

RENDU PROJET

ABITBOL Yossef, 3804139, DM INFO/EEA ,

SERRAF Dan , , INFO DANT,

PROJET FLOOD-IT,

Groupe 10.

INTRODUCTION:

Ce projet consiste à étudier des stratégies pour gagner au jeu d'inondation de grilles, appelé Flood-It . Le fonctionnement du jeu est simple, il y a $N \times N$ cases dans un carré , 2 cases adjacentes correspondent à 2 cases de même couleur si elles ont un côté horizontal ou vertical en commun. On appelle zone un ensemble connexe de cases et ZSG (zone supérieure gauche) la zone contenant la case située en haut à gauche de la grille. Le but est de colorier d'une nouvelle couleur la ZSG à chaque tour de jeu afin d'ajouter les cases adjacentes à la ZSG. Pour finir le jeu, on doit colorier totalement la grille avec une seule couleur. Ce projet se divise en 2 parties.

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 en termes de nombres d'itérations.




Diagram illustrating the Flood-It game interface and its parameters:

- ZSG**: Zone Supérieure Gauche (Top-Left Zone).
- Dimension ; nombres de cases**: Dimension of the grid and number of cases.
- Affichage**: 0 = sans la grille, 1 = avec la grille.
- Exo**: permet de choisir quelle fonction utiliser.
- Nb couleur**: Number of colors.
- Affichage**: Display format.
- dimension**: Grid dimension.
- Difficulté**: Difficulty level.
- Graine aléatoire**: Random seed.
- exo**: External function choice.

Example command line output:

```
./Flood-It_ 18 9 7 3 0 1
```

Annotation: Ces 3 cases sont adjacentes (These 3 cases are adjacent).

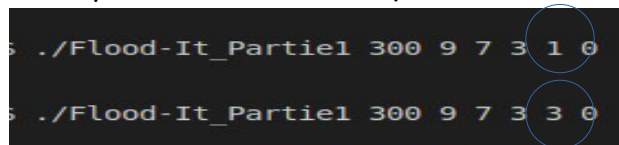
PARTIE 1

Comme expliqué dans l'introduction, l'objectif de la partie 1 est d'implémenter une solution pour ce jeu qui repose sur un tirage aléatoire: on tire au sort la couleur à jouer. Cette partie a donc pour but de finir le jeu rapidement et non de trouver le plus petit nombre d'itérations possibles. Comme les 3 versions utilisent le même principe, on aura le même nombre d'essais.

Cette partie est-elle même constituée de 3 exercices dont 1 bonus.

EXERCICE 1:

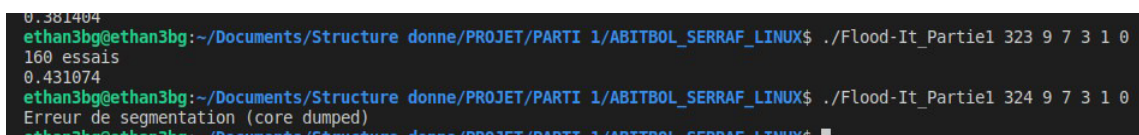
La première version est d'utiliser la récursivité, or on constate qu'arriver à une certaine dimension, la récursivité n'est plus valide, «erreur de segmentation». En effet, pour tester la récursivité, 2 lignes de commande sont possibles, soit en mettant 1 ou 3 pour l'exo. 1 pour essayer avec `trouve_zone_rec` et `séquence_aléatoire_rec` ou 3 avec `trouve_zone_imp` (exercice 2) et `séquence_aléatoire_rec`:



```
./Flood-It_Partiel 300 9 7 3 1 0
./Flood-It_Partiel 300 9 7 3 3 0
```

trouve_zone_rec: Fonction qui permet d'affecter dans la liste chaînée 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.

séquence_aléatoire_rec: Fonction 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.



```
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It_Partiel 323 9 7 3 1 0
0.381404
160 essais
0.431074
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It_Partiel 324 9 7 3 1 0
Erreur de segmentation (core dumped)
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$
```

Comme la récursivité est limitée, on va créer une fonction dans l'exercice 2 (bonus) en utilisant la dé-récursivité.

EXERCICE 2:(bonus)

C'est en manipulant une pile de cases qu'on va créer 2 nouvelles fonctions, `trouve_zone_imp` et `séquence_aléatoire_imp` qui ont le même but que les fonctions de l'exercice 1 mais de manière itérative.

Pour tester cela, on peut soit mettre 2 ou 4 dans l'exo qui va permettre avec 2 de tester `trouve_zone_rec` et `séquence_aléatoire_imp` et avec 4 `trouve_zone_imp` et `séquence_aléatoire_imp`.

On remarque que pour l'exercice 2 les fonctions ne sont pas limitées comme avec la récursivité mais elles sont moins rapides pour une dimension allant de 0 à la limite de la récursivité.

```
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It_Partie1 300 9 7 3 1 0
118 essais
0.295961
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It_Partie1 300 9 7 3 2 0
118 essais
1.726712
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It_Partie1 400 9 7 3 1 0
Erreur de segmentation (core dumped)
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It_Partie1 400 9 7 3 2 0
149 essais
7.081745
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It_Partie1 400 9 7 3 2 0
```

Comme on peut le voir, la différence de temps est énorme, même si la fonction itérative n'est pas limitée, elle est très lente, c'est pour cela que dans l'exercice 3 on va créer des fonctions à l'aide de structure acyclique qui vont nous permettre d'obtenir des résultats beaucoup plus rapides et non limités.

EXERCICE 3:

Les versions précédentes du jeu sont très lentes car elles réénumèrent 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. Afin d'avoir une meilleure efficacité que l'exercice 1 et 2 qui reposent uniquement sur une liste chaîne des cases de la zsg, on utilise ici une structure zsg qui en plus de la liste sera composé d'un tableau de liste chaîne qui correspond à la bordure de la zone et enfin une matrice de dimension 2 qui nous permettra de savoir rapidement à quelle structure une case appartient (zsg, bordure, non visité).

Pour tester cette fonction, il faut mettre `exo==5`.

Pour conclure, après avoir comparé les 3 exercices sur leur temps d'exécution en

fonction du nombre de cases et du nombre de couleurs , on a obtenu les 2 tableaux suivants :

TEST	Dim=10 et couleur =5	Dim=250 et couleur =5	Dim=323 et couleur =5	Dim=400 et couleur =5	Dim=3500 et couleur = 5
/* Recursif - Recursif */	0.000106 (2)	0.116266 (3)	0.105363 (2)	Erreur de segmentation	Erreur de segmentation
/* Recursif - Iteratif */	0.000145 (3)	0.394386 (4)	0.857698 (5)	2.582764(3)	670.447571 (2)
/* Iteratif - Recursif */	0.000384 (4)	0.105363 (2)	0.145150 (3)	Erreur de segmentation	Erreur de segmentation
/* Iteratif - Iteratif */	0.000555 (5)	0.397346 (5)	0.818675 (4)	2.629864 (2)	679.394043 (3)
/* Rapide */	0.000052 (1)	0.029459 (1)	0.058616(1)	0.067789 (1)	19.195766 (1)
Nombre essais	28	56	46	56	98

TEST	Dim=100 et couleur =5	Dim=100 et couleur =15	Dim=100 et couleur =25	Dim=100 et couleur =50	Dim=100 et couleur =100
/* Recursif - Recursif */	0.028967	0.061855	0.107862	0.186840	0.398028
/* Recursif - Iteratif */	0.059680	0.160178	0.293648	0.569102	1.232191
/* Iteratif - Recursif */	0.030896	0.070173	0.105066	0.188251	0.420001
/* Iteratif - Iteratif */	0.050744	0.158774	0.296681	0.552100	1.236966
/* Rapide */	0.015394	0.026480	0.032935	0.048867	0.069441
Nombre essais	36	196	334	606	1493

Après avoir fait une multitude de tests, nous avons observé que lorsque nous gardons **le même nombre de couleurs** à 5 et nous augmentons progressivement le nombre de cases(dimension: 10 250 323 400 3500), le nombre d'essais ne varie pas fortement contrairement au temps d'exécution. De plus, la récursivité est ici limitée à une certaine dimension.

Ensuite, nous avons gardé la **même dimension**, ici à 100, et nous avons augmenté le

nombre de couleurs progressivement {5,15,25,50,100}. Nous avons pu remarquer qu'en premier lieu le nombre d'essais modifiait fortement contrairement au temps d'exécution, qui varie peu. Aussi si nous augmentons fortement le nombre de couleurs, la limite de la récursivité diminue.

D'après ces analyses on a montré que la couleur a un impact direct sur le nombre d'essais et la dimension un impact direct sur le temps d'exécution. Pour finir, peu importe le nombre de cases ou de couleur, la fonction de l'exercice 3 est la plus performante. ensuite, en matière de temps d'exécution s'ensuit la fonction récursive sauf qu'elle est limitée, et enfin la fonction de l'exercice bonus, c'est la plus lente mais elle va plus loin que la récursive.

PARTIE 2

Comme expliqué dans L'introduction, la partie 1 consistait à gagner au jeu le plus rapidement possible, or la partie 2 a pour but de gagner au jeu en faisant le moins d'itérations possible. Pour cela, la partie 2 étant composée de 3 exercices, nous allons commencer par aborder le cas des graphes des zones.

Exercise 4:

Dans ce premier exercice de la partie 2, on va implémenter notre programme avec de nouvelles structures, plus particulièrement la structure graphe zone qui permet de regrouper le nombre de sommets dans le graphe, la liste chaîne des sommets de ce dernier et un tableau de pointeur sur sommet qui nous permettra de savoir rapidement à quelle zone appartient chaque sommet, ainsi que de créer la liste des sommets adjacents.

Pour cela, on a créé une fonction : `créer_graphe_zone` qui va créer le graphe entier. Pour le visualiser, on utilise la fonction `affiche_graphe` qui nous permet d'observer en mode texte un exemple d'affichage du fonctionnement du graphe des zone. Toutes ses fonctions sont présentes dans le fichier `Graphe.h/ Graphe.c` Pour tester cet affichage , on met 6 dans l'exo.

Exemple d'affichage :

```
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It 4 3 5 6 6 0
0 0 2 2
2 2 2 0
2 1 0 0
2 2 2 2
```

Matrice avant la création du graphe

Nombre de sommet dans le graphe : 4 Nombre de sommets dans le graphe

Numero : 3 1 couleur 1 nb case liste :

Case (2,1)

numero 2	couleur 0	nbcase 3
numero 1	couleur 2	nbcase 10

Sommets adjacents

Numero : 2 0 couleur 3 nb case liste :

Case (2,2) Case (2,3) Case (1,3)

numero 3	couleur 1	nbcase 1
numero 1	couleur 2	nbcase 10

Numero : 1 2 couleur 10 nb case liste :

Case (3,3) Case (3,2) Case (3,1) Case (3,0) Case (2,0) Case (1,0) Case (1,1) Case (1,2) Case (0,3) Case (0,2)

numero 3	couleur 1	nbcase 1
numero 2	couleur 0	nbcase 3
numero 0	couleur 0	nbcase 2

Numero : 0 0 couleur 2 nb case liste :

Case (0,1) Case (0,0)

numero 1	couleur 2	nbcase 10
----------	-----------	-----------

Affiche matrice pointeur sommet

0,0	0,0	2,1	2,1
2,1	2,1	2,1	0,2
2,1	1,3	0,2	0,2
2,1	2,1	2,1	2,1

Matrice des pointeurs du sommet du graphe

i,j

Couleur N° de sommet

Exercise 5:

Dans cet exercice, on va utiliser la fonction permettant de créer le graphe vu dans l'exercice précédent afin de créer une nouvelle structure `s_zsg2` présent dans `MaxBordure.h` qui est une reformulation de la structure de l'exercice 3 remplaçant un tableau de liste de cases par un tableau de liste de sommet afin de pouvoir choisir le sommet adjacent contenant le plus de cases. On teste en mettant 7 dans L'exercice.

```
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It 1503 12 5 6 7 0
168 essais
0.404231
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It 1503 12 5 6 5 0
334 essais
11.140508
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$
```

On remarque que la fonction est beaucoup plus efficace que celle dans l'exercice 3. En effet cela s'explique par le fait que d'une part ont créé initialement à l'aide du graphe les zones de couleurs qui sont représenté par les sommets ainsi en faisant basculer une couleur dans la zsg on parcourra une liste qui sera nécessairement plus courte car chaque sommet représente un ensemble de cases. Mais cette rapidité s'explique surtout par le fait qu'à chaque itération on sélectionne la couleur qui contient le nombre de case le plus grand dans la bordure ainsi en la faisant basculer on réalise forcément une meilleure performance qu'une couleur choisit aléatoirement dans la partie 1.

Exercice 6:

Enfin , la dernière stratégie vue dans ce projet est le parcours en largeur qui sera implémenté dans `ParcoursLargeur.h` et `ParcoursLargeur.c`.

Dans cet exercice, on va chercher le chemin le plus court entre la zsg et la zid (zone inférieure droite) afin de réaliser une séparation de notre grille en deux et ainsi résoudre la grille plus rapidement à l'aide `maxBordure` . Pour ce faire, on a dû créer une nouvelle bibliothèque `FileParcours.c` et `FileParcours.h` qui regroupe les fonctions du principe de file , enfiler défiler ... Pour tester cette fonction, il suffit d'entrer dans la ligne de commande 8 pour l'exercice.

```
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It 2000 30 50 100 7 0
104 essais
0.845384
ethan3bg@ethan3bg:~/Documents/Structure donne/PROJET/PARTI 1/ABITBOL_SERRAF_LINUX$ ./Flood-It 2000 30 50 100 8 0
92 essais
0.779370
```

Comme on peut observer sur cette image, la fonction `Parcours en l` Comme on peut observer sur cette image, la fonction `ParcoursLargeur` est plus efficace en termes d'essais et de temps par rapport à `maxbordure`. Notons toutefois qu'en fonction des paramètres donnés on pourra remarquer une rapidité particulière de la fonction `parcours en largeur` par rapport a `moxBordure`. Cela dépendra de la difficulté et du nombre de couleurs si la grille est plus ou moins facile le fait de séparer la grille en deux ne sera pas particulièrement plus rapide et meilleure que

maxBordure.

Pour conclure cette partie 2, on a comparé les 2 fonctions créées, MaxBordure et ResoudreParcoursLargeur selon 3 cas différents.

1^{er} cas) On a gardé le même nombre de couleurs à 5 et de difficulté à 10 et on a fait varier la dimension {500, 1500, 5000}, on a obtenu les résultats suivant

Parcours - Largeurs

Dimension	500	1500	5000
Essais	37	41	650
Temps	0.055339	0.449191	5.584368

Max - Bordure

Dimension	500	1500	5000
Essais	32	39	728
Temps	0.050844	0.432115	5.713031

On montre bien que pour des petites dimensions, Maxbordure est plus efficace en terme or, arrivé à une certaine dimension, Parcours en largeur est meilleure. Même chose pour le temps d'exécution.

2^e cas) On a gardé le même nombre de dimensions à 500 et de difficulté à 10 et on a fait varier le nombre de couleurs {50, 200, 800}, on a obtenu les résultats suivants

Parcours - Largeurs

Couleur	50	200	800
Essais	225	481	1272
Temps	0.053352	0.056087	0.056436

Max - Bordure

Couleur	50	200	800
Essais	242	513	1273
Temps	0.056152	0.054893	0.055741

Ici, on montre que pour des petits nombre de couleurs, le parcours en largeur est plus efficace en nombre d'itérations, mais arrivé a un certain nombre de couleurs, les 2 fonctions ont a peu près le même nombre d'essai. Le temps d'exécution est quasiment le même dans ce cas.

3^e cas) On a gardé le même nombre de dimensions à 300 et de couleurs à 100 et on a fait varier la difficulté {250, 1000, 10000}, on a obtenu les résultats suivants

Parcours - Largeurs

Difficulté	250	1000	10000
Essais	55	192	294
Temps	0.017451	0.013666	0.014348

Max - Bordure

Difficulté	250	1000	10000
Essais	58	240	351
Temps	0.018421	0.012383	0.013043

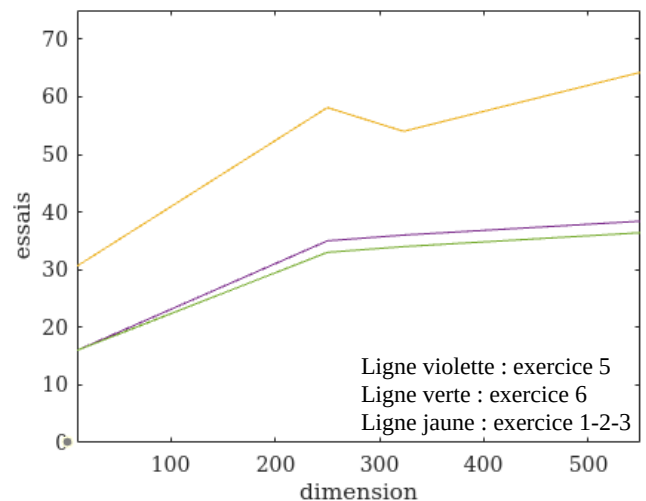
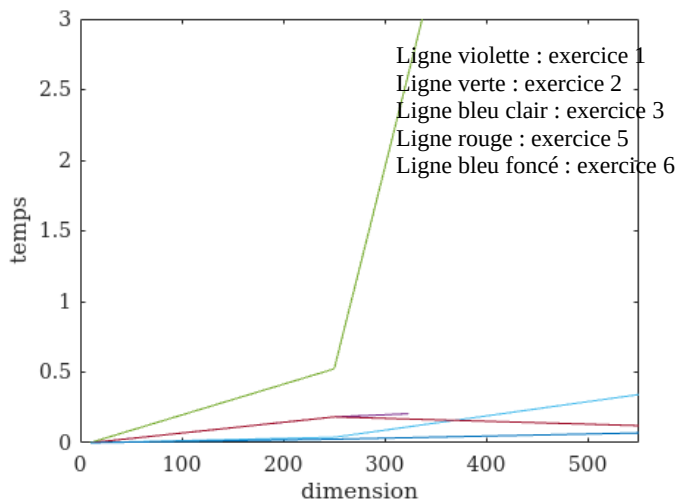
Dans ce dernier cas, Parcours en largeur est meilleure en terme d'itération mais moins bonne en temps d'exécution

BILAN FINAL:

Pour finir, on a comparé l'ensemble des 5 fonctions rencontrés dans notre projet, 3 dans la partie 1, fonction récursive, fonction dérécursié, et la fonction rapide utilisant la bordure, et 2 dans la partie 2, MaxBordure et parcours en largeur. On est arrivé aux conclusions suivantes:

1) Premier cas: On a gardé le même nombre de couleur(=5) et de difficulté(=7) et on a modifié la dimension {10,250,700} sur 10 graines différentes en calculant la moyenne à chaque fois, pour le nombre d'essais et pour le temps d'exécution.

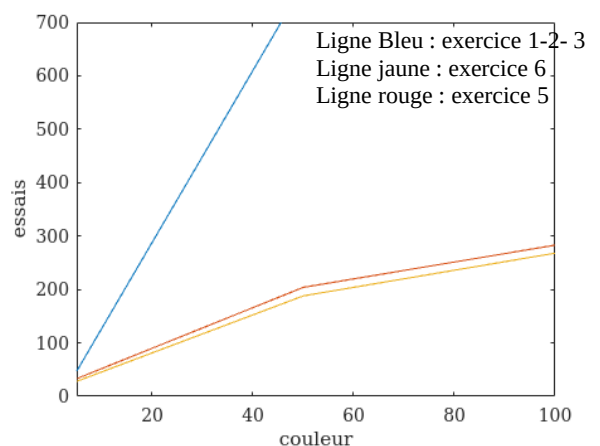
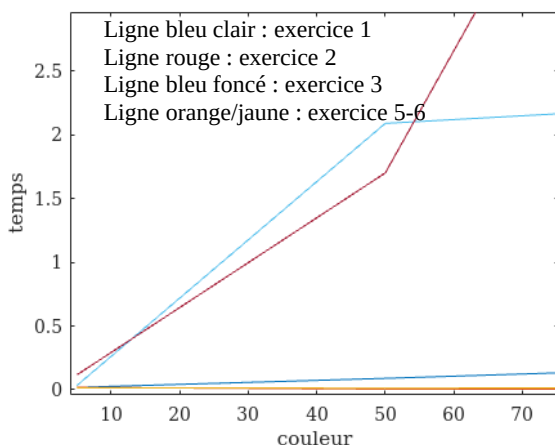
On obtiens: a gauche le temps et a droite le nombres d'essais.



On a montré que lorsqu'on change la dimension, en termes de temps, la plus performante est la fonction parcours en largeur suivie du MaxBordure après une certaine dimension et la moins performante est celle de l'exercice 2. De plus, comme nous le savons déjà, on peut remarquer la fonction récursive est limitée et ne peut pas être testée sur des grilles de mêmes paramètres que les autres. En termes d'essais, le parcours en largeur devance de peu maxbordure mais de beaucoup les fonctions des exercices 1, 2 et 3.

2) Deuxième cas: On a gardé le même nombre de dimension(=150) et de difficulté(=10) et on a modifié le nombre de couleur {5,50,150} sur 10 graines différentes en calculant la moyenne à chaque fois, pour le nombre d'essais et pour le temps d'exécution.

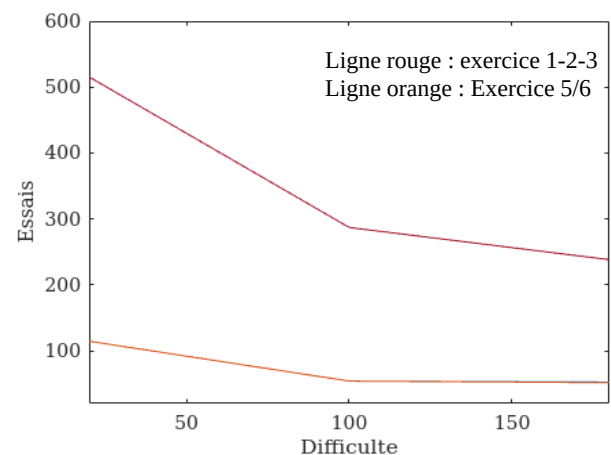
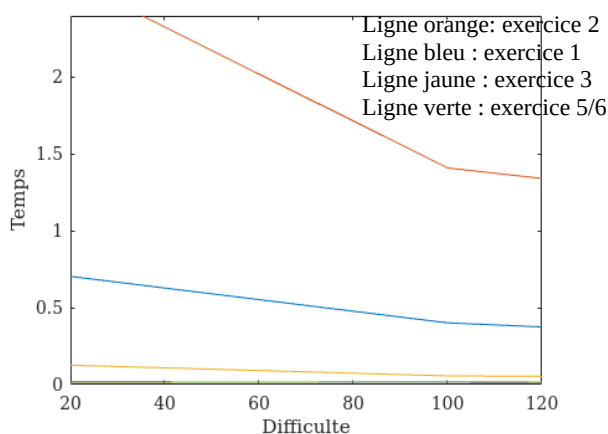
On obtiens: a gauche le temps et a droite le nombres d'essais.



Dans ce cas, le temps d'exécution est largement inférieur pour les fonctions des exercices 3,5 et 6, devancé par maxBordure et parcours en largeur, par rapport aux fonctions des 2 premiers exercices. D'autant plus que les nombres d'essais pour les 2 derniers exercices sont très inférieurs aux 3 premiers exercices.

3) Troisième cas: On a gardé le même nombre de dimension(=200) et de couleur(=30) et on a modifié le nombre de difficulté {10,100,200} sur 10 graines différentes en calculant la moyenne à chaque fois, pour le nombre d'essais et pour le temps d'exécution.

On obtiens: a gauche le temps et a droite le nombres d'essais.



Enfin, pour le dernier cas, la fonction récursive, la fonction dérécursifié et la fonction rapide utilisant la bordure ont un temps d'exécution et un nombre d'essais supérieurs aux fonctions max-Bordure et parcours largeur. Mais on remarque qu'avec une dimension d'une grille moyenne et le paramètre de difficulté qui se modifie on ne voit pas particulièrement la différence et surtout l'utilité d'utiliser parcours largeur par rapport a max bordure. Pour ce voir il faut utiliser des grilles grandes, avec un nombre de couleur et difficulté plus ou moins importante et on verra alors une réelle différence et une meilleure performance de parcours largeur par rapport à max bordure.

POUR CONCLURE, après avoir comparé les 5 fonctions en fonction des 3 variables (dimension, couleur, difficulté) on a montré que les fonctions de la partie 2 montrant la stratégie du parcours en profondeur et du maxbordure sont largement plus performantes que les fonctions de la partie 1, en termes de temps d'exécution et en terme du nombre d'itérations.