Rapport de Graphe

Préparé pour : Mathieu Liedloff, Ian Todinca
Préparé par : Victor Coutellier, Julien Gauttier
4 décembre 2015

INSTRUCTION D'UTILISATION

Le projet étant compilé sous JAVA 8, il requiert donc cette version pour s'exécuter. Ne possédant aucune dépendance, il suffit simplement d'exporter le projet dans un jar, et de l'exécuter avec en second argument le chemin vers le fichier .graph que l'on veut tester. La sortie texte affiche le graphe sous forme planaire si possible, et génère un fichier .dot dans le dossier contenant le .jar de l'application.

STRUCTURE DU PROJET

Architecture

Le projet JAVA utilise le modèle MVC, bien que peu développé car aucune vue n'est intégrée, seul le contrôleur est appelé par la class Main afin de lancer l'algorithme de test de planarité du graphe. Malgré tout, le modèle assez fourni donne tout un tas d'outils au contrôleur qui exécute l'algorithme sous sa forme la plus simple comme décrit dans le sujet.

Une distinction entre le graphe G à tester et le sous-graphe H généré lors du traitement de l'algorithme à été faite. En effet, des class MainGraph et SubGraph héritent de Graph afin de partager des attributs commun, mais leur fonctionnement étant bien distincts, des classes séparées on été crées pour plus de clarté.

L'objet SubGraph servant à plonger un chemin dans un graphe, ainsi qu'à gérer ses différentes faces, tandis que la l'objet MainGraph permet de gérer les fragments d'un graphe.

Les objets Face, Fragment et Noeud sont les seules autres objets du modèle, car ils sont directement utilisés dans l'algorithme, leur gestion est donc rendue simplifiée.

Des class utilitaires tel que Read et Write servent quant à elle à la lecture et à l'écriture du graphe.

Difficultés

La principale difficulté du projet fût la compréhension de l'algorithme, en effet, en apparence très simple, son implémentation nécessite de comprendre les subtilités des définitions, et un certains temps à été nécessaire pour intégrer tout ces concept. Mais une fois ces définitions maitrisés, il fût tout de suite plus simple de synthétiser les actions de l'algorithme pour enfin découper les différentes class utilisées.

La seconde difficulté fût le transfert des références des objets entres les différentes class. En effet, les points d'un graphe étant représentés par des objets, ils étaient utilisés dans tous les autres objets de notre modele, il fallait donc choisir entre passer l'objet ou une copie de ce dernier. De plus, des fonction equals() spécifiques ont du être implémentés afin de conserver l'intégrité de notre structure.

ANALYSE DE COMPLEXITÉ

Algorithme principal

L'algorithme principal se situe dans la class GraphController, et est ligne pour ligne l'algorithme donné dans le sujet, pour calculer sa complexité nous allons analyser plusieurs des sous algorithmes présents

Sous-algorithme

Différents sous algorithme sont présent dans le projet, tel que :

- Un parcours en profondeur pour trouver le premier cycle : n
- N parcours en profondeur dans chaque fragment pour trouver un chemin : n*n
- Parcours des différents graphes pour générer les fragments : n
- Parcours des chemins : n
- Comparaison de deux listes de noeuds : n*n

Traitement java

Les traitements inhérent à JAVA et aux fonctions du programme représentent une partie non négligeable de l'algorithme, car effectués de manière transparente par la machine virtuelle. En effet, un nombre conséquent de parcours de liste, de création d'objet et calcul de comparaison sont effectués. Malheureusement, ces temps sont relatifs à la machine virtuelle de JAVA et nous ne pouvons donc pas en tenir compte dans notre analyse.

Conclusion

A chaque itération les sous programmes les plus gloutons sont de type O(n) = n*n on peut en conclure que la complexité globale de notre implémentation est : $O(n) = n^3$.

Victor Coutellier, Julien Gauttier

Dépot git : https://github.com/alistarle/Projet-M1-Graphe