BOURAGBA SOFIANE
GROUPE TD C 14/01/2017

Rapport : TIC TAC TOE

Explication du morpion de base :

Pour ce projet, le morpion est en réalité une matrice 3x3, les lignes sont numérotées de 1 à 3 tandis que les colonnes le sont de A à C.

Premièrement on demande qui veut joueur en premier (l'ordi a pour code joueur 2 et l'utilisateur 1). Pour changer de joueur à chaque tour j'utilise la fonction modulo, en effet cette fonction (modulo 2) renvoie un nombre entier entre 0 et 1, de ce fait en ajoutant 1 j'obtiens un nombre entre 1 et 2 qui correspond au joueur actuel.

I. Les fonctions

La première fonction nécessaire pour la réalisation du morpion est bien la création de la matrice. Pour cela, une fonction **int** creer_morpion()** nous retourne un double pointeur, donc une matrice qui correspondra au plateau du jeu. La fonction malloc nous permet d'avoir un tableau de taille 3, et en appelant la même fonction malloc a chaque indice du tableau on se retouve donc avec une matrice 3x3.

Une fois notre morpion crée, il faut l'initialiser, d'où la fonction **void initiliser(int** morpion)** .Cette fonction est simple, elle parcours la matrice et y insère un 0 à chaque indice. Pourquoi ? Car comme je l'ai dit plus haut, le 1 associé à un pion de l'utilisateur, le 2 de l'ordi et le 0 correspond à une case vide.

Par la suite, j'ai crée une fonction qui affiche le morpion **void afficher(int** morpion)** cette fonction elle aussi étant simple, se contente d'afficher le morpion ainsi que ces axes, pour choisir une case (1 à 3 pour la verticale, a à c pour l'horizontale). Cependant, notre matrice est du type int, or on veut afficher des X, des O ou du vide qui correspond à l'état du jeu. De ce fait grâce à des conditions sur le contenu de la matrice je peux afficher les jetons des joueurs.

Pour pouvoir jouer, il faut bien évidemment demander une case à l'utilisateur, d'où les fonction **int demander_colonne()** et **int demander_ligne()** . Ces fonctions demandent a l'utilisateur d'entrer une ligne et une colonne tant qu'elles ne correspondent pas aux coordonnées possibles. Elles retournent donc la ligne et la colonne de la matrice.

Pour pouvoir gagner, différentes fonctions sont écrites : **int gain_vertical(int** morpion) int gain_horizontal(int** morpion) int gain_diagonal(int** morpion) int fin_jeu(int** morpion)**. Ces fonctions déterminent si oui ou non un joueur à gagner. Ainsi elles parcourent le morpion en vertical, horizontal et diagonal et renvoient 1 si un joueur à gagner 0 sinon.

Dans ce jeu, il n'y a pas forcément de vainqueur, le jeu peut se finir par un match nul, si le morpion est rempli, d'où la fonction **int plateau_full(int** morpion)**. Cette fonction renvoie 1 si il n'y a plus de case vides 0 sinon.

II. L'IA et l'arbre n-aire

Pour l'abre n-aire j'ai choisi la structure suivante :

```
typedef struct _arbre_IA
{
   int poids;
   int ligne;
   int colonne;
   int joueur;
   struct _arbre_IA* fils;
   struct _arbre_IA* frere;
   struct _arbre_IA* fils_au_pere;
   struct _arbre_IA* frere_au_pere;
}arbre_IA;
```

Le poids correspond à la valeur du jeu en cours, 0 si match null, -1 si l'utilisateur gagne, 1 si l'ordi gagne. La ligne et la colonne correspondent à la position à laquelle l'ordi doit joueur pour : Soit contrer l'utilisateur, soit gagner contre lui. Le joueur indique à qui appartient le pion. Mon arbre possède des fils et des frères. Cependant seul la racine ne possède pas de frère. En effet, la racine correspond à l'état initial du jeu. Les fils correspondent aux différents emplacements possibles pour les joueurs, mais de manières alternées. Les frères eux, correspondent aux différents emplacements pour un même et unique joueur à partir d'un état donné.

J'utilise la fonction **arbre_IA * initialisation_arbre()** pour initialiser mon arbre, une variable doit toujours être initialisée, j'ai choisi d'initialiser colonne et ligne à 3 car cela est impossible (emplacement de 0 à 2). Le poids initialisé à 0, car au pire c'est un match nul, au mieux c'est soit l'ordi, soit l'utilisateur qui gagne. Le reste est initialisé à NULL.

Plusieurs fonctions ont été nécessaires pour créer l'arbre, **int nombre_de_frere(arbre_IA* racine)** qui détermine le nombre de frère à partir d'un nœud donnée. **int nombre_case_vide(int** morpion)** Qui détermine le nombre de case vide d'un état du jeu et enfin **int** copier_matrice(int** morpion)** Qui fait une copie de la matrice donnée en paramètre.

A. Création de l'arbre

J'ai choisi de faire ce programme de manière itérative pour mieux voir ce qu'il se passe.

Dans un premier temps, je calcule le nombre de case vide, ce nombre est en fait le nombre de fils qu'aura la racine. Puis pour créer les frères d'un nœud je fais une copie du morpion actuelle.

Je m'occupe des fils de la racine, donc tant qu'il reste des cases vides, j'insère un jeton du joueur puis je l'associe au fils du nœud actuelle, puis je change de joueur, j'insère dans le morpion puis je l'associe au fils du nœud et ainsi de suite. Donc j'ai tous mes fils de la racine. Sans oublier de faire une copie de chaque morpion à chaque nœud de fils, cela me permettra de faire les frères à partir des copies.

Maintenant que les fils sont faits, je m'occupe des frères de chaque nœud des fils de la racine

J'utilise le même principe que pour les frères, je fais une copie du morpion actuelle et j'y insère un jeton d'un joueur, sauf que je garde le même joueur, puis j'enlève le jeton que je viens d'insérer et je l'insère autre part tant qu'il reste une place. Les frères sont maintenant fait. Il ne reste qu'à faire les fils des frères de la même manière que la racine et ainsi de suite.

Avant d'avoir fini mon arbre, s'il reste 0 place disponible, cela signifie que le morpion est alors une feuille de mon arbre, et lorsque c'est une feuille, j'évalue mon morpion, je lui associe donc son poids (0 si match nul, 1 si l'ordi gagne, -1 si l'utilisateur gagne).

Il ne me reste alors qu'à remonter l'information approprié à la racine. Ainsi, si dans une feuille, le poids est -1 alors, on remonte l'information au fils de la racine de la même profondeur l'information, cependant si pour un fils de la racine qui se situe plus près de celle-ci, une feuille à un poids de 1 alors on va privilégier le poids de 1. En effet, on préfèrera que l'ordi gagne plutôt que l'ordi contre le joueur. Cependant si pour toutes les feuilles, le poids est de 0, alors l'information remonté sera la plus éloigné de la racine. Vous remarquerez qu'un booléen continuer est créé dans la fonction. Pourquoi ? Simplement car si une feuille à un poids de 1 à une certaine profondeur de l'arbre, alors on n'ira pas voir ce qui se passe aux profondeurs supérieures car on sait que l'ordi à gagner.

Ainsi à la fin, la racine possède la ligne et la colonne à jouer, pour gagner ou contrer le joueur.