المدرسة الحسنية للأشغال العمومية
ECOLE HASSANIA DES TRAVAUX PUBLICS

Résolution de problèmes :

Jeu Puissance 4

Réalisation :
ZAHOUR Oumaima
HAOUA Asma
LAHRACHE Reda

Encadrement :
Mme. ADDOU

Sommaire

I. Introduction.....3
1. Contexte du projet.....3
2. Présentation du jeu.....3
3. Outils.....3

II. Modélisation du jeu.....4
1. Structure du jeu.....4
2. Arbre de résolution pour MiniMax.....9
3. Arbre de résolution pour AlphaBeta.....10

III. Implémentation du code.....11
1. Application Console.....11
a. Menu.....11
b. Joueur Vs Joueur.....12
c. Joueur VS Machine.....13
2. L'interface graphique.....17
a. La bibliothèque SDL.....17
b. L'exécution du programme.....18

IV. Conclusion

Introduction :

1. Contexte du Projet Puissance 4 :

Ce projet s'inscrit dans le cadre du module « résolution de problèmes » sous la supervision de Madame ADDOU. L'objectif est de programmer le jeu Puissance 4 (Connect4 en anglais) en langage C/C++, et d'implémenter une intelligence artificielle en utilisant nos connaissances acquises pendant ce module.

2. Présentation du jeu :

Puissance 4 est jeu de stratégie combinatoire abstrait, il se joue entre deux joueurs qui déposent à tour de rôle un disque coloré du haut d'une grille de 6 lignes et 7 colonnes suspendue verticalement. L'objectif est de construire un alignement horizontal, vertical ou diagonal de 4 disques.

3. Outils et bibliothèques employées :

Le travail en groupe était effectué à l'aide de la plateforme Microsoft Teams. Pour la réalisation de ce projet en langage C on a utilisé les outils suivants :

- ✓ Code::Blocks
- ✓ Photoshop
- ✓ Topaz Gigapixel AI
- ✓ ASCII Art
- ✓ La bibliothèque SDL
- ✓ La bibliothèque Windows

Modélisation du jeu :

1. Structure du jeu :

Espace d'état :

- La grille est représentée par la matrice **Plateau[6][7] .**
- Le 1er joueur est représenté dans le plateau par 1 (couleur bleu) et le 2eme joueur par 2 (couleur jaune).
- Le tour du jeu est fourni par un compteur **count** de type **int** (si **count%2==0** alors c'est le tour du 1er joueur sinon c'est le tour du 2ème joueur)

Objectif du jeu :

Chaque joueur essaye de construire un alignement de 4 pions horizontalement, verticalement ou en diagonal. Le premier arrivant à le construire est le gagnant.

Etat initial :

Au début la grille est totalement vide (dans notre code la matrice est totalement remplie par des zéros).

Etat final :

La partie du jeu se termine si l'un des joueurs gagne ou si la grille est totalement remplie (aucune case vide).

Règles du jeu :

Le joueur choisit une colonne parmi les 7 colonne de la grille à condition qu'elle soit non pleine.

La fonction heuristique :

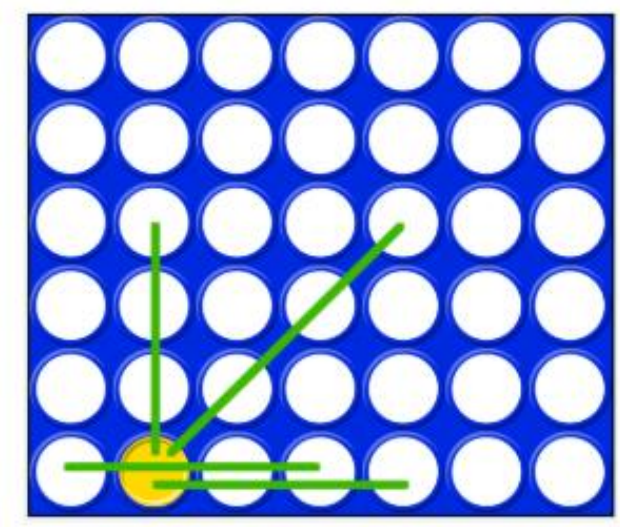
La fonction heuristique est une méthode de résolution de problème qui ne repose pas sur l'examen détaillé du problème. Cela consiste à travailler par approches successives. Par exemple, en faisant ressortir des similitudes avec des difficultés déjà rencontrées, afin de supprimer graduellement les alternatives et ne garder d'un échantillon de solutions.
Dans notre projet la fonction **h** est la différence entre le score du joueur et le score de la machine, donc deux cas se présentent :

- Si **h>0** alors le joueur est avantageux.
- Si **h<0** alors la machine est avantageuse.

Pour calculer cette fonction on a procédé les stratégies suivantes :

i. Le score des cases du plateau :

Ceci consiste à créer une matrice de même dimension que la grille du jeu, où chaque case contient le nombre d'alignements gagnants possibles si l'on place un pion à cet endroit. La figure suivante montre l'idée de cette stratégie :



4

On obtient alors la matrice suivante appelée dans notre code HEURISTIQUE[6][7]

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Pour le calcul du score partiel d'un état donné en se basant sur la matrice HEURISTIQUE on a défini la fonction **scoreHeuristique .**

ii. Le score des alignements :

L'idée de cette stratégie est d'attribuer à chaque alignement de 1,2,3 ou 4 pions un score dépendamment de la machine s'elle est en attaque ou en défense :

- Attaque :

Dans cette branche on calcule le nombre des pions alignés non bloqués par l'adversaire grâce aux fonctions suivantes :

nbrAllignementVideHoriz : retourne le nombre des pions alignés horizontalement et non bloqués .

nbrAllignementVideVertic: retourne le nombre des pions alignés verticalement et non bloqués .

nbrAllignementVideDiagD: retourne le nombre des pions alignés en diagonale droite et non bloqués .

nbrAllignementVideDiagG : retourne le nombre des pions alignés en diagonale gauche et non bloqués .

scoreAllignement : attribue à chaque nombre pions alignés en attaque un score comme le montre le code suivant

```
750   int scoreAlignement(int a)
751   {
752       if(a==0)
753           return 0;
754       if(a==1)
755           return 20;
756       if(a==2)
757           return 200;
758       if(a==3)
759           return 1000;
760       else
761           return 1000000;
762   }
```

- Defense :

Pour la défense on calcule le nombre des pions alignés de l'adversaire bloqués grâce aux fonctions suivantes :

nbrAllignementDefHoriz : retourne le nombre de pions adversaires alignés horizontalement et bloqués .

nbrAllignementDefVertic : retourne le nombre de pions adversaires alignés verticalement et bloqués .

nbrAllignementDefDiagD : retourne le nombre de pions adversaires alignés en diagonale droite et bloqués .

nbrAllignementDefDiagG : retourne le nombre de pions adversaires alignés en diagonale gauche et bloqués .

Remarque : ces fonctions renvoient

- Un nombre négatif si les pions adversaires alignés sont défendus des deux côtés.
- Un nombre positif si les pions adversaires alignés sont défendus d'un seul côté.
- 0 si les pions adversaires alignés ne sont pas défendus.

scoreDefense: attribue à chaque nombre un score comme le montre le code suivant

```
728  int scoreDefense(int a)
729  {
730      if(a==-1)
731          return 10;
732      if(a==-2)
733          return 150;
734      if(a==-3)
735          return 500;
736      if(a==0)
737          return 0;
738      if(a==1)
739          return 1;
740      if(a==2)
741          return 100;
742      if(a==3)
743          return 300;
744  }
```

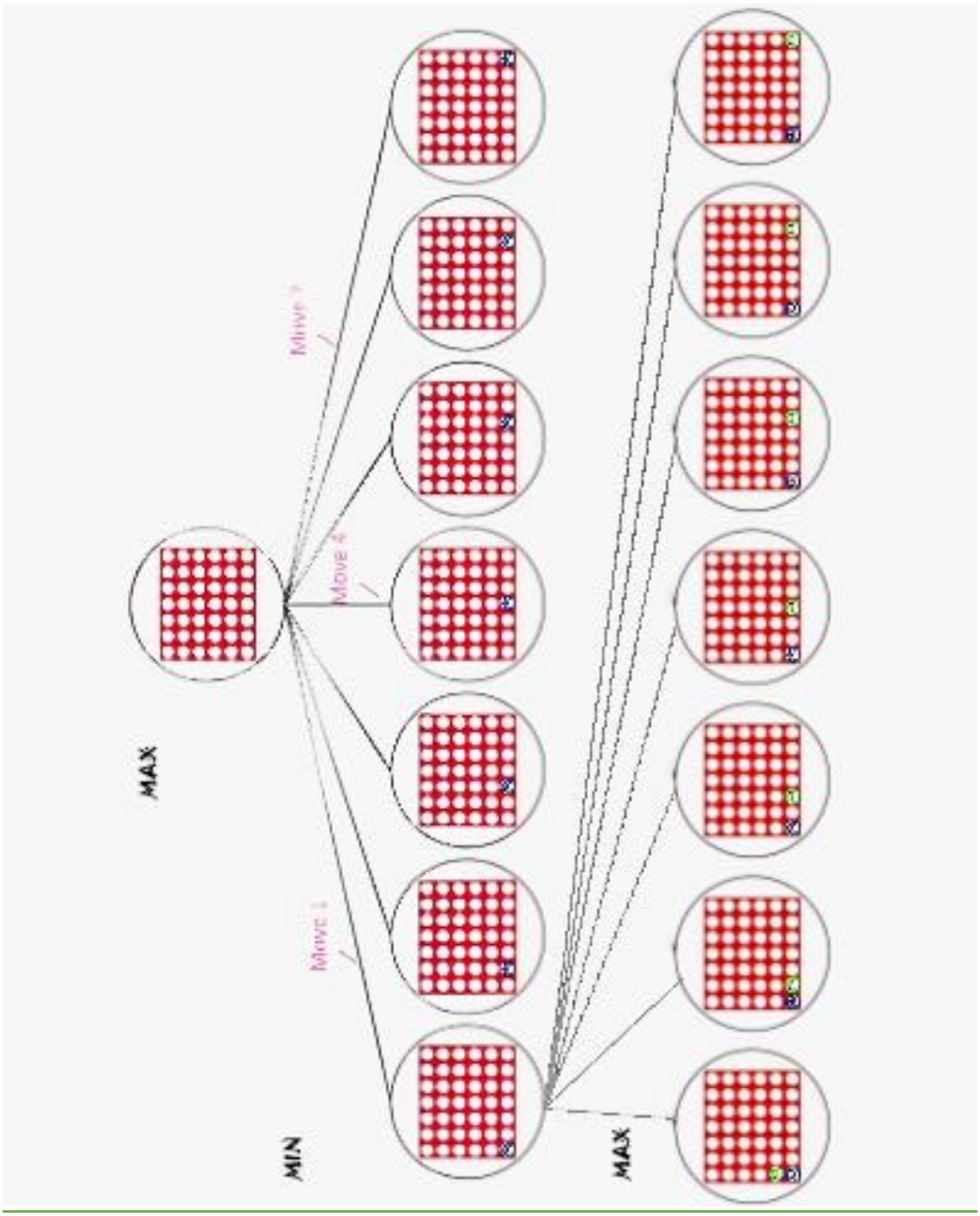
Dans les deux cas (attaque et défense) on a utilisé les fonctions suivantes :

- nbrHorizAllign, nbrVerticAllign, nbrDiagDroiteAllign, nbrDiagGaucheAllign : retournent le nombre de pion alignés (horizontalement/Verticalement/ En diagonale) à partir d'une position donnée.
- horiz4allign, vertic4allign, diagD4allign, diagG4allign : retournent 1 s'il y a un alignement de 4 pions, sinon elles retournent 0.

iii. Score total :

On a rassemblé les deux points précédents (i et ii) dans une seule fonction **score** qui calcule le score total d'un état donné.

2. Arbre de Résolution :



Implémentation du code :

1. Application console :

a. Menu :

Dans ce menu on a appelé la fonction logo dans on a écrit le nom du jeu CONNECT 4 et la fonction menu qui représente les choix *player vs player* , *player vs CPU* et *Exit.*

Les couleurs sont fournies par la fonction Color qui prend en premier argument la couleur de texte et dans le deuxième argument la couleur de fond (un entier de 0 à 15).

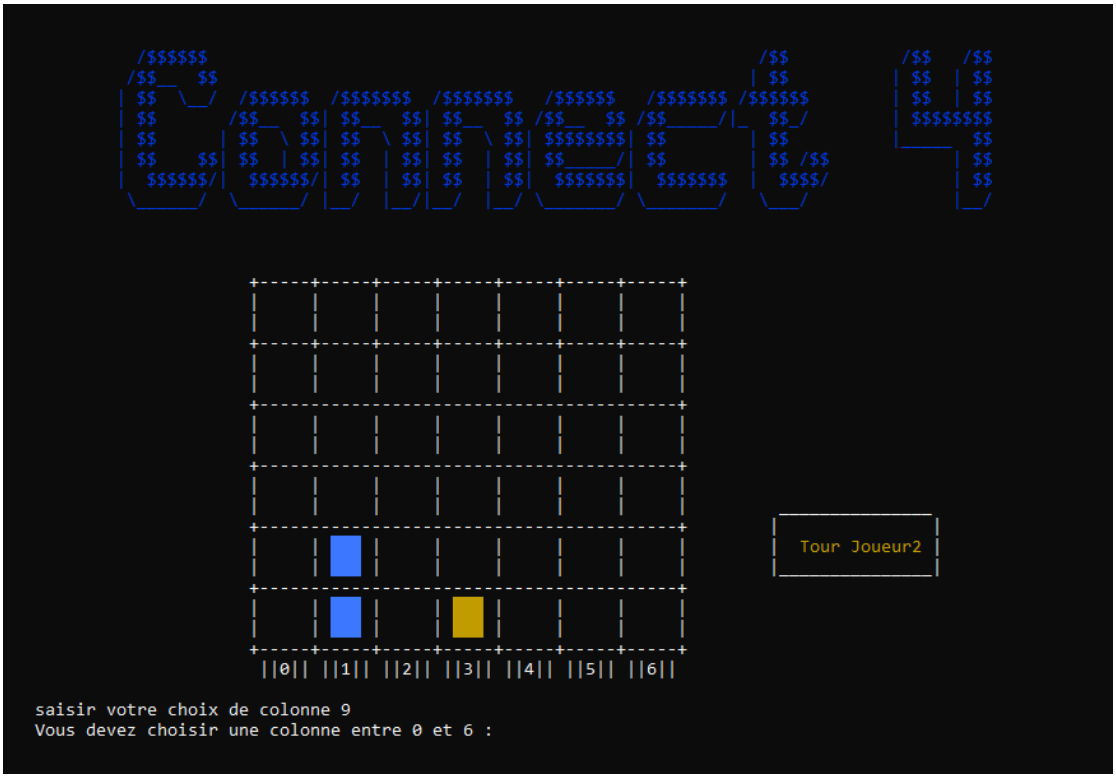
Le menu contient un message qui demande à l'utilisateur de saisir le numéro de son choix.

b. Joueur VS Joueur :

Si l'utilisateur a choisi le premier choix dans le menu c'est-à-dire Player VS Player alors l'interface ci-dessous s'affiche.

Cette interface contient le nom du jeu « Connect 4 » fourni par la fonction logo et la grille 6x7, dont les colonnes sont numérotées de 0 à 6, par la fonction grille. On observe que les pions sont présentés par des carreaux colorés (Bleu pour le premier joueur et Jaune pour le deuxième joueur) grâce à la fonction Afficher. De plus, on voit une zone à droite du plateau qui indique le joueur qui a le tour de jouer grâce à la fonction tour. En bas de l'interface il s'affiche un message qui demande au joueur de choisir le numéro de la colonne, en cas d'un choix qui n'appartient pas à 0 jusqu'à 6, on lui demande d'entrer une valeur comprise entre 0 et 6. Si l'un des joueurs à choisi une colonne pleine alors on lui affiche le message pour choisir une autre colonne non pleine.

Tous ces éléments qui précèdent sont rassemblé dans une seule fonction JoueurVsHumain qui fait appel aux fonctions PartieJoueur pour le 1er joueur et PartieJoueur2 pour le 2ème joueur.



c. Joueur VS CPU :

Menu :

Si l'utilisateur a envie de jouer contre la machine alors il choisit le 2ème choix dans le menu. Si c'est le cas alors l'interface ci-dessous s'affiche dont laquelle on a inclut le nom du jeu et deux choix (Minimax et Alphabeta) afin de permet à l'utilisateur de choisir avec quel algorithme veut jouer à travers un message qui s'e situe en bas de la fenêtre :

Modélisation du jeu pour la machine :

Avant de passer aux algorithmes de l'intelligence artificielle on doit d'abord parler de la conception du jeu pour la machine qui est réalisée grâce à un ensemble de structures et de fonctions rassemblées dans la fonction JouerVsIA.

Pour le stockage des nœuds correspondants à la machine on a opté à l'utilisation d'une structure qui définit une liste chaînée Liste (pile). Cette dernière contient un pointeur de type Noeud .

Chaque nœud est constitué d'une matrice **M[6][7]** est un pointeur **suivant** qui pointe sur l'élément suivant dans la liste comme le montre le code suivant :

```
 9  typedef struct
10  {
11      int M[6][7];
12      struct Noeud* suivant;
13  }Noeud;
14
15  typedef struct
16  {
17      Noeud* sommet;
18      int taille;   // la taille de la liste
19  }Liste;
```

Afin de garantir le bon fonctionnement pour la machine on a défini un ensemble de fonctions :

- viderPlateau : permet de vider la matrice en la remplissant par des zéros
- tailleListe : parcourt une liste commençant du sommet jusqu'à la fin et retourne sa taille
- Afficher : permet le coloriage des pions, s'elle trouve dans la matrice 1 elle colore la case en bleu et s'elle trouve 2 elle colore en jaune.
- PlateauPlein : retourne 1 si le plateau est complètement plein et 0 sinon.
- Hauteur : renvoie le nombre de pions dans une colonne dans une matrice
- Ajouter : ajoute une matrice à une liste de nœud
- Initialiser : initialise une liste de nœuds en la rendant vide.
- Vide : retourne 1 si la liste des nœuds et 0 sinon
- iemeMatrice : parcourt une liste et retourne la $i^{ème}$ matrice dans cette liste.

- genereSuccesseurs : génère 7 successeurs d'un état courant en choisissant à chaque fois l'une des 7 colonnes .
- egalite : recopie une matrice dans une autre matrice
- player : retourne 1 si le joueur qui entrain de jouer et 2 si la machine qui a le tour.
- Inverser : inverse un tableau d'entiers et de taille 7 dont le quel on stocke le score de chaque successeur.
- MAX : renvoie le maximum de deux entiers
- MIN : renvoie le minimum de deux entiers
- iMax : renvoie l'indice de maximum dans un tableau d'entiers
- iMin : renvoie l'indice de minimum dans un tableau d'entiers
- Max : renvoie le maximum d'un tableau d'entiers, cette fonction est utilisée dans l'algorithme minimax pour choisir le max pour la fonction heuristique si c'est le tour du joueur.
- Min : renvoie le minimum d'un tableau d'entiers, cette fonction est aussi utilisée dans l'algorithme minimax pour choisir le min pour la fonction heuristique si c'est le tour de la machine.
- partieFinie : cette fonction retourne 1 si l'un des joueurs a construit un alignement de 4 ou si le plateau est totalement plein.
- Utilité : retourne 1 si le joueur est le gagnent, -1 si la machine est la gagnante et 0 si aucun des joueurs n'a pas pu gagner.
- Adversaire : renvoie 2 si player retourne 1 et renvoie 1 sinon

L'algorithme MiniMax :

L'idée de l'algorithme est de développer complètement l'arbre de jeu, de noter chaque feuille avec sa valeur, puis de faire remonter ces valeurs avec l'hypothèse que chaque joueur choisit le meilleur coup pour lui. Cela signifie en pratique que :

- Le joueur choisit le coup amenant à l'état de plus grande valeur
- La machine choisit le coup amenant à l'état de plus petite valeur

```
410  int minimax(int T[6][7], int compteur,int p)
411  {
412      int a,i;
413      int D[7];
414      Liste l;
415      if(partieFinie(1,T) || partieFinie(2,T) || p==0 )
416      {
417          return h(T);
418      }
419      else
420      {
421          if(compteur%2==1)
422          {
423              l=genereSuccesseurs(T,compteur);
424              p--;
425              compteur++;
426              for(i=0;i<tailleListe(l);i++)
427              {
428                  nbr++;
429                  D[i]=minimax(iemeMatrice(i+1,l),compteur,p);
430              }
431              return Min(D);
432          }
433          else
434          {
435              l=genereSuccesseurs(T,compteur);
436              p--;
437              compteur++;
438              for(i=0;i<tailleListe(l);i++)
439              {
440                  nbr++;
441                  D[i]=minimax(iemeMatrice(i+1,l),compteur,p);
442              }
443              return Max(D);
444
445          }
446      }
447  }
```

En pratique, l'algorithme **MiniMax** n'est pas conseillé car il est rarement possible de développer l'ensemble de l'arbre de jeu :

- Limiter la profondeur d'exploration.
- Traiter des nœuds inutiles dans l'arbre.

Ces deux limites sont améliorées dans l'algorithme **AlphaBeta**

L'algorithme AlphaBeta :

L'idée de cet algorithme est d'élaguer certaines branches de l'arbre sans changer le principe de la fonction heuristique

```
470  int alphabeta(int a,int b,int compteur,int p,int T[6][7])
471  {
472      int i,v;
473      Liste l;
474      if(partieFinie(1,T) || partieFinie(2,T) || p==0)
475      {
476          return h(T);
477      }
478      else
479      {   if(compteur%2==0)    //turn MAX
480          {
481              v=-1000000;
482              l=genereSuccesseurs(T,compteur);
483              p--;
484              compteur++;
485              for(i=0;i<tailleListe(l);i++)
486              {
487                  nbr++;
488                  v=MAX(v,alphabeta(a,b,compteur,p,iemeMatrice(i+1,l)));
489                  if(v>b) return v;
490                  else a=MAX(a,v);
491              }
492          }
493          else   // turn MIN
494          {
495              v=1000000;
496              l=genereSuccesseurs(T,compteur);
497              p--;
498              compteur++;
499              for(i=0;i<tailleListe(l);i++)
500              {
501                  nbr++;
502                  v=MIN(v,alphabeta(a,b,compteur,p,iemeMatrice(i+1,l)));
503                  if(a>v) return v;
504                  else b=MIN(b,v);
505              }
506          }
507          return v;
508      }
509  }
```

2. Interface graphique :

a. La bibliothèque SDL :

- Installation de la bibliothèque **SDL** pour créer l'interface graphique du jeu.
- Installation de la bibliothèque **SDL_image** pour pouvoir traiter les images de différents formats (**png, jpg, jpeg** ..) et non seulement le format **bmp**

Les instructions essentielles pour réaliser ce travail sont :

- SDL_Init : chargement de la SDL dont l'argument est **SDL_INIT_VIDEO**(c'est la partie que nous chargerons le plus souvent) qui charge le système d'affichage (vidéo).
- SDL_Quit : arrêt de la SDL.
- SDL_SetVideoMode : Choix du mode de la vidéo, elle prend 4 arguments suivants :
  - La largeur de la fenêtre désirées (en pixels)
  - La hauteur de la fenêtre désirées (en pixels)
  - Le nombre de couleur affichables (en bits/ pixels)
  - Des options (des flags) (SDL_HWSURFACE, SDL_SWSURFACE, SDL_RESIZABLE, SDL_NOFRAME, SDL_FULLSCREEN, SDL_DOUBLEBUF)

- SDL_WM_SetCaption: changer le titre de la fenêtre en écrivant dans son premier argument le nouveau titre.
- SDL_Surface : on crée un pointeur sur la surface afin de la mémoriser.

- SDL_FillRect : colorer une surface. Elle prend 3 paramètres dans l'ordre :
  - Un pointeur sur la surface dans laquelle on doit dessiner
  - La partie de la surface qui doit être remplie. Pour le remplissage total de la surface on met NULL
  - La couleur à utiliser pour remplir la surface

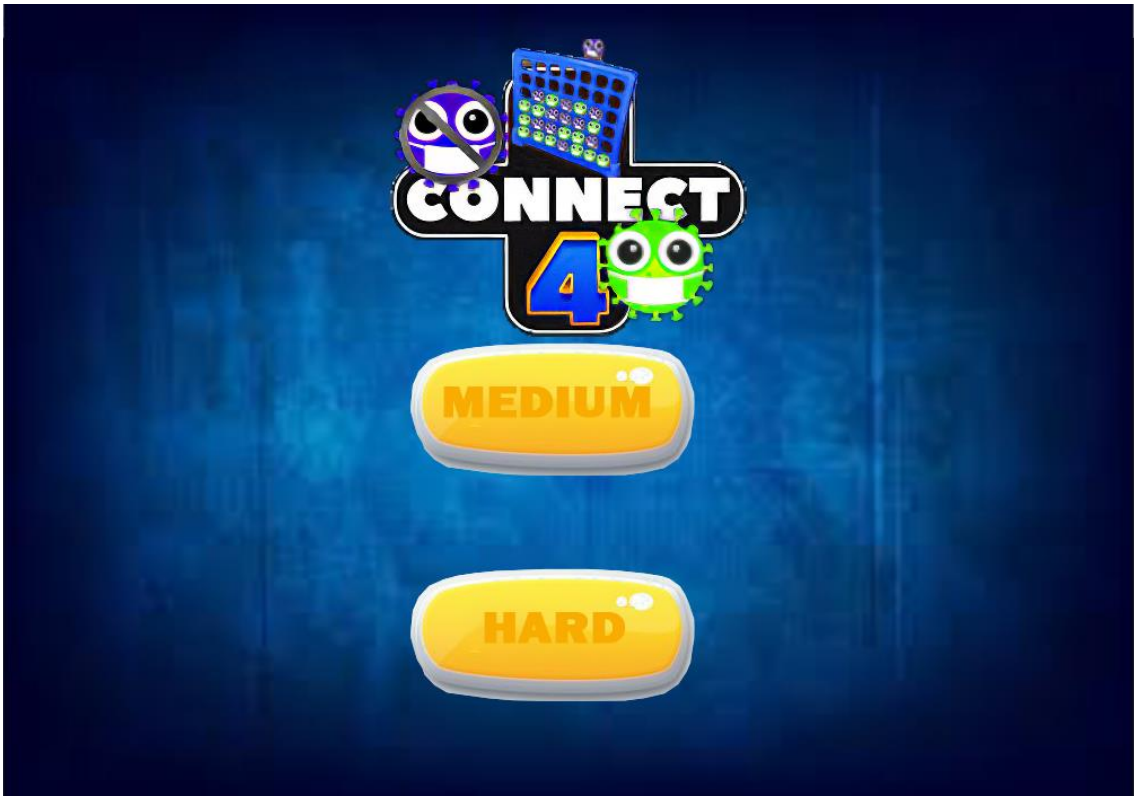
- SDL_Flip : Mise à jour de la fenêtre. Elle prend en argument le pointeur sur la surface
- SDL_FreeSurface : libérer la surface allouée dans la mémoire
- SDL_Rect : définir les coordonnées de la surface
- SDL_BlitSurface : coller la surface à l'écran. Elle prend en arguments :
  - La surface à coller
  - Une information sur la partie de la surface à coller (facultative). Ça ne nous intéresse pas car on veut coller toute la surface, donc on met NULL.
  - La surface sur laquelle on doit coller.
  - Un pointeur sur une variable contenant les coordonnées où devra être collée la surface.

- IMG_Load : charger les images. Elle prend en argument le nom du fichier à ouvrir.

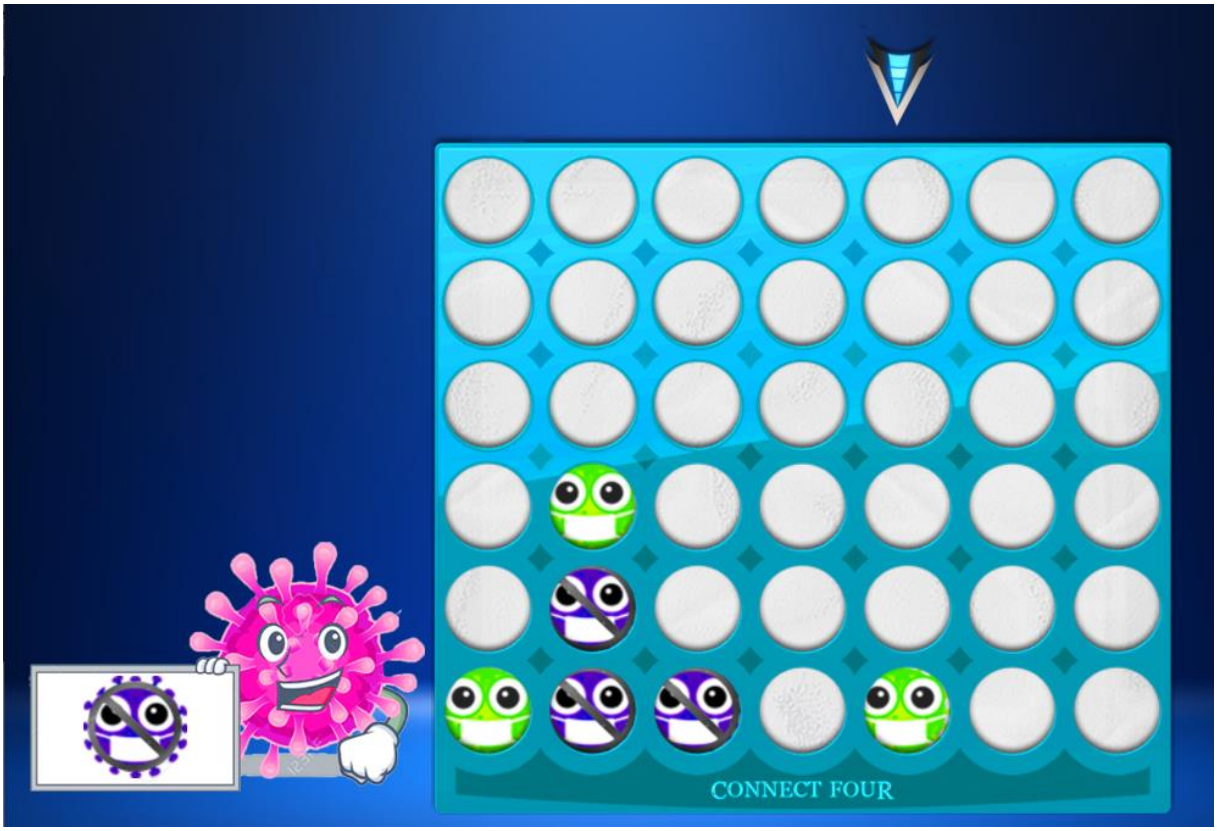
b. L'exécution du programme :

Lors de l'exécution, l'interface suivante s'affiche et contient les noms des membres du groupe, le logo du jeu et deux boutons, le 1er est EXIT pour quitter le jeu est le 2ème PLAY pour jouer.

Si on clique sur PLAY, l'interface suivant s'affiche pour choisir quel algorithme on veut jouer avec (MEDIUM : minimax avec p=4 et HARD : AlphaBeta avec p=5)

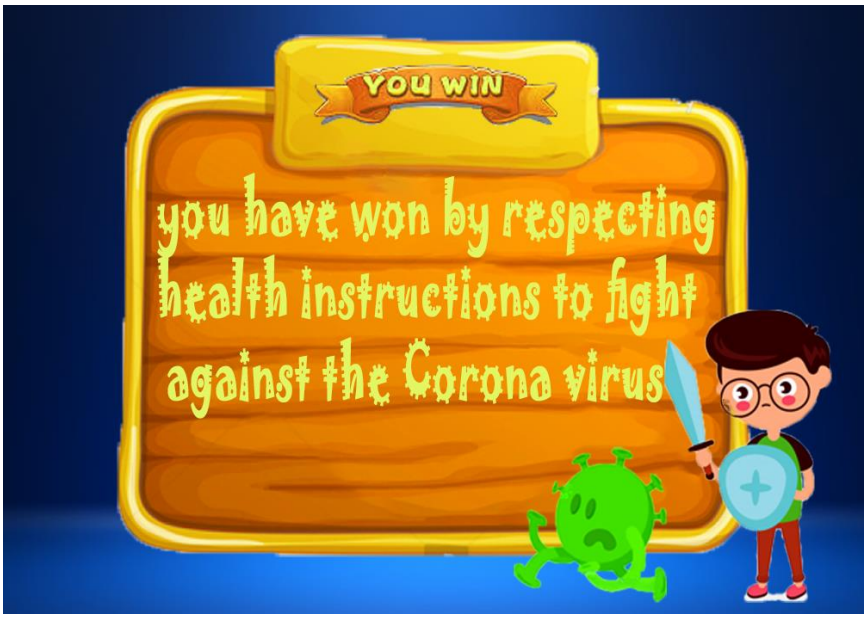


Après le choix de de l'algorithme, l'interface ci-dessous s'affiche. On constate qu'elle contient le plateau, un curseur qui suit la souris lors du choix de la colonne et une zone qui indique quel joueur qui a le tour.



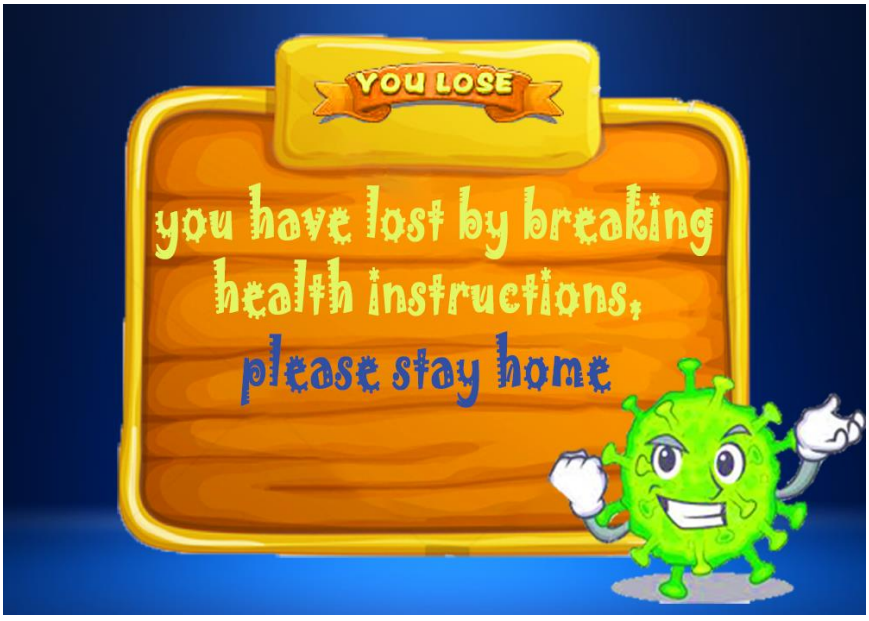
Quand le jeu se termine, les 3 cas suivants se présentent :

- Le joueur gagne :



- EX-AEQUO :

- La machine gagne :



Conclusion :

Ce projet a été effectuer dans le cadre du module Résolution de Problèmes sous l'encadrement de Mme. ADDOU que nous remercions de nous avoir donner cette opportunité de travailler sur un projet réel basé sur l'intelligence artificielle, à savoir la programmation du Jeu Puissance 4.

Grace à ce projet on a pu tester nos connaissances en résolution des problèmes acquises dans le 2ème semestre. On a également pu améliorer nos compétences en modélisation et développement des différents algorithmes de recherche afin de pouvoir construire une puissante intelligence artificielle.