ENSEIRB-MATMECA

Filière Informatique - 1ère année

Rapport de projet

Tuiles de Wang

Auteurs :
Enzo MEDINA
Benjamin PONTON

Encadrants :
M. David RENAULT
M. Jean-Luc BIENVENU

3 Novembre 2020 — 18 Décembre 2020

Table des matières

1 Introduction

1.1 Objectif du projet

Ce projet, sous forme d'un travail en binôme de 6 semaines, a pour objectif final la réalisation d'un jeu en langage de programmation C. Il vise à nous apprendre à travailler en groupe, à nous organiser, à rendre un travail dans les délais et à développer nos compétences techniques sur des sujets complexes, compétences requises pour un ingénieur. Le projet commence par une version initiale qui est ensuite améliorée à travers plusieurs 'achievements', qui seront expliqués au fur et à mesure dans ce rapport.

1.2 Principe du jeu

La version de base du jeu, Tilings, met en scène plusieurs joueurs qui s'affrontent sur un plateau de jeu. Ils possèdent tous un deck, composé de tuiles de Wang (carrés de 4 triangles colorés). Après que le premier joueur ait déposé une tuile au centre, les joueurs posent ensuite, à tour de rôle, la tuile au sommet de leur deck, en s'assurant que l'ensemble des tuiles soit connexe et que deux côtés de tuiles adjacents soient de même couleur. Si l'un des joueurs ne peut pas poser de tuile, il saute son tour et il range sa tuile au fond de son deck. La partie s'arrête, si lors d'un tour tous les joueurs ont sauté leur tour ou si l'un d'eux n'a plus de tuiles. Le décompte des points est calculé selon différentes méthodes au cours des achievements. Au début, le nombre de points est égal au nombre de tuiles posées.

2 Cadre de travail

2.1 Répartition des tâches

La répartition des tâches dans le binôme s'est faîte naturellement. Nous avons chacun choisi des fichiers pour éviter un maximum de conflits et donc gagner du temps. Enzo MEDINA s'est occupé en majorité de `color.c`, de `file.c` et de `project.c` et Benjamin PONTON de `tile.c` et de l'ensemble des tests en général (`test rule.c`, `test tile.c`, etc). Cependant, pour le fichier `rule.c`, qui contient la majorité des fonctions qui permettent le bon déroulement du jeu, nous avons travaillé ensemble.

2.2 Organisation du travail

Au niveau de l'organisation du travail, nous avions des séances le mardi et le vendredi de 13h50 à 18h10 avec nos encadrants. En travaillant durant ces horaires, nous avons alors pu dresser des comptes rendus pour chacune de ces séances afin d'organiser notre travail :

Séances	Compte Rendu	To do
Séance 03/11	- Découverte des fichiers fournis - Découverte du sujet	- Bien comprendre le sujet
Séance 06/11	- Fichier color.c et couleur.c implémentés	- Mettre en place l'initialisation
Séance 10/11	- Création des files - Début de la boucle de jeu - Début des règles du jeu	- Terminer la boucle de jeu - Finir les règles du jeu
Séance 13/11	- Fin de la boucle de jeu - Implémentations des fonctions en fin de jeu - Mise en place des paramètres "-n" et "-b"	- Tester le programme - Corrections des différents bugs
Séance 17/11	- Jeu fonctionne : * Corrections de divers bugs * Création des structures/fonctions manquantes - Création option -r pour un rendu visuel - Création option -s pour gérer le seed	- Commencer Achiev 1 - Implémenter différents paramètres
Séance 20/11	- Optimisations (Achiev0 fonctionne) - Upload du dossier achiev0 - Début de Achiev1 : * Création de 3 types de motifs * Rework du système de score * Ajout de fonctions liées aux motifs - Corrections des valeurs par défaut	- Faire les tests - Tester Achiev 1 - Commencer Achiev 2
Séance 24/11	- Système d'implémentation correcte des motifs - Meilleure compréhension des motifs - Tests Achiev 1 - Réflexion sur Achiev 2	- Corriger les erreurs Achiev 2 - Upload du dossier Achiev 1
Séance 27/11	- Correction du pointeur de fonction - Upload du dossier Achiev 1 - Création des fonctions majeures de Achiev 2	- Faire vérifier Achiev 1 - Fixer Achiev 2 - Potentiellement valider Achiev 2
Séance 01/12	- Achiev 1 validé - Corrections sur Achiev 2 - Début Achiev 3	- Valider Achiev 2 - Terminer Achiev 3
Séance 04/12	- Corrections des erreurs de Achiev 3 - Fin des fonctions king, update,.. de Achiev 3 - Début des tests Achiev 3 - Début de la rédaction du rapport	- Etablir un plan du rapport - Examen des dernières fonctionnalités - Correction du Makefile
Séance 08/12	- Corrections des erreurs de Achiev 3 - Tests complets de Achiev 3 - Ajout de couleurs pour les tests - Comptage des erreurs - Fixer les erreurs de paramètres - Correction du Makefile - Rédaction de Achiev 0	- Ecriture des commentaires - Avancer le rapport
Séance 11/12	- Redaction de Achiev1 et de Achiev2 - Amélioration des paramètres avec "-m" - Mise en place d'un meilleur rendu en cas d'erreurs	- Écriture du rapport - Corrections mineures Achiev 3
Séance 15/12	- Rapport presque terminé - Corrections de Achiev 3	- Finir le rapport - Écriture de commentaires - Vérificateur d'orthographe
Séance 18/12	- Finalisation du rapport - Ajouts de commentaires - Corrections bugs mineurs - Ajouts d'images de notre programme	RENDU DU PROJET

2.3 Outils

2.3.1 Git

Nous avons utilisé Git, un logiciel de gestion de code source, qui permet de cloner et de modifier facilement une version du projet. Nous avons stocké l'ensemble du projet et les différents tests sur "La Forge" du serveur "Thor Manager Project". De cette façon, il a été plus simple pour nous de récupérer le projet et de travailler chacun de notre côté, même à distance pendant le confinement. La Forge possède également ses propres tests et un historique détaillé des modifications de chacun.

De plus, Git permet aux participants d'un projet de travailler en même temps sur celui-ci, à condition d'être sur des fichiers différents ou sur des parties différentes d'un même fichier, sans que cela n'entraîne de conflits. Ainsi, nous avons pu nous répartir différentes tâches et travailler indépendamment sur des ajouts sans corrompre le code de l'autre : par exemple Enzo MEDINA définissait les couleurs des tuiles sur le fichier color.c pendant que Benjamin PONTON créait les tuiles dans le fichier tile.c.

2.3.2 MakeFile

Pour centraliser la compilation des différents fichiers, nous avons créé un MakeFile. Il compile tous les fichiers du projet, nous permettant avec la commande make d'obtenir un exécutable : project, qui correspond au jeu. Nous avons pu également, via ce script, utiliser les commandes make test et make clean pour réciproquement compiler les fichiers de tests et lancer l'executable test ou supprimer les fichiers compilés .o.

2.3.3 Valgrind

Valgrind est un outil de programmation pour débogger les projets. Il nous a été utile de nombreuses fois pour résoudre les problèmes de *Segmentation Fault* et nous a permis de corriger des fuites de mémoires importantes, erreurs, qui ne sont pas détectables par nos tests de part leur nature.

3 Mise en place de la version initiale

3.1 Problématiques

Nous avons débutons le projet avec 4 fichiers de base, color.[ch] et tile.[ch]. La première étape a consisté à les compléter pour permettre l'implémentation des structures tuiles et leurs couleurs associées. La deuxième étape a consisté à implémenter les joueurs dans une file d'attente pour leur permettre de jouer à tour de rôle. Enfin, nous avons créé la boucle de jeu et avons permis aux joueurs de poser leurs tuiles sur le plateau en s'assurant qu'ils respectent les règles. Ces étapes ont défini la version de base du projet. En parallèle, nous avons du créer un Makefile pour permettre la compilation des différents exécutables et de permettre l'ajout de paramètres supplémentaires pour gérer les parties de jeu (nombre de joueurs, taille du plateau, la graine aléatoire, etc).

Plusieurs problématiques et contraintes sont alors apparues :

Premièrement, nous avons du nous assurer de poser des bases solides dès le début du projet pour éviter les problèmes plus tard. Ainsi, nous avons du veiller, à travers tous les fichiers C ainsi que le Makefile, à garantir leur bon fonctionnement, en s'assurant que chaque fichier appelle uniquement ce dont il a besoin.

Deuxièmement, l'implémentation des différentes structures (**Player**, **Board**, **File**, **Tile**) a requis de la rigueur car elles sont dépendantes les unes des autres.

Troisièmement, les fichiers color.h et tile.h ne peuvent être modifiés. Nous avons du réfléchir correctement à l'implémentation du projet en tenant compte de cette contrainte.

3.2 Gestion des couleurs

Pour réaliser la premère étape, nous avons du tout d'abord compléter l'un des fichiers fournis : color.c. À l'intérieur, nous avons défini la structure **Color**, contenant une énumération de couleurs

possibles et avons défini un certain nombre de fonctions de base : récupérer le nom d'une couleur (*color_name*), récupérer un code pour transformer l'affichage du texte dans le terminal (*color_cstring*) et pouvoir récupérer une couleur via son nom (*color_from_name*).

L'énumération est composée de 15 couleurs associées à des nombres de 0 à 14 et d'une couleur vide associée à la valeur 15 pour un total de 16 valeurs possibles. Pour réaliser les fonctions, nous avons eu besoin de deux tableaux de 16 éléments chacun contenant les couleurs, l'un contenant des chaîne de caractères (*couleurs_string*) et l'autre contenant des structures (*couleurs*). Les couleurs dans les deux tableaux sont dans le même ordre que celui de l'énumération. Nous avons également défini une constante, *nb_color*, qui contient le nombre de valeurs possible.

La fonction (*color_name*) est donc implémentée grâce à un switch sur la valeur de l'énumération retournant le nom de la couleur si cette valeur est entre 0 et 15, "ERREUR" sinon.

De la même façon, (*color_cstring*) utilise un switch renvoyant le code couleur nécessaire pour changer la couleur du texte du terminal, sous forme "\e[38;5;code_couleurm" avec *code_couleur* étant une valeur spécifique pour chaque couleur. Par exemple, pour la couleur rouge, *code_couleur* est égal à 9, nous avons donc "\e[38;5;9m". Le numéro 15, correspondant à la couleur vide, renvoie lui aussi "ERREUR".

En ce qui concerne (*color_from_name*), nous effectuons une boucle avec i allant de 0 à (*nb_color* - 1). Dans la boucle, la fonction compare la chaîne de caractères passée en paramètre (*name*) et celle du tableau *couleurs_string* en position i. Si les chaînes sont identiques, alors la fonction retourne le pointeur sur la structure en position i du tableau *couleurs*. Les valeurs dans l'énumération et dans les deux tableaux sont les mêmes puisque les couleurs sont dans le même ordre.

3.3 Gestion des tuiles et de la file

En plus des couleurs, nous avons besoin de tuiles pour réaliser la première étape. Une tuile est une structure *tile* avec une couleur pour chacun des côtés (NORD, SUD, EST, OUEST). Nous avons, dans tile.h, une énumération *direction*, qui contient ces 4 directions associées à 0, 1, 2, ou 3. Nous avons donc décidé que *tile* contiendrait un tableau de taille 4, contenant des pointeurs de *color*. Cela nous a permis de réaliser très simplement *tile_edge*, en prenant la valeur du tableau correspondant à la position.

De la même façon, *tile_equals* profite également de cette implémentation car il suffit de faire une boucle de 0 à 3 et de comparer les valeurs dans les deux tableaux. La fonction *tile_is_empty* se contente d'appeler *tile_equals* avec la tuile vide. Cette tuile vide nous a posé plusieurs problèmes.

Nous avons décidé d'avoir une couleur EMPTY que nous avons associée à la tuile vide. Nous avons ansi évité des pointeurs *null* et nous avons pu utiliser les tuiles sans avoir besoin de tester si les pointeurs le sont. En revanche, retourner le pointeur sur la tuile vide sachant qu'on ne peut pas la créer dans la fonction fut un problème. Nous l'avons résolu en ayant une *empty_t* qui est une *static tile*. La fonction *empty_type* définit ses couleurs comme étant EMPTY avant de retourner son pointeur. Ce choix a fonctionné de notre côté mais a provoqué des erreurs lors du changement de fichier sur la forge.

En effet, d'autres implémentations utilisant des pointeurs null ne fonctionnaient donc pas. Nous avons du rajouter des tests pour vérifier que les tuiles n'étaient pas vides dans certaines fonctions de rule.c. Enfin, pour pouvoir jouer, nous avons besoin d'un deck de tuiles.

Un **deck** est une structure représentant tous les tuiles de la partie, elle contient un tableau de **deck_pair** nommé *cards* et *size* l'entier correspondant au nombre de **deck_pair** dans le **deck**. Un **deck_pair** est une structure représentant une tuile et son nombre dans le deck, elle contient un pointeur vers la tuile qu'elle représente et un entier n, le nombre de fois que la tuile est présente dans le **deck**.

La fonction *deck_init* remplit un **deck** d donné en paramètre. Pour cela nous avons défini 5 tuiles différentes :

1. Complétement Rouge
2. Complétement Violette
3. Nord et Est : Rouge / Sud et Ouest : Violet

4. Nord et Est : Violet / Sud et Ouest : Rouge
5. Nord et Ouest : Violet / Sud et Est : Rouge

Nous avons décidé de créer 20 exemplaires de chaque tuile dans le deck pour un total de 100 tuiles. Le **deck** d a 5 **deck_pair**, son *size* est mis à 5, pour chacun de ses **deck_pair** leurs n est initialisé à 20 et leur pointeur p est associé à l'une des 5 tuiles ci-dessus, chacun associé à une tuile différente.
Maintenant que l'on a défini les tuiles, il faut les distribuer et les associer à chaque joueur. Elles sont distribuées aléatoirement selon la seed donnée et elles sont stockées dans une file représentant la main d'un joueur. Pour éviter d'utiliser l'allocation de mémoire, nous avons défini la structure *file* comme étant composée de deux entiers et d'un tableau :

— **Debut** incrémenté à chaque fois que l'on sort un élément de la file avec *pop()*,
— **Fin** qui augmente lorsque l'on saute son tour et remplace une tuile avec *push()*
— **Tiles[]** le tableau de pointeur de *tile* de taille *MAX_TILE*. Durant la distribution, la fonction met le pointeur de *tile* en position **Fin** avant de l'incrémenter. Quand le joueur joue, le programme récupère le pointeur en position **Début** avant de l'incrémenter.

Ainsi, nous avons défini une file comme vide si **Début** est égal à **Fin**.

3.4 Gestion de la boucle de jeu

Maintenant que nous avons les outils principaux pour les joueurs à savoir les tuiles colorés et la file contenant les tuiles des joueurs, il est maintenant temps de commencer la partie principale de la mise en place du jeu initial. Nous avons séparé le travail à effectuer en deux fichiers :

Le fichier **Rule.c** contient à la fois les fonctions relatives au respect des règles et les fonctions d'initialisation comme présentées ci-dessous.

— **Fonctions essentielles relatives au respect des règles**

1. *int position_valide(struct board *b, const struct tile * t, int ligne, int colonne)*
   Vérifie, pour un plateau donné, si à un emplacement (l, c) la tuile t est posable ou non.
2. *int authorized_places(struct board * b, const struct tile * t, int positions_validesX[], int positions_validesY[])*
   Retourne le nombre de positions disponibles sur le plateau pour une tuile donnée.
3. *struct position select_position(int nb_pos_valide, int positionsX[],int positionsY[])*
   Sélectionne une place aléatoire parmi l'ensemble des positions valides.
4. *void place_tile(struct board * b, const struct tile * t, struct position p, int num_joueur)*
   Place une tuile à la position choisie par la fonction précédente. C'est ici que le programme définit le propriétaire de la tuile. Cela aura un impact dans les prochains achievments.
5. *int color_equals(struct color * c1, struct color * c2)*
   Vérifie si deux couleurs sont égales en comparant les strings renvoyées par *color_name*. C'est ici que le problème des pointeurs null s'est déclaré.

— **Fonctions relatives à l'initialisation du jeu**

1. *void init_players(int nbPlayers, struct player players [])*
   Récupère un tableau de structure *joueur* et leur attribue à chacun une file vide de tuiles et un score de 0.
2. *struct board board_init(int taille)*
   Construit un plateau en l'initialisant avec des tuiles vides sans propriétaires. C'est-à-dire, de 0 à la taille du plateau, pour les lignes et les colonnes, le programme met dans le plateau le pointeur de la tuile vide et il lui donne la valeur de référence -1 dans *from_player*.

Dans le fichier rule.h, il y a 3 structures :

— **La structure Position** permet à la fonction *select_position* de renvoyer 2 entiers, x et y, provenant des tableaux *positions_validesX* et *positions_validesY* via un unique indice choisi aléatoirement entre 0 et le nombre de positions valides. C'est la fonction qui crée la structure **Position** et la retourne.

— **La structure Player** a été créé pour réduire le nombre d'arguments dans les fonctions. En effet, nous utilisions régulièrement un tableau contenant le score de chaque joueur et un tableau contenant les files de cartes de chaque joueur. Plutôt coûteux en espace, nous avons préféré créer une structure *player* stockant la file de carte et le score d'un joueur. De cette façon, nous n'avons plus besoin que d'un unique tableau, *Players[]*, contenant la structure **Player** des joueurs.

— **La structure Board** permet de répondre à plusieurs demandes :
  — Sa variable *taille* permet d'obtenir la taille du bord, utile quand l'on souhaite parcourir tout le tableau.
  — Son *plateau* correspond à une matrice qui contient les pointeurs des tuiles placées ou de la tuile vide sinon.
  — Initialement une matrice, *from_player* permet d'obtenir le numéro du joueur qui a placé la tuile ou -1 si la tuile est vide. Durant l'achevement 0, son utilité est uniquement de tester si la tuile est vide ou non car si la case du plateau en ligne X et colonne Y est vide, *from_player[X][Y]* sera égal à -1. Ce tableau *from_player* prendra toute son importance lors de l'achivement 1 et évoluera jusqu'à l'achivment 3.

Il est important de revenir sur les fonctions *position_valide* et *authorised_places*.

postion_valide regarde un emplacement (ligne, colonne) et une tuile donnée en paramètres et doit nous renvoyer une réponse à la question "Peut-on placer cette tuile à cette position ?". Dans un premier temps, elle va vérifier qu'il n'y a pas déjà une tuile sur la position. Ensuite, si la position est libre, elle vérifie que les voisins sont de la bonne couleur ou vides tout en s'assurant qu'il y ait à minima un voisin.

authorised_places a besoin de deux tableaux *positions_validesX* et *positions_validesY*, dont la taille est égale au nombre de cases dans le plateau. Elle va parcourir l'ensemble des cases du plateau et appeler à chaque itération *postion_valide*. Si la position est valide, comme définie ci-dessus, alors *authorised_places* va incrémenter un compteur de case valide et va ajouter la ligne de la case dans *positions_validesX* et la colonne dans *positions_validesY*. A la fin du parcours, elle retourne le nombre de cases valides. Ainsi, la fonction récupère le nombre de cases valides avec leur position dans les tableaux *positions_validesX* et *positions_validesY*. Il suffit ensuite d'appeler *select_position* pour déterminer aléatoirement la position choisie parmi toutes les possibilités.

Le fichier **project.c** appelle les fonctions d'initialisation et crée la boucle de jeu :

— **Initialisation**

```
1         struct player players[MAX_PLAYERS];
2         init_players(nb_players, players);      // Initialise les joueurs
3         struct deck d;
4         deck_init(&d);                          // Initialise les decks
5         struct board b = board_init(board_long);
6         distrib(&d, nb_players, players);
7
8         // ---  On oblige manuellement le premier joueur à jouer au centre
9         struct position center = {.X=board_long/2, .Y=board_long/2};
10        place_tile(&b, pop(&(players[0].cards)), center, 0);
11
12        int skipped_turn = 0;
13        int active_player = 1;        // Initialise la boucle de jeu
14
```

— **Boucle de Jeu**

Algorithm 1 Boucle de jeu

```
while SkippedTurn < NombreJoueurs do
  if A Une Tuile then
    if Tuile Posable then
      On Pose la Tuile
      Enlève tuile du deck
      Passe au joueur suivant
    else
      Incremente Tour Passé
      Saute le tour du joueur
    end if
    // – Fin d'un tour ?
    if ActivePlayer == NombreJoueurs then
      Prochain Joueur = Joueur 0
    else
      On avance
    end if
  end if
  Affiche résultats
end while
```

Dans la boucle de jeu, on surveille lorsque l'on arrive à la fin d'un tour. La fin du tour est important à cause des nouvelles règles rejoutées par l'achievement 3.

Le mécanisme de la boucle de jeu écrit ci-dessus était fourni dans le sujet. La seule différence, entre notre algorithme et celui fournit, consiste à tester si un joueur n'a plus de tuiles. Dans ce cas, le programme arrête le jeu. Cette modification est apparue au fur et à mesure de l'avancement du projet ce qui explique cette différence.

Maintenant que nous avons le pseudo-code, il est nécessaire de le traduire en appelant les fonctions que nous avons définies précédemment :

— **Traduction "A une tuile"** :
Pour vérifier qu'un joueur a une tuile, le programme appelle la fonction *file_is_empty* sur le deck du joueur actif, c'est-à-dire sur la structure **Player**.*cards*. Pour récupérer la structure du joueur actif il faut utiliser le tableau *players*[la variable *active_player*] que initialisé avec *init_players*. Le programme récupères alors la tuile au sommet de notre joueur actif via la fonction *get_top* sur le deck.

— **Traduction "Tuile posable"** :
Définir si une tuile est posable quelque part revient à déterminer s'il existe une position valide sur le plateau pour cette tuile. Le programme récupère le nombre de positions valides avec l'appel de *authorized_places*. Deux cas possibles :

1. Il n'y a pas de case disponible
   Alors dans ce cas, la tuile n'est pas posable. Le programme saute le tour du jour en incrémentant *skipped_turn* et il renvoit la tuile au fond du deck avec le double appel *push* qui copie la tuile à la fin du deck la tuile et *pop* qui défausse la tuile au sommet.
2. Il y a une ou des case(s) disponible(s)
   Le joueur peut donc jouer. Le programme réinitialise la variable *skipped_turn* et il place une tuile sur une position valide via *place_tile*(tuile, *select_position*). Le changement de joueur se réalise en incrémentant *active_player*.

— **Traduction "Affiche les résultats"** La fonction *display_result* se charge directement d'afficher les scores et le gagnant. Dans les prochains achievments, il suffit d'appeler les fonctions qui calculent les scores avant *display_result*.

Après ces étapes, il manque une seule chose pour que le jeu fonctionne : nous n'avons pas encore distribué de cartes aux joueurs. Étant donné que la distribution n'est pas une règle du jeu, nous avons décidé de l'implémenter dans le fichier project.c, considérant que c'est une étape que le programme réalise lorsque nous voulons commencer à jouer ce qui en fait notre seule fonction définit dans ce fichier.

Le principe est simple :
- La fonction récupère la taille du deck
- Elle sélectionne un type de tuile
- S'il n'y en a plus, on supprime ce type
- Sinon, elle donne la tuile au joueur

3.5 Prise en compte des paramètres

L'autre consigne pour valider la version de base du projet consiste à permettre aux joueurs de sélectionner plusieurs paramètres. Nous avons pour réaliser cet objectif une fonction *parse_opts* fournie qui nous permet de récupérer les arguments donnés dans le terminal :

— n <entier> : Défini le nombre de joueurs
— b <entier> : Défini la taille du plateau
— s <entier> : Défini la graine aléatoire

Nous avons également ajouté un paramètre supplémentaire, sans options disponibles : -r.
Il permet de faire afficher le plateau, les tuiles placées et l'appartenance des joueurs sur les tuiles. Il n'est pas demandé dans le sujet de réaliser cette option mais il nous semblait primordial d'obtenir un résultat visuel pour tester nos fonctions.

3.6 Structure des fichiers

En structurant l'interdépendance des fichiers, nous répondons à la problématique de bien définir les structures **Player**, **Board**, **Deck** et **Tile** (pour respectivement les joueurs, le plateau, les files et les tuiles) entre elles.

Tout d'abord, on sait qu'il est nécessaire que les fichiers.c appellent les fichiers.h de même nom car ce sont ces derniers qui contiennent tous les prototypes. De plus, on sait que le fichier le plus "élementaire" correspond au fichier color.c car il définit simplement les couleurs. Donc, en partant de ce postulat, on sait d'ores et déjà que color.c ne doit appeler que color.h.
Ensuite, on sait que les tuiles sont définies par une couleur sur chacune de leur face et on sait que les decks des joueurs ou "file d'attente" ont besoin d'un ensemble de tuiles. Donc tile.c doit appeler color.h et file.c doit appeler tile.c.
Enfin, puisque rule.c définit l'ensemble des règles du jeu, on a besoin des decks, des tuiles et de leurs couleurs ansi rule.c doit appeler file.h, tile.h et color.h.
Il reste ensuite à project.c d'appeler les autres fichiers.h car il a besoin des prototypes de toutes les fonctions qu'il utilise. On obtient donc finalement une dépendance de fichiers illustrée comme ceci :

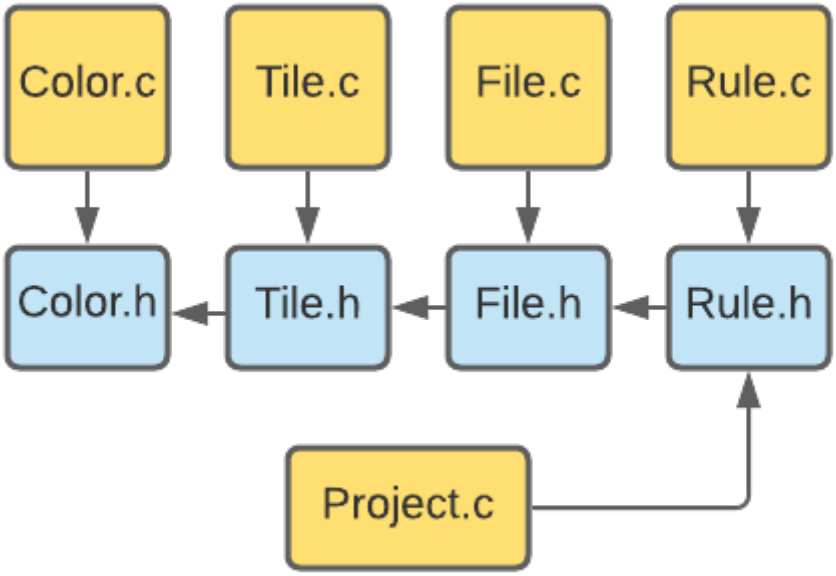

FIGURE 1 – Première interdépendance des fichiers

3.7 Tests

Il est essentiel de tester ces fonctions car, si dans certains cas, elles semblent fonctionner, elles pourraient être erronées et nous ne pourrions que difficilement remarquer ces erreurs en lançant le jeu. Ainsi, nous avons décidé de prendre une attention particulière sur les tests. Nous avons divisé les tests en plusieurs fichiers.
Le fichier `test.c` dirige les tests en appelant les fonctions *execute_test_nom_fichier*. Cette fonction est définie dans `test_nom_fichier.h` et est implémentée dans `test_nom_fichier.c`. Elle consiste à appeler les fonctions *test_nom_fonction*. Ce *nom_fichier* correspond aux fichiers que l'on souhaite tester et *nom_fonction* à ses fonctions.
Par exemple, `test_color.c` implémente *execute_test_color* qui appele *test_color_name*. Ainsi, nous avons un système qui centralise les tests de fichiers, eux mêmes découpés en tests de fonctions.

Pour cette version initiale, les tests de fonctions se contentent d'afficher 2 cas :
— Si nous obtenons un résultat différent de celui attendu : Une erreur est affiché sous la forme "ERROR : explication de l'erreur".
— Si le résultat est équivalent à celui attendu : "Le *test* est bien *test*".

Ce système de test évoluera lors de l'achievement 3 en affichant les erreurs en rouge et les bons résultats en vert. De plus, le nombre d'erreur de chaque fonction et de chaque fichier sera également renvoyé. Ce développement sera expliqué dans la partie "Ajouts implémentés".

Regardons maintenant les erreurs et problèmes que nous avons pu détectés grâce à nos tests :

— Les tests de `color.c` n'ont pas posé de problème important si ce n'est d'afficher le *cstring* renvoyé sans changer la couleur dans le terminal. Ce problème a été résolu en copiant le *cstring* retourné, en remplaçant son premier caractère par un "e" et en affichant \directement dans le *printf* (avant le %s) plutôt que de simplement appeler le *cstring* dans *printf*.

— Les tests de tile.c et rule.c se sont confrontés à un problème que nous n'avons pas su résoudre. Pour réaliser ces tests, nous avons besoin de tuiles spécifiques que nous connaissons. Par exemple, pour savoir si un emplacement est valide, nous devons connaître les couleurs de la tuile en question ainsi que les couleurs celles à proximité. Cependant, la couleur des tuiles dans le deck dépend de *deck_init*. Nos tests fonctionnent donc avec notre fichier tile.c mais ne peuvent fonctionner avec un autre fichier car les tuiles utilisées seraient différentes et ce changement modifirait donc les résultats attendus.

Voici un exemple de notre première version des tests :

```
Debut test rule.c
  Debut test board_init
    Taille du board valide
    Le board est bien compose de tuile vide
  Fin test board_init
  Debut test authorized_places
    Il y a bien 2 places valides
    Les cases valides sont bien aux bonnes positions
  Fin test authorized_places
  Debut test position_valide
    Une place entouree de case vide est bien invalide
    une case sur les bords est biens valide
    Deux couleurs differents donnent bien une place invalide
    Deux couleurs egales donnent bien une place valide
  Fin test position_valide
  Debut test place_tile
    La tuile a ete bien place
  Fin test place_tile
  Debut test select_position
    La position 5 2 est une position valide
  Fin test select_position
  Debut test color_equals
    Deux couleurs rouges sont biens consideres comme egale
    Deux couleurs rouge et magenta sont biens consideres comme non egale
  Fin test color_equals
  Debut test init_players
    Le joueur 0 a bien un score de 0
    Le joueur 1 a bien un score de 0
    Le joueur 2 a bien un score de 0
  Fin test init_players
Fin test rule.c
```

FIGURE 2 – Première version des test

4 Apparition de la notion de motifs

4.1 Problématiques

Dans cette nouvelle version du jeu, nous devons maintenant pouvoir implémenter des motifs, une combinaison spécifique de différentes tuiles. Il est également exigé que les méthodes pour construire les motifs soient suffisamment bien écrites pour pouvoir en rajouter aisément. Enfin, on ajoute la notion de propriété car il y a maintenant une importance à savoir quelle tuile appartient à quel joueur. Le calcul des points change également, le score correspond au nombre de motifs réalisé par les tuiles posées par chaque joueur.

4.2 Notion propriétaire

La première modification à faire dans notre jeu initial est d'ajouter un propriétaire pour chaque tuile. Si, précédemment, nous n'en avions pas besoin, car nous ajoutions des points dès qu'un joueur posait une tuile, il est maintenant requis de pouvoir reconnaître le propriétaire de chaque tuile à la fin du jeu. En effet, c'est en récupérant le numéro du joueur qui a posé la tuile centrale que nous pourrons ensuite

lui accorder des points via les motifs.

Pour ce faire, nous réutilisons la variable *from_player* de notre structure **Board**. Comme son nom l'indique, elle permettra de stocker le numéro du joueur qui pose une tuile à un emplacement (ligne, colonne) et nous permettra de remonter à son propriétaire quand auparavant, cette variable servait uniquement à tester s'il y avait ou non une tuile à ce même emplacement. On initialise cette matrice via la fonction *board_init* en spécifiant qu'une tuile vide n'a pas de propriétaire et donc prends la valeur -1. De cette façon, nous n'avons plus besoin d'appeler la fonction *empty_tile* pour vérifier si une tuile est vide ou non, il suffit de regarder si la matrice aux mêmes coordonnées possède un propriétaire. Ceci nous permet d'optimiser le programme car il est bien moins coûteux de faire une comparaison à un entier plutôt que de récupérer pour chaque côté la couleur de la tuile et de vérifier si elle est vide ou non.

4.3 Gestion des motifs

Nous souhaitons maintenant définir une structure **Motif**. Pour cela, nous créons un nouveau fichier (motif.h) qui contient notre définition. Nous avons besoin pour chaque motif :
- D'une fonction de test qui vérifie si le motif est présent sur le plateau.
- D'une variable qui définit si pour cette partie, on décide d'utiliser le motif.
- D'une variable score permettant d'évaluer les points relatifs à un motif.

La principale difficulté dans cette implémentation pour nous fut de définir un pointeur de fonction, car nous ne savions pas comment en créer un. En définitive, il suffit de préciser que c'est un pointeur, de préciser chaque type de variable dont aura besoin la fonction et bien sûr sa valeur de retour. On obtient donc le code ci dessous :

```
1  struct board;
2
3  struct motif
4  {
5      int (* test)(struct board *, int, int);    // Pointeur de fonction
6      int actif;
7      int score;
8  };
```

Maintenant que notre structure **Motif** est définie, il faut une fonction test spécifique à un motif particulier qui puisse le reconnaître sur le plateau. Dans notre jeu, nous avons créé 3 motifs :
- Une tuile entourée de 4 autres tuiles.
- Une tuile monochrome à proximité d'au moins deux tuiles bicolores.
- Une tuile monochrome à proximité d'au moins une tuile monochrome.

L'implémentation de ces motifs étant pratiquement similaire, nous n'allons décrire que la fonction test d'une tuile monochrome à proximité d'au moins deux tuiles bicolores.

1. **Fonction is_monocolor**

   Premièrement, on définit une fonction qui prends en paramètre une tuile et qui nous permet de savoir si elle est monochrome ou non. On récupère la couleur d'un côté de la tuile et on compare via la fonction *color_equals* si elle est égale aux couleurs des 3 autres directions avec l'appel *tile_edge*. De cette façon, on sait rapidement le type d'une tuile.

2. **Fonction motif_monocolor_with_two_bicolor**

   Deuxièmement, on crée la fonction qui teste si à un emplacement particulier, une tuile est une tuile centrale dans un motif. Pour cela, la fonction prends en paramètre des coordonnées (ligne, colonne) ainsi que le plateau de jeu. Voici son pseudo-code :

Algorithm 2 Pseudo-code Motif monocolor_with_two_bicolor

if N'est pas monocouleur **then**
  return **false**
end if
if N'est pas sur le bord gauche **then**
  On teste si il y a un bicolore à gauche
end if
if N'est pas sur le bord droit **then**
  On teste si il y a un bicolore à droite
end if
if N'est pas sur le bord haut **then**
  On teste si il y a un bicolore en haut
end if
if N'est pas sur le bord bas **then**
  On teste si il y a un bicolore en bas
end if
return $(Bicolore >= 2)$

Cette technique nous permet, avec seulement 4 comparaisons, de vérifier si une tuile est sur le bord ou non et elle gère tout les cas. Ces tests sont donc en complexité linéaire O(n) et nous avons préféré une implémentation de ce type plutôt que de traiter les cas où la tuile est sur le bord et de devoir traiter autrement les autres cas.

3. **Appel fonction de test** Enfin, on souhaite appeler maintenant pour chaque tuile non vide les fonctions de test des motifs. On définit pour cela la fonction *checking_motif* qui prends en paramètre le nombre de motifs à tester, les joueurs, le plateau et les motifs.

```
1  void checking_motif(struct player players[], struct board *b, int nb_motifs,
       struct motif motifs[])
2  {
3    for(int i=0; i<(b->taille); i++)
4      {
5          for(int j=0; j<(b->taille); j++)
6      {
7            int num = (b->from_player)[i][j];
8            if (num != -1)                          //Si la tuile est non vide
9        {
10         for (int m = 0; m < nb_motifs; m ++)
11         {
12               if (motifs[m].actif)          //Si on considère le motif pour cette
       partie
13               {
14                 if ( (*motifs[m].test)(b,i,j))  //Si la fonction trouve motif est
       vraie à un emplacement (i, j)
15                 {
16                   players[num].score += motifs[m].score;
17                 }
18               }
19             }
20       }
21     }
22     }
23 }
```

Il reste maintenant à initialiser les motifs dans project.c. Pour cela, nous créons un nouveau motif et nous remplissons ses paramètres en passant la fonction de test de motif en tant qu'adresse. Ainsi, nous rajoutons dans l'initialisation :

```
1    struct motif m1 = { .score = 3, .actif = 1, .test = &motif_monocolor_with_monocolor };
2    struct motif m2 = { .score = 5, .actif = 1, .test = &motif_monocolor_with_two_bicolor
       };
3    struct motif m3 = { .score = 10, .actif = 1, .test = &motif_full_surrond };
4
5    struct motif motifs[3] = {m1, m2, m3};
```

De plus, nous souhaitons appeler la vérification des motifs en fin de partie avant d'afficher les résultats. Nous rajoutons donc à la fin de la boucle de jeu l'appel à la fonction de test des motifs comme ceci :

Algorithm 3 Ajout de l'intégration des motifs à la boucle de jeu

while Joue **do**
  Le jeu continue comme précédemment
end while
Vérifie les motifs
Affiche les points des joueurs et le vainqueur

Cette implémentation des motifs permet de facilement rajouter des motifs ou d'ajouter un choix des règles où seul certains motifs sont actifs. Pour rajouter un motif il suffit :

- D'ajouter sa fonction de test dans rule.c et rule.h.
- Ensuite, de rajouter dans project.c, *struct motif mX = .score = x_score, .actif = 1, .test = &x_test* avec *x_test* correspondant au nom de la fonction de test que l'on a rajouté et *x_score* au score que nous voulons donner au motif.
- Enfin, de changer la valeur du nombre de motif en fonction du nombre de motif rajouté.

Pour le moment, nous avons donc tout les motifs qui sont actifs avec $mX.actif = 1$. Nous avons décidés de rajouter une option -m <entier> qui permette au joueur de supprimer un ou plusieurs motifs pour la partie. Dès lors que l'on met en argument le motif que l'on souhaite supprimer (numéroté de 1 à 3), le motif n'est plus considéré actif et donc nous n'appelons plus sa fonction test ni n'attribuons de points pour celui-ci.

4.4 Structure des fichiers

Pour la gestion des motifs nous avons besoin du fichier motif.h qui contient la définition de la structure **motif**, rule.h doit donc nécessairement inclure motif.h. La fonction de calcul des scores et les tests des motifs étant dans rule.h il n'est pas nécessaire d'avoir un fichier motif.c. Les autres dépendances n'ont pas évoluées depuis la dernière version.

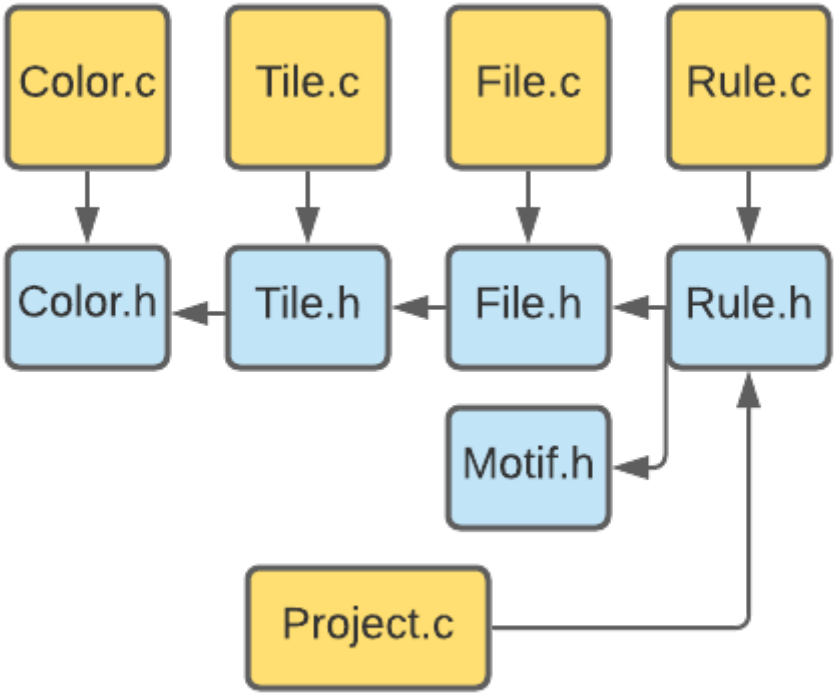

FIGURE 3 – Interdépendance des fichiers

4.5 Tests

Les fonctions correspondantes aux motifs ont été uniquement réalisées dans `rule.c`. Seul le fichier `test_rule.c` a changé. Nous avons divisé les tests en 2 fonctions :

- **La fonction *test_motifs*** Nous créons un plateau avec les 3 motifs et nous vérifions qu'ils sont bien présents avec les bonnes tuiles centrales. Nous vérifions également que sur les autres tuiles, elles ne détectent pas de motifs.
- **La fonction *test_check_motif*** On reprends le même plateau et cette fois, on vérifie que les scores attribués sont égaux aux scores que nous avions calculés pour ce plateau précis.

On remarquera que ces tests ont également les problèmes précédents car les résultats attendus des tests des motifs dépendent des tuiles utilisées et donc un changement des tuiles définies dans *deck_init* rend ces tests invalides.

5 Connections des tuiles

5.1 Problématiques

Nous souhaitons maintenant dans cette partie du sujet pouvoir remarquer les connections des couleurs entre les tuiles. Deux tuiles sont connexes si elles possèdent une arête ou un sommet commun. Si on a une connexité, on compte le nombre de tuiles et on attribue au(x) propriétaire(s) ayant le plus de tuiles dans cette zone connexe un score égal au carré du nombre des tuiles de la zone connexe.

5.2 Ajout de la connexité

Rajouter la connexité ressemble à ajouter un motif même si c'est un motif bien plus complexe. En effet, avec la connexité, nous avons de nouvelles difficultés :

- **Taille variable**
  Nous ne savons pas quand s'arrête la connexité ce qui complique la recherche du motif. Il faut veiller à ne pas créer de boucles infinies.
- **Pas de tuile centrale**
  Ici, toutes les tuiles peuvent potentiellement faire gagner des points, ce n'est pas une tuile centrale qui fait remporter tous les points à un seul joueur. Cela implique que tous les joueurs de toutes les tuiles mise en jeu doivent être considérées.
- **On étudie non plus une tuile mais les 4 côtés d'une tuile**
  Si les motifs étudiaient principalement une tuile dans son ensemble, ici on s'interroge sur les couleurs d'une tuile. Or une tuile possède 4 couleurs ce qui multiplie les tests.
- **Plusieurs gagnants possibles**
  On peut avoir autant de gagnants sur une connection que de joueurs sur le plateau. Il faut donc s'assurer que tout les gagnants gagnent des points ce qui augmente la complexité.

Pour répondre à ces difficultés, nous décidons de parcourir chaque tuile à la fin du jeu (à la manière de *checking_motif*). Pour chaque tuile non vide, nous étudions chacune des couleurs et nous regardons si nous les avons déjà étudié. Si ce n'est pas déjà fait, nous établissons un parcours en profondeur, c'est-à-dire qu'on traverse toute la connexité et on marque chaque couleur traversée comme étant déjà étudiée.

On pose la définition ci-dessus en pseudo code :

Algorithm 4 Pseudo-code de la connexité

```
for Ligne < taille do
  for Colonne < taille do
    for Couleur de tuile < 4 do
      if Non utilisé and Non vide then
        // – Une couleur ne peut faire partie que d'une seule connexité
        Parcourir en profondeur()
        Gérer score des gagnants
      end if
    end for
  end for
end for
```

Explications du parcours en profondeur :

1. Nous prenons un côté d'une tuile en argument.
2. La fonction vérifie si les autres côtés de la tuile sont de la même couleur.
3. Enfin, la fonction appelle, pour chaque côté de la même couleur, de manière récursive la fonction *parcours_en_profondeur* ce qui permet la diffusion de la connexité.

Algorithm 5 Pseudo-code du parcours en profondeur

```
if Tuile considérée non étudiée then
    Tuile considérée comme étudiée à partir de maintenant
end if
for Les autres couleurs de la tuile do
    Appel récursif de parcours_en_profondeur
end for
En fonction du côté proposé
On étudie le côté correspondant
if Côté correspondant n'est pas vide then
    if Couleurs similaires then
        Appel récursif de parcours_en_profondeur
    end if
end if
```

Maintenant que nous avons expliqué l'idée de base de ces deux fonctions, nous allons étudier leur implémentation qui est plutôt complexe.

```
1
2       void connexe(int nb_player, struct player players[], struct board *b)
3       {
4           // -- On définit une matrice de tableau
5           // ==> Permet d'étudier chaque couleur de chaque tuile du plateau
6           /*  * 0 : non traité
7               * 1 : déjà traité
8               connexe_used[l][c][0] : North
9               connexe_used[l][c][1] : South
10              connexe_used[l][c][2] : East
11              connexe_used[l][c][3] : West
12              connexe_used[l][c][4] : Tuile entiere
13          */
14        int connexe_used[MAX_BOARD_LONG][MAX_BOARD_LONG][5] = {{{0}}};
15
16        for(int l=0; l< b->taille; l++)
17          {
18              for(int c=0; c< b->taille; c++)
19              {
20                  for(int i=0; i<4; i++)              // Pour chaque couleur
21                  {
22                    if (connexe_used[l][c][i] == 0 && b->from_player[l][c] != -1) {
23                      //Initialisation d'une connexité
24                  struct zone_connexe z = {.nb_tuile = 0, .cases_players = {0}};
25
26  /* En effet, si une couleur n'est pas déjà étudié et non vide dans l'appel de connexe
27       Alors c'est forcément une connexité non traité que l'on vient de trouvé */
28
29                  for(int li=0; li< b->taille; li++) {
30                      for(int ci=0; ci< b->taille; ci++) {
31                    connexe_used[li][ci][4] = 0;
32                      }
33                  }
34
35                 //On lance le parcours récursif
36                parcours_en_profondeur(b, l, c, i, connexe_used, &z);
37
38          // --- On peux considérer à partir d'ici que l'on a terminé l'étude de la
       connexité
39          // On gère maintenant les scores
40
41                      int max = 0;
42                      int nb_players_max = 0;
43                      int players_max[MAX_PLAYERS];
44
45          // --- Sélection des joueurs qui ont le plus de case dans la zone connexe
```

```
46                      for (int k = 0; k < nb_player; k ++)
47                      {
48                          if (max < z.cases_players[k])
49                          {
50                              nb_players_max = 1;
51                              players_max[nb_players_max - 1] = k;
52                              max = z.cases_players[k];
53                          }
54                          else if ( max == z.cases_players[k])
55                          {
56                              nb_players_max ++;
57                              players_max[nb_players_max - 1] = k;
58                          }
59                      }
60
61                      // --- Distribution des points
62                          for (int v = 0; v < nb_players_max; v ++)
63                          {
64                              players[players_max[v]].score += (z.nb_tuile * z.nb_tuile) /
      nb_players_max;
65                          }
66                      }
67                  }
68              }
69          }
70      }
71
```

Cet algorithme s'appuie sur deux éléments qui vont être modifiés et utilisés lors des appels récursif de *parcours_en_profondeur* : La structure **zone_connexe** et la matrice de tableau *connexe_used*.

— La structure **zone_connexe** est utilisée pour l'attribution des scores et possède deux attributs : *nb_tuile* contient le nombre de tuiles pour le calcul des points et *cases_players* est un tableau associant le numéro des joueurs avec le nombre de cases qu'ils possèdent. Lors du troisième achievement, ce sera modifié en la somme des niveaux d'appartenance des cases de la connexité.

— La matrice de tableau *connexe_used* nous permet d'éviter de compter plusieurs fois la même case : Dans l'appel de la fonction *parcours_en_profondeur* à un emplacement (l, c), nous étudions un côté spécifique renseigné par l'entier *i* tel que : 0 (Nord), 1 (Sud), 2(East), 3(West). Si la valeur de *connexe_used[l][c][i]* est égale à 0, alors nous l'initialisons à 1. Si, en revanche, la valeur est d'ores et déjà à 1, alors le parcours s'arrête car la tuile à déjà été traitée.

Ensuite, nous étudions la valeur *connexe_used[l][c][4]* qui nous permet de déterminer si, pour ce parcours, la tuile a été compté. Si la valeur est égale à 0, alors nous la fixons à 1 et nous incrémentons le nombre de cases totales ainsi que le nombre de cases du propriétaire de la tuile dans la structure **zone_connexe**.

De cette manière, une case n'est comptée qu'une seule fois dans le même parcours même si plusieurs de ses côtés sont de la même couleur. De plus, une tuile peut être dans plusieurs parcours différents, notamment si elle est composée de plusieurs couleurs. C'est pour cela que dans la fonction *connexe*, entre chaque appel de *parcours_en_profondeur*, il faut remettre à 0 *connexe_used[l][c][4]* en plus de créer une nouvelle structure **zone_connexe**.

Enfin, le parcours va s'appeler sur les autres parties de la tuile si elles sont de même couleur. Pour terminer, le parcours va se propager en fonction du côté opposé de la direction du morceau de tuile qu'il étudie, c'est-à-dire, par exemple, étudier la couleur du côté Sud de la tuile (2, 2) avec l'appel *parcours_en_profondeur* du côté Nord de la tuile (3, 2).

5.3 Tests

Les tests de la connexité sont dans le fichier test_rule.c car les fonctions *connexe* et *parcours_en_profondeur* sont dans rule.c. Ces tests sont regroupés dans la fonction *test_zonne_connexe* qui va créer un plateau puis appeler les fonctions *test_parcours_en_profondeur*et *test_connexe*, qui testent respectivement *parcours_en_profondeur* et *connexe*, en leur donnant un plateau spécifique. Ces tests, se basant sur les couleurs des tuiles, seraient obsolètes au moindre changement de tuile dans le deck et un deck de moins de 5 tuiles provoquerais une erreur de segmentation car pour créer le tableau nous utilisons la 5ème tuile.

6 Influence des joueurs sur les tuiles

6.1 Problématiques

— Premièrement, nous souhaiton maintenant que les tuiles puissent changer de propriétaire. Pour cela, le programme attribue à chaque tuile un niveau d'influence pour chacun des joueurs. A chaque fois qu'un joueur joue une tuile, ce joueur la possède par une influence de 100, tous les autres n'ont aucune influence sur la tuile. Puis, à chaque tuile posée à côté de cette même tuile, le programme attribue une certaine influence en fonction des tuiles déjà pré-disposées. Enfin, à chaque fin de tour, une fonction réduit l'influence de tout les joueurs de 1 pour chacune des tuiles.

— Deuxièmement, nous souhaiton maintenant implémenter une mécanique aléatoire, Le King, qui choisit une case du plateau et en cas de découverte d'une tuile, réduirait l'influence du (des) propriétaire(s) de 50 sur la tuile et toutes les autres tuiles connexes sur une même ligne ou une même colonne que la tuile ciblée.

— Troisièmement, le(s) propriétaire(s) d'une connexité corresponde(nt) maintenant au(x) joueur(s) dont la somme des influences de toutes les tuiles de la zone connexe est maximale.

6.2 Implémentation de l'influence

L'influence permet de déterminer le(s) propriétaire(s) d'une case nous avons donc décider de modifié *from_player* du plateau pour l'implémenter. Il transforme la matrice *from_player[ligne][colonne]* en une matrice de tableau *from_player[ligne][colonne][numéro_joueur]*. Les noms *ligne* et *colonne* correspondent toujours à un emplacement (ligne, colonne) et *numéro_joueur* correspond au numéro du joueur. Ainsi, *from_player[l][c][numéro_joueur]* donne désormais le niveau d'appartenance du joueur *players[numéro_joueur]* sur la tuile en position (ligne, colonne).

L'implémentation de la réduction d'influence, nommée *update_propriete* défini dans rule.c, est donc triviale car il suffit d'une boucle sur toutes les lignes, les colonnes et les numéros des joueurs en vérifiant si l'influence des joueurs est supérieur à 0. Si c'est le cas, nous la réduisons de 1. Cependant, cela cause des problèmes avec nos précédentes utilisations de *from_player*.

6.3 Conséquences sur les fonctions existantes

L'initialisation de *from_player* change pour correspondre à sa nouvelle définition. Cependant le principe reste le même et lors de l'initalisation de plateau on met la valeur à -1 pour tous les numéros de joueurs sur toutes les lignes et les colonnes.

Une des premières utilisations modifiées immédiatement est notre test de case vide en comparant *from_player[ligne][colonne]* avec -1. Le nombre de joueurs est variable, mais il y a toujours au moins un joueur. Sachant que l'influence est initialisé à -1, que dès qu'une tuile est placée, un des joueurs à une influence de 100 et tout les autres ont une influence égale à 0 et que l'influence des joueurs sur une tuile ne peut descendre en dessous de 0, il suffit donc de comparer *from_player[ligne][colonne][0]* avec -1 pour déterminer si la case est vide ou non. Nous pouvons garder ce fonctionnement avec cette simple mise à jour.

Le placement d'une tuile change de 2 façons :

— En premier lieu, plutôt que d'avoir seulement le numéro du joueur qui place une tuile, il faut initialisé l'influence à 100 pour ce joueur sur cette tuile et 0 pour le reste d'entre eux. C'est trivial, une boucle met l'inflfluence de tout le monde à 0 puis on met à 100 la valeur *from_player[numéro du joueur qui la placée]* (donné en paramètre de la fonction).

— Deuxièmement, il faut maintenant implémenté l'ajout de l'influence sur les cases à côté en fonction du nombre de cases placées avant, ce qui est plus complexe. Nous avions d'abord décidé de compter le nombre de case non vide à côté de la case dont on doit modifier l'influence mais nous avons finalement décidé, pour réduire le nombre de tests, de rajouter l'attribut *nb_tuiles_placed_after*, initialisé à 0, dans la structure **Board**. Ainsi, il nous suffit donc d'augmenter cette valeur de 1 et d'ajouter à l'influence (100 - 10 * *nb_tuiles_placed_after[l][c]*) pour le joueur posant la tuile à proximité.

Les points pour un motif sont attribués au(x) propriétaire(s) de la tuile centrale du motif. Désormais, il peut y avoir plusieurs propriétaires dès lors que la tuile centrale à plusieurs joueurs avec un même niveau maximum d'influence. Nous avons donc décider de créer une fonction *proprietaires* qui prends en paramètre le plateau (b), une ligne (l), une colonne (c), le nombre de joueurs dans la partie *(nb_players)* et d'un tableau d'une taille supérieur ou égale à *nb_players (proprios)*. Elle remplis le tableau *proprios* avec les numéros des joueurs qui ont la valeur d'influence la plus grande pour la tuile en position (l, c) et retourne leur nombre. Grâce à cette fonction, il suffit de l'utiliser sur la tuile centrale d'un motif puis de faire une boucle sur les éléments de *proprios* de 0 au nombre retourné par *proprietaires*. Dans la boucle, nous aurons donc les numéros des joueurs pour lesquels il faut augmenter le score.

Pour les zones connexes, *cases_players* correspond désormais à la somme de l'influence de chaque case de la zone pour le joueur. Il suffit donc de changer, dans le *parcours_en_profondeur*, l'incrémentation de cette valeur en une somme de l'influence des joueurs d'une tuile de la connexité et de l'influence des joueurs des autres tuiles. Le programme stocke au fur et à mesure cette influence ttale dans *cases_players[numero_joueur]*. Le reste du comportement ne diffère pas.

6.4 Implémentation du King

Nous pouvons résumer l'action du King par une unique fonction que l'on crée dans rule.c car plutôt que de voir le King comme étant un joueur spécial, on peut considérer que c'est simplement une nouvelle règle dans le jeu. Elle prends plusieurs paramètres : Le plateau, le nombre de joueurs mais également un emplacement (ligne, colonne) aléatoire. Voici son pseudo-code :

Algorithm 6 Pseudo-code du King

```
if A l'emplacement (l, c) il y a une tuile then
  P = le(s) propriétaire(s)
  if P.Influence >= 50 then
    P.Influence -= 50
  else
    P.Influence = 0
  end if
end if
// - Etude connexité
// — Ligne
while Pas sur le bord gauche do
  Décalage vers la gauche
  P = le(s) propriétaire(s)
  if P.Influence >= 50 then
    P.Influence -= 50
  else
    P.Influence = 0
  end if
end while
while Pas sur le bord droit do
  Décalage vers la droite
  On récupère le(s) propriétaire(s)
  if P.Influence >= 50 then
    P.Influence -= 50
  else
    P.Influence = 0
  end if
end while
// — Colonne
...
```

6.5 Appel des fonctions dans la boucle de jeu

Dès lors, il suffit d'appeler les nouvelles fonctions à chaque fin de tour dans la boucle de jeu :

```
1     //Fin d'un tour
2     if (active_player == (nb_players - 1))
3       {
4         update_propriete(&b, nb_players);
5
6         int chosed_l = rand() % b.taille;
7         int chosed_c = rand() % b.taille;
8         active_player = 0;
9         king(&b, nb_players, chosed_l, chosed_c);
10      }
```

6.6 Tests

De la même façon que les fonctions pour la connexité, les fonctions gérant l'influence sont également dans `rule.c`. Les tests de l'influence sont donc également dans `test_rule.c`. Ces tests correspondent à une mise a jour des tests des fonctions modifiées (*place_tile*, *motifs*, ...) et des nouvelles fonctions (*king* et *update_propriete*). Ces fonctions, ne dépendant pas des couleurs, permettent aux tests de rester valide lors d'un changement de deck à condition que le deck contient toujours au moins 5 tuiles différentes car la 5ème tuile du deck est utilisée pour construire la structure **Board**.

7 Conclusion

7.1 Ajouts implémentés

En plus des consignes apportés par les différents achievments, nous avons implémentés des fonctions/outils supplémentaires pour les joueurs. Si une partie de ces modifications a déjà été expliqué, nous préférons tout de même les réécrire ici pour centraliser ces ajouts :

— **Amélioration des tests.**
Pendant une grande partie du projet, les tests s'affichait avec la couleur classique du terminal et affichait uniquement un message si tout fonctionnait correspond ou s'il y avait une erreur. Pour facilité la lecture, nous avons décidé d'afficher les erreurs en rouge, d'afficher les résultats attendus en vert et d'afficher le nombre d'erreurs total, par fichier et par fonction. Les fonctions de test ne sont alors plus des *void* mais des *int* et elles renvoient le nombre d'erreur trouvé. Pour cela nous avons créé deux fichiers outils.c et outils.h qui regroupent des fonctions utilent pour les tests : les changements de couleurs (rouge/vert/blanc) et un *template* pour afficher le nombre d'erreurs.

```
  Debut test update_propriete
    Une influence de 0 n'est pas reduite a la fin de chaque tour
    Une influence de 90 a ete reduite a 89 a la fin du tour
    Une influence de 50 a ete reduite a 49 a la fin du tour
    Une influence de -1 n'est pas reduite a la fin de chaque tour
  Fin test update_propriete
  Debut test king
    La case touche a bien ete reduite a 10
    La case en haut a bien ete reduite a 10
    La case en haut a bien ete reduite a 50
    La case touche a bien ete reduite a 10
    La case a droite a bien ete reduite a 0
    La case en haut non connexe n'a pas ete reduite
    La case touche a bien ete reduite a 0 pour le proprietaire
    Un non proprietaire de la case touche n'a pas eu son influence modifie
    La case en haut a bien ete reduite a 20 pour le proprietaire
    Un non proprietaire de la case en haut n'a pas eu son influence modifie
    Les deux proprietaires de la case touche ont bien eu lors influence reduite a 20
    Les deux proprietaires de la case touche ont bien eu lors influence reduite a 0 et 10
  Fin test king
Il n'y a pas d'erreur dans les tests de board_init
Il n'y a pas d'erreur dans les tests de authorized_places
Il n'y a pas d'erreur dans les tests de position_valide
Il n'y a pas d'erreur dans les tests de place_tile
Il n'y a pas d'erreur dans les tests de select_position
Il n'y a pas d'erreur dans les tests de is_monocolor
Il n'y a pas d'erreur dans les tests des motifs
Il n'y a pas d'erreur dans les tests de check_motif
Il n'y a pas d'erreur dans les tests de color_equals
Il n'y a pas d'erreur dans les tests de init_players
Il n'y a pas d'erreur dans les tests des zones connexes
Il n'y a pas d'erreur dans les tests de update_propriete
Il n'y a pas d'erreur dans les tests de king
Il n'y a pas d'erreur dans les tests de rule
Fin test rule.c
----------------------------------------

Il n'y a pas d'erreur dans les tests de color
Il n'y a pas d'erreur dans les tests de tile
Il n'y a pas d'erreur dans les tests de rule
Il n'y a pas d'erreur dans les tests

*** Fin des tests ***
```

FIGURE 4 – Version finale des test

— **Rendu visuel plateau via l'option -r**
Affiche le plateau avec les bords des tuiles en couleurs, avec à l'intérieur le numéro du propriétaire s'il y en a un unique "pl" sinon et l'influence du propriétaire si elle est inférieure à 100 ou "C+" dans le cas contraire.

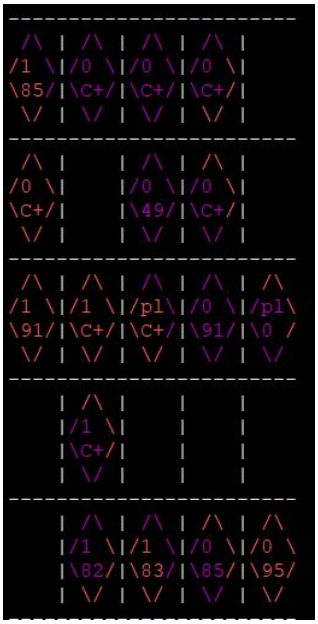

FIGURE 5 – Plateau de jeu

— **Supprimer des motifs via l'option -m**
En plus de pouvoir créer des motifs de façon générique, nous avons implémenté en bonus une option -m <entier> permettant de supprimer des règles du jeu un motif pour une partie. De cette façon, on peut à la fois rajouter facilement et supprimer facilement des motifs d'une partie.

— **Rendu visuel influence via l'option -i**
Affiche pour chaque tuile l'influence de chaque joueur.

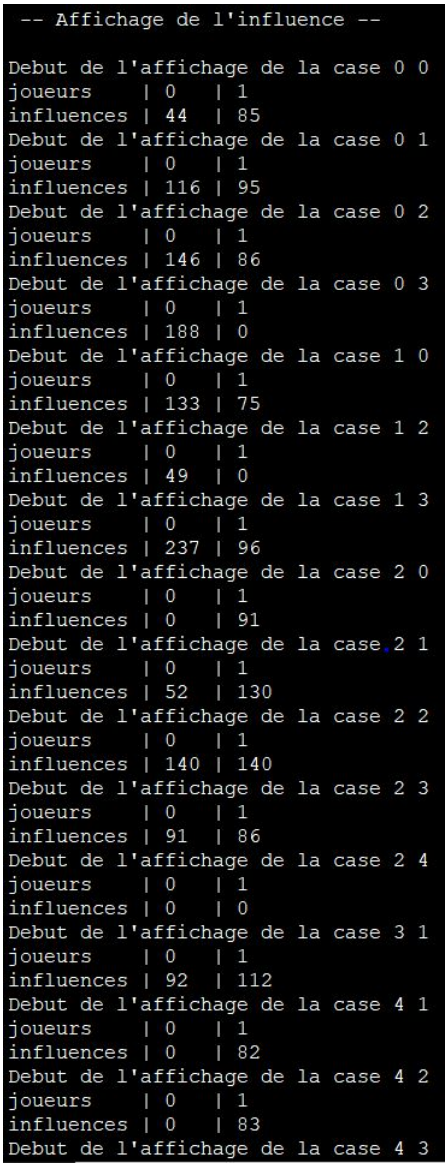

FIGURE 6 – L'influence des joueurs

— **Limiter la casse au niveau des options**
Qu'advient-il si nous décidons de jouer une partie avec un nombre de joueurs négatifs ? Ou un seul joueur ? De la même façon, que renvois notre programme si l'on lance une partie avec un plateau de taille négative ? Pour gérer ces cas "extrêmes", nous définissons des variables par défaut dans rule.h :

```
1 #define MAX_PLAYERS 20
2 #define MAX_BOARD_LONG 50
3 #define DEFAULT_BOARD_LONG 10
```

De cette façon, tous les problèmes sont résolus :
Si les joueurs tentent de jouer sur un plateau trop grand, d'appeler un nombre négatif de joueurs, etc, alors le système prends des paramètres par défaut c'est-à-dire 20 pour le nombre de joueurs et 10 pour la taille du plateau.

De plus, si le joueur met de mauvais arguments, par exemple "-v", alors le programme lui affiche comment l'utiliser :

```
./project: invalid option -- 'v'
Usage: ./project [-s seed]
Usage: ./project [-n number of players]
Usage: ./project [-b width of the board]
Usage: ./project [-r] Print results
Usage: ./project [-i] Print influence
Usage: ./project [-m which motif retired]
```

Ainsi, si l'utilisateur n'a pas lu le fichier README.txt, il peut quand même comprendre le fonctionnement du programme.

7.2 Ce que nous a apporté le projet

Ce projet nous a permis de développer des compétences techniques. Nous avons progressé dans notre maîtrise du langage C et du makefile lors du développement, dans notre connaissance de LaTeX pour l'écriture de ce rapport et dans notre utilisation de Git durant toute la durée du projet. Nous avons également appris à travailler en équipe, comment réfléchir ensemble aux problèmes posés par le sujet, avant de se répartir les tâches pour l'implémentation des solutions. Nous sommes très satisfait de ce projet et avons beaucoup apprécié notre collaboration en tant que binôme.