Projet Color Flood : Sprint 1

Équipe Gin Gin Mooney
Membres : Eléna Bouchaud, Hanna Ben Jedidia,
Hélène Tran et Nicolas Chevillard
Référent : Nicolas Chevillard

Introduction :

Le Color Flood est un jeu constitué d'une grille de n*n cases colorées. Le but du jeu est de colorer l'ensemble de la grille d'une même couleur. Pour cela, il faut convertir la couleur de la composante connexe principale (ensemble de cases connexes de la même couleur commençant par la case en haut à gauche) en une des couleurs des cases qui l'entourent et ainsi agrandir la composante connexe principale.

Le projet consiste à réaliser ce jeu en quatre sprints : rédaction et test du code permettant de définir, initialiser et manipuler les structures de données nécessaires au bon fonctionnement du jeu, exécution d'une partie, création d'un solveur et enfin réalisation du jeu complet avec un nombre de coups limité déterminé par le solveur.

Ce rapport présente les solutions apportées par l'équipe Gin Gin Money ainsi que son organisation pour la réalisation du sprint 1.

1- Solution proposée :

Pour réaliser le jeu, nous devons créer une grille de taille n*n composées de cases colorées. Ces cases doivent pouvoir changer de couleur, surtout si elles font partie de la composante connexe principale. Il faut donc que nous puissions déterminer les couleurs autour de chaque case et surtout autour de la composante connexe principale.

Nous avons décidé de créer, en parallèle de la grille, une structure de composantes connexes pour garder en mémoire la liste de toutes les composantes connexes. Cette structure contient les différentes cases qui font partie de la composante, sa couleur mais aussi les composantes qui lui sont voisines.

Nous avons choisit de créer quatre modules, un pour chaque structure. Le module Grille.c, Case.c, ComposanteConnexe.c et ListeComposanteConnexe.c. La description des modules est détaillée dans la partie Algorithmique du rapport.

2- Organisation de l'équipe :

Comme dit précédent, nous avons quatre modules à développer, commenter et tester. Nous avons décidé que chaque membre développera l'un des modules. Le module Grille.c a été pris en charge par Hélène, Case.c par Eléna, Composanteconnexe.c par Nicolas et ListeComposanteConnexe par Hanna. Le Makefile a été écrit par Nicolas. Le code commenté avec compilation sans erreurs ont été terminés le 20 février.

Ensuite, nous avons réalisé les tests unitaires. Le module Grille.c a été testé par Hanna, Case.c par Hélène, Composanteconnexe.c par Nicolas et ListeComposanteConnexe par Hélène. Nous avons finalisé les tests unitaires le 25 février.

3- Algorithmie :

3.1- Module Grille :

Le module Grille constitue le squelette du plateau : il définit le type énuméré Couleur et la structure de case. Les fonctions de base sont ici réalisées : les getters avec *getXCase*, *getYCase* et *getCouleurCase* et la fonction de changement de couleur de case *setCouleur*.

Allocation d'une grille carrée de couleurs de taille variable

La fonction *tableauVide* permet d'allouer de la mémoire pour créer une grille carrée sous forme de tableau à deux dimensions. La taille de la grille est mise en paramètre pour que l'utilisateur puisse entrer la taille qu'il souhaite.

Libération de l'espace mémoire occupé par une grille

Pour éviter le problème des fuites de mémoire, la fonction *liberationGrille* permet de libérer la mémoire occupée par la grille à la fin du jeu.

Initialisation de la grille à partir de valeurs aléatoires

Pour réaliser une grille de cette manière, il est nécessaire de créer une fonction *aleatoire* qui renvoie un nombre aléatoire entre 0 et 5 : chaque nombre sera associé à une couleur. Cela permet de créer la fonction *remplissageAleatoire* qui prend en paramètre la grille à remplir et qui, à l'aide d'une double boucle for, remplit chaque case du tableau avec la fonction *aleatoire*.

Initialisation de la grille à partir de valeurs contenues dans un fichier

Plusieurs fonctions ont été créées pour vérifier qu'il n'y a pas d'erreur lors du remplissage de la grille à partir d'un fichier : *erreurOuverture*, *erreurLongueur*, *erreurFinFichier* et *checkCouleur*. En cas d'erreur, le programme s'arrête.

Une autre fonction peut être utile pour générer une grille à partir d'un fichier : la fonction *creationFichier*. Elle s'appuie sur la fonction *aleatoire* pour créer n suites de n lettres aléatoires (B,V,R,J,M,G), n étant la taille de la grille.

Ces fonctions préliminaires permettent ainsi de créer la fonction *remplissageFichier* qui prend en paramètre le chemin du fichier et la taille de la grille.

3.2- Module Case

Ce module permet de gérer les listes de cases qui sont réutilisées dans la structure de la composante connexe. En effet, celle-ci contient la liste des cases contenues dans une composante connexe.

Nous avons donc créé tout d'abord une structure de pointeurs de cases. Puis nous avons développé différentes fonctions permettant de manipuler ces listes. La fonction *initListeCase* initialise la liste à NULL. t*estListeCaseVide* teste si la liste de pointeurs de cases est bien vide. *constructeurListeCase* permet de construire une liste de pointeurs de cases, celle-ci ajoute un élément en tête de la liste.

getValeurListeCase renvoie la tête de la liste tandis que *getSuivantListeCase* permet d'accéder à la queue de celle-ci.

liberationCase permet de libérer la mémoire d'une cellule et *liberationListeCase* utilise *liberationCase* afin de libérer l'espace mémoire pour une liste complète. Ces fonctions permettent de libérer l'espace mémoire et donc d'éviter les fuites de mémoire.

estPresentDansListeCase est une fonction qui permet de rechercher un pointeur de case dans la liste et indique si celui-ci est présent ou non dans cette liste.

concatenationListeCase prend deux listes en paramètres et ajoute l'ensemble des éléments de la deuxième liste à la première. Celle-ci renvoie la liste concaténée. Si celle-ci est détruite, la deuxième liste contenue dans la première est également détruite.

supprimeElementListeCase est une fonction qui va passer un à un tous les éléments de la liste et qui va supprimer l'élément pris en paramètre. L'espace mémoire utilisé par celui-ci sera libéré. La difficulté a été de libérer l'espace mémoire de cette élément sans perdre l'information permettant de passer au suivant de la liste.

3.3- Module Composante Connexe

3.3.1- Présentation globale

Ce module permet de gérer les composantes connexes. Il permet la création, des différentes composantes connexes, et les modifications de ces composantes connexes.

La structure de composante représente un ensemble de cases adjacentes de la même couleur. Chaque composante connexe est définie à partir d'une case de la grille. La structure de composante connexe contient une couleur, une liste de pointeurs vers les cases qui la constitue, et une liste de pointeurs vers d'autres composantes connexes qui sont ses voisines.

On peut trouver dans ce module les fonctions d'allocation et de libération de la mémoire pour les composantes connexes, cependant on ne peut pas les utiliser car elles sont déclarées en static pour éviter toute utilisation non contrôlée.

La fonction *creeComposanteConnexe* permet cependant d'obtenir un pointeur vers une composante connexe que l'on crée à partir d'une case, mais cette fonction n'est utile que pour les tests unitaires.

On peut aussi trouver les différentes fonctions pour changer la couleur d'une composante connexe, tester si deux composantes connexes sont identiques, ou obtenir les cases voisines à une composante connexe.

La structure de TabComposanteConnexe est une structure de liste pouvant contenir des ComposanteConnexe. C'est une structure crée pour le stockage des composantes connexes. Les fonctions d'allocation de gestion de la mémoire ont été implémentées.

Les trois fonctions utiles pour la suite sont :
listeComposanteConnexeGrille, qui permet d'obtenir toutes les composantes connexe d'une grille et de les stocker dans un TabComposanteConnexe.
creerVoisins, Qui crée tous les voisins de toutes les composantes connexes
testVictoire qui test si la longueur du TabComposanteConnexe vaut 1. Si c'est le cas, celà veut dire qu'on a plus qu'une composante connexe dans la grille et donc qu'on a gagné.

3.3.2- Exemple d'utilisation

Nous allons voir dans cette partie un exemple normal d'utilisation du module.
Au préalable, l'utilisateur dispose d'une grille de jeu remplie, c'est à dire d'un tableau 2D de cases avec des couleurs différentes dans les cases.

Tout d'abord on crée le TabComposanteConnexe avec la fonction *listeComposanteConnexeGrille*, on obtient ainsi toutes les composantes connexes. La dernière composante connexe de la liste étant celle en haut à gauche de la grille. Il faut récupérer son adresse avec un pointeur. On l'appellera composante connexe principale.

Ensuite, on crée les voisins des composantes connexes avec la fonction *creerVoisins*.

On peut alors progresser dans le jeu en changeant la couleur de la composante connexe principale à l'aide de la fonction *changementCouleur*. La fonction changement couleur modifie la composante connexe principale pour lui attribuer des nouveaux voisins et des nouvelles cases.

Dès que *testVictoire* renvoie 1, on a alors complété la grille avec une seule composante connexe.

3.4- Module Liste de Composantes Connexes

Ce module permet de gérer les listes de composantes connexes dont nous avons surtout besoin pour pouvoir connaître les composantes connexes voisines.

Tout d'abord, nous avons créé une structure de liste de pointeurs composantes connexes. Nous avons décidé de créer une liste de pointeurs pour pouvoir utiliser les fonctions du module ComposanteConnexe.c sans problème à la compilation.

Nous avons ensuite codé les fonctions qui permettent de gérer ces listes. La fonction *initListeComposanteConnexe* permet d'initialiser à NULL une liste. La fonction *estVideListeComposanteConnexe* permet de vérifier si une liste est vide. Le constructeur *constructeurListeComposanteConnexe* qui permet d'ajouter un élément en tête de la liste. Pour éviter les fuites de mémoires, nous avons ensuite créé le destructeur qui est décomposé en deux fonctions. *DestructeurListeComposanteConnexe* libère l'espace mémoire d'une liste. Pour cela, il faut libérer l'espace mémoire de chaque cellule de la liste, cette fonction appelle donc la fonction *destructeurCelluleListeComposanteConnexe* qui s'en charge.

Nous avons aussi les getters *getValeurListeComposanteConnexe. getSuivantListeComposanteConnexe* qui permettent aux autres modules d'avoir accès respectivement à la première composante connexe de la liste passée en argument et la queue de la liste.

La fonction *longueurListeComposanteConnexe* permet de calculer la longueur d'une liste. La fonction *rechercheElementListeComposanteConnexe* cherche la composante connexe passée en paramètre est présent dans la liste passée en paramètre. Si c'est le cas, elle renvoie un pointeur vers cet élément, sinon elle renvoie NULL. La dernière fonction de ce module est *supprimeElementListeComposanteConnexe* qui supprime la composante connexe passée en paramètre de la liste passée en paramètre. Dans cette fonction, on a dû faire appel aux destructeurs pour éviter toute fuite de mémoire.

4- Tests Unitaires :

4.1- Module Grille :

Dans la fonction *testCreationCases* nous testons le bon fonctionnement des fonctions *tableauVide, getXCase*, *getYCase*, *getCaseGrille*. Pour cela, nous créons un tableau vide de couleur et nous testons si les cases de ce tableau ont bien les bonnes coordonnées. Nous testons aussi le comportement de la fonction quand un utilisateur entre une taille négative de tableau : la fonction ne fait rien dans ce cas, aucun tableau n'est construit.

TestRemplissageFichier teste le bon fonctionnement des fonctions *remplissageFichier* et *getCouleurCase*. Pour cela, on initialise une grille avec la fonction *remplissageFichier* en passant en argument le fichier test préalablement créé. Puis, nous testons pour une ligne et une colonne si les couleurs qui ont été attribuées dans le tableau sont bien les couleurs données par le fichier. Les ligne et colonne choisies possèdent à elles deux toutes les couleurs pouvant exister dans une grille. Ainsi, nous avons bien testé que la fonction reconnaît bien toutes les couleurs et les attribue aux bonnes cases. Cependant, si le fichier rentré en paramètre possède des caractères autre que la première lettre de couleurs qui peuvent exister dans le grille alors la fonction *remplissageFichier* envoie un message d'erreur et arrête le programme. Nous ne pouvons donc pas tester avec C-Unit ce cas extrême, sinon le programme s'arrête de tourner.

Nous ne pouvons pas tester de la même façon la fonction *remplissageAleatoire* car nous ne connaissons pas les couleurs affectées aux différentes cases de la grille, nous ne pouvons donc pas vérifier si les couleurs sont les bonnes.

TestSetCouleur teste si le setter *setCouleur* fonctionne correctement. Pour cela, nous devons d'abord initialiser une grille et remplir l'une de ses cases avec le setter. Ensuite, on vérifie si la couleur de cette case est bien la couleur entrée précédemment.

Pour toutes les fonctions test où nous initialisons une grille, nous utilisons à la fin le destructeur *liberationGrille pour éviter toute fuite de mémoire.*

4.2- Module Case :

Dans le module Case, nous avons décomposé le test unitaire en deux fonctions *testListe1* et *testListe2*.

Le test testListe1

Les fonctions testées sont la fonction d'initialisation de liste (*initListeCase*), le test de vacuité (*testListeCaseVide*), le constructeur (*constructeurListeCase*), les getters (*getValeurListeCase* et *getSuivantListeCase*) et la fonction de libération de mémoire (*destructeurListeCase*).

Pour cela, une liste de cases l est initialisée avec la fonction *initListeCase* et on vérifie qu'elle est vide avec la fonction *testListeCaseVide*. Pour tester les autres fonctions, le constructeur de case *constructeurCase*, implémenté dans le module Grille, permet de créer les cases c et c2. On utilise le constructeur de listes de cases *constructeurListeCase* pour remplir la liste l et on vérifie qu'elle est cette fois non vide.

Connaissant la liste de cases qu'on a créé, on teste les getters *getValeurListeCase* et *getSuivantListeCase* en vérifiant la case en tête de liste et si le reste de la liste est vide ou

pas. Enfin, la fonction de libération de mémoire *destructeurListeCase* est testée en utilisant Valgrind.

Le test testListe2

Les fonctions testées sont les fonctions *estPresentDansListeCase*, *concatenationListeCase* et *supprimeElementListeCase*.

Pour réaliser des tests unitaires sur ces fonctions, on a créé plusieurs listes à l'aide du constructeur de listes de cases. Tout d'abord, on a vérifié si une case donnée est présente dans la liste ou pas. Ensuite, pour la concaténation de deux listes et la suppression d'un élément d'une liste, on crée une fonction *verificationEgaliteListe* qui permet de vérifier l'égalité de deux listes et on construit des listes de vérification avec le constructeur.

4.3- Module Composante Connexe

On ne réalise qu'un seul test. On crée un TabComposanteConnexe à partir d'une grille que l'on a remplis à partir d'un fichier. On teste si :

- La structure crée n'est pas vide.
- La structure contient le bon nombre d'éléments.
- Si chaque élément est bien initialisé sans voisins.

On crée alors les voisins pour chaque composante connexe. On teste ensuite si :

- Chaque composante connexe a le bon nombre de cases.
- Chaque composante connexe a la bonne couleur.
- Chaque composante connexe a le bon nombre de voisins.
- Toutes les cases sont bien assignées à une composante connexe.

On teste alors un changement de couleur, et on s'assure que le nombre de cases a bien changé, et que la composante connexe a bien la nouvelle couleur et le bon nombre de cases.

4.4- Module Liste de Composantes Connexes

Ici, nous avons fait deux tests unitaires différents. L'un testant toutes les fonctions de création de liste de composantes connexes et l'autre les autres fonctions de liste de composantes connexes.

La fonction *testCreationListeComposanteConnexe* teste le bon fonctionnement des fonctions *initListeComposanteConnexe*, *estVideListeComposanteConnexe,* en effet, on vérifie que la liste est bien vide en utilisant la fonction *estVideListeComposanteConnexe* qui vérifie si la liste est bien NULL. On teste la fonction *constructeurListeComposanteConnexe* avec la fonction *estVideListeComposanteConnexe*, la liste de composantes connexes ne devant plus être vide après l'ajout d'un élément. Par ailleurs, pour ajouter un élément en particulier, il a été nécessaire de créer la fonction *creeComposanteConnexe*. En effet, il n'est pas utile de créer une fonction n'ajoutant qu'un seul élément. On teste *getValeurListeComposanteConnexe* et *getSuivantListeComposanteConnexe* en vérifiant que la tête de la liste est bien celle ajoutée ainsi que le suivant de la tête est bien exacte également.

La fonction *testFonctionsDiversesListeComposanteConnexe* teste les fonctions *longueurListeComposanteConnexe*, *rechercheElementListeComposanteConnexe*,

supprimeElementListeComposanteConnexe. On teste la fonction *longueurListeComposanteConnexe* en vérifiant que la longueur associée à la liste est bien celle attendue. Pour la fonction *rechercheElementListeComposanteConnexe* on teste si lorsqu'on cherche un élément dans la liste qui est présent la fonction ne renvoie pas NULL tandis qu'un élément ne se trouvant pas dans la liste renvoie NULL. Pour la fonction *supprimeElementListeComposanteConnexe* on regarde si lorsqu'on supprime un élément qui n'est pas dans la liste celle-ci reste inchangée. On testera pour vérifier cela si la longueur de la liste reste la même. Lorsqu'on supprime un élément présent dans la liste, on vérifie si la longueur a été soustraite de moins 1 par rapport au début et également lorsqu'on cherche cet élément dans la liste, la fonction r*echercheElementListeComposanteConnexe* renvoie bien NULL.

Cette dernière fonction nous a apporté certains problèmes au niveau de la libération de mémoire que l'on n'arrive pas à résoudre.