Compte Rendu TP3
Analyse de Code, Listes d'Adjacence et Performance

Rayane KACHBI

9 novembre 2025

1 Introduction

L'objectif de ce TP était de consolider nos connaissances sur les graphes et les structures de données. Le travail s'est articulé autour de trois axes majeurs :

1. L'analyse et la correction de codes de coloration (C et Python) fournis, qui avaient été générés par IA et contenaient des bugs.
2. L'implémentation de structures de données plus générales : un arbre n-aire (via la méthode "Fils Gauche, Frère Droit") et un graphe par listes d'adjacence, en adaptant le code de nos TP précédents.
3. Une analyse de performance de notre algorithme Welsh-Powell (sur matrice $N \times N$) pour vérifier sa complexité en pratique.

2 Partie 1 : Analyse et Correction des Codes Fournis

L'énoncé indiquait que les codes C et Python fournis pour Welsh-Powell ne fonctionnaient pas correctement. Après analyse, j'ai identifié deux problèmes critiques.

2.1 Le bug d'initialisation (Code C)

Le code C initialisait le tableau de couleurs ainsi :

```
blueint color[MAX_VERTICES] = {-1};
```

En C, cette syntaxe n'initialise **que le premier élément** (‘color[0]‘) à -1. Tous les autres (‘color[1]‘, ‘color[2]‘, ...) sont initialisés à 0. L'algorithme croyait donc que la plupart des sommets étaient déjà colorés (couleur 0), ce qui le faisait échouer.

Correction : J'ai remplacé cette ligne par un ‘calloc‘ pour m'assurer que toute la mémoire du tableau ‘couleurs‘ soit bien initialisée à 0 (notre valeur pour "non coloré").

2.2 Le bug algorithmique (Code C et Python)

Le problème le plus important était une mauvaise implémentation de la logique de Welsh-Powell, présente à la fois dans le code C et le script Python. L'algorithme défaillant incrémentait la couleur (‘current$_c$olor++‘)$\grave{a}$ $l'int\acute{e}rieurdelaboucledetridessommets, dsqu'ilrencontraitunconflit.$

Correction : J'ai réimplémenté la logique correcte (celle de mon TP2) :

1. On fixe une ‘couleur$_a$ctuelle‘$(ex : 1)$.
1. On parcourt **toute** la liste des sommets triés. On assigne la ‘couleur$_a$ctuelle‘$touslessommetsquilepeuvent$
   $-$ $direquin'ontpasdevoisindjdecettecouleur$).

1. **Seulement après** cette passe complète, on incrémente ‘couleur$_a$*ctuelle*‘(2)*etonrecommenceunenouvelle*

La version corrigée (‘chatgpt$_c$*orrige.c*‘)*donneunnombredecouleurscorrect.*

3 Partie 2 : Nouvelles Structures de Données

3.1 Arbre N-aire (Fils Gauche, Frère Droit)

Pour transformer un arbre binaire en arbre quelconque (n-aire), j'ai suivi la méthode "Fils Gauche, Frère Droit" vue dans le `TP0.pdf`. Chaque nœud conserve deux pointeurs, mais leur signification change :

```
gray1 bluetypedef bluestruct NoeudNAire {
gray2     bluechar val;
gray3     bluestruct NoeudNAire* premierEnfant; green!60!black//green!60!black
              green!60!blackPointeurgreen!60!black green!60!blackversgreen!60!black
              green!60!blacklegreen!60!black green!60!black1green!60!blacker
              green!60!black green!60!blackenfant
gray4     bluestruct NoeudNAire* frereSuivant; green!60!black//green!60!black
              green!60!blackPointeurgreen!60!black green!60!blackversgreen!60!black
              green!60!blacksongreen!60!black green!60!blackfrere
gray5 } NoeudNAire;
```

La fonction ‘ajouterEnfant‘ gère l'ajout : si le parent n'a pas d'enfant, le nouveau nœud devient ‘premierEnfant‘. Sinon, on parcourt la liste des ‘frereSuivant‘ jusqu'au dernier enfant, et on l'ajoute à la fin.

3.2 Graphe par Listes d'Adjacence

Pour modéliser la carte d'Europe, j'ai abandonné la matrice d'adjacence pour une structure plus flexible, inspirée de la ‘liste.c‘ du TP0 et du `TP0.pdf`.

- J'ai un `Graphe` qui contient un tableau de `Sommet`.
- Chaque `Sommet` (un pays) contient son nom, son ID, et un pointeur `listeVoisins`.
- `listeVoisins` est la tête d'une liste chaînée (de type `CelluleVoisin`) où chaque cellule pointe vers un autre `Sommet`.

L'algorithme Welsh-Powell a été adapté pour cette structure : le tri se fait sur le tableau de sommets (en utilisant le `degre` stocké dans le sommet), et la fonction ‘peut$_c$*olorer$_l$iste*‘*parcourtla***liste chaînée***d*

4 Partie 3 : Visualisation et Analyse de Performance

4.1 Visualisation Python

En suivant la suggestion de l'énoncé, j'ai utilisé Python, ‘networkx‘ et ‘matplotlib‘ pour visualiser le résultat. J'ai créé un script ‘coloration$_c$*arte.py*‘*qui* :

Définit le graphe des 11 pays avec les bonnes arêtes.

Implémente la **logique corrigée** de Welsh-Powell (vue en partie 1).

Applique l'algorithme et récupère la liste des couleurs.

Utilise ‘networkx.draw‘ pour afficher le graphe, en assignant une couleur différente à chaque nœud selon le résultat de l'algorithme.

Le résultat (Figure 1) montre bien la carte colorée avec 4 couleurs. On peut vérifier visuellement qu'aucun pays adjacent n'a la même couleur (par exemple, "Fr" (violet) est bien entouré de couleurs différentes comme "Es" (cyan) ou "An" (beige)).

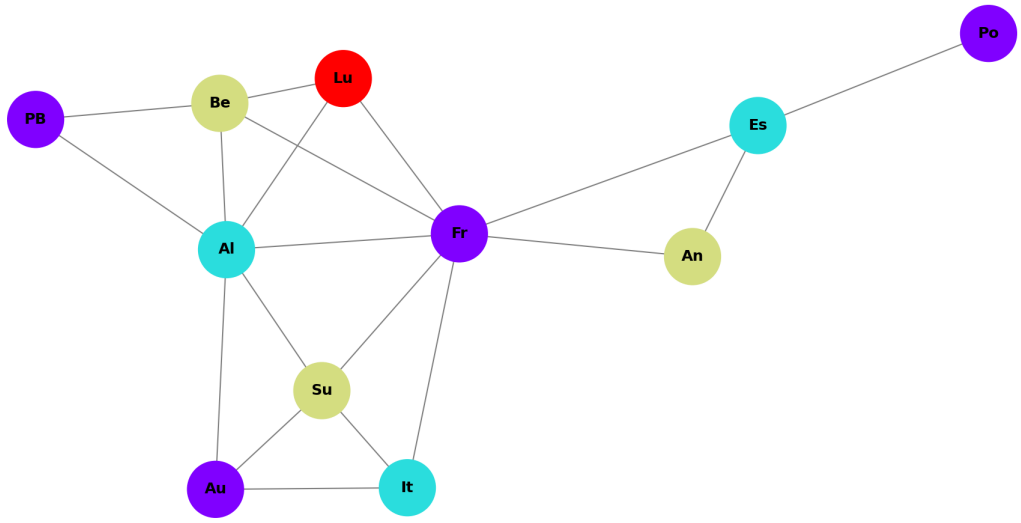

FIGURE 1 – Graphe des pays européens coloré avec l'algorithme corrigé (Python).

4.2 Analyse de Performance (Code C du TP2)

J'ai créé un programme `main_performance.c` qui génère des graphes aléatoires (densité 10%) de tailles N croissantes et mesure le temps d'exécution de l'algorithme Welsh-Powell (version matrice $N \times N$ du TP2).

En traçant les résultats (Temps en Y, N en X) sur une échelle linéaire (Figure 2), on observe une croissance non-linéaire rapide, typique d'une complexité polynomiale.

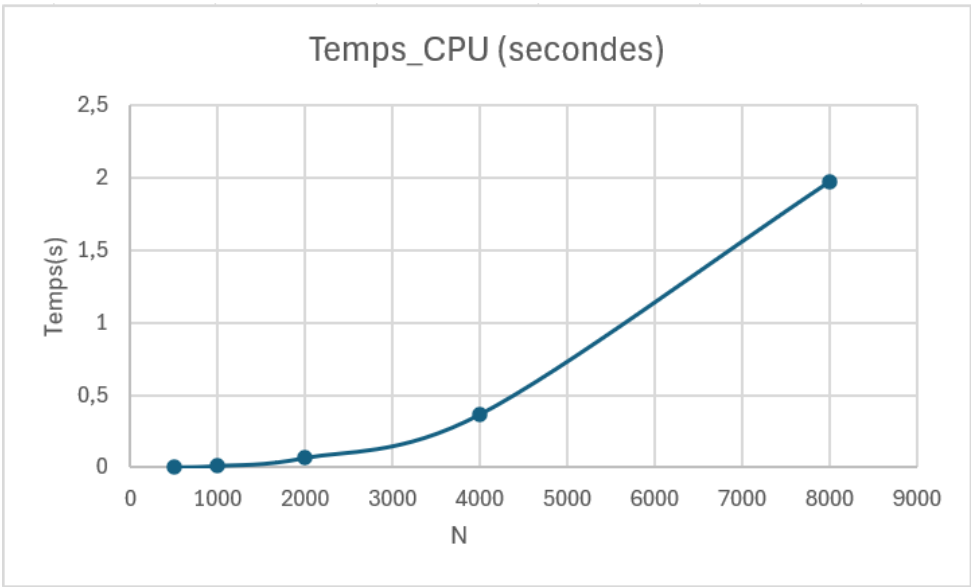

FIGURE 2 – Temps CPU en fonction de N (échelles linéaires).

Pour vérifier l'exposant, j'ai (comme demandé) passé les deux axes en **échelle logarithmique** (Figure 3). Les points s'alignent parfaitement sur une droite, ce qui confirme une

relation de type $Temps \approx N^k$.

En ajoutant une courbe de tendance "Puissance", Excel a calculé l'équation :

$$y \approx 10^{-9} \times x^{2.3547}$$

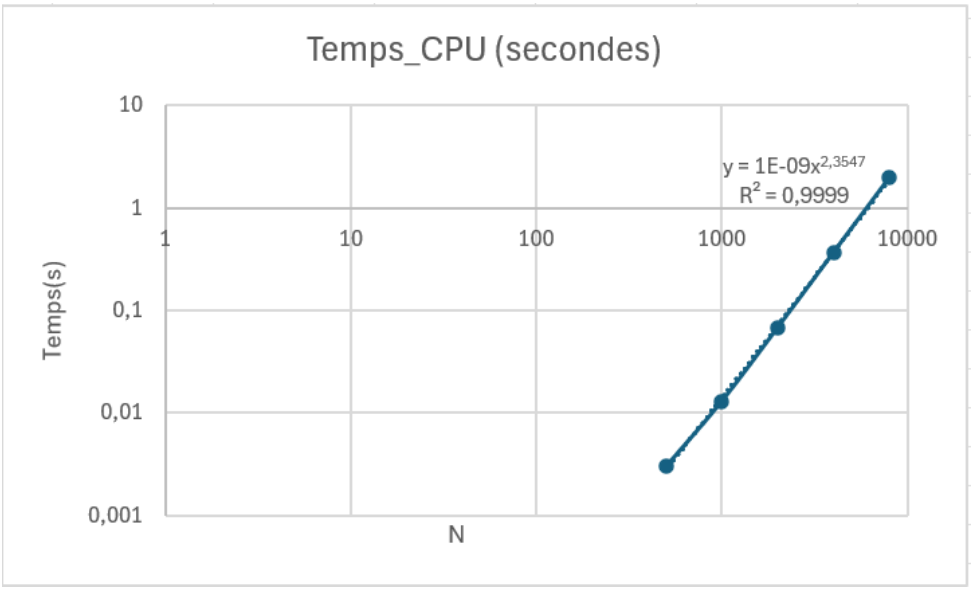

FIGURE 3 – Temps CPU en fonction de N (échelles log-log) avec courbe de tendance.

L'analyse théorique de l'algorithme est $\mathbf{O(K \times N^2)}$, où K est le nombre de couleurs et N le nombre de sommets. Le pire des cas ($K \approx N$) est donc $\mathbf{O(N^3)}$. Mon résultat expérimental ($O(N^{2.35})$) est parfaitement cohérent : il montre que pour un graphe aléatoire peu dense, K (le nombre de couleurs) croît bien plus lentement que N, et l'algorithme est en pratique plus rapide que son "pire des cas" $O(N^3)$.