Rapport de projet SDA

Eric CHEN HUNG

Christopher SPLITZ

Introduction

Dans le cadre de notre 4eme semestre en License Informatique, dans le module structure de données et algorithmes, nous avons réalisé un projet qui a pour objectif de d’améliorer nos connaissances, de manipuler différentes structures de données et de comparer différentes solutions possibles pour la réalisation du projet.

Dans ce projet, nous devions étudier le coloriage automatique d’une image PBM en noir et blanc. Pour cela, nous définirons des ensembles pour représenter les pixels de l’image, implémentés grâce aux structures vues en structure de données et algorithme.

Nous avons suivi l’approche proposée dans le sujet. Le projet étant à réaliser en binôme, nous avons décidé de répartir le travail de la façon suivante :

Lecture des fichiers et écriture des images-> Christopher / Eric

Implémentation des listes chainée -> Eric

Implémentation des arbres -> Christopher

Nous nous sommes départagés de cette façon car cela nous permet de gagner en efficacité, les implémentations des structures pouvant se faire indépendamment. Cependant nous avons travaillé ensemble sur la manipulation de fichier en premier pour pouvoir vérifier nos implémentations par la suite. Bien entendu, nous nous sommes également aidés pour l’implémentation quand il le fallait.

Lecture / écriture des données

1. Pour pouvoir colorier, il faut d’abord la lire. Pour cela, nous avons implémenté une fonction Read.

Avant d’implémenter cette fonction, nous avons implémenté les structures PBM et PPM, pour pouvoir stocker les images. La fonction Read prend le nom de l’image en paramètre et nous envoie une structure PBM stockant l’image en sortie.

Pour cela, la fonction ouvre l’image, alloue l’emplacement mémoire d’uns structure PBM et y stocker les informations contenues dans l’image.

1. Nous avons ensuite implémenté la fonction Write, qui à partir d’une structure PPM (image en couleur) créé un fichier image.

Cette fonction prend une PPM en entrée et crée un fichier image. La structure PPM contient le nombre magique, la largeur et la longueur de l’image ainsi qu’un tableau bidimensionnel contenant les couleurs des pixels.

1. La fonction Generate permet de créer une image aléatoire PBM.

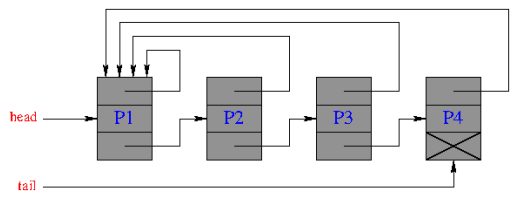
Nous avons décidé de mettre en paramètre de la fonction la largeur et la hauteur de l’image pour pouvoir dimensionner selon nos envies. Et nous pensons qu’une dimension aléatoire de l’image n’est pas nécessaire.

Implémentation des listes chainées

Pour implémenter les ensembles, nous avons décidé de suivre l’approches proposées dans le sujet. Nous avons alors défini 2 structures de données : 1 Chaine et 1 Liste.

La structure Chaine contient un tableau de 3 entiers pour la couleur, le représentant de l’ensemble, un pointeur vers le suivant et représente le chainon de la Liste qui n’est en faite qu’un pointeur vers la tête de la liste et vers la queue.

Voici de schéma de la liste données dans le sujet et que nous avons implémenté :



STRUCTURE GLOBAL DE L’IMAGE A TRAITE :

Même en créant une liste pour chacun des pixels. Nous avons besoin de repérer la position de chacun des pixels par rapport aux autres pour pouvoir connaitre les voisins.

Comme une image est un ensemble de pixel collé les uns à cotés des autres, nous avons décider de le représenter par un tableau bidimensionnel de listes qu’on a appelé une « TB ».

Chaque case du tableau contient un chainon de la liste et nous permet de connaitre facilement les cases autour ainsi que de nous faciliter leur accès.

1. Nous avons donc implémenté les fonctions permettant de définir un ensemble représentation d’un pixel dans l’image ainsi que de trouver son représentant.

La fonction Makeset prend en paramètre une valeur qui est en faite la valeur d’un pixel d’une image PBM.

Nous avons également implémenté quelques fonctions :

*listeNouv* : fonction renvoyant une liste vide

*estVide* : savoir si la fonction est vide

*insertion*: permet d’insérer un chainon dans une liste vide, elle permet également de remplir le tableau de valeur du chainon avec 0 si le pixel est noir ou avec une couleur aléatoire si le pixel est blanc.

La fonction Makeset fait appel à ces trois fonctions pour créer ces ensembles. Nous avons ensuite utilisé la fonction Makeset pour remplir notre TB en l’appliquant pour chaque pixel de l’image PBM.

Comme l’implantation des listes chainées contient un pointeur vers le représentant, la fonction Findset est simple car elle ne fait que renvoyer le pointeur.

1. Cout asymptotique :

Le cout asymptotique de Makeset en pire cas est linéaire car elle ne crée qu’un élément et ne parcours que le tableau de taille fixe 3 pour les valeurs du pixel. Son cout en mémoire est la taille de la structure Chaine.

Le cout asymptotique de Findset est également linéaire car elle ne renvoie qu’une valeur et son cout mémoire est quasi nulle car il n’y a pas d’allocation.

Cependant, la création et l’implémentation de notre structure TB a un cout asymptotique de 2n car pour n pixel dans l’image initiale, la fonction parcours le tableau de liste ainsi que le tableau de pixel de l’image PBM. De plus son cout mémoire est de n fois la taille de la structure Liste.

1. La fonction union permet de fusionner 2 listes entre elles.

Malheureusement, nous n’avons pas réussi à implémenter cette fonction. Nous avons tout d’abord essayé d’ajouter une 1 entier à la liste qui correspondrait à l’identifiant de la chaque ensemble. Cependant cette méthode ne correspondait pas à l’implantation des listes chainées car en effectuant l’association de 2 chaines juste par leurs identifiants et non pas par leurs chainages, la structure de liste chainée ne servirait à rien et l’union ne serait que l’union de 2 cases d’un tableau.

Nous avons donc opté pour une simple fusion de chaine en redirigeant les pointeurs. Cependant, utilisé sur des images complexes de grandes tailles les chaines à fusionner sont complexe et comme il n’y a aucun moyen d’accéder à la liste depuis la chaine, nous n’avons pas réussi à rediriger la tête et la queue des listes créant ainsi des ensembles difformes.

1. Complexité en pire cas d’une séquence de n création d’ensemble suivi des unions de l’ensemble.

Comme indiqué précédemment, la création des ensembles aura un cout linéaire en pire cas. De plus, l’union que nous avons implémenter devra dans le pire des cas parcourir tous les ensembles pour chaque ensemble crée pour pouvoir rediriger les pointeurs ; il a donc un cout en pire cas de n\*n = n^2.

1. Nous avons implémenté l’algorithme de coloriage grâce à une fonction nommée *voisin.*

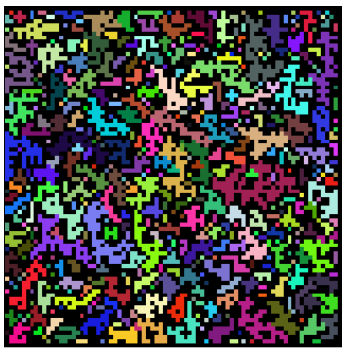
La fonction voisin parcours tout le tableau de liste pour regrouper les ensembles. Malheureusement, nous n’avons pas réussi à implémenter correctement la fonction union. Le programme ne marche alors pas correctement les une parti des pixels ne sont pas colorié de la bonne couleur.

Pour la tester, malgré notre échec, nous avons utilisé notre programme sur des images simples, sur des images aléatoires et enfin sur les images tests données dans le sujet.

IMAGES SIMPLES :



IMAGE ALEATOIRE :



IMAGES TESTS :



1. Complexité de l’ensemble du programme :

Voici la complexité de des fonctions utilisées :

Makeset = O(1)

Findset = O(1)

creerTB = O(2n)

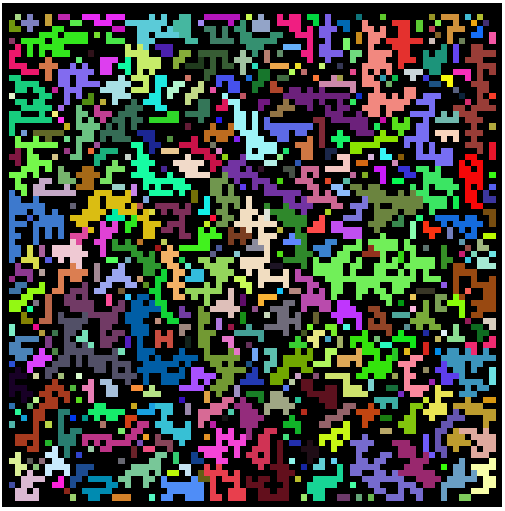
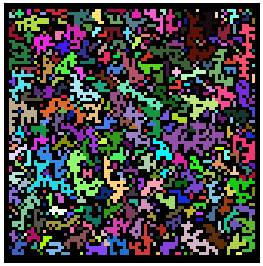
voisin + union = O(n^2)

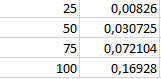
On peut donc en déduire que le cout asymptomatique est de 2n + n^2.

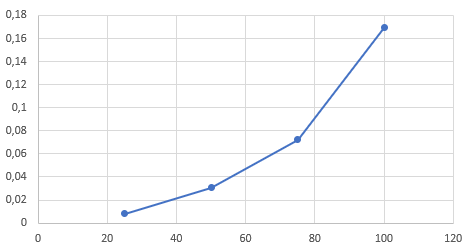
Le cout en mémoire une change pas car on ne crée pas de tableau ou de structure auxiliaire du l’algorithme, le cout en mémoire est donc le tableau de n liste + n chaines.

1. Test avec des images aléatoire :

Nous avons effectué différents test sur des images de taille différentes. Cependant, comme la fonction union ne fonctionne pas comme elle le devrait. Le temps devrait être plus élevé car dans le programme on n’a pas effectué tous les parcours.







Comme nous pouvons le constater, plus la dimension de l’image augmente plus la durée augmente de façon conséquente. Sachant que le temps mesuré dépend également de la complexité de l’image. Si l’image est trop simple, nous n’avons pas besoin de parcourir tous les pixels n fois.