Compte Rendu Global des TP

Implémentation de Structures de Données et Algorithmes

Novembre 2025

Rédigé par : Rayane KACHBI
Numéro étudiant : 22307872
TD Groupe 2

Table des Matières

Liste des Figures et des Listings

Introduction

Ce document retrace l'ensemble des travaux pratiques (TP) réalisés, dont l'objectif était de concevoir, d'implémenter et d'analyser diverses structures de données et algorithmes.

Partant de structures fondamentales (listes doublement chaînées), nous avons progressé vers des structures plus complexes (arbres binaires, arbres n-aires) pour aboutir à diverses implémentations de graphes. L'accent a été mis sur l'algorithme de coloration de Welsh-Powell, son application, l'analyse de ses performances et la comparaison de différentes stratégies de représentation des données (matrices vs listes).

Le projet final a été réorganisé en dossiers (TP0 à TP3), chacun contenant des sous-projets indépendants et fonctionnels pour une meilleure clarté.

1. TP0 : Fondations - Listes Doublement Chaînées

Ce TP fait suite aux travaux pratiques sur les listes doublement chaînées (TP11) et les arbres binaires (TP12) de l'année passée (L2).

L'objectif est d'explorer conceptuellement comment étendre les structures de données implémentées pour représenter des structures plus complexes : l'arbre générique (ou n-aire) et le graphe quelconque.

1.1. Extension vers un Arbre Générique

Problématique : La structure de nœud implémentée dans le TP12 (`arbre.c`) est spécifiquement binaire. Chaque nœud possède un nombre fixe de pointeurs enfants : un fils gauche et un fils droit.

```c
typedef struct{
    char val;
    int num_crea;
    struct noeud *gauche;
    struct noeud *droit;
} noeud;
```

Listing 1: Structure du nœud binaire (TP12).

Cette structure ne permet pas à un nœud de posséder trois enfants ou plus, ce qui est la définition d'un arbre générique (n-aire).

Solution : Pour permettre à un nœud d'avoir un nombre variable d'enfants sans allouer une liste ou un tableau pour chaque nœud, la solution la plus courante est la représentation "Fils Gauche, Frère Droit".

La structure du nœud est modifiée comme suit :

- Un pointeur (que l'on peut nommer `premierEnfant`) pointe vers le premier enfant du nœud.
- Un second pointeur (que l'on peut nommer `frereSuivant`) pointe vers le "frère" suivant du nœud, c'est-à-dire le prochain enfant du même parent.

Les enfants d'un nœud forment ainsi une simple liste chaînée, reliée par les pointeurs `frereSuivant`.

```c
typedef struct noeud_generique {
    char val;
    int num_crea;

    // Pointeur vers le premier enfant
    struct noeud_generique *premierEnfant;

    // Pointeur vers son frere suivant
    struct noeud_generique *frereSuivant;

} noeud_generique;
```

Listing 2: Structure d'un nœud pour arbre générique.

Cette approche conserve une taille de structure constante par nœud (toujours deux pointeurs) tout en offrant la flexibilité de représenter n'importe quel nombre d'enfants.

1.2 Extension vers un Graphe Quelconque

Problématique : Un graphe est une structure encore plus générale. Il n'y a plus de notion de hiérarchie (parent/enfant) ni de racine unique. Les relations peuvent être multidirectionnelles et des cycles (revenir à un nœud déjà visité) sont possibles.

Solution : Liste d'Adjacence La représentation par matrice d'adjacence est une solution, mais elle est très coûteuse en mémoire si le graphe est "creux" (peu d'arêtes).

Une solution plus flexible, qui réutilise les concepts de nos TPs, est la **liste d'adjacence**. Cette méthode combine l'idée d'un tableau (pour les sommets) et de listes chaînées (pour les arêtes/voisins), s'inspirant directement de notre TP11.

Le concept est le suivant :

1. On définit une structure pour un **Sommet** (un nœud du graphe).
2. On définit une structure pour une **Cellule** de liste chaînée, qui servira à lister les voisins.
3. Chaque structure `Sommet` contiendra un pointeur vers la tête d'une liste chaînée de ses "voisins" (les sommets auxquels il est connecté).

```c
// Déclaration anticipée pour permettre la référence croisée
typedef struct Sommet Sommet;

// Structure pour la cellule de la liste des voisins
// (Inspirée du TP11)
typedef struct CelluleVoisin {
    Sommet *voisin; // Pointeur vers le sommet voisin
    struct CelluleVoisin *suivant;
} CelluleVoisin;

// Structure pour le Sommet (le noeud du graphe)
// (Inspirée du TP12)
struct Sommet {
    char val; // Donnée du sommet
    int num;  // Identifiant

    // Tete de la liste chainee de ses voisins
    CelluleVoisin *listeVoisins;
};
```

Listing 3: Structures pour une liste d'adjacence.

Pour créer un graphe, on gère un tableau ou une liste de tous les `Sommet`s. Pour ajouter une arête (un lien) du `Sommet A` vers le `Sommet B`, il suffit de créer une `CelluleVoisin` pointant vers `B` et de l'insérer (en utilisant la logique de `InsererCellule` du TP11) dans la `listeVoisins` du `Sommet A`.

Cette méthode est efficace en mémoire pour les graphes peu denses et réutilise les principes de listes chaînées vus en TP.

2. TP1 : Graphes et Welsh-Powell (Représentation Matricielle)

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 standard (`int **adjacence`).
- Une extension utilisant une matrice d'adjacence linéarisée (`int *adjacence`).

2.1. Matrice d'Adjacence Standard (int **)

a. 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 :

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

Listing 4: Structure Graphe (Standard).

b. 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.

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

c. 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.

- **Calcul des degrés :** J'ai parcouru la matrice d'adjacence pour calculer le degré de chaque sommet (en comptant les `1` sur chaque ligne).
- **Tri des sommets :** J'ai utilisé `qsort` avec une fonction de comparaison personnalisée (`comparer_sommets`) pour trier les sommets par ordre de degré décroissant.
- **Coloration :** J'ai initialisé un tableau `couleurs` à 0 (avec `calloc`). J'ai ensuite itéré, en commençant par la `couleur_actuelle = 1`. Dans une boucle imbriquée, je parcourais ma liste de sommets triés.
- **Vérification :** Pour chaque sommet non coloré, j'appelais la fonction `peut_colorer(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.

- **Itération :** S'il restait des sommets non colorés, j'incrémentais `couleur_actuelle` et je recommençais.

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

2.2. Implémentation Linéarisée (Matrice N * 1)

a. Structure et Fichiers

J'ai créé de nouveaux fichiers `graphelineaire.h` et `graphelineaire.c`. La structure `Graphe` est modifiée pour n'avoir qu'un simple pointeur :

```c
typedef struct {
  int nb_sommets;
  int *adjacence; // La matrice (N*N) en un seul bloc
} Graphe;
```

Listing 5: Structure Graphe (Linéairisée).

b. 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, 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`).

Le fichier main 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.

3. TP2 : Applications Pratiques

Le TP2 a permis d'appliquer et d'étendre les codes précédents à des cas concrets.

3.1 Graphes - Welsh-Powell (Suite)

Cette première partie du TP consistait à étendre le code du TP1 sur la coloration de graphes.
L'objectif était d'ajouter une fonction pour charger un graphe depuis un fichier et d'appliquer l'algorithme sur un cas concret : la carte de 11 pays européens.

a. Chargement du Graphe

J'ai commencé par modifier graphe.h pour y ajouter les prototypes de deux nouvelles fonctions :

```c
// Charge un graphe depuis un fichier texte (ou stdin).
// La première ligne du fichier doit contenir le nombre de sommets.
// Les lignes suivantes contiennent la matrice d'adjacence (N lignes de N entiers).
Graphe* chargeGraphe(const char* nom_fichier);

// Affiche l'ordre de marquage (le tri des sommets par degré)
// et le résultat de la coloration.
void afficher_ordre_marquage_et_resultat(Graphe* g, int* couleurs, const char**
noms_sommets);
```

Listing 6: Prototypes des nouvelles fonctions du TP2 (graphe.h).

Dans `graphe.c`, la fonction `chargeGraphe` gère deux cas :

- Si `nom_fichier` est `NULL`, elle utilise `stdin` pour lire l'ordre du graphe puis la matrice ligne par ligne (j'ai ajouté un `printf` pour guider la saisie de chaque ligne).
- Si un nom de fichier est donné, elle ouvre ce fichier.

Dans les deux cas, elle lit d'abord le nombre de sommets, crée le graphe avec `creer_graphe`, puis remplit la matrice d'adjacence.

b. Application: Carte de l'Europe

J'ai modélisé le problème des 11 pays sous forme d'un fichier `carte_europe.txt`. Les sommets représentent les pays, et une arête (valeur `1`) existe si deux pays ont une frontière commune.

J'ai défini l'ordre suivant :
- 0: Fr (France)
- 1: Es (Espagne)
- 2: Po (Portugal)
- 3: An (Andorre)
- 4: It (Italie)
- 5: Au (Autriche)
- 6: Su (Suisse)
- 7: Al (Allemagne)
- 8: Lu (Luxembourg)
- 9: Be (Belgique)
- 10: PB (Pays-Bas)

Mon fichier `carte_europe.txt` contient donc (la première ligne est l'ordre, suivie de la matrice) :

```
11
0 1 0 1 1 0 1 1 1 1 0
1 0 1 1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0
... (et ainsi de suite pour les 11 lignes)
```

Mon `main_graphe.c` charge ce fichier, lance `coloration_welsh_powell`, puis appelle `afficher_ordre_marque` en lui passant un tableau de noms de pays pour un affichage clair.

c. Résultats et Analyse

L'exécution a produit les résultats attendus :

- **Ordre de marquage :** L'affichage montre bien le tri des sommets par degré décroissant. La France (7 voisins) et l'Allemagne (6 voisins) sont logiquement traitées en premier.
- **Coloration :** L'algorithme a utilisé **4 couleurs**.
  1. Couleur 1 : France (Fr) Portugal (Po) Autriche (Au) Pays-Bas (PB)
  2. Couleur 2 : Espagne (Es) Italie (It) Allemagne (Al)
  3. Couleur 3 : Andorre (An) Suisse (Su) Belgique (Be)
  4. Couleur 4 : Luxembourg (Lu)

c.1. Analyse de Complexité

La complexité de mon implémentation de Welsh-Powell est dominée par trois étapes :

1. **Calcul des degrés :** Je dois parcourir toute la matrice $N \times N$. Complexité : $O(N^2)$.
2. **Tri des sommets :** Le `qsort` sur N éléments a une complexité de $O(N \log N)$.

3. **Boucle de coloration :** Dans le pire des cas (K couleurs), je reparcours ma liste de N sommets. Pour chaque sommet, la fonction `peut_colorer` vérifie ses N voisins potentiels. Cela donne $O(K * N^2)$. Si K est proche de N (graphe complet), la complexité est $O(N^3)$.

La complexité globale de mon algorithme est donc en $O(N^2 + N \log N + N^3)$, ce qui se simplifie en $O(N^3)$.

c.2. Nombre Chromatique

Il est important de noter que Welsh-Powell est une **heuristique gloutonne**. Elle est **polynomiale** (rapide), mais ne garantit pas de trouver le nombre chromatique exact (le minimum de couleurs), qui est un problème **NP-difficile**.
Pour notre carte, 4 est une bonne coloration, mais on ne peut pas affirmer que c'est le nombre chromatique sans une analyse plus poussée.

3.2. Parcours d'Arbres Binaires

La seconde partie du TP demandait d'implémenter les parcours d'arbres binaires (**Préfixe**, **Infixe**, **Postfixe**) en se basant sur la structure du TP0.

a. Structure et Création

J'ai créé les fichiers `arbre.h` et `arbre.c`.
Pour tester les parcours, j'ai créé une fonction `creerArbreSynthetique()` qui construit l'arbre suivant :

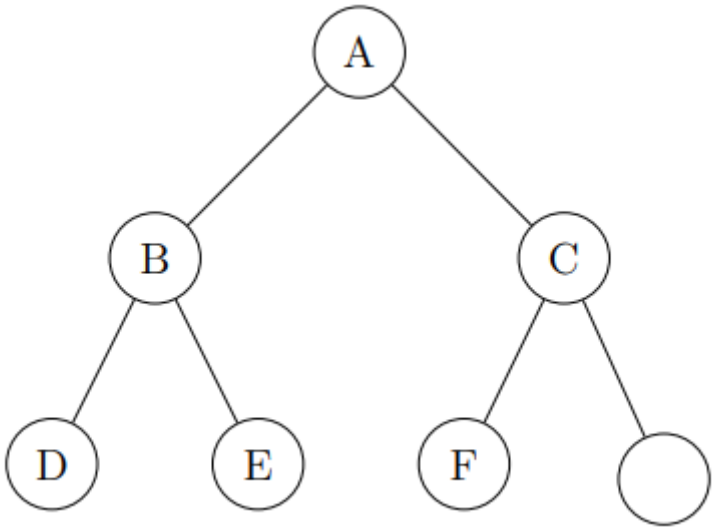

Figure 1: Arbre binaire généré par `creerArbreSynthetique()`

b. Implémentation des Parcours

J'ai implémenté les trois parcours de manière récursive.

b.1. Parcours Préfixe

```c
//Ordre: Racine → Gauche → Droit
void parcoursPrefixe(Noeud* racine) {
    // Cas de base de la récursion
    if (racine == NULL) {
        return;
    }

    // Visiter la Racine
    printf("%c ", racine->val);

    // Visiter le sous-arbre Gauche
    parcoursPrefixe(racine->gauche);

    // Visiter le sous-arbre Droit
    parcoursPrefixe(racine->droit);
}
```

Listing 7: Résultat : A B D E C F.

b.2. Parcours Infixe

```c
//Ordre : Gauche → Racine → Droit
void parcoursInfixe(Noeud* racine) {
    // Cas de base de la récursion
    if (racine == NULL) {
        return;
    }

    // Visiter le sous-arbre Gauche
    parcoursInfixe(racine->gauche);

    // Visiter la Racine
    printf("%c ", racine->val);

    // Visiter le sous-arbre Droit
    parcoursInfixe(racine->droit);
}
```

Listing 8: Résultat : D B E A F C.

b.3. Parcours Postfixe

```c
//Ordre : Gauche → Droit → Racine
void parcoursPostfixe(Noeud* racine) {
    // Cas de base de la récursion
    if (racine == NULL) {
        return;
    }

    // Visiter le sous-arbre Gauche
    parcoursPostfixe(racine->gauche);

    // Visiter le sous-arbre Droit
    parcoursPostfixe(racine->droit);

    // Visiter la Racine
    printf("%c ", racine->val);
}
```

Listing 9: Résultat : D E B F C A.

4. TP3 : Structures Avancées, Correction et Analyse de performance

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.

4.1. 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:

a. Le bug d'initialisation (Code C)

Le code C initialisait le tableau de couleurs ainsi :

```c
int 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` par défaut.

L'algorithme croyait donc que la plupart des sommets étaient déjà colorés (couleur `0`), ce qui le faisait échouer.

Correction : (`chatgpt_corrige.c`) 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").

b. 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_color++`) **à l'intérieur** de la boucle de tri des sommets, dès qu'il rencontrait un conflit.

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

1. On fixe une `couleur_actuelle` (ex : 1).
2. On parcourt **toute** la liste des sommets triés. On assigne la `couleur_actuelle` à tous les sommets qui le peuvent (c'est-à-dire qui n'ont pas de voisin déjà de cette couleur).
3. Seulement après cette passe complète, on incrémente `couleur_actuelle` (ex: 2) et on recommence une nouvelle passe avec les sommets restants.

La version corrigée (`chatgpt_corrige.c`) donne un nombre de couleurs correct.

4.2. Nouvelles Structures de Données

a. 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. Chaque nœud conserve deux pointeurs, mais leur signification change :

```c
typedef struct NoeudNAire {
  char val;
  struct NoeudNAire *premierEnfant; // Pointeur vers le 1er enfant
  struct NoeudNAire *frereSuivant;  // Pointeur vers son frère
} NoeudNAire;
```

Listing 10: Structure C pour un nœud d'arbre n-aire.

Implémentation : La fonction `ajouterEnfant` gère cette logique : si `premierEnfant` est nul, le nouveau nœud y est placé. Sinon, la fonction parcourt la liste chaînée des frères en suivant les pointeurs `frereSuivant` jusqu'à trouver le dernier enfant, et s'y attache.

b. Graphe par Listes d'Adjacence

Pour modéliser la carte d'Europe de manière plus flexible et efficace en mémoire (pour les graphes **"creux"**), la matrice d'adjacence a été remplacée par des **listes d'adjacence**.

- **Structure :** Le `Graphe` contient un tableau de `Sommet` (`tab_sommets`). Chaque `Sommet` contient son nom et un pointeur `listeVoisins`, qui est la tête d'une liste chaînée de `CelluleVoisin`.
- **Adaptation de Welsh-Powell :** Le tri `qsort` s'effectue sur un tableau de pointeurs de sommets (`Sommet**`).
- **Efficacité :** La fonction `peut_colorer_liste` est bien plus efficace : au lieu de scanner une ligne de N éléments dans la matrice, elle ne parcourt que la liste chaînée des D voisins directs du sommet (où D est le degré du sommet, $D < N$ pour un graphe creux).

4.3. Visualisation et Analyse de Performance

En suivant la suggestion de l'énoncé, cette partie a utilisé Python pour la visualisation et le code C du TP2 pour l'analyse de performance.

a. Visualisation Python

Le script `coloration_carte.py` utilise `networkx` et `matplotlib`. Il applique l'algorithme de `Welsh-Powell` (version Python corrigée) et utilise `networkx.draw` pour afficher le graphe. Le résultat (voir "Figure 2") montre la carte colorée avec 4 couleurs.

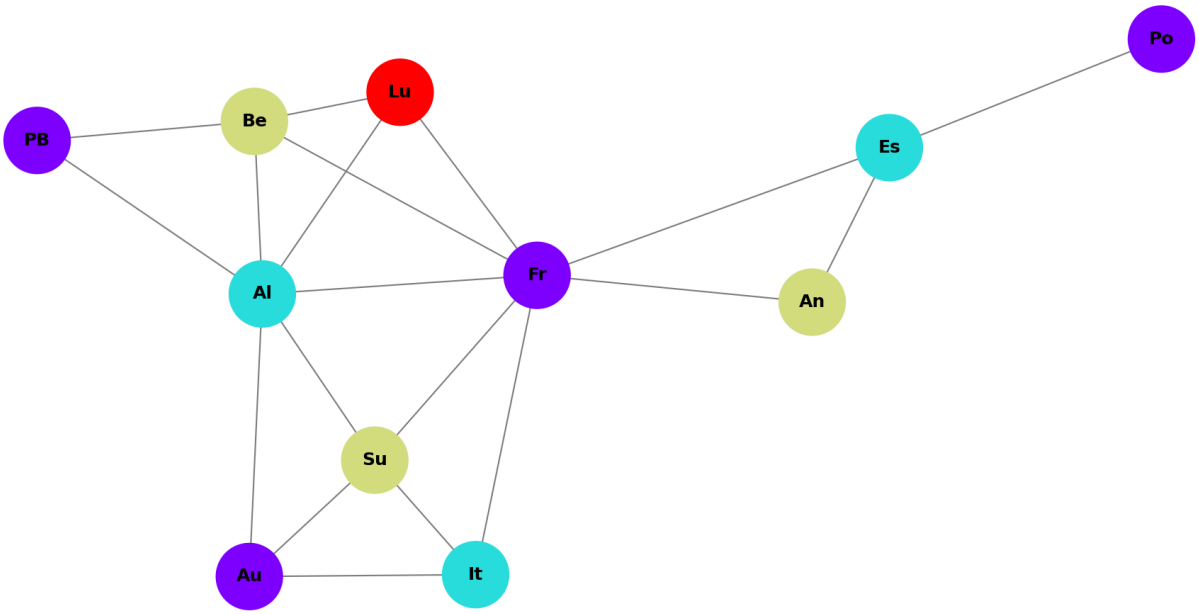

Figure 2: Graphe des pays européens coloré avec l'algorithme corrigé (Python).

b. 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 (voir "Figure 3"), on observe une croissance **non-linéaire rapide**, typique d'une complexité **polynomiale**.

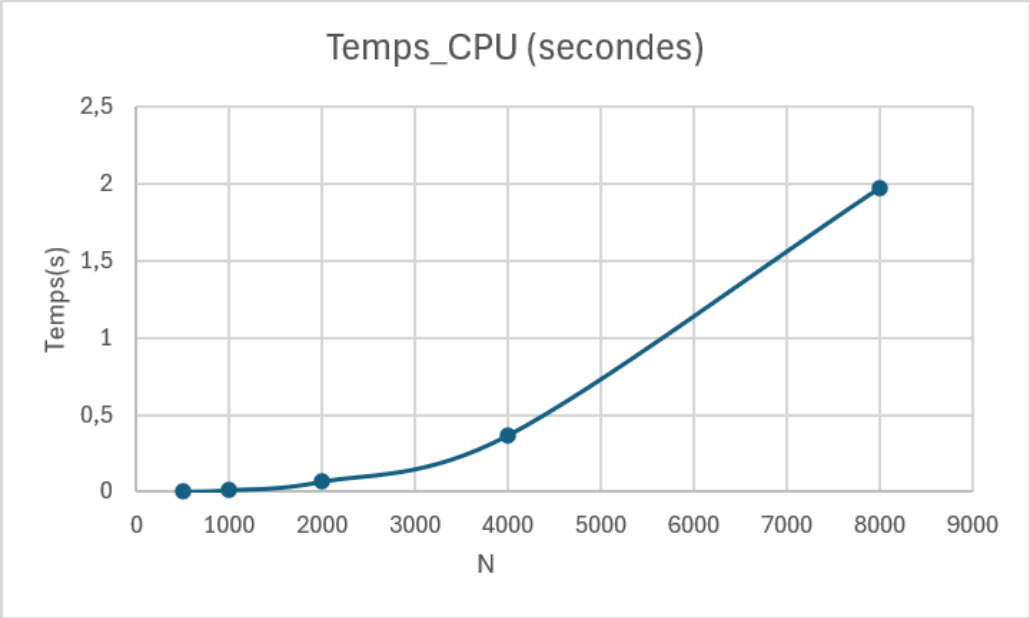

Figure 3: Temps CPU en fonction de N (échelles linéaires).

Pour vérifier l'exposant, j'ai passé les deux axes en **échelle logarithmique** (voir Figure 4). Les points s'alignent parfaitement sur une droite, ce qui confirme une relation de type $T \approx N^k$.

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

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

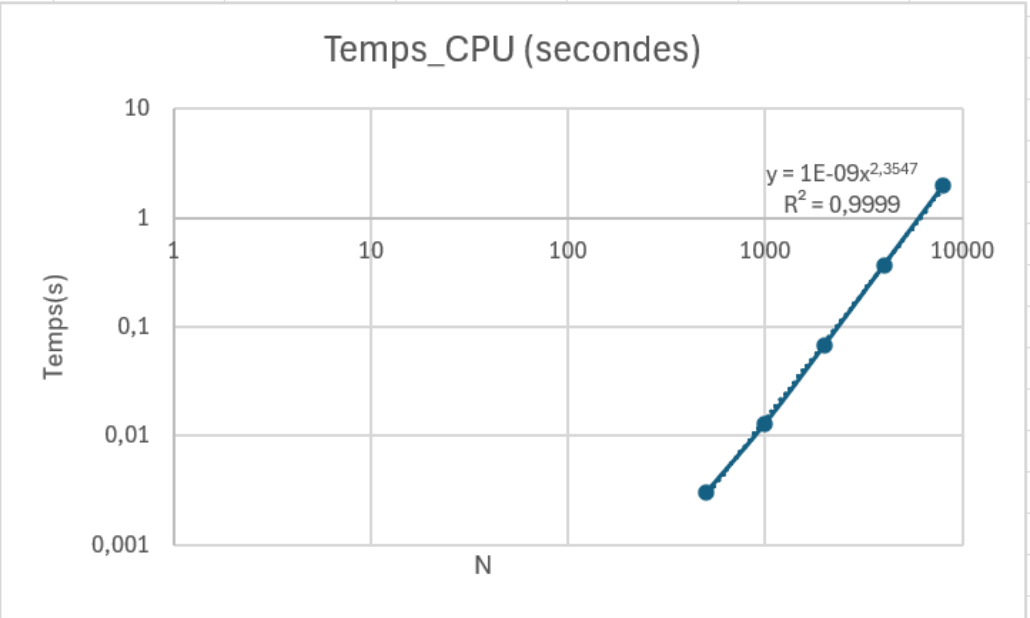

Figure 4: Temps CPU en fonction de N (échelles log-log) avec courbe de tendance.

L'analyse théorique de l'algorithme est $O(K \times N^2)$, où K est le nombre de couleurs et N le nombre de sommets. Le pire des cas ($K \approx N$) est donc $O(N^3)$.

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.

L'algorithme est en pratique **plus rapide que son "pire des cas" $O(N^3)$**.

Conclusion

Ce projet a démontré l'impact direct du choix des structures de données sur la performance des algorithmes.

L'implémentation de Welsh-Powell, d'abord avec des matrices puis des listes d'adjacence, a clairement illustré le compromis entre efficacité mémoire et complexité d'implémentation.

Le résultat principal est l'analyse de performance, qui a validé expérimentalement ($O(N^{2.35})$) une complexité meilleure que le pire cas théorique ($O(N^3)$) sur des graphes peu denses.

Finalement, la correction des codes IA a servi de leçon essentielle sur l'importance de la rigueur algorithmique.

Arborescence des Fichiers Source

Cette arborescence représente la structure des dossiers du projet.

```
TP A.A L3/
├── TP0/
│   ├── liste.c
│   ├── liste.h
│   ├── maListe.c
│   └── README.txt
│
├── TP1/
│   ├── README.txt
│   ├── graphe_matrice_lineaire/
│   │   ├── graphe_lineaire.c
│   │   ├── graphe_lineaire.h
│   │   └── main_graphe_lineaire.c
│   └── graphe_matrice_standard/
│       ├── graphe.c
│       ├── graphe.h
│       └── main_graphe.c
│
├── TP2/
│   ├── README.txt
│   ├── partie_arbre_binaire/
│   │   ├── arbre.c
│   │   ├── arbre.h
│   │   └── main_arbre.c
│   └── partie_graphe_welsh_powell/
│       ├── carte_europe.txt
│       ├── graphe.c
│       ├── graphe.h
│       └── main_graphe.c
│
└── TP3/
    ├── README.txt
    ├── 1_arbre_n_aire/
    │   ├── arbre_n_aire.c
    │   ├── arbre_n_aire.h
    │   └── main_arbre_n_aire.c
    ├── 2_graphe_liste_adjacence/
    │   ├── graphe_liste.c
    │   ├── graphe_liste.h
    │   └── main_graphe_liste.c
    ├── 3_correction_welsh_powell/
    │   └── chatgpt_corrige.c
    ├── 4_analyse_performance/
    │   ├── graphe.c
    │   ├── graphe.h
    │   └── main_performance.c
    └── 5_visualisation_python/
        ├── coloration_carte.py
        ├── coloration_corrigee.py
        └── visualiser_carte.py
```

Ressources

- Ce document a été composé avec l'outil de mise en page **Typst**.
- L'analyse de performance et la génération des graphiques ont été réalisées à l'aide de **Microsoft Excel**.
- L'assistant **IA Gemini** m'a aidé dans le formatage Typst ainsi qu'à la recherche d'information.