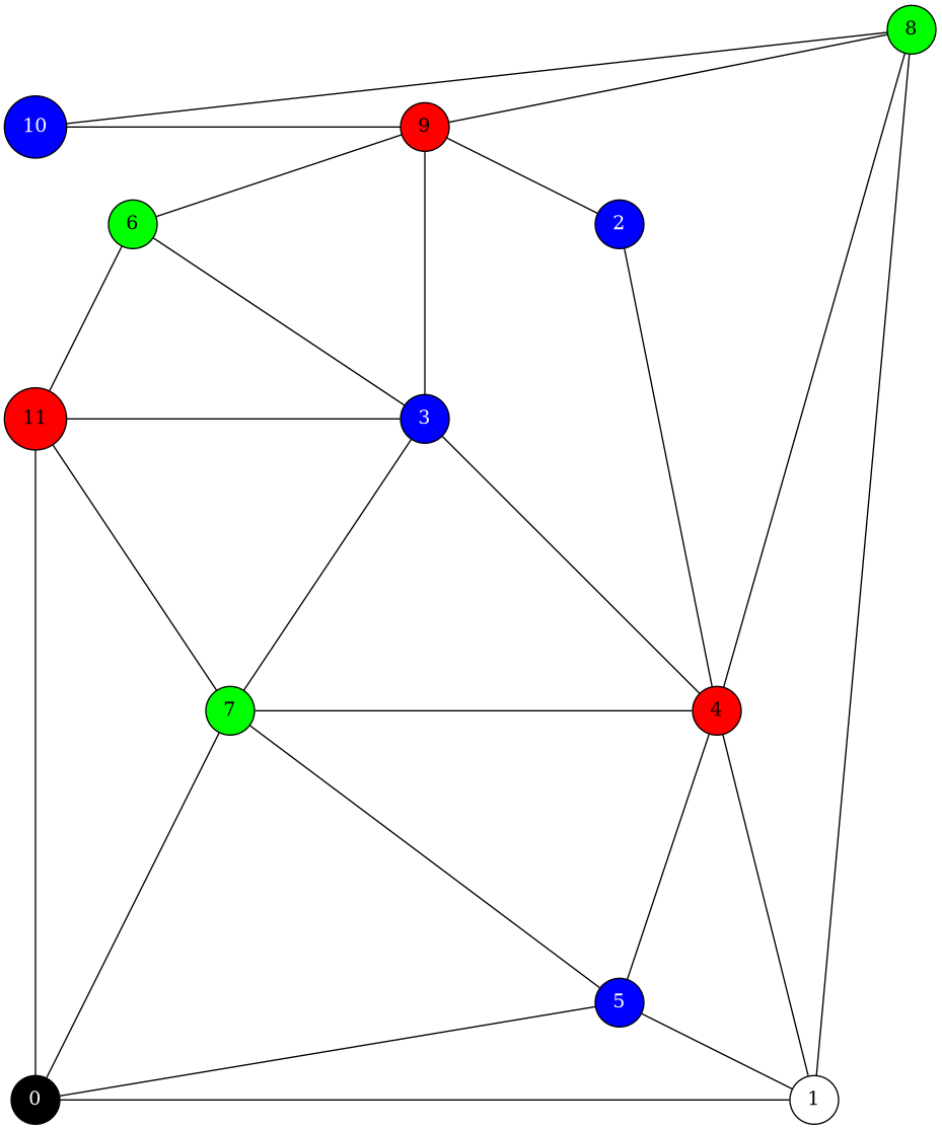
Rapport du projet de Modélisations, Graphes et Algorithmes

WALCAK Ladislas, RIBARDIERE Tom



1. Analyse de complexité en temps et espace

A. Complexité en temps

Soit N le nombre de sommets du graphe, et M le nombre de voisins d'un sommet. Dans le pire des cas:

Complexité de Graph::coloring() :

O(Vertex::colorize()) ~ O(N²)

Complexité de Vertex::colorize() :
Colorize est appelé sur les N sommets une et unique fois. Pour chaque sommets, dans le pire des cas, la recherche des couleurs des voisins est faite M fois (si le sommet est lié à tous les autres sommets). Pour finir, dans le pire des cas, aucune couleur n'est disponible après les 5 premier appels, et donc l'on doit colorier le sommet avec la fonction Vertex::flipIfUnreachable(), de complexité O(2N). Après simplification des appels à Vertex::flipIfUnreachable() en omettant le fait qu'il ne peut être appelé que N - 5 fois, on obtient une complexité de O(2N² + M), soit une complexité globale de O(N²)

O(N * (M + Vertex::flipIfUnreachable())) ~ O(2N² + M) ~ O(2N²)

Complexité de Vertex::flipIfUnreachable() :
La fonction Vertex::flipIfUnreachable() effectue un parcours en profondeur qui dans le pire des cas est de complexité O(N - 1) car un sommet est ignoré. Ensuite, si les deux sommets ne sont pas connectés, il faut inverser les couleurs de tous les sommets du parcours en profondeur, ainsi que la couleur du sommet ignoré, soit O(N). En simplifiant O(N - 1) en O(N), on obtient une complexité de O(2N).

O((N - 1) + N) ~ O(2N)

B. Complexité en espace

Soit N le nombre de sommets du graphe, et M le nombre de voisins d'un sommet.

Soit S la taille mémoire d'une chaîne de caractère.
Soit P la taille mémoire d'un pointer.
Soit I la taille mémoire d'un entier.

Taille des données :
(N * taille d'un Vertex) + taille d'un Graph

Taille de Graph :
Une map de N chaînes de caractères, chacune couplée à un pointeur vers une object Vertex. L'objet Graph est donc de taille N * (P + S)

Taille de Vertex :
Un vertex est composé d'un identifiant, un liste de pointer vers ses voisins, ainsi que de trois entiers, représentant la position x, y ainsi que sa couleur. L'objet Vertex est donc de taille (M * P) + 3 * I

Complexité de Graph::coloring() :
La fonction Graph::coloring() n'utilise aucun espace, elle se contente de lancer Vertex::colorize(). Elle a donc une complexité d'espace de O(1).

Complexité de Vertex::colorize() :
La fonction Vertex::colorize() créer une liste d'entiers *colors* de taille 5 * I. Elle crée également la liste d'entiers des couleurs des voisins, qui dans le pire des cas est aussi de taille 5 * I. Enfin, dans le pire des cas, tous les appels de Vertex::colorize() font appels à Vertex::flipIfUnreachable(). Si l'on néglige la taille des entiers et des pointeurs, l'on obtient $O(2N^2)$

$$
\begin{aligned}
&O(N * ((10 * I) + \text{Vertex::flipIfUnreachable()})) \\
&\quad \sim O(10N + (2N^2 * P)) \\
&\quad \sim O(10N + 2N^2) \\
&\quad \sim O(2N^2)
\end{aligned}
$$

Complexité de Vertex::flipIfUnreachable() :
La fonction Vertex::flipIfUnreachable() effectue un parcours en profondeur itératif. Il lui faut donc pour cela 2 listes qui contiennent des pointeurs vers des Vertex. Par la suite, si elle effectue une inversion de couleurs, aucun espace supplémentaire n'est requis. La complexité est donc de O(2 * N * P)

$$O(2 * N * P)$$

2. Compilation et exécution du code

Nous dépendons de 2 librairies: libgvc et libcgraph. Ces deux librairies sont contenues dans le package Graphviz.

Afin de compiler le programme, nous utilisons l'outil Cmake. Pour obtenir des informations détaillées sur le processus de compilation, nous vous renvoyons vers le fichier README.md fourni avec le programme.

Pour lancer le programme, il vous faut au minimum renseigner le chemin vers un fichier .graphe décrivant le graphe que vous souhaitez colorier. Dans le cas où vous souhaiteriez également obtenir une sortie graphique, vous devez également fournir le chemin vers un fichier .coords contenant les coordonnées des différents points du graphe.

Après l'exécution du programme, vous obtiendrez au minimum un fichier out.colors, ainsi qu'un fichier out.png si vous avez renseigné un fichier de coordonnées.

Attention, une entière confiance est placée dans les fichiers fournis par l'utilisateur, et donc aucune vérification ne sera faite sur la validité de ceux-ci.

Pour obtenir des informations détaillées sur le processus d'exécution du programme, nous vous renvoyons une nouvelle fois vers le fichier README.md fourni avec le programme.

3. Temps d'exécution graphes sur Celene

Nous avons calculé les temps d'exécution grâce à l'outil time de bash, et nous avons répété les exécutions 100 fois pour chaque graphe, afin d'avoir une moyenne.

Nombre de noeuds	**Temps d'exécution min**	**Temps d'exécution max**
10	0.003s	0.004s
12	0.003s	0.004s
50	0.003s	0.005s
100	0.004s	0.007s

4. Choix d’implémentation, difficultés rencontrées, solutions proposées

A. Choix d'implémentation

Nous avons décidé d’implémenter la version récursive de l’algorithme donné dans le sujet, car celle-ci nous semblait plus naturelle et compréhensible.

De plus, lors de la création de la classe Graph, nous avons fait le choix d’utiliser une std::unordered_map pour stocker les Vertex contenus dans le graphe, car cette structure de données permet un accès et une recherche des éléments en son sein en complexité O(1), permettant de grandement améliorer la rapidité du programme.

Enfin, la liste d'adjacence contenue dans la classe Vertex a été implémentée sous la forme d’un std::vector, car celle-ci doit garder l’ordre d’insertion des voisins.

B. Problèmes rencontrés

Nous avons eu des problèmes de compréhension de la brique 6. Et après avoir cherché et trouvé un moyen pour implémenter cette brique, nous avons eu des difficultés à tester cette même brique, car les graphes fournis n’en avaient pas besoin. Nous avons finalement pu tester notre implémentation avec un graphe que nous avons construit nous même avec des couleurs prédéterminées.

Le second problème auquel nous avons fait face a été la gestion des dépendances pour des librairies graphiques. En effet, dû à notre faible expérience avec l’outil Cmake, la liaison avec les librairies externes à été fastidieuse.

Enfin, le dernier problème rencontré est que la library Graphviz crée des fuites mémoires dans le programme, que nous avons pu remarquer grâce à l’outil Valgrind. Cependant, il nous est impossible de remédier à ce problème.