SORBONNE UNIVERSITE FACULTE DES SCIENCES

UFR D'INFORMATIQUE LICENCE 2

UE LU2IN006 : STRUCTURES DE DONNEES

RAPPORT DE PROJET : FLOOD-IT

PARTIE 1 et 2

LOUIZA AOUAOUCHE

DM IM GROUPE 9 - 3802084

ENCADRANT : Pierre FOUILLOUX

30 Mai, 2020

SOMMAIRE

INTRODUCTION :

Vitesse d'exécution, nombre d'itérations et complexité sont trois critères qui reviennent souvent dans la mesure de l'efficacité d'un programme en informatique. Dans ce projet composé de deux parties, nous allons les étudier sur le jeu de l'inondation, plus communément appelé « Flood-It » ou « Mad Virus ». La 1ère partie a pour objectif d'analyser plusieurs implémentations sur différentes structures de données en comparant la vitesse d'exécution. La seconde partie, quant à elle, consiste à étudier les meilleures stratégies pour gagner le plus rapidement possible une partie du jeu en termes de nombres d'itérations. Ainsi, la mise en commun des deux parties permettra de comparer le temps d'exécution et le nombre d'itérations du jeu.

OUTILS DE REALISATION :

Le code accompagnant ce présent document a été réalisé sous :

- Système d'exploitation : Microsoft Windows 10 Professionnel
- Langage de programmation : C
- Logiciel de visualisions des courbes : Plotly
- Ordinateur (temps de calcul) : DELL Inspiron 13 5000 - CORE i7 8th Gen

PARTIE 1 :

I. DESCRIPTION DU CODE

a) DESCRIPTION GLOBALE

Le code est scindé en différentes parties reliées aux structures manipulées. Il est, en effet constitué de :

	Makefile		Génère les exécutables
	Flood-It_Partie1.c		Contient le main qui teste les différentes structures
	Liste_case.c	Liste_case.h	Librairie des fonctions manipulant la structure de liste de cases
	API_Gene_instance.c	API_Gene_instance.h	Librairie de fonctions générant une matrice de jeu
	API_Grille.c	API_Grille.h	Librairie de fonctions manipulant la grille de jeu en SDL
	Fonctions_exo1.c	Entete_Fonctions1.h	Fonctions récursives du jeu manipulant des listes de cases
	Fonctions_exo2.c	Entete_Fonctions2.h	Fonctions impératives du jeu manipulant des listes de cases
	Fonctions_exo3.c	Entete_Fonctions3.h	Fonctions du jeu manipulant la structure S_zsg (cf EXERCICE 3)

Code fourni non modifié Code fourni modifié Code réalisé

Structures utilisées

Grille : Nécessaire aux fonctions d'affichage

Elnt_liste , ListeCase : Liste simplement chainée de case de coordonnées (i,j)

S_Zsg: Stocke la Zsg(Zone supérieure gauche) et la bordure associée dans des listes simplement chaînées

b) DESCRIPTION PAR EXERCICE

EXERCICE 1 :

Consiste à réaliser principalement deux fonctions en manipulant une liste de type Elnt_liste :

***.trouve_zone_rec**:* Construire une zone récursivement à partir des cellules adjacentes (haut, gauche, bas et droite) à une case (i,j) dans la grille M

***.sequence aleatoire rec**:* Utilise la fonction **trouve_zone_rec** avec i=0 et j=0 pour en construire le Zsg

EXERCICE 2 : PILE

Consiste à réaliser deux fonctions en manipulant une liste Elnt_liste en guise de pile :

.trouve_zone_imp : Dérécursifie ***trouve_zone_rec*** et permet ainsi de construire une zone à partir de la case (i,j) en utilisant principalement des comparaisons, des empilements (*ajoute_en_tete*) et des dépilements (*enleve_en_tete*).

***.sequence aleatoire imp**:*Utilise la fonction trouve_zone_imp avec i=0 et j=0 pour en construire le Zsg

EXERCICE 3 : STRUCTURE S_zsg :

Manipule la structure S_Zsg pour résoudre la grille via les fonctions suivantes :
.Init_Zsg; **Ajoute_Zsg** ;**Ajoute_Bordure** ;**Appartient_Zsg**; **Appartient_Bordure** : fonctions de manipulations
.agrandit_Zsg: Mise à jour des champs d'une S_Zsg et retourne le nombre de cases ajoutées
.sequence_aleatoire_rapide:Renvoie le nombre de couleurs necessaires pour gagner avec la version rapide

II. ANALYSE DE LA PERFORMANCE DE LA PARTIE 1 :

Afin de mesurer le temps d'exécution du programme pour chaque exercice, dans Flood-It_Partie1.c : on fixe le nombre de couleurs à 50 et on fait varier la ***dimension de 5 à 100*** et la ***difficulté de 0 à 80***. Ainsi et on récupère dans un fichier le ***nombre*** **d'essais** et le ***temps*** d'exécution pour gagner le jeu par la formule suivante :

temps_cpu=(float)(temps_final - temps_initial)/CLOCKS_PER_SEC;

EXERCICE 1 :

A partir du fichier EXO1.txt on obtient les courbes suivantes :

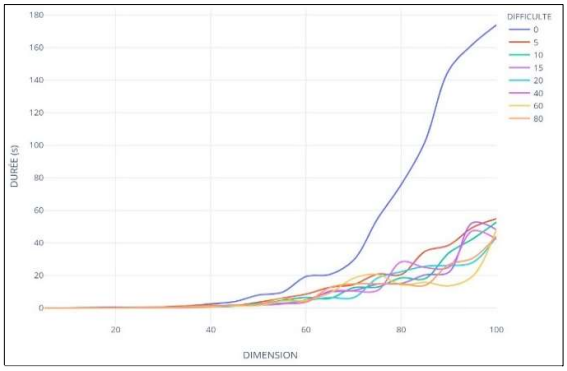

Figure 1 : Durée d'exécution en fonction de la dimension selon différents niveaux de difficulté (exercice 1)

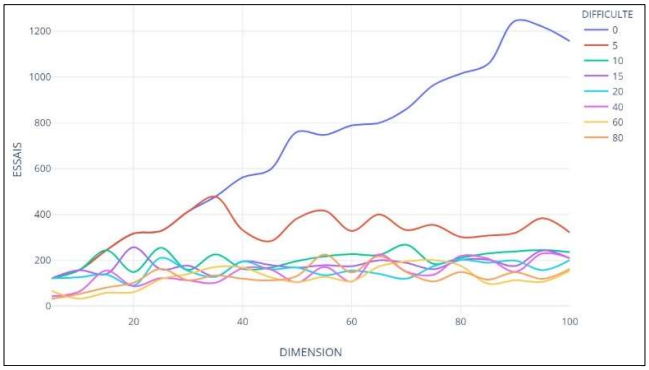

Figure 2 : Nombre d'essais en fonction de la dimension selon différents niveaux de difficulté (exercice 1)

Question 1.3 :

- D'abord, lors de l'exécution sans la fonction **Grille_attente_touche**, on observe graphiquement que plus la dimension est grande plus la grille prend du temps à être résolue.
- Cela confirme au niveau de la Figure 1 où plus la dimension est grande plus la durée d'exécution croit.
- On remarque également que plus le niveau de difficulté est petit (taille moyenne d'une zone petite) plus le nombre d'essais pour résoudre la grille est grand (Figure 2) ce qui se reflète logiquement sur la durée.
- La croissance est assez rapide : c'est plus que linéaire parfois donc la fonction de l'exercice 1 n'est efficace.

EXERCICE 2 :

A partir du fichier EXO2.txt on obtient les courbes suivantes :

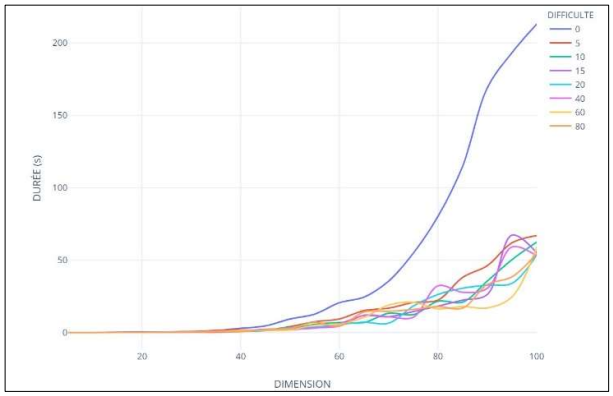

Figure 3 : Durée d'exécution en fonction de la dimension selon différents niveaux de difficulté (exercice 2)

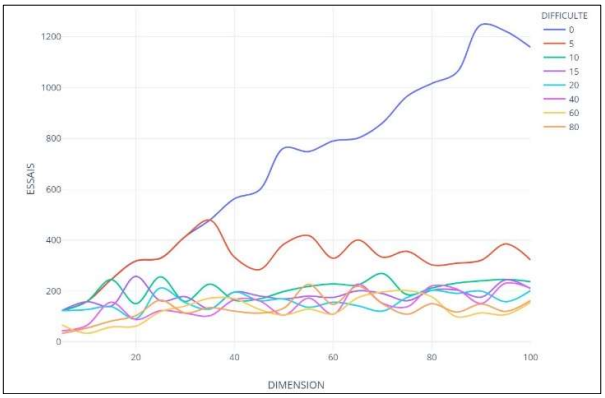

Figure 4: Nombre d'essais en fonction de la dimension selon différents niveaux de difficulté (exercice 2)

Question 2.2 :

- Comme dans la version récursive précédente, ici avec la fonction itérative on observe graphiquement que plus la dimension est grande plus la grille prend du temps à être résolu.
- Les courbes obtenues sont assez similaires ce qui explique pourquoi on a l'impression que c'est carrément le même programme que l'exercice 1 lors de l'observation graphique. En réalité ce n'est pas faux puisqu'il s'agit de la version itérative.
- Dans ce cas également, la durée croit approximativement de façon linéaire (voire plus), ce n'est donc pas efficace lorsqu'il s'agira de manipuler de grandes dimensions.

Que ce soit en itératif ou en récursif, la reconstruction de la Zsg ralentit considérablement le programme c'est ce qui nous pousse à omettre ces deux versions.

EXERCICE 3

A partir du fichier EXO3.txt on obtient les courbes suivantes :

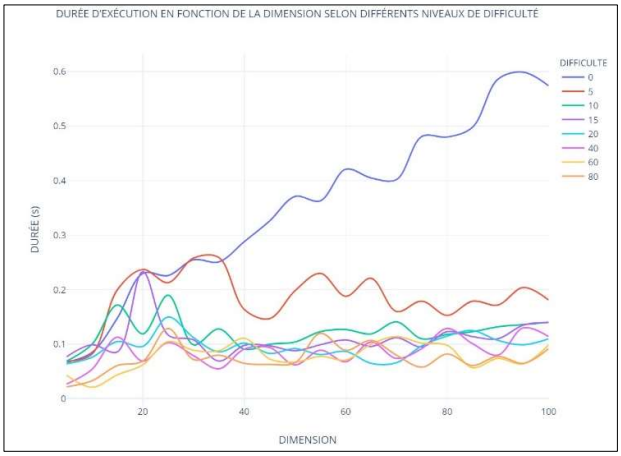

Figure 5

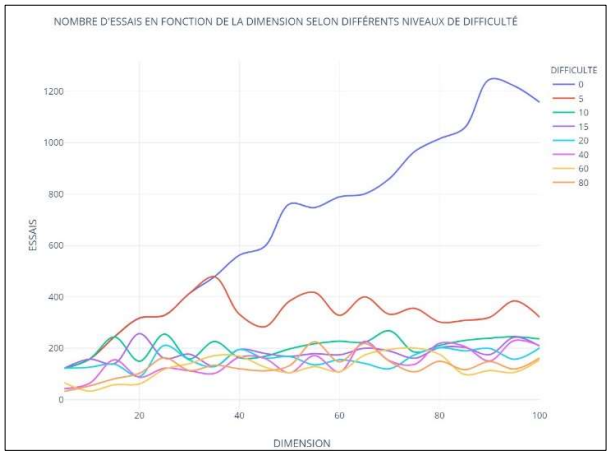

Figure 6

Question 3.4 :

- En utilisant les mêmes données que pour l'exercice 1 et 2 l'observation graphique est impressionnante : c'est très rapide en comparaison aux deux premières versions.
- A travers la Figure 5 on remarque que le temps est inférieur à 1s dans les cas étudiés alors dans les deux premières versions ça atteignait 200s.
- A noter que le nombre d'essais dans la Figure 6 est équivalent à ceux des Figure 4 et Figure 2. Ceci prouve la puissance de ce programme qui malgré un nombre d'essais élevé finit par résoudre la grille très rapidement.

Question 3.5 : COMPARAISON

On fixe le niveau de difficulté à 0 et on étudie la variation de la vitesse d'exécution (durée en fonction de la dimension) selon plusieurs nombres de couleurs (ici 5,10,15,20,40) .

On obtient ainsi différentes courbes selon l'exercice 1 2 ou 3 comme suit :

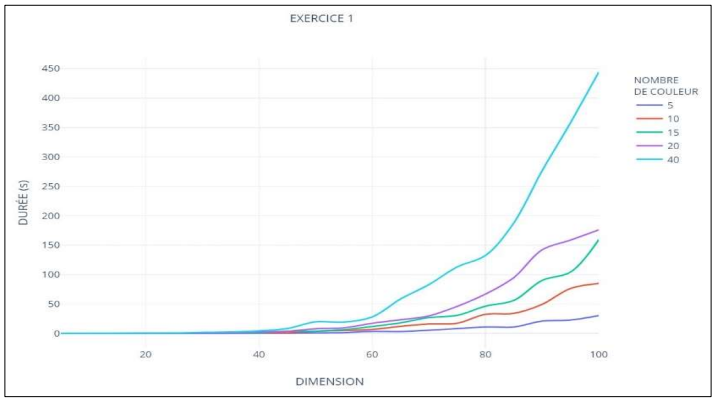

Figure 7 : DUREE EN FONCTION DE LA DIMENSION SELON PLUSIEURS COULEURS (exercice1)

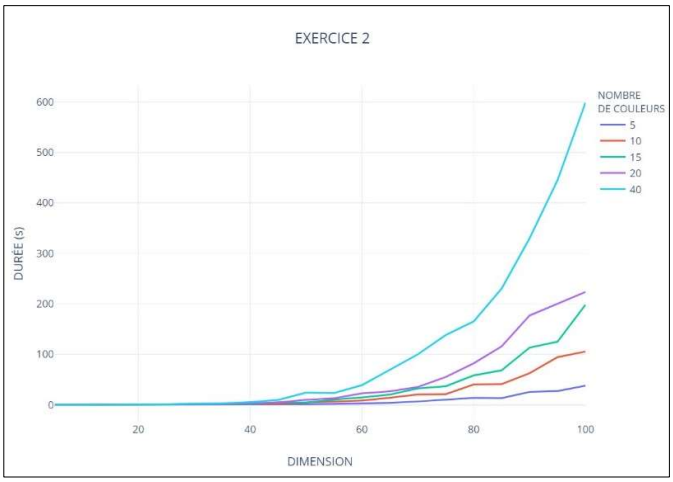

Figure 8 : DUREE EN FONCTION DE LA DIMENSION SELON PLUSIEURS COULEURS (exercice 2)

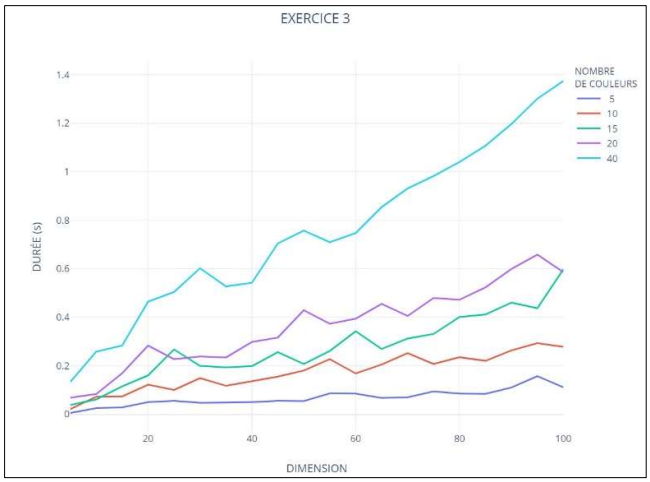

Figure 9 : DUREE EN FONCTION DE LA DIMENSION SELON PLUSIEURS COULEURS (exercice 3)

- De façon générale, on voit que plus le nombre de couleurs est grand plus la durée d'exécution est conséquente quel que soit l'exercice.
- A partir des Figures 7,8 et 9 prenons le nombre de couleurs= 40, dimension=100 on observe les résultats suivants par exercice : 1 : Durée≈ $450s$; 2 : Durée≈ $600s$; 3 : Durée≈ $1.4s$
  Que ce soit pour cet exemple ou pour d'autres points, on remarque que la version récursive est légèrement plus rapide que la version itérative ce qui peut s'expliquer par le fait que **trouve_zone_rec** ne fait appel qu'à elle-même tandis que **trouve_zone_imp** manipule plus de fonctions à partir de **Liste_case**.
  En revanche, en comparant avec le résultat de la version rapide la différence est flagrante, le temps gagné correspond au fait de ne pas recalculer la zone supérieure gauche à chaque fois.
- Dans les exos 1 et 2 la durée dépend excessivement du nombre de couleurs tandis que l'exercice 3, c'est dépendant mais cela reste négligeable.

CONCLUSION : Ceci nous amène donc à conclure que la structure de données la plus appropriée et la plus rapide en fonctions des différents arguments est S_Zsg pour cette première partie. La principale raison étant que les nouvelles cases à ajouter à la Zsg sont tout simplement dans la bordure au lieu de reconstruire cette zone à chaque fois.

PARTIE 2 :

I. DESCRIPTION DU CODE

c) DESCIPTION GLOBALE

En plus de ce qui a été vu dans la première partie voici comment est organisé le code pour cette seconde partie :

●	Fonctions_exo4.c	Entete_Fonctions4.h	Fonctions manipulant la structure Graphe_zone
●	Fonctions_exo5.c	Entete_Fonctions5.h	Fonctions manipulant Graphe_zone et Bordure_graphe
●	Fonctions_exo6.c	Entete_Fonctions6.h	

● Code réalisé

Structures utilisées *(en plus de celles citées dans la partie 1)*
Sommet : Stocke les informations relatives à un sommet du graphe
Cellule_som : Structure de listes chainées de sommets
Graphe_zone : graphe dont les sommets correspondent aux zones de la grille
Bordure_graphe : permet de stocker les sommets bordant la Zsg

EXERCICE 4 :

Consiste à mettre en place une nouvelle bibliothèque reliée à la structure de **Graphe_zone** dont :
.affichage_graphe ;ajoute_liste_sommet; detruit_liste_sommet ;ajoute_voisin ; adjacent
.cree_graphe_zone : permet de créer la structure Graphe_zone

EXERCICE 5:

Rassemble l'exercice 4 et l'idée de l'exercice 2 où on manipule Bordure_graphe et Graphe_zone par les fonctions :
.cree_graphe_zone2: ajoute l'initialisation du champ *marque* à la version cree_graphe_zone de l'exo 4
.cree_graphe_bordure ; maj_bordure_graphe : pour créer et mettre à jour Bordure_graphe
.detruit_bordure_graphe ; detruit_graphe_zone : pour libérer l'espace allouée par Graphe_zone et Bordure_graphe
.strategie_max_bordure : renvoie la taille de la séquence reliée à la stratégie max bordure

EXERCICE 6 :

Fait appel à la stratégie de max bordure après une première séquence de jeu réalisée par un parcours en largeur qui permet de définir le plus court chemin entre la Zsg et la Zid (zone inférieure droite) on y utilise :
creer_graphe_zone3 : ajoute l'initialisation du champ *distant et pere* à la version cree_graphe_zone2 de l'exo 5
plus_court_chemin : permet de mettre à jour le champ père et distant par le plus cours chemin le reliant à la racine en paramètre
strategie_parcours_largeur : renvoie la taille de la séquence reliant parcours en largeur et la stratégie max bordure

I. ANALYSE DE LA PERFORMANCE DE LA PARTIE 1 :

Par la même formule (calcul du temps d'exécution) que la Partie 1, nous allons étudier la durée que prennent les stratégies de la Partie 2 à résoudre la grille ainsi que la taille de la séquence correspondante.

EXERCICE 4 :

Question 4.3 :

L'exercice 4 est principalement basé sur la construction d'un graphe zone. Nous observons que cette construction n'est pas très couteuse en termes de temps d'exécution

étant donné que même à partir d'une grille de dimension

200 cela reste en dessous du millième de seconde

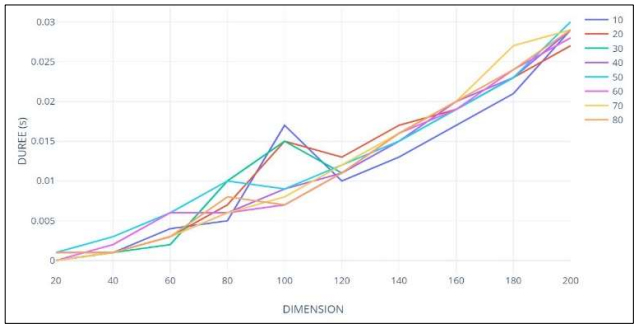

Figure 10 : Durée en fonction de la dimension selon plusieurs niveaux de difficultés et un nombre de couleurs à 50

EXERCICE 5 :

Etudions les courbes obtenues par l'exercice 5

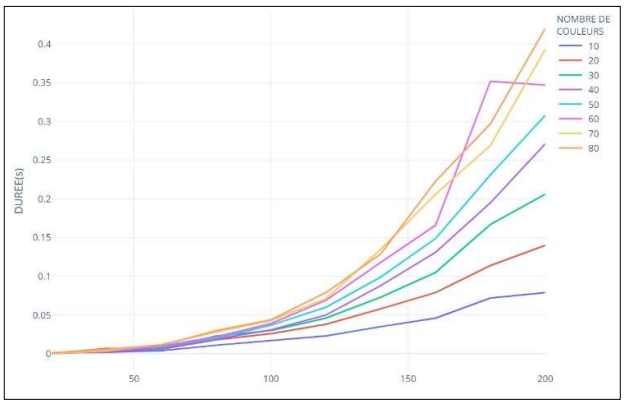

Figure 11 : Durée en fonction de la dimension selon différents nombre de couleurs avec nivdiff=0

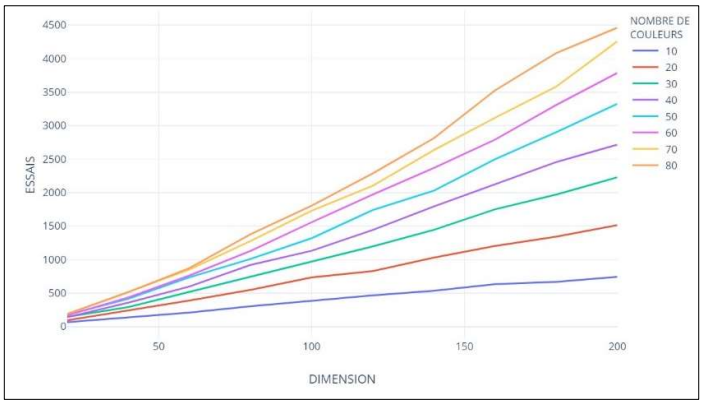

Figure 12 : nombre d'essais en fonction de la dimension selon différents nombre de couleurs avec nivdiff=0

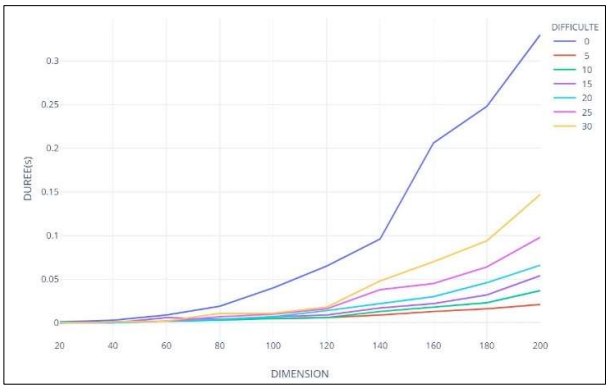

Figure 13 : Durée en fonction de la dimension selon différents niveaux de difficultés avec nbcl=50

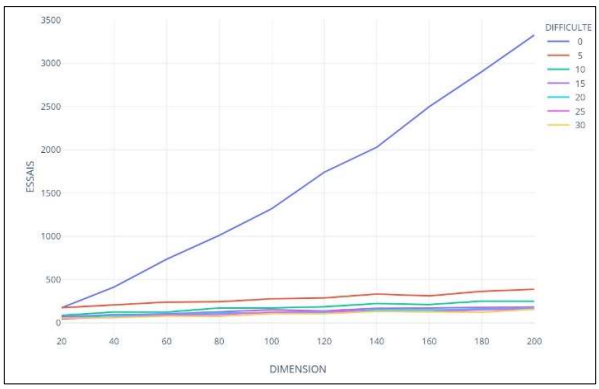

Figure 14 : Essais en fonction de la dimension selon différents nombre de couleurs avec nivdiff=0

L'efficacité de l'exercice 5 réside dans la possibilité d'ajouter plusieurs sommets à la fois à la Zsg à partir de la bordure.
On observe à travers les figures 11 et 13 que la durée d'exécution est assez faible en fonction de la dimension, du nombre de couleurs et de la difficulté croissante. De l'autre côté, les Figures 14 et 16 montrent le nombre d'essais est assez grand en comparant par exemple à ceux de la partie 1.
Ceci dit, l'exercice 5 est assez rapide et efficace en termes de vitesse d'exécution mais l'est beaucoup moins en terme de nombre d'itérations.

EXERCICE 6 :

Question 6.1 : Montrons qu'une séquence de couleurs de petite taille permettant d'incorporer la zone inférieure droite (Zid) à la zone Zsg peut se déduire d'un plus court chemin en nombre d'arêtes dans le graphe-zone.

- D'abord, un graphe zone G ainsi défini est dit connexe en théorie des graphes: quels que soient les sommet s1 et s2 de G il existe un chemin reliant s1 à s2. Par conséquent, dans un graphe théorique, une séquence de couleurs permettant d'incorporer la Zid à Zsg n'est autre qu'un chemin composé d'arêtes reliant les deux sommets de ces deux zones. Notons C ce chemin.
- Lorsque C a une taille minimale on dit alors qu'il s'agit d'un chemin élémentaire : la plus petite suite d'arêtes dans un graphe reliant les deux sommets s1 et s2. En théorie, ceci peut être obtenu par un parcours en largeur. Dans notre cas, en calculant la distance (taille de la séquence) de tous les sommets par rapport à Zsg, on obtient ainsi la plus courte séquence reliant Zsg à n'importe quel sommet dont Zid.
- Nous avons donc montré qu'une séquence de couleurs de taille minimale liant Zsg à Zid peut être obtenue par un plus court chemin en nombre d'arêtes dans le graphe-zone.

Question 6.3 :

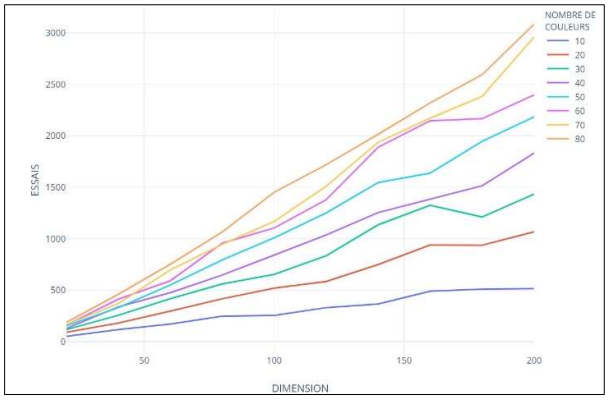

Figure 15 : Nombre d'essais en fonction de la dimension selon différents nombres de couleurs avec nivdiff=0

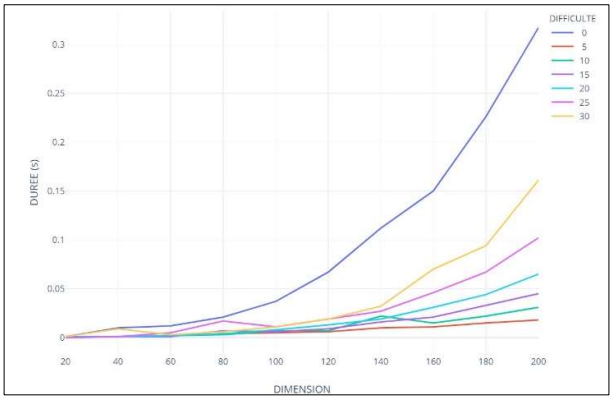

Figure 16 : Durée en fonction de la dimension selon plusieurs niveaux de difficultés avec nbcl=50

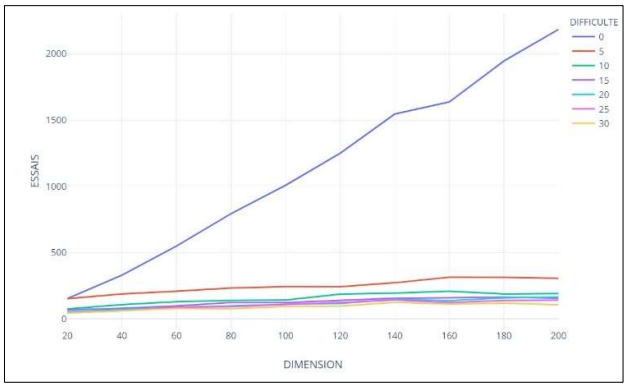

Figure 17 : Nombre d'essais en fonction de la dimension selon des niveaux de difficultés différents avec nbcl=50

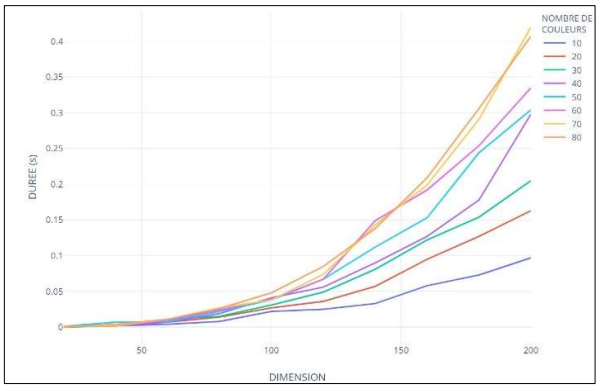

Figure 18 Durée en fonction de la dimension selon des nombres de couleurs différents avec nivdiff=0

A travers les figures 17 et 15 on observe un nombre d'essais assez énorme en comparant aux données de la partie 1. C'est également plus grand par rapport à l'exercice 5. La durée d'exécution, est quasi nulle, elle encore meilleure que celle de l'exercice 5.
On déduit alors que le parcours en largeur utilisé ici permet de gagner du temps en divisant la grille en 2, mais l'appel de la stratégie max bordure ajoute encore un nombre d'essais conséquents ce qui nous amène à conclure que l'exercice 6 est efficace en termes de vitesse d'exécution mais l'est beaucoup moins en nombre d'itération.

CONCLUSION DE LA COMPARAISON DES STRATEGIES :

A travers les différentes analyses précédentes entre les exercices de la Partie 1 et ceux de la Partie 2 on conclue ce qui suit : les exercices 1, 2, 3 : sont plus efficaces en termes de nombre d'itérations que les exercices 5 et 6. Inversement, les exercices 5 et 6 prennent moins de temps donc sont plus efficaces en termes de vitesse d'exécution.

CONCLUSION GENERALE :

Le choix de la stratégie pour gagner le jeu de l'inondation ou quelconque autre jeu de la même sorte dépend de l'utilisateur s'il recherche à être efficace en nombre de coups il optera ainsi pour les stratégies de la partie 1 et si en revanche il recherche à maximiser la vitesse d'exécution il optera pour les stratégies de la partie 2.