Rapport de Projet : Coloration de Graphes Par Tons

Moucer Bahdja, Abib Alicia, Guenoun Dalil

May 11, 2025

1 Introduction

1.1 Contexte

La coloration de graphes est un problème classique en informatique et en mathématiques discrètes. Il consiste à contribuer une couleur à chacun de ses sommets de manière que deux sommets reliés par une arête soient de couleur différentes. On cherche souvent à utiliser le nombre minimal de couleurs, appelé **le nombre chromatique**. Dans ce projet, nous nous intéressons à une variante particulière où la coloration doit satisfaire des conditions spécifiques liées aux distances entre les sommets.

1.2 Problématique

Le problème consiste à colorier un graphe non orienté de telle sorte que pour toute paire de sommets (x, y), le nombre de couleurs communes à leurs voisinages respectifs soit inférieur ou égal à la distance entre x et y. Formellement, $|\phi(x) \cap \phi(y)| \leq d(x, y)$ où $\phi(x)$ représente l'ensemble des couleurs du voisinage de x.

2 Travail réalisé

2.1 Structure de données

Nous avons implémenté une structure de graphe utilisant:

- Une matrice d'adjacence `adj` pour représenter les arêtes
- Une matrice de distances `dist` stockant les distances entre toutes les paires de sommets
- Un tableau de listes chaînées `coul` pour représenter les couleurs attribuées à chaque sommet

2.2 Choix des Listes Chaînées

Dans notre implémentation, nous avons principalement utilisé des **listes chaînées** pour représenter les ensembles de couleurs attribuées à chaque sommet. Ce choix à plusieurs avantage pour notre problème :

- **Flexibilité** : Les listes chaînées permettent une allocation dynamique de la mémoire, en fonction du nombre variable de couleurs que peut posséder un sommet.
- **Efficacité pour les opérations fréquentes** : Les principales opérations que nous effectuons sont :
  - L'ajout de nouvelles couleurs (`addfront`, `addend`)
  - La recherche d'une couleur (`lookup`)
  - Le calcul d'intersections entre ensembles de couleurs (`cardinter`)
- **Implémentation légère** : Notre structure de liste chaînée est minimaliste :

```
typedef struct List List;

struct List {
    int val;    // Valeur de la couleur
    List *next; // Pointeur vers l'élément suivant
};
```

2.3 Opérations Implémentées

Nous avons développé les fonctions de base pour manipuler ces listes :

Table 1: Fonctions principales des listes chaînées

Fonction	Description
`newitem`	Crée un nouvel élément avec une valeur donnée
`addfront`	Ajoute un élément en tête de liste
`addend`	Ajoute un élément en fin de liste
`lookup`	Recherche une valeur dans la liste
`cardinter`	Calcule le cardinal de l'intersection de deux listes
`freeall`	Libère toute la mémoire occupée par une liste

2.4 Justification du Choix

Pour notre problème de coloration, les listes chaînées sont le meilleur choix car :

- Les ensembles de couleurs évoluent dynamiquement pendant l'exécution des algorithmes
- Nous avons besoin fréquemment de calculer des intersections entre les différents ensembles de couleurs
- Pas de gaspillage de la mémoire car elle est allouée au fur et à mesure des besoins
- L'ordre des éléments dans la liste n'a pas d'importance pour notre problème

2.5 Algorithmes implémentés

2.5.1 Initialisation du graphe

- Génération de graphes aléatoires (`gengraph`)
- Génération de cycles (`gengraphcycle`)
- Calcul des distances via BFS (`bfs`, `initdist`)

2.5.2 Algorithmes de coloration

2.6 Algorithme Glouton

2.6.1 Principe

L'algorithme glouton parcourt les sommets dans un ordre arbitraire et effectue pour chacun :

1. Recherche de la plus petite couleur c valide
2. Vérification des contraintes de voisinage
3. Attribution de la couleur si possible

2.6.2 Implémentation

```
int colorsom(List **coul, int x) {
    int c;
    for(c = 1; convient(coul, x, c) == 0; c++)
        ;
    coul[x] = addend(coul[x], newitem(c));
    return c;
}
```

2.6.3 Complexité

- Complexité temporelle : $O(n^2 \times k)$ où k est le nombre chromatique
- Complexité spatiale : $O(n)$

2.6.4 Avantages/Limites

+ Simple à implémenter
+ Rapide pour des petits graphes
- Ne donne pas toujours la coloration optimale
- Sensible à l'ordre de parcours des sommets

2.7 Algorithme DSATUR

2.7.1 Principe

DSATUR est une heuristique qui choisit à chaque étape le sommet avec le nombre maximal de couleurs différentes dans son voisinage (saturation).

2.7.2 Implémentation clé

```
int dsatur(List **coul, int b) {
    int dsat[n], deg[n];
    initdeg(deg);
    memmove(dsat, deg, n * sizeof(int));
    actuvois(dsat, coul);
    s = maxdsat(dsat, deg);
    // ...
}
```

2.7.3 Complexité

- Complexité temporelle : $O(n^3)$ dans le pire cas
- Complexité spatiale : $O(n^2)$

2.7.4 Avantages/Limites

+ Donne généralement de meilleurs résultats que glouton
+ Plus intelligent dans l'ordre de sélection des sommets
- Plus coûteux en calcul que glouton
- Nécessite des structures auxiliaires

2.8 Algorithme Exact (Backtracking)

2.8.1 Principe

L'algorithme exact utilise le backtracking pour explorer systématiquement toutes les colorations possibles jusqu'à trouver la solution optimale.

2.8.2 Implémentation clé

```
void colorback(List **coul, int x, int k, int *stop) {
    if(x == n) *stop = 1;
    else {
        for(c = 1; c <= k; c++) {
            if(convient(coul, x, c)) {
                coul[x] = addfront(coul[x], newitem(c));
                colorback(coul, x+1, k, stop);
                if(*stop) return;
                coul[x] = delitem(coul[x], 0);
            }
        }
    }
}
```

2.8.3 Complexité

- Complexité temporelle : $O(k^n)$ (exponentielle)
- Complexité spatiale : $O(n)$

2.8.4 Avantages/Limites

+ Garantit de trouver la solution optimale
+ Permet de déterminer le nombre chromatique
- Impraticable pour n $>$ 15-20 sommets
- Complexité prohibitive

3 Performances

Table 2: Temps d'exécution (en ms) pour différents algorithmes

Taille du graphe	Glouton	Backtracking	DSATUR
5 sommets	2	15	5
7 sommets	3	120	8
10 sommets	5	-	12

4 Conclusion

Ce projet nous a permis de comparer les trois algorithmes de coloration de graphe par tons. Les trois résultats montrent bien le compromis classique en algorithmique : plus une solution est précise plus sont coup computationnel est élevé. L'algorithme glouton est efficace pour les solutions rapides, DSATUR offre une bonne équilibre qualité/performance, tandis que la méthode exacte marlgé son optimalité, sa complexité est prohibitive.

5 Bibliographie

- Les cours de graphes sur Plubel.
- The Practice of Programming de Brian W. Kernighan et Rob Pike .