Compte Rendu TP1
Algorithmes et Arbres

Rayane KACHBI

9 novembre 2025

1 Introduction

L'objectif de ce TP était d'implémenter l'algorithme de coloration de graphe de Welsh-Powell. Pour cela, la consigne demandait d'utiliser une structure de données basée sur une matrice d'adjacence.

J'ai divisé mon travail en deux parties principales :

— L'implémentation de l'algorithme avec une matrice d'adjacence classique (`int **adjacence red↪ `).

— Une extension (demandée s'il restait du temps) utilisant une matrice d'adjacence linéarisée (`int *adjacence`).

2 Partie 1 : Matrice d'Adjacence Standard (`int **`)

2.1 Structure et Fichiers

J'ai d'abord créé les fichiers `graphe.h` et `graphe.c`. La structure `Graphe` est définie dans le `.h` et contient le nombre de sommets ainsi qu'un pointeur double vers la matrice :

```
typedef struct {
    int nb_sommets;
    int **adjacence;  // La matrice (N x N)
} Graphe;
```

2.2 Gestion de la Mémoire

La fonction `creer_graphe(nb_sommets)` a nécessité une double allocation :

1. Un `malloc` pour le tableau des pointeurs de lignes (`nb_sommets * sizeof(int*)`).
2. Une boucle pour allouer chaque ligne (colonne) avec `calloc(nb_sommets, sizeof(int))`. J'ai utilisé `calloc` pour initialiser directement la matrice à 0.

La fonction `liberer_graphe(g)` effectue l'opération inverse : elle parcourt chaque ligne pour la libérer (`free(g->adjacence[i])`), puis libère le tableau de pointeurs (`free(g->adjacence)`) avant de libérer la structure graphe elle-même.

2.3 Algorithme de Welsh-Powell

Pour l'algorithme, j'ai d'abord créé une structure temporaire `SommetInfo` pour stocker l'ID d'un sommet et son degré.

1. **Calcul des degrés :** J'ai parcouru la matrice d'adjacence pour calculer le degré de chaque sommet (en comptant les '1' sur chaque ligne).

2. **Tri des sommets :** J'ai utilisé `qsort` avec une fonction de comparaison personnalisée (`comparer_sommets`) pour trier les sommets par ordre de degré décroissant.
3. **Coloration :** J'ai initialisé un tableau `couleurs` à 0 (avec `calloc`). J'ai ensuite itéré, en commençant par la `couleur`$_actuelle = 1.Dansuneboucleimbrique, jeparcouraismalistedesommetstris.$

3. `Vérification : Pour chaque sommet non coloré, j'appelais la fonction peut_colorer red↪ (g, sommet, couleur, couleurs). Cette fonction vérifie si le sommet est adjacent à un autre sommet ayant déjà cette couleur. Si non, la couleur lui est assignée.`
4. `Itération : S'il restait des sommets non colorés, j'incrémentais couleur`$_actuelleetjerecom$

L'accès à la matrice se fait de manière standard, avec `g->adjacence[i][j]`.

3 Partie 2 : Extension (Matrice Linéarisée)

Comme il me restait du temps, j'ai implémenté la version optionnelle avec une matrice linéarisée.

3.1 Structure et Fichiers

J'ai créé de nouveaux fichiers `graphe`$_lineaire.het$`graphe`$_lineaire.c.Lastructure$`Graphe`$estmodifiepourn'avoi$

3.2 Gestion de la Mémoire et Accès

`La gestion mémoire est simplifiée :`
- `creer_graphe(nb_sommets) ne fait plus qu'un seul calloc(nb_sommets * nb_sommets red↪ , sizeof(int)) pour allouer tout le bloc d'un coup.`
- `liberer_graphe(g) ne fait qu'un free(g->adjacence) puis un free(g).`

`Le principal changement a été d'adapter toutes les fonctions d'accès à la matrice (ajouter_arete, afficher_matrice, peut_colorer et le calcul des degrés) pour utiliser la formule d'accès linéarisé : g->adjacence[i * n + j] (où n est le nb_sommets).`

4 Conclusion

`Le fichier main.c m'a permis de tester les deux implémentations en changeant simplement le fichier .h inclus. L'algorithme fonctionne correctement dans les deux cas et a pu colorer le graphe "maison" donné en exemple avec 3 couleurs, ce qui est le nombre chromatique attendu.`