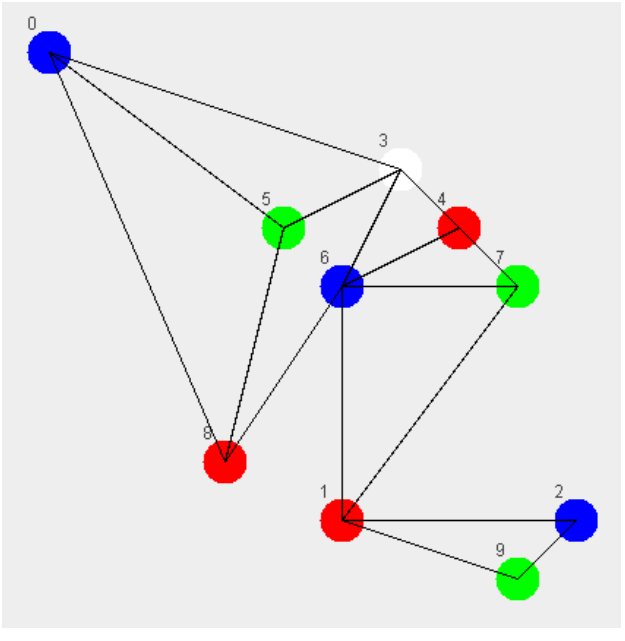
Modélisation, Graphe et Algorithmes
Projet 2020 - 2021



La 5-coloration des graphes planaires

SOMMAIRE

LANCEMENT DU PROGRAMME

Le programme a été développé en Java 13 avec Maven afin de faciliter l'écriture de celui-ci que ça soit pour la partie implémentation ou bien pour la partie graphique. De plus, nous avons choisi Maven afin d'importer si nécessaire des dépendances (ce qui était prévu à la base pour la partie graphique avec JGraphix), et d'avoir un lancement similaire indépendant de l'IDE.
Pour lancer le programme, il est donc nécessaire d'effectuer la procédure suivante :

- Installer maven
- Ouvrir un terminal et se placer dans le dossier
- Taper mvn package ou mvn package exec:java -D»exec.mainClass»=»mga.Launcher»

Note: Il est possible que la première commande ne fonctionne pour une raison inconnue.

Cela va lancer le programme avec les paramètres par défaut qui sont les suivant :

- Lecture du fichier de graphe « resources/JoliGraphe10.graphe »
- Lecture du fichier de coordonnée « resources/JoliGraphe10.coords »
- Écriture du résultat dans le fichier « resources/coloredGraph.colors »

Pour changer les fichier lus, il suffit de modifier les constantes de la classe « Launcher » :

- graphFile correspond au fichier de graphe
- graphPointPosition correspond au fichier donnant la position des sommets du graphe.

Pour finir, le programme produira deux résultats :

- Le premier correspond au couple <Sommet, Couleur> (écrit dans le terminal)
- Le second correspond au graphe coloré affiché dans une fenêtre

On notera également que la taille du graphe sur la fenêtre n'est pas forcément adaptée. Pour changer cela, il est nécessaire de modifier la constante « graphSizeMult » dans la classe Launcher. Plus cette variable est grande, plus le graphe sera grand.

Modélisation, Graphe et Algorithmes
Projet 2020 - 2021

CHOIX D'IMPLÉMENTATIONS

Nous avons décidé d'implémenter la version récursive de la 5-coloration des graphes, en effet, lors de notre travail la difficulté majeure à laquelle nous avons fait face a été l'implémentation de la brique 6. Le problème venait du fait que nous n'avions pas entièrement compris en quoi consistait la brique 6.
Après avoir passé un moment à réfléchir, nous avons trouvé une solution pour résoudre ce problème. Pour implémenté la brique 6 :
Soit un sommet x, nous récupérons les voisins y de x et pour chaque y nous récupérons la composante connexe associée à y. Après avoir obtenu la composante connexe nous vérifions si un y' n'appartient pas à la composante connexe de y, et si c'est le cas nous donnons la couleur de y' à x et y' prends la même couleur que y.
Ainsi nous avons résolu le problème rencontré avec la brique 6.

Une autre difficulté à été de gérer les sommets qui possédaient plus de 5 voisins, pour cela, nous avons décidé de donner à ces sommets une couleur qui n'a pas été utilisées dans leurs voisinages sinon nous essayons d'appliquer la méthode que la brique 6.

LES FONCTIONS UTILITAIRES

Nous avons choisi d'implémenter quelques fonctions utilitaires en plus afin de faciliter l'écriture du code et en essayant de diminuer au maximum leurs coût:
Fonction 1 : Algorithme -> echangerCouleur
Cette fonction s'effectue en temps constant ou au pire des cas en temps O(log(n)). Elle permet d'échanger deux éléments dans la map du graphe coloré.
Fonction 2 : Algorithme -> getUnusedColorInNeighbors
Cette fonction renvoie une liste des couleurs inutilisées par les voisins d'un sommet donné en paramètre. Elle s'effectue en temps O(n) ou au pire des cas, en temps O(n log(n))
Fonction 3 : Algorithme -> getRandomSommet
Renvoi un sommet aléatoire dans le graphe donné avec une complexité de O(1)
Fonction 4 : Utils -> getColorName
Renvoi le nom d'une couleur sous forme de chaine de caractère plutôt que le toString classique de la classe Color de AWT.

ANALYSE DE COMPLEXITÉ

FileSystem.readFile -> O(n+m)
Algo.coloringRec(graph) -> $n^{2\log(n)} + 4n^3 + 2mn^2 + 2n^{3\log(n)}$ -> $O(n^{3*\log(n)})$
GetUnusedColorsInNeighbours -> O(n log(n))
Brique 4 -> O(n)
Brique 5 -> O(1)
Brique 6 -> $O(n^2 + n^2 + nm + n^{2\log(n)})$ -> $O(n^2)$
Brique 5' -> O(1)
Brique 6' -> $O(n^2 + n^2 + nm + n^{2\log(n)})$ -> $O(n^2)$
CheckColoring -> O(n+m)
FileSystem.writefile -> O(n)
Affichage des couleurs -> O(n)
ReadGraphicFile -> O(n+m)

Donc la complexité en temps et en espace de notre implémentation est :
$O(n+m+n^{3\log(n)} + n + m + n + n + n + m)$ ===> $O(n^{3 * \log(n)})$ ===> $O(n^{3 * \log(n)})$

LES TEMPS D'EXÉCUTIONS

Les temps données sont des approximations		
Nombre de sommet	Temps d'exécution (sans le graphique)	Temps d'exécution (avec le graphique)
10 sommets	0.004484699 secondes	0.1494323 secondes
12 sommets	0.0081986 secondes	0.1766408 secondes
50 sommets	0.008333 secondes	0.1851763 secondes
100 sommets	0.01262 secondes	0.1755862 secondes

LA PARTIE GRAPHIQUE

Nous avons choisi d'utiliser la bibliothèque AWT + Swing de java pour dessiner le graphe du fait de la simplicité de la fonction « paintComponent ».
Tout d'abord on récupère les données du fichier .coords et du fichier .graph et on les placent dans deux listes contenant les tuples à 4 éléments afin de stocker toutes les informations nécessaires.
Ensuite, on transforme ces données afin d'adapter la taille du graphe.
Et enfin, on appel la fonction « paintComponent » qui permet de dessiner des formes sur la fenêtre (des ronds colorés pour les points et des traits noir pour les arêtes).