Projet : "Jeu d'inondation" (Flood-It)

Ce projet consiste à étudier des stratégies pour gagner au jeu d'inondation de grilles, appelé souvent Flood-It ou Mad Virus. Vous pouvez facilement trouver des versions de ce jeu sur internet ou sur smartphone.

Ce projet se divise en deux parties réparties sur plusieurs semaines . Afin de ne pas prendre de retard et de pouvoir profiter des séances correspondantes pour chaque partie du projet, un rendu intermédaire est prévu.

La partie 1 a pour but d'implémenter une *séquence aléatoire* pour le jeu : on compare plusieurs implémentations pour la mettre en place : l'objectif est là de comparer les vitesse d'exécution.
La partie 2 consiste à étudier des stratégies pour gagner le plus rapidement possible au jeu d'inondation en termes de nombres d'itérations.
On remarquera à la fin que ces deux critères (vitesse d'exécution et nombre d'itérations s'opposent souvent.

Rendus
- le rendu intermédiaire est constitué de la Partie 1 : il s'agit d'un rendu simple avec 2 pages de rapport.
- le projet dans son intégralité est à rendre pour la séance du TME 11.

Lors du TME final, une mini-soutenance de projet **de tous les membres du binôme** sera l'occasion de montrer votre code, son fonctionnement et ses performances. Pour le contenu du rendu, voir la fiche correspondante sur le site du module.

Il est impératif de **travailler régulièrement** car souvent l'exercice de la semaine est l'application directe de notions qui sont introduites en cours et en TD en parallèle au projet. Tout au long du projet, les chargés de TME sont là pour vous aider et vous guider.

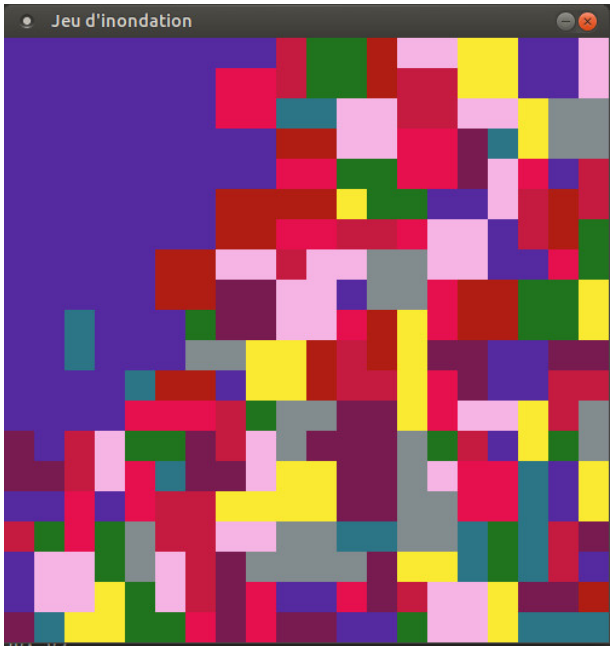
La qualité d'une stratégie sera observée à la fois en nombre d'inondations mais aussi selon la vitesse (complexité) de l'algorithme permettant le choix de l'inondation. Les binômes qui auront eu les meilleurs résultats auront un bonus sur leur note de projet !

Attention : Cet énoncé peut connaître des petites évolutions afin d'apporter des précisions ou des indications : voir les mises à jours sur le site.

Introduction

On considère une grille carrée dont chacune des $n \times n$ cases sont elles-même carrées. Chaque case possède au départ une couleur parmi c couleurs prédéfinies. Dans une telle grille, on dit que deux cases sont *adjacentes* si elles ont un côté horizontal ou vertical en commun. Un ensemble de cases est dit *connexe* si toutes ses cases sont de même couleur et positionnées de façon à ce que l'on puisse se déplacer de n'importe laquelle de ses cases à n'importe quelle autre de ses cases en passant uniquement par des cases adjacentes. On appelle *zone* un ensemble connexe de cases qui est maximal au sens de l'inclusion. On notera *Zsg* (zone supérieure gauche) la zone contenant la case située en haut à gauche de la grille.
L'action *"inonder"* correspond au fait de colorier d'une nouvelle couleur la zone Zsg. On peut remarquer que si la nouvelle couleur est choisie parmi les couleurs de cases adjacentes à Zsg, alors ces cases s'ajoutent à la zone Zsg. L'objectif du *jeu d'inondation* est de colorier totalement la grille avec une seule couleur en utilisant le moins de fois possible l'action "inonder".
On peut remarquer qu'à chaque tour de jeu, le joueur doit uniquement décider quelle couleur sera utilisée pour l'inondation de la zone supérieure gauche. On appelle *séquence de jeu* la séquence des choix successifs de couleurs d'un joueur.

Dans ce projet, nous n'allons pas considérer le jeu d'inondation en version interactive, c'est-à-dire une version où la séquence de jeu est décidé par un joueur humain. Nous allons considérer au contraire que la séquence est décidée par l'ordinateur. L'objectif est ici de rechercher de bonnes stratégies pour gagner à ce jeu, c'est-à-dire déterminer un nombre minimum d'inondation nécessaire pour gagner au jeu.



API Grille

• La SDL (Simple DirectMedia Layer)

Pour ce projet, vous disposez d'une API vous permettant d'afficher les éléments nécessaires à ce jeu d'inondation. Cette API est accessible sur le site du module. Elle comporte 2 bibliothèques API_Gene_instance et API_Grille.
Elle est basée sur la SDL (Simple DirectMedia Layer) qui doit donc être installée sur votre ordinateur auparavant. A la PPTI, la SDL est déjà installée. Sous linux, il suffit d'installer le paquet `libsdl1.2-dev` (et non l'ancien nom qui était marqué ici) qui correspond aux fichiers de développement de la librairie.
Pour compiler des exécutables avec la SDL, n'oublier pas d'ajouter la librairie dans la ligne de compilation (ajouter le flag `-lSDL` dans la ligne de compilation).

• Modélisation d'une grille de jeu

Une grille de jeu sera modélisée par un variable `int **M` correspondant à une atrice à deux dimensions dont chacune des cases contient le numéro de la couleur correspondante (les couleurs sont numérotées à partir de 0), codée par un tableau de tableau d'entiers à deux dimensions. La case supérieure gauche sera ainsi la case $M[0][0]$.
Deux librairies créees pour ce projet vous permet d'afficher et de générer aléatoirement des matrices :
- Les fonctions de `API_Grille` permette d'afficher la grille à l'écran. Une grille est manipulée par un pointeur sur une variable de type `Grille` qui va permettre de stocker les données nécessaires à l'affichage.
- les fonctions de `API_Gene_instance` peuvent remplir aléatoirement de telles matrices.
Il est à noter que, grâce à `API_Grille`, vous pouvez dissocier totalement la matrice M de son affichage : ainsi, vous allez faire évoluer le contenu de la matrice M lors du jeu ET vous demanderez par l'API_Grille de mettre à jour l'affichage. **Vu le temps d'affichage long pour de grandes grilles, il est recommandé de prévoir dès le début que votre programme puisse facilement ne pas faire systématiquement l'affichage.**

• `API_Gene_instance.h` et `API_Gene_instance.c`

- `Gene_instance_init`
Affecte des valeurs à une matrice M préalablement allouée de manière ce que les cases de la matrice soient affectées chacune à une couleur choisie aléatoirement.
Prend en paramètre la dimension n de la matrice, un nombre maximal de couleurs, un niveau de difficulté, une graine de génération aléatoire et une matrice entière M.
Le niveau de difficulté est un nombre entre 0 et 100 correspondant à la taille moyenne d'une zone dans l'instance. Cette taille est ainsi un pourcentage du nombre de cases.

• `API_Grille.h` et `API_Grille.c`
Contient la structure `Grille` qui contient les données nécessaires à la gestion de l'affichage tout au long d'un programme.

- `Grille_init`
Fonction d'initialisation d'un élément Grille de manière à ce que cet élément contienne par la suite toutes les données nécessaires à l'affichage de la grille.
Prend en paramètre la dimension n de la grillem et le nombre de couleurs ainsi que la taille en pixel désirée pour l'affichage de la grille à l'écran. Cette taille doit être inférieur à la résolution de l'écran.
En cas d'affichage trop réduit par rapport au nombre de cases, l'affichage correspondant n'affichera pas toutes les cases mais seulement certaines cases de manière à permettre un affichage représentatif de l'instance. De plus, dans ce cas, l'affichage est calculée pour être esthétiquement correcte, quitte à ne pas afficher sur la surface totale souhaitée de l'écran.

- `Grille_ouvre_fenetre`
Fonction permettant l'ouverture de la fenêtre graphique contenant une grille ainsi que l'affichage total de la grille.

- `Grille_ferme_fenetre`
Fonction permettant la fermeture de la fenêtre graphique contenant une grille.

- `Grille_attribue_couleur_case`
Fonction permettant l'affectation d'une case a une nouvelle couleur. Cette fonction ne provoque pas d'affichage à l'écran. Il faut pour cela utiliser la fonction suivante.

- `Grille_redessine_Grille`
Fonction permettant de redessiner une grille.

- `Grille_attente_touche`
Fonction permettant de mettre en pause le programme en attente de la pression d une touche.

- `Grille_free`
Liberation memoire d'un élément grille.

Remarque : si vous désirez fermer la fenêtre d'affichage en cours de programme, il se peut que, suivant les systèmes, la fenêtre refuse de se fermer. Si vous êtes sous ubuntu, afin de fermer cette fenêtre, vous pouvez lancer la commande `xkill` à partir d'un autre terminal puis cliquer sur la fenêtre récalcitrante.

Partie 1 : Séquence aléatoire

L'objectif de cette Partie 1 est d'implémenter une solution pour ce jeu qui repose sur un tirage alétaoire : on tire au sort la couleur à jouer ! Et oui, cela suffit pour finir ce jeu. On verra dans la Partie 2 comment faire le moins d'itérations possibles.
Cette Partie 1 n'a pas donc pour but de trouver le plus petit nombre d'itérations possibles, mais d'implémenter le jeu pour qu'il soit rapide : en effet, on ne connaît pas a priori la Zsg : il faut donc la calculer pour pouvoir l'inonder. Cette partie propose 3 façons de réaliser cela.

Q 0.1 Récupérer l'archive du Projet.
Il contient :

- Les 2 API `API_Grille` et `API_Gene`
- Une librairie `Liste_case` qui manipule une liste de cases de la matrice (c'est-à-dire tel que l'élément de la liste contient les coordonnées d'une case) : elle servira à coder les cases de la Zsg.
- et le squelette de votre programme `Flood-It_Partie1.c`.

Ce programme demande à l'utilisateur d'entrer en ligne de commande la dimension de la grille, un nombre de couleur, un niveau de difficulté, une graine de génération aléatoire, l'exo requis et si une variable binaire pour l'affichage.
Ce squelette permet de créer une matrice de jeu en utilisant `API_Gene_instance` ; puis qui affiche à l'écran une grille correspondant à ces paramètres avec `API_Grille` et enfin arrive à la ligne permettant de lancer le jeu selon les codes à fournir pour chacun des exos suivants.

Tester ce programme pour différentes valeurs d'entrée afin de voir l'affichage suivant les paramètres.

Exercice 1 – Aléatoire récursif

La première version la plus naturelle du code est d'utiliser la récursivité pour déterminer la Zsg.

Q 1.1 Créer une fonction **récursive**
`void trouve_zone_rec(int **M, int dim, int i, int j, int *taille, ListeCase *L);`
qui, étant donnée la matrice M, une case (i, j) de la matrice et une couleur cl, permet d'affecter dans la liste chaînée L les cases de la zone de la grille contenant la case (i, j). De plus, cette fonction met à jour une variable *taille* passée en paramètre qui indique le nombre de cases contenues dans la zone.

Q 1.2 Créer une fonction
`int sequence_aleatoire_rec(int **M, Grille *G, int dim, int nbcl, int aff);`
qui utilise la fonction précédente pour jouer avec une séquence de jeu alétatoire (tirage au sort une couleur différente de la couleur de la Zsg). La fonction retourne le nombre de changements de couleurs nécessaires pour gagner (ne comptabiliser que les tirages de couleurs différents de la couleur de la Zsg).
Cette fonction doit également réafficher la grille après modification (c'est-à-dire mettre à jour les affectations dans la structure d'affichage Grille puis l'affichage de la grille). **Pensez à ajouter un paramètre `aff` qui est à 1 si ce réaffichage doit avoir lieu ou non**.

Q 1.3 Insérer les deux fonctions précédentes dans le programme. Tester-le pour différentes tailles et difficulté. Que constatez-vous ?
Vous effectuerez vos remarques avec ou sans affichage de la grille de jeu (on peut aussi affichant la grille que de temps en temps).

Exercice 2 – Aléatoire dérécursifié (Exo Bonus)

A la fin de la questions précédente, on a remarqué que la version récursive précédente est limitée. Nous allons contourner cette difficulté en proposant une version non-récursive.

On peut remarquer que la bibliothèque `Liste_case` peut être utilisée pour manipuler une Pile de cases. Les méthodes Dépile et Empile sont exactement les méthodes `ajoute_en_tete` et `enleve_en_tete`.

Q 2.1 Dérécursifier la fonction trouve_zone_rec demandée en 1.1 de manière à obtenir une nouvelle fonction `trouve_zone_imp` qui utilise une pile de cases. Le principe de dérécursification d'une fonction consiste à utiliser une pile pour réaliser les mêmes opérations qu'un appel récursif.

Q 2.2 Ajouter les fonctions précédents dans votre programme pour un lancement de l'exo1bis (vous pourrez donc lancer au choix la version recursive ou imperative en modifiant la ligne de commande). Tester à nouveau. Que constatez-vous ?

Exercice 3 – Allons plus vite ! (Structure acyclique)

Les versions précédentes du jeu sont très lentes car elles ré-énumèrent à chaque itération les mêmes cases de la zone Zsg. Une méthode plus rapide, appelée ici `version_rapide`, va consister à conserver entre chaque itération la liste des cases de la zone Zsg. Pour permettre de connaître les cases qui vont s'ajouter à chaque tour de jeux, on doit alors connaître à tout moment les cases qui bordent Zsg. On appelle ici *bordure* de la zone Zsg de couleur c l'ensemble des cases de couleurs différentes de c qui sont adjacentes à cette zone.

Pour coder la zone Zsg et sa bordure, nous allons considérer la structure `S_Zsg` suivante

```
1  typedef struct {
2    int dim;    /* dimension de la grille */
3    int nbcl;     /* nombre de couleurs */
4
5    ListeCase Lzsg;    /* Liste des cases de la zone Zsg */
6    ListeCase *B;      /* Tableau de listes de cases de la bordure*/
7    int **App;     /* Tableau a double entree des appartenances */
8
9  } S_Zsg;
```

qui contient les données suivantes :
- $Lzsg$ de type `Liste_case` qui va contenir la liste de toutes les cases de la zone Zsg ;
- un tableau B de $nbcl$ cases ($nbcl$ étant le nombre de couleurs de l'instance). Le tableau contient les cases de la bordure de Zsg réparties par couleurs : chaque case $B[c]$ est une liste de type `Liste_case` qui contient la liste des cases de couleur c de la bordure de la zone Zsg.

- un tableau *App* de $n \times n$ cases permettant d'indiquer pour une case donnée i, j si elle est dans Zsg, dans sa bordure ou non-encore rencontré lors des explorations : $App[i][j] == -2$ si la case n'appartient ni à $Lzsg$, ni à la bordure ; $App[i][j] == -1$ si la case appartient à $Lzsg$; $App[i][j] == cl$ si la case appartient à $B[cl]$. Ce tableau va permettre de tester rapidement où est situé une des cases de la grille.

Q 3.1 Proposer une bibliothèque correspondant à la structure `S_Zsg` en codant les méthodes suivantes :
- `init_Zsg` qui initialise la structure ;
- `ajoute_Zsg` en $O(1)$ qui ajoute une case dans la liste $Lzsg$;
- `ajoute_Bordure` en $O(1)$ qui ajoute une case dans la bordure d'une couleur cl donnée ;
- `appartient_Zsg` en $O(1)$ qui renvoie vrai si une case est dans $LZsg$;
- `appartient_Bordure` en $O(1)$ qui renvoie vrai si une case est dans la bordure de couleur cl donnée.

Q 3.2 Ajoutez à cette bibliothèque une fonction
`int agrandit_Zsg(int **M, S_Zsg *Z, int cl, int k, int l)`
qui a pour but de mettre à jour les champs $Lzsg$ et B d'une `S_Zsg` lorsque qu'une case k, l de couleur cl, qui est dans la bordure $B[cl]$, doit basculer dans $Lzsg$ (qui sera coloriée de couleur cl). De plus, la fonction retourne le nombre de case qui a été ajouté à $Lzsg$.
La fonction `agrandit_zone` peut être décrite de la façon suivante :
A partir de la case (k, l) de couleur cl, on parcourt (par un algorithme impératif proche de celui de la question 2.1) toute la zone contenant (k, l) de couleur cl. On peut noter que lors d'un tel parcours, on parcourt également nécessairement les cases qui borde cette zone à explorer.
Ainsi, pour chacune des cases rencontrées lors de ce parcours :
- soit il s'agit d'une case de couleur cl, dans ce cas, on l'ajoute à $Lzsg$ si elle n'y est pas déjà ;
- soit il s'agit d'une case de couleur $cl_2 \neq cl$ qui n'est pas déjà dans $B[cl_2]$ et on l'ajoute alors à $B[cl_2]$.
Attention, lors du parcours, à ne pas sortir de la grille !

Q 3.3 En utilisant la fonction précédente, implémenter une fonction `strequence_aleatoire_rapide` qui accélére considérablement le jeu par rapport à l'exercice précédent.
Cette fonction peut suivre le principe suivant :
- Initialiser les listes Lsg et $B[cl]$, $cl = 0, ..., nbcl - 1$ à liste vide.
- Utiliser l'appel à la fonction `agrandit_zone` à partir de la case $(0, 0)$ et de la couleur $M[0][0]$: on obtient alors dans $Lzsg$ et B les valeurs initiales de la zone Zsg et de sa bordure.
- A chaque tour de jeu où le hasard choisit la couleur cl, il faut alors colorier toutes les cases de la zone Zsg à la couleur cl dans M ; puis pour chaque case (i, j) de $B[cl]$, utiliser `agrandit_zone` à partir de (i, j) et de la couleur cl ; enfin, il faut détruire la liste $B[cl]$ (dont les cases ont été ajoutées à Lzg lors de l'exécution de `agrandit_zone`).

Q 3.4 Modifier le programme `Flood-It_alea` pour utiliser cette nouvelle version. Que constatez-vous ?

Q 3.5 En utilisant les 2 versions (ou les 3 version si vous avez fait l'exo bonus) sans aucun affichage, comparer-leur temps d'exécution en fonction du nombre de cases et du nombre de couleurs (la stratégie étant la même, vous obtiendrez des séquences de même taille). Tracez des courbes permettant d'illustrer les comparaisons entre ces 3 versions.
Analysez et commentez les résultats obtenus.

RENDU INTERMEDIAIRE : Vous devez rendre les résultats obtenus pour la partie 1 à la date indiquée par mail. Pour cela, vous soumettrez sous moodle uniquement le fichier .tar contenant les

fichiers source, ainsi que les tailles de grille testées et les réponses aux questions (juste les réponses, pas de rapport demandé) : donc deux pages de rapport au maximum.
Nous vous incitons également à commencer le rapport final qui sera à rendre à la fin..