Rapport Projet Programmation impérative

Barnabé Geffroy

4 Janvier 2021

Table des matières

Pour vous renseigner sur le déroulement d'une partie du jeu, référez-vous au document README.adoc ou à la page du sujet du projet : `http://web4.ensiie.fr/~guillaume.burel/cours/IPI/projet_2020.html`.

1 Structure du code

Le programme du jeu est composé de trois interfaces, `stack_token.h`, `stack_position.h` et `board.h`, et de quatre modules, `stack_token.c`, `stack_position.c`, `board.c` et `game.c`. Voici leurs relations de dépendance suivi d'un descriptif de ceux-ci :

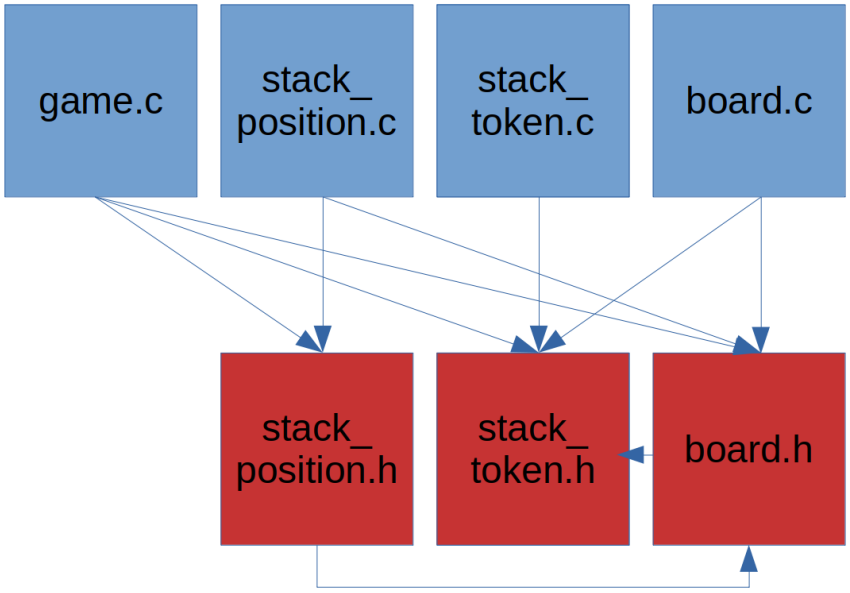


FIGURE 1: Dépendances des modules (bleus) et des interfaces (rouges).

1.1 Inteface et module `stack_token`

`stack_token` représente les cases du plateau par des piles de jetons. En effet, sur les cases du plateau on doit empiler des jetons et facilement retirer le dernier posé. L'implémentation d'une pile, munis des fonctions de base, est donc bien adapté. Les éléments de la pile sont alors les jetons du jeu.

Les jetons sont représentés par le numéro du joueur. Dans l'interface `stack_token.h`, on définit comme des entiers (`int`) `token_value` le type concret des éléments de la pile. L'interface contient également le type abstrait des piles et ces fonctions et procédures abstraites :

- `empty_stack_token` : création d'une pile de jeton vide
- `is_empty_stack_token` : test de vacuité
- `push` : ajout d'un jeton en haut de la pile
- `pop` : retrait du jeton en haut de la pile et donne sa valeur
- `get_head` : renvoie la valeur du jeton en haut de la pile
- `print_stack_token` : affichage de la pile
- `free_stack_token` : libération de l'espace mémoire alloué pour les éléments de la pile

Le module `stack_token.c` dépend de l'interface `stack_token.h` et implémente les piles sous forme de listes chaînées.

1.2 Interface et module `board`

`board` représente le plateau de jeu. Chaque case du plateau est une pile de jetons (`stack_token`), il dépend donc de l'interface `stack_token.h`. C'est un plateau carré type échiquier. Sa taille dépend de la valeur entrée en début de jeu.

Pour réaliser ce plateau l'interface `board.h` propose un type `board` composé d'un tableau 2D de `stack_position` et d'une taille. Afin de localiser les pièces du plateau, un type `position` est créé. Celui-ci localise les cases du plateau de 1 à la taille du plateau pour les lignes et les colonnes. Par exemple, la case en haut à gauche est localisé (1, 1), tandis que son indice dans le tableau 2D est (0, 0).

Le module `board.c` dépend de l'interface `board.h`. Voici les fonctions et procédures de l'interface qui sont implémentées dans le module :

- `init_position` : création d'un type `position` initialisé
- `create_board` : création d'un tableau vide d'une certaine taille
- `is_placable` : test si la `position` est une case du plateau
- `is_full` : test de plénitude du plateau (aucune case vide)
- `is_empty_board` : test de vacuité du plateau (toutes les cases vides). Note : cette fonction n'est pas utilisée pendant le jeu mais peut être utile pour une autre utilisation du plateau.
- `add_token` : ajout d'un jeton sur le plateau
- `remove_token` : retrait d'un jeton du plateau
- `most_present_elem` : comptage des points (nombre de jetons aux sommets) des joueurs et retourne leurs scores
- `on_top` : donne l'élément au sommet de la pile d'une case du plateau
- `print_board` : affichage du plateau avec les jetons aux sommets des pile des cases du plateau
- `free_board` : libération de l'espace mémoire alloué pour le plateau

1.3 Interface et module `stack_position`

`stack_position` est une pile d'éléments de type `position`. Ces piles seront utilisées dans le module `game.c` pour placer les jetons. L'interface et le module sont similaires, à quelques détails près, à ceux de `stack_token.h` et `stack_token.c`. Les éléments d'une pile sont cette fois de type `position`. La fonction `get_head` et la procédure `print_stack_token` n'ont pas d'utilité pour ces piles et ne sont pas implémentées. `free_stack_position` est implémentée mais celle-ci n'est pas utilisée dans le projet.

1.4 Module `game`

`game` du fichier `game.c` est le module contenant la fonction main. Il utilise les trois interfaces précédentes pour dérouler le jeu selon les règles. Il est composé et de la fonction `main`, d'une autre fonction et de deux procédures :

- `ask_player` : cette fonction demande au joueur d'entrer la position de la case du plateau qu'il souhaite jouer. Elle renvoie une `position` utilisée dans la procédure `play` (voir ci-dessous).
- `place_token` : cette procédure place un jeton sur le plateau et si nécessaire déroule la logique d'activation des cases sur le plateau. La case jouée peut s'activer et séquentiellement activer les cases adjacentes qui peuvent elles-mêmes activer leurs cases adjacentes encore non activées. Il y a en quelque sorte un niveau d'activation. Celui-ci est traité dans la procédure par une boucle `while` et une `stack_position`. Chaque tour de boucle va traiter un niveau d'activation et empiler les nouvelles cases à activer dans une `stack_position` pour le prochain niveau. Cela permet de s'assurer que les cases s'activent dans l'ordre de la règle du jeu pour éviter un mauvais traitement du placement des jetons.

— `play` : à l'aide de `ask_player` et `place_token`, cette procédure fait jouer à tour de rôle les joueurs (nombre de joueurs en argument) sur un plateau (initialement vide et de taille passée en argument) jusqu'à ce que le plateau soit rempli (aucune case vide).
— `main` : demande à l'utilisateur le nombre de joueurs et la taille du plateau et lance la procédure `play` en fonction des valeurs saisies.

1.5 Makefile

Ce fichier gère les dépendances présentées dans la figure 1 pour compiler sans incident le projet.

1.6 partie.txt et partie2.txt

Ces deux fichiers permettent de tester l'exécutable `prog` à l'aide des commandes :

```
./prog < partie.txt
./prog < partie2.txt
```

2 Développement du projet

2.1 Implémentation initiale

Pour réaliser ce jeu, j'ai procédé en plusieurs étapes. Premièrement, j'ai analysé attentivement le fonctionnement du jeu de manière à appréhender les spécificités du code à implémenter. Dans un premier temps, j'ai fait le choix de séparer le projet en plusieurs interfaces et modules, je n'avais toutefois pas encore pensé à `stack_position`.

Le fait de séparer le plateau, les piles, et le jeu amène une modularité et une grande lisibilité du projet. Cela permet également de pouvoir réutiliser une partie du code pour un autre projet (le plateau pour un jeu d'échecs par exemple). Je me suis donc assuré, dans un premier temps, que l'implémentation des piles satisfaisait toutes les contraintes nécessaires pour le jeu. J'ai fait le choix d'utiliser les listes chaînées car la mémoire allouée dépend exclusivement du nombre d'éléments de la pile. Étant donné le nombre conséquent de piles, et le nombre très variable de jetons dans ces piles, cette donnée m'a paru importante.

En m'inspirant du TP sur les matrices, j'ai pu dans un second temps, développer l'interface et le module du plateau. Les éléments physiques du jeu (plateau et pions) étant fonctionnels, j'ai implémenté le module `game.c` dont le but est de faire jouer les joueurs sur le plateau suivant les règles imposées.

2.2 Le module `game`

Le module de `game.c` a été la partie la plus technique à implémenter. J'ai préféré séparer le déroulement du jeu dans des fonctions et procédures pour gagner en clarté par rapport à une simple fonction `main`. Ainsi, j'ai débuté par implémenter la fonction demandant quelle case le joueur souhaite jouer. J'ai ensuite codé une procédure simplifiée de la pose du jeton. Celle-ci ne prenait en compte que le niveau d'activation initial (les cases voisines à la case choisie étaient pourvues de jetons). Cela m'a permis de vérifier si la procédure `play` fonctionnait correctement. J'ai réglé les petites coquilles du code de sorte à ce que le jeu se déroule comme dans l'énoncé, à l'exception de l'activation des cases.

2.3 `place_token`, récursion 1

Une fois que tout était opérationnel, j'ai commencé à implémenter la bonne version de `place_token` de manière récursive (voir code 1 de l'annexe). Celle-ci utilise un plateau (`act`), initialement vide, pour référencer les cases activées de sorte à ne pas reposer des jetons dessus.

Cette procédure fonctionne pour la partie test `partie.txt`. Néanmoins, cette partie ne traite pas un cas particulier du jeu.

En effet, dans `partie.txt` aucune des cases s'activant en même temps ne doivent placer un jeton sur la même case adjacente. Par exemple, considérons le plateau ci-dessus (figure 2) avec un seul jeton sur chaque case non vide :

FIGURE 2: Plateau de la situation étudiée, avant le tour du joueur 1

```
 |1 2 3 4
-+---------
1|2 2 2 2
2|  1 1
3|  1 1
4|
```

Le joueur 1 décide de placer son jeton sur la case (3,2) qui s'active. Les cases (2,2) et (3,3) vont s'activer et vont à leur tour déposer un jeton sur la case (2,3). Avant que cette dernière ne s'active, il y a trois jetons sur cette case (un déjà posé, et deux posés par l'activation des cases). Ainsi, après activation il reste un jeton du joueur 1 sur cette case. Les autres jetons posés et cases activées ne sont pas essentiels pour la compréhension du problème. L'algorithme du jeu devrait alors renvoyer à la fin du tour du joueur 1 le plateau suivant :

FIGURE 3: Plateau de la situation étudiée, après le tour du joueur 1 (situation souhaitée)

```
 |1 2 3 4
-+---------
1|2 1 1 2
2|1   1 1
3|1     1
4|  1 1
```

Mais la case (2,3) est vide avec mon implémentation (code 1). La procédure ne distingue pas le cas où plusieurs cases s'activent en même temps. Elle traite les cases à activer les unes après les autres dans un ordre arbitraire (ligne 26 du code 1). Une fois qu'une case est activée, aucun autre jeton ne peut être placé dessus. Aucun niveau d'activation n'est pris en compte.

Le premier niveau d'activation est la case choisie par le joueur, le second niveau est les cases activées par la première et ainsi de suite. Dans l'exemple le premier niveau est la case (3,2), le second les cases (2,2) et (3,3), etc. Il n'est pas possible d'effectuer une pose des jetons de manière simultanée. Toutefois lors de l'activation des cases, celles-ci ne doivent pas prendre en compte les cases activées lors du même niveau, de sorte qu'elle puisse tout de même distribuer des jetons sur des cases adjacentes où deux jetons sont déjà présents (dans l'exemple, la case (2,3)).

2.4 `place_token`, récursion 2

Je modifie alors la procédure (voir code 2) pour prendre en compte le niveau d'activation. Pour cela un compteur permet de référencer les niveaux d'activation. La procédure permet alors de distribuer un jeton sur une case activée si le niveau d'activation est le même. Mais les appels récursifs de la procédure font en quelque sorte un parcours en profondeur du graphe des cases activables. La case (2,2) est associée au niveau d'activation 4 car l'algorithme associe à (3,2) le niveau 1 puis à (3,3) le 2, puis à (2,3) le 3 et à (2,2) le 4 alors qu'il doit s'activer au niveau 2.

Pour mon implémentation, la récursion n'est donc pas adaptée pour traiter le problème. `place_token` doit parcourir en largeur le graphe des cases activables et ainsi procéder de manière itérative.

2.5 `place_token`, version finale

J'ai donc modifié la procédure (voir code 3) et obtenu une procédure réalisant correctement le placement des jetons.

Pour cela, mon idée était que chaque niveau d'activation soit une pile de `position` des cases à traiter. J'ai donc ajouté l'interface et le module `stack_position`.

J'ai ensuite implémenté l'interface d'une file de piles de `position` de sorte que les cases activées ajouteraient leurs cases adjacentes à une pile qui serait enfilée à la file. Les piles seront être traitées suivant l'ordre d'activation (les premières cases activées en premier, les dernières en dernier). Pour traiter un niveau d'activation, il suffit alors de défiler la file et de traiter tous les éléments de la pile au même niveau d'activation.

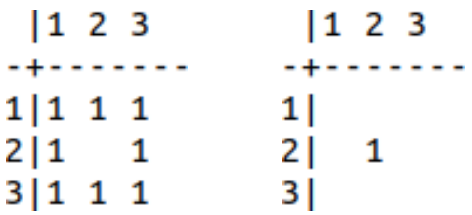
Cependant, en exécutant mon code je me rends compte que seulement deux piles sont présentes dans la file, celle du niveau d'activation courant et celle du prochain niveau d'activation. Cette file de `stack_position` n'est donc pas très pertinente et alourdit mon code.

Je n'utilise alors que deux piles de `position` (implémentées dans `stack_position`), le niveau d'activation courant (`cur_act_lvl`) et le suivant (`next_act_lvl`). Le corps de la procédure fonctionne alors avec deux boucles `while` (voir code 3). La première boucle (début ligne 22) s'assure que la pile des éléments du prochain niveau d'activation n'est pas vide. Si c'est le cas, les éléments de `next_act_lvl` sont transférés dans `cur_act_lvl` (lignes 26, 27 et 32). Il y a ensuite la seconde boucle (ligne 33 à 73) qui traite les éléments de `cur_act_lvl` sur un même niveau d'activation en ajoutant au plateau des cases activées (`is_activated`), les cases suivant leur niveau d'activation (ligne 53). Ces cases s'activent, on place leurs cases adjacentes dans `next_act_lvl` (ligne 70) et seront traitées lors de la prochaine boucle.

Pour le cas étudié précédemment (voir figures 2 et 3), cette fois la procédure obtient le bon résultat. Il n'y a pas réellement trois jetons sur la case (2,3) mais une fois activée on peut tout de même, lors du même niveau d'activation, placer sur cette case des jetons distribués par les cases adjacentes (ligne 44 il y a la vérification de l'état de la case).

Cette implémentation de `place_token` est celle présente dans mon rendu final. Elle vérifie l'exemple étudié (le fichier `partie2.txt` traite cet exemple) et est moins lourde que l'implémentation d'une file de `stack_position`. Le code vérifie également des exemples plus élaborés :

FIGURE 4: Plateau de taille 3 pour un joueur ; avant et après la pose d'un jeton sur la case (3,2)



2.6 Gestion de l'espace mémoire

Pour conclure, je me suis assuré que tous les espaces de la mémoire utilisés lors de l'exécution du jeu étaient libérés à la fin de leur utilisation. J'avais implémenté `free_stack_position` pour libérer les cases mémoire allouées aux piles de `position`, toutefois `cur_act_lvl` et `next_act_lvl` sont vides à la fin des boucles `while` (voir code 3). J'ai utilisé la commande `valgrind` pour m'assurer que toutes les cases mémoire étaient libérées à la fin de l'exécution du programme.

3 Limites du projet

3.1 Limite de la structure du code

La séparation en interfaces et modules donne une meilleure lisibilité et une modularité. Toutefois, les modules de pile `stack_token` et `stack_position` sont très similaires. Je n'ai pas réussi à trouver une façon de mutualiser les parties identiques dans le but de réduire les doublons.

3.2 Limite de certaines procédures et fonctions

Certaines procédure et fonctions peuvent produire des erreurs si elles sont mal utilisées. Pour ne pas alourdir le code, j'ai pris la décision de ne pas traiter les cas où les arguments donnés ne respectent pas les conditions imposées dans `/* @requires */`. Ce choix impose d'être très rigoureux sur les arguments donnés aux fonctions et procédures. Néanmoins, il m'a permis de produire un code plus aéré donc plus lisible et imposant seulement dans `/* @requires */` les conditions sur les arguments.

Par exemple, la procédure `add_token` du module `board.h` (voir code 4) impose que `p` soit une `position` d'une case de `*b` (ligne 4) mais ne le vérifie pas dans le corps de la procédure.

3.3 Limite de `place_token`

Cette procédure permet d'obtenir la répartition des jetons sur le plateau après chaque tour, toutefois le déroulé de la procédure est différent du déroulé réel. Dans un jeu réel, on pose le jeton sur une case puis si deux jetons appartiennent au même joueur alors la case s'active. Dans la procédure, le code sonde la case pour savoir quel jeton est en haut de la pile. Si c'est le jeton du joueur alors la case s'active et le jeton est retiré de la pile sans poser le second jeton.

Cela n'a aucune incidence sur le résultat final mais il peut entraîner une confusion ou une mauvaise compréhension de la procédure. J'ai fait le choix de ne pas poser un jeton sur un jeton identique pour éviter l'empilement puis le dépilement quasi immédiat du même jeton.

3.4 Limite de l'affichage

L'affichage du plateau est optimisé pour des plateaux strictement plus petits que 10 cases sur une ligne. À partir de 10, l'affichage n'est plus optimal. Mon but était de me rapprocher le plus du modèle de l'énoncé et ainsi reproduire à l'identique les exemple proposés.

Annexe

Code 1: Procédure `place_token` fonctionnant de manière récursive

```
void place_token(token_value token, position p, board *b, board *act)
{
/* if the token on the top of box chosen is different we try to place it */
    if (on_top(*b, p) != token)
    {
/* we place it if the position is not activated
or have been activated during this activation level */
        if (on_top(*act, p) ==-1 )
            add_token(token, p, b);
        return;
    }
    else
    {
/* the box has the same token on the top of the box at p
so the box is activated and the token removed */
        remove_token(p, b);
        if (on_top(*act, p) == -1)
            add_token(1, p, act);
/* initializes the position of the adjacent boxes where we want to place tokens */
        position up = p, down = p, right = p, left = p;
        up.y += 1;
        down.y -= 1;
        right.x += 1;
        left.x -= 1;
/* initalizes a table to work on the adjacent boxes equally */
        position to_add[4] = {up, right, left, down};
        int i;
/* @loop invariant : i is the index of an element of to_add*/
        for (i = 0; i < 4; i += 1)
        {
            if (is_placable(to_add[i], *b) && on_top(*act, to_add[i]) == -1)
                place_token(token, to_add[i], b, act);
        }
    }
}
```

Code 2: Procédure `place_token` prenant en compte le niveau d'activation (non fonctionnelle)

```
void place_token(token_value token, position p, board *b, board *act, int count)
{
    count += 1;
/* if the token on the top of box chosen is different we try to place it */
    if (on_top(*b, p) != token)
    {
/* we place it if the position is not activated
or have been activated during this activation level */
        token_value v = on_top(*act, p);
        if (v ==-1 || v  == count)
            add_token(token, p, b);
        return;
    }
    else
    {
/* the box has the same token on the top of the box at p
so the box is activated and the token removed */
        remove_token(p, b);
        if (on_top(*act, p) == -1)
            add_token(count, p, act);
/* initializes the position of the adjacent boxes where we want to place tokens */
        position up = p, down = p, right = p, left = p;
        up.y += 1;
        down.y -= 1;
        right.x += 1;
        left.x -= 1;
/* initalizes a table to work on the adjacent boxes equally */
        position to_add[4] = {up, right, left, down};
        int i;
/* @loop invariant : i is the index of an element of to_add*/
        for (i = 0; i < 4; i += 1)
        {
            if (is_placable(to_add[i], *b) && on_top(*act, to_add[i]) == -1)
                place_token(token, to_add[i], b, act, count);
        }
    }
}
```

Code 3: Procédure `place_token` fonctionnelle

```c
void place_token(token_value t, position p, board *b)
{
    /* intializes a counter which represents the activation level*/
    int count = 0;
    /*
     * intializes an empty board to note activated boxes
     * to note them, we will add the value of the counter to the box,
     * if we try to place a token on a position,
     * is_activated will tell if this box is activated
     * and if it is, what is its activation level thanks to the counter
     * The first activation level is the position chosen by the player,
     * the second one is the positions actived
     * thanks to the first activated position and so on.
    */
    board is_activated = create_board(b->size);
    /* intializes an empty stack of positions (i.e. stack_position)
       it represents the position for the next activation level,
       so initially just p */
    stack_position next_act_lvl = empty_stack_position();
    push_position(p, &next_act_lvl);
    /* @loop variant : the length of next_act_lvl */
    while (!is_empty_stack_position(next_act_lvl))
    {
        /* initializes a stack_position which will be the positions
        to place during this loop */
        stack_position cur_act_lvl;
        cur_act_lvl = next_act_lvl;
        /* sets next_act_lvl to an empty stack_position,
        it will contain the positions to place for the next loop
        (i.e. next activation level), if any (i.e. activated boxes) */
        next_act_lvl = empty_stack_position();
        /* @loop variant : the length of cur_act_lvl */
        while (!is_empty_stack_position(cur_act_lvl))
        {
            /* pops the head of cur_act_lvl and places its returned value in p */
            p = pop_position(&cur_act_lvl);
            /* if the token on the top of box chosen is different,
              tries to place it */
            if (on_top(*b, p) != t)
            {
                /* places it if the position is not activated
                 or have been activated during this activation level */
                token_value act = on_top(is_activated, p);
                if (act == -1 || act == count)
                    add_token(t, p, b);
            }
```

```
            else
            {
                /* the box has the same token on the top of the box at p
                 so the box is activated and the token removed */
                remove_token(p, b);
                /*adds the activation level to box of the board of activated boxes*/
                add_token(count, p, &is_activated);
                /* initializes the position of the adjacent boxes
                 where we want to place tokens */
                position up = p, down = p, right = p, left = p;
                up.y += 1;
                down.y -= 1;
                right.x += 1;
                left.x -= 1;
                /* initalizes a table to work on the adjacent boxes equally */
                position to_add[4] = {up, right, left, down};
                int i;
                /* @loop invariant : i is the index of an element of to_add*/
                for (i = 0; i < 4; i += 1)
                {
        /* if the position is on the board and its not yet activated, the positions
        is pushed in next_act_lvl and will be placed in the next activation level */
                    if (is_placable(to_add[i], *b)&&on_top(is_activated, to_add[i])==-1)
                        push_position(to_add[i], &next_act_lvl);
                }
            }
        }
        /* increments the counter and if next_act_lvl is not empty,
        runs the next activation level */
        count += 1;
    }
    free_board(&is_activated);/* frees the memory space of the board is_activated */
}
```

Code 4: Procédure `add_token`

```
/*
 * puts a token on the board
 * @args    : v, p, *b
 * @requires: p is a position on *b,
 *            b is pointing on a valid board
 * @assigns : *b
 * @ensures : adds v to the head of the stack of the box at p on *b
 */
void add_token(token_value v, position p, board *b)
{
    /* offset of 1 to place p at the right box of b.tab */
    push(v, &b->tab[p.y - 1][p.x - 1]);
}
```