



ENSEIRB-MATMECA

FILIÈRE INFORMATIQUE, 1 ÈRE ANNÉE

---

## RAPPORT : Projet Flood Filling

---

**Auteurs :**

- Abibi Aymane
- El bousty Badreddine
- Naami Anas
- Soufary Farouk

## Contents

# 1 Introduction

Dans le cadre du projet de programmation impérative du deuxième semestre, nous étions amenés à programmer en langage C afin d'atteindre l'ultime objectif du sujet, à savoir, faire tourner une partie du jeu Flood Filling entre deux joueurs. Des contraintes nous ont été données, mais aussi bon nombre de libertés. Parmi ces contraintes, nous citons par exemple l'utilisation d'une interface serveur-client, l'utilisation des bibliothèques dynamiques pour représenter les joueurs et la représentation des grilles du jeu par des graphes en utilisant d'abord des matrices creuses de la bibliothèque GSL comme des matrices d'adjacence pour ensuite les transformer en des matrices factorisées CSR.

Afin d'atteindre notre objectif dans ce projet, nous étions obligé de répartir les tâches entre les différents membres du groupe. Le travail a été décomposé en 4 sous objectifs essentiels à savoir :

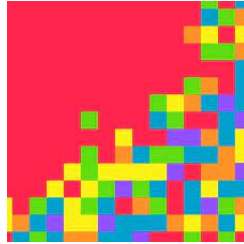
- Créer le support du jeu (graphes, coloration)
- Créer un serveur faisant tourner la partie
- Créer des bibliothèques dynamiques pour les joueurs
- Propager une couleur choisie par un joueur tout en respectant les règles du jeu
- Implémenter des stratégies pour les joueurs

Ce compte-rendu présente ainsi un résumé recouvrant le plus possible les réalisations majeures et les choix qui ont été discutés, décidés et implémentés par les différents membres du groupe.

# 2 Description générale du jeu Flood Filling

La partie commence avec deux joueurs, leurs positions de départ sont différentes. Les couleurs choisies par le joueur lui font nécessairement gagner des cases (augmenter son score), et n'appartiennent pas à un ensemble de couleurs interdites défini au début de la partie.

La partie se termine si les deux joueurs choisissent de passer leur tour successivement (ils choisissent de jouer NO\_COLOR). Le gagnant est celui qui possède le plus de cases, et un joueur ne respectant pas l'une de ces règles est déclaré perdant.



Les objectifs de ce projet sont détaillés ci-dessous :

- Pouvoir implémenter un ensemble de règles qui font jouer deux joueurs une partie, ainsi que des contraintes limitant les mouvements de chaque joueur.
- Puis on doit pouvoir implémenter un ensemble de clients qui jouent d'une façon automatique sur leurs propre grille, et qui peuvent inter-agir entre eux à l'aide d'un serveur contenant la grille originale et qui donne à chacun son coup.
- Implémenter un ensemble de règles et de stratégies et des algorithmes permettant la propagation des couleurs en prenant en considération les couleurs interdites pour chaque joueur : c'est la partie où on doit gérer les conflits.
- Effectuer un ensemble de tests permettant de valider nos algorithmes et de faire jouer nos joueurs avec des stratégies différentes.

### 3 Architecture du code et démarches suivies

### 3.1 graphe de dépendances

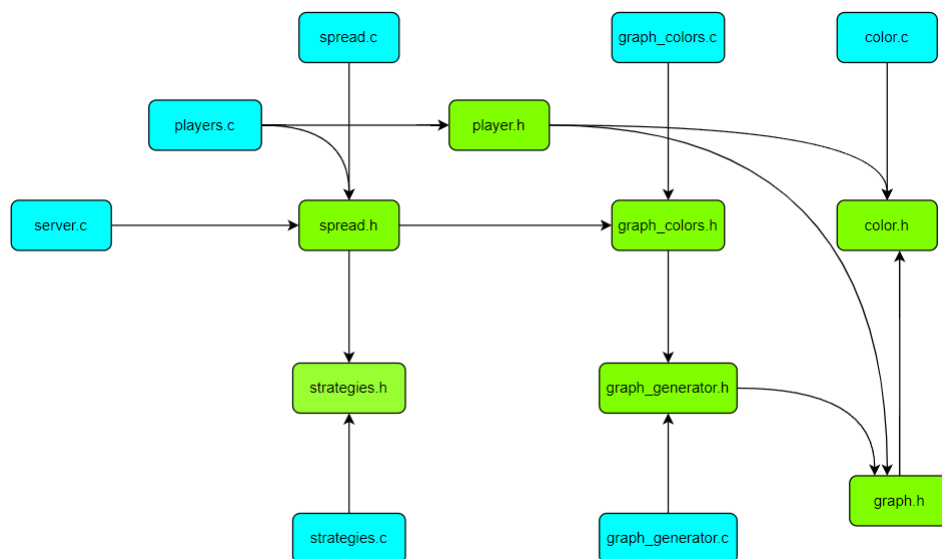


Figure 1: Graphe de dépendances

### 3.2 Générateur de graphes et couleurs

#### Générateur de graphes:

Jouer au Flood-Filling nécessite d'implémenter un certain nombre de graphes. Quatre types ont été implémenter : **SQUARE**, **HGRAPH**, **DONUT**, **TORUS**.

Tout d'abord, les graphes sont définis en tant qu'une structure de données contenant un nombre de sommets **n**, une matrice d'adjacence de taille **n.n** et un tableau contenant la position de départ de chaque joueur :

```

struct graph_t {
    size_t num_vertices;
    gsl_spmatrix_uint* t;
    size_t positions[NUMPLAYERS];
}

```

Les valeurs de la matrice d'adjacence sont des binaires, si les sommets  $i$  et  $j$  sont alors  $t[i,j] = 1$  sinon  $t[i,j] = 0$ .

### Exemple: SQUARE GRID

$$\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array} \Rightarrow \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Après avoir implémenté les types cités précédemment, la génération des graphes de notre choix se fait à l'aide de la fonction **graph\_init** où on alloue dynamiquement la mémoire nécessaire. La bibliothèque **GSL** offre la

```

struct graph_t *graph_init(int num_vertices,
                           enum graph_type_t type);

```

possibilité de créer des matrices creuses **COO** et **CSR**. Il s'avère qu'il est plus coûteux d'itérer sur une matrice **COO** que celle de type **CSR**, donc la fonction **graph\_compressed\_init** transformera la matrice du graphe retourner par **graph\_init** en une matrice **CSR**, ainsi on pourra la manipuler avec beaucoup plus de fluidité.

### Générateur de couleurs:

En se limitant que sur huit couleurs définit par **color\_t**:

```
enum color_t {
    BLUE=0, RED=1, GREEN=2, YELLOW=3,
    ORANGE=4, VIOLET=5, CYAN=6, PINK=7,
    MAX_COLOR=8, NO_COLOR=-1
};
```

De la même manière, la génération des couleurs nécessite des types de génération; RANDOM, CYCLIQUE, SYMÉTRIQUE.

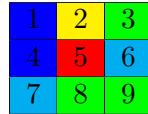


Figure 2:  
Random

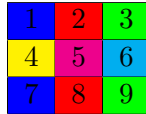


Figure 3:  
Cyclique

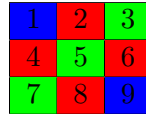


Figure 4:  
symétrique

Ensuite, on associe à chaque graphe un tableau de couleurs qui sera généré en fonction du type choisi grâce à la fonction `colors_init`.

```
enum color_t * colors_init(struct graph_t * graph,
                           enum color_type_t type_color,
                           int num_colors);
```

### 3.3 Implémentation des joueurs

Les joueurs sont implémentés comme un ensemble de fonctions qui gère l'interaction avec le serveur. Dans le fichier de chaque joueur, on déclare globalement, un `id` qui désigne le joueur au serveur, un graphe, un tableau de couleur et un tableau de couleurs non permises qu'on appelle tableau de `forbidden`. la copie du tableau de couleur va être mise à jour au fur et à mesure, de l'évolution de la partie.

Chaque joueur dispose de quatre fonctions :

- `player_get_name` : retourne le nom du joueur.
- `initialize` : prend en paramètre un `id`, un graphe, tableau de couleurs, tableau de `forbidden`. Cette fonction copie les paramètres de la

fonction dans les variables définies globalement dans le fichier. Ces paramètres constituent un environnement locale du joueur permettant de suivre la progression du jeu, et de fournir les coups qu'il veut jouer.

- **play** : prend en paramètres le coup joué par l'adversaire pour mettre à jour son propre tableau de couleur et calculer selon sa stratégie un coup qui va être retourner par la fonction.
- **finalize** : libère le graphe et le tableau de couleurs donnés aux joueurs par **initialize**.

### 3.4 Serveur

L'implémentation du jeu nécessite une interaction serveur-clients. Le serveur connecte les joueurs, assure le bon déroulement du jeu, garantie qu'aucun joueur ne triche, et met à jour la grille du jeu pour chaque joueur.

Le serveur alloue dynamiquement un graphe, un tableau de couleurs, ainsi qu'un tableau **forbidden** des couleurs interdites aux joueurs. Le serveur alloue une copie de graphe, une copie de tableau de couleurs, et une copie de tableau **forbidden** à chaque joueur. Ces copies seront passés aux joueurs en utilisant la fonction **initialize**.

Les joueurs appliquent leurs coups sur les copies fournies par le serveur et informe le serveur du coup choisi par le biais de la fonction **play**, si le coup vérifie les 3 conditions: n'appartient pas au tableau **forbidden**, ne connecte pas les deux territoires et augmente le territoire du joueur, le serveur applique ce coup sur la grille principale et l'envoie au prochain joueur.

Les clients sont chargés sous forme de bibliothèque dynamique et contiennent les mêmes noms de fonction, l'édition des liens qui se fait au moment de compilation n'est pas suffisante pour gérer les tours de rôle pour chaque joueur. La gestion des tours ainsi que l'interaction entre le serveur et les clients se fait à l'aide des fonctions de la bibliothèques **dlopen**, qui ouvre et prépare une bibliothèque partagée pour l'utilisation. Dans le cas où les deux joueurs, deux bibliothèques sont utilisées, on les charge dynamiquement en utilisant un tableau de longueur **NUM\_PLAYERS**, en occurrence 2:

Le *handle* retourné par chaque **dlopen**, est un chargement dynamique de fichiers passé en paramètre, pour importer donc les fonctions dans un



```
void handle[NUMPLAYERS] = {dlopen("libplayer1.so")
                           ,dlopen("libplayer2.so")}
```

Figure 5: Tableau `handle` qui ouvre et prépare les bibliothèques utilisées

*handle*, on utilise la fonction `dlsym`.

Ce mécanisme de chargement des fonction au besoin permet au serveur de bien gérer les tours de rôle, en accédant à chaque `handle` dans la tableau ??.

La partie se termine si le serveur n'arrive pas à propager la couleur choisie par un joueur, ou si les deux joueurs choisissent de passer leur rôle successivement. Le score est calculé à la fin de la partie pour déterminer le joueur gagnant.

Après la fin de la partie, on libère le graphe, le tableau de couleur, ainsi que le tableau `forbidden`, on libère aussi les copies données à chaque joueur en utilisant la fonction `finalize` puis on arrête le chargement des bibliothèques en utilisant la fonction `dlclose`.

## 4 Algorithmes

### 4.1 Propagations de couleurs

Afin d'appliquer le choix du joueur dans l'interface du jeu, nous étions amenés à implémenter une fonction dont l'objectif est de propager une couleur à partir de la position initiale du joueur en question tout en tenant compte de l'adjacence des cellules. En d'autres mots, propager une couleur à partir de la position initiale d'un joueur consiste à colorer l'ensemble des cellules constituant son territoire acquis. Le territoire conquis par un joueur est défini par l'ensemble des cellules doublement adjacentes (adjacence physique, et adjacence de couleur). Dès lors, les termes suivants seront utilisés pour décrire les aspects algorithmiques des fonctions que nous avons implémentés.

- **Voisins d'une cellule :** Les voisins d'une cellule sont les cellules qui partagent un côté avec elle. (Exemple : les voisins de la cellule 0 sont {1,5}, les voisins de la cellule 6 sont {1,5,7,11})

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

- **Adjacence physique** : Deux cellules sont physiquement adjacentes lorsqu'elles sont voisines. (Exemple : Les cellules 0 et 5 sont physiquement adjacentes, les cellules 0 et 6 ne le sont pas)
- **Adjacence double** : Deux cellules sont doublement adjacentes si elles sont physiquement adjacentes et ont la même couleur. (Exemple : les cellules 0 et 1 sont doublement adjacentes, les cellules 0 et 6 ne le sont pas, les cellules 0 et 5 ne le sont pas).
- **Partie connexe** : Est un ensemble de cellules connectées par double adjacence. (Exemple : les deux ensembles  $\{0,1,6,11,16\}$  et  $\{10,15\}$  sont deux parties connexes de tailles différentes)
- **Territoire d'un joueur** : Est la plus grande partie connexe contenant la position initiale du joueur. (Exemple : le territoire du premier joueur est  $\{0,1,6,11,16\}$ , le territoire du deuxième joueur est  $\{24\}$ )
- **Frontière d'un joueur** : Est l'ensemble des cellules physiquement adjacentes au territoire du joueur. (Exemple : la frontière du premier joueur est  $\{5,10,15,21,17,12,7,2\}$ , celle du deuxième joueur est  $\{19,23\}$ )
- **Frontière en profondeur** : Est l'ensemble des parties connexes dont l'une des cellules fait partie de la frontière du joueur. (Exemple : la frontière en profondeur du premier joueur est l'ensemble:  $\{\{5\},\{10,15\},\{21,22\},\{17\},\{7,12,13,14\},\{2\}\}$ )

La fonction `spread_colors` que nous avons implémenté permet à un joueur d'étaler son territoire en changeant la couleur de ce dernier par la couleur passée en argument et c'est dans cette action que nous retrouvons l'utilité de la double adjacence discutée ci-dessus.

Nous mentionnons que `spread_colors` représente une fonction intermédiaire servant à préparer les arguments d'une fonction récursive `visit_and_color`.

Ceci permet d'exploiter l'allocation automatique pour les variables auxiliaires nécessaires au processus de la propagation, cette méthode a été utilisée pour plusieurs autres fonctions afin d'éviter les déclarations des variables globales dans le serveur.

Voici un pseudo code décrivant le fonctionnement de `visit_and_color` :

```
visit_and_color(cell: cellule, c: couleur du territoire,
               colors: tableau de couleurs,
               nv: nouvelle couleur)

    T = les voisins de cell
    pour chaque voisin x dans T
        si colors[x]==c
            colors[x] = nv
            visit_and_color(x, c, colors, nv)
```

La fonction `spread_color` appelle la fonction décrite ci-dessus uniquement lorsque la couleur choisie par le joueur (donnée en argument) ne figure pas dans la liste des couleurs interdites. En pratique, la fonction renvoie 0 si la couleur choisie par le joueur est interdite et ne propage rien, dans le cas contraire la propagation est faite et la fonction renvoie 1.

Notons bien que le nombre d'appels récursifs lors de la propagation de couleur est limitée malgré le fait que la relation du voisinage est symétrique (potentiel d'une infinité d'appels entre deux cellules voisines lors d'un parcours en profondeur) car lors de chaque parcours nous changeons la couleur d'une cellule du territoire du joueur et par conséquent, celle-ci ne vérifiera pas la condition ( couleur de la cellule == couleur de la position initiale ) et donc n'engendrera pas un appel récursif.

Voici un exemple de 2 tours du jeu décrivant le fonctionnement de la fonction de propagation.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

(a) L'état de la grille initialement

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

(b) Le 1er joueur choisit bleu

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

(c) Le 2eme joueur choisit rose

## 4.2 Calcul du score

Comme mentionné avant, nous avons intérêt à calculer le score de chacun des deux joueurs à la dernière étape de la partie du jeu pour déclarer le joueur gagnant si personne des deux ne choisit une couleur interdite.

Par conséquent, nous avons implémenté une fonction `score_of_player` prenant en argument le graphe du jeu, l'ensemble des couleurs et un identifiant de joueur. Pour les mêmes raisons citées dans la partie précédente, cette fonction est une fonction intermédiaire qui alloue automatiquement les arguments nécessaires au calcul du score. Elle appelle la fonction `calculate_score` qui a un comportement assez proche de la fonction de propagation puisqu'elle prépare les voisins de la cellule qu'elle prend en argument et incrémente une variable locale déclarée dans la pile de la fonction intermédiaire via son pointeur pris en argument.

De plus, la similarité ne s'arrête pas à cet aspect, puisque le calcul se fait par parcours en profondeur suite à un appel récursif sur chacun des voisins lorsqu'il appartient au territoire du joueur.

Cependant, cette fois-ci, nous parcourons les cellules sans modifier leurs couleurs et par conséquent, nous devons marquer les cellules visitées pour limiter les appels récursifs, ceci est fait en passant à la fonction récursive un tableau "grey" de booléens. (`grey[i]!=0` signifie que la cellule `i` a été visitée)

## 4.3 Mise à jour des couleurs interdites

Pour chacun des joueurs, en addition aux couleurs interdites fixées avant le début de la partie, il existe bien d'autres couleurs qui peuvent leurs être interdites et qui varient selon l'état de la grille du jeu comme mentionné

dans la partie de description du jeu. Nous avons choisi de représenter ces couleurs par un tableau appelé forbidden dynamique et qui ressemble dans son implémentation au tableau forbidden invariable, une structure `color_set_t`:

```
struct color_set_t = {  
    char t[MAX_COLOR];  
}
```

La structure `color_set_t` contient un tableau de `char` qui représente la première lettre de la couleur. La quantité `MAX_COLOR` représente le nombre total des couleurs possibles et qui a été fixé en 8 couleurs.

Chacun des joueurs n'a pas le droit de choisir une couleur qui ne lui apporte aucun point, ceci veut dire que nous avons besoin de retrouver la frontière de chacun des deux joueurs. Pour ce faire, nous avons implémenté une fonction `get_Ctab_neighbors` qui prend en argument le graphe, le tableau de couleurs, l'identifiant du joueur, la couleur du territoire du joueur ainsi que deux tableaux, l'un qui compte le nombre d'occurrences (initialement rempli par des 0 et dont la taille est égale au nombre de couleurs) d'une certaine couleur dans la frontière du joueur et un deuxième qui sert à marquer les cellules déjà visitées. Cette fonction prépare les voisins de la cellule en entrée dans un tableau, puis pour chacun des voisins, vérifie s'il a la même couleur du territoire (appartient à la frontière) ou non. Dans le cas où il appartiendrait à la frontière, il est marqué comme visité puis un appel récursif est fait en le prenant en argument (parcours en profondeur). Dans le cas contraire, si sa couleur est désignée par l'enter `i`, la fonction incrémente la `i`-ème position du tableau compteur d'occurrences des couleurs.

Ceci donc nous permet d'identifier une partie des couleurs interdites en cherchant le complémentaire de l'ensemble des couleurs de la frontière retrouvé par `get_Ctab_neighbors`.

Il existe un deuxième cas de couleurs interdites, c'est le cas où les deux territoires des joueurs sont connectés (c-à-d au moins l'une des cellules de la frontière du premier joueur est adjacente à au moins l'une des cellules de la frontière du deuxième joueur). Dans ce cas, chacun des joueurs n'a pas le droit de choisir la couleur de son adversaire. Voici une figure décrivant ce cas :

Nous remarquons que la cellule 16 du territoire du premier joueur est

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

voisine à la cellule 21 du territoire du deuxième joueur, à cette étape, la couleur orange est interdite au joueur 1 et la couleur rouge est interdite au joueur 2.

Pour détecter une occurrence d'un tel cas dans nos parties du jeu, nous avons créé un ensemble de fonctions, chacune de ses fonctions avait sa propre utilité pour garantir le résultat final.

L'une des fonctions est la fonction `relatedComponentSmart` prenant en argument un graphe, le tableau de couleurs de la grille, une cellule, et un tableau vide. Cette fonction est responsable du remplissage du tableau en argument par la partie connexe du graphe qui contient la cellule en argument. Cette fonction est utilisée pour récupérer les frontières des deux joueurs puis les stocker dans deux tableaux déclarés automatiquement dans une fonction intermédiaire qui s'occupe de la phase suivante

Une fois que nos deux frontières sont fournies, il ne reste plus qu'à vérifier si il existe une cellule de la frontière du premier joueur qui est physiquement adjacente à l'une des cellules de la frontière du deuxième joueur. Pour ce faire, nous utilisons une troisième fonction prenant en argument le graphe du jeu et les deux tableaux contenant les deux frontières. Cette fonction effectue une première boucle sur l'ensemble des cellules de la première frontière et pour chacune de ces cellules, elle récupère les voisins dans un tableau local de taille 4. Ensuite, elle entre dans une deuxième boucle qui itère sur les cellules de la deuxième frontière, et pour chaque élément de cette frontière elle itère une dernière fois sur les éventuels 4 voisins de la cellule du premier joueur en question pour tester son adjacence à la cellule du deuxième joueur. Elle renvoie à la fin de son appel 1 lorsqu'il y a adjacence (les frontières sont connectées) sinon elle renvoie 0.

Grâce à la fonction décrite ci-dessus, nous avons pu détecter le cas de connexion entre les frontières des joueurs dans notre fonction de mise à jour des couleurs interdites et ainsi, nous ajoutons la couleur de l'adversaire dans le tableau forbidden dynamique.

### **Complexité des algorithmes :**

Nous avons remarqué que presque toutes les fonctions que nous venons de décrire s'appuient sur une même méthode algorithmique : le parcours en profondeur d'un graphe. Dans notre cas, le parcours ne se fait que sur les graphes représentés par les territoires des joueurs et qui sont d'ailleurs rarement de l'ordre du nombre de cellules (Ça ne peut se produire que si l'un des deux joueurs ne fait que passer sont tour, ie. choisir NO\_COLOR). Dans le cas de la fonction `visit_and_color` par exemple, le nombre de sommets concernés par un parcours est dans le pire des cas à peu près égale au nombre des cellules de la grille qu'on notera  $n$ . Dans la fonction de propagation, les instructions se limitent à la récupération des voisins de la cellule de l'entrée, chose qui est faite en un temps constant grâce à la factorisation des matrices d'adjacence, et le changement de couleur de la cellule (ou non) qui se fait également en temps constant. Sachant que le nombre d'appels maximale est égal à  $n$ , nous déduisons que la complexité temporelle de la fonction de propagation est linéaire. En addition, puisque le calcul du score ainsi que la mise-à-jour des couleurs interdites suivent à peu près la même démarche algorithmique que celle de la propagation des couleurs, nous déduisons que la complexité temporelle de ces fonctions est également linéaire.

## **4.4 stratégies**

### **Le joueur aléatoire :**

Représente notre premier joueur implémenté, il joue une couleur aléatoire non-présente dans le tableau des couleurs non permises (Forbidden\_colors), si aucune couleur ne lui est permise, il joue NO\_COLOR.

### **Couleur maximale :**

C'est une stratégie qui permet au joueur d'effectuer un coup en se basant sur le tableau `get_Ctab_neighbors`, en choisissant la couleur qui a le plus d'occurrences dans la frontière du joueur. .

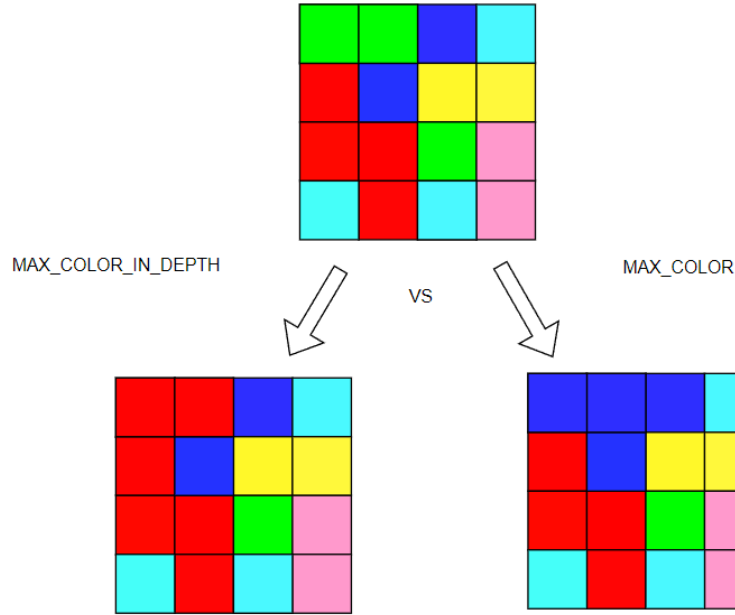


Figure 7: Max\_Color\_In\_Depth vs Max\_Color

### Couleur maximale en profondeur :

En un coup, une couleur peut mieux étaler le territoire du joueur sans qu'elle ne soit prédominante dans la frontière.

C'est pour cela qu'on implémente une fonction `get_Ctab_neighbors_smart` qui fournit un tableau contenant le nombre d'occurrences de chaque couleur en profondeur à partir de la frontière du joueur.

Pour le joueur 1 dans la figure ??, `get_Ctab_neighbors_smart` nous fournit un tableau avec la couleur rouge 4 fois itérée et la couleur bleue 2 fois itérée tandis que `get_Ctab_neighbors` donne un tableau d'occurrence des couleurs: (bleu=2,rouge=1), le coup permettant de mieux propager son territoire est fourni par le premier tableau .

Cette fonction utilise un raisonnement similaire à `get_Ctab_neighbors`, un parcourt en profondeur de la même manière pour obtenir les cellules de la frontière, il suffit ensuite d'appliquer la fonction `relatedComponentSmart` sur chaque élément pour obtenir les parties connexes de chaque composante dans la frontière.



Deux cellules de la frontière appartenant à la même partie connexe risquent d'être prises en compte deux fois, pour y remédier, on marque chaque cellule visitée de la partie connexe fournie par `relatedComponentSmart`. Avec les cellules de la frontière\_en\_profondeur, on obtient le nombre d'occurrences de chaque couleur qui nous donne le `MAX_COLOR_IN_DEPTH`.

### **Best moves\_in\_advance :**

En précisant le nombre (`num`) de coups qu'on veut prévoir en avance, cet algorithme permet d'avoir les coups qui maximisent le score après `num` coups de le début de partie, c'est la fonction `moves_in_advance` qui permet d'aboutir à ce résultat.

Le principe de cette stratégie est de simuler des éventuelles séquences de `num` coups permis dans une grille virtuelle pour ensuite choisir la meilleure combinaison maximisant le territoire du joueur.

On appelle la fonction avec le tableau de couleur initiale du joueur et les couleurs qui lui sont permises, un tableau `move_dynamic` de la taille des coups attendus `num`, qui sera remplie à fur à mesure avec les coups de taille `num` qui permettent au joueur de conquérir un maximum de cellule au début du jeu.


Un autre tableau de coups (`move_static`) sera fourni pour stocker les coups intermédiaires de taille inférieure à `num` et à chaque appel de la fonction, on effectue des copies du tableau de couleur et du tableau des coups `move_static`, on les modifie puis on prend pour le prochain appel de fonction ces nouveaux paramètres qui décrivent le nouvel état après avoir propagé une couleur permise sur la grille "virtuelle" du joueur.

On calcule le score de chaque combinaison de coup de taille `num` et on affecte à `move_dynamic` celle qui fournit le plus grand score.

La taille des grilles est prise en compte dans le choix de `num`. Par exemple pour une grille de taille 10x10 on prend `num = 5`. Cette valeur nous permet de toujours prendre l'avantage au début de la partie. Le joueur termine ensuite la partie avec l'algorithme du `MAX_COLOR_IN_DEPTH`. Cette stratégie nous a permis de décrocher la première place au ladder. En sachant que le joueur implémentant la stratégie du `MAX_COLOR_IN_DEPTH` est 5<sup>eme</sup> sur ce ladder.

### **Complexité de la fonction moves\_in\_advance :**

Cette complexité est illustrée sur la figure ??, on suppose que que 4

Flood score ladder for the 16/05/2022							
#	Repository	Team name	Player	SHA	Score	Crashes	Logs
1	projetss6-flood-15539		libplayer4.so	fd773d1	181	3	<a href="#">Games</a>
2	projetss6-flood-15541	Noah's Ark	dove.so	bb60717	172	8	<a href="#">Games</a>
3	projetss6-flood-15560		libminimax.so	b8f1ba9	167	8	<a href="#">Games</a>
4	projetss6-flood-15549	Babus Fan Club	lib_smart_client.so	fce99bb	149	4	<a href="#">Games</a>
5	projetss6-flood-15539		libplayer5.so	fd773d1	148	5	<a href="#">Games</a>
6	projetss6-flood-15524		libplayer2.so	eb51602	140	2	<a href="#">Games</a>
6	projetss6-flood-15531		libplayer2.so	0b7b4bb	140	4	<a href="#">Games</a>
8	projetss6-flood-15556		player1.so	81a5c06	132	7	<a href="#">Games</a>
9	projetss6-flood-15527	EGH 	client.1.so	fe9db3a	131	6	<a href="#">Games</a>
10	projetss6-flood-15334	The Deluge	pichu.so	57dbf74	128	2	<a href="#">Games</a>

couleurs existent dans le jeu bleu, jaune, rouge, vert, donc il a le droit d'en choisir trois à chaque tour.

Au début, le joueur possède la couleur bleue puis en prenant dans notre exemple  $num = 2$ , on obtient sur les 2 coups en avance qu'il prévoit,  $3^3$  appels de la fonction spread et  $3^2$  appel de la fonction score pour choisir la combinaison de deux coups avec le meilleur score .

Donc la complexité de moves\_in\_advance est dans le pire des cas si on a toujours  $Max\_Num\_colors - 1$  sur la frontière du joueur, en posant

$M = (Max\_Num\_colors - 1)$  on obtient pour la complexité:

$$M^{num+1} \times complexity(spread) + M^{num} \times complexity(score)$$

Donc la complexité avec N le nombre de sommets est :  $O(M^{num+1} \times N)$

## 5 tests et validation

La partie la plus importante du projet concerne les tests effectués pour valider les résultats des fonctions. En particulier, celles qui jouent un rôle fondamental dans la propagation des couleurs et l'exécution de nos stratégies. Afin de vérifier que notre programme fonctionne correctement et dans un premier temps, on a réalisé des parties entre différentes stratégies et d'un autre côté, nous avons effectué des tests unitaires pour les fonctions importantes en prenant compte des cas limite qui puissent poser des problèmes.

### 5.1 Exemples de partie entre stratégie

Nos joueurs p1, p2 et p3 utilisent des stratégies différentes et d'après nos implémentations p1 est censé gagner contre p2 et p3, et p2 doit gagner contre

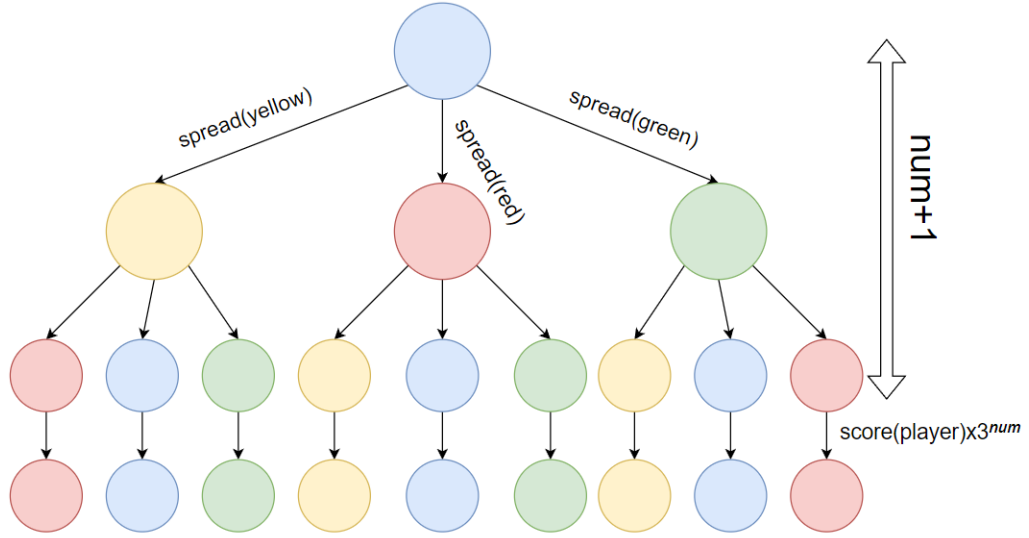


Figure 8: complexité de moves\_in\_advance dans un cas particulier

p3.

Alors, afin de vérifier cette hiérarchie, prenant comme test les parties disponibles sur le ladder :

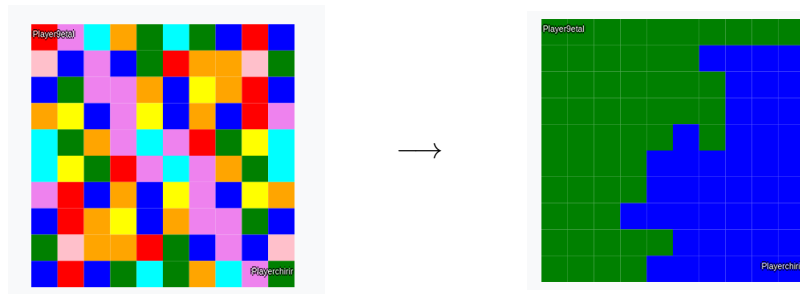


Figure 9: p1 vs p2 : 56 vs 44

En effet, le score de chaque partie et l'écart entre les scores de chaque joueur montre effectivement la hiérarchie entre les stratégies en terme de puissance.

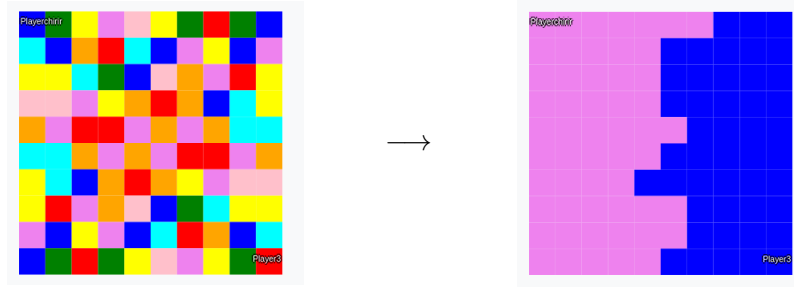


Figure 10: p2 vs p3: 54 vs 46

## 5.2 Tests

Les tests ont été choisis en prenant en considération les différents types de graphes, de colorations, de cas limite de la propagation de couleurs et des calculs d'occurrences aux frontières. La vérification des résultats est effectuée grâce à la fonction `assert` de la bibliothèque `assert.h`.

- Les premiers tests concernent la génération des graphes et des couleurs dans le fichier `test_graph.c`. Pour cela, on a effectué quatre tests, un pour chaque type de graphe, en choisissant un type de générateur de couleurs de notre choix pour chaque test.
- Les tests suivants sont ceux de la fonction `get_Ctab_neighbors`. Elle se distingue par son utilisation dans une stratégie et aussi par son rôle dans la propagation des couleurs. En fixant la coloration cyclique, ses fonctions de tests se basent sur deux choses : la position du joueur (0 ou  $n^2 - 1$ ) et le type du graphe.
- Les tests qui permettent de propager les couleurs sont effectués de la même manière que ceux de `get_Ctab_neighbors`, en ajoutant des tests sur les changements des tableaux `forbidden` avant et après la propagation.

## 6 Conclusion

### 6.1 Résultat final

Durant ce projet, nous étions capables de créer le jeu de flood-filling en simulant une partie entre différents joueurs et stratégies de jeux.

Cependant, il était possible d'améliorer certains points et de perfectionner

nos algorithmes ainsi que d'ajouter d'autres formes de graphes en ajoutant des obstacles (une autre couleur, le noir par exemple) bloquant la propagation des joueurs.

Néanmoins, quelques points ont posé plusieurs problèmes:

- Le problème de considérer et de gérer des couleurs interdites dynamiques et statiques, des couleurs qui sont interdites durant toute la partie et des couleurs qui changent après chaque propagation.
- Le débogage des erreurs qui parfois n'apparaissent que lors des parties avec d'autres serveurs.

## 6.2 Bilan personnel

Ce projet nous a permis de mettre en application nos connaissances en C (création d'un `Makefile`, des bibliothèques partagées et le débogage avec `GDB`), ainsi que nos connaissances algorithmiques en général (avec l'implémentation de quelques algorithmes de graphes). Nous avons aussi appris à travailler en équipe sur un projet de programmation tout en partageant plusieurs tâches. Notamment : apprendre à utiliser `GIT` pour partager son code, découper et se répartir le travail et s'entraider.