



PROJET - PROGRAMMATION AVANCÉS ET STRUCTURE DE
DONNÉES EN C
RAPPORT

Projet SDD : Puissance 4

Élèves :

Bouirdi SALMA

Aïssa PANSAN

Enseignant :

[John] CHAUSSARD

11 février 2023

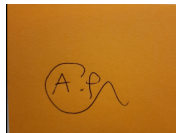
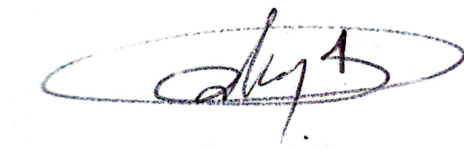
Sommaire

1	Engagement de non plagiat	2
2	Introduction générale	3
2.1	<u>Explication du jeu</u>	3
3	Développement du jeu	4
3.1	Interface utilisateur	4
3.2	Représentation graphique de la grille	4
3.3	Amélioration apportée : Colorisation de la grille et des joueurs	5
3.4	Modélisation de la grille	5
3.5	Placement du jeton	6
3.6	Test de gain	7
3.6.1	Principe	7
3.7	Difficultés rencontré	8
4	Codage de l'IA	10
4.1	Explication de la structure de données	10
4.1.1	La fonction clonning	11
4.2	Phase de l'algorithme du Minimax : Établissement de l'arborescence . . .	12
4.3	Implémentation en C	13
4.4	Détermination des enfants	13
4.5	Stratégie de parcours : DFS par récursivité	14
4.5.1	Difficultés rencontres	15
4.6	La fonction de score	17
4.7	Remonter des informations : L'algorithme du minimax	17
4.8	Principe	17
4.8.1	Difficultés rencontrées	19
5	Conclusion	20

1 Engagement de non plagiat

Nous, soussigné(e)s Aïssa Pansan et Bouirdi Salma, étudiant(e)s en 1er année d'école d'ingénieur à Sup Galilée, déclarons être pleinement conscient(e)s que la copie de tout ou partie d'un document, quel qu'il soit, publié sur tout support existant, y compris sur Internet, constitue une violation du droit d'auteur ainsi qu'une fraude caractérisée, tout comme l'utilisation d'outils d'Intelligence Artificielle pour générer une partie de ce rapport ou du code associé. En conséquence, nous déclarons que ce travail ne comporte aucun plagiat, et assurons avoir cité explicitement, à chaque fois que nous en avons fait usage, toutes les sources utilisées pour le rédiger. Fait à Drancy,Bagnolet , le 01/02/2024

Signatures :

A square orange stamp containing a handwritten signature in black ink, which appears to be 'A.P.'.A handwritten signature in black ink, which appears to be 'Bouirdi Salma', enclosed within a hand-drawn oval.

2 Introduction générale

Nous avons l'honneur de vous présenter le rapport de notre projet de programmation avancée en C et structures de données, réalisé en tant en 1re année d'ingénieur informatique.

Ce projet vise à développer un jeu de **Puissance 4** en langage C. Le projet comporte de nombreux enjeux, comme la représentation de la grille du jeu, le choix d'une structure de données adapté au jeu, l'implémentation d'une IA à l'aide d'une toute nouvelle structure de donnée et l'implémentation d'un algorithme pour l'IA. Face à ces différents enjeux, nous avons utilisé plusieurs outils pour contrôler et déboguer notre programme (valgrind, gdb...). Nous nous sommes notamment inspirés des structures de données déjà vu en cours tel que les listes chaînées pour l'arbre.

Notre rapport va présenter les détails de notre travail, en expliquant les décisions prises et les difficultés rencontrées, le développement. Il a également pour objectif de synthétiser notre démarche de résolution en détaillant succinctement les étapes clé du projet.

2.1 Explication du jeu

Le but du projet est de programmer le célèbres jeux de **Puissance 4**. Le principe du jeu, qui se joue à deux, consiste à faire à aligner 4 jetons de sa couleur dans une grille de jeu. On peut réussir à aligner ses jetons dans plusieurs directions (Selon une ligne, une colonne ou selon les diagonales). C'est toutes les possibilités de placement et les stratégies à employer pour bloquer son adverse qui rend ce jeu aussi intéressant. La version classique se joue avec une grille physique et des jetons de deux couleurs.

Dans le cadre de notre projet, nous allons développer une version du **Puissance 4** qui doit respecter toutes les contraintes du véritable jeu (Possibilité de placement des jetons, Gestion des joueurs...). De plus, notre programme comprendra deux modes de jeu : Multijoueurs (2 Joueurs humains) et un mode solo (Un humain contre une intelligence artificielle), l'IA est codée selon un algorithme qui sera présenté plus tard.

3 Développement du jeu

La programmation du jeu va passer par plusieurs phases. En somme, quand le joueur lancera une partie, voici l'action qui lui seront possibles (et aussi les actions effectuées dans le fond) On explique les grandes étapes du codage, à savoir.

0. On demande la taille de la grille 1. On contrôle la saisie du jeu 2. On choisit le type de jeu 3. On choisit où placer le jeton 4. On contrôle la saisie 5. On pose le jeton 6. On contrôle dans le même temps, si le coup est valide 7. Affiche la grille 8. On teste si le joueur à gagner avec ce coup 9. On recommence sinon.

3.1 Interface utilisateur

La 1re étape, incontournable de ce projet, fût d'élaborer l'interface utilisateur permettant de jouer au **Puissance 4 TM**. Il a été choisi d'abord d'interroger le joueur sur la taille de la grille. Le joueur peut choisir fixer librement les lignes et les colonnes de la grille du **Puissance 4 TM**. Cela dit, nous avons aussi implémenté un certain nombre d'instructions de contrôle afin de contrôler la saisie. De ce fait, il est impossible de :

Après avoir enregistré ces choix, l'utilisateur peut choisir son mode de jeu (Duo ou solo), lire les instructions ou quitter le jeu

Voici un récapitulatif du fonctionnement du menu selon un diagramme d'activité

- Écrire d'autres caractères que des chiffres, cela a été rendu possible en contrôlant le nombre de caractères lu via la fonction **scanf**
- Choisir une grille trop petite (Min 4*4)
- Choisir une grille trop grande, limité à du 15*15 (Sujet à modification en fct de l'IA)

3.2 Représentation graphique de la grille

La représentation de la grille de jeu se fait entièrement via l'affichage de la console. Afin, de s'approcher au maximum du véritable design d'une grille de **Puissance 4**, nous

avons décidé d'utiliser des caractères '=' pour les lignes et de '|' pour les colonnes. En répétant le même pattern suivant :

3.3 Amélioration apportée : Colorisation de la grille et des joueurs

En utilisant un formatage spécial de **printf**, on a pu après coup coloriser la grille avant de la rendre plus lisible.

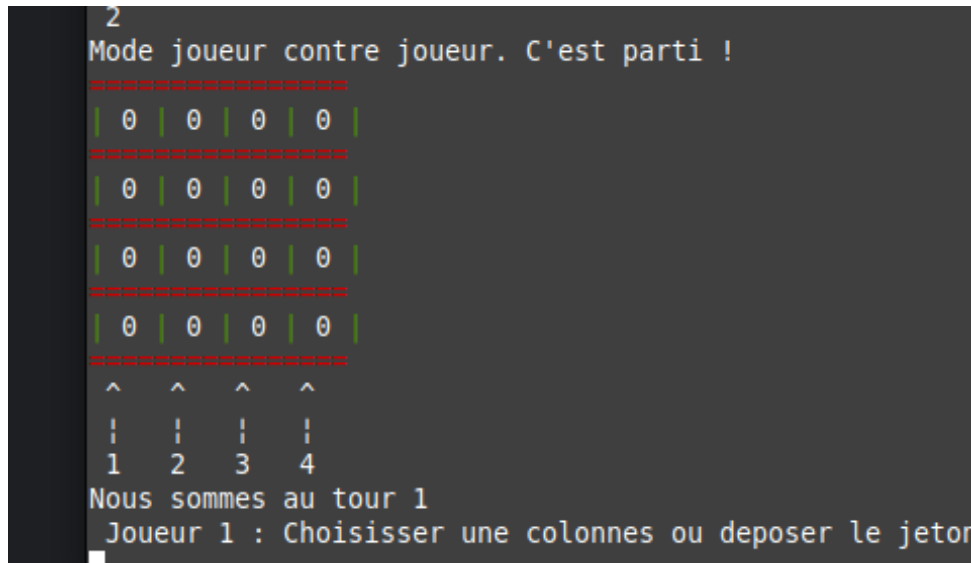


FIGURE 1 – Illustration d'un DFS sur un arbre

3.4 Modélisation de la grille

Dans notre projet, la grille de jeu est représentée par une structure de données "place". Cette structure contient un tableau 2D de type int de dimensions "lignes * colonnes", où chaque case peut avoir exactement trois valeurs : '0' pour une case non occupée, '1' pour une case occupée par le joueur 1, et '2' pour une case occupée par le joueur 2. Chaque case du tableau stocke également sa position (i, j) dans la grille. La structure "place" est ensuite référencée par un pointeur double dans une structure nommée "grille", ce qui facilite la manipulation de la grille.

Il faut aussi garder à l'esprit que l'indexage du tableau est représenté ainsi :

Dans le jeu Puissance 4, la gestion de la grille se fait via trois fonctions principales :

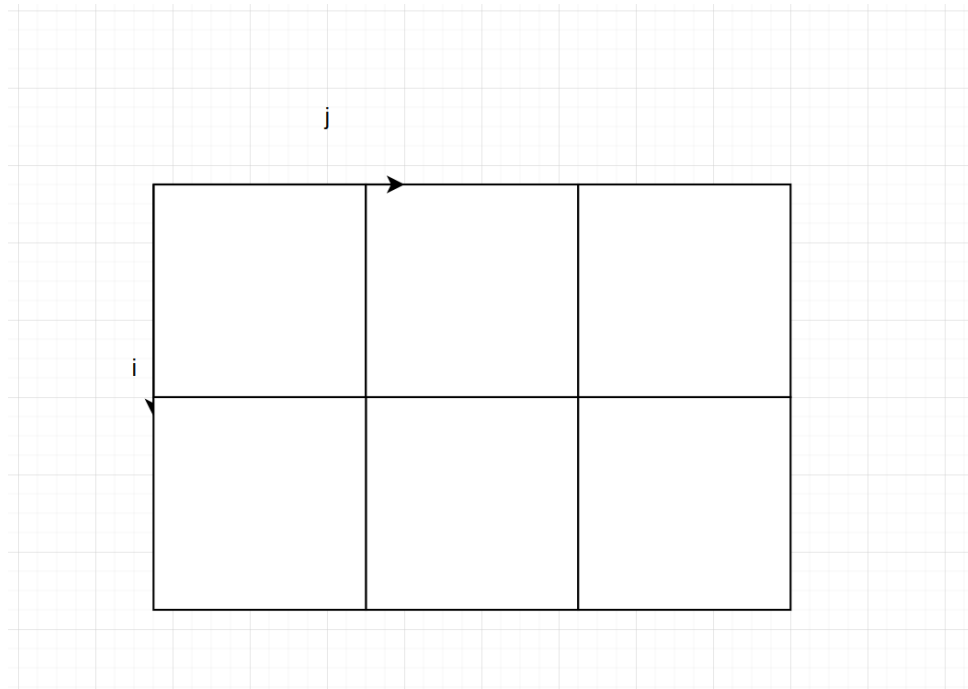


FIGURE 2 – Illustration d'un DFS sur un arbre

1) `alloc_grille` : Cette fonction alloue la mémoire pour une structure grille, définit ses dimensions en termes de lignes et colonnes, et initialise la grille.

2) `init_grille` : Elle s'occupe d'allouer un ensemble de pointeurs de type `place` pour chaque ligne de la grille. Pour chaque ligne, un tableau contenant les structures `place` est alloué pour représenter les colonnes, initialisant toutes les cases à 0.

3) `free_grille` : Cette fonction libère la mémoire allouée pour la grille, de la manière standard de libération.

3.5 Placement du jeton

Durant son tour dans le jeu, le joueur sélectionne une colonne pour y placer son jeton. Ainsi, le jeton doit descendre dans la colonne choisie jusqu'à rencontrer soit le fond de la grille, soit un autre jeton déjà placé. (Toute valeur différente de 0). Pour le placement d'un jeton, on va vérifier pour la colonne donnée si la case de la ligne en bas est vide tout en vérifiant "value". Si elle est occupée (valeur non nulle), on monte d'une ligne et on répète la vérification. Si toutes les lignes de la colonne sont occupées, le joueur doit choisir une autre colonne. Ce processus se déroule à l'aide d'une boucle (`while`) jusqu'à trouver une

case libre pour placer le jeton. La fonction **is_Already_Occupied** va renvoyer la ligne avec laquelle on peut placer le jeton. Sinon, on renvoie -1.

À la fin, la fonction **place_chip** va affecter dans la grille la position de la ligne renvoyée par **is_Already_Occupied** elle renverra 0 pour pouvoir sortir de la condition du while de la boucle de jeu. Mais si cette fonction avait renvoyé -1, on aurait averti l'utilisateur qu'il ne peut placer le jeton, auquel cas, le joueur devra recommencer la saisie dans la boucle de jeu.

3.6 Test de gain

3.6.1 Principe

La fonction `_Bool test_win2(grille* G, int32_t player)` est conçue pour vérifier si le joueur spécifié a gagné dans un jeu de grille à chaque tour . La fonction commence par identifier la position actuelle du dernier jeton placé, ensuite, elle examine toutes les directions possibles (horizontale), j(vertical) ou i+j/i-j(diagonal). On a donc 4 directions à teste.Pour chaque direction, elle compte les jetons consécutifs appartenant au joueur en parcourant de la direction positive puis dans la direction opposée . La fonction `offset_index` est employée pour modifier la position de recherche en fonction de la direction considérée. La vérification s'achève, si un jeton n'appartient pas au joueur ou si la limite de la grille est atteinte (vérifiée par la fonction "bord") . Si la fonction trouve un alignement de 4 jetons ou plus, elle affiche que le joueur a gagné et renvoie' 0 . Dans le cas où aucun alignement gagnant n'est trouvé après l'exploration de toutes les directions , la fonction revoie '1' et indique que personne n'a gagné .

*La fonction `offset_index` ajuste l'indexation des positions dans la grille en fonction de la direction spécifiée. * la fonction `_Bool bord` est utilisée pour s'assurer que la recherche reste dans les limites de la grille.

3.7 Difficultés rencontré

Durant notre étude du projet, on a rencontré des difficultés lors de l'implémentation de ce code, en effet comme le montre l'image lors teste d'exécution, on a remarqué que le résultat donné pour certains cas est contradictoire, donc on a modifié la fonction `est_win` afin qu'elle traite tous les cas particuliers.

```

deposer le jeton
4
=====
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=====
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=====
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=====
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
=====
| 1 | 2 | 1 | 2 | 0 | 0 | 0 |
=====
| 1 | 1 | 2 | 1 | 2 | 0 | 0 |
=====
| 1 | 1 | 2 | 2 | 2 | 1 | 0 |
=====
^   ^   ^   ^   ^   ^   ^
Â!  Â!  Â!  Â!  Â!  Â!  Â!
1   2   3   4   5   6   7
Personne n'a gagner pour le moment
Nous sommes au tour 17
Joueur 1 : Choisir une colonnes ou
deposer le jeton
4
=====
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=====
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=====
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
=====
| 2 | 0 | 0 | 1 | 0 | 0 | 0 |
=====
| 1 | 2 | 1 | 2 | 0 | 0 | 0 |
=====
| 1 | 1 | 2 | 1 | 2 | 0 | 0 |
=====
| 1 | 1 | 2 | 2 | 2 | 1 | 0 |
=====
^   ^   ^   ^   ^   ^   ^
Â!  Â!  Â!  Â!  Â!  Â!  Â!
1   2   3   4   5   6   7
Joueur 1,  gagn!
Jeux termin! bravo !

```

FIGURE 3 – Problème rencontré sur la première version de test_win

4 Codage de l'IA

Après avoir finalisé le mode de jeu à deux joueurs, l'étape suivante consiste à introduire un mode solo, autrement dit joueur contre AI . Pour développer une IA compétente, il est primordial d'implémenter l'algorithme Minimax, dont le fonctionnement sera expliqué dans les parties suivantes du rapport. Mais avant d'en venir là, il est important d'expliquer la structure de donnée choisie pour programmer cette IA.

4.1 Explication de la structure de données

La première étape de l'algorithme du Minimax est de construire, l'arborescence de tous les coups que peut jouer l'IA. Pour ce faire, on a décidé d'utiliser une structure d'arbre. L'avantage d'utiliser une telle structure est qu'elle nous permet de représenter plus facilement toutes les possibilités d'un nœud, à savoir tous les coups à jouer. De plus, le parcours d'une telle structure peut être facilité par l'emploi de fonction récursif, malgré leur coût élevé en mémoire. L'implémentation s'en retrouve plus facilitée

Nous avons implémenté une structure de nœud d'arbre appelé **node**, elle contient :

- Un champ **score** (entier), pour sauvegarder le score d'un nœud issu de l'algorithme du minimax.
- Un pointeur sur un type **place_case** qui pointe vers les coordonnées du coup joué.
- Un pointeur sur un type **grille** : Ce pointeur est essentiel pour avoir une représentation de la grille pour nœud donné. Il est à préciser que l'on pointe vers une copie de la grille et non l'original
- Un entier **height** qui représente la hauteur du nœud dans l'arborescence, permet de contrôler la profondeur de l'exploration.
- Un pointeur de type **node** vers le nœud parent
- L'entier **nb_child** dénombre le nombre d'enfants d'un nœud
- Un pointeur double vers un tableau de pointeur **node**. Ce tableau représente les enfants du nœud parent
- **leaf** est un booléen qui indique si le nœud est une feuille. Il est à zéro si ce n'est

pas le cas et à un sinon.

- `copied_grill` : (type grille) une copie de l'état actuel de la grille, permettant des simulations sans affecter le jeu original.

Par ailleurs, de nombreuses fonctions accompagnent cette structure afin d'initialiser les nœuds en fonction de leur état initial

init_tree :

Entrées : `grille*` G

Sortie : `node*`

Fonctionnement : Initialise le noeud racine. Le pointeur `copied_grill` pointe vers la grille du jeu, sans le cloner.

init_child :

Entrées : `grille*` G, `place_case*` x, `int32_t` player, `node*` parent

Sortie : `void`

get_number_of_child :

Entrées : `grille*` G

Sortie : `int32_t`

Fonctionnement : Cette fonction renvoie le nombre d'enfants, associé à la grille donnée en paramètre, qu'on a pu trouver en bouclant sur chaque colonne de la grille et en testant si la fonction `is_Already_Occupied` ne renvoie pas -1 (Cas d'une colonne pleine). Si telle n'est pas le cas, on incrémente un compteur qu'on renvoie à la fin de la boucle.

4.1.1 La fonction clonning

Nous avons plusieurs fois parler du fait qu'un nœud devait posséder une copie de la grille pour constituer l'arborescence. Cela dit, on ne peut se contenter de pointer directement tous les champs `copied_grill` vers G (auquel cas, chaque changement via le nœud

se répercutera sur G). On ne peut aussi utiliser la fonction **memcpy** qui permet d'allouer une nouvelle espace mémoire est de copier les champs de la source vers la destination, car notre structure **grill** contient des doubles pointeurs vers **grill_place**, le problème va ainsi se répéter.

La seule solution est ainsi de réallouer de la même manière la grille copiées que la fonction **alloc_grille** et **init_grille**. La seule différence étant que l'on va copier les champs de la grille source vers la grille destinations.

Des fonctions essentielles pour la construction et la gestion de l'arbre de décision sont utilisé dans l'algorithme Minimax pour l'IA du jeu. La fonction "init_tree" est responsable de l'initialisation de l'arbre en créant le nœud racine, avec une grille clonée pour permettre des simulations indépendantes. La fonction "cloning" revêt une importance capitale, car elle crée une copie exacte de la grille de jeu, permettant à l'IA d'explorer différentes possibilités sans altérer l'état réel du jeu. En outre, les fonctions "init_child" et "get_child" sont utilisées pour générer et attacher les nœuds enfants au nœud parent, représentant les différents coups possibles dans le jeu. Ces enfants sont générés en tenant compte des colonnes disponibles pour le placement des jetons. Enfin, la fonction "get_number_of_child" calcule le nombre de coups possibles à partir d'une configuration de grille donnée.

4.2 Phase de l'algorithme du Minimax : Établissement de l'arborescence

Le 1er étapes de l'algorithme du Minimax est de représenter l'arborescence des coups. On doit pouvoir avoir un arbre sur 5 étages, ou chacun des nœuds représente une possibilité de placement d'un jeton. Le 1er enfant d'un parent représenterait la grille du parent avec un jeton à la 1er colonnes et ainsi de suite.

4.3 Implémentation en C

L'IA suit les mêmes règles que le joueur, elle ne peut placer un jeton dans une colonne que si cette dernière est bien dans la grille ou si la colonne n'est pas pleine. De ce fait, on va pouvoir utiliser la fonction `is_Already_Occupied` pour tester si l'on peut mettre un jeton dans une colonne particulière. La première étape à déterminer est de trouver les enfants d'un nœud.

4.4 Détermination des enfants

Pour déterminer le tableau de `node**` d'enfant d'un nœud, il a d'abord fallu déterminer le nombre d'enfants lié à ce nœud, grâce à la fonction `get_number_of_child`. Par la suite, on alloue un tableau de `n node*`, ou `n` équivaut aux nombres de colonnes de la grille. On aurait pu allouer exactement `nb_child` de cases dans le tableau afin d'éviter de gaspiller de la mémoire. Cela dit, cela aurait causé des problèmes d'indexage de boucles, entre les positions des enfants n'ayant aucun coup possible et les autres. Il a alors été décidé de traiter deux cas :

- Si l'enfant admet une ligne via `is_Already_Occupied`, alors c'est qu'il représente un coup jouable. On enregistre la position du coup et on initialise l'enfant avec `init_child`
- Soit `is_Already_Occupied` vaut -1 et le coup n'est pas jouable. Dans ce cas, l'enfant est initialisée à `NULL`

```
for (uint32_t j = 0; j < max; j++) //i represente la colonnes
{
    parent->child[j] = (node *) malloc(sizeof(node)); // On alloue chaque
    enfant
    assert(parent->child[j]);

    line = is_Already_Occupied(parent->copied_grill, j, 0);

    if (line != -1) // On peut mettre le jeton dans une colonnes
    {
        x.lignes = line;
    }
}
```

```
x.colonnes = j;

init_child(parent->child[j], x, player, parent); // On met a jour
la position avec une grille copier
}

else //L'enfant ne donne aucune configuration
{
    parent->child[j] = NULL;
}

}
```

4.5 Stratégie de parcours : DFS par récursivité

On a réussi à déterminer pour nœud tous ces enfants, mais quid des enfants ? Il existe plusieurs stratégies pour développer l'arborescence de notre arbre au-delà de la racine, la plus simple étant un DFS par récursivité.

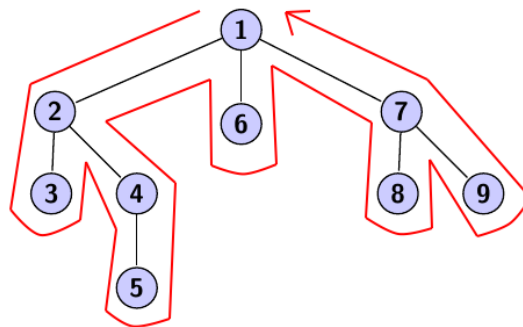


FIGURE 4 – Illustration d'un DFS sur un arbre

Le **Deep First Search** ou parcours en profondeur est une stratégie de parcours de graphes qui consiste à toujours visiter le 1er voisin rencontré d'un sommet courant. On remonte dans notre arborescence si le sommet n'a plus de voisins. C'est exactement le même principe que nous allons utiliser pour déterminer l'arborescence de l'arbre. Notre condition d'arrêt, dans notre cas, s'apparenterait à tester si le nœud courant est à **NULL**, ou bien si on atteint une hauteur de 5.

Le DFS est représenté par un appel récursif sur le premier enfant, en décrémentant la hauteur de -1.

4.5.1 Difficultés rencontrées

La version telle que présenté dans le rapport ne fut adopté directement. En effet, en raison du coup lié à des appels récursif, il pouvait être possible d'utiliser d'autre type de parcours sans récursivité. C'est notamment le cas du BFS(Breadth First Search) ou parcours en largeur d'abord. Ce type de parcours peut être réalisé par une structure de type file modifié qui prend en arguments comme donnée un pointeur sur un nœud. On alloue une file et on commence par ajouter dans la file dans la racine. Dans une boucle while, on ajoute dans la file par la queue chaque enfant du nœud courant. Une fois qu'on a déterminé tous les enfants d'un nœud, on l'enlève de la liste et on recommence.

```
void BFS_make(node* root)
{
    int32_t player = 2,i; // Player = 2 »IA || Player = 1 » Humain
    file* FIFO = init_file();
    add_tail(root,FIFO);
    while (est_vide(FIFO))
    {
        node* Node = get_node(FIFO);
        if( *(Node->child) == NULL )
        {
            printf("Alert noeud non initialis. Skip\n");
            nuke_tete(FIFO);
            continue;
        }
        else
        {
            ;
        }
    }
}
```

```
    if(get_child2(Node,player)) // On ajoute des enfant dans la liste
        uniquement si on trouve des enfants
    {
        for ( i = 0; i < Node->nb_child ; i++)
        {
            add_tail(Node->child[i],FIFO);
        }
    }
    nuke_tete(FIFO);
}
}
```

Cependant, malgré plusieurs tests et de nombreuses sessions de débogage, à l'aide de **gdb**. Nous n'avons pas réussi à faire fonctionner ce type de structure. Lors du parcours **gdb** nous indiquent qu'on manipule des nœuds non initialisés, ce qui provoqua des segmentation fault.

4.6 La fonction de score

La fonction de score est cruciale pour la performance de l'IA dans le jeu. En effet, elle évalue la configuration de la grille en vérifiant tous les segments de quatre cases consécutives, (lignes, colonnes ou diagonales). Le score attribué a chaque tour, indique sa pertinence stratégique, ce qui permet à l'IA de prendre les meilleures décisions possibles.

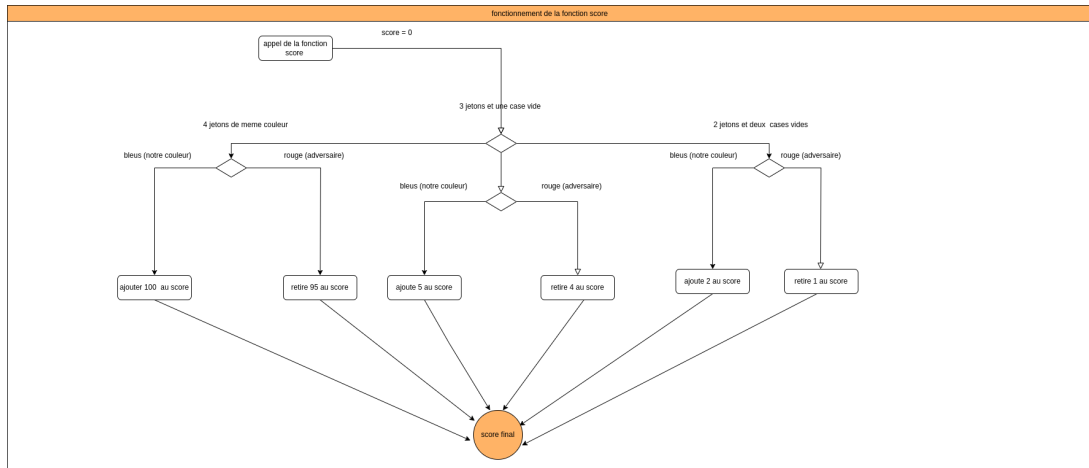


FIGURE 5 – Schéma du calcul de score

4.7 Remonter des informations : L'algorithme du minimax

Maintenant que nous avons à notre disposition un arbre avec tous les coups possible et fonction qui calcule le score d'une configuration de grille, il faut appliquer l'algorithme minimax en lui-même.

4.8 Principe

L'algorithme peut être décrit de la manière suivante :

```
int32_t minimax(node* N, int32_t depth, int32_t player, int32_t* best_col) {  
    if (depth == 0 || N->leaf) {  
        return evaluerGrille(N->copied_grill);  
    }  
}
```

```
if (player == 1) { // Maximiser pour le joueur 1
    int32_t best_score = INT32_MIN;
    for (int32_t i = 0; i < N->nb_child; i++) {
        if (N->child[i] != NULL) {
            int32_t current_score = minimax(N->child[i], depth - 1, 2,
                best_col);
            if (current_score > best_score) {
                best_score = current_score;
                *best_col = i; // Mettre à jour la meilleure colonne
            }
        }
    }
    return best_score;
} else { // Minimiser pour le joueur 2
    int32_t best_score = INT32_MAX;
    for (int32_t i = 0; i < N->nb_child; i++) {
        if (N->child[i] != NULL) {
            int32_t current_score = minimax(N->child[i], depth - 1, 1,
                best_col);
            if (current_score < best_score) {
                best_score = current_score;
                *best_col = i; // Mettre à jour la meilleure colonne
            }
        }
    }
    return best_score;
}
}
```

Un nœud est considéré comme une feuille grâce s'il n'a aucun enfant.

La fonction comparé va permettre de déterminer pour chaque nœud qui n'est pas une feuille le score minimum entre INF_MAX (qui équivaut à -999) et le score puis le meilleur score et la valeur renvoyée par minimax, uniquement dans le cas du joueur 1. Pour le joueur 2, on fait l'inverse.

À la fin de la récursion, on revient à la racine, on boucle sur tous les enfants jusqu'à obtenir l'enfant qui nous a permis d'obtenir le score de la racine. On cherche la position de sa colonne et on la renvoie.

C'est avec cet indice de colonne qu'on va faire jouer l'IA pour chaque tour

4.8.1 Difficultés rencontrées

L'implémentation finale de l'algorithme du minimax donne un résultat ambiguë. Lors d'un tour de jeu, l'IA choisit systématiquement la colonne située juste avant celle ou on a posé notre jeton. Ce comportement fait que le tour de l'IA est sauté, si l'on choisit la colonne 1. Enfin, l'IA va tenter de mettre un jeton à notre gauche, même si la colonne est remplie, ce qui va conduire à une erreur.

voici un test final du notre code :

```
L'IA joue...
le score de L'IA est -104
=====
| 0 | 0 | 0 | 2 |
| 0 | 0 | 0 | 2 |
| 0 | 0 | 0 | 2 |
| 1 | 1 | 0 | 2 |
| 1 | 1 | 0 | 2 |
=====
^   ^   ^   ^
|   |   |   |
1   2   3   4
L'IA à placé son jeton à la 3-ieme colonne
Joueur 2, à gagné
Jeux terminé bravo !
```

FIGURE 6 – test Final

5 Conclusion

Le projet de développement d'un jeu Puissance 4 a été une occasion enrichissante pour l'application de concepts avancés de programmation et d'intelligence artificielle. L'implémentation de l'algorithme Minimax, combinée avec une fonction d'évaluation soigneusement élaborée, a permis de créer une IA compétitive et stratégique. En effet, Les difficultés rencontrées, notamment dans l'optimisation des performances et le contrôle des états de jeu, ont contribué à une compréhension plus approfondie des structures de données et des algorithmes. Ce projet met en évidence l'importance d'une planification rigoureuse et d'une réflexion critique dans le développement de solutions logicielles complexes. Ainsi, Les compétences et connaissances acquises au cours de ce projet constituent un atout précieux pour relever les futurs défis en domaine professionnel de l'ingénierie informatique.