5-coloration

Badet Maxime
Schivre Nicolas

Analyse de la complexité

On considérera dans cette section n le nombre de sommets et m le nombre d'arêtes.

On notera Coloring_rec l'algorithme récursif de 5-Coloration et Coloring-it sa version itérative.

Temps:

Brique 1 : Pour cette brique, la complexité est O(n) car on a une boucle qui parcourt tous les sommets dans le pire des cas.

Brique 3 : Pour cette brique, la complexité est O(n+m) à cause du parcours en largeur. Il y a 2 autres fonctions avec une complexité négligeable (inférieur à celle du parcours en largeur) graphBicolor qui récupère un sous-graphe composé de 2 couleurs en O(n) (un seul parcours des sommets) et l'inversion des couleurs qui s'effectue en O(n) également.

Brique 4 et 5 : Pour cette brique, la complexité est O(m) car on parcourt chaque voisins du sommet v, et, il peut dans le pire des cas en avoir m.

Brique 6 : Pour cette brique, la complexité est O(m*(n+m)) car on récupère les composantes connexes de tous les voisins du sommet v dans le graphe G/{v} via plusieurs parcours en largeur (au plus le nombre de voisins de v, soit m dans le pire des cas). Le parcours des composantes connexes pour savoir si un voisin du sommet v n'est pas dans cette composante se fait en $O(m^2)$ ce qui est négligeable par rapport à la complexité précédente.

Brique 7 : Dans cette brique, nous utilisons la fonction sorted de python qui implémente le tri Timsort qui s'exécute dans le pire des cas en O(n*log(n)).

BFS : Est la fonction de tri en largeur qui retourne au pire une liste en temps O(n+m).

Coloring_rec : A chaque appel de la fonction, on fait un appel récursif avec un sommet en moins jusqu'à ce qu'il n'y ai plus de sommets dans le graphe. On fait donc au total n appels récursif. Pour chaque appel on applique la brique 1,4,5,6 qui propose un temps d'exécution majoré par la brique 6 qui est 0(m*(n+m)). On a donc une complexité totale qui est au pire des cas O(n*(m*(n+m))).

Colorint_it : Tout d'abord, on tri les sommets selon les conditions de la brique 7 soit en 0(n*log(n)) afin de ne parcourir qu'une seule fois chaque sommet par la suite de manière gloutonne. Dans le parcours des sommets, on traite le graphe en retirant les i-1 premiers sommets et leur correspondance dans les voisinages ce qui s'effectue en O(m) ce qui est négligeable par rapport au tri proposé par la brique 7. Cependant, l'utilisation des briques 3,4,5 et 6 reste majorée en temps d'exécution par cette dernière soit O(m*(n+m)). Le contenu du parcours est donc supérieur en temps par rapport à celui du tri, on a donc une complexité finale qui dans le pire des cas est O(n*(m*(n+m))).

Espace:

Brique 1 : Cette brique ne prend pas d'espace.

Brique 3 : Dans cette brique nous avons besoin de stocker la couleur du sommet d'origine. De plus, nous devons stocker le grapheBicolor qui peut avoir au pire une taille O(n), ainsi que le graphe compConnexe (composante connexe) qui peut également avoir dans le pire des cas une taille O(m).

Briques 4 et 5 : On a une seule variable qui prend à l'initialisation les 5 couleurs possibles.

Brique 6 : On a la même variable que pour les briques 4 et 5 qui regroupent toutes les couleurs.
À ça s'ajoute graphMinusX qui est une copie du graphe, donc au pire de taille O(n). De plus, on a compsConnexe qui regroupe chaque composante connexe de chaque sommet et qui peut donc au pire être de taille O(n*m). Enfin, compConnexe et la composante connexe d'un sommet et donc est de taille au plus O(m) comme pour la brique 3.

Brique 7 : La fonction sorted à besoin d'au pire O(n), si chaque sommet a tous les autres sommets comme voisins.

BFS : Est la fonction de tri en largeur qui retourne au pire une liste de taille O(n).

Coloring_rec : On va pour cette version de l'implémentation de la 5-coloration avoir besoin d'un espace de O(n) pour la copie du graphe. Puis on a un choix à faire entre les brique 4 et 5 et la brique 6, la moins bonne en espace étant la brique 6 elle nous donne O(n)+O(n*m).

Colorint_it : Pour commencer, on stocke le résultat de la brique 7 qui est en O(n). On aura aussi besoin de stocker le sommet qui est en cours d'analyse et tous ses voisins, ce qui représente un stockage de O(n). De plus, grapheDegenere et une copie du graphe donc nous avons de nouveau un O(n). Pour finir on doit choisir entre les brique 4 et 5 et la brique 6 et la moins bonne en espace est la brique 6 de taille O(n)+O(n*m).

Exécution du programme

Pour exécuter le programme, il suffit de saisir la commande python3 projet.py.
Des commandes supplémentaires peuvent être ajoutées en fonction de vos besoins :

- Pour spécifier le chemin des graphes : python3 projet.py -c "chemin des graphes". Vous pouvez modifier le chemin du graphe que vous souhaitez exploiter en remplaçant "chemin des graphes" par le chemin depuis l'emplacement du fichier projet.py. Notez qu'il est important de ne pas inclure l'extension du fichier .graphe, mais uniquement son chemin et son nom. Le programme ajoutera automatiquement les extensions .graphe et .coords au besoin.
- Pour afficher graphiquement le graphe : python3 projet.py -a. Cela lance l'affichage visuel du graphe à l'aide de la bibliothèque matplotlib.
- Pour passer de la version récursive à la version itérative des implémentations que nous avons réalisé : python3 projet.py -i.

- Pour afficher plus de données dans le terminal, montrant la coloration des sommets sans avoir besoin du fichier .colors : python3 projet.py -v.

Enfin, vous pouvez combiner ces différentes commandes selon vos besoins.

Temps d’exécution

Graphe 50 sommets:

Lorsqu'on applique l'algorithme de manière récursive sur un graphe de 50 sommets, le temps d'exécution est de 0,0001237 secondes, tandis qu'en utilisant l'approche itérative, le temps d'exécution est de 0,0004467 secondes. Cela représente une efficacité environ 3 à 4 fois moindre avec la version itérative.

Graphe 12 sommets:

En ce qui concerne un graphe de 12 sommets, le temps d'exécution pour la version récursive est de 0,0000371 secondes, tandis que pour la version itérative, il est de 0,0000607 secondes.

choix d’implémentation

Dans l’optique de nous focaliser sur l'implémentation, nous avons privilégié le langage python pour sa facilité de programmation.
Pour implémenter les graphes, nous avons choisi les dictionnaires afin de pouvoir chercher les sommets par leur numéro.
Si la 5-coloration n'est pas réalisable, comme dans le cas d'un sous-graphe complet de taille 5 tel que le graphe complet K6, nous optons pour l'utilisation de la couleur jaune afin de visualiser plus clairement l'origine du conflit.