Rapport du projet d'IPI 2020

- étudiant de l'ENSIIE

Jeudi 24 décembre 2020

1 Fonctionnement général du jeu

1.1 Contexte

Le but de ce projet est d'implémenter une plate-forme de jeu de plateau.
On place, dans un plateau de taille $taille^2$, des piles de jetons de couleur, chaque joueur possédant une couleur différente. Initialement, le plateau est vide.
Pour des raisons plus pratiques, on choisit de représenter les couleurs par des nombres représentant la couleur de chaque joueur.

Le jeu se joue à un nombre quelconque de joueurs qui jouent chacun à tour de rôle. Le jeu consiste à avoir le plus de jetons possibles de sa couleur en haut des piles une fois que le plateau est complètement rempli.

Chaque joueur joue à tour de rôle. À chacun de ses tours, le joueur pose un jeton de sa couleur sur une des cases du plateau, en l'empilant sur les jetons déjà présents le cas échéant.
Si les deux jetons au sommet de la pile sont de la même couleur, la case est activée :

- les deux jetons sont retirés de la pile;

- des jetons de la couleur du joueur sont ajoutés sur les quatre cases voisines, à condition qu'elles n'aient pas déjà été activées à ce tour du joueur;

- les jetons ajoutés sur les cases voisines peuvent activer celles-ci, si elles avaient aussi un jeton de cette couleur en haut. Des jetons sont alors ajoutés sur les cases de leurs voisines qui n'ont pas déjà été activées ce tour, et ainsi de suite;

1.2 Les principales fonctionnalités

Au début de la partie, on demande le nombre de joueurs *nbre*, puis la taille du plateau *taille*. Chaque joueur est identifié par un entier entre 1 et *nbre* qui est sa couleur.

À chaque tour du joueur i:

- on affiche le plateau vu du dessus;

- on demande au joueur d'entrer une coordonnée de la case;

- on affiche la pile à la case correspondante;

- on demande confirmation au joueur pour déposer un jeton sur cette case. Si la réponse commence par n, on recommence à afficher le plateau, à demander une coordonnée, à afficher le contenu de la case et à demander la confirmation. Si la réponse commence par o, on place le jeton et on active les cases si appropriées;

Si le plateau est rempli, c'est-à-dire qu'il y a au moins un jeton sur chaque case, alors on compte le nombre de jetons au sommet des cases pour chaque joueur et c'est celui qui en a le plus qui gagne.

2 Démarche générale

2.1 Les structures de données

2.1.1 Les piles

La pile est la structure la plus importante du jeu, puisque'elle constitue les cases du plateau.
Pour l'implémenter, j'avais le choix entre une structure de tableau de taille fixe et une structure de liste chaînée.

J'ai choisi de la représenter sous la forme d'une liste chaînée dont chaque maillon contient une pièce du jeu sous la forme d'un entier correspondant au numéro du joueur. Il s'agit du choix le plus judicieux en raison de sa représentation: le dernier élément qui a été ajouté est le premier à sortir.

```
typedef int value;

struct stack {
    value v;
    struct stack* next;
};

typedef struct stack* stack;
```

L'implémentation sur les piles sont les opérations basiques de :

- créer une pile vide;

- savoir si une pile est vide;

- dépiler le sommet d'une pile;

- empiler un nouvelle élément au dessus de la pile;

- afficher le contenu de la pile;

Ce qui se rapporte aux piles est conciliée dans les fichiers *stack.c* et *stack.h*.

2.1.2 Le plateau

J'ai choisi de représenter le plateau par un tableau 1D, alloué dynamiquement de taille $taille^2$, de piles. Cette représentation est assez défavorable pour plusieurs raisons dont :

- une case est plus difficile à identifier dans un tableau 1D dans lequel la coordonnée vaut :

$$r = taille * (i - 1) + j - 1$$

- la majeure partie des fonctions, procédures nécessiteront plus d'arguments;

En fait, j'ai décidé de limiter au maximum les allocations dynamiques lors de ce projet par souci de créer, probablement, des fuites de mémoire. Effectivement, pour construire une matrice de piles, rappelons qu'il faut d'abord créer un tableau, alloué dynamiquement de taille *taille*, puis parcourir, à l'aide d'une boucle for, cette liste et allouer dynamiquement à chaque composante un tableau de taille *taille* de pile.

La création du tableau se fait dans le *main.c*.

Les différentes opération implémentées sur le plateau sont :

- afficher le plateau vu du dessus;

- modifier le plateau suite au choix du joueur lors de son tour;

- déterminer si la partie est terminée;

- afficher le score si la partie est terminée;

Ce qui se rapporte au plateau est conciliée dans les fichiers *rules.c* et *rules.h*.

2.2 Interaction entre l'utilisateur et le jeu

2.2.1 La fonction qui fait le lien entre l'utilisateur et le jeu

Pour la récupération d'une entrée de l'utilisateur, plutôt que faire un *scanf* directement, comme recommandé, j'ai utilisé *fgets*, avec lequel on récupère une ligne en entier, et j'ai utilisé *sscanf* dessus.
Ceci est, en particulier, utile lors de l'entrée de la coordonnée de la case dans laquelle un joueur veut empiler son jeton.

2.2.2 Les problèmes de terminaison

La fin de partie dépend entièrement des joueurs comme on peut le voir dans certaines des boucles implémentées dans le *main.c*.

D'une part, il suffit qu'un joueur hésite indéfiniment sur la case dans laquelle il veut placer le jeton pour que la partie ne se termine jamais (cf $main.c - ligne45$), autrement dit qu'il entre constamment une phrase ne commençant pas par la lettre *o* lors de la confirmation.

```
c='n';
    while (c!='o') {
        /* Demande la case choisie par le joueur numero n */
        printf("Quelle case choisissez vous ?\n");
        fgets(buf, 256, stdin);
        sscanf(buf,"%i %i",&i,&j);
        printf("\n");

        /* Affiche le contenu de la case */
        printf("Contenu de la case:\n\n");
        r=taille*(i-1)+j-1;
        show(s[r]);

        /* Demande confirmation du joueur n pour placer le jeton dans la case inseree */
        printf("Voules vous mettre les jeton ici ?\n");
        fgets(buf, 256, stdin);
        sscanf(buf, "%1c",&c);
        printf("\n");
    }
```

D'autre part, (cf $main.c - ligne36$), si les joueurs jouent mal, alors, la partie peut ne pas avoir de fin. Par exemple, il suffit aux joueurs d'empiler leur jeton dans la même case pour que ça ne s'arrête jamais.

Je prends donc en considération que les joueurs sont sérieux et ingénieux. Dans ce cas, il y a terminaison de la première boucle (en supposant qu'il y ait seulement une voire deux à trois itérations) et de la seconde boucle (en supposant qu'il y ait peu d'empilements et d'activations superflues, $taille^n - k + 1$ est un variant de boucle avec n de l'ordre de $taille * 2$ et k le nombre de jetons dans le plateau)

2.3 Les opérations implémentées sur le plateau

2.3.1 La fonction d'affichage

Pour faire interagir au mieux l'utilisateur et le jeu, il est nécessaire d'afficher à chaque tour le plateau. J'ai procédé de la manière suivante : J'affiche, d'abord, la première ligne du plateau. Puis, je parcours le tableau 1D pour afficher le haut de chaque pile.
$stack * s$ est le tableau 1D représentant le plateau et t l'entier représentant la taille du plateau.

```
void shown (stack* s, int t) {

    /* Affichage de la premiere ligne du plateau, ie des numeros de colonnes */
    printf("   |");
    int i,k;
    for (i=1;i<=t;i+=1)
        printf("%3i",i);
    printf("\n");

    printf("---+");
    for (i=1;i<=t;i+=1)
        printf("---");
    printf("\n");

    /* Afficahge du reste */
    printf("  1|");
    i=1;
    k=0;
    while (k<t*t) {
    /* t*t-k+1 est un variant de boucle, donc la boucle se termine */
        if (is_empty_stack(s[k])) printf("   ");
        else printf("%3i",(s[k])->v);
        if (k+1==t*t)
            printf("\n\n");
        else if ((k+1)%t==0) {
            i+=1;
            printf("\n   |\n%3i|",i);
        }
        k+=1;
    }
    printf("\n\n");
}
```

2.3.2 L'ajout d'un jeton

Pour l'ajout d'un jeton, j'ai réalisé une fonction récursive.
La fonction *active* ajoute le pion à la case voulue. Si la pile possède déjà un pion du joueur du tour, alors la case est activée : dans ce cas, à la place d'ajouter le pion à la case, on enlève le pion du joueur de la case. Puis, en fonction des coordonnées de la case, on ajoute des pions du joueur du tour chez les voisins de la case, pour cela on appelle récursivement *active*.
Ce qui garanti la terminaison est l'utilisation de la liste l :
- si $l[r] == 1$ alors la case, ayant déjà été activé à ce tour, n'est pas activée;

- dans le cas contraire, il suffit soit d'ajouter le pion à la case correspondante, soit de l'activer si les conditions sont remplies;

```
stack* active(stack* s, int i, int j, int n, int t, int* l) {

    int r=t*(i-1)+j-1;

    /* 1er cas: la case a deja ete active; dans ce cas, ne rien faire */

    if (l[r]==1)
        return s;

    /* 2eme cas: la case est vide, on ajoute simplement le jeton*/

    else if (is_empty_stack(s[r]) && l[r]==0)
        push(n,&(s[r]));

    /* 3eme cas: la case est non-vide mais la couleur du dernier jeton differe de celle du
    /* On ajoute simplement le jeton */

    else if ((s[r])->v!=n && l[r]==0)
        push(n,&(s[r]));

    /* 4eme et dernier cas: sinon, la case est activee */
    /* Des dispositions sont prises en fonction de la case activee */

    else {

        /* On fait savoir que la case a ete activee    ce tour */
        l[r]=1;
        pop(&(s[r]));
```

```
        /* si la case activee est de coordonnees 1 1 */
        if (i==1 && j==1) {
            s=active(s,i+1,j,n,t,l);
            s=active(s,i,j+1,n,t,l);
        }
        /* si la case activee est de coordonnees 1 t */
        else if (i==1 && j==t) {
            s=active(s,i,j-1,n,t,l);
            s=active(s,i+1,j,n,t,l);
        }
        /* si la case activee est de coordonnees t 1 */
        else if (i==t && j==1) {
            s=active(s,i-1,j,n,t,l);
            s=active(s,i,j+1,n,t,l);
        }
        /* si la case activee est de coordonnees t t */
        else if (i==t && j==t) {
            s=active(s,i,j-1,n,t,l);
            s=active(s,i-1,j,n,t,l);
        }
        /* si la case activee appartient a la premiere ligne mis a part les deux extremites
        else if (i==1) {
            s=active(s,i,j-1,n,t,l);
            s=active(s,i+1,j,n,t,l);
            s=active(s,i,j+1,n,t,l);
        }
        /* si la case activee appartient a la derniere ligne mis a part les deux extremites
        else if (i==t) {
            s=active(s,i,j-1,n,t,l);
            s=active(s,i-1,j,n,t,l);
            s=active(s,i,j+1,n,t,l);
        }
        /* si la case activee appartient a la premiere colonne mis a part les deux extremi
        else if (j==1) {
            s=active(s,i-1,j,n,t,l);
            s=active(s,i,j+1,n,t,l);
            s=active(s,i+1,j,n,t,l);
        }
        /* si la case activee appartient a la derniere colonne mis a part les deux extremi
        else if (j==1) {
            s=active(s,i-1,j,n,t,l);
            s=active(s,i,j-1,n,t,l);
            s=active(s,i+1,j,n,t,l);
        }
        /* si la case ne respecte aucun des criteres precedents */
        else {
            s=active(s,i,j-1,n,t,l);
            s=active(s,i-1,j,n,t,l);
            s=active(s,i+1,j,n,t,l);
            s=active(s,i,j+1,n,t,l);
        }
    }
    return s;
}
```

2.3.3 Le test de fin de partie

A chaque tour, *nofinish* parcourt le tableau 1D de piles et cherche l'existence d'une pile vide. Si c'est le cas, il retourne 1 et dans le cas contraire 0.
La valeur obtenue constitue, en fait, un booléen (cf $main.c - ligne36$).

Si la partie se termine, alors il reste à afficher le score de chaque joueur. Pour cela, il suffit de parcourir le tableau 1D est de compter le nombre de jeton au-dessus de chaque pile de chaque joueur.

```
int nofinish(stack* s, int t) {
    int k=0;
    while (!(is_empty_stack(s[k])) && k<t*t)

    /* t*t-k+1 est un variant de boucle, donc la boucle se termine */

        k+=1;
    return (k<t*t);
}

int* score(stack* s, int* l, int n, int t) {
    int i,j;
    for (i=0; i<t*t; i+=1) {
        for (j=1; j<=n; j+=1) {
            if ((s[i])->v==j)
                l[j-1]+=1;
        }
    }
    return l;
}
```

3 Test du projet

Le projet semble marcher correctement : il y a compilation et lors de l'exécution, tout se passe comme prévu. D'ailleurs, voici le score qui est retourné après avoir lu les commandes sur l'entrée standard du fichier partie.txt (en entrant *./prog < partie.txt*) :

Score du joueur 1 : 3
Score du joueur 2 : 6

4 Les difficultés rencontrées

4.1 La fonction récursive *active*

La fonction qui m'a posé le plus de problème est certainement *active*.
En effet, si les quatre cas traités sont disposés dans le mauvais ordre, alors, il peut y avoir des erreurs de segmentations. Par exemple, il faut placer $(is - empty - stack(s[r])\ ET\ l[r] == 0)$
avant $((s[r]) - > v! = n\ ET\ l[r] == 0)$. De même, mieux vaut placer $(l[r] == 1)$ en premier.

De plus, lors de l'activation d'une case, il ne faut pas oublier de traiter les différents cas, car toutes les cases n'ont pas le même type de voisins (d'après ma disjonction de cas, je dénombres 9 cas différents).

4.2 Idées d'optimisation

D'un côté, il est possible d'alléger visuellement le code en créant plus de fonctions, procédures et de fichiers (notamment pour la fonction *active*).

De l'autre, il est également possible d'améliorer l'interaction entre l'utilisateur et le jeu.
Je n'ai, par exemple, pas pensé à créer une commande pour quitter le jeu (si les joueurs n'aboutissent pas ou se trompent en saisissant le nombre de joueurs ou la taille du plateau).
Il était, d'ailleurs, possible de régler les problèmes de terminaison cités précédemment (cf 2.2.2) :

- pour la première boucle mentionnée, il suffit de considérer un compteur à chaque tour et lorsque celui-ci, lors d'un tour, dépasse une certaine limite, on force l'arrêt du jeu;

- pour la seconde boucle mentionnée, on considère de même un compteur associé à chaque case (tous initialement à 0) : quand un joueur décide de placer son jeton dans une case, on incrémente le compteur de +1 et ainsi de suite. Donc, lorsque un des compteurs dépasse une certaine limite, on affiche un message d'erreur et on force l'arrêt du jeu;

5 Manuel d'utilisation

Pour les règles du jeu, référez vous à *Fonctionnement général du jeu.*

Pour lancer le programme, vous aurez besoin d'avoir *gcc* et *make* sur votre ordinateur (si vous ne les avez pas, sous Linux, entrez *sudo apt install gcc* et *sudo apt install make*).
Pour lancer le jeu, placer vous dans le répertoire où se trouve les fichiers associés au projet (ie. *stack.c*, *stack.h*, *rules.c*, *rules.h*, *main.c* et $Makefile$).
Attention, aucun autre $Makefile$ ne devra figurer dans ce répertoire.

Entrez *make* dans le terminal.
Puis entrez *./prog* dans le terminal.

Entrez un entier lorsqu'on vous demande le nombre de joueurs.
Entrez un entier lorsqu'on vous demande la taille du plateau (dans la mesure du raisonnable/ il est recommandé de commancer par une petite taille si c'est la première fois).

La partie commence.
Chaque tour sera identique :

- Entrez la coordonnée de la case sur laquelle vous souhaitez poser votre jeton (Exemple: pour la case à la ligne 1 et à la colonne 1, j'entre 1 1);

- Entrez *oui* ou *o* (en fait, n'importe quel mot commençant par la lettre *o*) si vous confirmez. Dans le cas contraire, entrez *non* ou *n* (en fait, n'importe quel mot ne commençant pas par la lettre *o*) et les questions seront redemandées au joueur du même tour;

Lorsque la partie est terminée, le score de chaque joueur est affiché.
Si vous souhaitez recommencer, il suffit de reprendre à partir du 3ème paragraphe.

Bonne(s) partie(s) !