Projet final IPI - Jeu de plateau

Programmation Impérative

Lucas RODRIGUEZ

23 Décembre 2020

Table des matières

1 Introduction

1.1 Objectif général

L'objectif principal de ce projet est d'implémenter un jeu de plateau donné en énoncé, en utilisant les ressources de la programmation impérative. Nous prendrons donc soin d'utiliser des structures de données adaptées au problème et de les implémenter judicieusement, afin de mener à bien la conception du programme final.

Pour cela, nous nous intéresserons plus particulièrement à des problématiques de gestion de mémoire, afin d'optimiser le plus possible l'exécution de notre programme, et d'éviter tout gaspillage lors d'allocations dynamiques.

Nous découperons notre travail en plusieurs fichiers distincts afin de rendre l'ensemble du rendu lisible et compréhensible par un lecteur extérieur.

1.2 Principe et fonctionnement

Tout d'abord, attardons nous sur quelques notations.

1.2.1 Notations et définitions

Ces notations seront valables tout le long de ce rapport. Les variables analogues seront implémentées sous le même nom.

— j : nombre de joueur
— n : taille d'un plateau
— $i \in [\![1, j]\!]$: numéro d'identification du joueur
— $\Sigma = [\![1, j]\!]$: ensemble des numéros des joueurs

Définition 1. *La grille est un matrice de cases.*

Définition 2. *Une case (x, y) est un emplacement pour sa pile correspondante.*

1.2.2 Fonctionnement général du jeu

Le jeu se présente graphiquement sous forme de plateau carré.

Nous demandons à l'utilisateur de rentrer dans la console (lecture sur le flux d'entrée standard (stdin)) d'abord j, puis n.

Chaque joueur est alors identifié par un entier i entre 1 et j.

A chaque tour i (car chaque joueur joue à tour de rôle dans l'ordre croissant des i), on effectue successivement les tâches suivantes :

1. Affichage du plateau dans son état actuel
2. Demande à l'utilisateur d'entrer les coordonnées d'une case souhaitée pour placer un jeton
3. Affichage de la pile située à la case correspondante
4. Confirmation pour validation du placement du jeton
5. Traitement d'activation des cases si nécessaire selon le schéma ci-dessous

Le jeu se termine lorsque le plateau possède au moins un jeton sur chacune des cases le composant.

Même si les quatre premières étapes sont longues à composer, la dernière a constitué la plus grande charge de travail, car c'est celle qui a nécessité de "jongler" entre plusieurs structures de données à la fois. De plus, c'est cette étape 5 qui m'a demandé le plus de réflexion, le plus de travail, et également celle sur laquelle je suis resté bloqué le plus longtemps. Je prendrai soin par la suite de détailler précisément les problèmes rencontrés, les pistes étudiées et enfin la solution trouvée et par conséquent implémentée.

1.2.3 Traitement de l'activation des cases

Pour cette étape 5, nous considérons l'ensemble des quatre cellules adjacentes à celle-ci (N/S/E/O) comme définition du voisinage d'une case ; dans la mesure où elles sont toutes dans la grille. Dans le cas contraire, on fait les modifications nécessaires (cas de coins ou de bords).

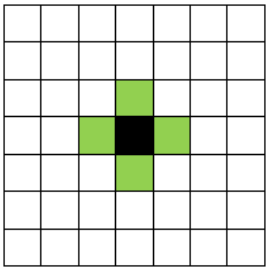

FIGURE 1 – Les cases vertes représentent le voisinage de Von Neumann

A i fixé, lorsque l'on place un jeton i sur une pile (représentée par une case), il faut vérifier si les 2 jetons au-desus de la pile (c'est-à-dire celui que l'on vient de poser et celui juste en dessous) sont identiques ou non.

Dans le cas où les jetons sont identiques, la case est **activée**. On retire alors les 2 jetons de cette pile, puis on place alors sur le voisinage N/S/E/O de cette case, des jetons de la même couleur, à condition que cette case ne soit pas déjà activée par i à ce tour-ci. On effectue ce processus jusqu'à ne plus avoir de cases activées à traiter.

2 Conception de la solution

Avant de détailler la stratégie envisagée, détaillons les structures de données implémentées.

2.1 Structures de données

Nous devons implémenter un plateau où chaque case est une pile où l'on peut empiler et dépiler quand on le souhaite des jetons.

La solution la plus simple consiste alors à construire un objet semblable à une matrice où chaque élément est une pile.

Il faudra alors créer des fonctions primitives adaptées à chacune des structures.

Concrètement, nous allons découper l'implémentation de ces structures en 2 modules distincts :

— stack : pour la gestion les piles
— grid : pour la gestion du plateau

Remarque 1. *L'écriture d'un module se fait dans un fichier distinct des autres pour rendre l'organisation structurelle du projet cohérente et lisible par tous.*

Remarque 2. *J'ai réalisé les implémentations suivantes en utilisant l'éditeur de code Visual Studio Code, combiné au compilateur gcc, le tout sous un environnement UNIX.*

2.1.1 Module stack

Pour implémenter ce module, on définit en premier lieu les types abstraits et concrets utiles pour la création d'une pile. Il nous faut donc un type pour la représentation des jetons que l'on empilera, et une structure pour la pile elle-même.

Remarque 3. *Afin d'obtenir une implémentation propre et flexible, j'ai préféré utiliser des listes chaînées, elles-même implémentées par mes soins, afin d'être plus économe en mémoire.*

```
typedef int jeton;

struct stack_base{
    jeton v;
    struct stack_base *next;
};
typedef struct stack_base *stack;
```

On définit également les prototypes des primitives pour la gestion du contenu des piles.

```
stack empty_stack();
int is_empty_stack(stack s);
void push(jeton j, stack *s);
void pop(stack *s);
void print_stack(stack s);
```

La déclaration des types abstraits et concrets et des prototypes des fonctions sont disponibles dans le fichier **stack.h**. L'implémentation des fonctions est disponible dans le fichier **stack.c**.

2.1.2 Module grid

Nous devons également créer un module pour la création et l'affichage d'une grille de piles.

Remarque 4. *L'implémentation du type stack doit être réalisée avant.*

```
struct matrix{
    stack **t;      // Matrice de piles
    int n;          // Taille de la matrice
};
typedef struct matrix grid;
```

Puis, on déclare les prototypes suivants :

```
grid grid_init(int n);
void grid_print(grid g);
void grid_print_stack(grid g, int x, int y);
void grid_insert(grid g, int x, int y, int k);
int is_grid_complete(grid g);
```

J'ai fait le choix de manipuler des tableaux 2D dynamiques, afin de ne pas être limité lors d'éventuelles implémentations futures.

2.2 Implémentation des premières étapes

Dans cette sous-partie, nous allons détailler la démarche prise afin de concevoir l'implémentation des 4 premières étapes décrites ci-dessus.

Ces étapes reposent pour la plupart sur des entrées utilisateurs : j, n, coordonnées, confirmation, ...

Entrée utilisateur Afin de gérer les entrées utilisateurs, on utilise un buffer. Ici un exemple pour la saisie du nombre de joueur :

```
#define BUFFER_LENGTH 256
char buffer[BUFFER_LENGTH];

int j;
printf("Nombre de joueurs (j) : ");
fgets(buffer, sizeof(buffer), stdin);
sscanf(buffer, "%d", &j);
```

Pour la gestion de la saisie des coordonnées des cases ainsi que de la confirmation, j'ai préféré créer une fonction qui centralise toutes les commandes essentielles.

On utilise une fonction `ask_coords` afin de gérer les entrées de coordonnées et de confirmation par l'utilisateur, sur l'entrée standard (stdin). [1]

Pour plus de détails, voir le fichier main.c.

2.3 Implémentation de l'activation des cases

Cette étape a été pour moi la plus compliquée à mettre en place. Voici mes pistes de résolution :

2.3.1 1ère tentative : utilisation des files

Idée L'idée était d'implémenter cette étape en utilisant une file. On enfile à chaque fois, les sommets que l'on doit traiter ensuite. Tant que la file n'est pas vide, cela indique au programme qu'il reste des éléments à traiter. Dans ce cas, nous ne passons pas au joueur suivant et nous continuons à réaliser le traitement de la case défilée.

Implémentation J'ai donc implémenté un 3ème module à mon projet : le module **queue**. Il permet de créer une file, puis d'enfiler ou de défiler.

Les éléments constituants de cette file d'attente sont des tableaux de 2 entiers : correspondant aux coordonnées x et y des cases à traiter dans le futur.

1. On pourra également utiliser l'opérateur de redirection du contenu d'un fichier vers l'entrée standard.

Lorsqu'une case est activée, on enfile les cases de son voisinage, pour ensuite les traiter de la même manière, à condition qu'elles n'aient pas été déjà activées.

Problème rencontré Le problème majeur de cette solution est relative au fonctionnement même de la file. En effet, je pensais naïvement avant mon implémentation finale que la file allait se comporter comme dans l'exemple donné dans l'énoncé.

Cependant, le fonctionnement de défilement est **successif**, tandis que le traitement du voisinage du jeu doit se faire **simultanément**.

En effet, au lieu de traiter les cases du voisinage d'une case en même temps, elle les traite une par une.

Si à un moment donné, la file contient les cases (A, B, C) où A est la case prête à être défilée. Lorsqu'on défile A, on modifie la grille en fonction de A, puis on défile B et on modifie la grille déjà modifiée par l'action de A.

On effectue ainsi modifications par modifications alors que le fonctionnement du jeu détaillé dans l'énoncé prône une modification simultanée des cases, sur une même grille.

$\Longrightarrow$ **L'utilisation d'une file n'est donc pas une bonne option.**

2.3.2 2ème tentative : utilisation de la récursivité

Afin de trouver une solution au problème soulevé ci-dessus, nous nous sommes tourné vers la récursivité.

Nous nous sommes concentrés sur la création d'une nouvelle fonction de traitement pour une case. Cette dernière traite d'abord la case, puis regarde son voisinage en cas d'activation. En fonction de l'activation des cases voisines, il effectue un nouvel appel sur ces cases. Et ainsi de suite, jusqu'à ce que la fonction n'active plus de nouvelles cases.

On peut démontrer la terminaison de cette fonction.

Proposition 1 (Terminaison de l'algorithme)**.** *L'exécution de la fonction de traitement des cases nécessite un nombre fini d'opérations. Par conséquent, elle s'exécute en un nombre fini de temps.*

Démonstration. Le nombre de cases du plateau est n^2. On se place au début du tour $i \in \Sigma$.

Vu que nous considérons dans ce problème uniquement les 4-voisinages des cases, un appel de la fonction va donc appeler au plus les 4 cases du voisinage (en fonction de leur éventuelle activation passée). Dans l'hypothèse où à chaque appel de la fonction, la configuration de la grille permet l'activation chacune des cases du voisinage, le nombre de cases activées va tendre, au fil des tours, vers n^2 [2]

Lorsqu'il atteint n^2, on ne peut donc plus activer de nouvelles cases, donc on ne peut plus faire d'appel à la fonction. La fonction au tour i a activé toute les cases, donc on passe au tour $i + 1$ (où aucune case n'est activée).

La terminaison de cette fonction est vérifiée par la taille finie de la grille et par le fait qu'on ne peut pas traité les cases déjà activées. ■

$\Longrightarrow$ **L'utilisation de la récursivité est donc une bonne solution.**

2. Le nombre de cases activées est majoré par n^2.

2.4 Fin du programme

Lorsque la grille est remplie, la partie principale du traitement est terminée. Il faut ensuite compter le nombre de jetons de chaque joueur, sur la tête de chaque pile. Celui qui possède le plus grand nombre de jetons gagne la partie

Proposition 2. *Le programme se termine.*[3]

Démonstration. En effet, à chaque étape, on change de tour et on remplit petit à petit les cases de la grille. A partir d'un certain tour, la grille sera complètement remplie. ■

Remarque 5 (Etape primordiale)**.** *A la toute fin, quand on n'a plus besoin des variables, le programme **doit** libèrer toute la mémoire allouée dynamiquement lors de la création des maillon dans les piles ou bien lors de la création de la grille ou d'autres tableaux éventuels. Pour cela, on dépile chaque pile, puis on exécute autant de free, que l'on a exécuté de malloc.*

2.5 Compilation du programme final

Pour exécuter le programme,

1. Ouvrir un terminal dans le répertoire du projet.
2. Exécuter la commande `make`.
3. Exécuter la commande `./prog` (ou `./prog < partie.txt` pour tester avec la partie de test).

3 Conclusion

3.1 Résultat avec le fichier test

Au final, en exécutant `./prog < partie.txt`, nous obtenons le tableau suivant :

```
  |   1  2  3
--+-----------
1 |   2  1  1
2 |   1  2  2
3 |   2  2  2
```

Le joueur gagnant est le joueur 2.

3.2 Limites du projet

Ce jeu de plateau est destiné à être joué avec plusieurs individus. Néanmoins, lorsqu'on décide d'y jouer avec de nombreux joueurs, il se peut que le nombre d'appels récursifs lors de la phase de traitement peut très vite monter. En effet, 1 appel provoque au plus 4 autres appels, et chacun de ses 4 appels peut entraîner

3. Justification de la terminaison pour la première boucle while

4 autres appels maximum chacun. On a donc une complexité exponentielle en fonction du nombre de cases.

L'intégrité du jeu est donc garantie pour des valeurs de j et n raisonnablement petites.

3.3 Conclusion générale

Afin de conclure cette UE, ce projet m'a permis tout d'abord de mettre en application la plupart des compétences acquises durant ce semestre. J'ai pu une fois de plus implémenter des structures de données clées telles que les piles et les tableaux 2D dynamiques (même les files même si elles ne m'ont pas servi au final).

Ce projet a également été l'occasion de comprendre encore plus la puissance de la programmation impérative en C, et plus particulièrement via l'importance d'une bonne gestion de la mémoire de l'ordinateur.

Enfin, le fait d'avoir une échéance comme celle-ci m'a permis d'organiser des sessions de travail propres à ce projet, de parfaire ma compréhension de sujets comme les pointeurs et de perfectionner ma rédaction en LaTeX.

4 Annexes techniques

Il s'agit du logigramme du projet.

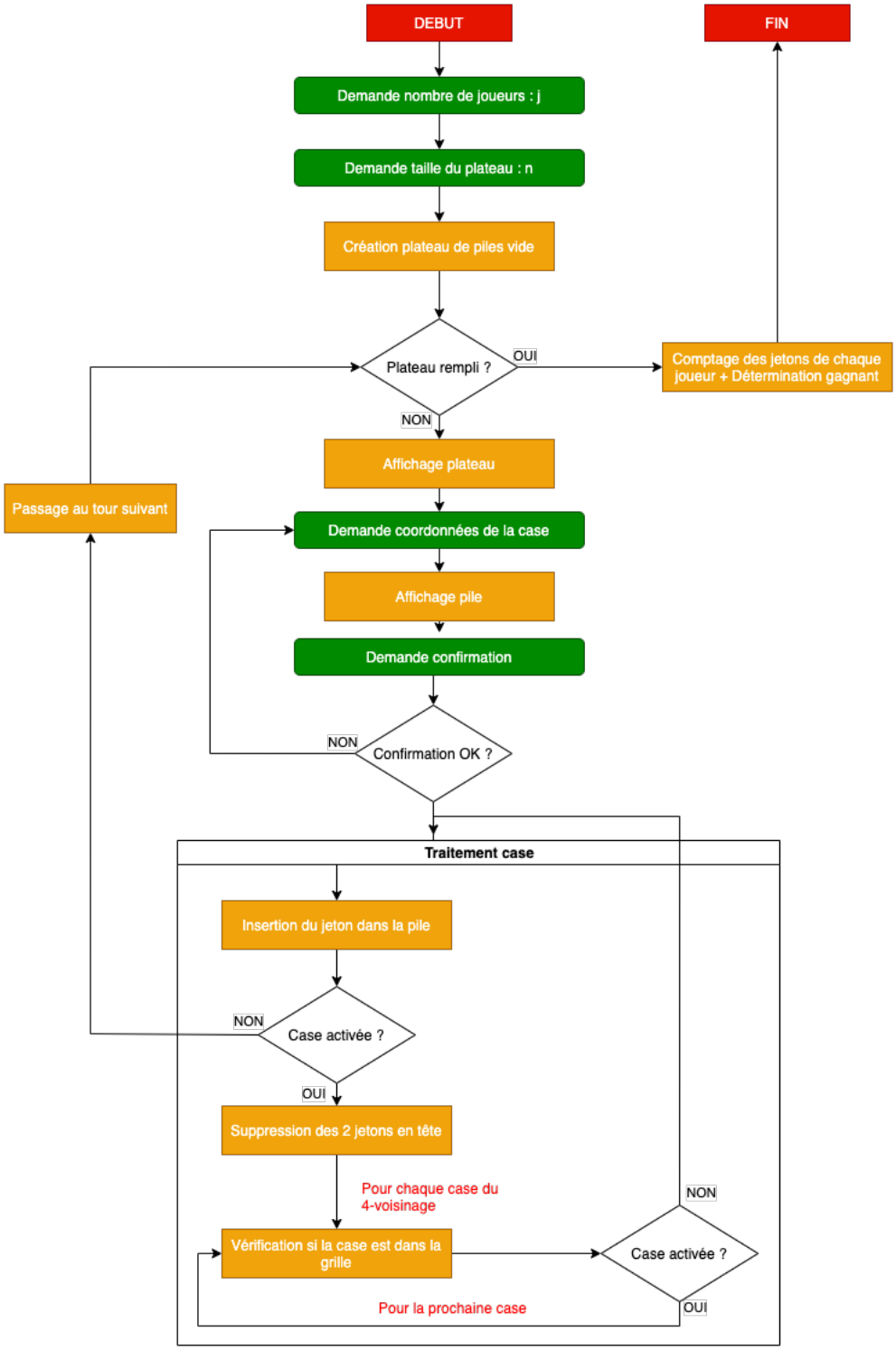

FIGURE 2 – Logigramme