Flood

Soulié Alexandre

Paul Maël

Ladagnous Louis-Victor

Lohézic Victor

Département Informatique

Table des matières

1 Introduction **3**
1.1 Présentation du sujet et objectif 3
1.2 Cadre de travail 3

2 Analyse des problèmes rencontrés **3**
2.1 Clients 3
2.2 Serveur 4
2.2.1 Graphe du jeu 4
2.2.2 Déroulement du jeu 4

3 Structures algorithmiques implémentées **4**
3.1 File et parcours en largeur 5

4 Conception de solutions algorithmiques **7**
4.1 Clients 7
4.1.1 Client aléatoire 8
4.1.2 Client glouton 9
4.1.3 Client type Monte-Carlo 9
4.2 Serveur 10
4.2.1 Graphe du jeu 10
4.2.2 Déroulement du jeu 16

5 Afficheur **17**

6 Tests **20**

7 Limites **20**

8 Conclusion **21**

1 Introduction

1.1 Présentation du sujet et objectif

Dans ce projet, nous allons devoir créer des clients de jeu pour le jeu *Flood* ainsi qu'un serveur pouvant générer une partie entre deux clients. Les clients seront sous la forme de bibliothèques dynamiques, et le serveur un exécutable.

1.2 Cadre de travail

La dualité serveur/client fortement imposée par le sujet nous pousse à répartir le travail en ce sens. Pour répartir les tâches, nous avons décidé que M.Paul et M.Soulié s'occuperont des différents clients, tandis que M.Lohézic et M.Ladagnous s'occuperont des implémentations autour du serveur.

2 Analyse des problèmes rencontrés

2.1 Clients

La première réflexion était sur les différentes implémentations possibles et stratégies de jeu. Plusieurs idées n'ont pas été retenues ou implémentées : tenter de s'emparer du centre le plus rapidement possible, entourer l'adversaire le plus rapidement possible, algorithme génétique, algorithme min-max... Ont été retenus un algorithme aléatoire, un algorithme glouton ainsi qu'un algorithme type Monte-Carlo.

Tous ces clients nécessitent une reconnaissance du territoire acquis par le client, et donc nécessairement un parcours de graphe. Il était donc nécessaire de choisir un type de parcours de graphe adapté à notre implémentation des graphes, les matrices d'adjacences. On finira par opter pour un parcours en largeur, plus intéressant qu'un parcours en profondeur ici car on cherche les plus proches voisins dans un grand nombre de cas.

Un dernier problème est la séparation nette entre client et serveur. Il est nécessaire pour notre client de simuler lui-même de son côté la partie, afin de ne pas faire évoluer le client à l'aveugle. Il aura donc fallu trouver des solutions notamment pour garder une carte à jour et pouvoir la faire évoluer selon les coups du client adverse et du client courant. Encore une fois, ce problème de graphe utilisera un parcours en largeur.

2.2 Serveur

2.2.1 Graphe du jeu

Il s'agit de réfléchir au graphe du jeu. La structure de celui-ci est donnée par le sujet :

$$\textbf{graph\_t} \begin{cases} \textbf{num\_vertices :} & \text{entier non signé d'au moins 16 bits} \\ \textbf{t :} & \text{matrice creuse d'entiers de taille } n \times n \\ \textbf{positions[NUM\_PLAYERS] :} & \text{tableau d'entiers non signés} \end{cases}$$

L'entier non signé représente le nombre de sommets que comporte le graphe. Quant à la matrice, c'est une matrice d'adjacence. Le tableau indique les positions de départ des joueurs. Le sujet nous invite à travailler sur 4 familles de graphes de formes différentes, avec des tailles variables. Il est nécessaire alors de penser à une fonction suffisamment générique pour pouvoir initialiser tous les types de graphes de jeu. De plus, une coloration doit y être apportée. Celle-ci peut prendre différentes formes, c'est à dire une coloration cyclique en fonction du numéro des sommets ou encore une coloration aléatoire. L'aspect générique de la fonction est d'autant plus important. Il ne faut pas oublier que les couleurs associées aux positions de démarrage des deux joueurs sont forcément différentes.

2.2.2 Déroulement du jeu

En premier lieu, il a fallu se poser la question de l'initialisation de notre plateau, la communication de ce dernier aux deux clients participants au jeu et du chargement des bibliothèques dynamiques pour le bon déroulement du jeu. Il s'agira donc d'utiliser dlopen afin de charger les deux clients correctement, puis de créer des copies du plateau initial et de les communiquer aux deux clients.

Dans un deuxième temps, il a fallu implémenté le déroulement d'une partie en communiquant les changements effectués par un client à l'autre. De plus, la fin de la partie devra être gérée par le serveur mais également par les clients. Le gagnant de la partie sera déterminé par un parcours en largeur effectué sur le graphe obtenu à la fin de la partie.

3 Structures algorithmiques implémentées

Le plateau de jeu étant un graphe coloré et pour étudier les différentes couleurs autour d'un joueur qui permettront d'établir une stratégie de jeu, il a fallu mettre en place des structures et des algorithmes permettant d'effectuer

une recherche efficace des couleurs entourant un joueur. Pour cela, nous avons choisi d'implémenter une file, ainsi qu'un algorithme de parcours en largeur qui repose sur la file.

3.1 File et parcours en largeur

Pour pouvoir stocker les différentes informations sur les cases environnantes de la position actuelle d'un joueur, nous avons implémentée une file. Nous avons tout d'abord dû définir sa structure, ainsi que celle de ses éléments.

$$\textbf{element} \begin{cases} \textbf{data :} & \text{void}^* \\ \textbf{prev :} & \text{struct element}^* \\ \textbf{next :} & \text{struct element}^* \end{cases}$$

$$\textbf{queue} \begin{cases} \textbf{first :} & \text{struct element}^* \\ \textbf{last :} & \text{struct element}^* \\ \textbf{cpy :} & \text{void}^* \longrightarrow \text{void}^* \\ \textbf{del :} & \text{void}^* \longrightarrow \text{void} \\ \textbf{cmp :} & (\text{void}^*, \text{void}^*) \longrightarrow \text{int} \end{cases}$$

Comme le montre les définitions ci-dessus, nous avons tout d'abord défini la structure des éléments de la file. Ceux-ci contiennent un pointeur *void** vers une donnée *data*, l'utilisation du type *void** est intéressante puiqu'elle permet de faire en sorte que la file puisse contenir des données de n'importe quel type (int, size_t, etc.). Un élément contient également dans sa structure un pointeur vers l'élément qui le précède *prev* dans la file, ainsi que vers l'élément qui le suit *next*. L'intérêt est que l'on peut ainsi parcourir la file dans les deux sens. La file contient quant à elle un pointeur vers son premier élément $first$, ainsi que sur son dernier élément *last*, toujours dans le but de pouvoir parcourir la file dans les deux sens. Elle contient également trois pointeurs de fonctions :

- la fonction *cpy* permet de copier une donnée lors de son ajout dans la file,
- la fonction *del* permet de libérer une donnée lorsqu'elle est retirée de la file,
- la fonction *cmp* permet de comparer les données de deux éléments de la file.

Ensuite, des fonctions de manipulation de la file ont été implémentées. Une fonction de création, une fonction d'ajout d'élément (ajout à la fin de la file), une fonction de suppression d'élément (suppression du premier élément) et une fonction de libération. Il y également une fonction pour déterminer si

la file est vide, ainsi que sa taille. En ce qui concerne les éléments de la file, une fonction de libération et une fonction d'accès à la donnée contenue ont été implémentées.

Nom :	élément	
Opérations :		
element_free	élément	$\longrightarrow$ aucun
element_data	élément	$\longrightarrow$ donnée

Nom :	File	
Utilise :	élément	
Opérations :		
queue_empty	cpy $\times$ del $\times$ cmp	$\longrightarrow$ file
queue_add	file $\times$ donnée	$\longrightarrow$ aucun
queue_pop	file	$\longrightarrow$ élément
queue_free	file	$\longrightarrow$ aucun
queue_is_empty	file	$\longrightarrow$ booléen
queue_size	file	$\longrightarrow$ entier

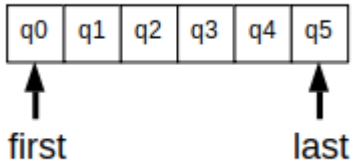


FIGURE 1 – Exemple de file

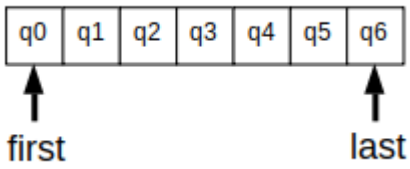


FIGURE 2 – Ajout d'un élément dans la file de la figure 1

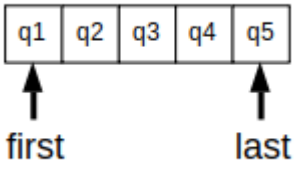


FIGURE 3 – Suppression d'un élément dans la file de la figure 1

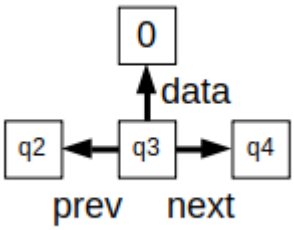


FIGURE 4 – Zoom sur un élément dans la file de la figure 1

Cette structure de file a été implémentée dans le but de procéder à des parcours en largeur, notamment pour déterminer le territoire de chaque joueur et les couleurs disponibles pour jouer.

4 Conception de solutions algorithmiques

4.1 Clients

L'interface qu'on nous demande d'implémenter comporte quatre fonctions : `get_player_name, initialize, play et finalize`.

Si `get_player_name` est assez triviale, les autres demandent plus de réflexion. Chaque client est muni de quatre variables statiques qui évoluent au fur et à mesure de la partie. Le graphe, les couleurs du graphe, l'identifiant ainsi que les couleurs interdites sont stockés statiquement. La fonction `initialize` effectue une simple copie de ses paramètres sur ces éléments, et libère l'espace qui était alloué aux éléments donnés en paramètres. La fonction `finalize` quant à elle s'occupe de libérer la mémoire allouée durant l'entièreté du processus.

La fonction qui diffère le plus selon les implémentations client est `play`. Elle est la fonction qui définit le cœur du client, et est la seule qui sera redéfinie pour chaque client. De ce fait, l'interface des clients propose d'excellentes

qualités modulaires. Cette fonction prend en paramètre le dernier coup joué par l'adversaire afin de pouvoir baser sa stratégie sur ce paramètre.

Voilà quelles ont été les différentes solutions algorithmiques retenues :

4.1.1 Client aléatoire

Le but principal de ce client était d'abord de comprendre l'interface et de perfectionner les points qui ne gravitent pas autour de la fonction `play`. Ce client a aussi permis de construire une base saine pour `play` pour construire les futurs clients. C'est cette base que l'on veut décrire ici.

La fonction `play` reçoit en argument le dernier coup joué par l'adversaire, et puisqu'aucune communication entre le client et le serveur n'est permise, il faut donc faire évoluer notre graphe et ses couleurs. On écrit donc une fonction qui met à jour notre variable statique de couleurs afin d'avoir une mise à jour en temps réel du plateau de jeu. Cette fonction se base sur un parcours en largeur du graphe pour détecter quels sommets mettre à jour selon la couleur et l'identifiant donné . Cette fonction est appelée dès le début de la fonction `play` pour obtenir un plateau de jeu correctement à jour. Il faut maintenant décider d'une couleur à jouer. Afin d'éviter de jouer une couleur qui ne ferait pas augmenter notre surface, on décide encore une fois d'utiliser le parcours en largeur pour obtenir une liste complète des couleurs potentiellement jouables. On effectue enfin un tri de ces couleurs pour ne pas jouer une couleur qui serait interdite par le serveur ou encore pour ne pas jouer une couleur qui nous ferait capturer la surface de l'adversaire.

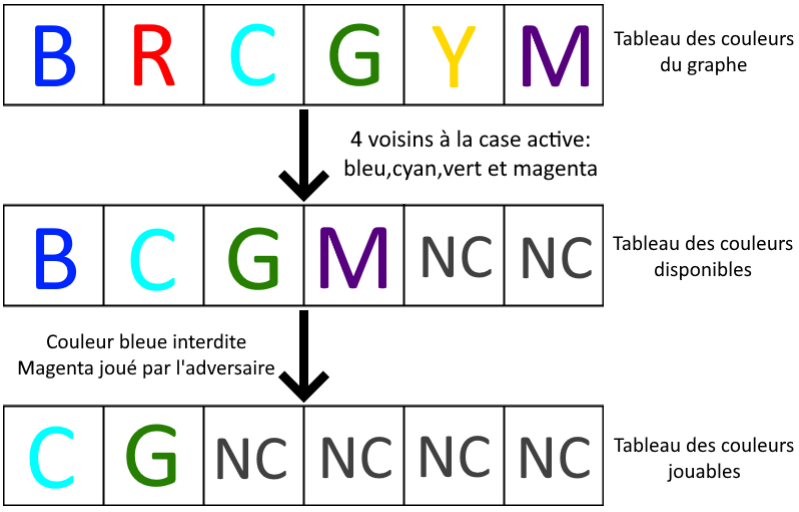


FIGURE 5 – Exemple de processus de sélection de couleurs

Tout ce qui est décrit précédemment est commun à tous les clients. Cela

rend les clients encore plus modulaires. La partie propre à chaque client est le choix de la couleur jouée parmi les couleurs jouables.

Dans notre cas, ici, nous choisirons une couleur au hasard dans le tableau des couleurs jouables, avant de mettre à jour le plateau de jeu en fonction de la couleur choisie et la retourner à l'utilisateur. La mise à jour du plateau de jeu selon la couleur choisie est aussi une constante dans le développement des clients.

Côté complexité, en notant V le nombre de sommets et E le nombre d'arcs, un parcours en largeur possède une complexité en $O(E+V)$. Or nos graphes sont sous la forme présentée dans la Figure 7a. Ainsi, en cachant que $V = m^2$, on en déduit que $E = 2 \times m(m-1)$, d'où finalement une complexité dans le pire des cas de l'ordre de $O(m^2)$ et donc finalement une complexité que l'on peut donner en $O(V)$. Une fois le tableau des couleurs disponibles obtenu, il faut éliminer les couleurs indésirables pour obtenir les couleurs jouables. Cet élimination se fait en temps linéaire du nombre maximal de couleurs disponibles soit $O(V)$, et on obtient ainsi pour obtenir le tableau des couleurs jouables une complexité en $O(V) = O(m^2)$. Cette complexité est très satisfaisante, et nous permet d'abattre beaucoup de travail pour une complexité relativement basse.

4.1.2 Client glouton

Nous nous demandons à présent comment trouver un algorithme de jeu efficace afin de maximiser nos chances de victoire. Le premier algorithme auquel on peut penser est un algorithme glouton qui se base sur le maximum local. L'algorithme choisit la couleur qui propose la plus grande extension de territoire au moment de l'appel. Cette solution est finalement meilleure que la solution aléatoire, mais se montre très inefficace dans certains cas. Pour améliorer cette méthode, on peut par exemple penser sur le long terme plutôt que sur le simple instant présent. C'est cette réflexion qui mènera au prochain algorithme.

4.1.3 Client type Monte-Carlo

La réflexion faite de creuser plus en profondeur donne ici naissance à l'idée d'un algorithme "type Monte-Carlo". L'idée derrière cet algorithme est de jouer un grand nombre de parties au hasard depuis une situation de départ, puis de regarder quel premier coup a mené au plus grand pourcentage de victoire. Cet algorithme n'est **pas déterministe** car il ne donne aucune assurance d'un résultat 100% correct, pour autant, son temps d'exécution est lui déterministe, nous permettant d'être certain de la terminaison de

l'algorithme.

En supposant que l'algorithme propose de jouer N parties pour chacune des C couleurs, et que chaque partie soit complète (i.e. aboutisse à la victoire d'un des deux clients ou alors à l'égalité de la partie), en notant V le nombre de sommets du graphe et P la durée d'une exécution de `play` du client aléatoire, on peut avoir alors dans le pire des cas une complexité de l'ordre de $O(NCVP)$.

En sachant que $P = V$ et que $C << V$, on obtient une complexité en $O(NV^2)$. Dans la pratique, pour obtenir des résultats probants, on choisira $N \geq V$, et on obtient alors une complexité cubique en le nombre de sommets. Cette complexité est estimée largement à la baisse et la réalité prouve que cette complexité est énorme. Pour réduire le temps d'exécution, on pourra jouer sur plusieurs paramètres :

— N le nombre de parties jouées par couleur,

— la profondeur des parties jouées.

Expliquons cette notion de profondeur de jeu : l'algorithme jusqu'à présent jouait des parties qui finissait nécessairement. Pour réduire fortement la complexité, on peut décider d'une profondeur de jeu, soit le nombre de coups totaux joués par les deux joueurs. La diminuer à un nombre fixe fait passer la complexité à une complexité quadratique en le nombre de sommets, ce qui est bien plus acceptable.

4.2 Serveur

Nous avons traité les arguments grâce à la bibliothèque optarg. Les arguments sont les suivants : -m pour spécifier la largeur du plateau de jeu, -t pour spécifier la forme du plateau de jeu, -c pour spécifier le nombre de couleurs utilisées dans la partie, -a pour spécifier l'algorithme de coloration utilisé et -f pour spécifier le nombre de couleurs interdites pour chaque joueur.

4.2.1 Graphe du jeu

Pour la partie serveur, nous avons défini une structure pour le plateau de jeu :

$$\textbf{board} \begin{cases} \textbf{g :} & \text{pointeur vers une structure graph\_t} \\ \textbf{c :} & \text{tableau d'énumérations couleurs} \end{cases}$$

La structure pour les graphes est la suivante :

$$\textbf{graph\_t} \begin{cases} \textbf{num\_vertices :} & \text{size\_t} \\ \textbf{t :} & \text{pointeur de fonction vers gsl\_spmatrix\_uint} \\ \textbf{positions :} & \text{tableau de size\_t} \end{cases}$$

Le nombre de sommets est représenté par num_vertices. Pour modéliser le graphe du jeu, nous avons choisi de numéroter les cases du plateau de jeu de la manière suivante :

1	2	3
4	5	6
7	8	9

FIGURE 6 – Numérotation du plateau de jeu

Dans la figure (6), les sommets adjacents au sommet 1 sont 2 et 4. Les sommets adjacents au sommet 5 sont 2, 4, 6 et 8. On comprend alors que notre graphe de jeu est un graphe non orienté. Le sujet nous invite à représenter en mémoire ce graphe sous la forme d'une matrice d'adjacence. Soit $G = (V, E)$ un graphe orienté simple, où V représente l'ensemble des numéros des sommets et E les arêtes du graphe G, la matrice d'adjacence A de G est une matrice de taille $|V| \times |V|$ telle que :

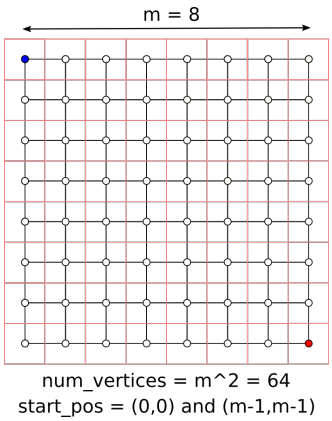
$$a_{ij} = \begin{cases} 1 & \text{si } (v_i, v_j) \in E \\ 0 & \text{sinon.} \end{cases}$$

Dans l'exemple de la figure (6), la matrice d'adjacence est la suivante :

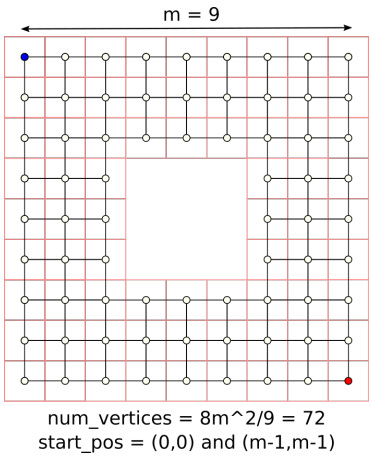
$$\begin{array}{ccccccccc} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array}$$

Dans cet exemple avec un graphe carré, nous pouvons observer que la matrice d'adjacence est constituée d'un grand nombre de zéros. En effet, une colonne ou une ligne de cette matrice ne peut pas avoir plus de quatre 1 car une case du plateau a au maximum quatre voisins. On comprend alors le choix de la structure de données imposée par le sujet. Nous devons utiliser une matrice creuse fournie par la bibliothèque GSL.

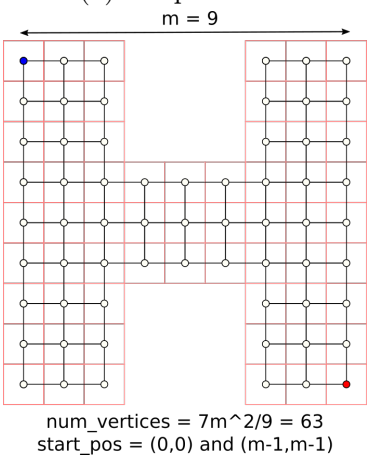
Pour initialiser le graphe, nous avons une fonction qui initialise le jeu. Celle-ci prend en compte le type de graphe souhaité. Les graphes peuvent être carrés, hachés, toriques ou en forme de donuts.



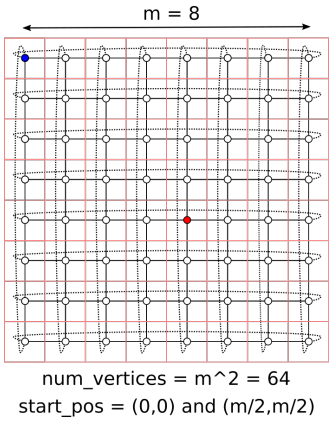
(a) Graphe carré



(b) Graphe en forme de donut



(c) Graphe haché



(d) Graphe torique

FIGURE 7 – Différentes formes possibles de graphe

Pour initialiser le graphe du jeu, on commence par allouer une matrice GSL de type COO. Il est possible d'ajouter un paramètre appelé `nzmax` qui correspond au nombre de coefficients de la matrice non nuls. La fonction qui génère le graphe possède dans sa signature un pointeur de fonction pour `nzmax`. En effet, le nombre de coefficients non nuls dépend du type de graphe. Par exemple pour un plateau de jeu de dimension carré $m \times m$:

$$4 \times (m-2) \times (m-2) + 3 \times (m-2) \times 4 + 2 \times 4. \tag{1}$$

De manière analogue, on a déterminé les formules pour les autres types de graphe. Le programme parcourt toutes les cases d'indices (i, j). Une fonction `filter` permet de dire si la case d'indice (i, j) appartient bien au plateau de jeu. Cette fonction permet la généralisation de la fonction générant le plateau de jeu. Cette fonction diffère en fonction du type de graphe, c'est pourquoi elle est transmise à la fonction d'initialisation par un pointeur de fonction. Si la case appartient au plateau de jeu, on vérifie en plus si la case est sur un coin ou un bord du plateau. En effet, le nombre de voisins n'y est pas identique. On a alors défini les énumérations suivantes :

```
enum line {
    NONE_LINE, UP_IN, LEFT_IN, RIGHT_IN, DOWN_IN, UP_OUT,
    LEFT_OUT, RIGHT_OUT, DOWN_OUT
};

enum corner {
    NONE_CORNER, UP_RIGHT1, UP_RIGHT2, UP_LEFT1, UP_LEFT2,
    DOWN_LEFT1, DOWN_LEFT2, DOWN_RIGHT1, DOWN_RIGHT2
};
```

Pour illustrer le sens des ces dernières, voici une illustration appliquée sur la figure (7) :

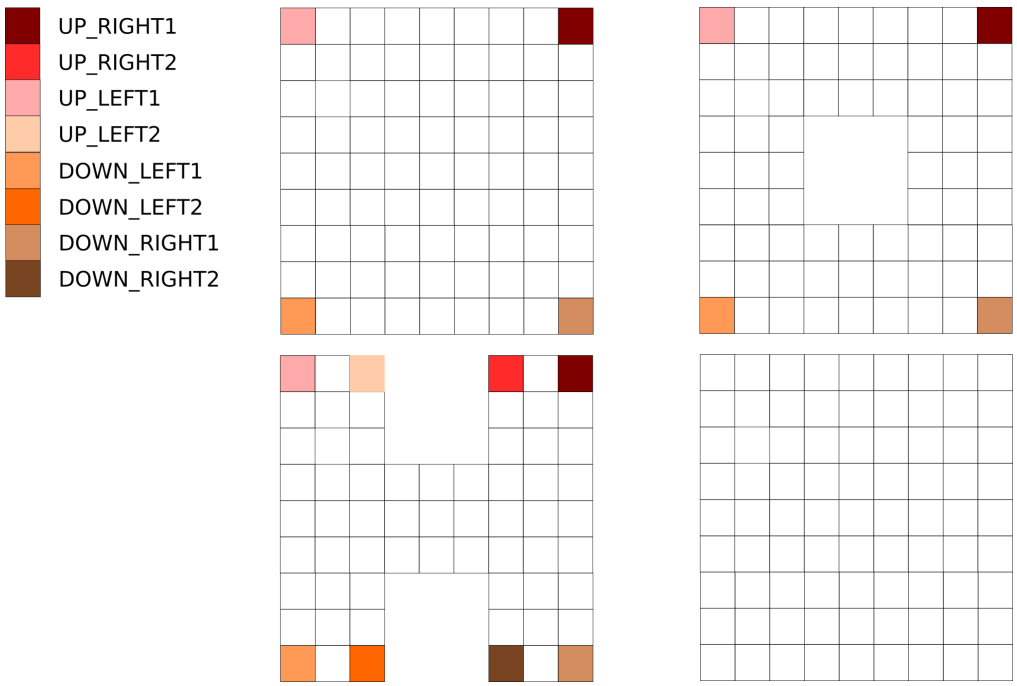


FIGURE 8 – Illustration du fonctionnement de l'énumération pour les coins

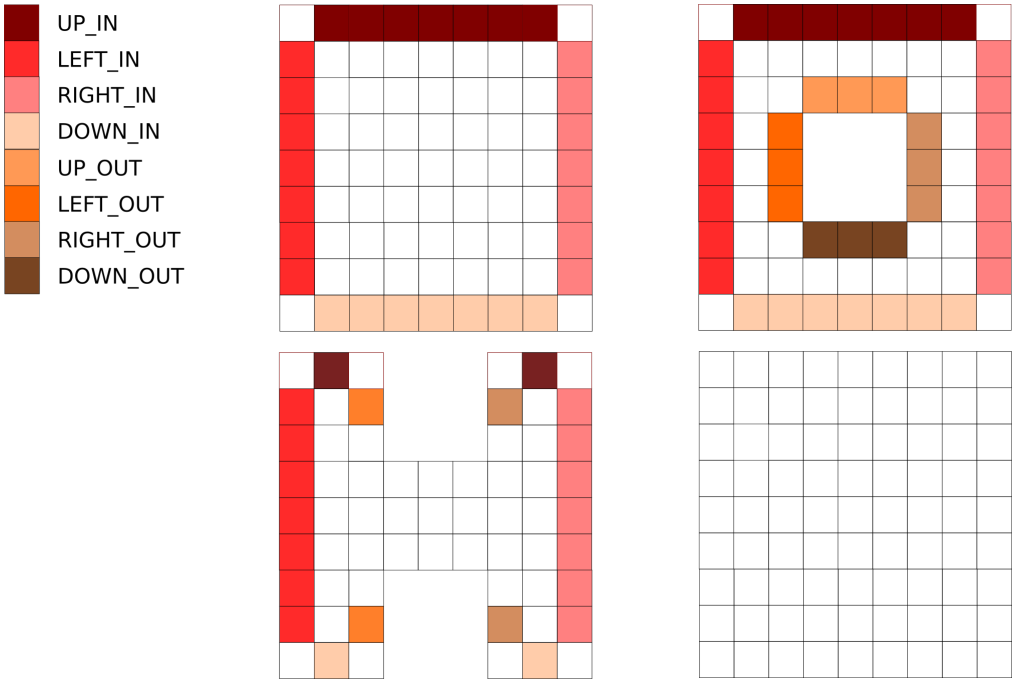


FIGURE 9 – Illustration du fonctionnement de l'énumération pour les lignes

Ainsi en fonction de l'énumération renvoyée par `is_border` ou `is_corner`, une fonction va compléter neighbour et va modifier par effet de bord une variable dont la structure est :

$$\textbf{neighbour} \begin{cases} \textbf{up :} & \text{entier initialisé à -1} \\ \textbf{down :} & \text{entier initialisé à -1} \\ \textbf{left :} & \text{entier initialisé à -1} \\ \textbf{right :} & \text{entier initialisé à -1} \end{cases}$$

Ensuite, en fonction de la position des voisins obtenus, on actualise la matrice d'adjacence. La matrice d'adjacence est une matrice creuse GSL. Nous les convertissons par la suite en matrice de type CSR.

Pour la coloration du graphe, un second pointeur de fonction lui est transmis. Ce pointeur de fonction permet d'obtenir une fonction plus générique et donc d'appliquer des colorations différentes. Nous affectons les couleurs en remplissant un tableau de dimension $m \times m$ avec m la dimension du monde. Pour les couleurs, on utilise l'énumération suivante :

```
1 enum color_t {
2   BLUE=0, RED=1, GREEN=2, YELLOW=3,
```

```
3   ORANGE=4, VIOLET=5, CYAN=6, PINK=7,
4   MAX_COLOR=8,
5   NO_COLOR=-1,      // Color when no move is possible
6 };
```

Lorsque la fonction **filter** avec les coordonnées $i \times j$ de la case est vérifiée, on remplit la case correspondante en appelant la fonction du pointeur de fonction. Dans le cas contraire, l'élément du tableau est NO_COLOR.

Notre parcours est donc le suivant :

pour ligne allant de 0 à $m - 1$ faire
    pour colonne allant de 0 à $m - 1$ faire
        monde[ligne * m + colonne]
    ;
;

Ainsi, ce type de parcours permet de profiter au mieux du principe de localité spatiale. En effet, lorsqu'une cellule mémoire est accédée (en lecture ou écriture), un accès à une cellule mémoire proche a lieu peu de temps après. La mémoire cache s'appuie alors sur cette idée pour prédire les données qui vont sûrement être nécessaires au programme dans un futur proche. Cependant, parfois il est plus simple de visualiser les cases du monde par ses coordonnées. Par conséquent, la fonction qui nous donne l'indice correspondant dans le tableau est très utile. Lorsqu'on itère sur les cordonnées, on prend garde de toujours profiter de la localité spatiale en accédant aux cellules mémoires consécutivement :

pour i allant de 0 à m^2 faire
    monde[ligne * m + colonne]
;

Ce principe de localité spatiale permet de justifier aussi la conversion en matrice de type CSR. En effet, les valeurs non nulles de la matrice sont stockées consécutivement dans un tableau unidimensionnel. Ce sont ces valeurs qui intéressent le client et le serveur.

Nous avons par la suite codé une fonction qui génère un entier entre 0 et le nombre maximum de couleur moins 1.

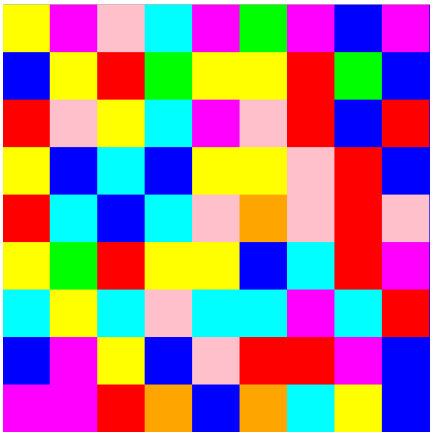


FIGURE 10 – Coloration aléatoire

Pour apporter une coloration cyclique au graphe, nous renvoyons le numéro du sommet modulo le maximum de couleurs.

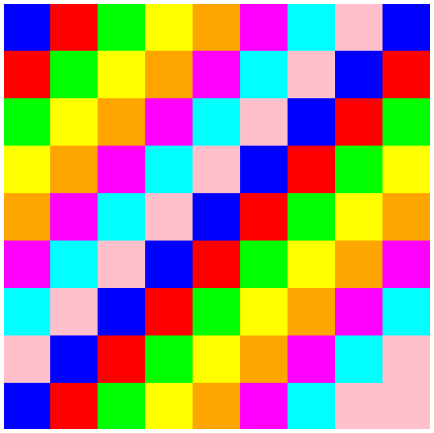


FIGURE 11 – Coloration cyclique

4.2.2 Déroulement du jeu

Tout d'abord nous avons implémenté une structure de joueur qui correspondra à chaque client.

$$\textbf{player} \begin{cases} \textbf{id :} & \text{size\_t} \\ \textbf{get\_player\_name :} & \text{void} \longrightarrow \text{char const}^* \\ \textbf{initialize :} & (\text{size\_t, struct graph\_t}^*, \text{enum color\_t}^*, \\ & \text{struct color\_set\_t}^*) \longrightarrow \text{void} \\ \textbf{play :} & \text{struct move\_t} \longrightarrow \text{struct move\_t} \\ \textbf{finalize :} & (\text{void}^*, \text{void}^*) \longrightarrow \text{int} \end{cases}$$

Une fois cette structure implémentée, on déclare ensuite un tableau de players qui contiendra les deux clients. À l'aide de la ligne de commande `dlopen (argv[i], RTLD_LAZY)`, on peut ainsi ouvrir chaque client et utiliser par la suite la fonction `dlsym` pour initialiser chaque joueur.

Il s'agit maintenant de générer un plateau aléatoire en accord avec les options traitées dans la partie INPUT. Une fois généré, ce plateau doit être copié deux fois pour être communiqué à chaque client. Pour cela, la fonction `memcpy` est utilisée. Or, cette fonction n'est pas compatible avec les matrices GSL. Il faut donc utiliser la fonction `gsl_spmatrix_uint_memcpy` afin de copier la matrice GSL, puis de communiquer ces copies de matrices aux deux clients. Le jeu peut ainsi commencer.

Le jeu se déroule selon le pseudocode fourni dans le sujet. Il s'agit donc de faire jouer chaque joueur et de mettre à jour le plateau à chaque tour tant que les deux joueurs n'ont pas tous les deux jouer la couleur NO_COLOR. La fonction `update_map` étant aussi utilisée par les clients, elle est déjà détaillée dans la partie 4.1. Une fois la partie terminée, il faut obtenir le gagnant de la partie (ou éventuellement une égalité). La fonction `get_winner` se base sur un parcours en largeur pour compter le nombre de cases appartenant à chaque joueur afin de déterminer le gagnant. Le parcours en largeur effectué dans la fonction `get_winner` est défini dans la partie 3.1.

Une fois la partie terminée, à l'aide de la fonction `finalize` de chaque joueur, on met fin au jeu. Les clients doivent s'occuper de la libération des blocs alloués pour les copies du plateau, tandis que le serveur libère l'espace alloué pour le plateau original grâce à la fonction `free(board)`. De même que pour les copies, on doit utiliser la fonction `gsl_spmatrix_uint_free` pour libérer les matrices GSL.

5 Afficheur

Pour pouvoir vérifier la bonne implémentation des graphes, ainsi que la bonne implémentation des stratégies de jeu utilisées par les différents clients, il est nécessaire d'avoir un afficheur permettant de visualiser l'évolution du

graphe au cours d'une partie. Cette afficheur est basé sur la bibliothèque SDL qui permet, en lui donnant la taille WIDTH × HEIGHT de la grille, ainsi que les couleurs du graphe au format RGB, d'afficher le graphe coloré du jeu.

Pour ce faire, on utilise une fonction d'affichage, nommée `graph_disp` qui prend le tableau des couleurs et un pointeur vers le graphe à afficher en entrées. Par la suite, en connaissant le nombre de sommets du graphe, le type de graphe est récupéré grâce à la fonction `graph_type` et la largeur du graphe est récupérée avec la fonction `matrix_m`. Le tableau des couleurs est ensuite converti par la fonction `convert` qui permet d'avoir un nouveau tableau de couleurs au bon format, c'est-à-dire qu'il contient des NO_COLOR, qui seront par la suite représentés par la couleur noire, aux endroits où le graphe n'est pas défini (cas du vide central pour le donut, et des vides en haut et en bas pour le H). Les couleurs sont ensuite transformées au format RGB par la fonction `rgb_color`. Une boucle permet enfin d'afficher les couleurs au format exploité par la bibliothèque SDL.

Pour un graphe donné, l'établissement de son type et de sa largeur m se fait à partir de son nombre de sommets $num\_vertices$, seule information utile à l'affichage contenue dans sa structure. Il y a 3 cas qui se présente :

- cas n°1 : $\sqrt{num\_vertices}$ est un entier, on a donc $m = \sqrt{num\_vertices}$ et le type de graphe est soit carré, soit torique (on ne différencie pas les deux car cela n'a aucune importance pour l'affichage),
- cas n°2 : $\sqrt{2} \times \sqrt{num\_vertices} \times \frac{3}{4}$ est un entier, m est donc égal au résultat du calcul précédent et le type de graphe est donut,
- cas n°3 : $\sqrt{7} \times \sqrt{num\_vertices} \times \frac{3}{7}$ est un entier, m vaut donc le résultat du calcul précédent et le type de graphe est H-graphe.

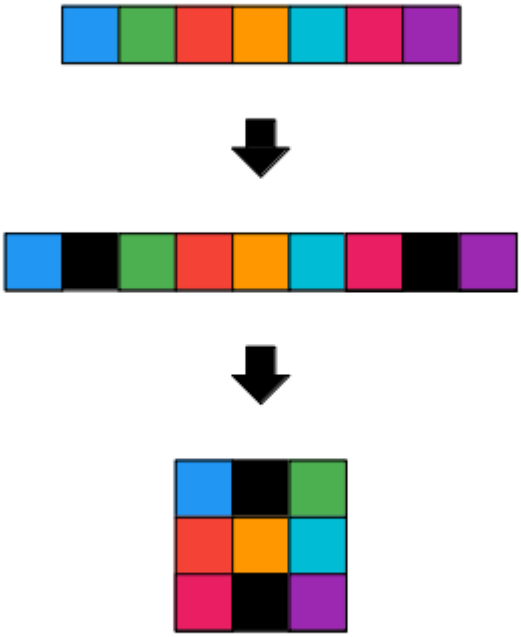


FIGURE 12 – Fonctionnement de l'afficheur pour un H-graphe

La figure 12 ci-dessus décrit le processus pour afficher le graphe correspondant au tableau de couleurs : $[bleu, vert, rouge, orange, cyan, rose, violet]$. Ce tableau contenant 7 couleurs, on détermine aisément que le type de graphe correspondant est un H-graphe, on détermine également sa largeur m. Ensuite, on inclut des NO_COLOR au bon endroit dans le tableau qui sont représentés en noir. En connaissant la largeur m du graphe, on peut afficher correctement les codes couleurs RGB de manière à ce qu'ils soient interprétés par la bibliothèque SDL. Au cours du projet, nous avons changé l'implémentation des tableaux de couleurs pour les graphes en H et les donuts, de manière à intégrer directement les NO_COLOR dans le tableau des couleurs dès la création du graphe. L'étape de transformation des tableaux des couleurs n'est donc plus nécessaire.

6 Tests

Les tests sont une partie importante du développement dans un projet informatique. Ils permettent notamment d'assurer le bon fonctionnement des différentes fonctions, ainsi que de déceler les bugs et fuites mémoires dans le code en lançant l'outil *valgrind* sur les exécutables de tests.

Pour vérifier le comportement de nos fonctions, nous avons pour chaque fichier $file.c$, un fichier $file\_test.c$ contenant les tests unitaires de toutes les fonctions de $file.c$, et nous avons également un fichier $file\_test\_main.c$ qui permet d'exécuter les tests de $file\_test.c$. Dans le projet, nous n'avons fait que des tests unitaires, cela nous permet de s'assurer du bon comportement des fonctions en les prenant séparément, mais pas de vérifier qu'elles fonctionnent lors de leur utilisation conjointe. Il aurait fallu faire des tests fonctionnels pour tester le bon fonctionnement d'un module dans son entièreté, puis des tests d'intégration pour vérifier la cohérence des fonctions de chaque module et tester le fonctionnement de plusieurs modules entre eux. Enfin, nous aurions pu faire des tests de recette pour attester la validité du projet. Cependant, le visionnement de parties types grâce à l'afficheur permet de vérifier visuellement le bon fonctionnement des fonctionnalités de base telles que la construction de graphe ou encore le fait que les deux joueurs jouent des couleurs autorisées. L'utilisation de l'option *--coverage* permet également de vérifier les lignes testées dans nos différents tests et donc de valider leur comportement.

7 Limites

Les clients "complexes" ont rapidement été touchés par la réalité des choses. Le temps d'exécution du client Monte-Carlo par exemple a été sujet à de nombreux problèmes, excédant souvent la barre des 15 secondes d'exécution sur des graphes 10×10. Même après réduction de la profondeur de jeu ou du nombre de parties par couleurs, le temps d'exécution reste long. Cela s'explique par le fait que ce qui a été calculé dans la Section 4.1.3 était l'exécution de la simple fonction `play`, et non pas d'une partie entière avec le client. Une partie entière avec le client peut emmener la complexité dans le pire des cas vers $O(V^4)$ ce qui est clairement mauvais. Les réductions de profondeur sont donc clairement nécessaires. A titre d'exemple, supposons une partie jouée entre notre client Monte-Carlo et un client qui peut jouer instantanément. Prenons le pire des cas, où notre algorithme joue les parties en entier, et où le coefficient multiplicatif de notre complexité est $N >> 1$. Supposons un processeur 1GHz et supposons que chaque tour d'horloge effec-

tue un calcul. Avec $V = 100$ (soit une grille 10×10), on obtient $N \times 100^4$ tours d'horloge à effectuer, effectués en $0,1N$ s par le processeur. Avec $N >> 1$, on voit que les temps d'exécution peuvent rapidement devenir gigantesques.

8 Conclusion

Le sujet, bien que très intéressant à nos yeux, était trop court pour développer l'ensemble de nos idées. Les multiples clients qui auraient dû voir le jour n'ont pas pu être implémentés notamment par manque de temps. Ce projet à tout de même pu mettre en lumière la difficulté du travail en groupe, mais a aussi permis de mettre en oeuvre les connaissances acquises des cours de programmation impérative avancée et d'atelier algorithmique.