Adrien Guittard

Rapport de projet : Puissance 4

Note : dans le fichier jeu.h j'ai pas mal détaillé le fonctionnement de chaque fonction.

Pour être efficace, j'ai procédé par étapes :

-Tout d'abord implémenter le jeu de base avec une grille 7x7

-Ensuite rajouter les modifications du sujet

-S'occuper de la présentation et corriger d'éventuels bug

-Implémenter un système de sauvegarde de la partie en cours

-Et le plus compliqué pour la fin, implémenter un joueur intelligent

(je n'ai pas eu le temps de finir d'implémenter le joueur intelligent, j'ai donc simplement laissé le joueur aléatoire)

Première étape : Le jeu de base, (dans cette partie on parle seulement du jeu sans les règles ajoutées).

On commence par créer une structure "board" qui contiendra toute les informations relative a la partie en cours, elle contiendra 4 champs : une matrice 7x7, "board" qui représente la grille de jeu; un tableau longueur 7, 'height', qui gardera en mémoire la hauteur de chaque colonne; un entier "player", qui permettra de savoir à tout moment quel joueur est en train de jouer et un autre entier "turn", qui comptera le nombre de tours passés.

J'ai commencé par créer une variable de type "board" nommée "game" qui représente la partie. J'ai ensuite implémenté une fonction qui affiche la matrice représentant la partie en cours pour pouvoir tester les prochaines fonctions.

Vient ensuite 3 fonctions qui marchent ensemble : la première "play", qui prend en paramètre une colonne, va modifier la matrice après chaque tour, elle se sert des 2 suivantes "win" et "isPossible" pour vérifier avant chaque tour d'abord si on peut jouer sur la colonne mise en paramètre et ensuite vérifier si le fait de jouer sur cette colonne fait remporter la partie au joueur, alors la partie s'arrête et on affiche la matrice ainsi qu'un message de fin, sinon la fonction play continue jusqu'à une fin : soit un joueur gagne soit quand le nombre de tour (game.turn) est égale à 49 c'est une égalité.

La fonction "win" permet donc de savoir si le fait de jouer sur la colonne mise en paramètre fait gagner la partie au joueur. Elle est divisée en 4 sous fonctions qui vérifient chacune si il y'a un alignement horizontal, vertical, ou sur l'une des 2 diagonales (les 4 fonctions "check…") . Chaque sous-fonction fonctionne sensiblement de la même manière : on se positionne sur la case à partir de laquelle on souhaite vérifier s'il y a un alignement. On commence par regarder dans une direction et on s'arrête quand on rencontre un jeton de l'autre joueur ou une case sans jeton et on repart dans l'autre sens a partir de la case juste après la 1ère. S'il y a 4 ou plus que 4 jetons alignés, la partie est remportée par ce joueur. Si dans aucune des 4 fonctions ne renvoie un gagnant, la fonction "win" renvoie faux. Le fait de regarder seulement autour de la case jouée plutôt que de vérifier s'il y a un alignement dans toute la matrice à la fin de chaque tour permet de diminuer grandement le nombre de calculs par tour. Mais pour les autres fonctions, qui font tourner la grille, on est obligé de regarder dans toute la matrice s'il y a un alignement. D'où les premières lignes de chaque fonction "check…" qui sont ré-utilisées dans une autre fonction (expliquée plus bas).

La fonction "isPossible" vérifie simplement si le chiffre passé en paramètre est bien entre compris entre 0 et 6 et si la colonne est pleine ou non. Pour cela on utilise le tableau "height" de la structure initialisé au début. Par exemple, pour vérifier si la 5ème colonne est pleine ou non, il suffit de vérifier si height[5] > 6.

Dans la fonction “play”, comme énoncé plus haut, on vérifie d’abord si la case est jouable (“isPossible”), ensuite si jouer sur cette case va faire gagner le joueur en train de jouer (“win”), si ce n'est pas le cas on peut jouer simplement en modifiant dans la matrice la case, correspondant à la colonne passé en paramètre appelée ‘x’ et la ligne donnée par la case x du tableau “height”, par le numéro du joueur en train de jouer. On incrémente ensuite la case du tableau pour les prochains tours et on modifie la variable joueur. Cette variable vaut 1 ou 2 et change entre chaque tour. Sans oublier d’afficher la matrice pour que l’utilisateur puisse voir les modifications apportées.

Nous avons à présent un puissance 4 qui fonctionne !

Deuxième étape : Les modifications du sujet.

Le sujet impose d’ajouter 3 mouvements : faire une rotation du plateau à 90° vers la droite, vers la gauche et retourner le plateau.

Les 3 opérations sont implémentés sensiblement de la même façon, on commence par ré-initialiser le tableau “hight” puis, en fonction du mouvement que l’on implémente, on parcourt le tableau d’une certaine manière et à chaque fois que l’on rencontre un chiffre différent de 0 on le met dans une copie de de la matrice 7x7 et on garde en mémoire l’état de chaque colonne dans la copie de la matrice à l’aide du tableau “hight” que l’on met à jour au passage pour la suite de la partie. À la fin on copie la matrice obtenue dans l’ancienne. (Pour plus de détails voir code).

Il reste ensuite à vérifier si le changement effectué à fait gagner l’un des deux joueurs. Pour cela on crée une nouvelle fonction “checkBoard” qui réutilise les 4 fonctions “check…” détaillées plus haut qui vérifie dans toute la matrice s’il y a un alignement. Pour cela à la place d’appeler chaque fonction “check…” sur chacune des cases de la matrice (énormément de calculs) on appelle les fonction “checkDiagonal1”, “checkVertical” et “checkDiagonal2” sur chaque case de la 4ème ligne de la matrice et la fonction “checkHorizonatal” sur chaque case de la 4ème colonne. On vérifie ainsi tous les cas possibles en un minimum de calculs. On veut que la fonction nous renvoie soit faux (0) si aucun des deux joueurs n’a gagné, vrai (1 ou 2) si le joueur ‘1’ ou le joueur ‘2’ remporte la partie, mais aussi le 3ème cas (3) si les 2 joueurs ont un alignement de 4 jetons.

Troisième étape : La présentation.

Rien de très intéressant à détailler ici, j’ai rajouter des printf et soigné la présentation pour avoir, après cette étape, un jeu qui tourne correctement et qui est simple à comprendre pour l’utilisateur.

À ce stade, il reste plus que le système de sauvegarde et l’IA à implémenter.

Quatrième étape : Le système de sauvegarde.

Pour pouvoir sauvegarder la partie en cours, il suffit de copier l’état de la partie en cours, qui est donc le contenu de la variable “game” définit en début de la partie, dans un fichier que l’on pourra charger lors de la prochaine utilisation. Pour copier le contenu de la variable, on copie chaque champs de la structure a la suite les un des autres dans ce fichier que j’ai appelé save.txt (Ici le .txt est facultatif). La matrice sera donc sauvegardée sur les 49 premiers caractères, le tableau “height” sur les sept suivants et les entiers “player” et “turn” sur les deux suivants. Une fois la partie sauvegardée on peut afficher un message et quitter le programme.

Avant chaque début de partie, on va donc demander à l’utilisateur s’il souhaite reprendre une précédente partie ou s’il souhaite en démarrer une nouvelle. Pour charger une partie, on a deux cas, soit aucune partie n’est sauvegardée et donc lors de l’ouverture du fichier “save.txt” le pointeur de type FILE pointera sur NULL, dans ce cas on affiche un message et on commence un nouvelle partie, sinon le pointeur pointera sur le fichier save.txt et on a juste à assigner à chaque champs de la variable “game” le contenue du fichier save.txt. On fait la même chose qu'au-dessus mais dans l’autre sens : les 49 premiers caractères sont ceux de la matrice, etc …

Cinquième et dernière étape : Le joueur autonome.

Comme précisé plus haut, je n'ai pas eu le temps de finir d'implémenter un joueur intelligent (beaucoup d'examens prévu à la rentrée et rajoute seulement 1 points par rapport à un joueur aléatoire), j'ai quand même laissé le code du joueur intelligent à la toute fin du fichier jeu.c au cas où vous voudriez y jeter un coups d'oeil.

Tout d'abord il faut demander à l'utilisateur avant de commencer ou reprendre une partie sauvegardé quels joueurs souhaite-il remplacer par un joueur autonome. Pour cela, j'ai créé une structure nommée 'ia' qui ne contient qu'un seul champ, un tableau de longueur 2.

Si le joueur 1 doit être remplacé par un joueur aléatoire, on change la valeur de la première case et de même pour le joueur 2.

Il suffit ensuite de créer la fonction qui renvoie un coups possible aléatoirement. Pour cela on commence par générer un nombre aléatoirement compris entre 0 et le nombre de colonne non pleine + 3. Si ce nombre - le nombre de colonne non pleine est compris entre 1 et 3, on fait une rotation, sinon on renvoie le numéro de la colonne non pleine correspondant à ce nombre.

Il reste juste à vérifier à chaque tour si c'est un bot qui va jouer ou un vrai utilisateur, pour cela on utilise la structure ia.