

## 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 intermédiaire 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 vitesses 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 décidée 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.

Le site du module vous propose deux archives pour les deux versions de la SDL :

en effet Il existe deux versions de la SDL : SDL1.2 et SDL2. Il semble que la SDL2 fonctionne tout aussi bien sous linux que MACOS, alors que SDL1.2 a des problèmes sous MACOS.

- Pour SDL1.2 : installer le paquet `libsdl1.2-dev` qui correspond aux fichiers de développement de la librairie.

Dans le Makefile proposé dans l'archive, noter que pour compiler des exécutables avec la SDL, il ne faut pas oublier d'ajouter la librairie dans la ligne de compilation en ajoutant le flag `-lsdl` dans la ligne de compilation.

- Pour SDL2 : installer le paquet `libsdl2-dev` qui correspond aux fichiers de développement de la librairie.

Dans le Makefile proposé dans l'archive, noter que pour compiler des exécutables avec la SDL, il ne faut pas oublier pas d'ajouter '`sd12-config --cflags`' pour la ligne de commande de compilation et '`sd12-config --libs`' pour la ligne de commande de linkage.

- Modélisation d'une grille de jeu

Une grille de jeu sera modélisée par un variable `int **M` correspondant à une matrice 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 bibliothèques créées 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 grille 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 inférieure 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 à 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 aléatoire : 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 aléatoire (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 afficher 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 accélère considérablement le jeu par rapport à l'exercice précédent.

Cette fonction peut suivre le principe suivant :

- Initialiser les listes *Lsg* et  $B[cl]$ ,  $cl = 0, \dots, nbcl - 1$  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 *Lzsg* 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 soumettez 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..



## Partie 2 : Stratégies pilotées

Nous nous intéressons à présent à déterminer de bonnes stratégies pour gagner à ce jeu d'inondation. Pour cela, nous allons nous appuyer sur la structure de données Graphe.

### Exercice 4 – Graphe des zones

Le niveau de difficulté d'une grille de ce jeu d'inondation dépend de sa taille, mais aussi de la taille de ses zones. En effet, une grille très grande qui aurait des zones elles-aussi très grandes, peut avoir un nombre de zones très petit. Ainsi la difficulté de résolution d'une grille doit être estimée en fonction de ces zones. Le générateur d'instances fourni propose de générer des instances plus ou moins difficiles en utilisant ce principe.

Afin de pouvoir proposer des méthodes efficaces, nous allons manipuler un "graphe des zones" construit à partir de la grille et où chaque zone correspond à un sommet. Ainsi un sommet de ce graphe correspond à une zone, c'est-à-dire à la liste des cases de cette zone. De plus, il existe une arête entre deux sommets-zones  $s_1$  et  $s_2$  s'il existe une case dans la zone  $s_1$  qui est adjacente à une case de la zone  $s_2$ .

Ce graphe peut être défini à partir du code suivant.

```

1  typedef struct sommet Sommet;
2
3  /* Element d'une liste chaînée de pointeurs sur Sommets */
4  typedef struct cellule_som{
5      Sommet * sommet;
6      struct cellule_som * suiv;
7  } Cellule_som;
8
9
10 struct sommet{
11     int num; /* Numero du sommet (sert uniquement a l'affichage) */
12     int cl; /* Couleur d'origine du sommet-zone */
13     ListeCase cases; /* Listes des cases du sommet-zone */
14     int nbcase_som; /* Nombre de cases de cette liste */
15
16     Cellule_som * sommet_adj; /* Liste des arêtes pointeurs sur les sommets
17                                adjacents */
18 };
19
20 typedef struct graphe_zone{
21     int nbsom; /* Nombre de sommets dans le graphe */
22     Cellule_som *som; /* Liste chaînée des sommets du graphe */
23     Sommet ***mat; /* Matrice de pointeurs sur les sommets indiquant
24                    a quel sommet appartient une case (i,j) de la grille */
25 } Graphe_zone;

```

Cette structure de graphe est particulière à ce projet.

Dans cette structure `Graphe_zone`, le graphe contient une liste chaînée de tous ses sommets-zone afin de pouvoir les parcourir facilement. Néanmoins cette liste chaînée ne sera pas efficace pour tous les algorithmes : en effet, l'objectif de cette structure est de permettre efficacement d'indiquer, pour une

case de la grille, à quel sommet-zone la case appartient : c'est pourquoi la structre contient également une matrice `mat` telle que `G.mat[i][j]` soit un pointeur sur le sommet-zone contenant la case  $(i, j)$ .

**Q 4.1** Créer les fonctions suivantes :

- `ajoute_liste_sommet` qui ajoute un pointeur sur `Sommet` à une liste chaînée de `Cellule_som` passée en paramètre.
- `detruit_liste_somme` qui détruit une liste chaînée de `Cellule_som` SANS détruire les sommets pointées par cette liste.
- `ajoute_voisin(Sommet *s1, Sommet *s2` qui met à jour deux sommet `s1` et `s2` en indiquant qu'ils sont adjacents l'un de l'autre
- et une fonction booléenne `adjacent` qui renvoie vrai si et seulement si deux sommets sont adjacents.

**Q 4.2** On s'intéresse à présent à la fonction `creer_graphe_zone` qui va créer le graphe tout entier.

Pour cela, on peut suivre le principe suivant : "créer d'abord les sommets sans les arêtes, puis ensuite créer les arêtes".

Ce principe peut se décrire de la façon suivante :

- Initialiser la structure de graphe avec 0 sommet : la matrice `mat` contient le pointeur `NULL` pour chaque case.
- Pour chaque case  $(i, j)$  de la grille où `G.mat[i][j] == NULL`, créer un sommet vide de cases puis utiliser la fonction `trouve_zone` pour déterminer la zone correspondante. Mettre ensuite à jour la matrice `mat` pour toutes les cases de la zone.
- Une fois que tous les sommets ont été créés et que la matrice `mat[i][j]` donne un pointeur sur le sommets contenant la case  $(i, j)$ , il est possible d'indiquer quelles sont les sommets voisins en parcourant à nouveau toutes les cases de la grille.

Faite bien attention à de ne pas stocker plusieurs fois la même information : lorsque l'on stocke le fait que deux sommets `s1` et `s2` sont adjacents dans le graphe, il faut vérifier que cette information n'est pas déjà dans la structure.

**Q 4.3** Créer une fonction d'affichage du graphe (en mode texte) et tester vos fonction sur de petits exemples.

---

## Exercice 5 – Stratégie max-bordure

---

La bordure vue dans la partie 1 ne permet pas de concevoir des algorithmes intéressants. En effet, la bordure de la partie 1 concerne uniquement les cases qui bordent la zone `Zsg`. Il est bien plus utile de considérer une bordure dans le graphe des zones ! En effet, lorsque l'on colorie la `Zsg`, on assimile une zone adjacente toute entière !

A partir de la structure de graphe définie dans l'exercice précédent, on appellera pour la suite *bordure-graphe* de `Zsg` les sommets du graphe qui sont adjacentes aux sommets composant la `Zsg`. D'un point de vue structure de données, on peut reprendre le principe de la partie 1 : une bordure-graphe sera codée comme un tableau de `nbc` listes chaînées. Mais cette fois-ci ce sont des listes chaînées de pointeurs sur sommets-zones. Chaque liste répertorie les sommet-zones de même couleur.

On s'intéresse ici à une stratégie simple mais très efficace : "toujours choisir la couleur qui est la plus représentée dans la bordure-graphe". Pour cela, on va stocker en plus dans la structure bordure-graphe la taille de chaque liste correspondant aux couleurs représentées dans la bordure.

A tout moment, on devra rapidement savoir si un sommet est :

- dans la zone Zsg
- dans la bordure
- ou non encore visité.

Pour cela, on peut ajouter des champs au struct **Sommet** un champs **int marque**; associé à chaque **Sommet** : il permet d'indiquer le statut d'un sommet (0 si dans Zsg, 1 si dans bordure, 2 si non visité). Attention ! Penser à initialiser au départ tous les champs *marque* des sommets.

**Remarques** : Deux remarques utiles :

- On peut noter que pour afficher rapidement, il est également utile de stocker la liste des sommets composant la Zsg.
- On peut remarquer qu'à présent la matrice *M* des couleurs ne sera plus utilisée...

**Q 5.1** Créer la fonctions de mise à jour de la bordure-graphe : qui met à jour la bordure-graphe en basculant une couleur de la bordure dans la Zsg.

**Q 5.2** Créer la fonction implémentant la stratégie max-bordure.

Noter que le jeu s'arrête quand la bordure est vide pour toute couleur.

**Attention** à bien désallouer les listes de **Cellule\_som** (SANS détruire les **Sommet**) à chaque fois que l'on vide un champ de la bordure.

---

## Exercice 6 – Stratégie parcours en largeur

---

Une autre stratégie simple (et assez efficace) consiste à chercher à incorporer rapidement la zone inférieure droite dans la zone Zsg : en effet dans ce cas, l'instance est coupée en deux, ce qui permet d'avoir une bordure large. Une fois effectuée cette découpe, la stratégie max-bordure précédente sera plus efficace pour terminer l'inondation.

Dans cet exercice, nous allons déterminer la plus courte séquence de couleurs permettant de faire en sorte que la Zsg contienne la case inférieure droite.

**Q 6.1** Montrer qu'une séquence de couleurs de petite taille permettant d'incorporer la zone inférieure droite à la zone Zsg peut se déduire d'un plus court chemin en nombre d'arêtes dans le graphe-zone.

**Q 6.2** Implémenter un algorithme de parcours en largeur adapté à la structure de graphe-zone. Adapter ce code pour obtenir en sortie un plus court chemin en nombre d'arêtes entre deux sommets du graphe. Pour cela, vous pourrez stocker dans le struct *Sommet* les informations suivantes.

```

1  int distance; /* Nombre d'arêtes reliant ce sommet a la racine
2                du parcours en largeur */
3  Sommet *pere; /* Pere du sommet dans l'arborescence du parcours en largeur */

```

Le champ *distance* est à l'initialisation mis à une valeur très grande et correspond à la fin de l'algorithme de parcours à la valeur en nombre d'arcs du plus court chemin. Similairement, le pointeur *pere* est initialisé à NULL au départ puis correspond à la fin de l'algorithme de parcours au père du sommet dans l'arborescence du parcours en largeur : ce pointeur permettra ensuite de déterminer le

chemin entre le sommet racine et un autre sommet du graphe (en remontant l'arborescence à partir de ce dernier sommet).

**Q 6.3** Implémenter la stratégie suivante : tout d'abord effectuer les choix de couleurs correspondant au chemin déterminé par l'algorithme de la question précédente. Ensuite, terminer le jeu en utilisant la stratégie max-bordure

---

### Exercice 7 – D'autres stratégies ? (bonus)

---

**Q 7.1** Proposer vous-même de nouvelles stratégies basées ou non sur les outils présentés ici. Essayez de rivaliser avec les stratégies proposées !

**Q 7.2** La comparaison peut se faire par calcul de temps CPU (en déconnectant l'affichage qui est très lent pour de grandes grilles) mais surtout par calcul de la taille de la séquence de couleurs. Comparer vos stratégies sur cette base !

**Les binômes obtenant les meilleurs scores auront des points en bonus.**

---

### Exercice 8 – RENDU FINAL

---

**Q 8.1 Comparaisons des stratégies :** En utilisant un ensemble suffisant de grilles, tester et comparer les différentes stratégies (aléatoire de la partie 1 et celles de la partie 2) rencontrée ici. Comparer leur nombre d'itérations ainsi que leur temps d'exécution (sans affichage).

#### Q 8.2 Rendu

Rassembler l'ensemble de votre travail comme indiqué dans le document "Comment rendre ?" mis en ligne sur le site du module. Veillez en particuliers à soigner l'analyse statistique de comparaisons !

Vous devez rendre l'intégralité du travail (y compris ce qui a été rendu en partie 1. Il s'agit là du travail complet comportant les fichiers sources, les fichiers résultats **et un rapport complet (y compris les statistiques)**). Veuillez vous reporter au document concernant les rendus sur le site ou sur votre brochure de TD.

#### Q 8.3 Mini-soutenance

A l'occasion d'un contact à distance, vous **présenterez votre travail à votre chargé de TD** lors d'une mini-soutenance de quelques minutes. **A cette occasion**, vous serez interrogé à la fois sur les aspects théoriques du projet et sur le code que vous avez rendu.