MATTON Nicolas 3502362
SABRIE Thomas 3300172

FLOOD IT - RAPPORT

PARTIE I

Questions :
1.4

Nous pouvons constater une augmentation considérable du temps d’exécution qui varient en fonction de la taille et de la difficulté, ce qui semble logique.
En effet la difficulté diminuant la taille des zones de couleurs différentes, plus de tests sont nécessaire pour trouver la bonne couleur.
Quand a l'augmentation de la taille,il en vas de même.
Les listes de cases deviennent de plus en plus grandes.

1.6
Nous pouvons voir que la fonction impérative est bien plus rapide que la fonction récursive

2.4
Nous constatons que la fonction acyclique est bien plus rapide que les deux précédentes.
Cela fait suite aux test que nous effectuions dans les précédentes version à savoir regarder si une case appartenait a une zone, ce qui prend beaucoup de temps à un certain niveau(complexité en O(taille de la liste)) ;
Cependant dans celle-ci nous n'avons pas à regarder dans la liste grace au tableau App qui nous indique en complexité O(1) si la case appartient ou non.

2.5
Après avoir tracé les courbes en ayant effectué différents test,
nous pouvons constater :
- En fonction de la couleur, la structure acyclique reste quasiment constante a l'inverse des deux autres, et est inférieur à celles-ci
- En fonction de la dimension, la structure acyclique est aussi bien plus rapide que les deux précédentes

Conclusion : La structure acyclique est bien plus rapide que la recherche impérative, elle même plus rapide que la recherche recursive.

PARTIE II

Exercice 3 :

Création du graphe :

Il faut initialiser tous les sommets, et déterminer la zone de même couleur correspondante. On procède de cette façon : on parcourt toutes les cases : si la case n'est pas encore associée à un sommet, on initialise le sommet, puis on trouve la zone de même couleur que cette case grâce à la fonction trouve_zone_imp, et on fait pointer toutes les cases trouvées par la fonction sur notre sommet. La vérification du début permet de ne pas stocker plusieurs fois la même information.

On crée ensuite les voisins de cette façon : On crée de listes temporaires de type cellule-som, on parcourt les 2 listes en même temps, si 2 sommets sont adjacents on les ajoute chacun à la liste de sommets adjacents de l'autre.

Exercice 4 :

Stratégie max bordure :
Pour cette stratégie , nous avons décidé de procéder de cette façon:
-On n'utilise qu'un seul sommet (en l'occurrence le premier sommet en haut a gauche) auquel on ajoute les cases des sommets appartenant a la plus grande liste de sommets adjacents de même couleur et avec le plus grand nombre de cases
-Pendant le parcours des sommets de la plus grande liste on :

-ajoute les case au sommet principal

-ajoute les sommets adjacents de celui ci au sommet principal(si il ne lui appartiennent pas déjà)

Ceci se fait grâce à la fonction maj_bordure_graphe, appelée par la fonction stratégie_max_bordure, tant qu'il y a une bordure.
-par la suite nous détruisons la bordure (fonction détruire bordure) et nous en créons une nouvelle grâce au sommet unique qui est mis a jour (cela est fait par la fonction maj_bordure_graphe qui appelle init_bordure

Il s'agit donc d'une fusion des sommets, nous pourrions donc détruire les sommets fusionnés… L'avantage de notre technique est que nous n'utilisons qu'un seul sommet ,ce qui est un grand gain de temps et d'espace mémoire

Exercice 5 :

1) En parcourant les différentes combinaisons de sommets pour partir de la case en haut à gauche et arriver à la case en bas à droite et en retenant la plus courte (grâce

aux pointeurs de chaque sommet sur son père) c'est à dire faire un parcours en largeur, on peut stocker la couleur correspondant à chaque père dans une liste de couleurs.

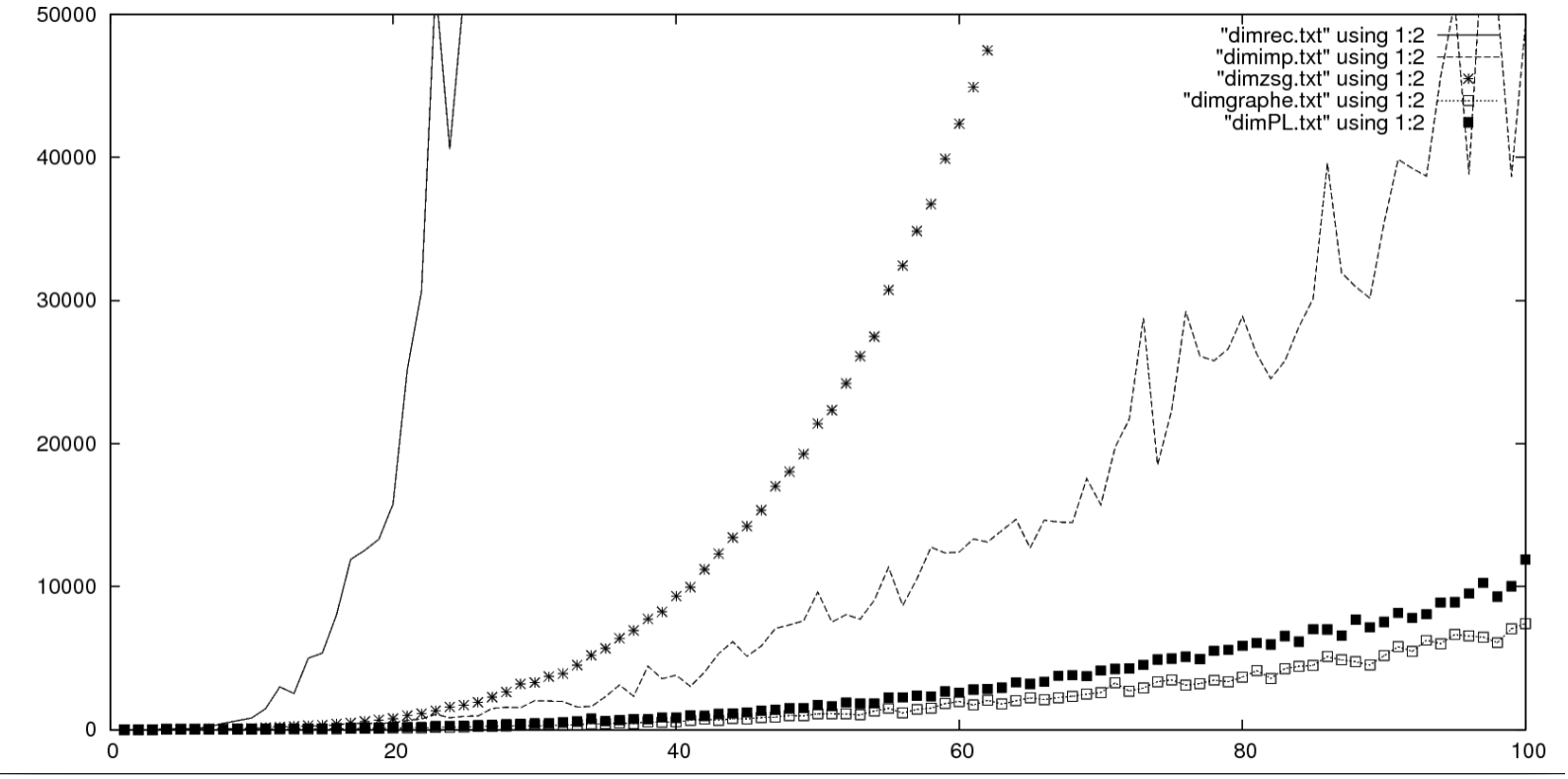
L'algorithme de plus cours chemin fonctionne en 2 parties. Tout d'abord parcours_largeur, qui analyse tous les chemins de la case [0][0] à la case [dim][dim], et qui retient le plus court. Cette fonction change la marque des différents sommets (pour visité, 2 pour non visité et 0 pour appartenant au sommet), et en les faisant pointer sur leur père .

On lance ensuite l'algorithme plus_court_chemin, qui parcourt une première fois la liste des sommets du chemin à l'envers (en remontant par les pères), et qui crée une liste de couleurs correspondant à l'ordre dans lequel les sommets du chemin apparaissent. (le premier père étant le sommet [0][0]. Dans une deuxième boucle, il lance la fonction mise_a_jour_zsg, qui va opérer les différents changements de couleur dans l'ordre de la liste, en prenant en compte et en modifiant les marques des sommets. mise_a_jour_zsg se charge aussi de redessiner la grille.

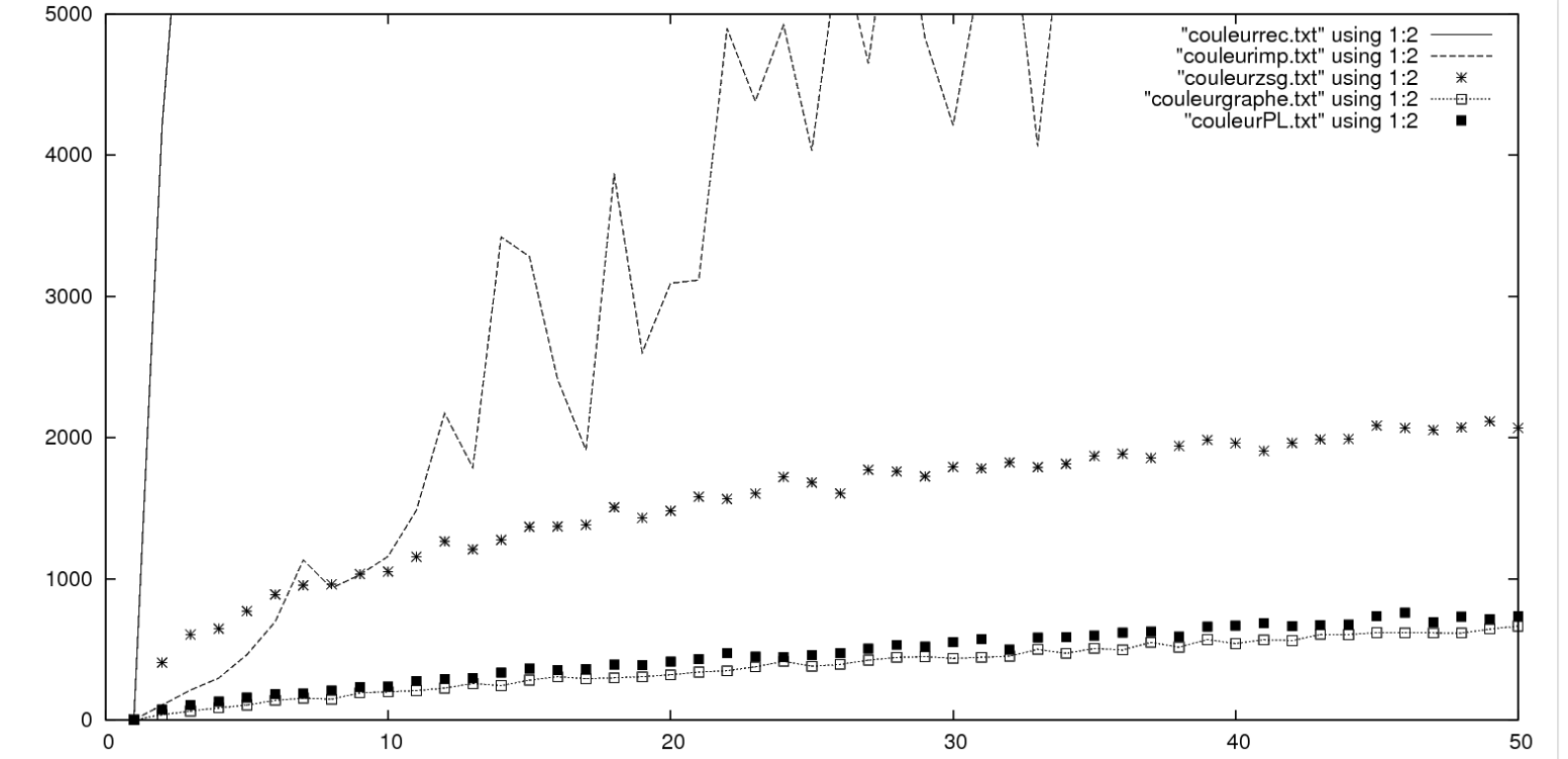
Quand ces 2 étapes sont terminées, on lance la stratégie_max_bordure, qui va avoir comme base le premier sommet, qui fait donc maintenant toute la diagonale, et elle pourra donc opérer sur les 2 cotés de la grille en même temps, améliorant sa vitesse.

Nous avons fait une deuxième stratégie des plus courts chemins, dans laquelle on la lance aussi pour les chemins de [0][0] à [0][dim-1] et de [0][0] à [dim-1][0], ce qui améliore encore la vitesse de l'algorithme.

Nombre de coups, courbes :



temps en fonction de la dimension (5 couleurs)



temps en fonction de la couleur (dimension fixe à 20)

Il est facile de faire des observations sur les courbes : Aussi bien sur la courbe en fonction des couleurs que la courbe en fonction des dimensions de la grille, 2 algorithmes sont remarquables de par leur vitesse : l'algorithme max bordure et l'algorithme de parcours en largeur. Bien que le nombre de coups du parcours en largeur soit moins élevé (environ 1300 pour la version normale et 1000 pour la version avec 3 parcours, contre 1700 pour le max bordure), le max bordure reste un tout petit peu plus rapide que le parcours en largeur. Vient ensuite l'algorithme des zones, qui est plus lent mais assez constant, puis l'algorithme impératif, encore plus lent et totalement inconstant, et enfin l'algorithme récursif, qui est constant mais dont les valeurs explosent très rapidement.