Rapport TP algorithmique et programmation

Coloriage d’images

# Réponses

## Question 1

Pour vérifier le format du fichier, nous pouvons vérifier que la première ligne correspond bien au « nombre magique » P1. Ensuite nous vérifions que chaque caractère de chaque ligne du contenu du fichier soit un 0 ou un 1 ou un caractère de séparation, si ce n’est pas le cas on lève une exception. On lève aussi une exception si la taille de fichier indiquée en début de fichier ne correspond pas au nombre de caractères lus.

## Question 2

Le cas d’erreur à traiter est une erreur lors de la création ou l’écrasement du fichier.

Pour tester cette fonction nous avons implémenté une fonction testEcriture() dans testWriteFic.cpp. Cette fonction crée un tableau de pixel de quatre cases et un objet WriteFic avec le nombre de ligne du futur fichier à 2 et le nombre de colonne à 2. On lance la fonction writeThePPMFic qui transcrit ce tableau dans un fichier ppm avec une couleur par pixel qui sera nommé result.ppm. Il n’y a qu’une couleur par pixel car ceci n’appartient pas à un ensemble. Pour vérifier les résultats on a juste à regarder le fichier result.ppm qui doit être constitué de la manière suivante : deux ligne de chacune un pixel qui ont chacun une couleur différente.

## Question 3

Pour tester la fonction generate, nous avons fait la fonction testGenerate qui exécute le generate puis lis le fichier généré pour vérifier si celui-ci est conforme.

## Question 4

Pour makeSet(x) qui correspond au constructeur de la classe « Ensemble » fut testé ainsi : on construit un ensemble et ensuite on vérifie que le head de l’ensemble correspond bien au pixel qu’on a passé en paramètre.

Le findset(x) correspond à la fonction getHead() de la classe Pixel. Pour la tester on fait un makeset(x) puis on vérifie que findset(x) renvoi bien x.

## Question 5

MakeSet(x) correspond au constructeur de la classe « Ensemble ». La complexité asymptotique relative au nombre d’affectation est de 3 ce qui correspond à Θ(1). Le coût en mémoire est de 3 car il y a trois affectations.

FindSet(x) correspond à la fonction getHead() de la classe Pixel. Sa complexité asymptotique relative au nombre d’affectation est de Θ(1) car il n’y a qu’un « return ». Son coût en mémoire est de zéro.

## Question 6

Soit X et Y deux ensembles. On commence par faire pointer tous les pixels de Y vers le représentant de X. C’est-à-dire que chaque pixel de Y aura pour attribut « \_head » un pointeur vers le « \_head » de X, qui est le représentant de l’ensemble X.

Ensuite on rattache la tête de Y à la queue de x. La queue de X devient la queue de Y.

Enfin on détruit le contenant Y qui ne sert plus, le contenant X étant devenu l’ensemble XY.

Dans le pire des cas la complexité asymptotique dans le pire des cas est de O(n) car cela correspondrait à faire (n -1) changement de pointeur vers le nouveau représentant et un changement de pointeur vers le premier pixel de Y pour la tail de X. Le coût en mémoire correspondrait à la taille des deux ensembles.

## Question 7

Dans le pire des cas la complexité asymptotique serait de O(n²) en effet dans le pire des cas il faudrait dans un premier temps faire n création d’ensemble.

Ensuite on unit les ensemble S1 et S2, S1,2 avec S3 comme on ne prend pas compte de la taille dans le pire des cas S1,2 va changer le représentant de chacun de ses pixels comme étant le représentant de S3 ainsi de suite. On a donc un coût de 2 pour l’union S1,2, de 3 pour l’union S1, 2, 3 … et de n pour l’union avec Sn. .

## Question 8

Une structure de données permettant de réaliser ces actions plus rapidement serait de stocker la taille dans l’ensemble afin d’ajouter le plus petit ensemble à la fin et donc de diminuer le coût de l’union.

En effet, vu qu’à chaque union d’ensembles, l’ensemble qu’on uni doit changer son représentant et donc tous les pointeurs des pixels vers le représentant. Le mieux est de faire le moins d’affectation en prenant le plus petit ensemble.

Une autre structure de données permettant de réaliser ces actions plus rapidement serait d’organiser les éléments appartenant à un ensemble en un **Arbre.**

La matrice de pixel serait donc enfaite une forêt (plusieurs ensembles représentés par des arbres).

## Question 9

Cf algo\_union dans union.h

## Question 10

Si on crée n ensembles, nous avons un coût de n. Si ensuite tous ces ensembles entre eux on a une complexité de O(n).

En effet comme on se souci de la taille des ensembles lorsqu’on fait l’union de S1 avec S2 et de S1, 2 avec S3, le coût est toujours de deux : changement du représentant pour un ensemble à un pixel, et changement de la tail du plus grand ensemble.

Ce qui fait un coût de

## Question 12

Commençons par le cout mémoire :

Lors de la lecture du fichier, l’objet *LecteurImage* alloue n\*m\*16 octets. n et m étant la hauteur et la largeur de l’image. 16 octets représente la taille d’un objet *Pixel*.

Exemple : Pour une image de 2567 de largeur et 1091 de hauteur. Le cout mémoire sera de : 44,8Mo

Complexité asymptotique en pire cas :

* De l’algorithme d’union des ensembles

On applique au n\*m ensembles la fonction d’union addEnsemble\_inTail. (les ensemble sont composés au départ que d’un seul pixel)….

## Question 13

## Question 14

Pour éliminer le pointeur tail il suffit de rajouter une sentinelle. En effet ainsi lors de l’union de deux ensemble S1 et S2 avec S1  plus grand que S2 on procède de la manière suivante :

On fait pointer tous les pixels de S2àpartirdesondeuxièmepixel (le premier étant la sentinelle) vers le représentant de S1 (soit la sentinelle), à la fin de la boucle on fait pointer le dernier pixel de S2 vers le deuxième pixel de S1.

De cette façon nous n’avons pas besoin de tail et nous gardons la même complexité.