PG108 - Unix / Langage C
Projet : Puissance 4

Yannick Bornat - yannick.bornat@enseirb-matmeca.fr
Rémi Giraud - remi.giraud@enseirb-matmeca.fr
Éloi Navet - eloi.navet@u-bordeaux.fr
Jonathan Saussereau - jonathan.saussereau@ims-bordeaux.fr

2024 - 2025

1 Introduction

Le projet consiste à écrire un programme permettant de jouer au "Puissance 4". Dans un premier temps, nous nous contenterons d'écrire les fonctions de gestion et d'affichage. Dans un second temps, nous nous attaquerons à l'algorithme de jeu par le programme.

Rendu. Ce projet sera évalué par un rendu de votre code à chaque fin de séance via thor [1]. Il vous est donc fortement conseillé de sauvegarder les versions intermédiaires de votre programme lorsque vous avez réussi une étape importante. Vous rédigerez également un court rapport pour lequel vous est fourni un exemple de trame [2].

2 Environnement de départ

Règles du jeu. Ce jeu se compose d'une grille de 7 colonnes par 6 rangées. Chacun leur tour, les joueurs glissent des jetons (jaune ou rouge dans la version originale) dans une des colonnes où il reste de la place (le jeton occupe alors la position la plus basse). Le gagnant est celui qui arrive à aligner 4 jetons de sa couleur.

Structure du code. Ce projet devra être implémenté dans un seul fichier source nommé *puissance4.c*. Commentez votre code, vos fonctions, vos tests, en expliquant vos choix lorsque cela s'avère pertinent (quand vous avez choisi de créer une fonction auxiliaire, choisi un type de données ou qu'il vous était laissé le choix d'une solution pour répondre à un problème).

Nous utiliserons des variables globales pour simplifier la structure logicielle.
Ces variables seront :

- `NomJoueur1` : chaîne de caractère dont la taille maximale est 30
- `NomJoueur2` : chaîne de caractère dont la taille maximale est 30
- `JoueurEnCours` : entier de type `char` qui indique le numéro du joueur dont c'est le tour.
- `GrilleDeJeu` : un tableau de 42 cases qui sera vu comme un tableau de 7x6 (largeur x hauteur) de type `char`. Chaque case i correspond à un emplacement (c,r) de la grille de jeu tel que :
  $0 =< c =< 6$
  $0 =< r =< 5$
  $i = r*7+c$.

1. `https://thor.enseirb-matmeca.fr/ruby`
2. `https://remi-giraud.enseirb-matmeca.fr/teaching/course.php?course=PG108`

Pour chaque case de `GrilleDeJeu`, la valeur 0 signifie qu'elle est vide, la valeur 1 signifie qu'elle contient un jeton du joueur 1, la valeur 2 signifie qu'elle contient un jeton du joueur 2. Les autres valeurs sont interdites.

3 Implémentation

3.1 Initialisation

Écrivez la fonction `init` qui demande le nom de chacun des deux joueurs, qui stocke ces noms dans les variables `NomJoueur1` et `NomJoueur2`, et qui initialise `GrilleDeJeu` pour qu'elle ne contienne que des 0.

3.2 Affichage

Écrivez la fonction `display` qui affiche la grille de jeu, puis le nom du joueur qui doit placer son jeton. Les jetons du joueur 1 seront symbolisés par des '`X`', et les jetons du joueur 2 seront symbolisés par des '`O`'. Les cases vides seront symbolisées par des '.'

À ce stade, vous devriez être en mesure de tester votre programme (donc faites-le).

À fin de debuggage, il peut être utile de prévoir l'affichage d'un quatrième symbole dans la grille permettant de déterminer si une valeur autre que 0, 1 ou 2 y a été stockée.

Astuces UNIX (avancées)

Les terminaux UNIX sont compatibles avec le standard VT100. Ce standard définit des séquences (dites d'échappement) permettant une mise en forme basique du texte. Par exemple, il est possible de changer la couleur de texte, ou de modifier la position du curseur (entre autres). Ces commandes commencent toutes par le caractère non affichable numéro 27 (27=0x1B, donc noté `\x1B` dans une chaîne de caractères). Les commandes suivantes pourraient vous être utiles :

- `\x1B[2J` : effacer le terminal
- `\x1B[H` : placer le curseur en haut à gauche du terminal
- `\x1B[1m` : afficher le texte en gras
- `\x1B[91m` : afficher le texte en rouge
- `\x1B[93m` : afficher le texte en jaune

Les possibilités sont nombreuses, pour les connaître en détail, cherchez les commandes VT100 dans votre moteur de recherche préféré (ça donnera aussi de bons résultats avec un moteur de recherche que vous n'aimez pas...).

3.3 Phase de jeu

Écrivez la fonction `colonne_de_joueur` qui demande une colonne dans laquelle placer un jeton. Le joueur doit répondre un numéro de colonne entre 1 et 7 (inclus). Si le joueur a répondu une valeur de colonne incorrecte, la fonction lui demande un autre numéro de colonne (jusqu'à obtenir un numéro de colonne correct).

La fonction `colonne_de_joueur` renvoie un valeur compatible `c`, c'est à dire que le numéro de colonne est entre 0 et 6. Vous pouvez temporairement afficher cette valeur pour faciliter le debuggage.

Après avoir testé la fonction `colonne_de_joueur`, modifiez-la pour qu'elle place le jeton dans la grille de jeu. Vous pouvez tester cette fonction en l'appelant dans une boucle infinie après l'initialisation (la fonction affichage doit également être dans la boucle infinie).

Après avoir testé la modification précédente, modifiez-la pour qu'elle vérifie également si la colonne demandée n'est pas pleine. Si c'est le cas, la fonction considère que la colonne n'est pas valide.

N'oubliez pas de TESTER vos fonctions aussi souvent que possible!!!

3.4 Finalisation d'une version basique

À ce stade, il ne manque pas grand chose pour obtenir une version jouable à deux. Mais le programme est incapable de repérer la fin du jeu... Finalisez le programme pour qu'il soit jouable.

Pour déclencher la fin du jeu, vous le ferez de façon manuelle en interrompant l'exécution avec `ctrl-C`.

3.5 Détection d'une grille pleine

À partir de maintenant, nous allons nous intéresser à détecter la fin du jeu. Pour cela, nous allons utiliser la variable `JoueurEnCours`. Si elle vaut 11, c'est que le joueur 1 a gagné, si elle vaut 12, c'est le joueur 2 qui a gagné, si elle vaut 10, c'est une égalité, la grille est pleine sans qu'aucun des joueurs n'aie réussi à aligner 4 jetons.

Écrivez la fonction `detecte_plein` qui vérifie si la grille est pleine. Si c'est le cas, la fonction renvoie 1 et modifie `JoueurEnCours`, sinon, elle renvoie 0.

Modifiez votre programme pour qu'il tienne compte de `JoueurEnCours` pour la fin de la boucle principale.

3.6 Détection d'un vainqueur

Pour vérifier si quelqu'un a gagné, il n'y a pas le choix, il faut vérifier toutes les possibilités. Pour cela, nous allons créer la fonction `TestVainqueur`. Cette fonction reçoit la grille de jeu comme argument et effectuera ses tests sur la grille de jeu. Si la fonction trouve un vainqueur, elle renvoie le numéro du vainqueur, sinon, elle renvoie 0. Une fois sorti de la boucle principale, le programme doit bien entendu afficher le nom du vainqueur, ou préciser qu'il y a égalité.

Lignes horizontales d'abord

Créez la fonction `TestLigne` qui teste si, quelque part dans la grille, il y a 4 jetons en ligne horizontale. Cette fonction est destinée à être une sous-fontion de `TestVainqueur`. Le test doit s'effectuer sur une grille reçue en paramètre, pas directement sur `GrilleDeJeu`.

Testez si ce test fonctionne.

Lignes verticales ensuite

Créez la fonction `TestColonne` qui teste si, quelque part dans la grille, il y a 4 jetons en ligne verticale. Cette fonction est le pendant de `TestLigne` et fonctionne de la même façon.

UNIX, pour gagner en productivité

Testez (Encore). Vous remarquez que les tests deviennent un peu pénibles, car il devient assez long de produire la situation à valider. Pour vous en sortir, il y a la possibilité d'utiliser une redirection UNIX. Écrivez à l'avance toutes les réponses dans un fichier texte (une réponse

par ligne). Vous pouvez alors utiliser la redirection au lieu de taper directement les réponses grâce à la syntaxe suivante :

```
./puissance4 < fichier_de_test
```

Diagonales (pente positive)

Ce sera la fonction `TestDiagPos`. Pour le test des diagonales de pente positive, l'algorithme est proche de celui de `TestColonne`, mais au lieu de tester les séries (x,y) = (k,i) (ou k est la colonne testée, de 0 à 6), on teste les séries (x,y) = (k+i, i) où k identifie la diagonale testée (de -2 à 3). Il faut également tenir compte du fait que cette façon de parcourir les éléments d'une diagonale àmène à des coordonnées de case qui n'existe pas, et qu'il faudra donc repérer pour les ignorer.

TESTEZ cette fonction

Diagonales (pente négative)

Ce sera la fonction `TestDiagNeg`, elle s'intéressera aux coordonnées de la forme (x,y)=(k+i, 5-i) pour k de -2 à 3.

Une fois cette fonction testée, le jeu est jouable à deux joueurs.

3.7 Jeu contre le programme

Pour jouer contre l'ordinateur, nous considérerons qu'il suffit de ne pas donner de nom pour le deuxième joueur (on pourra utiliser `fgets` plutôt que `scanf`). Il faut ensuite déterminer quelle est la colonne que doit choisir l'ordinateur (en fait votre programme) à son tour de jeu.

La méthode aléatoire

Pour mettre en place le jeu contre le programme, nous utiliserons une méthode de choix aléatoire. À défaut d'être intelligente, cette méthode est rapide et permettra de se concentrer sur les modifications du programme déjà existant.

- Écrivez une fonction `CalculeColonneAleatoire` qui renvoie un entier. Cette fonction tire une valeur aléatoire à l'aide de la fonction `rand()` disponible dans `stdlib.h` (consultez `man 3 rand` pour plus d'informations). Cette valeur sera ramenée entre 0 et 6 grâce à la fonction modulo (`%`). Tant que la colonne correspondant à cette valeur est pleine dans `GrilleDeJeu`, la valeur sera augmentée de 3 puis ramenée entre 0 et 6 par un modulo, sinon, la valeur est renvoyée.
- Ajoutez une variable globale `Joueur2Virtuel` de type `char`. Si cette valeur est différente de 0, cela signifie que le jeu se fait contre l'ordinateur.
- Modifiez la fonction init pour qu'elle détermine la valeur de `Joueur2Virtuel` en fonction de `NomJoueur2`.
- Modifiez la fonction `colonne_de_joueur` pour que, si on joue contre le programme et que c'est au programme de jouer, le numéro de colonne ne soit pas demandé à l'utilisateur, mais fourni par la fonction `CalculeColonneAleatoire`.

Testez vos modifications (vous ne devriez pas avoir de problème pour gagner contre le programme)

La méthode min-max

Une méthode plus pertinente (à défaut d'être efficace) consiste à tester toutes les possibilités. Cette méthode (communément appelée min-max) simule donc ce qu'il se produit pour un jeton

dans chaque colonne. Pour anticiper la suite du jeu, toutes les ripostes du joueur sont également testées, et ainsi de suite. La succession de coups est testée grâce à une fonction récursive. Dans chaque cas, la fonction retourne un résultat indiquant si la suite de coups a mené à une victoire, une défaite, ou ni l'un ni l'autre.

L'explosion combinatoire des cas à envisager rend inenvisageable une exploration exhaustive des évolutions possibles. On s'arrête alors à quelques coups (7 dans notre cas). Cette technique est essentiellement défensive et considère que le joueur humain est de très bon niveau, elle amène donc un comportement qui cherche à ne pas perdre plutôt qu'à élaborer une stratégie de victoire.

L'écriture de cette fonction dépasse très largement le niveau d'exigence de cet exercice. Cette partie est donc fournie dans les fichiers *ai_player.c* et *ai_player.h*. *AI_player.c* contient la fonction `CalculColonneAI` qui a besoin d'accéder à `TestVainqueur` et à `GrilleDeJeu`. L'objectif de cet exercice est donc d'effectuer une compilation multi-fichiers avec le fichier *AI_player.c*. Écrivez votre fichier d'entête (.h) et modifiez votre programme pour que la compilation fonctionne avec deux fichiers C séparés.

AI_player.c est écrit de telle sorte que votre fichier d'en-tête se nomme *puissance4.h*. Modifiez cette ligne au besoin.

3.8 Jeu adaptatif

La technique Min-Max donne déjà de très bons résultats, cependant, elle reste limitée face à un joueur expérimenté. Pour résoudre ce dilemme, il est possible de rendre l'algorithme adaptatif. La technique consiste à modifier le tableau priority dans *AI_player.c* pour qu'elle s'adapte au joueur que le programme affronte. Ce tableau permet de pondérer les choix à effectuer lorsque la recherche Min-Max ne parvient pas à identifier un cas unique. Plus la valeur d'un élément est élevée, plus la case correspondante sera favorisée.

La technique est simple, mais se montre tout de même efficace : En fin de partie, les cases correspondant à une pièce du vainqueur voient leur valeur augmentée, celles correspondant à une pièce du perdant sont diminuées.

Pour implanter cette technique, il faut bien sûr que priority soit stocké dans un fichier pour être modifié, et non plus en dur dans le programme.

1. Modifiez vos fichiers pour que priority soit maintenant un tableau de `float` déclaré dans votre programme (seul *AI_player.h* n'a pas besoin d'être modifié).
2. Modifiez votre partie du programme pour que priority soit chargé depuis le fichier *priority.txt* (que vous aurez téléchargé) au début du programme.
3. Ajoutez les fonctions de mise à jour de priority en fin de partie :
   — en cas de défaite, la valeur est réduite selon la formule suivante : v = v*0.9
   — en cas de victoire, la valeur est augmentée selon la formule suivante : v = 2 + v*0.9 (cette valeur est bien augmentée et converge vers 20 si v est inférieur à 20)
4. Enregistrez priority dans le fichier *priority.txt* après mise à jour. Vous pouvez faire cet modification/enregistrement, même sur une partie entre deux joueurs humains, le programme gagnera ainsi de l'expérience même lorsqu'il joue pas.

4 Pour aller plus loin

Liste non exhaustive d'améliorations possibles qui peuvent être développées en parallèle.

4.1 Temps moyen de jeu (+)

Que ce soit pour l'utilisateur ou pour la machine, on peut également calculer le temps moyen de jeu pour chaque partie. Après chaque partie, affichez le temps mis par la machine ou le joueur ainsi que la moyenne si vous effectuez plusieurs parties.

4.2 Couleur (+)

Pour un aspect visuel plus poussé, on peut intégrer de la couleur à tous les affichages.

4.3 Nombre de cases (+++)

On pourrait imaginer jouer à Puissance N avec un nombre de cases différent. Les modifications à faire au programme sont nombreuses dans ce cas.

4.4 D'autres idées ?

Libre à vous de proposer d'autres améliorations de votre programme (log avec profil, high score, temps limite pour jouer, ...).