Projet IF2B P22

Really Bad Chess

—

THOMAS YANN - LAMEY NATHAN
Groupe U - TC02

Introduction :

Notre projet consistait à développer en langage C le code du jeu Really Bad Chess, une variante du jeu d'échecs.

Pour la réalisation de ce projet, nous avons utilisé GitHub pour pouvoir travailler en collaboration sur le projet même en étant à distance. Git est un outil puissant permettant une sauvegarde en branche des différentes modifications apportées au programme et permettant la fusion de certaines d'entre elles, nous permettant de pouvoir travailler sur le projet en simultané pour au final fusionner nos deux projet en gardant les modifications voulues et avoir un projet commun.

Voici le lien de notre projet GitHub en ligne et notre documentation généré par Doxygen.

Le concept de Really Bad Chess s'appuie sur le fait que les pièces que recevra le joueur sont complètement aléatoires. Un joueur pourra se retrouver avec 5 Reines contre un joueur avec seulement des pions. D'où le nom de la variante "Really Bad Chess", qui est un jeu totalement inégal pour les deux joueurs. La projet reprend les règles des échecs classiques mais on ne codera pas certaines règles comme les roques, les promotions ou les prises en passant.

Voulant tous les deux partir en branche info a la fin de notre tronc commun, nous étions motivé à la bonne réalisation du projet. Nous avons choisi ce sujet car nous apprécions tous les deux les échecs et que par coïncidence, Nathan était par ailleurs en train de coder pour lui-même un jeu d'échecs en JavaScript ce qui nous a poussé d'autant plus vers ce sujet.

Au travers de ce rapport nous allons tout d'abord présenter les différents modules de notre projet puis expliquer les fonctions qui y sont présentes. Nous expliquerons ensuite les différentes fonctionnalités du programme pour enfin parler des difficultés que nous avons rencontrées.

I. Présentation générale :

Au cours de notre projet nous avons créé 4 fichiers .h et leurs fichiers .c associés en complément de notre main.c pour regrouper nos fonctions en fonctions de leurs utilisations et ainsi aider à la compréhension générale du projet :

- **saisie.c/.h**

  Cette paire de fichiers regroupe toutes nos fonctions qui demandent une information à l'utilisateur ou qui affiche une information directement destinée à ce dernier.

- **board.c/.h**

  Notre paire de fichiers board contient tout d'abord notre structure de pions et toutes les fonctions qui sont liées au plateau de jeu.

- **verif.c/.h**

  Cette paire de fichiers regroupe toutes les fonctions qui concernent les vérifications des déplacements et des échecs / mat.

- **save.c/.h**

  Cette paire quant à elle regroupe les fonctions pour sauvegarder et charger une partie de jeu.

- **game.c/.h**

  Notre paire de fichiers game va regrouper des fonctions qui étaient de base en vrac dans notre .main. Elles nous ont servi à épurer notre fichier .main pour permettre une meilleure lisibilité du programme.

II. Présentation des fonctions :

a. saisie.c/.h

Ce module va fonctionner en dépendance avec le module vérifie, principalement pour la fonction printErr. Elle va également fonctionner avec notre main.

- **askMenu() :** cette fonction est utilisée dans le main est sert à l'utilisateur de choisir l'option qu'il veut dans notre menu.
- **askTailleplateau() :** cette fonction demande simplement la taille de plateau que le joueur souhaite entre 6 et 12.
- **askDeplacement() :** cette fonction va demander au joueur le déplacement qu'il souhaite exécuter et va le stocker dans un tableau de 2 x 2 appelé "move", tableau qui va être régulièrement utilisé dans les différents modules du programme.
- **askOuiNon() :** cette fonction permet de poser une question à l'utilisateur à laquelle il peut répondre par oui ou par non.
- **sign() :** cette fonction simple va retourner le signe d'un nombre qui est rentré par un 1 ou un -1. Elle est notamment utilisée pour les vérifications des déplacements.
- **printErr() :** cette fonction est la fonction qui va retourner les messages d'erreur pour le joueur en fonction des vérifications faites dans le module verif (ou ne vas rien faire si le mouvement est valide).
- **setColor():** fonction affichant les couleurs sur le texte et le fond.

b. board.c/.h

Ce module est lié à notre main et au module verif pour la fonction chercherRois.

- **genererPlateau() :** cette fonction va générer dans chaque case du tableau représentant notre plateau une pièce. Elle générera des pièces de type vides pour toutes les cases autres que les 2 premières et 2 dernières lignes.
- **afficherPlateau() :** cette fonction va servir à afficher le plateau de jeu en fonction des informations qui sont stockées dans chaque case de notre plateau. Elle permet aussi de mettre les couleurs grâce à la fonction setColor du module saisie.
- **executeMove() :** cette fonction est celle qui nous permet d'effectuer les déplacements de nos pièces à travers le plateau.
- **undoMove() :** cette fonction va annuler le dernier mouvement effectué. Elle sert notamment pour la vérification échec et celle mat.

- **chercherRois() :** cette fonction permet de localiser les rois sur le plateau de jeu. Elle est utilisée dans le fonction verifEchec.

c. verif.c/.h

Le module verif est lié au module saisie pour les fonctions sign et printErr, cette dernière fonction qui va indiquer au programme si le mouvement est valide ou non.

- **verifMouvement() :** cette fonction va centraliser les différentes vérifications et retourner les codes de chaque fonction en fonction de la pièce que le joueur veut bouger sous une seule et même variable.
- **verifDeplacement() :** cette fonction va récupérer la variable de la fonction verifMouvement et va l'utiliser pour déterminer dans quel cas on se trouve pour les vérifications. Elle va renvoyer un entier qui va être utilisé pour retourner un message d'erreur au joueur.
- **verifPion() :** vérification du déplacement d'un pion.
- **verifFou() :** vérification du déplacement d'un fou.
- **verifCavalier() :** vérification du déplacement d'un cavalier.
- **verifTour() :** vérification du déplacement d'une tour.
- **verifDame() :** vérification du déplacement d'une dame.
- **verifRoi() :** vérification du déplacement d'un roi.
- **verifEchec() :** vérification de si un des rois est en échec.
- **verifMat() :** vérification d'un potentiel échec et mat quand un échec est détecté.

d. save.c/.h

Ce module est directement lié à notre main.c mais également au game.c pour la sauvegarde du jeu.

- **saveGame() :** cette fonction va servir à enregistrer la partie en cours dans un fichier. Elle stock dans l'ordre la taille du plateau, le nombre de tour joué et ensuite les informations de chaque case de notre plateau
- **loadGame() :** cette fonction sert à charger une partie. Elle lit ce qui se trouve dans le fichier save.txt puis traduit pour rentrer les informations là où il faut et reprendre la partie comme elle s'est arrêtée.

e. game.c/.h

Ce module est lui aussi directement lié à notre main.c.

- **game() :** cette fonction gère tout le déroulement d'une partie.

III. Fonctionnalités du programme :

a. Déroulement du programme

Lors du lancement du programme, le joueur se voit confronter au menu principal. Il a alors le choix entre commencer une nouvelle partie (1), reprendre la dernière partie sauvegardée (2) ou alors quitter le programme (3).

Reprendre la dernière partie (2) va faire appel à la fonction loadGame et va faire reprendre la partie au joueur comme il l'avait laissé quand il a sauvegardé.

Si le joueur veut démarrer une nouvelle partie, le programme va lui demander quelle taille de plateau il souhaite avoir puis la partie va commencer. Le plateau va alors s'afficher sur la console et un texte apparaîtra pour indiquer à quel joueur c'est le tour, le joueur pourra alors rentrer dans la console le mouvement qu'il souhaite effectuer sous la forme a1b2 (coordonnée de départ et coordonnée d'arrivée, en majuscule ou en minuscule). Si le mouvement est valide, rien ne s'affiche et le programme va continuer et passer au deuxième tour en affichant à nouveau le plateau et en demandant au joueur suivant le mouvement qu'il souhaite jouer.

Mais si le mouvement n'est pas valide (donc que les vérifications ont détecté une erreur), un message s'affiche, indiquant l'erreur qu'a commis le joueur. D'autres messages peuvent s'afficher au cours de la partie comme quand un joueur est en échec, le programme l'affiche.

Quand un joueur est échec et mat, le programme indique quel joueur à gagné et le programme se termine en libérant la mémoire des différents tableaux dynamiques. Il va ensuite demander au joueur s' il souhaite rejouer et se terminer dans un cas ou repartir du menu dans l'autre.

b. main.c

Concrètement, une partie va se dérouler de la sorte:

Une fois les paramètres de la sauvegarde et les variables ont été initialisés, le programme rentre dans une boucle pour la proposition du menu au joueur.

Après que le joueur ait fait son choix, le programme va faire appelle à la fonction game() qui va tourner tant que la partie ne sera pas finie. Cette fonction va tout d'abord créer le tableau "move" et va ensuite gérer toute la partie avec les conditions de mat, d'échecs etc.. Cette fonction va être l'épicentre du jeu car la plupart des fonctions de notre projet vont avoir des répercussions sur cette fonction.

Par la suite, une fois la partie terminée, le programme va libérer la mémoire des tableaux dynamiques qui ont été créés.

c. Choix réalisés

Au cours de l'avancement du projet nous avons dû effectuer différents choix pour mener au mieux la réalisation du projet.

-L'un des premiers choix fut la structure de nos pièces. Avant le début du projet nous avons réfléchi à dans quelle direction nous devions nous diriger pour le plateau, les pièces etc..

Après avoir décidé que notre plateau serait un tableau dynamique, nous avons donc réfléchi à comment gérer nos pièces sur le plateau. Notre première idée fut de rentrer dans chaque case du plateau où il y avait une pièce, l'initiale du nom de la pièce et l'initiale de sa couleur (par exemple PN pour un pion noir), mais nous avons très rapidement abandonné cette idée pour nous focaliser sur les structures.

La structure que nous avons choisi pour nos pièces est la suivante:

```
typedef enum {
    VIDE, PION, FOU, CAVALIER, TOUR, DAME, ROI
    //0   1     2    3         4     5     6
} TypePiece;

typedef enum {
    NONE, BLANC, NOIR
    //0   1      2
} CouleurPiece;

typedef struct {
    TypePiece typePiece;
    CouleurPiece couleurPiece;
    int nbMove;
} Piece;
```

Chaque pièce est définie par son type de pièce, sa couleur et nous avons ensuite rajouté le nombre de mouvement effectué par la pièce (pour les pions qui peuvent avancer de 2 cases pour leur premier mouvement). Nous avons également pensé à rajouter un type de pièce vide pour que chaque case du plateau possède une structure ce qui nous à faciliter le travail pour les vérifications de déplacement.

-Un autre choix qui à été fait fut de retravailler intégralement toutes les fonctions de vérifications pour optimiser notre programme et éviter d'avoir un trop grand nombre de lignes ce qui pourrait nuire à la compréhension du programme. Par exemple, la vérification des cavaliers est passée de 39 lignes à seulement 8.

-En ce qui concerne nos déplacements, qui sont utilisés dans les vérifications, nous avons choisi de créer un tableau de 2 x 2 pour enregistrer le mouvement que le joueur souhaite effectuer, comme décrit précédemment.

Move	Départ	Arrivé
x	[0;0]	[0;1]
y	[1;0]	[1;1]

Ce tableau “move” nous à été extrêmement utile car il nous permettait de localiser la pièce que le joueur voulait déplacer et la case où il voulait se rendre. Combiné avec notre structure des pièces, les vérifications ont été simples à faire.

```
} else if (board[move[0][1]][move[1][1]].couleurPiece == board[move[0][0]][move[1][0]].couleurPiece) {
    validite = 6;   //mange propre piece
```

Par exemple ci dessus. “board[move[0][1]][move[1][1]].couleurPiece” représente la couleur de la pièce de la coordonnée du plateau de la pièce que le joueur veut bouger. Ici on la compare à la couleur de la pièce qui va être aux coordonnées d’arrivés que le joueur à choisi.

-Par la suite, quand nous nous approchions de la fin du projet, nous avons décidé d’essayer de mettre des couleurs à notre plateau. Pour cela nous avons créer une fonction setColor et une structure comprenant un grand nombre de couleur pour pouvoir voir quel couleur nous allions prendre et pour pouvoir changer quand nous le voulions. Nous avons donc implémenté notre fonction setColor dans notre fonction pour afficher le plateau pour avoir notre plateau en couleur.

-Enfin, l’un des derniers choix que nous avons réalisés fut de nettoyer un peu notre fichier main.c. En effet, cela faisait longtemps que notre main était en désordre et sa compréhension en était devenue compliquée. C’est pour cela que nous avons créé une nouvelle paire de fichier, la game.c/.h pour y mettre des parties de notre main sous forme de fonctions. Nous y avons alors créé la fonction game() qui répertorie tout le fonctionnement d’une partie de jeu.

IV. Difficultés rencontrées :

a. Échec / mat

L'une des principale difficulté que nous avons rencontré est sur la vérification des échecs et des mats. Nous avons eu des difficultés durant toute une semaine, sur les échecs et les mats.

Au début nous pensions que le problème venait de la fonction verifMat car c'est la dernière à avoir été faite et que nous n'avions pas trop de problème avec notre fonction verifEchec jusqu'alors.

Finalement, nous n'avons trouvé que très tard, après quelques jours de recherche, que l'un des problèmes venait de la verifEchec à cause d'un dépassement de mémoire en recherchant le code d'erreur que le programme nous renvoyait sur internet. Nous avons donc passé en revue tout le code pour au final nous apercevoir que le problème venait de certains malloc que nous n'avions pas libérés avec des free et d'astérix qui avaient été oubliés dans la définition des malloc.

Un peu plus tard, nous avons trouvé un autre problème sur la fonction verifMat. En effet, quand l'un des joueurs était en situation d'échec et mat mais que le joueur dans cette situation pouvait mettre l'autre roi en échec, alors la situation se renversait. Le problème venait du fait que la boucle qui vérifiait si un roi était en échec ne vérifiait que le dernier échec qui à été enregistré, donc si l'autre roi passait en échec, alors le premier roi n'était plus détecté comme telle pour le tour. Nous avons donc réglé le problème en utilisant des booléens qui permettent aux deux rois d'être vérifiés en même temps et d'empêcher ce genre d'erreur de se reproduire.

b. Mouvement Pion

Un autre problème auquel on à été confronté à été le déplacement des pions. Pour vérifier que le déplacement était valide dans notre fonction verifPion nous avions dans un premier temps une fonction beaucoup trop longue qui passait en revu chaque cas possible mais quand nous avons décidé de refaire intégralement les vérifications, la verifPion à aussi été refaite mais elle ne marchait plus correctemet. On s'en ait rendu compte quand à un moment, nous testions la verifMat et que le programme nous à annoncé que pour éviter l'échec et mat, il fallait que le pion se décale de deux cases à l'horizontal, ce que nous avons essayé de faire et ce qui à marché.

Nous avons donc passé en revu tout le code et l'avons refait une troisième fois pour essayer de trouver une solution. Le problème venait en fait d'un manque de conditions dans la fonction ce que nous avons réglé en retravaillant la fonction. Notre fonction verifPion n'est maintenant plus aussi condensée qu'avant mais cependant, elle fonctionne correctement.

Bien que le pion soit la pièce la plus simple du jeu d'échecs, c'est également celle qui possède le plus de règles de déplacement et qui est donc la plus difficile à programmer.

Conclusion :

Pour conclure, notre projet à évolué tout au long de sa réalisation avec l'implémentation de nouveaux modules jusqu'à la fin du projet, des idées implémentés que nous avons changés par la suite, des modules que nous avons entièrement refait pour condenser le code ou encore des idées d'amélioration qui sont arrivés jusqu'aux derniers moments du projet.

Malheureusement certaines de nos idées n'ont pas pu être implémentées par manque de temps, principalement des idées concernant l'affichage du jeu sur la console. Par exemple, nous avions pour idée qu'a chaque tour qui passait, la console se viderait, permettant de ne pas saturer la console et offrant une meilleur expérience de jeu. Une autre de nos idées aurait été un meilleur affichage avec le plateau centré au centre de la console et les règles et instructions de jeu affichés à chaque tour une fois la console vidé, en lien avec notre idée précédente.

Nous avons fait un doxygen de notre projet mais malheureusement nous n'avons pas pris assez de temps pour travailler dessus et il ne ressemble au final pas à ce que nous aurions voulu que ce soit.

En prenant un peu de recul, nous aurions pu améliorer certaines choses ou en faire différemment. Par exemple nos fonctions auraient pu être mieux répartis à travers les modules car certaines d'entres elles se trouvent dans un module alors qu'une autre fonctions qu'il lui est complémentaire est dans un autre (la fonction chercherRois du module board ou la fonction sign du module saisie qui sont uniquement utilisés dans le module verif). Une autre chose que nous aurions pu faire est d'encore plus épurer notre main.c, nous l'avons un peu fait avec la création du module game.c/.h mais le main reste encore trop chargé et trop peu lisible.