Université Sorbonne Paris Nord
Sup Galilée
Spécialité Informatique

Matière : Structures de données

Développement du Jeu "Puissance 4" contre une IA

Étudiants :
Anis TRABELSI
Zeineb BOUJMIL

Enseignant :
John CHAUSSARD

31 janvier 2024

Engagement de non-plagiat

Nous, soussigné(e)s Anis Trabelsi et Zeineb Boujmil, étudiant(e)s en 1ere année spécialité Informatique 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 Paris , le 31/01/2024

Signatures :

Table des matières

1 Introduction

Ce rapport est le fruit d'un parcours académique dédié à l'étude approfondie de la matière structure de donnée , une branche essentielle de notre formation , à l'école d'ingénieur sup galilée . L'objectif de ce document est de synthétiser les connaissances acquises et de les appliquer à un cas pratique, Jeu puissance 4, illustrant ainsi la synergie entre théorie et pratique.

Le développement d'un jeu de Puissance 4 constitue le cœur de notre projet. La première partie de notre rapport est consacrée à la mise en place d'un jeu qui permet à deux joueurs de s'engager dans une lutte stratégique sur ce tableau classique (grille). Nous exposons minutieusement les choix de conception et les structures de données qui facilitent une expérience utilisateur fluide et réactive.
Progressant vers une innovation majeure, la seconde partie du rapport introduit un nouvel adversaire : une intelligence artificielle (IA) sophistiquée. Conçue pour rivaliser avec l'acuité tactique humaine, cette IA s'appuie sur l'algorithme Minimax, pour évaluer les coups et anticiper les stratégies adverses. En partageant le développement de cette IA, nous dévoilons la fusion entre les théories informatiques et l'intelligence artificielle pour forger un compétiteur virtuel redoutable.

2 Première partie : Puissance 4 à 2 Joueurs

Pour développer ce jeu, nous avons opté pour une structure de données personnalisée, nommée "tabPuissance4", qui encapsule les dimensions de la grille de jeu et la grille qui est un tableau de deux dimensions (une matrice) de caractères pour bien pouvoir stocker les couleurs comme 'R' et 'J' .
En intégrant les dimensions comme des paramètres configurables au sein de cette structure , nous favorisons une expérience utilisateur plus interactive et personnalisable, tout en conservant une fondation solide pour les mécanismes du jeu.

2.1 Diagrammes

Pour décortiquer et résoudre efficacement le problème de la détection des alignements de jetons dans le jeu Puissance 4, nous avons choisi d'utiliser **des diagrammes d'activité**. Ces outils de modélisation visuelle sont particulièrement efficaces pour représenter le flux de contrôle ou le flux de données à travers diverses activités et actions. Ils nous permettent de visualiser clairement le déroulement logique de nos fonctions.
Pour simplifier les choses et clarifier la vision des diagrammes on a décomposé ces derniers en parties . Commancant par le diagramme principale de la fonction main :

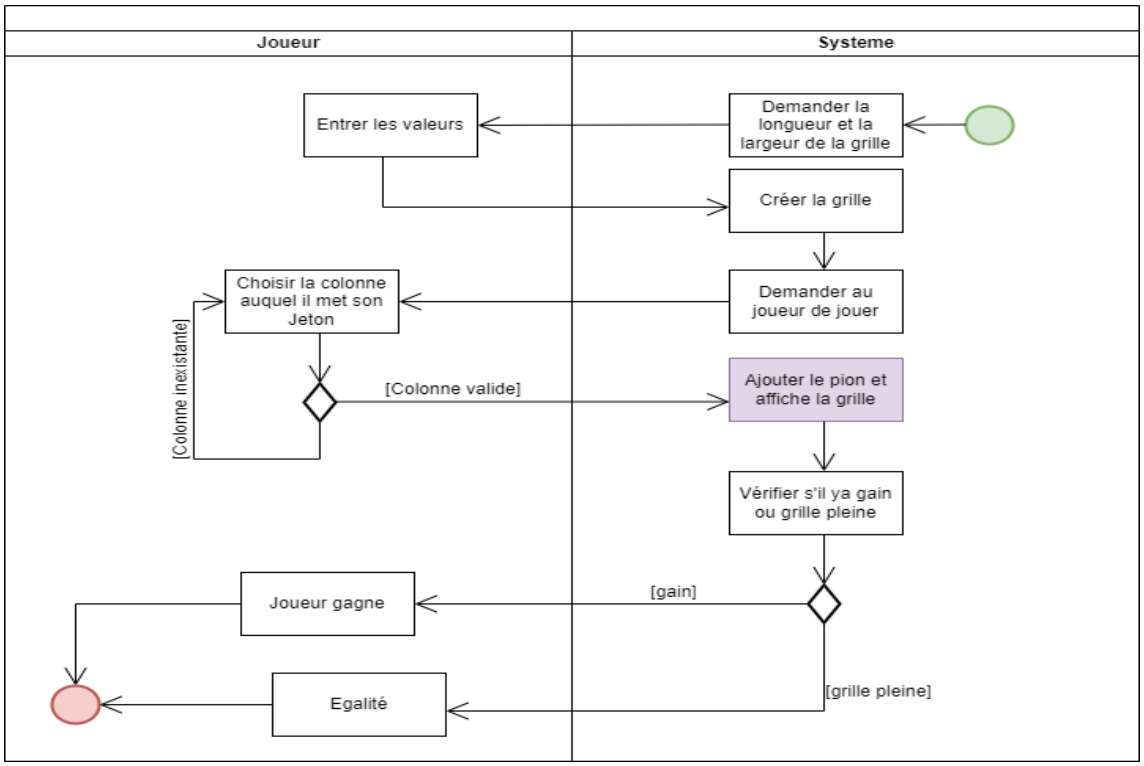


FIGURE 1 – Diagramme d'activité du jeu « Puissance 4 » entre deux joueurs

Suivant les couleurs , on présente le diagramme de la fonction add_dans_grille

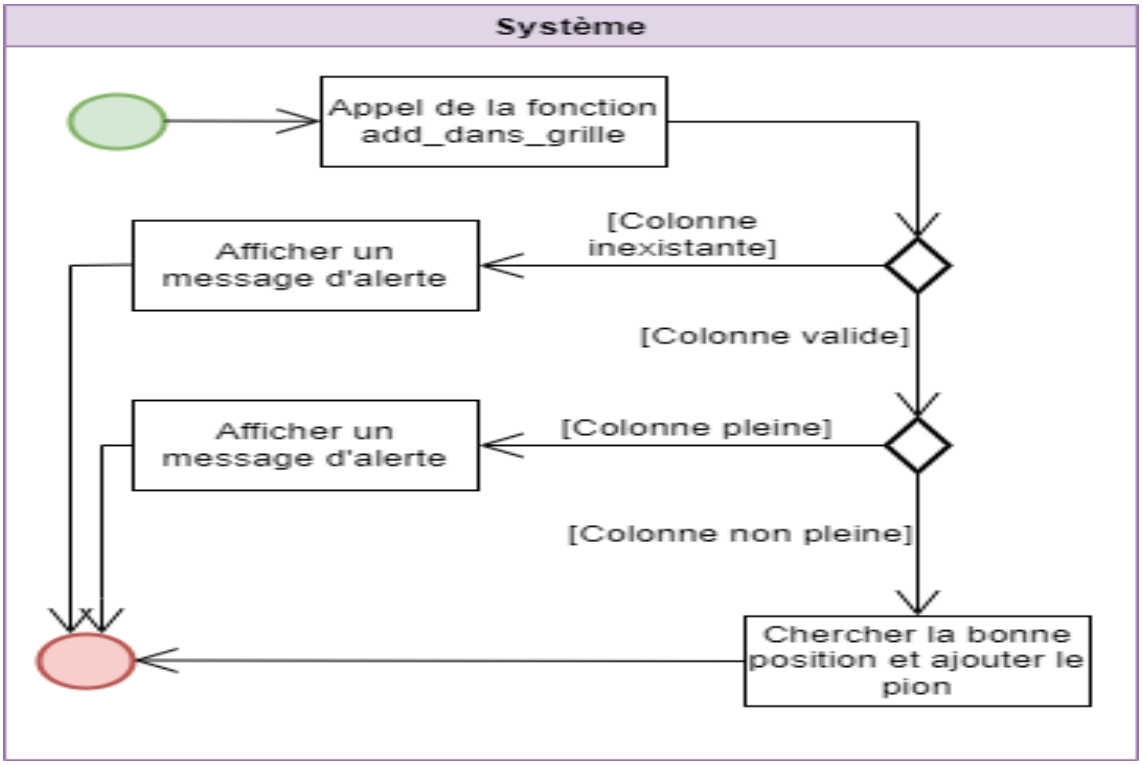


FIGURE 2 – Diagramme d'activité de la fonction « add_dans_grille »

2.2 Code

Listing 1 – La structure tabPuissance4

```
1 typedef struct _tabPuissance4
2 {
3     char **grille;
4     uint32_t largeur;
5     uint32_t hauteur;
6 } tabPuissance4;
```

On commence par allouer et initialiser notre grille par le caractère 'n' comme vide ce qui va nous servira après ..

Listing 2 – Allocation et Initialisation de la structure

```
1 tabPuissance4 *new_tabPuissance4(uint32_t largeur, uint32_t hauteur
    )
2 {
3     tabPuissance4 *tp4 = malloc(sizeof(tabPuissance4));
4     assert(tp4 != NULL);
5
6     tp4->grille = malloc(largeur * sizeof(char *));
7     assert(tp4->grille != NULL);
```

```
8
9       for (uint32_t i = 0; i < largeur; i++)
10      {
11          tp4->grille[i] = calloc(hauteur, sizeof(char));
12          assert(tp4->grille[i] != NULL);
13
14          // Initialiser chaque element de la colonne avec 'n'
15          for (uint32_t j = 0; j < hauteur; j++)
16          {
17              tp4->grille[i][j] = 'n';
18          }
19      }
20
21      tp4->largeur = largeur;
22      tp4->hauteur = hauteur;
23
24      return tp4;
25  }
```

Avançons désormais vers la présentation de l'implémentation des fonctionnalités fondamentales, celles qui constituent la pierre angulaire de notre développement .
On commence par la fonction "**add_dans_grille**" qui prend en paramètre la structure tabPuissance4 , l'indice du colonne à laquelle on ajoute le pion dans la premiere ligne vide , et un caractère qui désigne le premier caractère de la couleur du jeton du joueur en question.

Listing 3 – La fonction add_dans_grille

```
1   void add_dans_grille(tabPuissance4 *tp4, uint32_t indice_colonne,
        char c)
2   {
3       if (indice_colonne >= tp4->largeur)
4       {
5           fprintf(stderr, "Indice␣de␣colonne␣invalide.\n");
6           return;
7       }
8
9       int32_t j = tp4->hauteur - 1;
10
11      while (j >= 0 && tp4->grille[indice_colonne][j] == 'n')
12      {
13          j--;
14      }
15
16      if (j >= 0)
17      {
18          tp4->grille[indice_colonne][j + 1] = c;
19      }
20      else
21      {
22          tp4->grille[indice_colonne][0] = c;
23      }
```

```
24 }
```

On a aussi les fonctions de vérification. Ces fonctions sont cruciales, car elles itèrent la grille de jeu afin de détecter des alignements de 4 jetons de même couleur passée en paramètre et consécutifs, que ce soit verticalement, horizontalement, ou diagonalement.

verif_vertical parcourt une seule colonne de la grille, du haut vers le bas jusqu'à ce qu'elle trouve 4 jetons de même couleur passée en paramètre et consécutifs. Dans le pire des cas, elle parcourt la hauteur totale H de la grille. La complexité est donc $O(H)$.

Listing 4 – La fonction verif_verticale

```
1  _Bool verif_verticale(tabPuissance4 *tp4, uint32_t indice_colonne)
2  {
3      int32_t j = tp4->hauteur - 1;
4
5      while (j > 0 && tp4->grille[indice_colonne][j] == 'n')
6      {
7          j--;
8      }
9
10     char couleur_cherchee = tp4->grille[indice_colonne][j];
11     uint32_t cpt_cases_contigues_verticales = 0;
12
13     for (int32_t i = tp4->hauteur - 1; i >= 0; i--)
14     {
15         if (tp4->grille[indice_colonne][i] == couleur_cherchee)
16         {
17             cpt_cases_contigues_verticales++;
18             if (cpt_cases_contigues_verticales >= 4)
19                 return 1;
20         }
21         else
22         {
23             cpt_cases_contigues_verticales = 0;
24         }
25     }
26
27     return 0;
28 }
```

De manière similaire à verif_verticale, **verif_horizontale** cette fonction une seule ligne, mais elle vérifie toutes les colonnes de cette ligne. Le nombre maximal d'itérations correspond à la largeur de la grille L .
Ainsi, la complexité est O(L).

Listing 5 – La fonction verif_horizontale

```
1  _Bool verif_horizontale(tabPuissance4 *tp4, uint32_t indice_colonne
       )
2  {
3      int32_t j = tp4->hauteur - 1;
4
```

```
5       while (j > 0 && tp4->grille[indice_colonne][j] == 'n')
6       {
7           j--;
8       }
9
10      char couleur_cherchee = tp4->grille[indice_colonne][j];
11      uint32_t cpt_cases_contigues_horizontales = 0;
12
13      for (uint32_t i = 0; i < tp4->largeur; i++)
14      {
15          if (tp4->grille[i][j] == couleur_cherchee )
16          {
17              cpt_cases_contigues_horizontales++;
18              if (cpt_cases_contigues_horizontales >= 4)
19                  return 1;
20          }
21          else
22          {
23              cpt_cases_contigues_horizontales = 0;
24          }
25      }
26
27      return 0;
28  }
```

La fonction de vérification en digonale , prend en paramètre tp4 qui est un pointeur vers la structure représentant le plateau de jeu et indice_colonne qui est l'indice de la colonne où le dernier jeton a été joué .

Premièrement , La fonction commence par trouver le jeton le plus bas dans la colonne spécifiée qui n'est pas vide (c'est-à-dire différent de 'n') et sera stockée dans la variable ligne_ajoutee .Ce travail se fait avec la boucle while à la ligne 8 du code présenté.

Deuxièment , la couleur du jeton trouvé sera stockée dans la variable couleur_cherchée Après , le calcule des indices minimum (col_min) et maximum (col_max) des colonnes qui pourraient potentiellement contenir un alignement de 4 jetons diagonaux avec le jeton trouvé.

A la ligne 34 du code , on a initialiser cpt1 , cpt2 , cpt3 , cpt4 a 1 qui sont respectivement, Compteur pour la Diagonale Haut Gauche , Compteur pour la Diagonale Bas Droite, Compteur pour la Diagonale Haut Droite , Compteur pour la Diagonale Bas Gauche . Après calcul des alignement et la mise à jour des valeurs des pointeurs , la ligne 104 renvoie un resultat tel qu'elle vérifie s'il y a alignement de 4 jetons conséqutifs directement sur une diagonal et s'il ya alignement en combinant les diagonales :
(cpt1 + cpt2 - 1) >= 4 : Vérifie s'il y a 4 jetons consécutifs ou plus en combinant les diagonales haut gauche et bas droite. Le -1 est nécessaire pour ne pas compter deux fois le jeton à l'intersection des deux diagonales (meme direction).

(cpt3 + cpt4 - 1) >= 4 : De même, vérifie pour la combinaison des diagonales haut droite et bas gauche.

En gros , Pour chaque direction diagonale, la fonction itère au maximum 3 fois dans chaque direction à partir de la pièce jouée pour vérifier si quatre pièces consécutives de la même couleur sont alignées. Cette valeur de "3" ne change pas quel que soit la taille de la grille ; elle est constante car elle est déterminée par les règles du jeu Puissance 4 qui requièrent un alignement de 4 pièces consécutives pour gagner.
Les boucles internes qui vérifient les diagonales s'arrêtent soit après avoir trouvé une pièce qui ne correspond pas ou après avoir compté jusqu'à 3 pièces dans une direction. Cela signifie que peu importe la taille de la grille, ces boucles ne dépendent pas de la taille de la grille (largeur ou hauteur). Elles sont donc considérées comme des opérations à temps constant, O(1), puisqu'elles effectueront au maximum 6 itérations (3 dans chaque direction pour deux directions opposées) pour chaque diagonale.

Listing 6 – La fonction verif_diagonale

```
1  _Bool verif_diagonale(tabPuissance4 *tp4, uint32_t indice_colonne)
2  {
3      uint32_t ligne_ajoutee = tp4->hauteur - 1, col_min, col_max;
4
5      // Trouver la hauteur de la derni re pi ce ajout e
6      char couleur_cherchee;
7
8      while (ligne_ajoutee > 0 && tp4->grille[indice_colonne][
           ligne_ajoutee] == 'n')
9      {
10         ligne_ajoutee--;
11     }
12
13     couleur_cherchee = tp4->grille[indice_colonne][ligne_ajoutee];
14
15     if (indice_colonne >= 3)
16     {
17         col_min = (indice_colonne - 3 > 0) ? indice_colonne - 3 :
               0;
18     }
19     else
20     {
21         col_min = 0;
22     }
23
24     if (indice_colonne + 3 < tp4->largeur)
25     {
26         col_max = (indice_colonne + 3 < tp4->largeur) ?
               indice_colonne + 3 : tp4->largeur - 1;
27     }
28     else
29     {
30         col_max = tp4->largeur - 1;
31     }
```

```
32
33      uint32_t k, j;
34      uint32_t cpt1 = 1, cpt2 = 1, cpt3 = 1, cpt4 = 1;
35
36      // diagonale haut gauche
37      k = indice_colonne;
38      j = ligne_ajoutee;
39      while (k > col_min && j > 0)
40      {
41          k--;
42          j--;
43          if (tp4->grille[k][j] == couleur_cherchee)
44          {
45              cpt1++;
46          }
47          else
48          {
49              break;
50          }
51      }
52
53      // diagonale bas droite
54      k = indice_colonne;
55      j = ligne_ajoutee;
56      while (k < col_max && j < tp4->hauteur - 1)
57      {
58          k++;
59          j++;
60          if (tp4->grille[k][j] == couleur_cherchee)
61          {
62              cpt2++;
63          }
64          else
65          {
66              break;
67          }
68      }
69
70      // diagonale haut droite
71      k = indice_colonne;
72      j = ligne_ajoutee;
73      while (k > col_min && j < tp4->hauteur - 1)
74      {
75          k--;
76          j++;
77          if (tp4->grille[k][j] == couleur_cherchee)
78          {
79              cpt3++;
80          }
81          else
```

```
82          {
83              break;
84          }
85      }
86
87      // diagonale bas gauche
88      k = indice_colonne;
89      j = ligne_ajoutee;
90      while (k < col_max && j > 0)
91      {
92          k++;
93          j--;
94          if (tp4->grille[k][j] == couleur_cherchee)
95          {
96              cpt4++;
97          }
98          else
99          {
100             break;
101         }
102     }
103
104     return (cpt1 >= 4 || cpt2 >= 4 || cpt3 >= 4 || cpt4 >= 4 || (
            cpt1 + cpt2 - 1) >= 4 || (cpt3 + cpt4 - 1) >= 4);
105 }
```

2.3 Validation et Test

Cette section est dédiée à l'évaluation approfondie de notre jeu à travers une série de tests rigoureux et de cas d'usage. L'objectif est de démontrer non seulement la fiabilité et la robustesse de notre solution, mais aussi sa pertinence et son efficacité face à des scénarios réels et variés.

```
PS C:\Users\zeineb\desktop\sdd\proj> gcc -Wall puissance_4.c -o prog
PS C:\Users\zeineb\desktop\sdd\proj> .\prog
Donner la largeur de la grille:
7
Donner la hauteur de la grille:
6
Joueur R, choisissez une colonne (0-6) : 1
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   | o |   |   |   |   |   |
 --- --- --- --- --- --- ---
  0   1   2   3   4   5   6

Joueur J, choisissez une colonne (0-6) : 2
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   | o | o |   |   |   |   |
 --- --- --- --- --- --- ---
  0   1   2   3   4   5   6

Joueur R, choisissez une colonne (0-6) : 1
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   | o |   |   |   |   |   |
|   | o | o |   |   |   |   |
 --- --- --- --- --- --- ---
  0   1   2   3   4   5   6
```

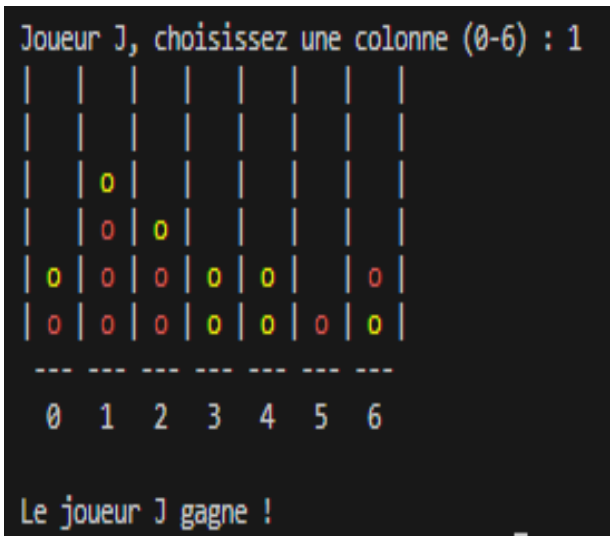
Continuons a jouer jusqu'a l'un des deux joueurs gagne :

Victoire verticale :

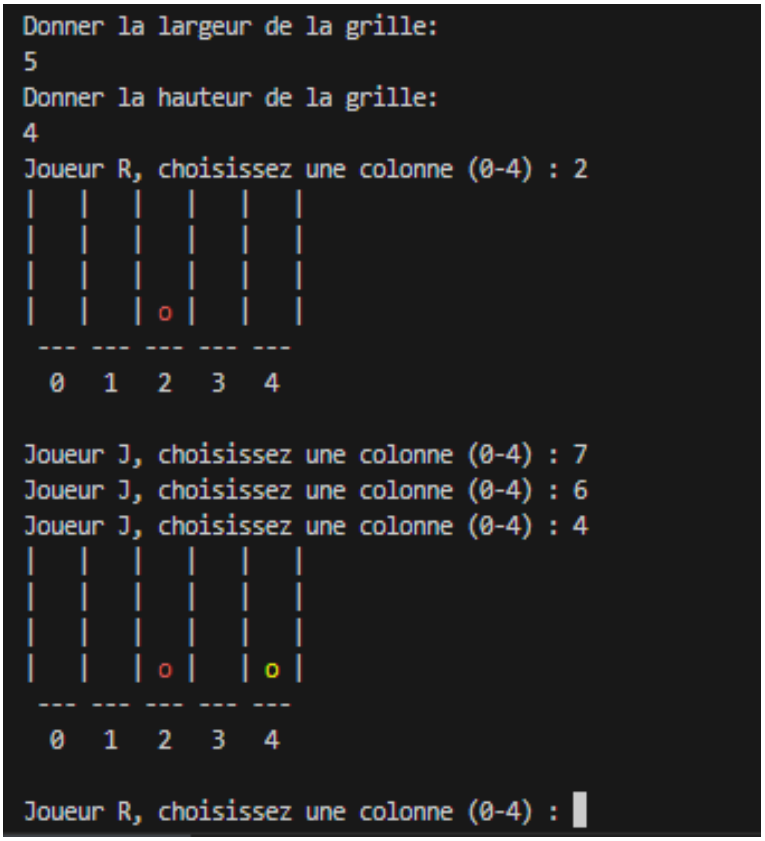
```
Joueur R, choisissez une colonne (0-6) : 1
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   | o |   |   |   |   |   |
|   | o | o |   |   |   |   |
|   | o | o |   |   |   |   |
| o | o | o | o |   |   |   |
 --- --- --- --- --- --- ---
  0   1   2   3   4   5   6

Le joueur R gagne !
```

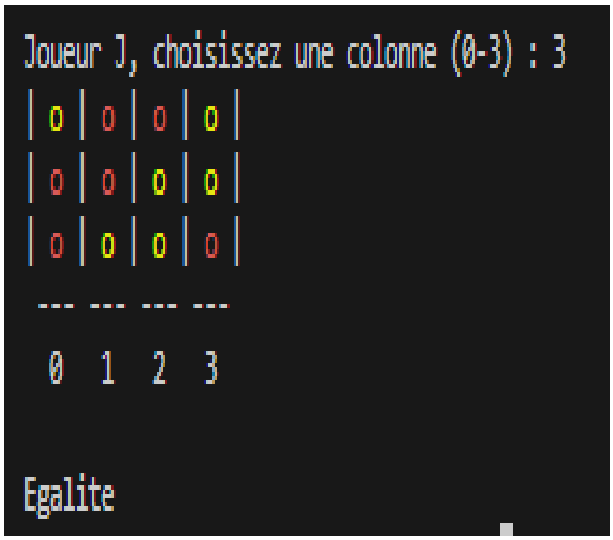
Victoire horizontale :



Entrons une valeur de colonne inexistante , le programme redemande d'entrer une autre :



Cas d'une égalité :



3 Deuxième partie : Puissance 4 avec une IA

Introduisons un nouveau défi dans notre jeu Puissance 4 : l'intégration d'une intelligence artificielle. Cette avancée transforme l'expérience de jeu en introduisant un adversaire virtuel, l'ordinateur, capable de rivaliser avec les joueurs humains. Dans les sections suivantes, nous détaillerons la conception et l'implémentation d'une IA sophistiquée qui, grâce à l'algorithme Minimax et une formule de calcule du score , peut anticiper les mouvements de l'adversaire et prendre des décisions stratégiques. Préparez-vous à plonger dans les détails de cette fascinante rencontre entre l'homme et la machine, où chaque mouvement compte et où la stratégie est reine.Dans cette partie , on combine la section Diagrammes et code vu la relation étroite entre eux .

3.1 Diagrammes et Code

Une nouvelle structure, appelée "Node", ayant pour objectif de créer un arbre contenant les différentes combinaisons de coups possibles provenant d'un état donné d'une grille, ainsi elle contient d'autres informations telles que le joueur en cours (sa couleur) et l'état d'un nœud (feuille ou non), le nombre des enfants du noeud en question.

Listing 7 – La structure Node

```
1 typedef struct Node
2 {
3     tabPuissance4 *grille;
4     struct Node **children;
5     int numChildren;
6     char currentPlayer;
7     bool isTerminal; // Ajout  pour marquer si le noeud est
          terminal
8 } Node;
```

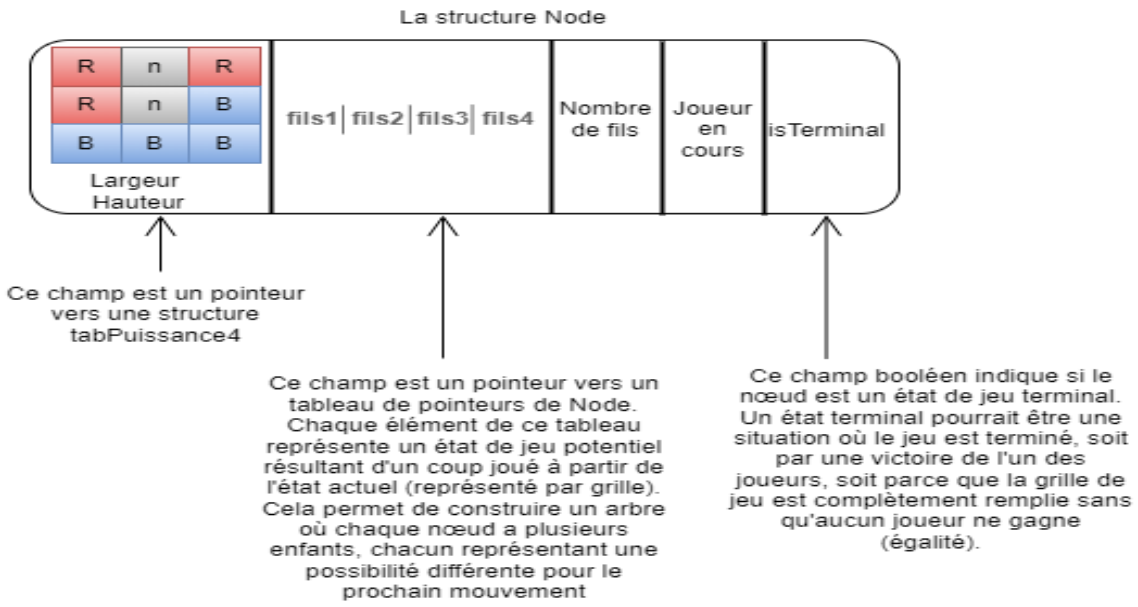


FIGURE 3 – Representation graphique de la structure « Node »

Presentons le diagramme d'activité général de la fonction main :

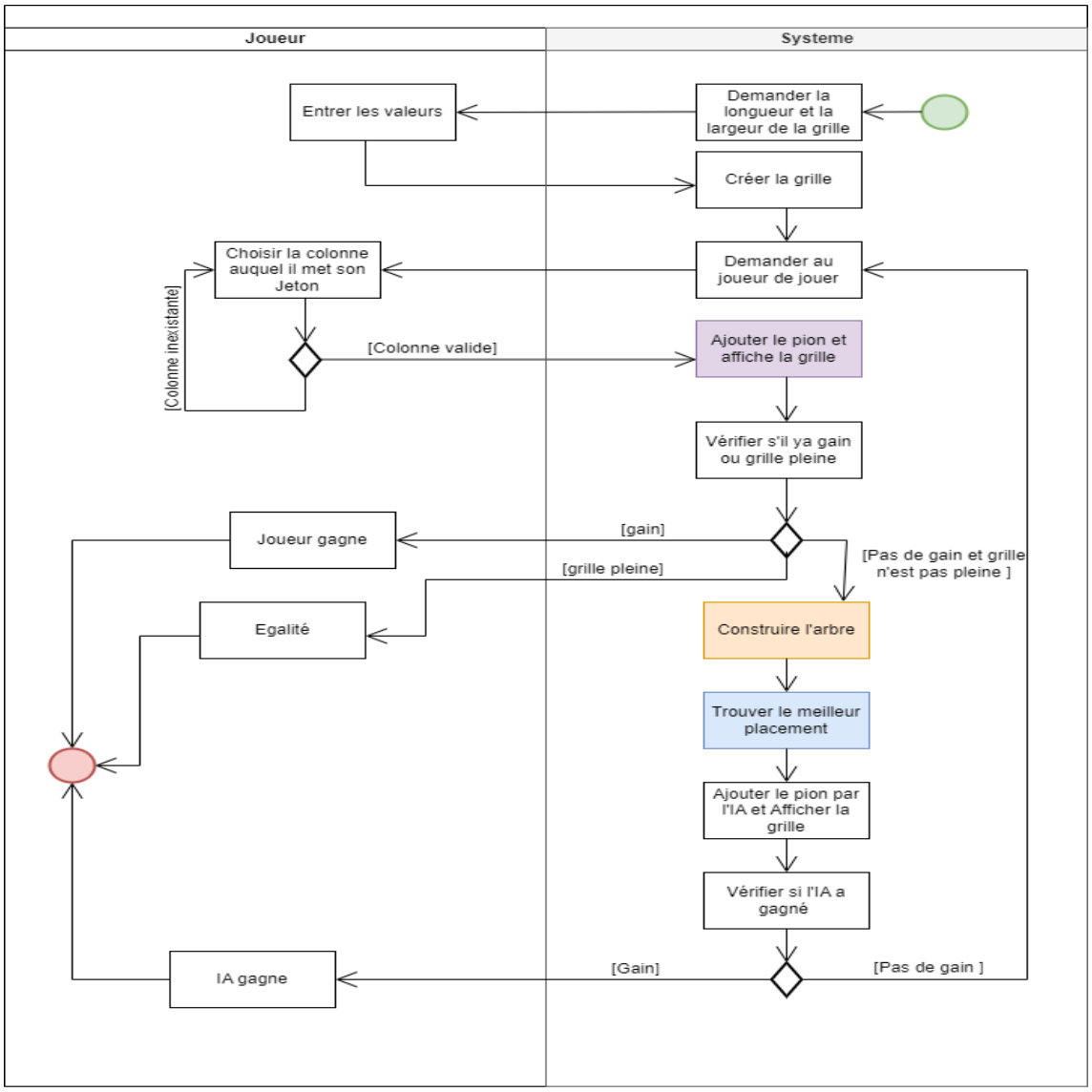


FIGURE 4 – Diagramme d'activité du jeu « Puissance 4 » entre un joueur et une IA

Ce diagramme fait appel à :
La fonction add_dans_grille presenté prècedemment.

La fonction Build-Tree :
Les paramètres de cette fonction sont :

- `node` : un pointeur vers le nœud actuel de l'arbre de jeu.
- `currentPlayer` : le joueur qui doit jouer dans l'état de jeu actuel.
- `cols` : le nombre de colonnes dans le jeu (largeur de la grille).
- `rows` : le nombre de rangées dans le jeu (hauteur de la grille).
- `depth` : la profondeur actuelle dans l'arbre de jeu, utilisée pour limiter la taille de l'arbre.

Cette fonction est récursive, et sa condition d'arrêt est que la profondeur spécifiée (`depth`) est égale à 0, ce qui signifie que la limite de profondeur de l'arbre est atteinte. Dans les jeux avec de nombreuses possibilités comme le Puissance 4, l'arbre de jeu peut croître exponentiellement avec chaque niveau supplémentaire. Une profondeur de 5 peut être choisie pour rester dans des limites de calcul raisonnables, surtout si l'algorithme doit s'exécuter sur du matériel limité en ressources. De plus, pour obtenir 4 jetons successifs de même couleur, une profondeur de 5 est largement suffisante.

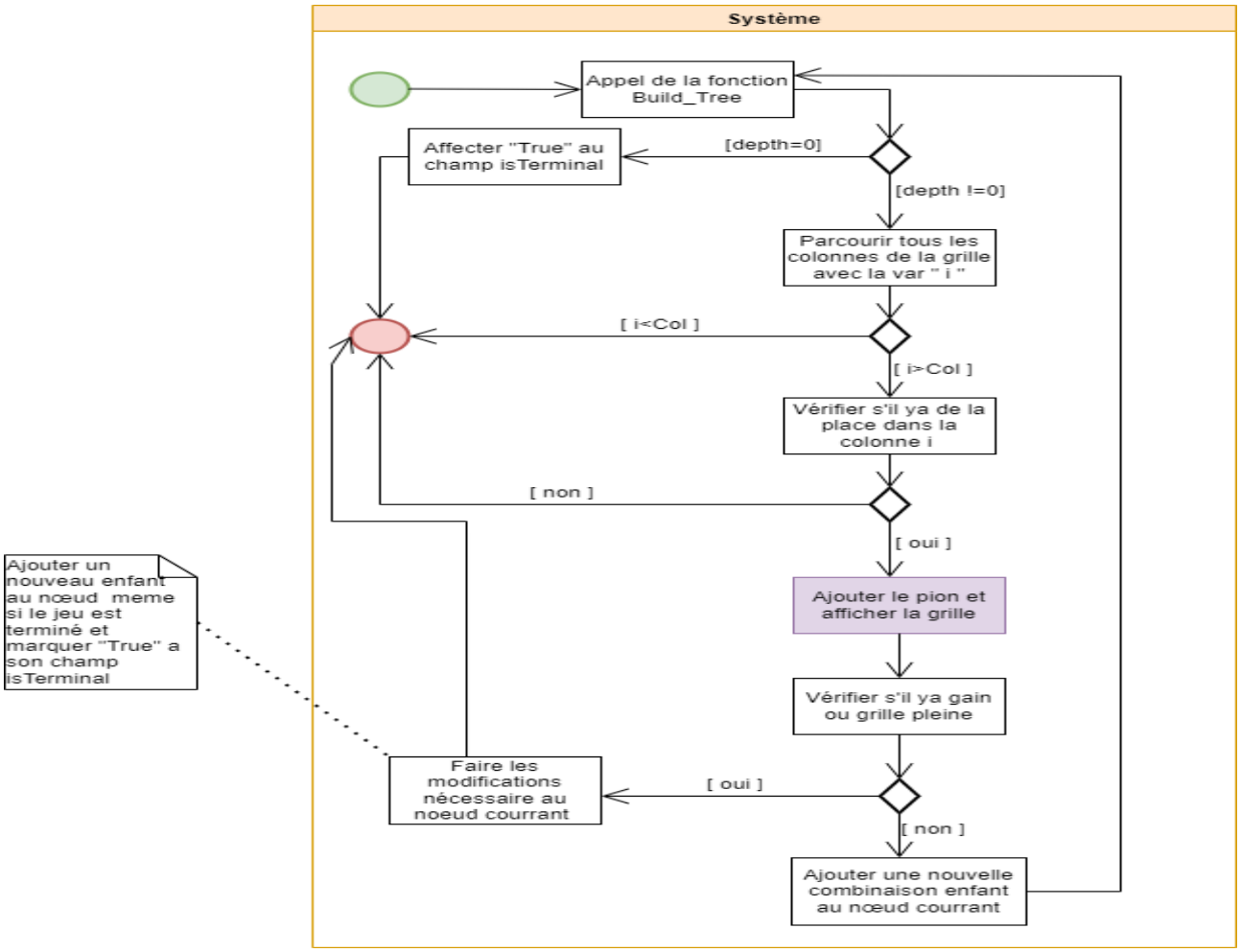


FIGURE 5 – Diagramme d'activité de la fonction « Build_Tree »

Lors de la construction de l'arbre de jeu, il est crucial de simuler ce processus en alternant entre les joueurs à chaque niveau de l'arbre. Cela garantit que l'arbre reflète fidèlement la dynamique du jeu, où chaque coup par un joueur est suivi d'un coup par l'autre joueur ce que fait la ligne 10 du code .

Pour maintenir les états de jeu distincts et non modifiés à chaque niveau de l'arbre de jeu , on se sert avec la fonction **copygrille** .

À la fin du processus exécuté par la fonction buildTree, on obtient un arbre de jeu où chaque nœud a ses propres enfants, chacun représentant un état de jeu possible résultant d'un coup joué à partir de l'état représenté par le nœud parent.

Listing 8 – La fonction `buildTree`

```
1  void buildTree(Node *node, char currentPlayer, int cols, int rows,
       int depth)
2  {
3      // V rifier la condition d'arr t  bas e sur la profondeur
4      if (depth == 0)
5      {
6          node->isTerminal = true;
7          return;
8      }
9
10     char nextPlayer;
11     if (currentPlayer == 'J')
12     {
13         nextPlayer = 'R';
14     }
15     else
16     {
17         nextPlayer = 'J';
18     }
19
20     // Initialiser les enfants
21     node->children = calloc(cols, sizeof(Node *));
22
23     for (int i = 0; i < cols; ++i)
24     {
25         if (canPlay(node->grille, i))
26         {
27             tabPuissance4 *newgrille = copygrille(node->grille);
28             add_dans_grille(newgrille, i, currentPlayer);
29
30             // V rifier si le jeu se termine apr s ce coup
31             if (verif_verticale(newgrille, i) ||
32                 verif_horizontale(newgrille, i) ||
33                 verif_diagonale(newgrille, i) ||
34                 grille_est_pleine(newgrille))
35             {
36                 // Cr er un n ud  m me si le jeu se termine
```

```
37                  node->children[i] = createNode(newgrille, cols,
                        currentPlayer);
38                  // Marquer le n ud comme terminal
39                  node->children[i]->isTerminal = true;
40              }
41              else
42              {
43                  // Cr er un n ud enfant si le jeu ne se termine
                        pas
44                  node->children[i] = createNode(newgrille, cols,
                        nextPlayer);
45                  buildTree(node->children[i], nextPlayer, cols, rows
                        , depth - 1);
46              }
47          }
48          // Si canPlay est faux, la branche n'est pas d velopp e
49      }
50  }
```

Remarque :
La complexité de l'algorithme Minimax est exprimée en fonction de la profondeur de l'arbre de jeu (`depth` = 5) et du facteur de branchement b (nombre de colonnes), qui est le nombre moyen d'enfants par nœud d'où $O(b^{depth})$.

La profondeur de notre arbre est fixée à 5 et le facteur de branchement b est le nombre de colonnes (car chaque colonne peut potentiellement avoir un nœud enfant, c'est-à-dire un coup possible).

La fonction getBestMove :

Cette fonction prend en paramètre :

- `node` : un pointeur vers le nœud actuel de l'arbre de jeu, représentant l'état actuel du jeu.
- `depth` : la profondeur restante de l'arbre à explorer.
- `isMaxPlayer` : le booléen indiquant si le joueur actuel est le joueur maximisant.
- `lignes` et `colonnes` : les dimensions de la grille du jeu.

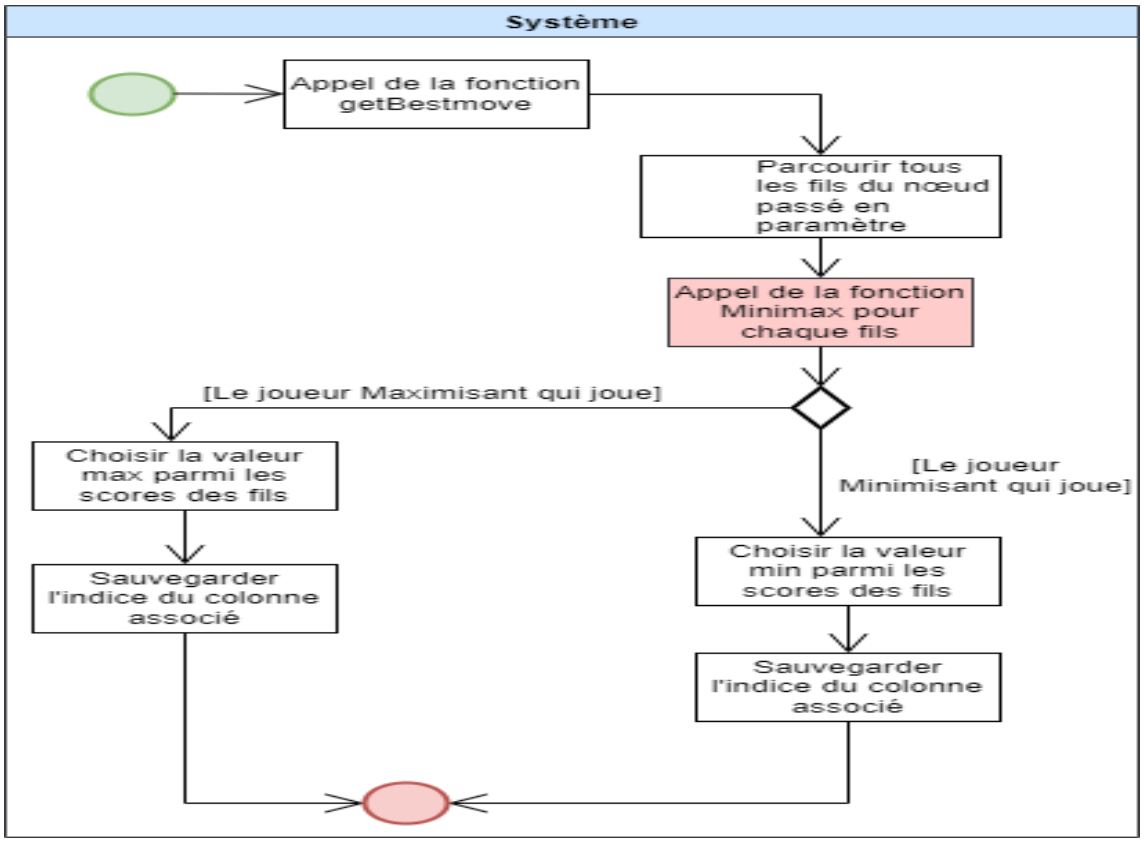


FIGURE 6 – Diagramme d'activité de la fonction « getBestMove »

La combinaison de for et if dans les lignes 7 et 9 : Pour chaque enfant valide (node->children[i] != NULL), la fonction évalue le nœud en appelant récursivement la fonction minimax avec une profondeur réduite (depth - 1) et en alternant le rôle du joueur (de maximisant à minimisant ou vice versa).

INT_MIN et INT_MAX sont des constantes en C définies dans l'en-tête de la bibliothèque **limits.h**. Elles représentent respectivement la valeur minimale et maximale qu'un int (nombre entier) peut avoir dans un programme C qui nous ont servi pour calculer a chaque appel les valeurs Max ou Min des scores des fils .

Listing 9 – La fonction getBestMove

```
1 int getBestMove(Node *node, int depth, bool isMaxPlayer, int lignes
    , int colonnes)
2 {   int bestMove = -1;  // Initialiser avec une valeur non valide
3     if (isMaxPlayer)
4     {
```

```
5           bestEval = INT_MIN;
6       }
7       else
8       {
9           bestEval = INT_MAX;
10      }
11      // Parcourir tous les coups possibles
12      for (int i = 0; i < node->numChildren; ++i)
13      {
14          if (node->children[i] != NULL)
15          {
16              // Calculer la valeur pour le n ud enfant en appelant
                    r cursivement minimax
17              int eval = minimax(node->children[i], depth - 1, !
                    isMaxPlayer, lignes, colonnes);
18              // Pour le joueur maximisant, choisir le coup avec la
                    valeur maximale
19              if (isMaxPlayer && eval > bestEval)
20              {
21                  bestEval = eval;
22                  bestMove = i;  // Mettre    jour le meilleur coup
23              }
24              // Pour le joueur minimisant, choisir le coup avec la
                    valeur minimale
25              else if (!isMaxPlayer && eval < bestEval)
26              {
27                  bestEval = eval;
28                  bestMove = i;  // Mettre    jour le meilleur coup
29              } } }
30      return bestMove;  // Retourner le meilleur coup    jouer}
```

Remarque :
Cette fonction a la meme complexité que Build_Tree

getBestMove appelle a son tour la fonction Minimax :
Comme notre fonction est recursive , sa condition d'arret est si la profondeur est 0 ou si le nœud est un état terminal (node->isTerminal) . A ce moment , l'algorithme évalue la grille du jeu en utilisant la fonction score et retourne cette valeur.
Les scores des nœuds terminaux sont ensuite "remontés" dans l'arbre. À chaque nœud intermédiaire, l'algorithme choisit la meilleure évaluation pour le joueur concerné :
Si c'est le tour du joueur maximisant, Minimax choisit le coup avec la valeur maximale parmi les enfants de ce nœud.
Si c'est le tour du joueur minimisant, Minimax choisit le coup avec la valeur minimale.

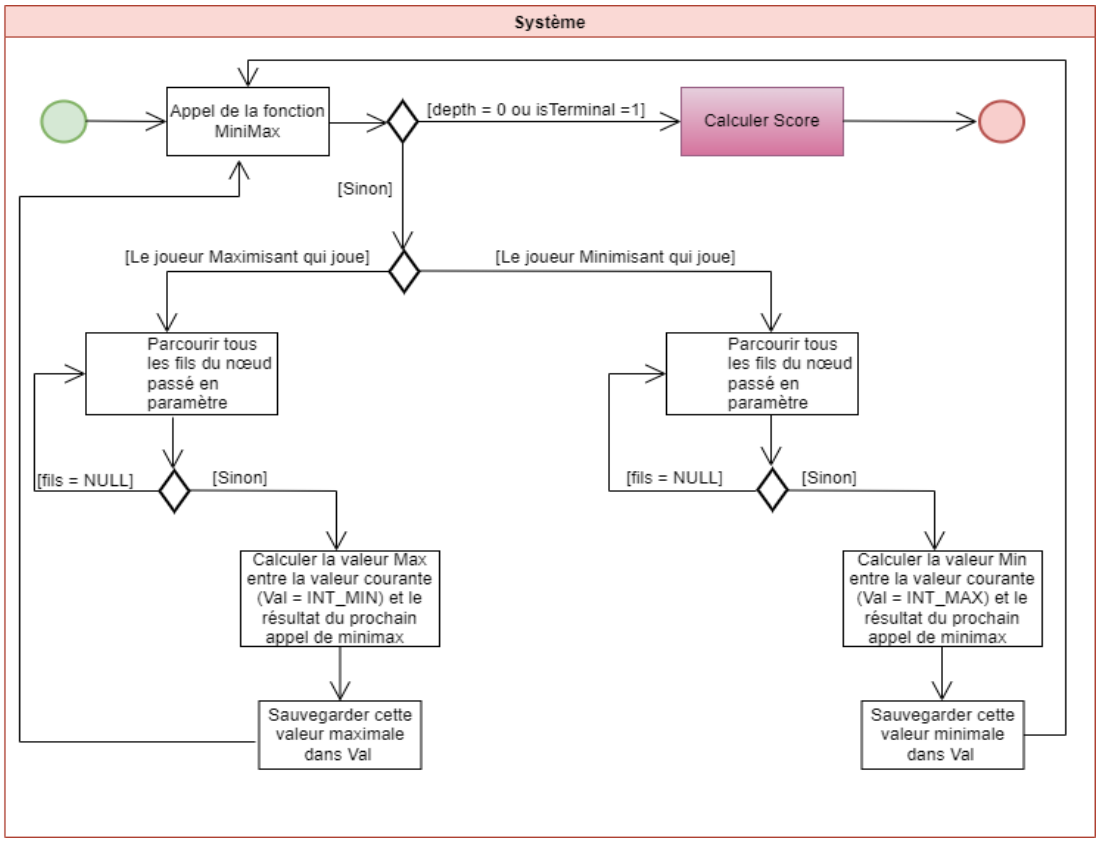


FIGURE 7 – Diagramme d'activité de la fonction « Minimax »

L'objectif de cet algorithme qui utilise un tel arbre, est de prévoir les conséquences des coups non seulement pour le joueur actuel mais aussi pour l'adversaire. Cela aide à évaluer les coups non seulement en fonction de leur impact immédiat mais aussi en fonction de leur impact potentiel sur les tours futurs de l'adversaire. Ceci est clair lorsque la fonction minimax prend False en parametre isMaxPlayer si le joueur Max est en train de jouer et True dans le cas contraire .
Cette alternance aide à anticiper et à planifier en fonction des réactions potentielles de l'adversaire. En comprenant comment l'adversaire pourrait répondre à un certain coup, le joueur peut choisir des stratégies qui sont bénéfiques à long terme.

Listing 10 – La fonction Minimax

```
int minimax(Node *node, int depth, bool isMaxPlayer,int lignes,int
    colonnes)
{
    // V rification si le n ud est terminal ou si la profondeur
       maximale est atteinte
    if (depth == 0 || node->isTerminal)
    {
        return score(node->grille); //  valuer  et retourner le
           score du n ud actuel
    }

    int value;
    if (isMaxPlayer)
    {
        value = INT_MIN;
        // Parcourir les enfants du n ud actuel
        for (int i = 0; i < node->numChildren; ++i)
        {
            if (node->children[i] != NULL)
            {
                // Choisir la valeur maximale
                value = max(value, minimax(node->children[i], depth
                    - 1, false,lignes,colonnes));
            }
        }
    }
    else // Joueur minimisant
    {
        value = INT_MAX;
        // Parcourir les enfants du n ud actuel
        for (int i = 0; i < node->numChildren; ++i)
        {
            if (node->children[i] != NULL)
            {
                // Choisir la valeur minimale
                value = min(value, minimax(node->children[i], depth
                    - 1, true,lignes,colonnes));
            }
        }
    }

    return value; // Retourner la meilleure  valuation  pour le
       n ud actuel
}
```

Remarque :
Cette fonction a la meme complexité que Build_Tree

Aussi Minimax appelle la fonction score :
Ce diagramme explique bien la formule utilisée pour avoir prévoir les differents combinaisons qui peuvent etre jouées

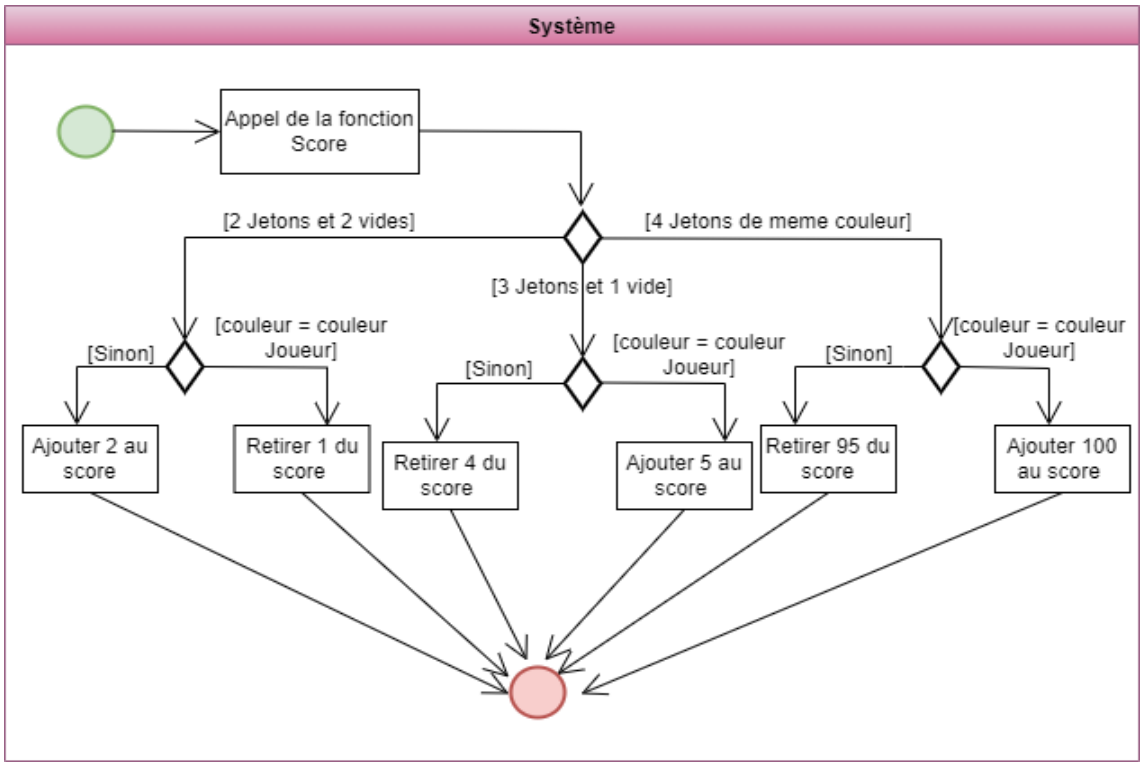


FIGURE 8 – Diagramme d'activité de la fonction « Score »

Listing 11 – La fonction Score

```
1  int score(tabPuissance4 *tp4)
2  {
3      int score = 0;
4
5      // Combinaisons horizontales
6      if (tp4->largeur > 3)
7      {
8          for (uint32_t i = 0; i < tp4->hauteur; i++)
9          {
10             char chaine[5] = {'\0'}; // La cha ne de 4 caract res
                   + le caract re nul
11             for (uint32_t j = 0; j <= tp4->largeur - 4; j++)
12             {
13                 for (uint32_t k = 0; k < 4; k++)
14                 {
15                     chaine[k] = tp4->grille[j + k][i];
16                 }
17                 score += evaluateWindow(chaine); //   valuation
                       apr s la compl tion de la cha ne
18             }
19         }
```

```
20      }
21
22      // Combinaisons verticales
23      if (tp4->hauteur > 3)
24      {
25          for (uint32_t i = 0; i <= tp4->hauteur - 4; i++)
26          {
27              char chaine[5] = {'\0'};
28              for (uint32_t j = 0; j < tp4->largeur; j++)
29              {
30                  for (uint32_t k = 0; k < 4; k++)
31                  {
32                      chaine[k] = tp4->grille[j][i + k];
33                  }
34                  score += evaluateWindow(chaine); //   valuation
                        apr s la compl tion de la cha ne
35              }
36          }
37      }
38
39      // Combinaisons diagonales
40      if (tp4->largeur > 3 && tp4->hauteur > 3)
41      {
42          char chaine1[5] = {'\0'};
43          char chaine2[5] = {'\0'};
44          for (uint32_t i = 0; i <= tp4->hauteur - 4; i++)
45          {
46              for (uint32_t j = 0; j <= tp4->largeur - 4; j++)
47              {
48                  for (uint32_t k = 0; k < 4; k++)
49                  {
50                      chaine1[k] = tp4->grille[j + k][i + k];
51                      chaine2[k] = tp4->grille[j + 3 - k][i + k];
52                  }
53                  score += evaluateWindow(chaine1); //   valuation
                        apr s la compl tion de la cha ne
54                  score += evaluateWindow(chaine2); //   valuation
                        apr s la compl tion de la cha ne
55              }
56          }
57      }
58      return score;
59  }
```

Cette fonction itère à travers différentes portions de cette grille pour évaluer les combinaisons horizontales, verticales, et diagonales.

Combinaisons Horizontales :
La complexité de cette partie est O(height * (width - 3)).

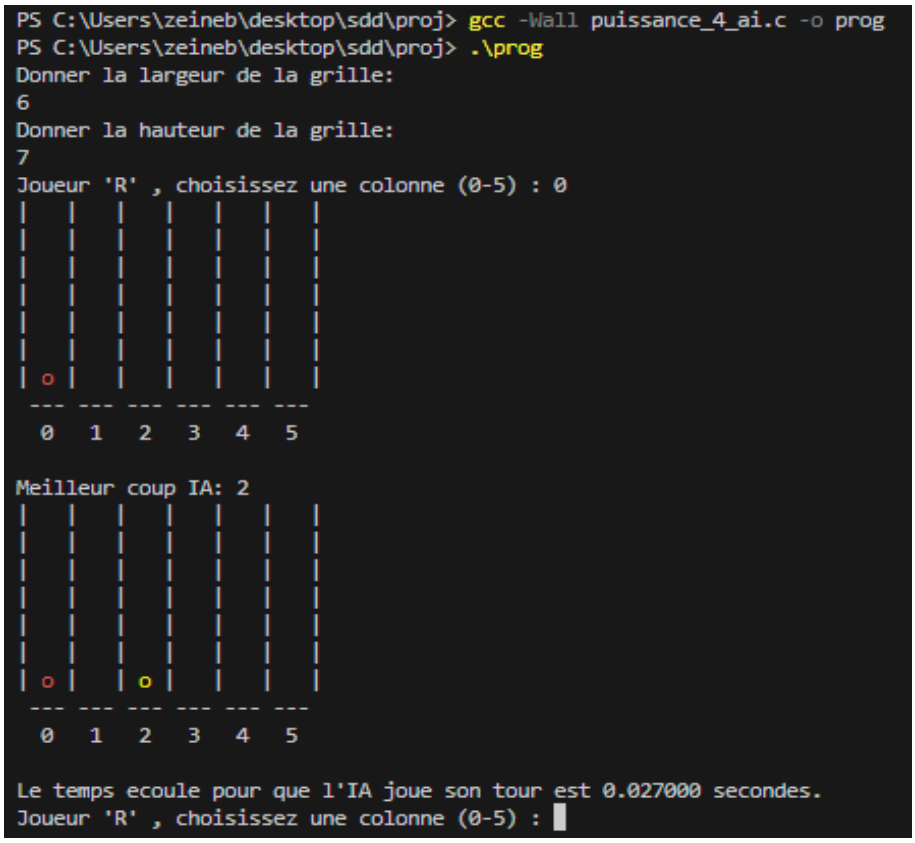
Combinaisons Verticales : La complexité de cette partie est O(width * (height - 3)).

Combinaisons Diagonales :
La complexité de cette partie est O((width - 3) * (height - 3)).

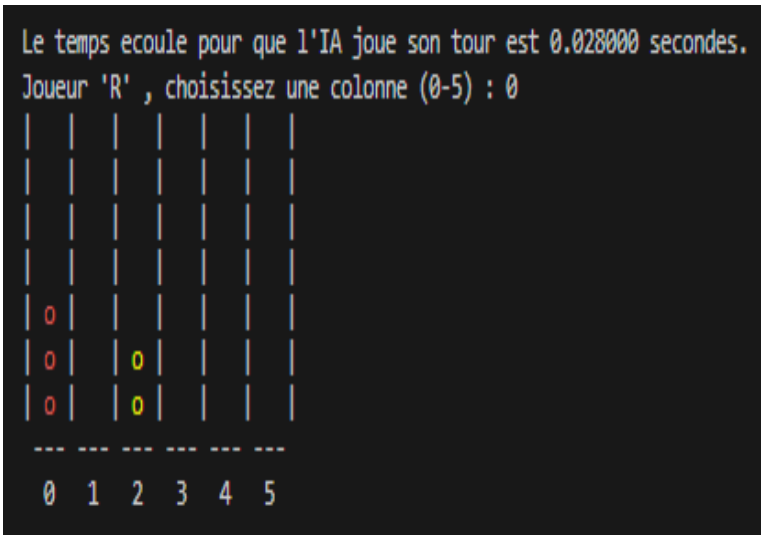
La complexité totale de la fonction score est donc la somme de ces trois parties. Cependant, puisque toutes ces parties sont essentiellement linéaires par rapport aux dimensions de la grille, la complexité totale peut être simplifiée en O(width * height).

3.2 Validation et Test

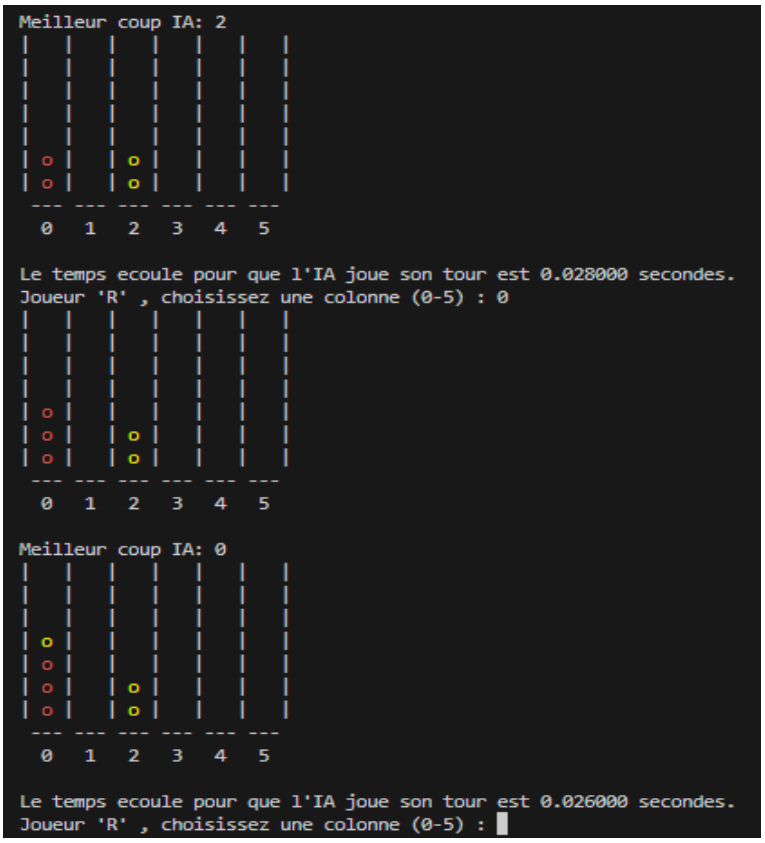
Le programme demande comme d'habitude la valeur du hauteur et largeur de la grille en question :



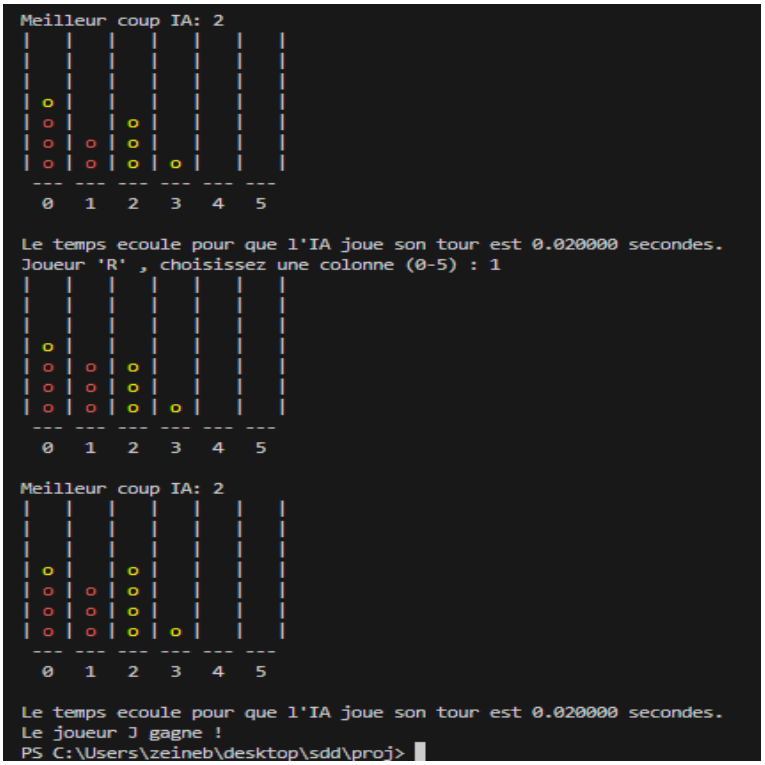
Le joueur manque un jeton Rouge dans la colonne 0 et gagne!!



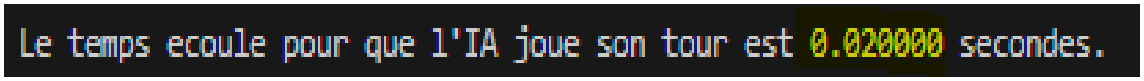
Mais non , notre IA est bien intelligente elle a mis son jeton Jaune a la colonne 0



Finalement , l'IA a gagné :



Remarque : Le temps necessaire pour que l'IA effectue son tour est efficacement réduit



4 Perspectives futures

Premier point : Pour améliorer les fonctions verif_verticale et verif_horizontale, nous pouvons adopter une approche similaire à celle utilisée pour verif_diagonale, en nous focalisant sur une recherche localisée autour du dernier jeton placé, plutôt que de balayer toute la colonne ou la ligne.

Deuxieme point : L'adoption d'une stratégie dans laquelle nous limitons l'algorithme Minimax à une profondeur de recherche de 5 n'est pas parfaite et peut être améliorée, notamment en intégrant l'élagage alpha-bêta. La limitation de la profondeur à 5 est une décision pragmatique, visant à équilibrer entre la précision de la prise de décision et la performance en termes de temps de calcul

L'intégration de l'élagage alpha-bêta peut etre une solution prometteuse à ce cas en :

- ◇ Augementant de la Profondeur de Recherche : En réduisant le nombre de branches à évaluer, l'élagage alpha-bêta permet d'explorer l'arbre de jeu plus en profondeur sans augmenter significativement le temps de calcul. Cela signifie que l'IA peut regarder plus loin dans le futur du jeu, augmentant ainsi la qualité de ses stratégies et prédictions.
- ◇ Optimisant des Performances : L'élagage alpha-bêta élimine les branches de l'arbre de jeu qui ne contribuent pas à la prise de décision, optimisant ainsi les performances en termes de vitesse et d'efficacité. Cela permet à l'IA de réaliser des analyses plus approfondies en un temps plus court. En résumé, l'élagage alpha-bêta représente une optimisation cruciale pour les performances dans le Puissance 4, offrant une meilleure utilisation des ressources tout en maintenant la qualité des décisions.

Troisième point : L'utilisation du machine learning (ML) pour améliorer notre jeu Puissance 4 . En fait , Le but est de permettre à l'intelligence artificielle de ne pas seulement suivre des règles prédéterminées comme le calcul de score et la recherche du min et max, mais d'apprendre et de s'adapter en fonction des expériences passées

5 Conclusion

En conclusion, notre voyage dans l'univers du jeu de Puissance 4 numérisé nous a mené à l'intersection de la programmation stratégique et de l'intelligence artificielle. Nous avons construit, avec des conditions précises, un arbre de décision qui est le pilier de notre algorithme Minimax, permettant à l'IA de prendre des décisions éclairées. Le parcours de cet arbre, réalisé par des fonctions récursives, représente le cœur battant de notre IA, lui conférant la capacité de simuler et d'anticiper une multitude de scénarios de jeu.

Cette exploration a renforcé notre compréhension que l'intelligence artificielle dans les jeux n'est pas seulement une question de programmation, mais aussi une forme d'art qui nécessite une compréhension approfondie de la stratégie humaine.

En définitive, ce projet a été une formidable démonstration de la façon dont la technologie peut être façonnée pour créer non seulement un adversaire virtuel compétent mais aussi un outil qui défie et développe l'ingéniosité humaine.