Rapport Propagatio – Arthur Chansel / 324265

Analyse

Mon programme utilise jusqu'à 3 fonctions par tâche, certaines sont appelées dans le main, d'autres sont appelées depuis une autre fonction.
J'ai créé une structure « Grille » assez complète. Cette structure représente le graphe comme un « objet », avec plusieurs propriétés : matrice d'adjacence, taille, tableau « visited », tableau multidimensionnel de séparation, et liste des degrés de séparation moyen. Cet outil m'a permis de réutiliser des informations déjà calculées grâce à un nom clé plus lisible.
Pour chacune des tâches, j'ai décomposé les fonctions en sous-problèmes (principe d'abstraction) dont les résultats mis ensemble m'ont permis de résoudre un problème de plus grande ampleur. Ces sous-problèmes peuvent être testés indépendamment, et ils fournissent une sortie robuste : *scaffolding*.
Tâche 1 : Le programme vérifie le format et les dimensions du fichier .pbm. Si tout va bien, la matrice est stockée et symétrisée dans une structure (Grille) « mat_adj » déclarée au début du main.
Tâche 2 : Le programme parcourt de manière récursive la matrice d'adjacence, et met à jour un tableau « visited » associé aux nœuds. À la fin de cette étape, une fonction vérifie la connexité du graphe en inspectant simplement l'état du tableau visited.
Tâche 3 : Une fonction récursive me permet de créer une liste des nœuds qui se situent à un degré de séparation d'un nœud de référence. Pour cette tâche, le nœud de référence est 0. Enfin une autre fonction affiche la page[1] 0.
Tâche 4 : Je réutilise la fonction récursive de la tâche 3 pour chacun des nœuds. Je stock toutes ces informations dans un tableau multidimensionnel[2]. Une fonction calcule le degré moyen de séparation de chaque nœud en se basant sur ce tableau. Elle est appelée par une autre fonction pour faire la moyenne de toutes les moyennes.

[1, 2] Pour mieux me représenter ce tableau multidimensionnel, j'ai imaginé que c'était un livre : chaque page contient plusieurs lignes/listes, et chaque liste contient plusieurs éléments. La page n répertorie tous les nœuds en fonction de leur degré de séparation d'un nœud de référence n. Ces analogies sont utilisées dans le programme.

Complexité

J'estime l'ordre de complexité de la tâche 4 à O(N^3 + 2N^2 + 9N + 7). Pour de grandes valeurs, on peut considérer un ordre de complexité d'environ O(N^3).

Time « user » :

T07 :	0.50s	(taille : 511x511)
T08 :	2.74s	(taille : 1023x1023)
T09 :	16.33s	(taille : 2047x2047)
T10 :	110.85s	(taille : 4095x4095)

On peut observer que le temps d'exécution augmente (approximativement) d'un facteur 6 à chaque test. Sachant qu'on augmente aussi le nombre de nœuds dans chaque fichier test d'un facteur 2, le temps de calcul grandit selon une complexité polynomiale : environ O(N^2,6). Or l'ordre de complexité calculé nous indique que le temps d'exécution devrait lui augmenter d'un facteur 8 (= 2^3).
Le temps en pratique est plus court que ce que l'on avait prédit. Déjà c'est plutôt une bonne nouvelle, il vaut mieux que le temps d'exécution soit plus court que nos attentes. Mais cherchons à comprendre d'où vient cette différence. L'énoncé du projet déclare que l'ordre de complexité de l'algorithme récursif qui détermine les (n-1) degrés de séparation d'un seul nœud de départ est O(N^2). J'ai utilisé cette donnée dans mon calcul. Mais il est possible que le véritable ordre de complexité de ma fonction récursive soit majoré par O(N^2). Ainsi l'ordre de complexité de mon programme serait plus petit que O(N^3) et cela expliquerait cette différence.

Pseudocode

Tâche 3
Entrées modifiées : mat

Initialiser(depart)
Ajouter(depart, 0)
Initialiser(temp)
Ajouter(temp, depart)
Ajouter(mat.degres, temp)
mat.visited(0) ← 1
algo_recursif(1, 0, mat, depart)
afficher_page(mat)

---

algo_recursif
Entrées non modifiées : step, start
Entrées modifiées : mat, list

Initialiser(new_line)
Pour i allant de 1 à **Taille**(list)
    Pour j allant de 1 à mat.taille
        Si non mat.visited(j) **ET** mat.tab(list(i))(j) = 1
            mat.visited(j) ← vrai
            Ajouter(new_line, j)
new_line ← **tri**(new_line)
Si Taille(new_line) > 0
    Ajouter(mat.degres(start), new_line)
    algo_recursif(step + 1, start, mat, new_line)

---

afficher_page
Entrées modifiées : mat

Pour l allant de 1 à **Taille**(mat.degres(0))
    Pour i allant de 1 à **Taille**(mat.degres(0)(l))
        afficher mat.degres(0)(l)(i)