



---

## RAPPORT DE PROJET FLOOD-FILLING

---

Semestre 6 – Année universitaire 2021-2022

# Table des matières

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>1</b>  |
| 1.1      | Présentation du sujet . . . . .                                      | 1         |
| 1.2      | Définition du cadre . . . . .  | 2         |
| 1.3      | Définition des différentes parties . . . . .                         | 2         |
| <b>2</b> | <b>Analyse et conception algorithmique</b>                           | <b>2</b>  |
| 2.1      | Type abstrait "graphe" . . . . .                                     | 2         |
| 2.2      | Fonctions avancées pour la gestion du graphe . . . . .               | 3         |
| 2.2.1    | Conception de fonctions au service du client et du serveur . . . . . | 3         |
| 2.2.2    | Parcours de graphe . . . . .   | 5         |
| 2.2.3    | Gestion d'ensembles de sommets . . . . .                             | 5         |
| 2.3      | Conception d'un client . . . . .                                     | 5         |
| 2.4      | Conception du serveur . . . . .                                      | 7         |
| <b>3</b> | <b>Réalisation et implémentation</b>                                 | <b>8</b>  |
| 3.1      | Étapes de réalisation . . . . .                                      | 8         |
| 3.2      | Implémentation de la gestion des graphes . . . . .                   | 8         |
| 3.2.1    | Importation d'une librairie dynamique . . . . .                      | 8         |
| 3.2.2    | Allocations dynamiques . . . . .                                     | 9         |
| 3.2.3    | Réalisation du parcours en profondeur . . . . .                      | 9         |
| 3.2.4    | Implémentations des ensembles de sommets . . . . .                   | 9         |
| 3.3      | Réalisation du serveur . . . . .                                     | 11        |
| 3.3.1    | Gestion des arguments . . . . .                                      | 11        |
| 3.3.2    | Chargement dynamique . . . . .                                       | 11        |
| 3.3.3    | Gestion d'une partie . . . . .                                       | 12        |
| 3.4      | Implémentations du client . . . . .                                  | 13        |
| 3.4.1    | Compilation en librairie dynamique . . . . .                         | 13        |
| 3.4.2    | Les différentes stratégies réalisées . . . . .                       | 13        |
| 3.5      | Documentation . . . . .  | 15        |
| <b>4</b> | <b>Validation</b>  | <b>15</b> |
| 4.1      | Les outils et le développement de tests . . . . .                    | 15        |
| 4.2      | Complexités des implémentations . . . . .                            | 16        |
| <b>5</b> | <b>Conclusion</b>  | <b>16</b> |
| 5.1      | Résultat du projet . . . . .   | 16        |
| 5.2      | Idées d'amélioration du projet . . . . .                             | 17        |

# 1 Introduction

## 1.1 Présentation du sujet

L'objectif de ce projet est de créer un jeu de type Flood-filling, semblable à OpenFlood ou FloodIt. Le but de ces jeux est de s'approprier tout un plateau en changeant de couleur pour élargir sa zone. Cependant, le jeu proposé ici est différent car il s'agit d'une adaptation de ce concept à deux joueurs selon les principes suivants :

- chaque joueur commence à une extrémité du graphe (sauf dans le cas d'un graphe torique, dans lequel un des joueurs commence au "milieu") de sorte que les joueurs soient les plus éloignés possibles.
- Une partie se déroule au tour par tour. Lorsque c'est à un joueur d'effectuer son coup, il peut décider de changer de couleur, ou bien de passer son tour.
- La partie s'arrête lorsque les deux joueurs passent leur tour consécutivement. Le gagnant est celui qui possède le plus grand territoire.

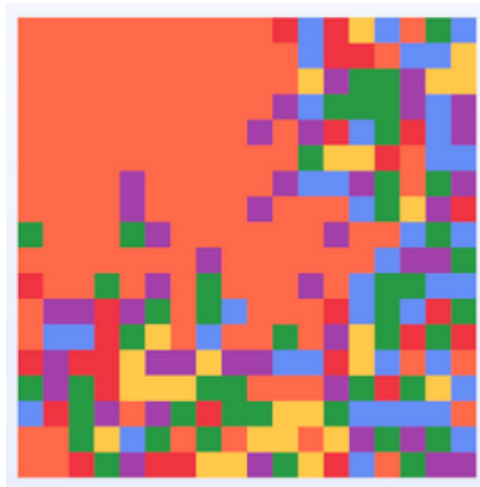


Figure 1: Exemple de partie tiré du jeu OpenFlood

À ces concepts généraux s'ajoutent certaines règles, afin de rendre le jeu jouable :

- Un joueur ne peut pas jouer de couleur qui n'augmente pas son territoire.
- Un joueur ne peut pas absorber la zone de son adversaire.
- Un joueur ne peut jouer que les couleurs qui lui sont disponibles (les couleurs disponibles pour chaque joueur sont définies en début de partie).

Dans ce projet, nous devons créer un serveur capable d'héberger un tel jeu, mais aussi des clients joueurs capables d'interagir avec le serveur pour jouer à ce jeu.

## 1.2 Définition du cadre

Le langage de programmation utilisé dans ce projet est le C. Le serveur est compilé sous la forme d'un exécutable `server`, de même que nos différents tests seront compilés sous la forme d'un exécutable `alltests`. Les joueurs sont quant à eux compilés sous la forme de bibliothèques dynamiques. Afin de compiler ces différents fichiers, nous utilisons un fichier `Makefile`, avec entre autre une règle `make install` pour déplacer tous les fichiers ci-dessus une fois compilés dans un dossier `install`. Afin de mener à bien ce projet, nous avons à notre disposition un dépôt git muni de tests basiques relevant de la compilation ou de la syntaxe.

Par ailleurs, le sujet nous impose une interface client/serveur commune pour toutes les implémentations du client ou du serveur. Ces fonctions interface seront développées dans la partie sur les clients.

## 1.3 Définition des différentes parties

Afin de fournir un code de qualité, il est important de séparer les différentes tâches à réaliser dans plusieurs catégories :

- les fonctions de structure des graphes : celles-ci serviront à la représentation et la manipulation de graphes basiques (sans coloration) à l'aide de matrices GSL.
- les fonctions avancées des graphes : ces fonctions auront pour objectif de représenter et manipuler les plateaux de jeu, donc des graphes avec coloration. Ces fonctions utilisent les fonctions basiques des graphes.
- les fonctions des clients : ces fonctions utilisent les fonctions complètes des graphes sans connaître les structures internes aux graphes.
- les fonctions du serveur : elles veillent au bon déroulement de la partie et s'occupent également de la gestion des clients. Elles utilisent donc les fonctions avancées des graphes ainsi que les fonctions des clients.

Ces différentes couches d'abstraction du code seront respectivement implémentées dans `graph_basic.c`, `graph_utils.c`, les différents fichiers de joueurs et `server.c`. Cette répartition augmente la modularité et donc l'adaptabilité de notre code.

# 2 Analyse et conception algorithmique

## 2.1 Type abstrait "graphe"

Dans cette partie, nous allons discuter de notre choix de considérer le graphe comme un type abstrait de données. Initialement, il y avait plusieurs représentations possibles de

graphes, donc pour pallier à ce problème, il est mieux d'écrire un fichier `graph_struct.h` dans lequel on mettra notre propre implémentation de la `struct graph`. Ce fichier ne doit pas être inclus par un autre fichier que ceux implémentant le type abstrait "graphe" afin de garder l'abstraction du type sur l'ensemble du programme.

Voici l'ensemble des primitives de type abstrait "graphe" et sont définies en C dans le fichier `graph_basic.h`, leur nom permet de comprendre explicitement leurs objectifs:

|  |                        |   |                   |
|--|------------------------|---|-------------------|
| <code>graph__empty :</code>                  | entier                 | → | graphe            |
| <code>graph__free :</code>                   | graphe                 | → |                   |
| <code>graph__add_edge :</code>               | graphe×sommets×sommets | → | graphe            |
| <code>graph__get_neighbors :</code>          | graphe×sommets         | → | liste de sommets  |
| <code>graph__get_color :</code>              | graphe×sommets         | → | couleur           |
| <code>graph__set_color :</code>              | graphe×sommets×couleur | → | graphe            |
| <code>graph__get_positions :</code>          | graphe                 | → | couple de sommets |
| <code>graph__set_positions :</code>          | graphe×sommets×sommets | → | graphe            |
| <code>graph__get_number_of_vertices :</code> | graphe                 | → | entier            |
| <code>graph__compress :</code>               | graphe                 | → | graphe            |

La fonction `graph__compress` est moins explicite, en effet le sujet impose d'utiliser les matrices GSL sous un certain format. À l'initialisation la matrice GSL incluse dans le graphe est sous un format par défaut. Il faut donc prévoir une fonction permettant de convertir celle-ci afin de la mettre dans le bon format (comprimé).

Il existe d'autres primitives non explicitées (`graph__get_colors_copy` par exemple) ici car elles ont pour but de réaliser des copies ou d'obtenir un graphe à partir de données.

## 2.2 Fonctions avancées pour la gestion du graphe

### 2.2.1 Conception de fonctions au service du client et du serveur

- `struct graph * create_square_graph(int n);`

Cette fonction permet de créer un graphe carré qui sera une grille de taille  $n \times n$  avec  $n$  le premier argument de la fonction. Elle utilise la fonction `graph_empty` élémentaire puis remplit les champs nécessaires par la suite.

- `void coloring_graph_randomly(struct graph * g);`

Cette fonction permet d'attribuer à chaque sommet une couleur. Cela colorie le graphe aléatoirement en changeant le champ `colors` de `g` qui est un tableau de taille le nombre de sommets au carré -1 vu qu'on travaille avec des graphes carrés jusqu'à là.

- `void display_graph_as_square(struct graph * g,  
int width,`

```
int is_sdl);
```

Cette fonction permet d'afficher un graphe carré de largeur `width` et de longueur `width` avec un troisième argument `is_sdl` qui permet de prendre en compte si l'affichage se fera sur une fenêtre SDL ou non avec la convention  $x \times width + y$  pour convertir un couple  $(x, y)$  dans une matrice à un indice dans un tableau classique.

- `int number_vertices_from_player(struct graph * g, int player);`

Cette fonction agit sur le joueur. C'était le but depuis le départ. Elle permet depuis la position d'un `player` accessible depuis la structure `g` de retrouver le nombre de sommets que peut gagner ce joueur soit le nombre de sommets dans la composante connexe obtenue après avoir fait un parcours en profondeur depuis le sommet qui représente la position du joueur et compter le nombre de sommets ayant la même couleur et modifier par conséquent le tableau de booléens concerné.

- `void number_vertices_winnable_from_player(struct graph * g,  
int player,  
int result_by_color[]);`

Cette fonction permet de changer par effet de bord le tableau `result_by_color`. Elle permet d'agir sur le prochain coup du joueur par anticipation. Depuis un graphe et un joueur, on agit sur la position de ce dernier en récupérant la composante connexe de son territoire (après avoir récupéré la couleur du sommet qui représente la position du joueur en paramètre avec les fonctions élémentaires précédentes) et en ajustant le tableau `result_by_color` qui est un tableau de booléens.

- `void coloring_graph_from_player(struct graph * g,  
int player,  
enum color_t color);`

Cette fonction permet depuis une structure de graphe `g`, un `player` (soit le premier soit le deuxième sur un tableau de jeu ordinaire) et une couleur de colorier tout le groupe de sommets conquis par le joueur par la couleur optimale choisie en paramètre (c'est différent des stratégies là : on impose la couleur et on récupère la composante connexe si le joueur avait choisi cette couleur ; on considère que la position du joueur est coloriée en `color` (2ième argument de la fonction) et on récupère la composante connexe avec la fonction `get__component` relative à cette couleur depuis le sommet où se trouve le joueur).

- `int is_touching_opponent(struct graph * g);` Cette fonction permet de tester si, dans un graphe donné, le groupe de sommets qui concerne le premier joueur et celui qui concerne le deuxième s'intersectent. C'est à dire renvoyer un booléen qui traduit si les 2 composantes connexes relatives aux deux joueurs ont un sommet en commun ou non.

### 2.2.2 Parcours de graphe

Dans l'objectif de permettre au client et au serveur d'utiliser des fonctions sur les graphes, il faut concevoir un moyen de calculer des composantes connexes de sommets de même couleur. Le parcours de graphe depuis un sommet semble être indispensable. Il existe plusieurs types de parcours, les principaux sont: le parcours en profondeur (DFS en anglais) et le parcours en largeur (BFS en anglais).

Les deux parcours ont plusieurs implémentations sont possibles mais le parcours en profondeur est facilement implémentable en récursif et ne nécessite pas de structures particulières. C'est donc le parcours en profondeur que l'on retient pour la conception et l'implémentation

### 2.2.3 Gestion d'ensembles de sommets

On représente l'ensemble indirectement par une structure qui contient un tableau de booléens et comme premier champ (cette structure ne contient que deux champs) la longueur de ce dernier. Et cela vérifie la relation pour tout  $i < \text{length}$  :

$$t[i] = 1 \iff i \text{ est un sommet appartenant à cet ensemble discret de sommets}$$

Au fait, cette présentation présente un faible coût mémoire d'un côté et de l'autre côté elle est efficace. Cela nous a aidé à filtrer les sommets qui nous intéressent tout en mettant à 1 les bons éléments ou à 0 si l'indice (le sommet) ne nous intéresse pas. La longueur est toujours le nombre de sommets du graphe étudié.

## 2.3 Conception d'un client

Dans le sujet, une interface nous est imposée pour les clients :

- `char const* get_player_name`

Cette fonction retourne le nom du joueur sous la forme d'une chaîne de caractères non modifiable. Puisqu'un fichier correspond exactement à un joueur et puisque le nom d'un joueur ne doit pas varier, on peut définir le nom du joueur manuellement dans cette fonction.

Cette fonction a été imposée afin de pouvoir personnaliser "l'apparence" des joueur avec un nom particulier, ce qui permet de les différencier visuellement lors de parties entre les différents joueurs (que ce soit lors de tests en local ou dans le cadre de la *ladder* organisé par le corps enseignant).

- **void initialize**

La fonction **initialize** est la première fonction du joueur appelée par le serveur et ne doit être appelée qu'une seule fois. Elle lui transmet les informations concernant l'état initial de la partie (identifiant du joueur, structure du plateau, coloration du plateau et couleurs interdites) et n'attend pas de réponse de la part du joueur, d'où l'absence de retour de la fonction.

La structure et la coloration du plateau de jeu fournis à la fonction sont des copies propres au joueur, afin qu'il ne puisse pas fausser l'information reçue par les autres joueurs. Puisque ces copies ont été allouées dynamiquement, il faudra libérer la mémoire qu'elles occupent lorsqu'elles ne seront plus utiles.

- **void finalize**

La fonction **finalize** est appelée par le serveur en fin de partie, et a pour but de libérer toute la mémoire allouée par le joueur au cours de la partie.

- **struct move\_t play**

Les fonctions précédentes (en particulier **initialize** et **finalize**) ne demandent pas de réelle réflexion algorithmique puisqu'elles relèvent uniquement de la programmation. En effet, leur principal objectif est d'assurer la bonne gestion de la mémoire. La fonction **play** prend pour argument le dernier coup de l'adversaire et doit retourner le coup choisi par le joueur. Cette fonction représente donc la stratégie d'un joueur et contient quasiment toute la difficulté algorithmique d'un client (si l'on ignore le fonctionnement des graphes, qui sont vus par le client sous la forme d'un type abstrait de données). Dans la suite de cette section, nous étudierons la conception de cette fonction.

Dans un jeu, une bonne stratégie passe tout d'abord par le respect de certaines règles. Les règles du jeu de Flood sont les suivantes :

1. Le client ne peut pas jouer une des couleurs interdites en début de partie.
2. Le client ne peut pas jouer une couleur qui n'étend pas son territoire. Concrètement, le client ne peut pas jouer une couleur X s'il ne possède aucun voisin de couleur X.
3. Le client ne peut pas annexer la zone de l'adversaire. Concrètement, le client ne peut pas jouer la couleur de l'adversaire si son territoire est voisin de celui de l'adversaire.



Afin de vérifier la première règle, il suffit de parcourir toutes les couleurs existantes et de vérifier pour chacune si elle fait partie des couleurs interdites, qui sont fournies au client lors de l'appel à la fonction `initialize`.

Pour que la deuxième règle soit vérifiée, une fonction de parcours de graphe indiquant le nombre de cellules qu'il est possible d'obtenir pour chaque couleur sera implémentée.

Enfin, la vérification de la troisième règle nécessitera une fonction de parcours de graphe pour déterminer si les zones des deux joueurs sont voisines. Si c'est le cas, il faut donc vérifier que le joueur ne joue pas la couleur de l'adversaire.

## 2.4 Conception du serveur

A partir d'un client, on peut renvoyer l'action que l'on effectue sur un graphe. Cependant on est très vite limité sur les fonctionnalités et on se rend compte que pour faire jouer deux clients entre eux, on a besoin d'une interface commune afin de vérifier le bon déroulement de la partie, le serveur.

Le serveur constitue le cœur même du projet. On peut séparer ses tâches en deux parties distinctes, l'initialisation et le déroulement de la partie.

Lors de la phase d'initialisation, le serveur doit s'occuper de charger les deux clients qui s'affrontent imposés comme étant des bibliothèques dynamiques dans le sujet. Pour réutiliser les fonctions de chaque joueurs, ces dernières sont rangées dans une structure de donnée listant pour chacun des joueurs les fonctions auxquels le serveur va faire appel. Les deux structures de données sont ensuite rangé ensemble dans un tableau afin d'accéder plus facilement aux joueurs.

Dans cette phase, le serveur doit également créer un graphe valide en prenant en compte les différentes options qui ont été rentré qui permettent de configurer la partie. Une fois le graphe crée, le serveur communique le graphe initial à chacun des joueurs et commence le déroulement de la partie.

Lors du déroulement de la partie, le serveur va demander à chacun des joueurs de jouer un coup à tour de rôle en leur communiquant le coup joué par leur adversaire. Après chaque coup renvoyé par un joueur, le serveur à plusieurs choses à vérifier:

- Dans un premier temps il va respecter si le coup du joueur respecte les différentes règles du jeu, à savoir ne pas jouer de couleur qui ne fait gagner aucune cases (`NO_COLOR` exclu) ou qui sont interdites et ne pas jouer la couleur de l'adversaire si les joueurs sont voisins. Si une des règles n'est pas respecté, la partie s'arrête et l'autre joueur est désigné vainqueur.
- Dans un second temps, si le coup est valide on vérifie si les deux derniers coups étaient `NO_COLOR` signifiant que les deux joueurs considèrent qu'ils ne peuvent pas jouer et donc la partie prend fin. Si la partie prend fin, on compte le nombre de

case de chaque joueur et on désigne le vainqueur ou une égalité si ils ont le même nombre de case.

- Si la partie n'a pas prit fin, le graphe du serveur est actualisé selon le coup que vient de choisir le joueur et on réitère le processus avec le joueur adverse.

## 3 Réalisation et implémentation

### 3.1 Étapes de réalisation

En début de projet, nous nous étions fixé pour objectif d'avoir un prototype de relation client/serveur qui compile. Pour cela, nous avons donc "implémenté" un serveur légèrement réduit (il ne vérifie pas la validité des coups) ainsi qu'un client minimal (ses fonctions ont des valeurs de retour constantes pouvant être NULL, NO\_COLOR. . .).

Ensuite, nous avons implémenté en parallèle les fonctions liées aux graphes ainsi que la vérification des coups par le serveur et par le client. Cela a abouti à une première version fonctionnelle du projet.

Enfin, nous avons développé de nouvelles fonctions sur les graphes afin de pouvoir coder un joueur avec une stratégie "glouton". À ce stade, notre projet est fonctionnel et propose un serveur ainsi que les trois clients évoqués précédemment.

### 3.2 Implémentation de la gestion des graphes

On a commencé à diviser le code en fonction élémentaires et non élémentaires. Les fonctions élémentaires sont les fonctions de base. Elle sert notamment à accéder à des champs de structures en temps constant ce qui nous sera bénéfique par la suite en terme de modularité et de compréhension de code de la part des autres équipes. Les sous-sections suivantes reprennent les idées du 3.1 et ce qu'on a dit maintenant avec plus de détails.

#### 3.2.1 Importation d'une librairie dynamique

L'utilisation de GSL est fondamentale en terme de complexité pour la réalisation du projet. En fait l'utilisation des structures de graphes contenant des champs matriciels de type GSL a un faible coût mémoire. C'est pour cela qu'on importe le dossier `/net/ens/renault/local/save/gsl-2.6/install` qui contient deux dossiers : `lib` et `include`. `lib` est un dossier qui rassemble les bibliothèques `gsl` qu'on utiliserait par la suite (les implémentations) et `include` contient tous les fichiers headers `.h` (les prototypes de fonctions avec leurs spécifications en commentaires). Pour utiliser les fonctions fournies par la bibliothèque `gsl`, on fait un `#include <gsl/file.h>` dans le fichier en question et pour compiler on ajoute l'option `-I` à `gcc` avec `-I<chemin_vers_les_headers_gsl>` qui sera dans ce cas : `/net/ens/renault/local/save/gsl-2.6/install/include` et après

avoir fini la deuxième étape de compilation, on fait l'édition de liens avec les différents fichiers et les implémentations des fonctions spécifiques à la manipulation gsl, soit avec l'option `-lm -lgsl -lgslcblas`.

Enfin, on indique à l'éditeur de lien le chemin où il trouvera ces bibliothèques, soit dans `lib` avec l'option `-L` et l'argument `/net/ens/renault/save/gsl-2.6/install/lib`.

### 3.2.2 Allocations dynamiques

Pour gérer efficacement la mémoire, nous utilisons les tableaux sous forme de pointeurs alloués dynamiquement, que nous pouvons redimensionner au besoin. Nous utilisons aussi un `struct size_t_list` comme type de retour de fonction pour éviter d'allouer trop de ressources en mémoire et simplifier les `free` par la suite. Pour la structure `graph`, on implémente une fonction `graph__free` qui s'occupe de faire un `free` pour toutes les ressources allouées avant avec la fonction `graph_empty` et qui va être appelé à chaque fois qu'on utilise une fonction qui utilise indirectement celle-ci tout à la fin comme par exemple la fonction `graph_t__get_copy`.

### 3.2.3 Réalisation du parcours en profondeur

- `size_t DFS_color(struct graph * g,  
enum color_t color,  
int * visited,  
size_t from);`

Cette fonction permet de modifier par effet de bord le tableau de booléens `visited`. Elle prend en quatrième argument un `size_t from` qui représente un sommet, une couleur spécifique puis elle va effectuer un parcours à partir de cette fonction pour déterminer tous les sommets qui possèdent cette couleur et dont le père si on remonte jusqu'à `from` est une suite de sommets de même couleur. `visited` à la fin de la fonction est un tableau de booléens où chaque `visited[i]` est soit 0 soit 1, c'est à dire soit il possède la même couleur soit non. Elle renvoie aussi le nombre de sommets possédant la même couleur dans ce contexte c'est à dire le nombre de 1 dans le tableau `visited`.

### 3.2.4 Implémentations des ensembles de sommets

- `struct size_t_list get_component(struct graph * g, size_t from);`

Au tout début du fichier, on a défini le type `size_t_list` comme une structure contenant la longueur de la liste ainsi que le tableau lui-même possédant cette longueur d'entiers positifs. En fait cette fonction reprend l'idée de la fonction précédente,

mais cette fois-ci, elle reprend le parcours en profondeur depuis le sommet `from`, récupère le nombre de sommets ayant la même couleur adjacents à la composante connexe et puisque le tableau `visited` qu'on a passé au début est modifié par effet de bord, alors le tableau stocké dans la structure `size_t_list` contiendra tous les indices `j` tel que `visited[j]` est différent de 0. On dispose actuellement d'une fonction qui utilise la fonction récursive précédente et qui renvoie depuis un sommet la composante connexe qui représente l'ensemble des sommets adjacents entre eux ayant la même couleur.

- `struct size_t_list get_component_outer_border(struct graph * g,  
size_t from);`

Cette fonction permet de récupérer depuis un sommet `from` après avoir récupéré la composante connexe en question qui représente le sous-graphe depuis un parcours en profondeur (depth for search) dont tous les sommets a la même couleur tous les sommets qui sont au bord de cette dernière. On utilise toujours la méthodologie de modification de tableau alloué par effet de bord pour avoir à la fin un tableau `t` tel que :

$$t[i] \neq 0 \iff i \text{ est un sommet à la bordure externe de la composante connexe de from}$$

- `size_t component_intersection_size(struct size_t_list component1,  
struct size_t_list component2);`

Cette fonction permet de façon très générale depuis 2 groupes de sommets (2 composantes pas forcément connexes donc pas forcément de même couleur) de retrouver la liste des sommets en commun. Par exemple, pour `component1` et `component2`, on aura comme retour de fonction un tableau de booléens `t` tel que  $t[i] = 1 \equiv i$  est un sommet commun entre les 2 composantes `component1` et `component2`.

- `struct size_t_list component_intersection(struct size_t_list component1,  
struct size_t_list component2);`

Cette fonction permet de retrouver depuis 2 tableaux de sommets `component1` et `component2` l'ensemble des sommets qui sont en commun. On retrouve toujours l'idée du tableau de booléen pour représenter un ensemble de sommets ayant une caractéristique en commun.

## 3.3 Réalisation du serveur

### 3.3.1 Gestion des arguments

Pour lancer une partie avec le serveur deux arguments sont obligatoires, les joueurs situés dans le dossier `install`. On cherche dans la ligne d'exécution l'emplacement où sont indiqués le chemin vers les bibliothèques dynamiques des joueurs que l'on conserve pour ensuite pouvoir les charger.

D'autres arguments sont optionnels et permettent de configurer le graphe et la partie du serveur :

- `-m`: cette option permet de choisir la longueur du graphe
- `-t`: cette option permet de choisir la forme du graphe, carré, en forme de H ou torique.
- `-c`: cette option permet de sélectionner le nombre de couleur qui compose le graphe.
- `-a`: cette option permet de spécifier la seed du graphe afin de répéter un aléatoire.
- `-r`: cette option permet de choisir le nombre de partie que l'on joue entre les deux joueurs.

Pour récupérer ces différents arguments on utilise `getopt` qui nous permet de détecter les options avec `-`. Pour chaque option présente, on remplace la valeur ou lettre par défaut du serveur par la valeur ou lettre donnée en option.

### 3.3.2 Chargement dynamique

Afin de lancer une partie, il faut que le serveur puisse avoir accès aux fonctions des joueurs. Les joueurs n'étant pas toujours les mêmes mais ayant le même nom de fonction pour le serveur, on utilise des bibliothèques dynamiques afin de charger les joueurs et de passer leur fonction au serveur.

Les joueurs sont contenus dans une structure de donnée construite de la façon suivante:

- `handle`
- `initialize`
- `play`
- `finalize`
- `get_player_name`

La variable `handle` contient le pointeur vers la zone mémoire où a été chargé l'instance du joueur tandis que les autres variables contiennent un pointeur vers les différentes fonctions utiles pour le serveur.

En utilisant `dlopen` sur la bibliothèque dynamique d'un joueur, on crée une instance étant une copie de la bibliothèque dynamique puis on nous renvoie un pointeur indiquant où la bibliothèque a été chargée. Grâce à ce pointeur, on peut ensuite chercher dans cette instance les fonctions qui nous intéressent dans la bibliothèque dynamique en conservant un pointeur vers ses fonction. La structure de donnée est ensuite envoyée au serveur qui la stocke dans un tableau afin de facilement pouvoir passer d'un joueur à un autre.

Une fois les parties finies, on libère l'espace mémoire des bibliothèques chargées en mémoire et on modifie le pointeur de fonction des structures de données vers `NULL` afin d'éviter des erreurs de segmentation.

### 3.3.3 Gestion d'une partie

Avant de rentrer dans la partie on définit le premier joueur comme un nombre aléatoire étant soit 0, soit 1 correspondant à la place du joueur dans le tableau de structure de données. Lorsqu'une partie est en cours, on répète une boucle d'instruction tant que aucun vainqueur n'a été désigné. La première instruction de la boucle consiste à passer au joueur suivant puis de demander le coup qu'il effectue, une fois cette action choisie on procède à plusieurs vérifications:

- On appelle la fonction `valid` qui prend en argument le joueur, son coup et le graphe du serveur. Si le coup est `no_color` cela est équivalent à passer son tour ce qui est un coup valide. S'il s'agit d'une autre couleur deux règles doivent être vérifiées. Cela peut être fait simplement en utilisant des fonctions du graphe afin de vérifier si le coup choisit fait bien gagner des cases au joueur actif mais également que le coup choisit n'est pas le même que le dernier coup adverse dans le cas où les deux joueurs se touchent. Si aucune de ses règles est enfreinte, le coup est valide, sinon l'autre joueur gagne la partie.
- Dans le cas où le coup des deux derniers joueurs est de passer leur tour (`NO_COLOR`) la partie est considérée comme finie et on appelle la fonction `compute_winner` qui à partir du graphe du serveur détermine le gagnant de la partie ou une égalité.

Tout au long de l'itération le graphe est affiché sur le terminal de commande sous la forme d'une matrice où les couleurs sont représentées par des nombres. Chaque étape du graphe est séparée par le symbole `#` ce qui permet de voir le déroulement de la partie. L'affichage du graphe n'a pas été choisi de façon aléatoire mais a été créé de façon à pouvoir être utilisé avec `SDL` afin d'avoir une meilleure idée du déroulement de la partie, comme on peut le voir sur la figure 2 ci-dessous `SDL` permet de mieux voir ce qui se

passé à chaque instant sans chercher à comprendre chaque nombre à l'écran. L'affichage a cependant été désactivé dans le cas où nous jouons plus d'une partie afin de prendre le moins de temps possible pour l'exécution des parties.

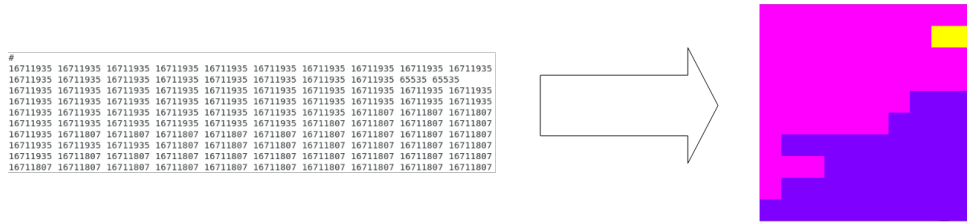


Figure 2: Affichage du graphe sur le terminal et sur SDL

## 3.4 Implémentations du client

### 3.4.1 Compilation en librairie dynamique

Dans le sujet, il nous est imposé de compiler les joueurs sous la forme de bibliothèques dynamiques. On utilise pour cela les options `-fPIC` et `-shared` lors de leur compilation. Cela permet au serveur de charger les clients pendant l'exécution à l'aide de la bibliothèque `dlopen`. De cette manière, il est possible d'effectuer plusieurs parties en une seule exécution du programme.

### 3.4.2 Les différentes stratégies réalisées

Sur la durée du projet, nous avons implémenté les "stratégies" de joueurs suivantes :

- `player_test.c`

C'est le joueur le plus basique imaginable qui ne provoque pas d'erreur de compilation : `get_player_name` renvoie `NULL`, `initialize` et `finalize` ne font rien, et `play` renvoie systématiquement `{NO_COLOR}`. Il a été utile en début de projet précisément pour avoir une première version du code qui compile. Il est cependant dangereux de l'exécuter car la gestion de la mémoire est absente, alors que la plupart des arguments de `initialize` doivent être libérés par le client même si l'allocation est faite par le serveur. C'est pourquoi nous avons par la suite retiré sa compilation de la commande `make`.

- `player_valid.c`

Il s'agit du premier vrai joueur d'après la définition évoquée durant la phase de conception : il gère correctement la mémoire et effectue des coups valides. Afin de gérer la mémoire, nous avons ajouté une structure `player` dans ce fichier. Ainsi, la fonction `initialize` stocke les différents arguments fournis par le serveur dans `current_player`, une variable globale de type `struct player` qui contient toutes

les informations sur le joueur et l'état de la partie. Pour cela, on fait appel à la fonction `graph__merge_from` qui rassemble la structure du graphe et sa coloration dans une même instance de type `struct graph`. Ensuite, la mémoire allouée pour ces champs est libérée lors de l'appel à `finalize` grâce à la méthode `graph__free` prévue à cet effet.

La stratégie de `player_valid` est simplement de toujours respecter les règles. Pour cela, nous avons implémenté une fonction `get_valid_moves` qui prend en argument un tableau `valid[]` ainsi que la couleur de l'adversaire et qui retourne un entier. Grâce à la fonction `number_vertices_winnable_from_player`, nous stockons les gains possibles pour chaque couleur dans un tableau `possible_wins` en début de fonction. Ce tableau sera utile dans la suite de cette fonction. Pour chaque couleur `c`, les trois règles du jeu Flood-filling sont vérifiées :

- Si le `c`-ième élément du tableau `forbidden` fourni par le serveur vaut 1, alors la couleur n'est pas valide.
- Si le `c`-ième élément du tableau `possible_wins` n'est pas strictement positif, cela signifie que le joueur n'étendrait pas son territoire en jouant cette couleur. Cette couleur n'est donc pas valide.
- Grâce à la fonction `is_touching_opponent`, nous pouvons savoir si les deux joueurs sont voisins. Cette information permet de vérifier la troisième règle : le coup n'est pas valide si le joueur joue la couleur de l'adversaire alors qu'ils sont voisins.

Après avoir défini le nombre `n` de coups valides et les avoir stockés dans le tableau `possible_moves[]`, un coup est choisi aléatoirement en sélectionnant l'élément `possible_moves[rand()%n]`. Dans le cas où aucun coup n'est autorisé (c'est-à-dire `n=0`), le joueur passe son tour en jouant `NO_COLOR`.

- `player_glouton.c`

Dans ce joueur, la gestion de la mémoire est similaire à celle de `player_valid.c` : les entrées sont toujours gérées grâce à la structure `player`. Cependant, le joueur adopte ici une stratégie légèrement plus évoluée que des coups aléatoires : en effet, il choisit à chaque tour la couleur qui étend le plus son territoire (tout en respectant les règles, bien entendu). Pour cela, nous avons implémenté une fonction `get_gains_by_color`, qui est une version étendue de `get_valid_moves`. En effet, cette fonction stocke les gains possibles dans un tableau `possible_gains` en tenant compte des règles (si une couleur `c` ne respecte pas l'une des règles du jeu, on a `possible_gains[c]=0`). Le joueur sélectionne ensuite la couleur `c` telle que `possible_gains[c]` soit maximal. Si tous les éléments de `possible_gains` valent 0, le joueur passe son tour en jouant `NO_COLOR`.



## 3.5 Documentation

Il a été choisi d'utiliser Doxygen: un système de génération de documentation automatique. Pour cela chaque fonction doit être précédée par un commentaire organisé et devant respecter une convention avec un symbole par ligne, `@brief`, `@param` et `@return`. Ces symboles permettent à Doxygen d'identifier les résumés, paramètres et les valeurs de retour des fonctions.

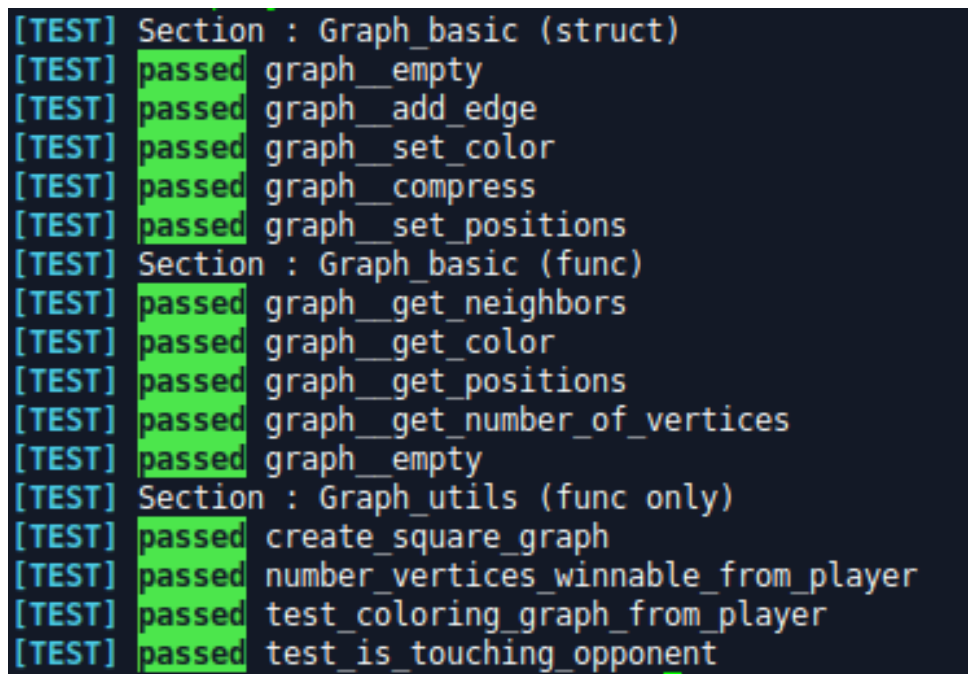
Puis il faut générer un fichier par la commande `doxygen -g` nommé `Doxyfile.in`, on modifie les variables qui servent à la documentation par la suite, notamment `EXTRACT_ALL`, `OPTIMIZE_FOR_C`, `RECURSIF` aux bonnes valeurs `YES` ou `NO`.

Pour finir il est possible de visualiser la documentation obtenue en latex (donc sous forme de pdf) ou html (donc sur navigateur sous forme de page web).

## 4 Validation

### 4.1 Les outils et le développement de tests

Une série de tests automatiques ont été développés, des tests unitaires pour l'ensemble des fonctions importantes/critiques sur les graphes et le serveur. Pour celles de l'implémentation du type abstrait graphe, des tests structurels (le programme a connaissance de la structure interne du type) et fonctionnels (le programme ne l'a pas) sont mis en place.



```
[TEST] Section : Graph_basic (struct)
[TEST] passed graph_empty
[TEST] passed graph_add_edge
[TEST] passed graph_set_color
[TEST] passed graph_compress
[TEST] passed graph_set_positions
[TEST] Section : Graph_basic (func)
[TEST] passed graph_get_neighbors
[TEST] passed graph_get_color
[TEST] passed graph_get_positions
[TEST] passed graph_get_number_of_vertices
[TEST] passed graph_empty
[TEST] Section : Graph_utils (func only)
[TEST] passed create_square_graph
[TEST] passed number_vertices_winnable_from_player
[TEST] passed test_coloring_graph_from_player
[TEST] passed test_is_touching_opponent
```

Figure 3: Exemple d'affichage sur le terminal d'une série de tests

La figure 3 montre la convention utilisée pour afficher et visualiser le résultat d'une série de tests. Cette convention est réalisée par les fichiers `testool.h` pour l'utilisation de

quelques macros de pré-compilation et `testool.c` pour un affichage adaptatif.

De plus l'utilisation de l'outil `gcov` permet de savoir quelles lignes de code des fichiers source ont été visitées, après exécution de l'ensemble (ou une partie) des tests.

## 4.2 Complexités des implémentations

Dans cette partie nous étudier les complexités temporelles et spatiales des différentes parties du programme. Cela permet de connaître d'approximer l'évolution du temps de calcul et de l'espace mémoire d'une exécution, en fonction de l'évolution des paramètres d'entrée.

Nous admettrons pour la suite qu'une allocation de mémoire de  $n$  octets est une opération de complexité temporelle en  $O(n)$  (car dans la majorité des cas l'espace mémoire alloué est entièrement initialisé par la suite) et de complexité spatiale évidente en  $O(n)$  aussi. Nous admettrons aussi que l'opération de libération d'un bloc mémoire alloué, l'allocation d'une "sparse matrix" GSL et la modification d'une de ses valeurs sont réalisées en temps constant ( $O(1)$ ).

Pour la suite nous noterons  $n$  la longueur et largeur de la grille de jeu (un graphe carré image de la grille de côté  $n$  possède  $n^2$  sommets).

Pour l'implémentation du type "graphe", chaque fonction définies dans la partie conception opèrent en temps et espace constant  $O(1)$ , à l'exception de `graph__empty` de complexités (temporelle et spatiale) en  $O(n^2)$  dû aux allocations. Les fonctions de copie d'un champ alloué du graphe sont aussi de complexités en  $O(n^2)$ .

Ainsi, les complexités issues des fonction implémentées à partir de ces dernières sont facilement calculables. Par exemple la création et coloration d'un graphe carré sont réalisées en temps quadratique selon  $n$ .

## 5 Conclusion

### 5.1 Résultat du projet

Lors de ce projet, il nous a été possible de réaliser les objectifs suivants :

- Créer un serveur fonctionnel qui permet de lancer des parties entre deux joueurs en prenant en compte plusieurs options, avec notamment la possibilité de jouer plusieurs parties à la chaîne.
- Réaliser un affichage du graphe en adéquation avec les contraintes de SDL afin d'avoir une meilleure visualisation des parties.

- Créer des clients respectant les contraintes du sujet ce qui leur permet d'être facilement réutilisable par d'autres serveurs.
- Créer des stratégies de joueur variées.
- Implémenter des fonctions sur les graphes adaptées aux besoins du serveur et du client basées sur des fonctions des fonctions élémentaires.

Ce projet nous a permis de renforcer nos connaissances en C et de mettre en pratique des choses que nous avons vu en cours comme la compilation et l'utilisation de bibliothèques dynamiques, gérer des allocations et de la mémoire dynamique avec `dlopen` mais également de mieux structurer et optimiser le code. Le projet étant à faire par groupe de 4, cela nous a aussi permis de nous rendre compte de l'importance de la documentation ou encore du test des fonctions, mais également et surtout l'importance et la force de travailler en équipe.

## 5.2 Idées d'amélioration du projet

Pour améliorer le projet, plusieurs idées à court et long terme pourraient être mises en place:

- Tout d'abord nous aurions pu finir les options comme le nombre de couleur, la forme ou encore le nombre de couleur interdite qui ne sont actuellement pas pris en compte dans les paramètres d'initialisation
- Nous aurions également pu faire un client minmax qui semble être la meilleure stratégie possible pour ce jeu (sans prendre en compte les contraintes de temps et d'espace) ou même rechercher d'autres implémentations efficaces et surtout moins gourmandes en temps de calcul. Un client utilisant le principe de machine-learning par exemple.
- À plus grande échelle, nous aurions également pu réfléchir à changer les objectifs du jeu en changeant la règle pour le gagnant de "celui qui a le plus de case" à "celui qui prend la couleur de l'autre en premier" incitant donc à repenser les règles et les conditions de victoire du jeu mais également repenser de nouvelle stratégie de jeu.