;
                    } else if (grid[x][y] == EMPTY_CELL){
                        open++;
                        if (open > 1) break;
                    } else {
                        blocked++;
                        if (blocked > 1) break;
                    }
                }

                } if (count > 0 && grid[i][j] == symbol) score+= 5*count; }

    nuisance(GRID_SIZE, grid, adversaire, directions, &score);
}

```

Les fonctions MinMax et elage :

Ces deux fonctions sont très similaires, leurs seuls différence se résume à quelques arguments et conditions supplémentaire dans elage.
cependant, même si il y a peu de différence, l'élage alpha-bêta offre une compléxité en temps bien inférieur au minmax classique.
De ce fait nous n'allons étudier que la fonction elage, l'expliquer revient aussi à expliquer MinMax si l'on retire l'élage.

La fonction prend en argument la taille du plateau, le plateau, un booléen indiquant si il s'agit du joueur maximisant, du symbol du joueurs, deux pointeurs permettant de donner la position évalué comme la meilleur, le nombre de tour servant à la fonction d'évaluation ainsi que alpha et bêta initialisé respectivement à INT_MIN et INT_MAX

L'algorithme va passer en revu chaque, si elle est valide et qu'elle rentre dans les critère de IsNear (vu précédemment) alors il va effectuer le scorage de manière récursive.

Cette partie de la fonction n'utilise pas une copie du plateau pour l'évaluation, à la place elle va placer le pions, l'évaluer et l'enlever, le choix définitifs se faisant après avoir décidé quel emplacements serait le plus optimal. Cette façon de faire est très utilisé dans les jeux de plateau comme le Gomoku qui passe beaucoup en revu la grille qui est parfois très grande.

Pour résumer, cela optimise la complexité en temps et espace en évitant les copies.

Une part d'aléatoire est aussi implémenter, si l'évaluation actuelle est égal à celle précédente, alors il y a une chance sur 2 qu'elle change pour cette nouvelle positions, les score étant égaux, les positions ont la même probabilité de gagner. De cette façon, l'ia possède une part d'imprévisibilité et les partie sont différentes les unes des autres.

l'élage est ensuite réaliser pour pouvoir passer les positions qui ne sont pas prometteuse et ainsi optimiser les performances en temps de l'algorithme.

(pour ne pas rallonger plus que nécessaire le rapport, je n'affiche ici que la partie maximisante de l'algorithme, la partie minimisante se comportant de la même manière, mais cherche comme son nom l'indique à minimiser)

```
int elage(int GRID_SIZE, char grid[GRID_SIZE][GRID_SIZE], int depth, bool isMax, char player, int* bestRow, int* bestCol, int turn, int alpha, int beta, currentDepth) {
    if (depth == 0) {
        return eval(GRID_SIZE, grid, player, turn);
    }

    if (isMax) {
        int maxEval = INT_MIN;
        for (int row = 0; row < GRID_SIZE; row++) {
            for (int col = 0; col < GRID_SIZE; col++) {
                if (grid[row][col] == EMPTY_CELL && isNear(GRID_SIZE, grid, row, col)) {
                    grid[row][col] = player;
                    int eval = elage(GRID_SIZE, grid, depth - 1, false, player, bestRow, bestCol,
turn, alpha, beta);
                    grid[row][col] = EMPTY_CELL;

                    if (eval > maxEval || (eval == maxEval && rand() % 11 > 0)) {
                        maxEval = eval;
                        if (currentDepth == depth)
                            *bestRow = row;
                            *bestCol = col;
                    }
                }
            }
        }
        if (maxEval > alpha) {
            alpha = maxEval;
        }
        if (beta <= alpha)
            break;
        }
    }
    if (beta <= alpha)
        break;
    }
    return maxEval;
} else {
```

Point d'amélioration :

Mon projet souffre de plusieurs problèmes, et différents points d'améliorations sont possibles en voici une liste, qui n'est à mon avis pas exhaustive.

- L'IA se focalise bien trop sur le blocage, et ceux-malgré mes tentatives d'équilibrer le score en faveur de l'agressivité.
- L'IA ne prend pas en compte les placement dit "fork" c'est à dire les points stratégiques, parfois séparés par un ou plusieurs espaces vides, permettant de réaliser des alignement multiple de 5
- L'IA ne prend pas très bien en compte les alignement faisable si il y a des espaces entre les pions
- Mon élage alpha-beta est étonnamment moins efficace que mon minmax classique, non pas sur les performances en temps, mais sur le jeu en lui-même, l'IA joue principalement en diagonale (sur la première case respectant les conditions de IsNear à mon avis). Une fois le haut du plateau atteint elle semble rejouer normalement à partir de là. alors qu'il n'y a pas ce soucis dans le minmax classique.

Le problème peut venir de deux endroits, soit mon implémentation de alpha-bêta est mal réalisé (ce qui est possible j'ai eu du mal à le comprendre) ou alors ma fonction d'évaluation, c'est aussi probable étant donné qu'elle souffre de plusieurs problèmes et lacunes. Cependant, étant donné que ma fonction d'évaluation fonctionne plutôt bien dans le minmax classique, il est plus probable que le bug viennent de mon implémentation de l'élage.