Compte rendu AAP

Ryan Blais

Antoine Aujoulat

Cassio Balligand

Introduction

L'objectif de notre projet était de développer la structure d'un super morpion pour participer à un tournoi où différents algorithmes s'affrontent dans des parties de super morpion.

Au début, nous avons mis en place la structure de base du super morpion. Ensuite, nous nous sommes concentrés sur le développement de trois programmes clés. Ces programmes étaient essentiels pour assurer la bonne fonctionnalité du jeu et sa capacité à rivaliser dans le tournoi.

Structure du jeu

Dans notre projet, nous avons développé plusieurs fonctions clés pour notre jeu de morpion en C.

1. **Notre fonction char_to_valeur** : Cette fonction a été créée pour convertir un caractère en une valeur de type T_valeur. Elle retourne CROIX pour ‘x’, ROND pour ‘o’, et VIDE pour tout autre caractère. Elle est essentielle pour interpréter les symboles des joueurs sur le plateau.
2. **La fonction int_to_char** : Cette fonction transforme un entier compris entre 1 et 9 en son équivalent caractère. Elle facilite la gestion des positions sur le plateau de façon intuitive pour les joueurs.
3. **Notre fonction valeur_to_char** : Cette fonction convertit une T_valeur en caractère. Pour une case avec une croix ou un rond, elle retourne ‘x’ ou ‘o’

respectivement. Pour une case vide, un compteur est augmenté et le chiffre correspondant est retourné.

4. **La fonction oppose** : Utilisée pour obtenir la valeur opposée dans T_valeur, cette fonction est clé pour changer de joueur après chaque coup.
5. **Notre fonction coup** : Cruciale pour la mise à jour du plateau après chaque mouvement. Elle vérifie si la partie est finie, si c'est le bon tour, et si la case sélectionnée est libre, avant de mettre à jour le jeu.
6. **La fonction fen_to_grille** : Cette fonction sert à convertir un code FEN en un plateau de jeu. Elle parcourt la chaîne FEN et remplit le plateau selon les règles du morpion.

Ces fonctions ont permis de mettre en place un jeu de morpion efficace et interactif en C, gérant divers aspects tels que l'état du plateau, les tours des joueurs, et la représentation graphique du jeu.

```
#include "morpion.h"
#include <stdio.h>
#include <stdlib.h>



T_valeur char_to_valeur(char valeur) {
    switch(valeur) {
        case('x') : return CROIX;
        case('o') : return ROND;
        default : return VIDE; //si c'est un chiffre
    }
}


char int_to_char(int k) { //transforme un chiffre entre 1 et 
                          //en un caractère de ce même chiffr
    switch(k) {
        case 1 : return '1';
        case 2 : return '2';
        case 3 : return '3';
```

```
        case 4 : return '4';
        case 5 : return '5';
        case 6 : return '6';
        case 7 : return '7';
        case 8 : return '8';
        case 9 : return '9';
        default : fprintf(stderr,"veuillez entrer un chiffre (
    }
}

char valeur_to_char(T_valeur valeur) {
    static int compteur = 0; //nombre de cases vides d'affilé
    switch(valeur) {
        case CROIX : compteur = 0; return 'x';
        case ROND : compteur = 0; return 'o';
        case VIDE : compteur ++; return int_to_char(compteur)
    }
}

T_valeur oppose(T_valeur valeur) {
    switch(valeur) {
        case CROIX : return ROND;break;
        case ROND : return CROIX;break;
        default : return VIDE;break;
    }
}

T_grille coup(T_grille grille, int position, T_valeur coup_jo
    if (grille.jeu_fini != 0) {fprintf(stderr,"la partie est (
    if (grille.trait == oppose(coup_joue)) {fprintf(stderr,"c
    if (grille.cases[position].valeur != VIDE) {fprintf(stder

    else {
        grille.cases[position].valeur = coup_joue;
        grille.jeu_fini = eval_fin(grille);
        grille.trait = oppose(coup_joue);
        return grille;
        }
```

```
}

T_grille fen_to_grille(char * code_fen) {
    T_grille grille;
    grille.trait = VIDE;
    grille.jeu_fini = 0;
    int position = 0;
    int compteur = 0; //nombre de cases vides que l'on  a ren
    int i = 0; //nb de chiffres ds le code fen
    int compteurs[MAXLEN] = {0}; //tableau de compteurs, comp
    while( (code_fen[position] != ' ') && (position + compteu
            (grille.cases[position+compteur-i]).valeur = char
            if (char_to_valeur(code_fen[position]) != VIDE) {
            else {
                compteur++;
                compteurs[i]++;
                if (compteurs[i] == code_fen[position] - '0')
                    position++;
                    i++;

                 }
            }

    }
    grille.trait = char_to_valeur(code_fen[position+1]);
    grille.jeu_fini = eval_fin(grille);
    return grille;
}
```

Nous avons ensuite mis en place des fonctions pour gérer l'évaluation de la fin du jeu et l'affichage du plateau de morpion.

1. **Fonction** `valeur_to_int` : Nous l'avons créée pour convertir une valeur de type `T_valeur` en un entier. Cela nous permet de coder facilement l'état du jeu : 0 pour VIDE, 1 pour ROND, et 2 pour CROIX. Cette conversion est particulièrement utile pour déterminer l'état du jeu après chaque coup.

2. **Fonction** `eval_fin` : Cette fonction est le cœur de notre logique de jeu. Elle évalue si la partie est finie en vérifiant les différentes combinaisons gagnantes possibles sur le plateau. Si trois cases consécutives (verticalement, horizontalement ou diagonalement) ont la même valeur (ROND ou CROIX) et ne sont pas VIDES, cela signifie qu'un joueur a gagné. La fonction retourne alors le chiffre correspondant au joueur gagnant. Si toutes les cases sont remplies sans vainqueur, la fonction retourne 3 pour un match nul. Sinon, elle retourne 0, indiquant que la partie continue.
3. **Fonction** `print_grille` : Nous avons conçu cette fonction pour afficher le plateau de jeu dans la console. Chaque case du plateau est représentée par 'x' pour CROIX, 'o' pour ROND, ou un point pour VIDE. Après chaque troisième case, un saut de ligne est inséré pour bien structurer le plateau en 3x3.

Grâce à ces fonctions, notre jeu de morpion peut non seulement suivre l'état du jeu, mais aussi déterminer quand la partie se termine et afficher le plateau de manière claire et intuitive pour les joueurs. Notre implémentation assure une expérience de jeu fluide et facile à suivre.

```
int valeur_to_int(T_valeur valeur) { //renvoie le chiffre cor
    switch(valeur) {
        case VIDE : return 0; //pas réellement besoin de celu
        case ROND : return 1;
        case CROIX : return 2;
    }
}

int eval_fin(T_grille grille) { //fonction que l'on peut larg
    if ( (grille.cases[1].valeur == grille.cases[0].valeur) &
        && (grille.cases[0].valeur != VIDE) )
        {return valeur_to_int(grille.cases[0].valeur);} // oo
                                                        // ..
                                                        // ..
    if ( (grille.cases[3].valeur == grille.cases[0].valeur) &
        && (grille.cases[0].valeur != VIDE) )
        {return valeur_to_int(grille.cases[0].valeur);} //  o
                                                        //  o
                                                        //  o
```

```
    if ( (grille.cases[4].valeur == grille.cases[0].valeur) &&
        && (grille.cases[0].valeur != VIDE) )
         {return valeur_to_int(grille.cases[0].valeur);} //  (
                                                         //
                                                         //

    if ( (grille.cases[4].valeur == grille.cases[3].valeur) &&
        && (grille.cases[3].valeur != VIDE) )
         {return valeur_to_int(grille.cases[3].valeur);} //
                                                         //  (
                                                         //

    if ( (grille.cases[7].valeur == grille.cases[6].valeur) &&
        && (grille.cases[6].valeur != VIDE) )
         {return valeur_to_int(grille.cases[6].valeur);} //
                                                         //
                                                         //  (

    if ( (grille.cases[4].valeur == grille.cases[1].valeur) &&
        && (grille.cases[1].valeur != VIDE) )
         {return valeur_to_int(grille.cases[1].valeur);} //
                                                         //
                                                         //

    if ( (grille.cases[5].valeur == grille.cases[2].valeur) &&
        && (grille.cases[2].valeur != VIDE) )
         {return valeur_to_int(grille.cases[2].valeur);} //
                                                         //
                                                         //

    if ( (grille.cases[4].valeur == grille.cases[2].valeur) &&
        && (grille.cases[2].valeur != VIDE) )
         {return valeur_to_int(grille.cases[2].valeur);} //
                                                         //
                                                         //  (

    else {
```

```
        int i;
        int compteur = 0; //nombre de cases jouées
        for(i=0;i<MAXLEN;i++) {
            if(grille.cases[i].valeur!=VIDE) {compteur++;}
        }
        if(compteur == MAXLEN) {return 3;} //nul si toutes le
        else {return 0;} //si toutes les cases ne sont pas re
    }
}


void print_grille(T_grille grille) {
    int i;
    for(i=0;i<MAXLEN;i++) {
        switch((grille.cases[i]).valeur) {
            case CROIX : printf("x  ");break;
            case ROND : printf("o  ");break;
            default : printf("•  ");break;
        }
        if ((i+1)%3==0) printf("\n");
    }

}
```

Dans cette section de notre code pour le jeu de morpion, nous avons implémenté deux fonctions avancées, `nbmarqueurs` et `minimax_alpha_beta`.

1. **Fonction** `nbmarqueurs` : Nous avons créé cette fonction pour compter le nombre de cases occupées par un marqueur spécifique (CROIX ou ROND) sur le plateau. Elle parcourt toutes les cases et incrémente un compteur chaque fois qu'elle trouve une case contenant la valeur spécifiée. Cette fonction est utile pour évaluer l'état du plateau de jeu dans différentes situations.
2. **Fonction** `minimax_alpha_beta` : Cette fonction est l'élément central de notre algorithme pour le morpion. Elle utilise l'algorithme Minimax avec élagage Alpha-Beta pour déterminer le meilleur coup possible. L'algorithme évalue les mouvements possibles et choisit celui qui maximise les chances de victoire pour le joueur actuel tout en minimisant celles de l'adversaire.

- **Évaluation du Score** : Nous utilisons la fonction `eval_fin` pour évaluer l'état actuel du jeu. Si le joueur maximisant a gagné, la fonction retourne un score élevé, et inversement pour le joueur minimisant. En cas de match nul, elle retourne 0.
- **Parcours des Coups Possibles** : L'algorithme explore les différents coups possibles. Pour le joueur maximisant, il cherche à maximiser le score, tandis que pour le joueur minimisant, il cherche à le minimiser.
- **Élagage Alpha-Beta** : L'élagage est utilisé pour réduire le nombre de branches de l'arbre de jeu explorées. Cela améliore l'efficacité de l'algorithme en évitant les calculs inutiles sur les branches qui ne peuvent pas influencer le résultat final.

L'implémentation de l'algorithme Minimax avec élagage Alpha-Beta assure d'évaluer efficacement les coups et choisir le meilleur mouvement possible à chaque tour.

```
int nbmarqueurs(T_grille grille, T_valeur valeur) {
    int i;
    int compteur = 0;
    for(i=0;i<MAXLEN;i++) {
        if (grille.cases[i].valeur == valeur) {
            compteur++;
        }
    }
    return compteur;
}
int minimax_alpha_beta(T_grille grille, int depth, int alpha,
    int score = eval_fin(grille);

    // Si celui qui maximise a gagné, on retourne son score.
    if (score == 10) return score + 10000 * (nbmarqueurs(gril

    // Si celui qui minimise a gagné, on retourne son score.
    if (score == -10) return score - 10000 * (nbmarqueurs(gri

    // Si c'est un match nul, on retourne 0.
    if ((eval_fin(grille) == 0) || ((eval_fin(grille) == 3)))
```

```
    if (is_maximizing_player == 0) {
        int meilleur_coup = -1000;

        for (int i = 0; i < MAXLEN; i++) {
            if (grille.cases[i].valeur == VIDE) {
                grille.cases[i].valeur = CROIX;
                int val = minimax_alpha_beta(grille, depth + 
                meilleur_coup = max(meilleur_coup, val);
                alpha = max(alpha, meilleur_coup);
                grille.cases[i].valeur = VIDE;

                // élagage alpha
                if (beta <= alpha) break;
            }
        }
        return meilleur_coup;
    }
    else {
        int meilleur_coup = 1000;

        for (int i = 0; i < MAXLEN; i++) {
            if (grille.cases[i].valeur == VIDE) {
                grille.cases[i].valeur = ROND;
                int val = minimax_alpha_beta(grille, depth + 
                meilleur_coup = min(meilleur_coup, val);
                beta = min(beta, meilleur_coup);
                grille.cases[i].valeur = VIDE;

                // élagage beta
                if (beta <= alpha) break;
            }
        }
        return meilleur_coup;
    }
}
```

Dans la partie de notre code qui suit, nous avons défini le fichier d'en-tête `morpion.h` pour notre jeu de morpion en C. C'est ici que nous déclarons toutes les structures, énumérations, constantes et prototypes de fonctions utilisés dans le jeu.

1. **Définition des Préprocesseurs** :
   - `#ifndef MORPION_H`, `#define MORPION_H` et `#endif` : Ces lignes garantissent que le fichier d'en-tête ne soit inclus qu'une seule fois, évitant ainsi les problèmes de redéfinitions multiples.
2. **Constante** `MAXLEN` : Nous avons défini `MAXLEN` comme 9, représentant le nombre total de cases dans un plateau de morpion standard (3x3).
3. **Énumération** `T_valeur` :
   - `typedef enum {ROND, CROIX, VIDE}` : Cette énumération représente les différents états possibles d'une case sur le plateau : occupée par un rond, une croix, ou vide.
4. **Structure** `T_case` **et** `T_grille` :
   - `T_case` contient une valeur de type `T_valeur`, indiquant l'état de la case.
   - `T_grille` est une structure représentant le plateau de jeu. Elle contient un tableau de `T_case`, un marqueur pour le joueur actuel (`trait`) et un indicateur de l'état du jeu (`jeu_fini`).
5. **Structure** `Coup` :
   - Cette structure est utilisée pour stocker le score et la position d'un coup dans l'algorithme Minimax.
6. **Prototypes de Fonctions** :
   - Nous déclarons les fonctions pour la conversion de types (`int_to_char`, `char_to_valeur`, `valeur_to_char`), la gestion des coups (`coup`, `oppose`), l'interprétation du code FEN (`fen_to_grille`), l'évaluation de l'état du jeu (`eval_fin`), l'affichage du plateau (`print_grille`), et les fonctions avancées pour l'IA (`minimax_alpha_beta`, `nbmarqueurs`).

En résumé, ce fichier d'en-tête est le fondement de notre jeu de morpion. Il organise et déclare toutes les structures et fonctions nécessaires pour le bon fonctionnement du jeu, assurant une structure claire et une meilleure maintenabilité du code. Grâce à cela, nous pouvons facilement gérer différentes parties du jeu dans différents fichiers source, tout en conservant une cohérence globale.

```
#ifndef MORPION_H
#define MORPION_H

#define MAXLEN 9

typedef enum {ROND , CROIX , VIDE} T_valeur;

typedef struct {
    T_valeur valeur;
} T_case;

typedef struct {
    T_case cases[MAXLEN];
    T_valeur trait;
    int jeu_fini; //0 si le jeu n'est pas fini, 1 si les rond
} T_grille;

typedef struct {
    int score;
    int position;
} Coup;


char int_to_char(int k);
T_valeur oppose(T_valeur valeur);
T_valeur char_to_valeur (char valeur);
char valeur_to_char(T_valeur valeur);
T_grille coup(T_grille grille, int position, T_valeur coup_jo
T_grille fen_to_grille(char * code_fen);
int eval_fin(T_grille grille);
void print_grille(T_grille grille);
int minimax_alpha_beta(T_grille grille, int depth, int alpha,
int nbmarqueurs(T_grille grille, T_valeur valeur)
#endif
```

Nous avons par la suite choisi de revoir une partie de notre code pour le jeu de morpion, nous avons apporté quelques modifications.

1. **Modification dans** `valeur_to_int` : Nous avons ajouté un `case` par défaut à la fonction `valeur_to_int`, renvoyant -1. Cela permet de gérer toute valeur inattendue de `T_valeur`.
2. **Fonction** `nbmarqueurs` : Nous avons ajouté cette fonction pour compter le nombre de marqueurs (CROIX ou ROND) présents sur le plateau. C'est utile pour évaluer l'état du jeu et pour l'implémentation de notre algorithme Minimax.

Dans la partie du code suivant, nous avons implémenté la fonction `negamax`, qui est une variante de l'algorithme Minimax, ainsi que plusieurs fonctions auxiliaires pour notre jeu de morpion. Expliquons ce que nous avons fait :

1. **Fonction** `negamax` :
   - **Objectif** : Elle est conçue pour trouver le meilleur coup possible en prenant en compte la profondeur (`depth`), les valeurs alpha et beta pour l'élagage, et le joueur actuel.
   - **Logique** : Si la profondeur est nulle ou si le nœud est terminal (aucun coup supplémentaire possible), la fonction évalue le plateau et retourne un coup avec cette évaluation. Sinon, elle génère tous les coups possibles (enfants) pour le joueur actuel, évalue chaque coup en utilisant récursivement `negamax` et choisit le coup avec le score le plus élevé.
2. **Fonction** `estNoeudTerminal` :
   - Elle vérifie si un plateau de jeu donné (`grille`) est dans un état terminal, c'est-à-dire si la partie est finie.
3. **Fonction** `evaluer` :
   - Elle évalue le plateau de jeu du point de vue du joueur spécifié. Si le joueur a gagné, elle renvoie une valeur positive. Si le joueur a perdu, elle renvoie une valeur négative. Sinon, elle renvoie la différence entre le nombre de marqueurs du joueur et de son adversaire.
4. **Fonction** `genere_enfants` :
   - Elle génère tous les coups possibles pour le joueur à partir d'une position donnée sur le plateau et les stocke dans un tableau `enfants`. Elle retourne également un tableau contenant les indices des coups générés.

5. **Fonctions `max` et `min` :**
    - Ces fonctions sont des implémentations standard pour trouver le maximum ou le minimum entre deux valeurs. Elles sont utilisées dans l'algorithme `negamax` pour l'élagage alpha-beta.

En résumé, grâce à l'algorithme `negamax` et aux fonctions auxiliaires, nous pouvons évaluer efficacement les positions et prendre des décisions stratégiques pour vaincre le bot adverse.

```
Coup negamax(T_grille node, int depth, int alpha, int beta, T
    if (depth == 0 || estNoeudTerminal(node)) {
        return (Coup){evaluer(node, joueur), -1}; } // Positi



    Coup val = {INT_MIN, -1};
    T_grille enfants[MAXLEN];
    int nombre_enfants;
    int * num_enfants = genere_enfants(node, joueur, enfants,

    for (int i = 0; i < nombre_enfants; i++) {
        if (-negamax(enfants[i], depth - 1, -beta, -alpha, op
            val.score = -negamax(enfants[i], depth - 1, -beta
            val.position = num_enfants[i];
        }

        if (val.score >= beta) {
            return val; // élagage beta.
        }

        alpha = max(alpha,val.score);
    }

    return val;
}

int estNoeudTerminal(T_grille grille) {
    int resultat = eval_fin(grille);
```

```
    return resultat != 0;
}

int evaluer(T_grille grille, T_valeur joueur) {
    int res = eval_fin(grille);
    if (res == valeur_to_int(joueur)) {
        return 10 + nbmarqueurs(grille,joueur) - nbmarqueurs(
    } else if (res == valeur_to_int(oppose(joueur))) {
        return -10 + nbmarqueurs(grille,joueur) - nbmarqueurs
    } else {
        return nbmarqueurs(grille,joueur) - nbmarqueurs(grill
    }
}

int * genere_enfants(T_grille grille, T_valeur joueur, T_gril
    // Génère tous les coups possibles à partir de la grille 
    // et les stocke dans le tableau 'enfants'. Met à jour 'n
    // renvoie un tableau contenant les numéros des cases aya
    *nombre_enfants = 0;
    int res[MAXLEN] = {-1,-1,-1,-1,-1,-1,-1,-1,-1};
    int k = 0;
    for (int i = 0; i < MAXLEN; ++i) { // au max 9 enfants
        if (grille.cases[i].valeur == VIDE) {
            res[k] = i;
            k++;
            enfants[*nombre_enfants] = grille;
            enfants[*nombre_enfants].cases[i].valeur = joueur
            ++(*nombre_enfants);
        }
    }
    return res;
}

int max(int a, int b) {
    if(a>b) return a;
    else return b;
}
```

```
int min(int a, int b) {
    if(a<b) return a;
    else return b;
}
```

Structure Super-Morpion

Enfin, nous avons développé un jeu de super morpion, qui étend le jeu classique de morpion en incorporant plusieurs grilles de morpion dans une grande grille. Voici comment nous avons structuré notre code pour cette version avancée :

1. **Fonction** `coup_super_grille` : Nous avons créé cette fonction pour gérer les coups joués dans le super morpion. Lorsqu'un coup est joué, nous vérifions d'abord si le joueur joue dans la bonne grille (déterminée par le dernier coup joué). Ensuite, nous mettons à jour la grille spécifique avec le coup joué et changeons le trait pour l'ensemble des grilles. Nous mettons également à jour l'état de la partie en évaluant si le super morpion est terminé.
2. **Fonction** `super_coup` : Cette fonction est appelée après chaque coup. Elle vérifie si l'un des neuf morpions est gagné et met à jour la grande grille du super morpion en conséquence. Nous avons également inclus un cas spécial pour les matchs nuls.
3. **Fonction** `eval_fin_super_grille` : Nous avons développé cette fonction pour évaluer si le super morpion est terminé. Si la grande grille est gagnée ou nulle, la partie est finie. Sinon, nous vérifions chaque grille pour voir si la partie continue.
4. **Fonctions** `puissance`, `concat`, `char_to_string`, `decoupe`, `coupe` : Ces fonctions auxiliaires sont utilisées pour diverses tâches, comme la conversion des caractères en points, la concaténation des chaînes, la transformation des caractères en chaînes et la découpe d'une chaîne selon les règles du super morpion.

```
#include "super_morpion.h"
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>


T_superGrille coup_super_grille(T_superGrille super_grille, i
    //coup dans la super grille
    int k = super_grille.dernier_coup_joue % 10; //grille dan
    if ( (numero_grille != k) && ((super_grille.grilles[k-1])
        fprintf(stderr,"veuillez jouer dans la même grille qu
    }
    T_grille grille = super_grille.grilles[numero_grille];
    super_grille.grilles[numero_grille] = coup(grille, positi
    for (int i=0; i<MAXLEN;i++) {
        super_grille.grilles[i].trait = oppose(coup_joue);
    }
    super_grille.dernier_coup_joue = 10*numero_grille + posit
    //convention de la notation 84 pour 8ème grille 4ème case
    super_grille.super_jeu_fini = eval_fin_super_grille(super
    super_grille.trait = oppose(coup_joue);
    return super_coup(super_grille); //au cas ou un des morpi
}

T_superGrille super_coup(T_superGrille super_grille) {
    //Joue dans la grande grille si l'un des 9 morpions est g
    //Le trait de la grande grille est toujours vide
    int i;
    for(i=0;i<MAXLEN;i++) {
        switch(eval_fin(super_grille.grilles[i])) {
            case 0 : return  super_grille;break;
            case 1 : super_grille.grilles[MAXLEN] = coup(supe
                     super_grille.grilles[MAXLEN].trait = VID
                     super_grille.super_jeu_fini = eval_fin_s
                     return super_grille;break;
            case 2 : super_grille.grilles[MAXLEN] = coup(supe
                     super_grille.grilles[MAXLEN].trait = VID
                     super_grille.super_jeu_fini = eval_fin_s
                     return  super_grille;break;
            case 3 : super_grille.grilles[MAXLEN] = coup(supe
                    super_grille.super_jeu_fini = eval_fin_su
```

```
                    return  super_grille;break;
        }
    }
}

int eval_fin_super_grille(T_superGrille super_grille) { //répé
                                                        //(c
    if(eval_fin(super_grille.grilles[MAXLEN]) != 0) return eva
    else {
        int i;
        for(i=0;i<MAXLEN;i++) {
            if(eval_fin(super_grille.grilles[i]) == 0) return
        }
        return 3;
    }
}

int puissance(char car) {
    switch (car) {
        case 'O' : return 9;break;
        case 'X' : return 9;break;
        case 'x' : return 1;break;
        case 'o' : return 1;break;
        default : return car - '0';break; // si c'est un chif
    }
}

char * concat(char * mot1, char * mot2) {
    int l1 = strlen(mot1);
    int l2 = strlen(mot2);
    char * res = malloc((l1+l2+1) * sizeof(char)); //+1 pour
    int i;
    strcpy(res, mot1);
    strcat(res, mot2);
    return res;
}

char * char_to_string(char trait) {
```

```
    switch(trait) {
        case 'o' : return "o";break;
        case 'x' : return "x";break;
        default : return "vide";break;
    }
}


char ** decoupe(char * code_fen, char trait) {
    int compteur_puissance = 0;
    char ** res = malloc(MAXLEN * sizeof(char *));
    int position = 0;
    int i = 0; //numero de la grille
    int debuts[MAXLEN+1] = {0}; //positions des débuts de gri
    int fins[MAXLEN+1] = {0}; //positions des fins de grille 
    fins[MAXLEN-1] = strlen(code_fen) - 5;
    //on enleve le trait, le dernier coup et les deux espaces
    while ( (code_fen[position] != ' ') && (i<MAXLEN) && (pos
        compteur_puissance = compteur_puissance + puissance(c
        fins[i]++;
        if (compteur_puissance % 9 == 0) {
            debuts[i+1] = fins[i];
            fins[i+1] = debuts[i+1];
            res[i] = concat (concat(coupe(code_fen,debuts[i],
            //on rajoute l'espace et le trait à la fin du déc
            i++;

        }
        position++;
    }
    return res;
}


char * coupe(char * mot, int debut, int fin) {
    char * res = malloc((fin - debut + 2) * sizeof(char)); //
    int i;
    for (i = debut; i <= fin; i++) {
```

```
            res[i - debut] = mot[i];
        }
        res[i - debut] = '\0'; // Ajoutez le caractère nul à la f
        return res;
    }
```

Dans la suite de notre code pour le super morpion, nous avons implémenté deux fonctions principales pour convertir une chaîne FEN en une super grille de morpion et pour afficher cette super grille. Voici comment nous avons procédé :

1. **Fonction** `fen_to_super_grille` :
    - **Conversion de FEN** : Cette fonction commence par convertir le code FEN en un ensemble de grilles de morpion. Nous utilisons `decoupe` pour diviser le code FEN en différentes grilles.
    - **Traitement des Grilles** : Pour chaque grille, nous vérifions si elle représente une grille complète (marquée par 'O' ou 'X') ou une grille normale de morpion. Pour les grilles complètes, nous les remplissons avec le caractère correspondant ('O' ou 'X'). Pour les autres, nous utilisons `fen_to_grille` pour les convertir.
    - **Mise à Jour des États** : Après avoir traité toutes les grilles, nous mettons à jour l'état du jeu en appelant `super_coup` et `eval_fin_super_grille`.
2. **Fonction** `print_super_grille` :
    - **Affichage de la Super Grille** : Nous utilisons `fen_to_super_grille` pour convertir le code FEN en une super grille, puis nous parcourons chaque petite grille pour afficher son contenu. Chaque case est représentée par 'x', 'o', ou un point.
    - **Affichage de la Grande Grille** : À la fin, nous affichons également l'état de la grande grille, en utilisant la même logique que pour une grille de morpion classique.

```
T_superGrille fen_to_super_grille(char * code_fen) {
    int len = strlen(code_fen);
    char trait = code_fen[len-1];
    char ** fen_grilles = decoupe(code_fen,trait);
```

```
    T_superGrille super_grille;
    super_grille.trait = trait;
    int i;
    for (i=0;i<MAXLEN; i++) {
        if ( (strncmp(fen_grilles[i], "O", 1) == 0) || (strnc
            char * valeur = fen_grilles[i];

            // Corrigez la boucle interne pour ne pas écraser
            int k;
            for (k=0;k<MAXLEN; k++) {
                super_grille.grilles[i].cases[k].valeur = *va
            }
            super_grille.grilles[i].trait = VIDE;
            super_grille.grilles[i].jeu_fini = eval_fin(super
        }
        else {
        super_grille.grilles[i] = fen_to_grille(fen_grilles[i
        }
    }

    T_superGrille res = super_coup(super_grille);
    res.super_jeu_fini = eval_fin_super_grille(res);
    res.dernier_coup_joue = (code_fen[len-4] - '0') * 10 + co
    return res;
}




void print_super_grille(char * code_fen) {
    T_superGrille super_grille = fen_to_super_grille(code_fen
    int i;
    int j;
    for(j=0;j<MAXLEN;j++) {
        for(i=0;i<MAXLEN;i++) {
            switch((super_grille.grilles[i/3 + 3* (j/3)]).cas
                case CROIX : printf("x  ");break;
                case ROND : printf("o  ");break;
```

```
                default : printf("•  ");break;
            }
            }
        printf("\n");
    }
    printf("\n");
    //ensuite on affiche la grande grille en reprenant le cod
    //pour une grille de morpion classique
    int k;
    for(k=0;k<MAXLEN;k++) {
        switch(((super_grille.grilles[MAXLEN-1]).cases[k]).va
            case CROIX : printf("X  ");break;
            case ROND : printf("O  ");break;
            default : printf("•  ");break;
        }
        if ((i+1)%3==0) printf("\n");
    }
```

Dans ce segment de code d'après, nous avons défini la structure et les fonctions pour notre jeu de super morpion, en étendant les concepts du jeu de morpion classique. Voici comment nous avons structuré notre code :

1. **Directive de Préprocesseur** :
    - Nous avons utilisé `#ifndef`, `#define`, et `#endif` pour éviter les inclusions multiples de ce fichier d'en-tête, ce qui est une bonne pratique en C pour éviter les problèmes de redéfinition.
2. **Structure** `T_superGrille` :
    - Cette structure est le cœur de notre jeu de super morpion. Elle contient un tableau de `T_grille`, représentant les neuf grilles de morpion classique plus une grille supplémentaire servant de grande grille pour le super morpion. Elle inclut aussi le dernier coup joué, l'état du jeu (`super_jeu_fini`) et le joueur actuel (`trait`).
3. **Déclaration des Fonctions** :

- `coup_super_grille` : Gère les coups joués dans une grille spécifique du super morpion.
- `super_coup` : Joue dans la grande grille du super morpion lorsque l'un des morpions est gagné.
- `eval_fin_super_grille` : Évalue si le super morpion est terminé.
- `fen_to_super_grille` : Convertit un code FEN en une super grille de morpion.
- `print_super_grille` : Affiche la super grille de morpion.
- `puissance`, `concat`, `decoupe`, `coupe`, `char_to_string` : Fonctions utilitaires pour traiter des chaînes de caractères et des calculs spécifiques au jeu.

```
#ifndef SUPER_MORPION_H
#define SUPER_MORPION_H

^m#include "morpion.h"

typedef struct {
    T_grille * grilles;
    int dernier_coup_joue;
    int super_jeu_fini;
    T_valeur trait;
 } T_superGrille; // 9 grilles de morpions classiques
                                    // + 1 grille servant de "g

T_superGrille coup_super_grille(T_superGrille super_grille, i
T_superGrille super_coup(T_superGrille super_grille);
int eval_fin_super_grille(T_superGrille super_grille);
int puissance(char car);
char * concat(char * mot1, char * mot2);
char ** decoupe(char * code_fen, char trait);
char * coupe(char * mot, int debut, int fin);
T_superGrille fen_to_super_grille(char * code_fen);
void print_super_grille(char * code_fen);
int puissance(char car);
```

```
char * char_to_string(char trait);
#endif
```

Programme 1 : TTTREE

Dans ce premier programme de notre code pour le jeu de morpion, nous avons codé une fonction `main` pour exécuter le jeu qui se réfère aux fonctions `grille_to_dot` et `generate_dot_rec` Expliquons ce que nous faisons dans cette version complète :

1. **Fonction `main`** :
   - **Traitement des Arguments** : La fonction `main` commence par traiter le premier argument passé au programme (`argv[1]`), qui est supposé être une notation FEN représentant une position spécifique sur le plateau de morpion.
   - **Conversion de la FEN en Grille** : Nous utilisons la fonction `fen_to_grille` pour convertir cette notation FEN en une structure `T_grille`, représentant l'état actuel du plateau de jeu.
   - **Gestion des Erreurs** : Si la conversion échoue (`grille.jeu_fini == -1`), le programme signale une erreur et se termine.
2. **Génération du Graphique DOT** :
   - **Initialisation** : La fonction `main` initialise la génération du graphique DOT avec `printf("digraph {\\n");`. Cela marque le début d'un graphique dans la notation DOT.
   - **Génération Récursive** : Elle appelle ensuite `generate_dot_rec` pour générer récursivement le graphique DOT, en passant la grille initiale, un numéro de nœud initial (0) et le joueur actuel.
3. **Fonctions `grille_to_dot` et `generate_dot_rec`** (comme expliqué précédemment) :
   - `grille_to_dot` génère la représentation graphique d'un état spécifique du jeu.
   - `generate_dot_rec` explore récursivement tous les coups possibles à partir d'un état de jeu donné et génère des nœuds et des liens dans le graphique DOT pour représenter ces coups.

En résumé, ce code nous permet de visualiser le jeu de morpion en utilisant la notation DOT pour générer un graphique.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "morpion.h"


void grille_to_dot(T_grille grille) {
    static int nb_noeuds = 0;
    printf("m%d [shape=none label=<<TABLE border='0' cellspac
    printf("<TR>\n");
    for (int i=0; i<3; i++) {
        printf("<TD bgcolor='white'>%c</TD>\n", valeur_to_cha
    }
    printf("</TR>\n");
    printf("<TR>\n");
    for (int i=3; i<6; i++) {
        printf("<TD bgcolor='white'>%c</TD>\n", valeur_to_cha
    }
    printf("</TR>\n");
    printf("<TR>\n");
    for (int i=6; i<9; i++) {
        printf("<TD bgcolor='white'>%c</TD>\n", valeur_to_cha
    }
    printf("</TR>\n");
    if (grille.trait == ROND) printf("<TR><TD bgcolor='red' c
    if (grille.trait == CROIX) printf("<TR><TD bgcolor='green
    nb_noeuds++;
}

void generate_dot_rec(T_grille grille, int noeud, T_valeur jo
    grille_to_dot(grille);

    if (eval_fin(grille) != 0) return;
```

```
    for (int i = 0; i < MAXLEN; i++) {
        if (grille.cases[i].valeur == VIDE) {
                T_grille new_grille = grille;
                new_grille.cases[i].valeur = joueur;
            new_grille.trait = oppose(joueur);
            printf("  m%d -> m%d;\n", noeud, noeud + 1);
                generate_dot_rec(new_grille, noeud + 1, oppos
        }
    }
}


int main(int argc, char *argv[]) {


    char *fen = argv[1];
    T_grille grille = fen_to_grille(fen);
    joueur = grille.trait;

    if (grille.jeu_fini == -1) {
        fprintf(stderr, "Erreur lors de la conversion de la F
        fclose(dotFile);
    }

    printf("digraph {\n");
    generate_dot_rec(grille, 0, joueur);
    printf("}\n");

    return 0;
}
```

Programme 2 : Sm_refresh

1. Nous avons conçu les fonctions `grille_to_dot_bis` et `grille_gagnée_to_dot_bis` pour créer des représentations graphiques des grilles de morpion en utilisant le

langage DOT. La première affiche une grille normale, tandis que la seconde montre une grille où un joueur a gagné, chacune avec un style distinctif. Nous avons utilisé des codes HTML spécifiques pour refléter les différents états des grilles.

2. La fonction `superGrille_to_dot` nous a permis d'afficher la super grille du jeu. Elle parcourt toutes les petites grilles du super morpion, utilisant la représentation graphique appropriée pour chaque grille selon son état.
3. Avec la fonction `echecs_to_coup`, nous avons traduit un coup d'échecs en un coup de super morpion. Cette fonction est cruciale pour convertir les entrées utilisateur en actions concrètes dans le jeu.
4. Dans la fonction `main`, nous avons traité les interactions avec l'utilisateur. Après avoir extrait et converti le coup de l'argument de la ligne de commande, nous avons initialisé la super grille. Ensuite, nous avons joué le coup indiqué et calculé le meilleur coup en réponse. Enfin, nous avons affiché l'état actuel du jeu et sa représentation graphique.

Cette méthodologie nous a aidés à augmenter l'interactivité et la visibilité de notre jeu de super morpion, facilitant la compréhension des stratégies et des états du jeu pour les utilisateurs.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "morpion.h"
#include "super_morpion.h"
#include "sm_refresh.h"


void grille_to_dot_bis(T_grille grille) {
    printf("<TD bgcolor='white'> <TABLE border='0' cellspacin
    printf("<TR>\n");
    for (int i=0; i<3; i++) {
        printf("<TD bgcolor='white'>%c</TD>\n", valeur_to_cha
    }
    printf("</TR>\n");
    printf("<TR>\n");
```

```
    for (int i=3; i<6; i++) {
        printf("<TD bgcolor='white'>%c</TD>\n", valeur_to_cha
    }
    printf("</TR>\n");
    printf("<TR>\n");
    for (int i=6; i<9; i++) {
        printf("<TD bgcolor='white'>%c</TD>\n", valeur_to_cha
    }
    printf("</TR>\n");
    printf("</TABLE>\n");
    printf("</TD>\n");
}

void grille_gagnée_to_dot_bis(T_grille grille) {
    if (grille.jeu_fini == 1) {
        printf("<TD bgcolor='black'> <TABLE border='0' cellsp
    }
    else if (grille.jeu_fini == 2) {
        printf("<TD bgcolor='white'> <TABLE border='0' cellsp
    }
}

void superGrille_to_dot(T_superGrille superGrille) {
    printf("digraph  { a0 [shape=none label=<<TABLE border='0
    for (int i=0; i<9; i++) {
        if (superGrille.grilles[i].jeu_fini ==1 || superGrill
            grille_gagnée_to_dot_bis(superGrille.grilles[i]);
        }
        else {
            grille_to_dot_bis(superGrille.grilles[i]);
        }
    }
    printf("  </TR> </TABLE> >]; }\n");
}

char * echecs_to_coup(char * echecs) {
    int i = echecs[0] - '0';
    int j = echecs[2] - 'a';
```

```
    int k = echecs[3] - '0'-1;
    char coup[2];
    coup[0] = int_to_char(i);
    coup[1] = int_to_char(j+3*k);
    return coup;
}



int main(int argc, char ** argv) {
    char * echecs = argv[1];
    char * coup = echecs_to_coup(echecs);
    int numero_grille = coup[0] - '0';
    int numero_case = coup[1] - '0';


    static T_superGrille superGrille;
    for (int i=0; i<MAXLEN; i++) {
        for (int j=0; j<MAXLEN; j++) {
            superGrille.grilles[i].cases[j].valeur = VIDE;
        }
    }

    superGrille = coup_super_grille(superGrille, numero_grill

    char * coup_bot = meilleur_coup(superGrille, oppose(super
    superGrille = coup_super_grille(superGrille, coup_bot[0]-

    print_super_grille(superGrille);
    superGrille_to_dot(superGrille);

    return 0;
}
```

Nous avons créé le fichier d'en-tête `prog1.h` pour centraliser les déclarations de fonctions importantes :

1. **Directives de Préprocesseur** : Nous avons utilisé `#ifndef`, `#define`, et `#endif` pour éviter les inclusions multiples du fichier d'en-tête.
2. **Inclusions de Fichiers** : Les fichiers `morpion.h` et `super_morpion.h` sont inclus pour accéder aux structures et fonctions nécessaires.
3. **Déclarations de Fonctions** :
   - `grille_to_dot_bis` et `grille_gagnée_to_dot_bis` pour représenter graphiquement les grilles de morpion.
   - `superGrille_to_dot` pour afficher la super grille du jeu.
   - `echecs_to_coup` pour convertir un coup d'échecs en coup de super morpion.

```
#ifndef PROG1_H
#define PROG1_H
#include "morpion.h"
#include "super_morpion.h"

void grille_to_dot_bis(T_grille grille);
void grille_gagnée_to_dot_bis(T_grille grille);
void superGrille_to_dot(T_superGrille superGrille);
char * echecs_to_coup(char * echecs);
#endif
```

Programme 3 : Sm-Bot

On a créer des fonctions stratégiques pour améliorer l'algorithme :

Intégration des Fichiers Nécessaires :

- Nous avons commencé par inclure les fichiers d'en-tête et les implémentations nécessaires, tels que `super_morpion.h`, `sm-bot.h`, et `morpion.h`. Cela nous a fourni toutes les structures de données et fonctions nécessaires pour le jeu.

1. **Développement de la Fonction `valeur_case` :**
    - Cette fonction attribue des valeurs stratégiques aux différentes cases de la grille, avec un accent particulier sur les cases centrales et les coins. Elle joue un rôle crucial dans l'évaluation stratégique des coups possibles.
2. **Elaboration de la Stratégie de Jeu avec `strategie` et `meilleur_coup` :**
    - La fonction `strategie` calcule un score pour chaque action possible dans la super grille en utilisant `minimax` et `negamax`.
    - `meilleur_coup` utilise cette stratégie pour sélectionner la grille et la case optimales à jouer, augmentant ainsi nos chances de succès dans le jeu.
3. **Gestion de l'Entrée Utilisateur et Affichage dans la `main` :**
    - Enfin, dans la fonction `main`, nous avons traité l'entrée utilisateur (code FEN) pour initialiser le jeu, calculé le meilleur coup possible, et mis à jour l'état de la super grille. Nous avons également assuré l'affichage du nouveau coup et de l'état de la grille après chaque action.

Chacune de ces étapes a été fondamentale pour construire un programme de super

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "super_morpion.h"
#include "sm-bot.h"
#include "morpion.h"
#include "morpion.c"
#include "super_morpion.c"


int valeur_case(int position) { //cases plus importantes que
    switch(position) {
        case 0 : return 5;
        case 1 : return 2;
        case 2 : return 5;
        case 3 : return 2;
        case 4 : return 10;
        case 5 : return 2;
```

```
        case 6 : return 5;
        case 7 : return 2;
        case 8 : return 5;
        default : return 0;
    }
}


SuperCoup strategie(T_superGrille super_grille, T_valeur joue
    int score_total = 0;
    SuperCoup super_coup;
    super_coup.score = INT_MIN;
    int scores[MAXLEN];
    for(int i = 0;i<MAXLEN;i++) {
        score_total = score_total + valeur_case(i) * minimax(
        scores[i] = valeur_case(i) * minimax(super_grille.gri
        if(scores[i] > super_coup.score) {
            super_coup.score = scores[i];
            super_coup.position = i;
        }
    }
    return super_coup;
}

char * meilleur_coup(T_superGrille super_grille, T_valeur jou
    char res[2];
    SuperCoup coup = strategie(super_grille, joueur, depth);
    T_grille meilleure_grille = super_grille.grilles[(coup.po
    int meilleure_case = negamax(meilleure_grille,depth, INT_
    res[0] = int_to_char(coup.position);
    res[1] = int_to_char(meilleure_case);
    return res;

}

int main(int argc, char ** argv) {
    char * fen = concat(argv[1],"\0");
    int time = argv[2] - '0';
```

```
    if(time <= 0) return 0;
    T_superGrille super_grille = fen_to_super_grille(fen);
    char * best_coup = meilleur_coup(super_grille,super_grill
    T_superGrille new_grille = coup_super_grille(super_grille
    print_super_grille(new_grille);
    printf("%s",meilleur_coup(super_grille,super_grille.trait
    return 0;
}
```

Perspective

Programme 1 : Réussir à parcourir en profondeur l'arbre pour afficher les noeuds et les flèches entre les noeuds correctement. Pour y parvenir, nous devons peaufiner notre algorithme de parcours en profondeur pour qu'il gère efficacement la structure de données complexe de notre jeu, notamment au niveau des super grilles

Programme 2 : Régler le soucis de segmentation fault qui empêche le bon affichage de la super grille au format sémi-graphique. Notre priorité est de déboguer et de résoudre ces erreurs. Cela impliquera une revue minutieuse du code pour identifier les zones où l'accès à la mémoire est mal géré ou où les références à des données non valides se produisent.

Programme 3 : Réussir à mettre en place une stratégie plus avancée qui tiendrait compte de la grille où sera jouer le coup suivant et donc anticiper non seulement les mouvements immédiats mais aussi planifier des coups à l'avance en fonction des actions probables de l'adversaire.

Conclusion :

En conclusion, les défis les plus importants que nous avons rencontrés se situaient dans le développement des programmes 2 et 3. Ce projet a été très exigeant en

termes de temps et d'effort, notamment dans la programmation de la structure complexe du super morpion et la résolution des problèmes de segmentation fault. De plus, la coordination et la communication sur nos programmes se sont avérées complexes, en particulier à cause de la distance entre nous. Ces défis ont rendu le projet à la fois difficile et instructif.