Compte rendu de la partie B du TP1

Ali Gouarab

15 Novembre 2025

Structure des dossiers et du code

Pour le TP1, j'ai choisi de structurer mon travail dans un dossier `tp1` avec trois sous-dossiers, `src`, `include` et `bin`. Le dossier `include` contient le fichier d'en-tête `graphe.h`, regroupant les structures et les prototypes pour manipuler les graphes. Le dossier `src` contient les fichiers sources, `graphe.c` et `graphe_lineaire.c` pour l'implémentation des fonctions, ainsi que `main.c` et `main_lineaire.c` contenant les fonctions `main()` pour exécuter les programmes. Enfin, le dossier `bin` est réservé aux exécutables générés après compilation. Cette organisation suit le principe de compilation séparée et permet de maintenir le code lisible, modulaire et facilement extensible.

Implémentation de l'algorithme de Welsh-Powell

Définition et principe de l'algorithme

L'algorithme de **Welsh-Powell** est utilisé pour colorer les sommets d'un graphe de manière à ce que deux sommets adjacents n'aient jamais la même couleur. La version que nous avons implémentée utilise une **matrice d'adjacence** (`int **adjacence`) pour représenter le graphe. Chaque cellule `adjacence[i][j]` vaut 1 si les sommets `i` et `j` sont connectés, et 0 sinon.

- Calculer le degré de chaque sommet.
- Trier les sommets par ordre décroissant de degré.
- Assigner les couleurs en suivant l'ordre des sommets triés, en évitant que deux voisins partagent la même couleur.

Fichier `graphe.h`

Le fichier `graphe.h` contient :

- La structure `Graphe` n'est pas définie ici mais dans `graphe.c` pour permettre la version normale avec matrice 2D ou la version linéarisée avec tableau 1D.
- Les prototypes des fonctions pour manipuler le graphe : `creerGraphe()`, `ajouterArete()`, `libererGraphe()`, `welshPowell()`, `trierSommetsParDegre()`, `couleurUtiliseeParVoisin()`, `afficherMatrice()`, `afficherDegres()`, `afficherColoration()` et `exporterDot()`.

Fichier `graphe.c`

Le fichier `graphe.c` contient la définition de la structure `Graphe` ainsi que l'implémentation de toutes les fonctions déclarées dans `graphe.h` :

- **Graphe** : structure représentant le graphe, contenant :
  - `nbSommets` : nombre de sommets,
  - `adjacence` : matrice 2D (int **) pour les arêtes,
  - `couleurs` : tableau des couleurs des sommets,
  - `degres` : tableau des degrés des sommets.

```
struct Graphe {
    int nbSommets;
    int **adjacence;
    int *couleurs;
    int *degres;
};
```

- **creerGraphe(int nbSommets)** : crée un graphe avec une matrice 2D, initialise les couleurs à -1 et les degrés à 0.

```
Graphe *g = (Graphe *)malloc(sizeof(Graphe));
g->nbSommets = nbSommets;
```

Cette étape alloue dynamiquement un objet `Graphe` et initialise le nombre de sommets.

```
g->adjacence = (int **)malloc(nbSommets * sizeof(int *));
for (int i = 0; i < nbSommets; i++)
    g->adjacence[i] = (int *)calloc(nbSommets, sizeof(int));
```

On crée un tableau de pointeurs pour représenter les lignes de la matrice et on alloue chaque ligne avec `calloc` pour initialiser toutes les cellules à zéro. Cela signifie qu'aucune arête n'existe encore.

```
g->couleurs = (int *)malloc(nbSommets * sizeof(int));
g->degres = (int *)malloc(nbSommets * sizeof(int));
```

On réserve de la mémoire pour stocker la couleur et le degré de chaque sommet.

```
for (int i = 0; i < nbSommets; i++) {
    g->couleurs[i] = -1;
    g->degres[i] = 0;
}
```

Toutes les couleurs sont initialisées à -1 pour indiquer qu'aucun sommet n'a encore été colorié et tous les degrés sont mis à zéro.

```
return g;
```

La fonction renvoie le pointeur vers le graphe prêt à être utilisé.

- **ajouterArete(Graphe *g, int sommet1, int sommet2)** : ajoute une arête entre deux sommets et met à jour leurs degrés.

```
g->adjacence[sommet1][sommet2] = 1;
g->adjacence[sommet2][sommet1] = 1;
```

Ces deux instructions inscrivent une arête entre `sommet1` et `sommet2` dans la matrice d'adjacence. Le graphe étant non orienté, la matrice doit rester symétrique, donc on met à 1 les deux positions correspondantes.

```
g->degres[sommet1]++;
g->degres[sommet2]++;
```

Les degrés des sommets concernés sont incrémentés pour refléter l'ajout de cette arête. Cette information sera utilisée plus tard pour trier les sommets dans l'algorithme de Welsh–Powell.

- **trierSommetsParDegre(Graphe *g, int *sommetsOrdre)** : trie les sommets par ordre décroissant de degré.

```
for (int i = 0; i < g->nbSommets; i++) {
    sommetsOrdre[i] = i;
}
```

Cette première boucle remplit le tableau `sommetsOrdre` avec les indices des sommets. Chaque case contient simplement son numéro, ce qui permet ensuite de trier non pas les degrés, mais l'ordre des sommets selon leurs degrés.

```
for (int i = 0; i < g->nbSommets - 1; i++) {
    for (int j = 0; j < g->nbSommets - i - 1; j++) {
        if (g->degres[sommetsOrdre[j]] < g->degres[sommetsOrdre[j + 1]]) {
            int temp = sommetsOrdre[j];
            sommetsOrdre[j] = sommetsOrdre[j + 1];
            sommetsOrdre[j + 1] = temp;
        }
    }
}
```

Ces deux boucles réalisent un tri à bulles : à chaque passage, on compare les degrés des sommets référencés par `sommetsOrdre`. Si un sommet a un degré plus faible que le suivant, leurs positions sont échangées. À la fin du tri, le tableau contient les sommets classés du plus fort degré au plus faible, ce qui servira directement à l'algorithme de Welsh–Powell.

- **couleurUtiliseeParVoisin(Graphe *g, int sommet, int couleur)** : vérifie si un sommet a un voisin utilisant déjà une couleur donnée.

```
int couleurUtiliseeParVoisin(Graphe *g, int sommet, int couleur) {
    int n = g->nbSommets;
    for (int i = 0; i < n; i++) {
        if (g->adjacence[sommet * n + i] == 1 && g->couleurs[i] ==
            couleur) {
            return 1;
        }
    }
    return 0;
}
```

La fonction parcourt tous les sommets pour vérifier lesquels sont voisins du sommet donné grâce à la matrice d'adjacence. Si un voisin possède déjà la couleur recherchée, la fonction renvoie 1 immédiatement. Sinon, après avoir vérifié tous les sommets, elle renvoie 0 : aucun voisin n'utilise cette couleur.

- **welshPowell(Graphe *g)** : applique l'algorithme de Welsh–Powell pour colorier le graphe.

```c
int *sommetsOrdre = malloc(g->nbSommets * sizeof(int));
trierSommetsParDegre(g, sommetsOrdre);

printf("Ordre des sommets par degre decroissant: ");
for (int i = 0; i < g->nbSommets; i++) {
    printf("%d(degre=%d) ", sommetsOrdre[i], g->degres[sommetsOrdre[i]]);
}
printf("\n\n");
```

On commence par trier les sommets par degré décroissant : les sommets les plus connectés seront colorés en premier. Cela réduit le nombre total de couleurs nécessaires car les sommets ayant le plus de contraintes sont traités en priorité.

```c
int couleurCourante = 0;

for (int i = 0; i < g->nbSommets; i++) {
    int sommet = sommetsOrdre[i];

    if (g->couleurs[sommet] == -1) {
        g->couleurs[sommet] = couleurCourante;
        printf("Sommet %d -> Couleur %d\n", sommet, couleurCourante);

        for (int j = i + 1; j < g->nbSommets; j++) {
            int autreSommet = sommetsOrdre[j];

            if (g->couleurs[autreSommet] == -1 &&
                !couleurUtiliseeParVoisin(g, autreSommet,
                    couleurCourante)) {
                g->couleurs[autreSommet] = couleurCourante;
                printf("Sommet %d -> Couleur %d\n", autreSommet,
                    couleurCourante);
            }
        }

        couleurCourante++;
        printf("\n");
    }
}
```

Le coloriage commence avec la première couleur (0). On attribue cette couleur au sommet de plus grand degré, puis on essaie de l'attribuer à tous les autres sommets qui ne lui sont pas adjacents. Ainsi, une même couleur est utilisée pour un maximum de sommets compatibles. Une fois impossible d'étendre davantage cette couleur, on passe à la couleur suivante. Le processus se répète jusqu'à ce que tous les sommets soient coloriés. Cette stratégie gloutonne respecte la règle fondamentale du coloriage : deux sommets adjacents ne partagent jamais la même couleur.

```c
free(sommetsOrdre);
```

Le tableau contenant l'ordre trié est libéré une fois le coloriage terminé.

- **afficherMatrice(Graphe *g)** : affiche la matrice d'adjacence du graphe.

```c
void afficherMatrice(Graphe *g) {
    printf("Matrice d'adjacence:\n");
    printf("   ");
    for (int i = 0; i < g->nbSommets; i++) {
        printf("%d ", i);
    }
    printf("\n");

    for (int i = 0; i < g->nbSommets; i++) {
        printf("%d: ", i);
        for (int j = 0; j < g->nbSommets; j++) {
            printf("%d ", g->adjacence[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

La fonction affiche la matrice d'adjacence sous forme de tableau lisible : les indices des sommets sont imprimés en en-tête, puis chaque ligne montre les connexions du sommet correspondant. Une valeur `1` indique la présence d'une arête, tandis qu'un `0` signifie l'absence de liaison.

- **afficherDegres(Graphe *g)** : affiche le degré de chaque sommet.

```c
void afficherDegres(Graphe *g) {
    printf("Degres des sommets:\n");
    for (int i = 0; i < g->nbSommets; i++) {
        printf("Sommet %d: degre = %d\n", i, g->degres[i]);
    }
    printf("\n");
}
```

La fonction parcourt tous les sommets et affiche directement leur degré, calculé lors de l'ajout des arêtes.

- **afficherColoration(Graphe *g)** : affiche les couleurs attribuées aux sommets et le nombre total de couleurs utilisées.

```c
void afficherColoration(Graphe *g) {
    printf("Resultat de la coloration:\n");

    int maxCouleur = -1;
    for (int i = 0; i < g->nbSommets; i++) {
        if (g->couleurs[i] > maxCouleur)
            maxCouleur = g->couleurs[i];
    }

    printf("Nombre de couleurs utilisees: %d\n\n", maxCouleur + 1);

    for (int c = 0; c <= maxCouleur; c++) {
        printf("Couleur %d: ", c);
        for (int i = 0; i < g->nbSommets; i++) {
            if (g->couleurs[i] == c)
                printf("%d ", i);
        }
        printf("\n");
    }
}
```

Elle recherche d'abord la couleur maximale utilisée, puis affiche la liste des sommets regroupés par couleur, ce qui permet de visualiser clairement le résultat du coloriage.

- **exporterDot(Graphe *g, const char *nomFichier)** : génère un fichier DOT compatible Graphviz pour visualiser le graphe et ses couleurs.

```c
void exporterDot(Graphe *g, const char *nomFichier) {
    FILE *f = fopen(nomFichier, "w");
    if (!f) {
        perror("Erreur lors de la creation du fichier DOT");
        return;
    }

    fprintf(f, "graph G {\n");
    fprintf(f, "    node [style=filled];\n");

    const char *couleursPalette[] =
        {"red","blue","green","yellow","pink",
         "cyan","violet","gold","coral","lime"};

    for (int i = 0; i < g->nbSommets; i++) {
        const char *c = couleursPalette[g->couleurs[i] % 10];
        fprintf(f, "    %d [fillcolor=%s, label=\"%d\\n(c%d)\"];\n",
                i, c, i, g->couleurs[i]);
    }

    for (int i = 0; i < g->nbSommets; i++) {
        for (int j = i + 1; j < g->nbSommets; j++) {
            if (g->adjacence[i][j])
                fprintf(f, "    %d -- %d;\n", i, j);
        }
    }

    fprintf(f, "}\n");
    fclose(f);
}
```

La fonction crée un fichier DOT contenant chaque sommet avec une couleur d'affichage, puis écrit toutes les arêtes sans doublons. Le fichier peut être visualisé avec Graphviz.

- **libererGraphe(Graphe *g)** : libère toute la mémoire allouée.

```c
void libererGraphe(Graphe *g) {
    for (int i = 0; i < g->nbSommets; i++)
        free(g->adjacence[i]);

    free(g->adjacence);
    free(g->couleurs);
    free(g->degres);
    free(g);
}
```

Elle libère toutes les lignes de la matrice d'adjacence, puis les autres tableaux, avant de libérer la structure elle-même pour éviter toute fuite mémoire.

Fichier `main.c`

Le fichier `main.c` contient la fonction `main()` qui utilise les fonctions de `graphe.c` pour :

- Créer et initialiser le graphe.
- Ajouter des arêtes entre les sommets.
- Appliquer l'algorithme de Welsh-Powell pour colorer le graphe.
- Afficher la matrice, les degrés et la coloration finale.
- Exporter le graphe au format DOT.

Tests et Exemples d'exécution

Pour vérifier le bon fonctionnement de l'algorithme de Welsh-Powell, nous avons utilisé la fonction `main()` avec le graphe a 5 sommets suivant :

```
Graphe *g1 = creerGraphe(5);
```

Ce bloc crée un graphe de 5 sommets, avec toutes les couleurs initialisées à -1 et les degrés à 0.

```
ajouterArete(g1, 0, 1);
ajouterArete(g1, 0, 2);
ajouterArete(g1, 0, 3);
ajouterArete(g1, 1, 2);
ajouterArete(g1, 1, 3);
ajouterArete(g1, 2, 3);
ajouterArete(g1, 2, 4);
ajouterArete(g1, 3, 4);
```

Ce bloc ajoute les arêtes du graphe. Chaque appel met à jour la matrice d'adjacence et incrémente les degrés des sommets concernés.

```
welshPowell(g1);
```

Le graphe est maintenant colorié en respectant la règle que deux sommets adjacents ne peuvent pas avoir la même couleur.

```
afficherColoration(g1);
exporterDot(g1, "graph1.dot");
```

Ce bloc affiche les couleurs attribuées aux sommets et génère un fichier DOT pour visualiser le graphe avec Graphviz.

```
libererGraphe(g1);
```

Libère toute la mémoire allouée pour le graphe afin d'éviter les fuites.

Résultat après exécution (Terminal)

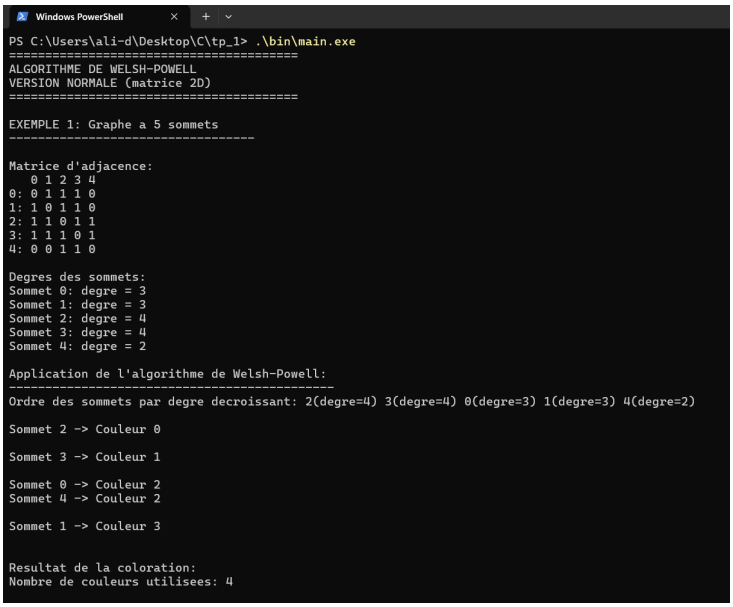

FIGURE 1 – Résultat de l'exécution dans le terminal

Visualisation du graphe (Graphviz)

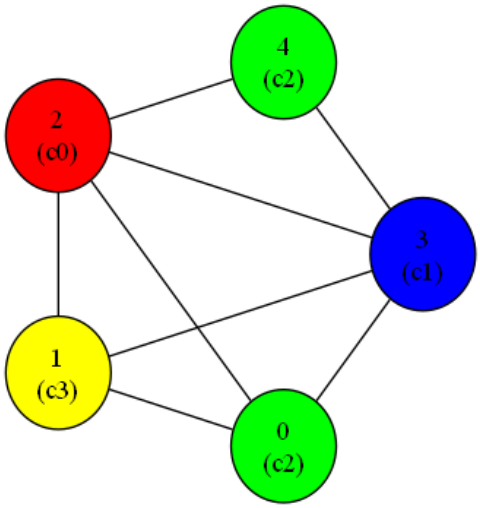

FIGURE 2 – Représentation graphique du graphe colorié

Extension vers une version linéarisée

La **version linéarisée** du graphe consiste à représenter la matrice d'adjacence non pas sous forme d'un tableau 2D classique (`int **adjacence`), mais sous forme d'un **tableau 1D** (`int *adjacence`) de taille `nbSommets * nbSommets`. L'accès à l'élément correspondant à l'arête entre le sommet i et le sommet j se fait via l'indice calculé `i * nbSommets + j`. Cette approche permet de simplifier la gestion mémoire et peut améliorer les performances pour certains accès séquentiels. La principale différence avec la version 2D est donc la façon dont les arêtes sont stockées et indexées, tout en conservant la logique de l'algorithme inchangée.

Comparaison entre la version 2D et la version linéarisée

Structure de la matrice d'adjacence

- **Version normale :**

```
int **adjacence;
```

Cette version utilise un tableau de pointeurs vers des tableaux, ce qui permet un accès direct par `adjacence[i][j]`.

- **Version linéarisée :**

```
int *adjacence;
```

Ici, un seul tableau 1D est utilisé. L'accès à l'élément correspondant aux indices `i` et `j` se fait via `adjacence[i * nbSommets + j]`. Cette méthode réduit légèrement la consommation mémoire et simplifie la libération de la mémoire, mais nécessite de calculer l'index.

Allocation dans `creerGraphe()`

- **Version normale :**

```
g->adjacence = (int **)malloc(nbSommets * sizeof(int *));
for (int i = 0; i < nbSommets; i++) {
    g->adjacence[i] = (int *)calloc(nbSommets, sizeof(int));
}
```

Chaque ligne de la matrice est allouée séparément avec `calloc`, ce qui initialise toutes les valeurs à zéro. La libération de la mémoire nécessite de libérer chaque ligne puis le tableau de pointeurs.

- **Version linéarisée :**

```
g->adjacence = (int *)calloc(nbSommets * nbSommets, sizeof(int));
```

Un seul bloc de mémoire est alloué pour toute la matrice. Cela simplifie la libération de mémoire et réduit légèrement l'overhead, mais l'accès aux éléments nécessite un calcul d'index linéarisé.

Accès dans `ajouterArete()`

- **Version normale :**

```
g->adjacence[sommet1][sommet2] = 1;
g->adjacence[sommet2][sommet1] = 1;
```

L'accès se fait directement via deux indices, ce qui est intuitif et lisible. La syntaxe reflète la structure 2D de la matrice.

- **Version linéarisée :**

```
int n = g->nbSommets;
g->adjacence[sommet1 * n + sommet2] = 1;
g->adjacence[sommet2 * n + sommet1] = 1;
```

L'accès utilise un calcul d'index pour simuler la 2D sur un tableau 1D. Cela réduit l'overhead mémoire mais nécessite de multiplier par le nombre de sommets pour chaque accès.

Accès dans `couleurUtiliseeParVoisin()`

- **Version normale :**

```
if (g->adjacence[sommet][i] == 1 && g->couleurs[i] == couleur)
```

L'accès à la matrice 2D est direct, simple et lisible. On vérifie facilement si le voisin possède déjà la couleur recherchée.

- **Version linéarisée :**

```
int n = g->nbSommets;
if (g->adjacence[sommet * n + i] == 1 && g->couleurs[i] == couleur)
```

L'accès est calculé via un index unique. Cela simule une matrice 2D dans un tableau 1D et économise de la mémoire, mais rend la lecture légèrement moins intuitive.

Note importante

Le résultat de l'exécution est le même qu'avec la version 2D.