Compte Rendu TP2
Graphes et Parcours d'Arbres

Rayane KACHBI

9 novembre 2025

1 Partie 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.

1.1 Chargement du Graphe

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

```
/**
 * @brief Charge un graphe depuis un fichier texte (ou stdin).
 * ...
 */
Graphe* chargeGraphe(const char* nom_fichier);

/**
 * @brief Affiche l'ordre de marquage (tri par degre)
 * et le resultat de la coloration.
 * ...
 */
void afficher_ordre_marquage_et_resultat(Graphe* g,
    int* couleurs, const char** noms_sommets);
```

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.

1.2 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, 1 :Es, 2 :Po, 3 :An, 4 :It, 5 :Au, 6 :Su, 7 :Al, 8 :Lu, 9 :Be, 10 :PB. Mon fichier `carte_europe.txt` contient donc :

```
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_marqu` en lui passant un tableau de noms de pays pour un affichage clair.

1.3 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**.
  - Couleur 1 : France (Fr) Portugal (Po) Autriche (Au) Pays-Bas (PB)
  - Couleur 2 : Espagne (Es) Italie (It) Allemagne (Al)
  - Couleur 3 : Andorre (An) Suisse (Su) Belgique (Be)
  - Couleur 4 : Luxembourg (Lu)

1.3.1 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×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 $O(K \times N^2)$. 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 $O(N^3)$.

1.3.2 Nombre Chromatique

L'algorithme de Welsh-Powell est une heuristique gloutonne. Il est rapide ($O(N^3)$ est polynomial, donc "raisonnable"), mais il ne garantit **pas** de trouver le nombre chromatique (le nombre minimum de couleurs).

Trouver le nombre chromatique exact est un problème NP-difficile. On ne sait pas le résoudre en temps polynomial. 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 (même si dans ce cas précis, c'est probable car l'Allemagne, la France, la Suisse et le Luxembourg forment un "clique" de 4, K4, qui nécessite à lui seul 4 couleurs).

2 Partie 2 : Parcours d'Arbres Binaires

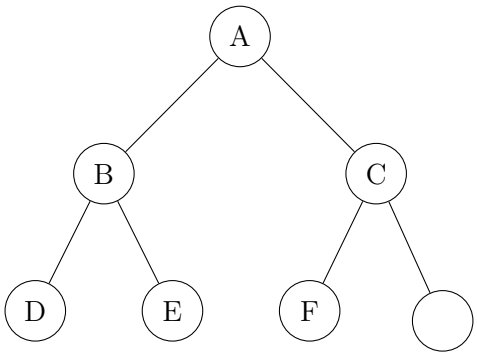
La seconde partie du TP demandait d'implémenter les parcours d'arbres binaires (Préfixe, Infixe, Postfixe) en se basant sur la structure du `TP0.pdf`.

2.1 Structure et Création

J'ai créé les fichiers `arbre.h` et `arbre.c`. La structure `Noeud` utilisée est la suivante :

```
typedef struct Noeud {
    char val; // Valeur du n ud
    struct Noeud* gauche; // Pointeur vers le fils gauche
    struct Noeud* droit;  // Pointeur vers le fils droit
} Noeud;
```

Pour tester les parcours, j'ai créé une fonction `creerArbreSynthetique()` qui construit l'arbre suivant de manière "ad hoc" :



2.2 Implémentation des Parcours

J'ai implémenté les trois parcours de manière récursive.

2.2.1 Parcours Préfixe (Racine → Gauche → Droit)

L'ordre de visite est : (1) Traiter la racine, (2) Visiter le sous-arbre gauche, (3) Visiter le sous-arbre droit.

```
void parcoursPrefixe(Noeud* racine) {
    if (racine == NULL) return;

    printf("%c␣", racine->val); // 1. Racine
    parcoursPrefixe(racine->gauche); // 2. Gauche
    parcoursPrefixe(racine->droit);  // 3. Droit
}
```

Résultat obtenu : `A B D E C F`

2.2.2 Parcours Infixe (Gauche → Racine → Droit)

L'ordre de visite est : (1) Visiter le sous-arbre gauche, (2) Traiter la racine, (3) Visiter le sous-arbre droit.

```
void parcoursInfixe(Noeud* racine) {
    if (racine == NULL) return;

    parcoursInfixe(racine->gauche); // 1. Gauche
```

```
    printf("%c ", racine->val); // 2. Racine
    parcoursInfixe(racine->droit);  // 3. Droit
}
```

Résultat obtenu : `D B E A F C`

2.2.3 Parcours Postfixe (Gauche → Droit → Racine)

L'ordre de visite est : (1) Visiter le sous-arbre gauche, (2) Visiter le sous-arbre droit, (3) Traiter la racine.

```
void parcoursPostfixe(Noeud* racine) {
    if (racine == NULL) return;

    parcoursPostfixe(racine->gauche); // 1. Gauche
    parcoursPostfixe(racine->droit);  // 2. Droit
    printf("%c ", racine->val); // 3. Racine
}
```

Résultat obtenu : `D E B F C A`

3 Conclusion

Les deux parties du TP ont été réalisées. La partie sur les graphes m'a permis de finaliser l'algorithme de Welsh-Powell et de l'appliquer à un cas réel, tout en analysant sa complexité. La partie sur les arbres a été une bonne révision des parcours récursifs fondamentaux.