Vendredi, le 22 mai 2020.

Rapport du Projet de Structures de Données

Titre du Projet : “Jeu d’inondation” (Flood-It)

Noms du binôme :

Saichi Lina.

Meziane Mohamed.

Groupe : Mono-disciplinaire Informatique , groupe 06.

Responsable : Daniela Genius.

Introduction :

Le jeu implémenté dans ce projet est le jeu d'inondation connu sous le nom de Flood-it.

Le principe de ce jeu est très simple : nous disposons d'une zone de jeu carrée composée de `dim` cases par `dim` cases (tel que *dim* est un nombre entier qui représente la dimension de la grille).
Chacune des cases est remplie d'une couleur sur `nbcl` possibles. Nous devons nous arranger pour que toutes les cases soient de la même couleur.

Pour cela, le programme choisit une couleur pour la case en haut à gauche et cela peint toutes les cases adjacentes de cette couleur : le raz de marée. On dira qu'une case est adjacente si elle est de la couleur de la première case avant le changement et si elle touche une case de cette couleur soit par le haut/bas, soit par la gauche ou la droite (on ne considère pas les cases en diagonales).
Nous noterons la `Zsg` (zone supérieure gauche) la zone contenant la case située en haute a gauche de la grille.

Reformulation du sujet :

Dans la première partie du projet, nous avons pour but d'implémenter une séquence aléatoire pour ce jeu. Nous allons comparer plusieurs implémentations pour le mettre en place.Notre objectif a la fin sera de comparer les vitesses d'exécution de ces implémentations. On évaluera donc leur efficacité en temps de calcul.

Dans la seconde partie, nous allons étudier les meilleurs stratégies pour gagner a ce jeu (Dans la 1ere partie on développe la rapidité et ici on développera aussi le nombre d'itérations pour avoir des résultats meilleures) . Pour cela nous allons utiliser la structure Graphe.

PARTIE I :

Description du workflow :

Dans cette partie , on s'intéresse à implémenter une solution qui repose sur un tirage aléatoire en essayent de faire marcher le programme en moins du temps possible sans prendre compte de son nombre d'itérations.
Pour cela on a implémenté 3 versions :

1- version récursive :

Dans le fichier fonctions exo1.c :

trouve_zone_rec : qui renvoie dans une liste l'ensemble des cases adjacentes a une case dont les coordonnées sont passées en paramètre de la fonction.

séquence_aléatoire_rapide : qui utilise la première fonction pour itérer sur l'ensemble des cases de la grille en jouant avec une séquence de jeu aléatoire.Cette fonction renvoie le nombre de changements de couleurs nécessaires pour gagner.

2-version itérative :

Dans le fichier exercice 2.c :

Nous avons transformé les deux fonctions récursives en itératives en utilisant une pile de cases. Nous avons utilisé les deux fonctions similaires à empiler et dépiler fournies dans l'archive Liste Case : ajoute en tête et enlève en tête

3-version rapide :

Dans le fichier version rapide.c :
La différence avec les deux autres versions c'est que celle-la conserve entre chaque itération la liste des cases de la zone supérieure gauche ainsi que celles qui appartiennent à sa bordure. Nous avons donc implémenté la structure S_Zsg qui contient une liste de cases nommée *Lzsg* qui représente la liste des cases de la Zsg, ainsi qu'un tableau *B* ou chaque case pointe vers une liste de cases qui font partie de la bordure et qui ont la même couleur. Par exemple, B[0] contient la liste des cases qui bordent la Zsg et qui sont de couleur 0.
On peut donc savoir entre deux itérations quelles sont les cases qui se sont ajoutées à la Zsg.
Pour implémenter cette version nous avons codé les fonctions : **agrandit_zsg** qui fait basculer une case dans la zsg ainsi que toutes ses cases adjacentes de la même couleur, ainsi que la fonction **sequence_aleatoire_rapide** qui fait appel a la fonction précédente et fait jouer le jeu. Nous avons rajouté la fonction **detruit_zone** qui désalloue la mémoire des structures utilisées.

Réponses aux questions posées:

1.3 : Nous remarquons que la récursivité nous cause une erreur de segmentation pour des valeurs de dimensions élevées.

Cela s'explique par le grand nombre d'appel récursifs effectués de plus en plus profond , sachant que chaque appel utilise la pile de l'ordinateur gérée avec la mémoire automatique ,et si on ne dispose plus de mémoire et les appels sont aussi en cours ça nous cause l'erreur de segmentation ou bien stack-overflow. Cette dernière erreur arrive lorsque les adresses de retour successives se

cumulent dans la pile jusqu'à ce que son contenu dépasse l'espace qui lui est alloué. Lorsque la pile dispose de son propre segment, certains systèmes le détectent et affichent le message "Stack overflow".

2.2 : Maintenant qu'on gère manuellement la pile on constate qu'on a plus le problème de stack-overflow ou bien d'erreur de segmentation.

Cependant, le programme n'a pas évolué en terme de vitesse d'exécution.

3.4 Nous constatons cette fois-ci que l'exécution du programme s'effectue en une vitesse plus grande par rapport au 1er et 2eme exercice. Le programme devient plus rapide.

3.5 Comparaison entre les 3 versions en terme de temps d'exécution :

1. En fonction du nombre de cases :

Afin de comparer le temps des 3 versions en fonction du nombre de cases (paramètre *dim* donné au programme), nous allons fixer tous les autres paramètres et faire varier la dimension de la grille.

Voici le test effectué :

- dimension = variante
- nombre de couleurs = 20
- niveau de difficulté = 20

- graine = 8
- exercice = { 1, 2, 3 }
- affichage = 0

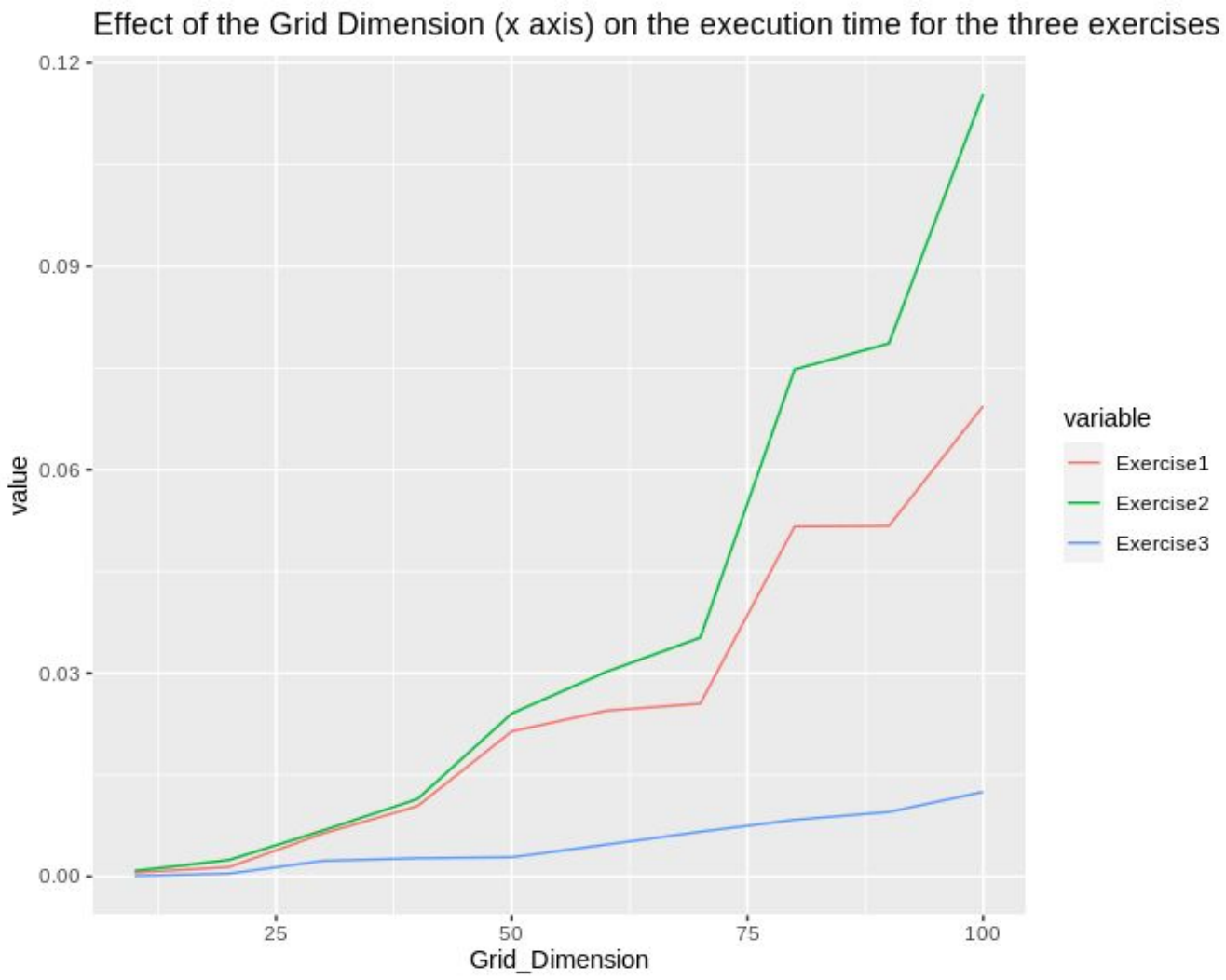
Les résultats obtenus sont représentés dans ce tableau : le temps y est exprimé en secondes.

Tab2 : variation du temps d'exécution du programme en fonction du nombre de cases pour chacun des 3 exercices.

Dimension	Exercice 1	Exercice 2	Exercice 3
10	0.000582	0.000839	0.000093
20	0.001394	0.002440	0.000438
30	0.006352	0.006806	0.002314
40	0.010385	0.011434	0.002702
50	0.021409	0.024046	0.002846
60	0.024445	0.030178	0.004712
70	0.025506	0.035237	0.006604
80	0.051617	0.074774	0.008368
90	0.051702	0.078610	0.009527
100	0.069379	0.115375	0.012479

Le tableau 1 résume les temps d'exécution respectifs pour les trois exercices en fonction du nombre de dimensions. Nous observons que le rang des valeurs des temps d'exécution varie de l'ordre des microsecondes aux millisecondes, et passe d'une valeur minimale de 93 microsecondes à 69 millisecondes.

Le graphe traduisant ces données :



La figure 1 représente l'évolution du temps d'exécution de chaque exercice en fonction du nombre de cases (la dimension de la grille). Nous observons que les courbes rouge et verte représentant l'exercice 1 et 2 respectivement ont une

tendance quasi-similaire, contrairement à la courbe de l'exercice 3. De plus, pour des grilles de dimension inférieure à 60, les deux courbes se superposent. A partir de 3600 cases (60 dimensions), le temps d'exécution de l'exercice 2 croît plus vite que l'exercice 1. Ceci se traduit par la courbe verte (Exercice 2) qui se trouve au dessus de la courbe rouge (Exercice 1).

Concrètement, pour un nombre de cases égal à 4900 (dimension = 70) , les temps d'exécution de l'exercice 1 et l' exercice 2 sont de de 0.02 et 0.03s respectivement.
Pour 10 000 cases (5000 x 2), le temps d'exécution de l'exercice 1 et 2 sont de 0.06 et 0.1s respectivement, soit un écart de 0.04s. Cela se traduit par une multiplication de l'écart du temps d'exécution par 4 lorsque le nombre de cases à double (passage de 5000 à 10000 cases).
Par conséquent, l'exercice deux montre un temps d'exécution plus important , et est donc plus lent que les deux autres exercices.
Enfin, pour la figure bleue qui représente l'exercice 3, nous observons que son temps d'exécution évolue moins vite comparé aux deux exercices précédents. En effet, les temps d'exécution nécessaires pour 60 et 100 dimensions sont de 4 et 10 millisecondes respectivement. Ainsi, le passage de 70 à 100 dimensions, soit 6400 cases, nécessite un temps d'exécution de 6 millisecondes. Cette durée est d'un ordre de grandeur plus petit que le temps d'exécution des deux autres exercices.

2. En fonction du nombre de couleurs :

Afin de comparer le temps des 3 versions en fonction du nombre de couleurs (paramètre `nbcl` donné au programme), nous allons fixer tous les autres paramètres et faire varier le nombre de couleurs.

Voici le test effectué :

- dimension = 50

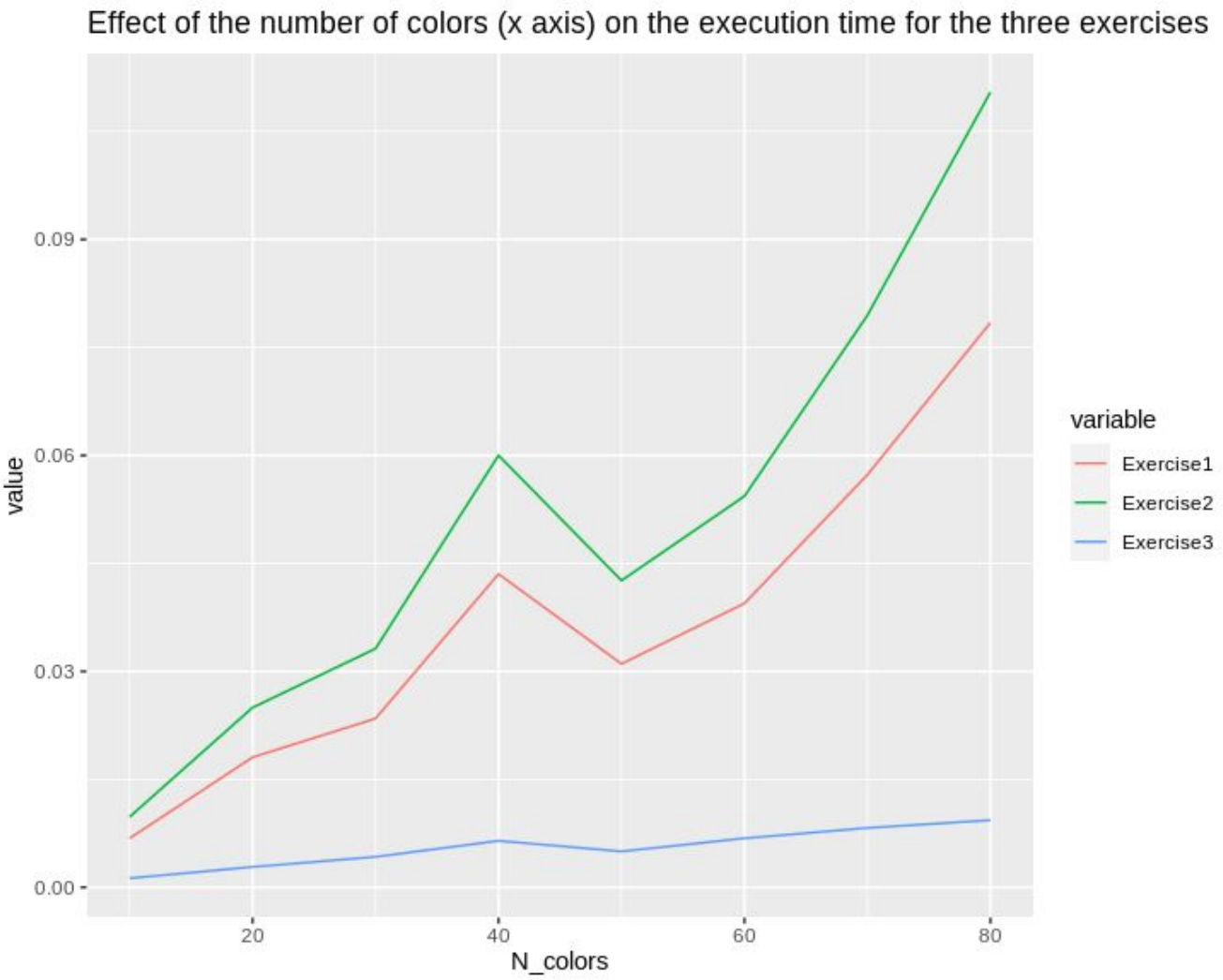
- nombre de couleurs = variante
- niveau de difficulté = 20
- graine = 8
- exercice = { 1, 2, 3 }
- affichage = 0

Les résultats obtenus sont représentés dans ce tableau : le temps y est exprimé en secondes.

Tab1 : variation du temps d'exécution du programme en fonction du nombre de couleurs pour chacun des 3 exercices.

Nombre de couleurs	Temps pour l'exercice 1	Temps pour l'exercice 2	Temps pour l'exercice 3
10	0.006825	0.009794	0.001277
20	0.018080	0.024981	0.002851
30	0.023460	0.033175	0.004250
40	0.043530	0.060010	0.006472
50	0.031053	0.042627	0.005001
60	0.039457	0.054351	0.006825
70	0.057319	0.079471	0.008268
80	0.078434	0.110457	0.009343

Nous avons représenté les données de ce tableau dans un graphe :



Le graphe ci-dessus représente l'évolution du temps d'exécution de chaque exercice en fonction du nombre de couleurs.
Dans ce graphe, on peut observer que les courbes rouge et verte qui représentent l'exercice 1 et 2 respectivement ont la même tendance.
Néanmoins, on observe une plus grande lenteur du cote de l'exercice 2.
En effet, pour un nombre de couleurs égal à 40, le temps d'exécution des exercices 1 et 2 sont de 0.04 et 0.06 s.

On observe également, qu'à partir de 60 couleurs, le temps d'exécution de ces deux exercices devient très grand et est strictement croissant.

Enfin, pour la figure bleue qui représente l'exercice 3, on observe que le temps d'exécution est indépendant du nombre de couleur.
En effet, pour un nombre de couleurs égal à 20 le temps d'exécution est de 0.002, et pour un nombre de couleur égal à 80, le temps d'exécution est de 0.009. On peut dire que c'est à peu près le même ordre de grandeur (comparé aux deux autres exercices ou le temps a décuplé).
On observe également que le temps d'exécution de l'exercice 3 est largement plus petit que celui des deux autres exercices.

Cela peut s'expliquer par le fait que dans l'exercice 3 , nous conservons à chaque itération la liste des cases de la Zsg. Alors que, dans les exercices précédents nous ré-énumérons à chaque itérations les mêmes cases de la Zsg.

Par ailleurs, quelque soit la version utilisée, le nombre d'itérations reste le même. Et donc, la rapidité de l'exercice 3 peut s'expliquer par le fait que nous ne refaisons pas les mêmes calculs à chaque itération.

Conclusion partielle :

On en conclut que la structure acyclique représentée dans l'exercice 3 est plus rapide que la version récursive (exercice 1) qui elle même est plus rapide que la version itérative (exercice 2) et ce quelque soit le nombre de cases ou le nombre de couleurs.

Conclusion premiere partie :

D'après les résultats obtenus ainsi que les graphes d'évolution, on en déduit que la meilleure structure est celle de l'exercice 3.
Ainsi, si nous voudrions implémenter une séquence de jeu aléatoire pour le jeu d'inondation, il serait préférable d'utiliser cette structure, car elle nous permet de minimiser le temps d'exécution et d'aller plus vite. Par ailleurs, le fait de connaître a chaque itération les cases qui bordent la Zsg, nous permettront de réviser notre stratégie de choix de la couleur. En effet, au lieu d'un tirage aléatoire on pourrait compter le nombre de cases de la bordure qui ont la même couleur afin de jouer cette dernière et cela nous permettra d'ajouter un plus grand nombre de cases à la Zsg.
En guise de perspective il serait intéressant d'utiliser la bordure de la Zsg lors d'une séquence de jeu décidée par un joueur humain en lui proposant les couleurs de la bordure afin d'effectuer son action d'inondation.

PARTIE II :

Dans cette deuxième partie, nous allons étudier les meilleurs stratégies pour gagner a ce jeu (Dans la 1ere partie on développe la rapidité et ici on développera aussi le nombre d'itérations pour avoir des résultats meilleures) . Pour cela nous allons utiliser la structure Graphe.

Dans les fichiers de l'exercice 4:

Le fichier header contient : les structures du graphe, des sommets et de la liste chaînée des sommets.
Commençons par la structure du sommet : celle-ci contient le *numéro* de ce sommet (qui est unique et est attribué à la création du graphe) , un entier *cl* qui est la couleur de ce sommet, une ListeCase *cases* ou on stocke toutes les cases qui constituent ce sommet (ce sont des cases adjacents et de même couleur qu'on récupérera avec la fonction trouve_zone de la première partie et qui pointeront toutes vers le même sommet qui leur correspond),
un entier *nb_case_som* qui est la longueur de la liste que je viens de citer, et enfin une *Liste de sommets* adjacents a ce sommet.
Ensuite, la structure Cellule_som qui est simplement une liste chaînée de sommets.
Enfin, la structure du Graphe_zone qui contiendra la *liste de tous les sommets*, un entier *nb_som* qui est la longueur de cette dernière, et enfin une *matrice* de pointeurs qui indique pour chaque case a quel sommet elle pointe.

le fichier.c contient quant à lui les fonctions **ajoute_liste_sommet** qui ajoute un sommet à une liste de sommets, **detruire_liste_sommet** qui détruit une liste de sommets sans libérer les sommets.
Par ailleurs , elle contient la fonction **creer_graphe_zone** qui permet de créer un graphe et d'initialiser tous ses champs, ainsi que **detruire_graphe_zone** qui permettra de désallouer la mémoire utilisée. Elle contient également quelques fonctions d'affichage.

Dans les fichiers de l’exercice 5:

Dans cet exercice, on s'intéresse à manipuler la bordure de la zone supérieure gauche pour faire jouer le jeu, celle-ci sera composée non pas de cases adjacentes, mais de sommets qui jouxtent la zone supérieure gauche. Ainsi,lorsqu’on cherchera à agrandir la zone supérieure gauche, on assimilera un sommet avec toutes les cases qui le constituent. Et Bien-sur pour un souci d’optimisation on choisira un sommet avec le plus grand nombre de cases.
On a créé la structure Somme_Zsg qui contient :
Une liste de sommets *Lzsg* qui représentent les sommets de la Zsg, ainsi qu’un tableau *B* ou chaque case pointe vers une liste qui représente un ensemble de sommets de la bordure et qui ont la même couleur. Par exemple : B[0] sera une liste de sommets qui bordent la Zsg et qui ont la couleur 0.

L’objectif c’est donc de mettre à jour la bordure a chaque coups de jeu en basculant a chaque fois la couleur qui a le plus de sommets dans la bordure. afin de réaliser cela on a créé les fonctions suivantes :
-**maj :** qui permet de mettre à jour la bordure-graphe en basculant une couleur de la bordure dans la Zsg
-max_bordure : elle crée et initialise le graphe zone ainsi que la structure Somme_Zsg et appelle la fonction maj précédente pour faire jouer le jeu.

Dans les fichiers de l’exercice 6 :

Dans cet exercice nous allons étudier la stratégie qui consiste à chercher la plus courte séquence de changement de couleurs qui permettra d’incorporer rapidement la zone inférieure droite à la Zsg.
Nous allons tout d’abord effectuer un parcours en largeur du graphe qui va nous permettre de traiter le problème de la plus courte chaîne : calculer, pour tout sommet u , le nombre minimum d’arêtes d’une chaîne de u à une racine qu’on donne en paramètre. Pour calculer le parcours en largeur du graphe, on utilisera une structure de File.
Au démarrage de l’algorithme, les distances sont initialisées à l’infini, sauf celle de l’origine s. La file permet de gérer la bordure du sous-parcours en largeur. La gestion de la priorité assure que l’on choisit tout le temps un adjacent du premier sommet ouvert du sous-parcours. Le test sur la distance permet aussi de s’assurer que tout sommet placé dans la file n’est ni déjà ajoutée, ni dans F.

Enfin, nous allons écrire une fonction **jeu_strategie** qui va d’abord effectuer les choix de couleurs correspondant au chemin déterminé par l’algorithme du parcours en largeur. Ensuite, terminer le jeu en utilisant la stratégie max-bordure.

Réponses aux questions posées:

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

Dans notre projet, le graphe zone G est composé d'un ensemble de sommets. Chaque sommet possède une couleur ainsi qu'une une liste de sommets adjacents.
Une arête est la représentation de cette adjacence : on dit qu'il existe une arête entre deux sommets si ces deux derniers sont adjacents.
Par la suite, nous voulons calculer un parcours en largeur de notre graphe, un parcours en largeur d'origine s est un parcours L=(s,v1,v2,...,vn) tel que pour tout sous parcours Lk = (v1,..., vk) avec k < n, vk+1 est un sommet adjacent du premier sommet ouvert de Lk.
Ce parcours visite les sommets niveau par niveau: d'abord tous les sommets à distance 1 de s, puis à distance 2, puis 3..etc
A partir de ça, nous allons construire une graphe de liaison en largeur et oriente de notre graphe G qu'on appellera A(L), alors pour tout sous parcours Lk = (v1,..., vk) avec k < n, vk+1 a pour unique père le premier sommet ouvert de Lk.
Maintenant on sait que pour tout sommet u, le chemin de la racine s à u de A(L) est associé à une plus courte chaîne de G entre s et u.
Enfin, une plus petite séquence de couleurs peut se déduire de cette plus courte chaîne, car pour arriver à joindre la Zsg et la Zid nous allons commencer par le sommet inférieur droit et remonter de père en père jusqu'à arriver au sommet supérieur gauche, et donc de cette liste généalogique de sommets, nous aurons une séquence de couleurs qui correspondent aux couleurs de ces sommets.

8.1 : Comparaison entre les 2 versions de la partie II en terme de temps d'exécution et nombre d'itérations :

1.En fonction du nombre de cases :

Afin de comparer le temps des 2 versions en fonction du nombre de cases (paramètre `dim` donné au programme), nous allons fixer tous les autres paramètres et faire varier la dimension de la grille.

Voici le test effectué :

- dimension = variante
- nombre de couleurs = 20
- niveau de difficulté = 20
- graine = 8
- exercice = { 5, 6 }
- affichage = 0

Les résultats obtenus sont représentés dans ce tableau : le temps y est exprimé en secondes.

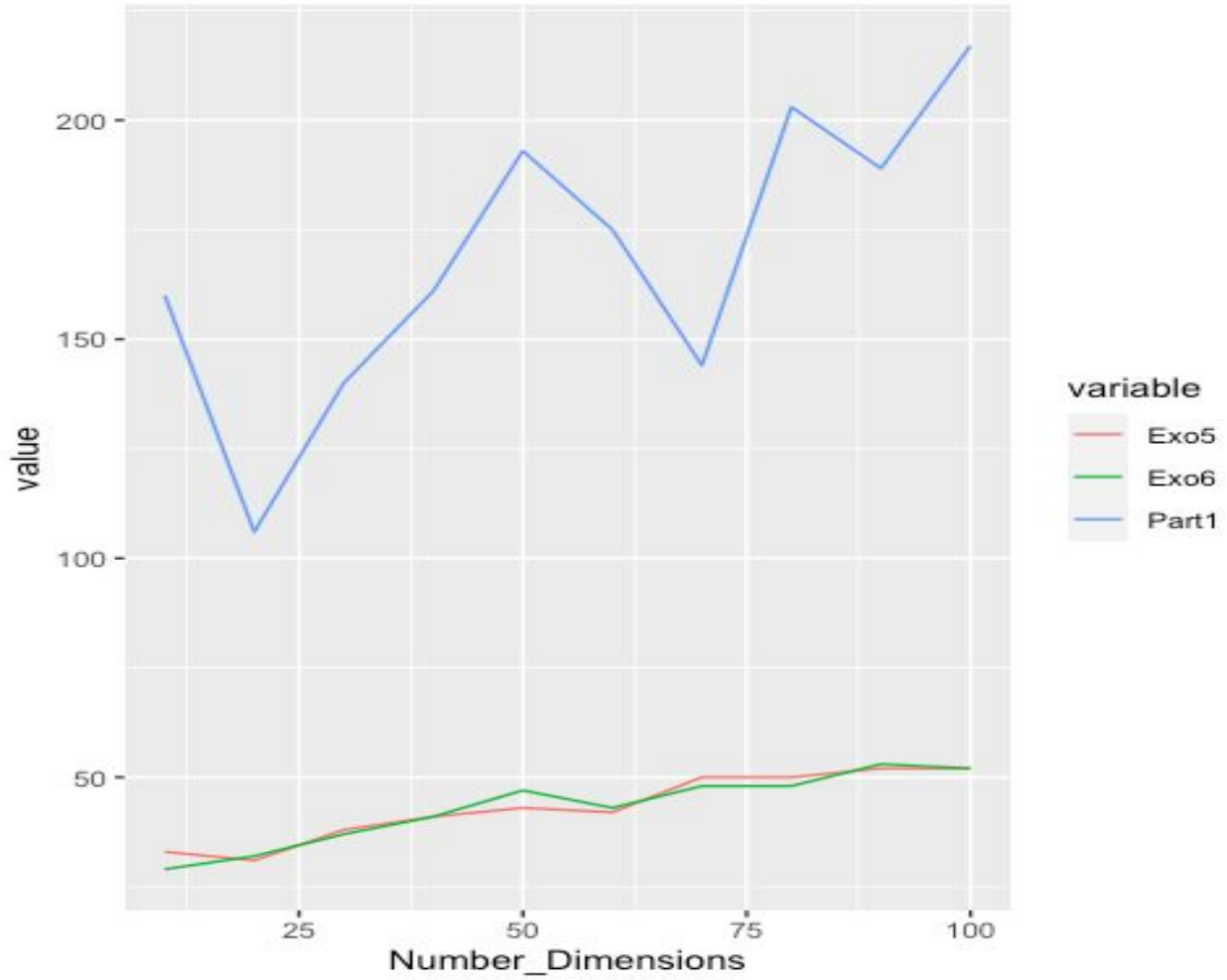
Tab3 : variation du temps d'exécution du programme en fonction du nombre de couleurs pour chacun des exercices 5 et 6.

Dimension	Exercice 5		Exercice 6	
	Temps	Iterations	Temps	Iterations
10	0.000074	33	0.000093	29
20	0.000142	31	0.000187	32
30	0.000232	38	0.000245	37
40	0.000352	41	0.000496	41
50	0.000532	43	0.000584	47
60	0.000816	42	0.000773	43

70	0.000967	50	0.000968	48
80	0.001227	50	0.00131	48
90	0.001778	52	0.002502	53
100	0.002460	52	0.002079	52

Nous avons représenté ces résultats dans deux graphes : l'un représente le nombre d'itérations entre les exercices 5 et 6 ainsi que l'un des exercices de la partie 1 (ici le choix de l'exercice importe peu car les 3 premiers ont tous le même nombre d'itérations) , et l'autre représente la variation du temps d'exécution des exercices 5 et 6 ainsi que **l'exercice 3** de la première partie. Nous justifions le fait de choisir de comparer l'exercice 3 seulement car il s'est révélé être le plus rapide entre les versions de la première partie.

Graphe 1 :



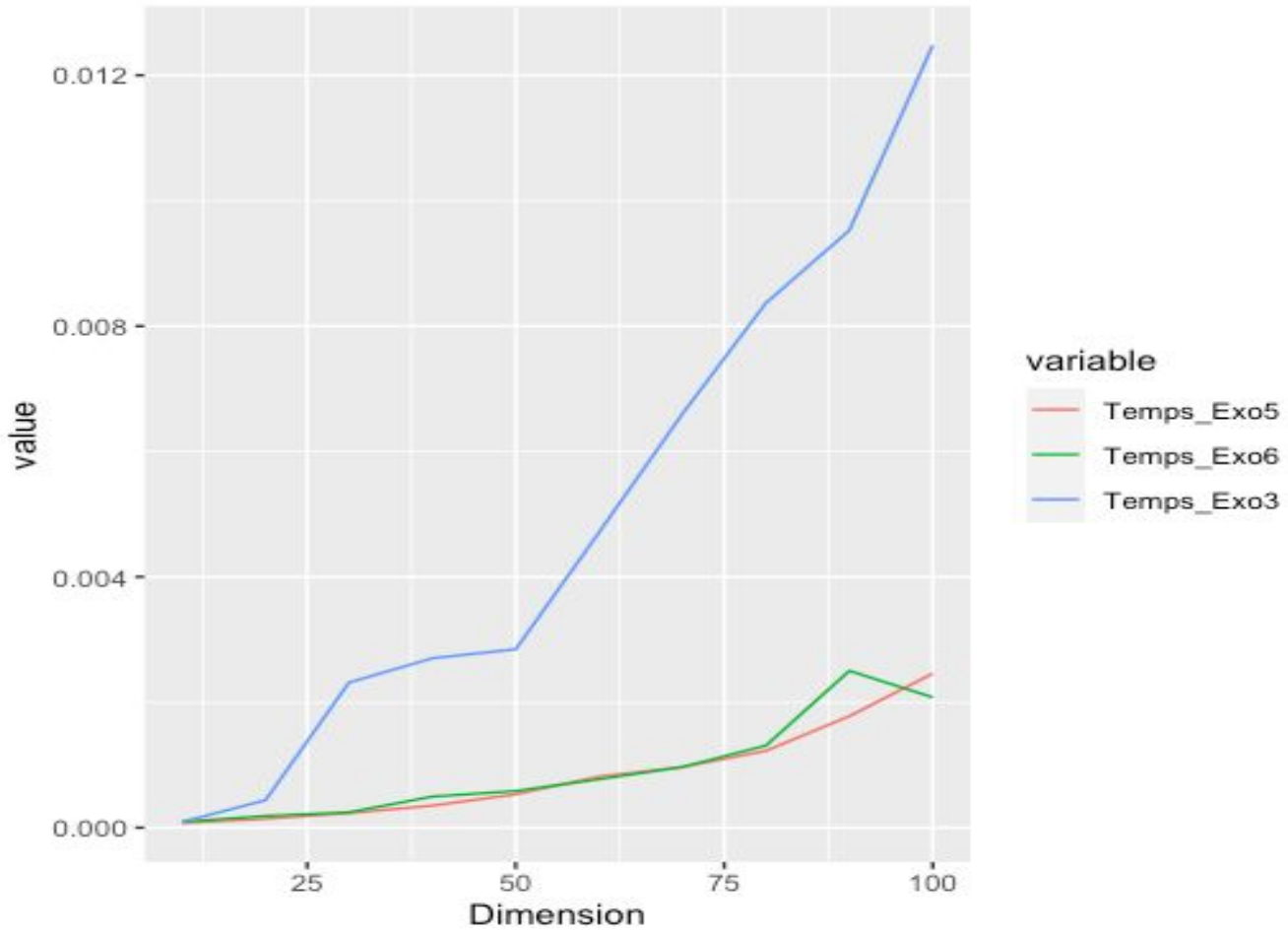
Effect of the Grid Dimension on the number of iterations for both exercices 5,6 and 3.

Le graphe ci-dessus représente l'évolution du nombre d'itérations des exercices 5,6 ainsi que l'exercice 3 en fonction de la dimension de la grille.
Dans ce graphe, on peut observer plusieurs fluctuations en ce qui concerne l'exercice 3. Par exemple pour une dimension de 50 et 90, le nombre d'itérations est de 193 et 189.
De plus, on observe que l'exercice 5 et 6 ont une même distribution avec une légère différence entre les deux. Néanmoins, il augmentent tous les deux en fonction de la dimension de la grille. Pour une dimension de 100, on observe

que le nombre d'itérations des exercices 5 et 6 sont de 50 itérations, tandis que l'exercice 3 il s'élève à plus de 200 itérations**.**

La différence significative entre ces résultats peut être justifiée par le fait que dans la première partie, le choix des couleurs se fait d'une manière aléatoire, le programme alors prend plus de temps pour trouver les bonnes combinaisons de couleurs qui permettront de finir le jeu. Alors que, dans la deuxième partie on adopte une stratégie bien définie, qui est que dans l'exercice 5, nous choisissons la couleur du sommet de la bordure qui a le plus de cases, ce qui permet d'incorporer à la Zsf le plus de cases possibles en une seule fois et d'aller plus vite. Dans l'exercice 6, nous cherchons d'abord à trouver la plus courte séquence de couleurs qui permettent de joindre la Zsg à la Zid, puis nous continuons avec la stratégie de l'exercice 5. Les différences entre l'exercice 5 et 6 peuvent être expliquées par le fait que dans certains cas, la bordure qui sépare la Zsg et Zid est large, ce qui engendrera une plus grande distance entre les deux sommets.

Graphe 2 :



Effect of the Grid Dimension on the execution time for both exercices 5,6 and 3.

Le graphe ci-dessus représente l'évolution du temps d'exécution (en secondes) des exercices 5,6 ainsi que l'exercice 3 en fonction du nombre de cases de la grille.

Dans ce graphe, on peut observer une plus grande lenteur en ce qui concerne l'exercice de la première partie. Il garde néanmoins une allure strictement croissante.

De plus, on observe que l'exercice 5 et 6 ont une même distribution dans l'intervalle [0,80]. Néanmoins à partir de 80, on observe que la courbe de l'exercice 5 reste strictement monotone contrairement à la courbe de l'exercice 6. En effet, pour des dimensions égales à 80 et 90, les temps d'exécution de l'exercice 6 sont respectivement 0.00131s et 0.002502s. On peut remarquer

aussi que la courbe de l'exercice 6 est légèrement au dessus de l'exercice 5 , ce qui veut dire qu'elle est un légèrement plus lente.
Pour une dimension de 100, on observe que le temps d'exécution des exercices 5 et 6 sont dans l'ordre de 0.002s (avec une légère augmentation du cote de l'exercice 5), tandis que le temps de l'exercice 3 est de 0.012s. Ce qui est à peu près **5 fois plus grand.**

2.En fonction du nombre de couleurs :

Afin de comparer le temps des 2 versions en fonction du nombre de couleurs (paramètre `nbcl` donné au programme), nous allons fixer tous les autres paramètres et faire varier le nombre de couleurs.

Voici le test effectué :

- dimension = 50
- nombre de couleurs = variante
- niveau de difficulté = 20
- graine = 8
- exercice = { 5, 6 }
- affichage = 0

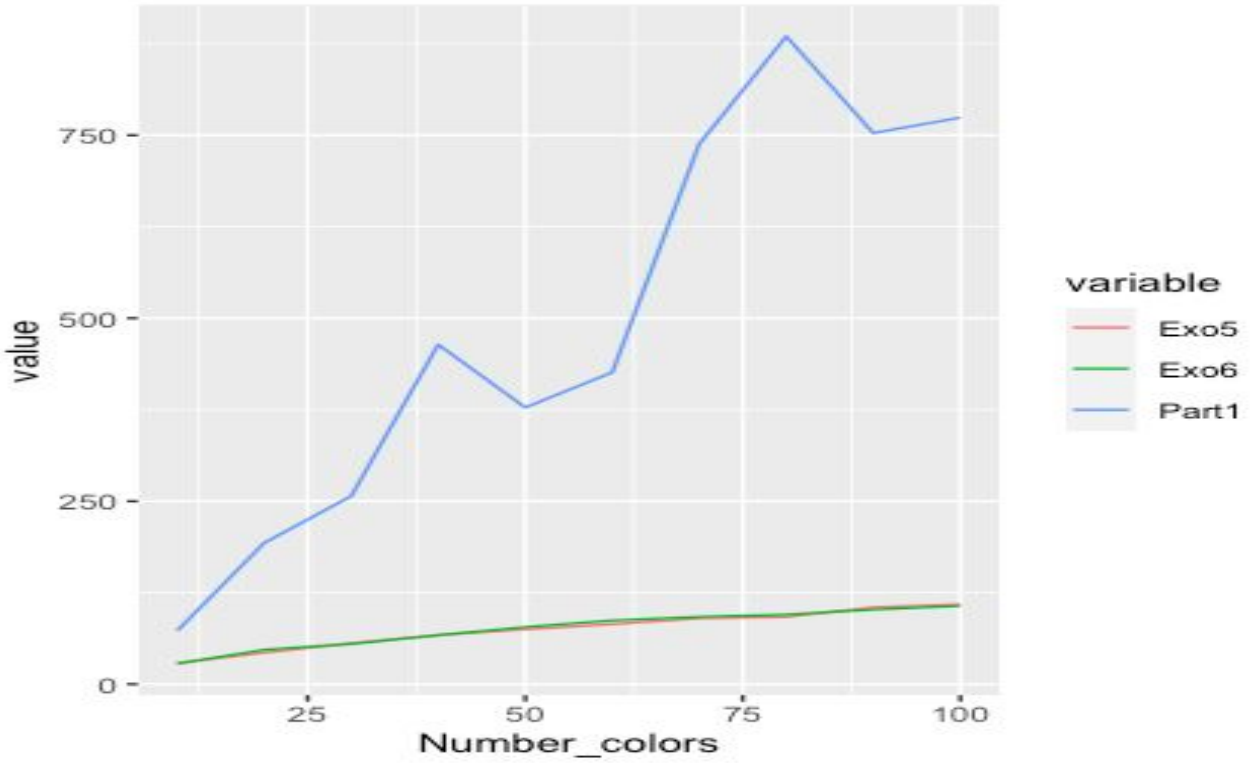
Les résultats obtenus sont représentés dans ce tableau : le temps y est exprimé en secondes.

Tab4 : variation du temps d'exécution du programme en fonction du nombre de couleurs pour chacun des 2 cours

Nombre de Couleurs	Exercice 5		Exercice 6	
	Temps	Iterations	Temps	Iterations
10	0.000592	29	0.000614	28
20	0.000747	43	0.000655	47
30	0.000861	56	0.000807	55
40	0.000643	67	0.000568	67
50	0.001134	75	0.000836	78
60	0.001730	82	0.000698	87
70	0.000939	90	0.000778	92
80	0.000924	92	0.000908	95
90	0.000588	105	0.000683	102
100	0.000694	109	0.000729	107

Comme dans la partie précédente, nous avons représenté ces résultats dans deux graphes : l'un représente le nombre d'itérations entre les exercices 5 et 6 ainsi que l'un des exercices de la partie 1, et l'autre représente la variation du temps d'exécution des exercices 5 et 6 ainsi que l'exercice 3 de la première partie.

Graphe 1 :

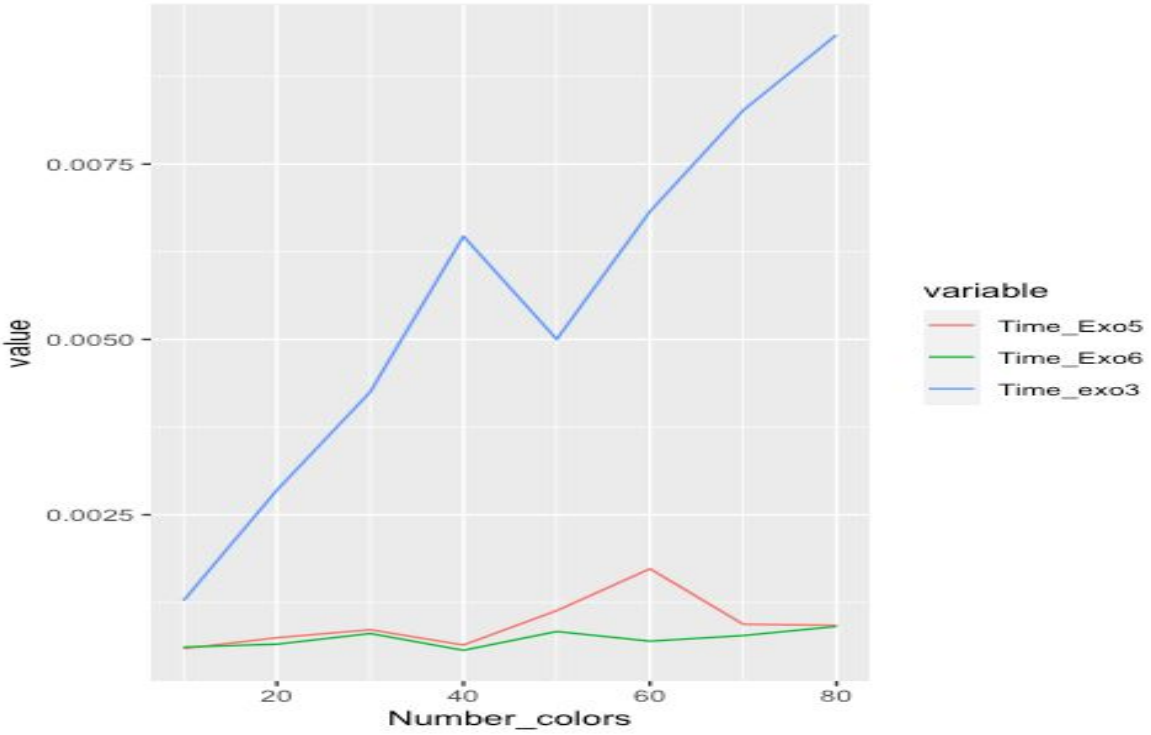


Effect of the number of colors on the number of iterations for both exercices 5,6 and 3.

Le graphe ci-dessus représente l'évolution du nombre d'itérations des exercices 5,6 ainsi que l'exercice 3 en fonction du nombre de couleurs. Dans ce graphe, on peut observer que les exercices 5 et 6 ont une allure identique et qui est croissante.

En ce qui concerne l'exercice 3, on observe aussi une allure croissante. Son graphe augmente plus vite que les exercice 5 et 6. Il atteint des valeurs très grandes : plus de 750 itérations pour un nombre de couleur égale à 100. Néanmoins, pour le même nombre de couleurs, les exercices 5 et 6 ne dépassent pas 125 iterations

Graphe 2 :



Effect of the number of colors on the execution time for both exercices 5,6 and 3.

Le graphe ci-dessus représente l'évolution du temps d'exécution (en secondes) des exercices 5,6 ainsi que l'exercice 3 en fonction du nombre de couleurs. Dans ce graphe, on observe une augmentation du coté de l'exercice 3. Le graphe de l'exercice 6 semble ne pas monter ce qui indique qu'il n'est pas affecté par l'augmentation du nombre de couleurs . En ce qui concerne l'exercice 5, on observe une augmentation dans l'intervalle [50,70] avec des valeurs qui atteignent 0.001730s contre 0.000698s pour l'exercice 6.

Il est clair d'après les deux graphes précédents que les versions de la deuxième partie sont plus rapides et finissent le jeu en moins de temps que celles de la première partie. Il est à noter que la comparaison en terme de vitesse d'exécution s'est faite avec l'exercice 3 de la première partie, qui est bien plus rapide que les deux premiers.

Conclusion deuxieme partie :

D'après les résultats obtenus dans cette deuxième partie, nous arrivons à la conclusion que la deuxième partie est plus optimale et plus rapide que la première. Cela s'explique par le fait que dans la deuxième partie nous avons représenté la grille sous forme de zones qui contiennent des cases qui ont les même propriétés,ce qui facilite le traitement de toutes ces cases. Et finalement , ceci a abouti à un algorithme plus rapide et plus efficace.

Conclusion Finale :

Au cours de ce projet, nous avons conçu plusieurs structures et stratégies modélisant le jeu d'inondation à travers lequel nous avons essayé d'améliorer la rapidité et l'efficacité. Commençant par la première partie dans laquelle nous avons fait un modèle basique aléatoire aboutissant sur des temps d'exécution moyens mais néanmoins qui faisait marcher le jeu. Par la suite, on a conçu une structure bien plus élaborée qui a fait augmente la vitesse considérablement.
A travers ce projet on a appris à concevoir différentes structures satisfaisant un problème donné, on a aussi appris a ameliorer l'efficacite de ces stratégies.