Fil Rouge-compte-rendu

Sommaire

I. Introduction

Le but de ce fil rouge est de coder un super-morpion fonctionnel, capable de jouer contre un autre super-morpion en choisissant des coups optimaux. Pour ce faire nous avons décidé de voir ce dernier comme 9 morpions simple qu'on pourra traiter avec quelques règles de jeux en plus. C'est pourquoi dans une première partie nous implémenteront les outils de gestion du morpion simple et ensuite nous les étendront au super-morpion.

II. Morpion simple

A.DÉTAIL DU CODE posGraph.c

Ce fichier et le code qu'il contient sont le coeur de notre approche pour traiter le problème du morpion. Il permet de gérer les positions sur le plateau de jeu notamment par l'utilisation d'une chaîne de caractère plus simple à manipuler pour nous. En effet, les positions du jeu sont décrites par les chaînes de caractère fen.

Cependant ces dernières ne sont pas toutes de la même taille. On a donc décidé de remplacer toute occurence de chiffre par un nombre équivalent de point. L'important dans ce code est de pouvoir stocker l'état de la partie, l'état précédent et le joueur devant jouer. Pour cela on crée posGraph.
PosGraph est une structure qui contient la position du tableau, la position précédente (utile pour savoir dans quel morpion on joue dans le super-morpion) et le joueur qui doit jouer.

```
typedef struct posGraph {
    char pos[MAXLEN];
    char joueur;
    char last[MAXLEN];
} posGraph;
```

Figure 1 : *définition de la structure posGraph*

•fenToPosGraph(char* fen)

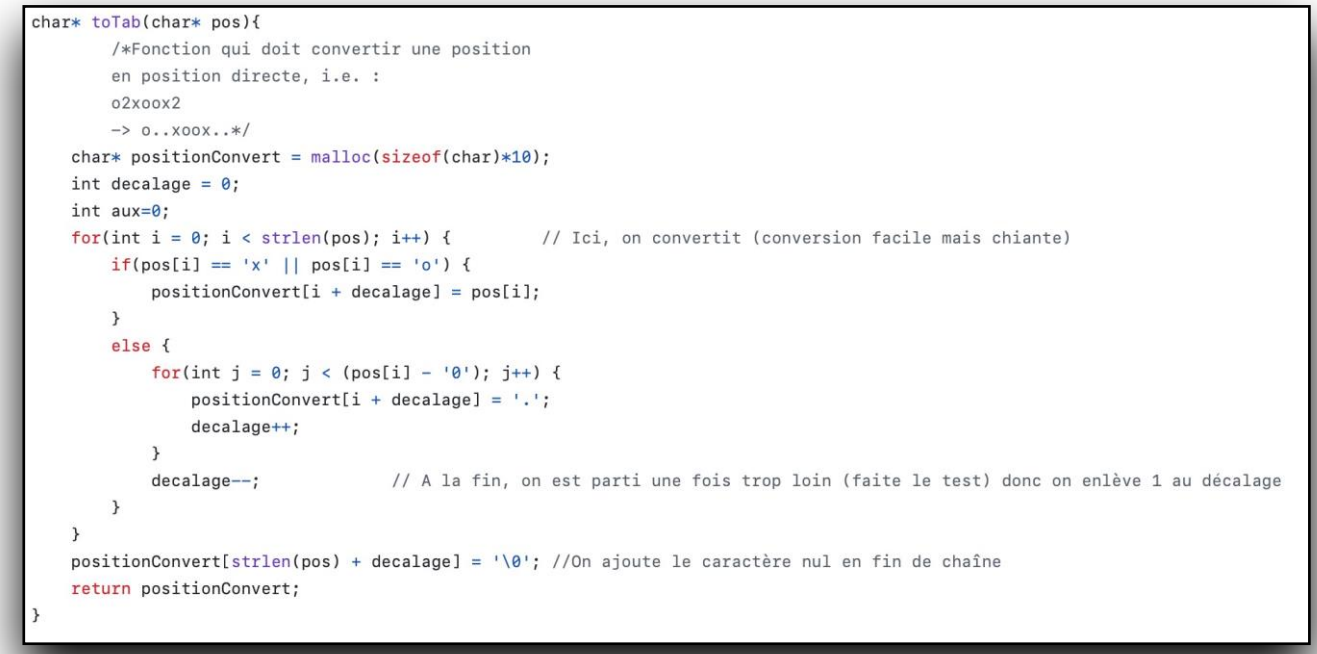
Cette fonction prend en argument un **posGraph**. Son objectif est de pouvoir retourner la position dans laquelle se trouve le jeu grâce à un **fen** passé en entrée. Pour cela on vient créer une structure **posGraph position** qui va être initialisé avec la fonction **strcpy( )** avec le contenu du **fen**. Nous verrons par la suite que **strcpy()** pose des problèmes notamment en terme d'allocation de mémoire. Enfin, pour éviter tout problème on vient mettre un caractère de retour à la ligne dans **position.joueur.**

•NewPosition(char* pos, char sigleJoueur)

Cette fonction prend en argument la position simplifié du jeu et le sigle du joueur devant jouer. Elle permet seulement de créer un **posGraph** contenant toute les informations nécessaires.

•toTab(char* pos)

Pour simplifier le problème, nous avons eu l'idée de changer le formalisme d'un fen. Ainsi, au lieu d'avoir des chiffres indiquant le nombre de cases vides (dans le bon ordre), nous préférons manipuler des points, pour éviter tout problèmes liés à la conversion des int en char et de longueurs de chaînes. Grâce à la position direct, nos chaînes contenant toute l'information d'un plateau de jeu on la même longueur.
Ainsi, nous avons besoin d'une variable permettant de mesurer le décalage en longueur à chaque rencontre d'un chiffre comptant initialement pur une unité de longueur et comptant maintenant pour sa valeur. En effet, si nous sommes à l'occurence **i** de la boucle et qu'on rencontre un 2, on va mettre à la position **i** un point et également en i+1. Cependant l'occurence suivante de la boucle sera **i+1**, créant ainsi un problème.
Puis on distinguer tous les cas: un chiffre devient sa valeur en nombre de point, un **'x'**et un **'o'** restent identiques.
Pour indiquer la fin de la chaîne, on rajoute le caractère **'\0'** servant de condition d'arrêt.



```c
char* toTab(char* pos){
        /*Fonction qui doit convertir une position
        en position directe, i.e. :
        o2xoox2
        -> o..xoox..*/
    char* positionConvert = malloc(sizeof(char)*10);
    int decalage = 0;
    int aux=0;
    for(int i = 0; i < strlen(pos); i++) {         // Ici, on convertit (conversion facile mais chiante)
        if(pos[i] == 'x' || pos[i] == 'o') {
            positionConvert[i + decalage] = pos[i];
        }
        else {
            for(int j = 0; j < (pos[i] - '0'); j++) {
                positionConvert[i + decalage] = '.';
                decalage++;
            }
            decalage--;             // A la fin, on est parti une fois trop loin (faite le test) donc on enlève 1 au décalage
        }
    }
    positionConvert[strlen(pos) + decalage] = '\0'; //On ajoute le caractère nul en fin de chaîne
    return positionConvert;
}
```

Figure 2 : *fonction toTab*

- **toPosGraph(char* positionC)**

Cette fonction a uniquement pour but de proposer une réciproque à la fonction **toTab( )**. **morpion.c**

Ce fichier à pour vocation de gérer le bon déroulement d'une partie de morpion simple. Il utilise notamment les fonctions de gestions vues précédemment et vient proposer de nouvelles fonctions liées aux déroulement de la partie.

- **Game(posGraph position)**

La fonction game est une fonction qui permet de faire le lien avec l'utilisateur. Elle gère la partie. À partir d'un posGraph, elle donne la position actuelle dans laquelle se trouve le morpion elle utilise notamment la fonction **showTable ( )** pour afficher l'état du morpion. Ensuite, elle utilise seulement i**sNodeTerminal( )** pour savoir si la partie est terminée ou non. Si non, elle demande le déroulement d'un nouveau tour de jeu grâce à **Tour( )**. Lorsque la boucle **while** est terminé, la partie l'est également. Si on a une égalité **Game( )** ne renvoie rien du tout, en revanche si il y a un gagnant, l'identité du perdant est contenue dans le **posGraph** dernièrement modifié. En effet, le dernier joueur ayant joué à crée une position terminale mais avant que celle-ci soit détectée, le **posGraph** a été mis à jour contenant le prochain joueur devant jouer. Ainsi, il suffit d'échanger **position.joueur** pour déterminer le gagnant.

```c
void Game(posGraph position){                //Fonction qui gère la partie
    printf("Voici la position actuel : %s\n", position.pos );
    printf("La table du morpion est la suivante : \n");
    showTable(position);
    char c = 'o';        //Le premier joueur a jouer est o
    while(isNodeTerminal(position) != 1){
        Tour(&position);
    }
    if(position.joueur=='o') position.joueur='x';
    else position.joueur='o';
    printf("La partie est finie. Le vainqueur est %c\n", c);
}
```

Figure 3 : *fonction Game*

•showTable (posGraph *position)

Cette fonction permet d'avoir un affichage graphique dans le terminal du morpion. On vient convertir en chaîne directe la position du morpion contenue dans un **posGraph( )**. Ensuite, on vient afficher graphiquement chaque 'x', 'o' ou '.'. Pour avoir l'affichage 3x3, il suffit de revenir à la ligne à chaque fois que la position dans la chaîne du caractère considéré est un multiple de 3. Comme la chaîne commence en C à 0, il faut faire attention au décalage d'indice.

•isNodeTerminal(posGraph position)

On essaye ici de savoir si une position est terminale ou non. Pour ce faire, on va manipuler des chaînes directe et analyser tous les cas possibles :

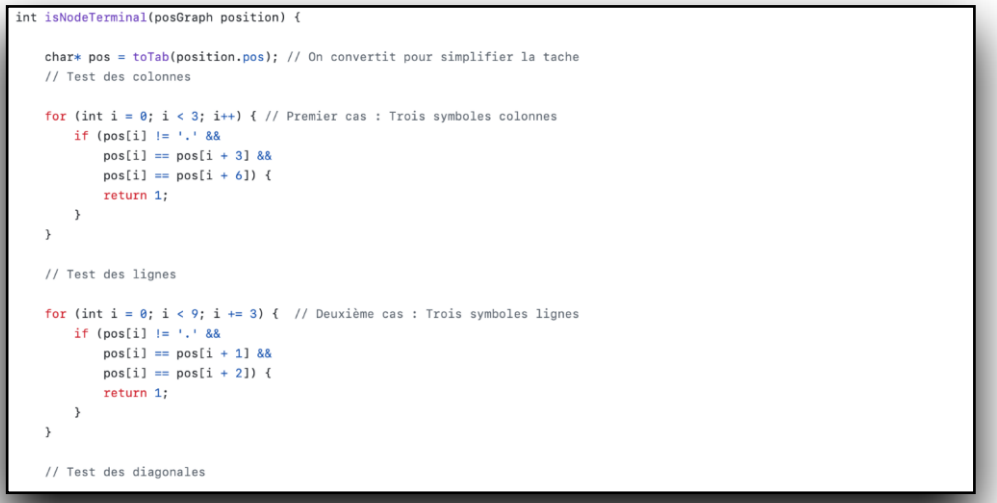
Cas n°1, on a une colonne avec le même caractère (**≠'.'**): la position est terminale et on renvoie 1. On boucle sur le nombre de colonne *ie* 3.

Cas n°2, on a une ligne avec le même caractère (**≠'.'**): la position est terminale et on renvoie 1. On boucle sur le nombre de ligne *ie* 3.

Cas n°3 et n° 4, les diagonales contiennent un seul et même caractère : la position est terminale et on renvoie 1. Pour des soucis de facilité, on traite les deux cas séparément

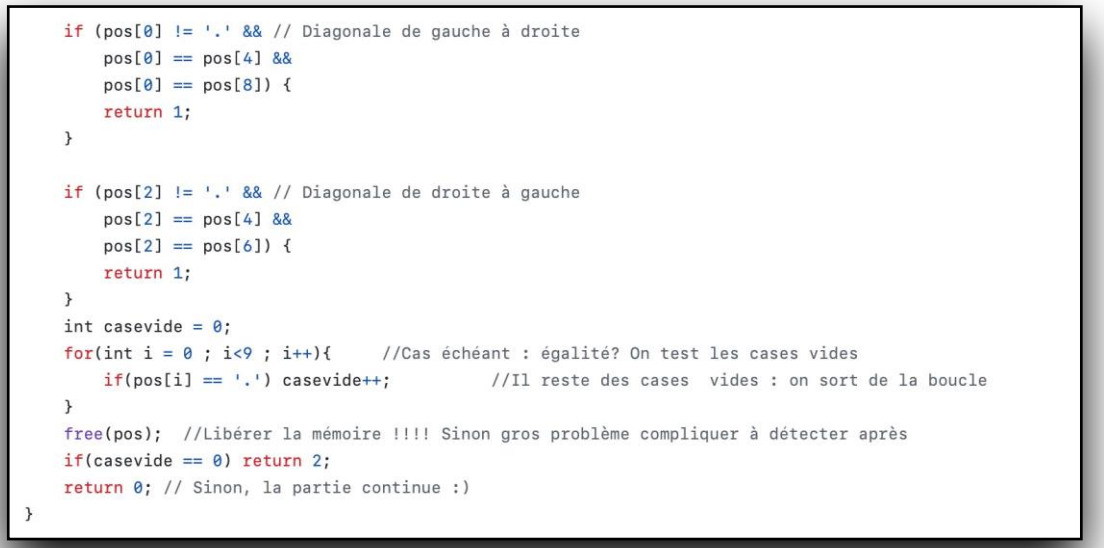
Cas n°5: il n'y a plus de cases vides et la position n'est pas terminale. On renvoie 2, symbole d'une égalité.

Cas n°6, les conditions précédentes ne sont pas vérifiées: la position n'est pas terminale est on renvoie 0, la partie n'est pas finie.



```c
int isNodeTerminal(posGraph position) {

    char* pos = toTab(position.pos); // On convertit pour simplifier la tache
    // Test des colonnes

    for (int i = 0; i < 3; i++) { // Premier cas : Trois symboles colonnes
        if (pos[i] != '.' &&
            pos[i] == pos[i + 3] &&
            pos[i] == pos[i + 6]) {
            return 1;
        }
    }

    // Test des lignes

    for (int i = 0; i < 9; i += 3) {  // Deuxième cas : Trois symboles lignes
        if (pos[i] != '.' &&
            pos[i] == pos[i + 1] &&
            pos[i] == pos[i + 2]) {
            return 1;
        }
    }

    // Test des diagonales
```

Figure 4 : *fonction isNodeTerminal*



```c
    if (pos[0] != '.' && // Diagonale de gauche à droite
        pos[0] == pos[4] &&
        pos[0] == pos[8]) {
        return 1;
    }

    if (pos[2] != '.' && // Diagonale de droite à gauche
        pos[2] == pos[4] &&
        pos[2] == pos[6]) {
        return 1;
    }
    int casevide = 0;
    for(int i = 0 ; i<9 ; i++){     //Cas échéant : égalité? On test les cases vides
        if(pos[i] == '.') casevide++;          //Il reste des cases  vides : on sort de la boucle
    }
    free(pos);  //Libérer la mémoire !!!! Sinon gros problème compliquer à détecter après
    if(casevide == 0) return 2;
    return 0; // Sinon, la partie continue :)
}
```

Figure 5 : *suite de la fonction isNodeTerminal*

•coupToInt(char* coup)

Les instructions de l'utilisateur seront données aux fonctions par les lettres **'a'**, **'b'**, **'c'** pour les colonnes et **1**, **2**, **3** pour les lignes . Ainsi, cette fonction permet juste de convertir l'identifiant de la case dans laquelle on a voulu jouer en un numéro de case allant de 1 à 9.

•intToCoup(int position)

Cette fonction réalise seulement la réciproque de **coupToInt( )**. En effet, le joueur choisit juste la case avec un chiffre de 1 à 9.

•Tour(posGraph *position)

Tour( ) est une fonction permettant de gérer toutes les actions lié au jeu d'un joueur. Cette fonction manipule des chaînes directes et demande en premier lieu au joueur où ce dernier veut jouer. Une fois l'instruction rentrée, on vient vérifier que la position est bien libre : **'.'**. Si c'est le cas, on met à la bonne position dans la chaîne directe (la chaîne commence à 0, d'où le décalage d'indice). Pour choisir le caractère à mettre dans la chaîne on regarde seulement le joueur devant jouer dans **position.joueur**. Puis enfin on remodifie l'état de la partie dans le **posGraph position.pos**. Si on ne peut pas jouer dans la case choisit, la fonction renvoie un message d'avertissement et ne change rien. Ce cas-là est bien traité par la condition dans la boucle **while** de **Game( ).**

•TourAuto(char *position, char*coup, char sigleJoueur)

Cette fonction permet de réaliser le coup rentré par un joueur en mettant à jour la position du morpion. Elle prend en entrée toutes les données nécessaires pour actualiser le morpion.

•decisionTree(posGraph position, int parentID, int moveID, int isJoueurAuTrait)

Cette fonction récursive permet de trouver le meilleur coup et de traiter toutes les possibilités du morpion. Son cas de base est là terminaison de la partie, c'est à dire si la fonction **isNodeTerminal** renvoie 1 ou 2. Dans le premier cas, la partie est gagné on renvoie une valeur très grande, dans le second cas on renvoie 0. Si la partie n'est pas terminé on va venir parcourir toutes les cases vides du morpion et rappeler la fonction autant de fois qu'il y'a de cases vide avec à chaque une de ces dernières remplies par le joueur au trait. Puis on vient changer le signe de la valeur renvoyée car le minimax cherche à maximiser notre coup et à minimiser celui de l'adversaire. Il s'agit de l'algorithme minimax usuel.

```
int decisionTree(posGraph position, int parentID, int moveID, int isJoueurAuTrait) {
    int val = -10000; //Par convention, cette valeur est -infini
    if(parentID==-1) nombreDeNoeud++;       //On traite le cas particulier du premier coup
    nombreDeNoeud++;
    if (isNodeTerminal(position) != 0) {
        if(isNodeTerminal(position) == 2) {
            addEvaluation(moveID,0);
            return 0;
        }
        addEvaluation(moveID,-100);
        return -100;
    } else {
        for (int i = 0; i < 9; i++) {
            char* positionConvert = toTab(position.pos);
            if (positionConvert[i] == '.') {

                // On copie la position actuelle dans un nouveau tableau
                char newPositionPos[9];
                strcpy(newPositionPos, positionConvert);

                // On effectue les modifications nécessaires
                newPositionPos[i] = position.joueur;

                // On créé une nouvelle structure avec le tableau modifié
                posGraph newPosition;
                strcpy(newPosition.pos, toPosGraph(newPositionPos));

                if (position.joueur == 'o') {
                    newPosition.joueur = 'x';
                } else {
                    newPosition.joueur = 'o';
                }
```

Figure 6 : *fonction DecisionTree*

```
                if (parentID == -1) {
                    // On appelle récursivement la fonction avec la nouvelle position
                    addNode(newPosition, 0, nombreDeNoeud,1-isJoueurAuTrait);
                    val = max(val,-decisionTree(newPosition, 0, nombreDeNoeud,1-isJoueurAuTrait));
                } else{
                    addNode(newPosition, moveID, nombreDeNoeud, 1-isJoueurAuTrait);
                    val = max(val, -decisionTree(newPosition, moveID, nombreDeNoeud,1-isJoueurAuTrait));

                }
            }
            free(positionConvert);
        }
        addEvaluation(moveID, val);
        return val;
    }
}
```

Figure 7 : *suite de la fonction DecisionTree*

•makeDecisionTree(posGraph position)

Elle utilise la fonction précédente et le fichier **DOT** pour afficher l'arbre des possibilités.

B. JEUX DE TESTS

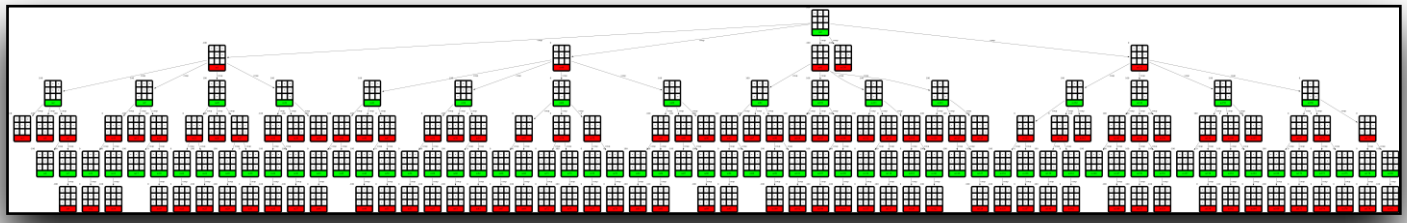


Figure 8 : *arbre de décision*

On retrouve bien l'arbre des possibilités total pour un morpion simple.
Dans un deuxième test, on donne la position de départ suivante à notre bot: « oxx1xo3 »

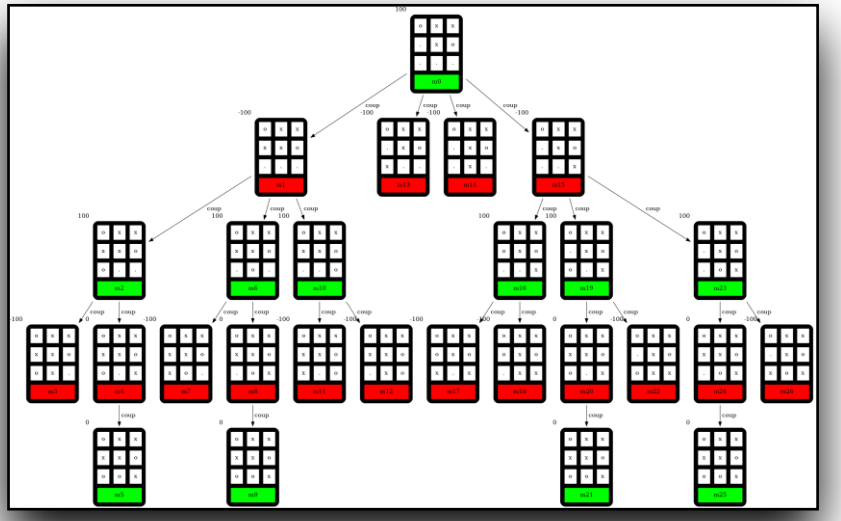


Figure 9 : *arbre de décision avec évaluation*

L'arbre des évaluations est alors cohérente avec le cours

C. DIFFICULTÉS RENCONTRÉES

La principale difficulté rencontrée a été de savoir bien énuméré les morpions pour bien les relier dans le .dot. En effet, il faut bien que l'énumération parcours chaque branche en entier pour les nommer. Sinon, les liens entre les différentes positions se font dans tous les sens.

III.Super-morpion

Notre super morpion s'inspire fortement du fonctionnement du morpion de base. Nous avons tenté de réadapter les règles de ce dernier au super-morpion. Ainsi, nous considérons le super-morpion comme un ensemble de 10 morpions simples et nous pouvons alors utiliser les fonctions de gestions créées précédemment.

A. DÉTAIL DU CODE

posGraphUltimate.c

•CoupOpti (posGraph position, int parentID, int moveID, int isJoueurAuTrait)

CoupOpti permet de récupérer des informations sur le coup optimal à jouer dans un morpion passé en paramètre. Par un raisonnement récursif, cette fonction renvoie l'évaluation maximale du morpion mais s'occupe surtout de sauvegarder dans un tableau le meilleur coup à jouer (celui avec la plus grande évaluation).

•FenToPosGraph (char* fen)

Cette fonction permet de transformer une chaîne Fen représentant l'état du super morpion en une structure de données **posGraphUltimate** utilisée pour représenter la position actuelle du jeu. Pour la créer nous avons adapté l'implémentation de la même fonction pour le morpion en tenant des contraintes du super morpion. Pour ce faire on parcourt la chaîne fan et on s'intéresse d'abord aux cas où un sous morpion est gagné soit par o soit par x. Dans ce cas on initialise la sous grille avec que des « o » ou que des « x » selon le gagnant puis on met à jour un tableau de caractères temporaires alloué dynamiquement (initialisé par des « . ») pour stocker l'état du morpion principal. Si aucun joueur n'a gagné la sous grille, on parcourt la chaîne pour convertir la sous grille en une structure **posGraph**. Enfin, on copie le morpion principale dans la structure position après l'avoir convertit en **posGraph**. **morpion_to_S.c**

•GameUltimate (posGraphUltimate position)

La fonction **GameUltimate**, par analogie avec la fonction **Game** du morpion simple, fait le lien avec l'utilisateur et gère la partie. Cette fois ci, à l'aide d'une boucle elle affiche les grilles de manière séparé les sous grilles du super morpion ainsi que la grille du morpion principal à la fin. Elle indique à quelle profondeur l'algorithme va travailler. Tant que la partie n'est pas finie, elle transmet dans un fichier dot l'état du morpion puis fait jouer un joueur à l'aide de la fonction **TourUltimate**. La nouvelle position est mis à jour et c'est à l'autre joueur de jouer avec le second appel à la fonction TourUltimate. On enregistre les modifications de ce joueur et on indique que c'est à l'autre de jouer. On affiche alors le morpion principal suite au coup joué par les deux joueurs et on répète cela jusqu'à la fin de partie. Lorsque le jeu est terminé, on génère un fichier dot décrivant l'état final du morpion et on indique le gagnant.

```c
void GameUltimate(posGraphUltimate position){
    for(int i=0; i<9;i++){
        printf("///*****%d*******///\n",i+1);
        showTable(position.morpion[i]);
    }
    char* dernier_coup = calloc(10,sizeof(char));
    if(DEBUG == 1){
        printf("/********** Entrée dans le mode DEBUG **********/\n");
        printf("/*********** Profondeur choisie : %d ***********/\n", profondeur );
    }
    while(isNodeTerminal(position.morpion[9]) !=1){
        writeMorpionUltimate(position);
        do{
            position = TourUltimate(position,&dernier_coup);
        }while(COUP_DEJA_JOUE ==1); //On teste le cas où le position est déjà jouée
        writeMorpionUltimate(position);
        position.joueur = (position.joueur == 'o' ? 'x' : 'o');
        position = TourUltimateBot(position,&dernier_coup);
        position.joueur = (position.joueur == 'o' ? 'x' : 'o');
        printf("Voici le morpion principal : \n");
        showTable(position.morpion[9]);
    }
    writeMorpionUltimate(position);
    printf("Bravo, la partie est finie\n");
}
```

Figure 10 : *fonction GameUltimate*

•TourUltimateBot (posGraphUltimate position, char*pointeur_last_coup[])

Ce programme permet à l'ordinateur de jouer le meilleur coup en fonction de l'état actuel du jeu. Il met également à jour la position. Il indique tout d'abord que c'est au tour du bot de jouer et note la grille dans laquelle il doit jouer en fonction du dernier coup. De manière évidente si la partie est finie ou nulle, il renvoie la position. Si dans la sous grille la partie n'est pas finie, on parcourt chacune des cases des morpions non complétés pour retenir le coup ayant une évaluation maximale.
Le programme met à jour la position du morpion suite au coup joué.

```c
posGraphUltimate TourUltimateBot(posGraphUltimate position, char* pointeur_last_coup[]) {
    printf("Au tour du bot : \n");
    int maximum = -INFINI;
    int grille = coupToInt(*pointeur_last_coup);
    if(isNodeTerminal(position.morpion[9]) != 0){
        return position;
    }
    if (isNodeTerminal(position.morpion[grille-1]) != 0) {
        int coupSave[9];
        int evaluationSave[9];
        CoupOpti(position.morpion[9], -1, 0, 0, coupSave, evaluationSave,0.9);
        for(int i = 0;i<9;i++){
            if(evaluationSave[i]>maximum){
                grille = coupSave[i]+1;
                maximum = evaluationSave[i];
            }
        }
    }
    maximum = -INFINI;
    int coup_a_jouer[3];
    char joueur_temp = position.joueur;
    evaluation(position, grille, profondeur, 1, coup_a_jouer,1);
    position.joueur = joueur_temp;
    if(DEBUG == 1) printf("\nCoup joué par le bot: %d\n", coup_a_jouer[0] + 1 );
    strcpy(position.morpion[grille - 1].pos, TourAuto(position.morpion[grille - 1].pos, intToCoup(coup_a_jouer[0] + 1), position.joueur));

    strcpy(*pointeur_last_coup, intToCoup(coup_a_jouer[0] + 1));

    if (isNodeTerminal(position.morpion[grille - 1]) == 1) {
        char* positionConvert = toTab(position.morpion[9].pos);
        if (position.joueur == 'o') positionConvert[grille - 1] = 'o';
        else positionConvert[grille - 1] = 'x';
        strcpy(position.morpion[9].pos, toPosGraph(positionConvert));
        free(positionConvert);
    }

    for (int i = 0; i < 9; i++) {
        printf("///------%d-------///\n", i + 1);
        showTable(position.morpion[i]);
    }
    return position;
}
```

Figure 11 : *fonction TourUltimateBot*

•Evaluation (posGraphUltimate position, int coup_precedent, int horizon, int coup_a_jouer, int joueurautrait)

Le programme permet de de gérer le temps restant et l'évaluation du meilleur coup. Il vérifie le temps restant pour prendre des décisions en se basant sur les limites de temps fixées. Si le temps restant dépasse une certaine limite, le programme retourne une valeur minimale (représentant une condition d'échec ou de limite de temps). Ensuite, le programme initialise les variables pour stocker les meilleurs coups et évaluations. Si ce n'est pas la première itération on change le joueur qui doit jouer. Si l'itération est égale à la profondeur voulue, l'algorithme évalue les coups possibles pour le coup précédent. Il trouve le coup le plus optimal en

fonction des évaluations et le sauvegarde. Si ce n'est pas la dernière itération, explore les possibilités de coups et leurs évaluations. Pour chaque coup possible : il simule le coup dans la position actuelle, calcule récursivement la valeur de l'évaluation pour cette nouvelle position, évalue le coup potentiel en combinant l'évaluation du coup, l'évaluation du coup précédent et l'évaluation de la position. Enfin il retourne la meilleure évaluation trouvée lors de l'itération et le sauvegarde.

•TourUltimate(posGraphUltimate position, char* pointeur_last_coup[ ])

La fonction permet de gérer un tour de jeu. On distingue deux cas : celui où l'on est en début de partie et celui où un coup a déjà été joué. Pour commencer, on vérifie si c'est le début de la parti ou on veut jouer dans un morpion fini (c'est à dire que si le dernier coup joué vaut 0). On demande au joueur de rentrer la grille, la colonne et la ligne au format « 9 b 2 » si il veut jouer dans la sous grille 9, 2ème colonne et 2ème ligne par exemple. On vérifie si la position demandée est déjà occupée ou non. Si ce n'est pas la cas, on enregistre la nouvelle position du morpion et on remplace le dernier coup joué par celui-ci. Le programme vérifie ensuite si le coup a permis au joueur de gagner la grille, si c'est le cas le morpion principal est mis à jour. La fonction affiche ensuite l'état du morpion. Ensuite, si un coup a déjà été joué : on regarde si la grille correspondant au dernier a été gagné et le joueur peut alors jouer où il veut sinon la grille lui est imposé. Dans ce cas on demande au joueur où il souhaite jouer dans la grille. La programme s'assure que le coup est permis et si c'est le cas il met à jour la grille. Comme précédemment il vérifie si le coup à permis au joueur de gagner la grille. Enfin il affiche l'état du super morpion.

gestionGraphUltimate.c

•writeMorpionUltimate (posGraphUltimate positionU

Le programme permet de générer un fichier au format **DOT** pour représenter graphiquement un tableau de jeu du morpion en utilisant les informations de la structure **posGraphUltimate**. Pour chaque cellule de morpion, il vérifie si la cellule est gagnante. Si c'est le cas, il crée une représentation graphique de la cellule en utilisant une table **HTML** à l'intérieur du fichier **DOT**. Ensuite, il remplit la cellule avec une couleur correspondant au joueur ayant gagné cette cellule ('o' pour blanc, 'x' pour noir). Si la cellule n'est pas gagnante, le programme convertit la position de la cellule en tableau pour afficher le contenu de la cellule. De la même manière, il crée une représentation graphique. Enfin il emplit la cellule avec le contenu du tableau représentant la position du morpion.

DEBUG et SMPATH

Le programme doit être sensible aux différentes variables d'environnement DEBUG et SMPATH comme précisé dans le cahier des charges. C'est donc ce que nous avons implémenté en dernier. Le mode débug affiche notamment les évaluations des coups testés mais aussi la position du bot jouée et pleins d'autres informations pour vérifier si le programme fonctionne bien :

```
/********** Entrée dans le mode DEBUG **********/
/*********** Profondeur choisie : 2 ***********/
```

Ici par exemple, en initialisant la variable d'environnement DEBUG, un texte apparaît avec la précision de la profondeur choisie, ce qui peut-être très utile.

B. JEUX DE TESTS

Lancement Jeu en 1v1 avec un bot :

Le jeu de morpion est vide et le joueur o est invité à jouer. Le joueur décide de joueur dans la grille, deuxième colonne, deuxième ligne. Pour cela il a tapé « 9 b 2 ». C'est ensuite au tour du bot de jouer. Il est contraint de jouer dans la grille 5. On s'aperçoit qu'il a joué « 5 c 3 ». C'est ensuite à nouveau au joueur o de jouer dans la grille 9 compte tenu du coup précédent.

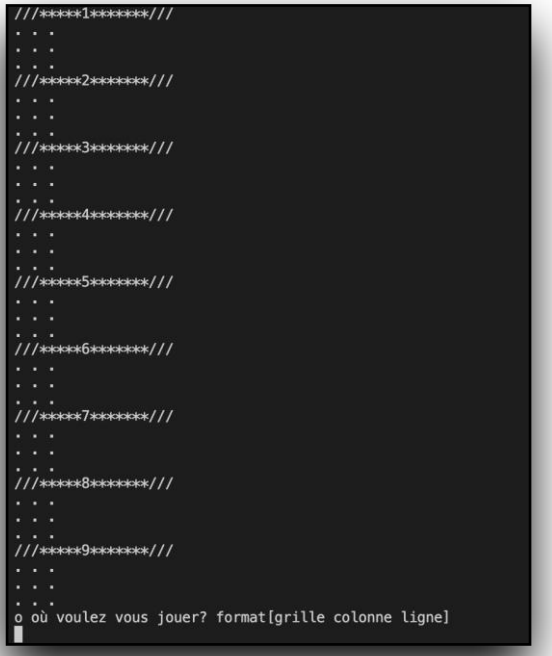


Figure 12 : *lancement de la partie*

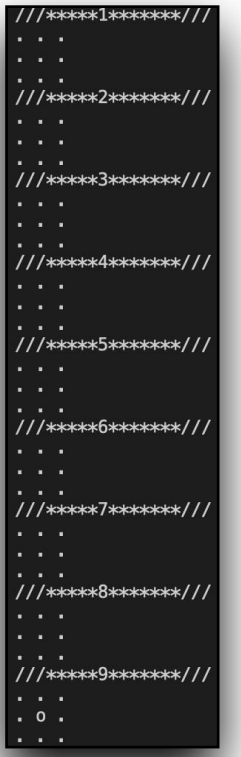


Figure 13 *: affichage du morpion après avoir joué « 9 b 2 »*

Figure 14: *affichage du coup du bot et du morpion principal*

Jeu avec un bot 1v1 lors d'une partie en cours :

On démarre le morpion avec une position déjà établie : "6xoxOOOX2xo1ox1oXx2xo4oox4ox o". D'après la chaîne Fan c'est au joueur o de jouer et le terminal propose à celui-ci de rentrer son coup. Il peut jouer où il

veut car le dernier coup l'envoyait sur une grille complète. Celui-ci décide de jouer « 6 b 2 » pour compléter la sous grille 6. Le bot peut alors jouer où il veut car le coup l'envoie sur la grille 5. On constate enfin que le morpion principal est mis à jour avec le coup du joueur o.

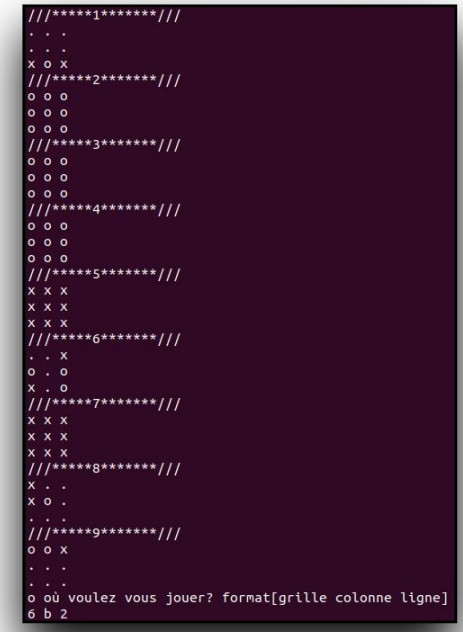


Figure 15 *: affichage du morpion avec disposition rentrée au préalable. Le joueur jour « 6 b 2 »*

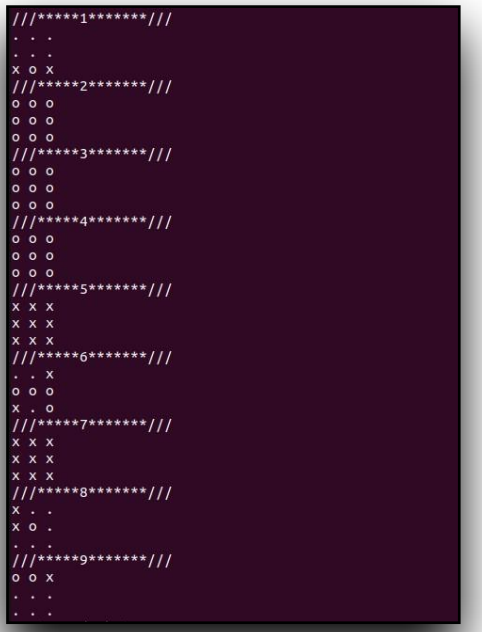


Figure 16 : *affichage des morpions suite coup joué*

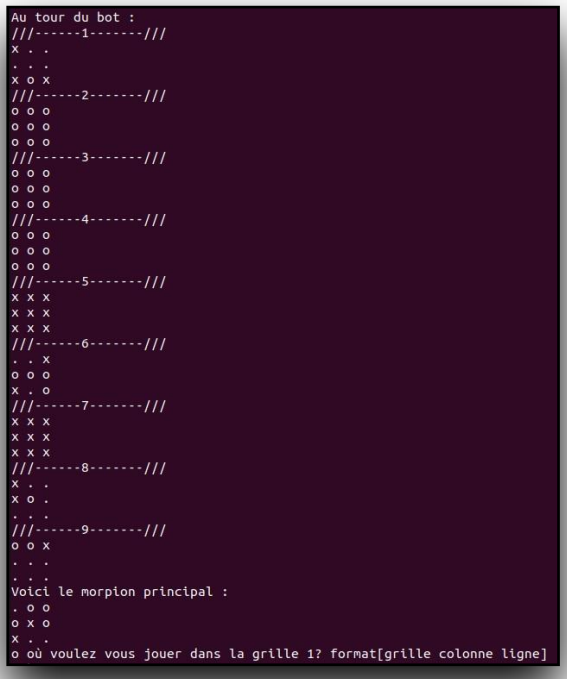


Figure 17 : *au tour du bot de de jouer et affichage du morpion principal*

Affichage graphique du super morpion :

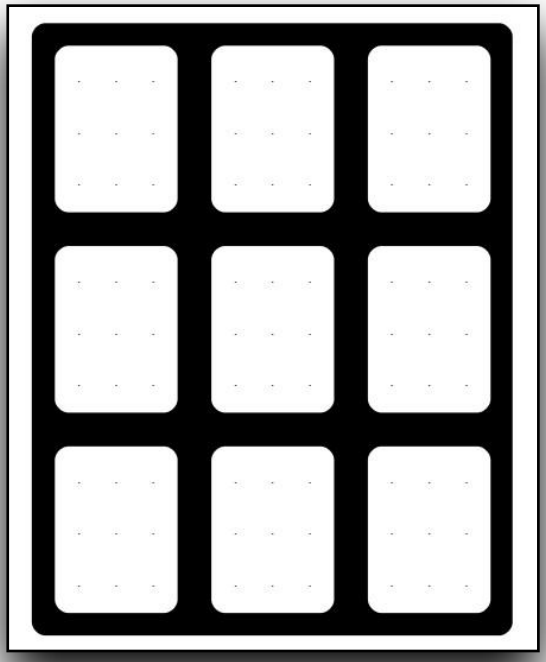


Figure 18 : *affichage graphique du super morpion*

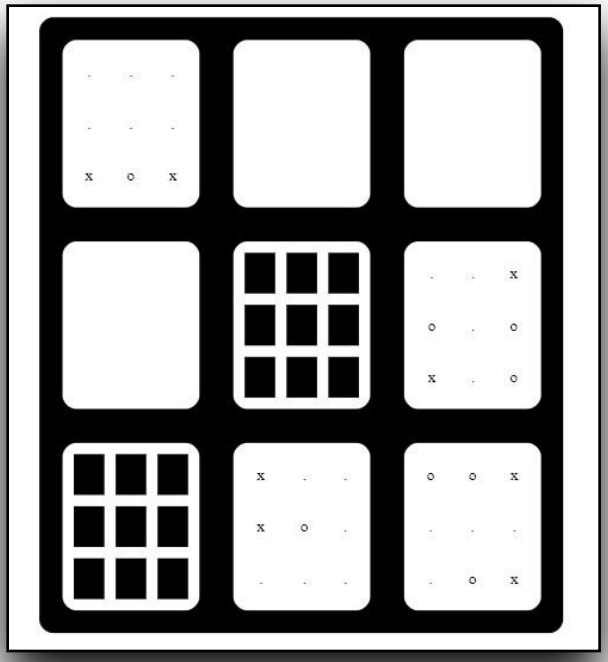


Figure 19: *affichage graphique du super morpion lors d'une partie en cours*

C. DIFFICULTÉS RENCONTRÉES

Pour découvrir tous les problèmes liés à l'exécution du code, nous avons utilisé la commande **GDB** qui nous a permis de résoudre les problèmes de segmentation fault, entre autres.

IV. Comment trouver le meilleur coup ?

Dans cette partie, nous avons adapté le programme **sm-refresh** produit dans la partie précédente pour pouvoir à partir d'un FEN donné, du dernier coup et du temps restant avant la fin de la partie d'obtenir le meilleur coup suivant à jouer.

A. STRATEGIES

Pour cela, nous avons simplement adapté le programme précédent pour qu'il nous renvoie simplement le coup à jouer et non plus une partie entière. Néanmoins, nous avons aussi mis en place des stratégies pour optimiser la recherche du coup

Après réflexion, nous avons décidé de considérer que le coup optimal est souvent dans la case du milieu. De ce fait, lorsque la case du milieu du morpion où le bot doit jouer est libre, il va avoir de grande chance de jouer dedans. Il ne joue pas dedans seulement s' il a de grandes chances de perdre le coup suivant. Autrement dit, si le morpion du milieu est perdant pour lui, il ne jouera pas la case du milieu puisque trop risqué pour lui

Par analogie avec cette stratégie, nous avons aussi développé une ouverture classique : le premier coup dans le cas où le morpion est vide se fera toujours en 5 5, c'est-à -dire en plein milieu du super morpion.

Ces stratégies et quelques autres optimisations permettent d'optimiser le programme précédent. Au delà de ça, nous avons fixé une profondeur de 4 qui alliait à la fois rapidité et justesse du coup. En effet, parfois lorsque la profondeur est trop importante, les différents appels de la fonction **CoupOpti** entraînent un message d'erreur **"Terminated"** (crash) du terminal. Les pistes d'améliorations pour ce programme sont donc de réussir à comprendre d'où vient l'erreur qui est indiscernable même en utilisant l'outil de débogage **GDB**.
Il faut aussi éviter d'utiliser de manière trop importante la manipulation de chaînes de caractères qui est malheureusement un problème majeur de notre fil rouge.
Aussi, Il faut éviter de dépasser le temps de jeu.

B. GESTION DU TEMPS

Pour gérer le temps, nous avons importé un nouveau module de gestion du temps utilisant majoritairement le fichier d'entête **<time.h>** et des fonctions permettant d'obtenir le temps à n'importe quel instant.
Pour permettre au programme de bien se plier au délai, lorsque le temps pour un coup dépasse 23 secondes, la fonction récursive évaluation renvoie **-INFINI**, de ce fait, seul le meilleur coup trouvé au bout de 23 secondes sera pris en compte. Il reste alors 7 secondes au programme pour remonter les appels récursifs et conclure, ce qui est largement suffisant.

C. EXEMPLE

Nous avons mis directement le code en pratique sur une position initialement rempli avec uniquement la case au milieu par le joueur **'x'**. Nous obtenons l'illustration des coups recommandés par le bot sur une plateforme d'affichage en ligne (nous reportions manuellement les coups joués puisqu'il n'y a plus de gestion graphique dans ce livrable). A chaque étape on demande à l'ordinateur le meilleur coup à jouer, l'ordinateur joue contre lui-même. On reporte les coups recommandés dans l'affichage en ligne :

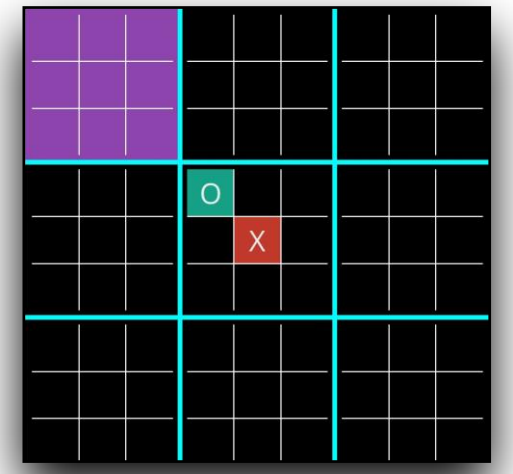


Figure 20 : *1er tour de jeu*

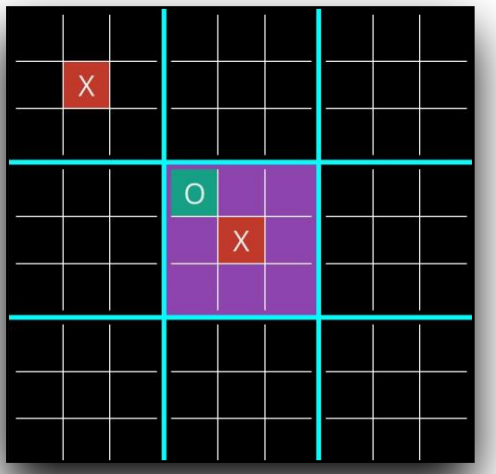


Figure 21 : 2ème tour de jeu

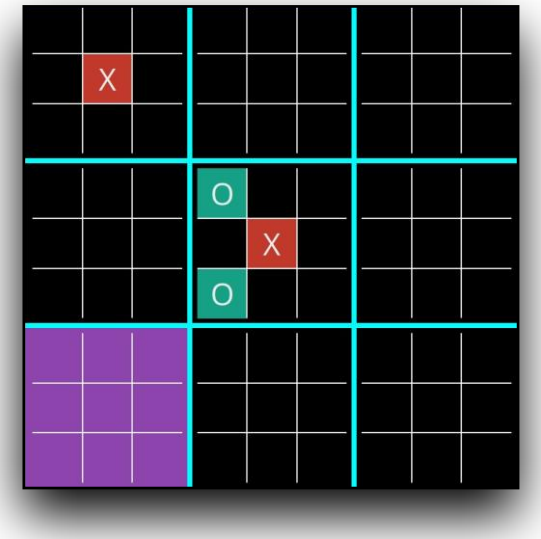
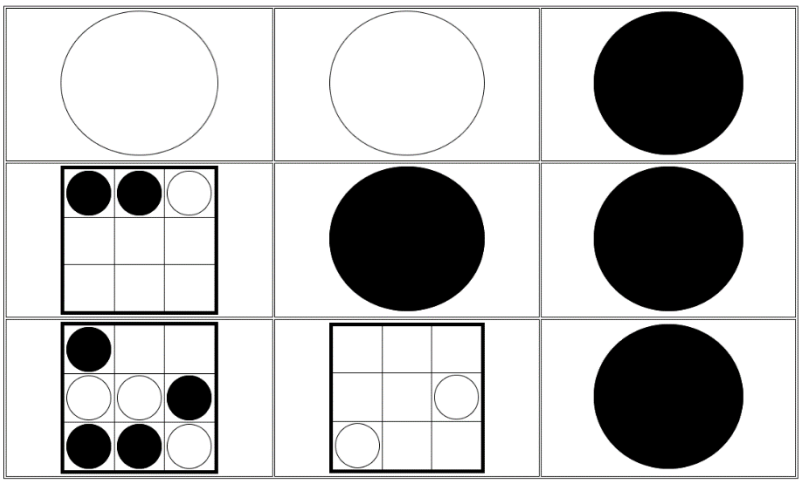


Figure 22 : *3ème tour de jeu*

Le jeu peut donc continuer ainsi jusqu'à ce que la partie se termine.

Nous avons donc fait un essai du programme qui va être utilisé lors du tournoi pour faire affronter notre bot contre lui-même. On obtient finalement la grille suivante :



Après plusieurs essais (configurations initiales différentes etc…), le programme semble fonctionner correctement, ce qui est une réussite. Néanmoins, les pistes d'améliorations sont nombreuses : notre stratégie de jouer le coup du milieu n'est finalement peut-être pas la meilleure, ou en tout cas, il faudrait mettre des conditions plus fortes. D'autre part, nous aurions aussi pu fixer une profondeur plus importante : les coups joués sont rapides et nous n'utilisons pas le temps que nous avons de manière optimale.

V. Conclusion

Pour conclure, ce fil rouge nous a permis de développer toutes les compétences apprises durant les séances d'AAP sur un exemple concret. Notamment, l'utilisation de structures et de fonctions vu en cours a permis d'optimiser notre programme.

Néanmoins, il y a encore certaines limites à notre programme comme la complexité et l'allocation de mémoire qui est très importante mais aussi la justesse des coups de notre bot dû notamment à l'absence d'élagage et de stratégies vraiment efficaces.