

Compte rendu de la partie B du TP2

Ali Gouarab

15 Novembre 2025

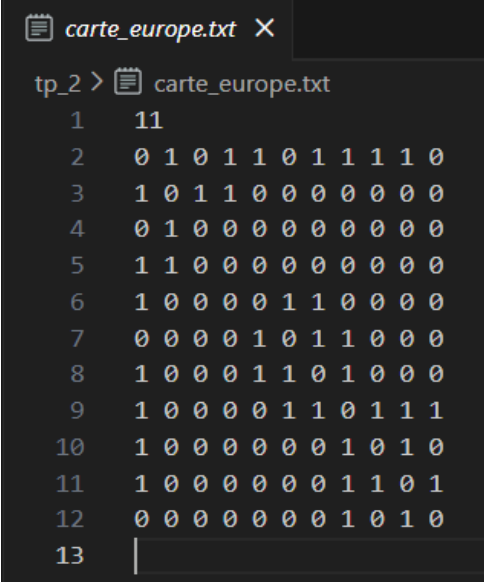
Structure des dossiers et du code

Pour le TP2, j'ai organisé mon travail dans un dossier principal `tp_2` contenant trois sous-dossiers : `src`, `include` et `bin`. Le dossier `src` contient les fichiers sources pour les deux exercices : `arbre.c` et `monArbre.c` pour la manipulation des arbres binaires, ainsi que `graphe.c` et `monGraphe.c` pour la manipulation des graphes et l'implémentation de l'algorithme de Welsh-Powell. Le dossier `include` contient les fichiers d'en-tête `arbre.h` et `graphe.h` regroupant les structures de données et les prototypes de fonctions correspondants. Le dossier `bin` est réservé aux exécutables générés après compilation.

À la racine du dossier `tp_2` se trouve également le fichier `carte_europe.txt`, qui représente la matrice d'adjacence des pays européens et sera utilisé pour tester l'algorithme de coloriage

1 Suite de Welsh-Powell et matrice d'adjacence

Dans cette partie, nous allons créer une fonction `main()` de test permettant de charger un graphe représenté par une matrice d'adjacence, depuis un fichier externe. Chaque cellule de la matrice contient un poids 0 ou 1 et la première ligne du fichier indique le nombre de sommets.



```
tp_2 > carte_europe.txt
1  11
2  0 1 0 1 1 0 1 1 1 1 0
3  1 0 1 1 0 0 0 0 0 0 0
4  0 1 0 0 0 0 0 0 0 0 0
5  1 1 0 0 0 0 0 0 0 0 0
6  1 0 0 0 0 1 1 0 0 0 0
7  0 0 0 0 1 0 1 1 0 0 0
8  1 0 0 0 1 1 0 1 0 0 0
9  1 0 0 0 0 1 1 0 1 1 1
10 1 0 0 0 0 0 0 1 0 1 0
11 1 0 0 0 0 0 0 1 1 0 1
12 0 0 0 0 0 0 0 1 0 1 0
13 |
```

FIGURE 1 – Carte des pays européens représentée sous forme de matrice d'adjacence

Le programme affichera également l'ordre de marquage ou le chemin de visite des sommets, conformément à l'algorithme de Welsh-Powell. L'objectif est de reproduire le résultat du TP0 sur la carte des pays européens et d'effectuer une analyse rapide de la complexité de l'algorithme.

Chargement d'un graphe à partir d'un fichier

La fonction `chargeGraphe()` permet de créer un graphe à partir d'un fichier texte contenant le nombre de sommets et la matrice d'adjacence du graphe (poids 0 ou 1). Voici une explication étape par étape du code :

▷ Ouverture du fichier :

```
FILE *f = fopen(nomFichier, "r");
if (!f) {
    perror("Erreur ouverture fichier");
    return NULL;
}
```

On tente d'ouvrir le fichier en lecture. Si l'ouverture échoue, la fonction affiche un message d'erreur et retourne `NULL`.

▷ Lecture du nombre de sommets :

```
int nbSommets;
if (fscanf(f, "%d", &nbSommets) != 1) {
    fprintf(stderr, "Erreur lecture nombre de sommets\n");
    fclose(f);
    return NULL;
}
```

La première ligne du fichier doit contenir le nombre de sommets. On vérifie que la lecture s'est bien passée, sinon on ferme le fichier et on retourne `NULL`.

▷ Création du graphe :

```
Graphe *g = creerGraphe(nbSommets);
if (!g) {
    fclose(f);
    return NULL;
}
```

On utilise la fonction `creerGraphe()` pour allouer la mémoire du graphe et initialiser les tableaux de couleurs, degrés et matrice d'adjacence. Si l'allocation échoue, on ferme le fichier et retourne `NULL`.

▷ Lecture de la matrice d'adjacence :

```
for (int i = 0; i < nbSommets; i++) {
    for (int j = 0; j < nbSommets; j++) {
        if (fscanf(f, "%d", &g->adjacence[i][j]) != 1) {
            fprintf(stderr, "Erreur lecture matrice position [%d][%d]\n",
                i, j);
            libererGraphe(g);
            fclose(f);
            return NULL;
        }
        if (g->adjacence[i][j] == 1) {
            g->degres[i]++;
        }
    }
}
```

Chaque valeur de la matrice est lue et stockée dans `g->adjacence[i][j]`. En cas d'erreur de lecture, le graphe est libéré, le fichier fermé et `NULL` est retourné. Si une arête existe (1), le degré du sommet

correspondant est incrémenté.

► **Fermeture du fichier et confirmation :**

```
fclose(f);  
printf("Graphe charge avec succes : %d sommets\n", nbSommets);  
return g;
```

Une fois toute la matrice lue, le fichier est fermé. La fonction affiche un message confirmant la réussite du chargement et retourne le pointeur vers le graphe créé.

Cette fonction permet donc de transformer un fichier externe en une structure de graphe exploitable pour l'algorithme de Welsh-Powell ou d'autres traitements.

Note importante

Les autres fonctions du programme sont similaires à celles du TP précédent, avec quelques améliorations et séparations, la logique générale reste toutefois identique.

Tests et Exemples d'exécution

Pour vérifier le bon fonctionnement de l'algorithme de Welsh-Powell sur la carte de l'Europe, nous avons utilisé la fonction `main()` suivante :

```
Graphe *g = chargeGraphe("carte_europe.txt");
```

Ce bloc charge le graphe à partir du fichier texte contenant le nombre de sommets et la matrice d'adjacence des pays européens. Si le fichier ne peut pas être ouvert, un message d'erreur est affiché et le programme se termine.

```
afficherGraphe(g);
```

Ce bloc permet d'afficher la matrice d'adjacence du graphe chargé afin de vérifier que le graphe a bien été créé.

```
welshPowell(g);
```

Applique l'algorithme de Welsh-Powell pour colorier le graphe en respectant la règle que deux sommets adjacents ne peuvent pas avoir la même couleur.

```
libererGraphe(g);
```

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

Correspondance indices / pays

Pour interpréter les résultats de coloration, chaque sommet est identifié par un indice correspondant au pays suivant :

Indice	Pays
0	France
1	Espagne
2	Portugal
3	Andorre
4	Italie
5	Autriche
6	Suisse
7	Allemagne
8	Luxembourg
9	Belgique
10	Pays-Bas

Résultat après exécution (Terminal)

```

Windows PowerShell
Graphe charge avec succes : 11 sommets

Matrice d'adjacence (11 sommets):
  0  1  2  3  4  5  6  7  8  9 10
0: 0  1  0  1  1  0  1  1  1  1  0
1: 1  0  1  1  0  0  0  0  0  0  0
2: 0  1  0  0  0  0  0  0  0  0  0
3: 1  1  0  0  0  0  0  0  0  0  0
4: 1  0  0  0  0  1  1  0  0  0  0
5: 0  0  0  0  1  0  1  1  0  0  0
6: 1  0  0  0  1  1  0  1  0  0  0
7: 1  0  0  0  0  1  1  0  1  1  1
8: 1  0  0  0  0  0  0  1  0  1  0
9: 1  0  0  0  0  0  0  1  1  0  1
10: 0  0  0  0  0  0  0  1  0  1  0

=== ALGORITHME DE WELSH-POWELL ===
Coloration de graphe par ordre decroissant de degre

Degres des sommets:
Sommet 0 : degre 7
Sommet 1 : degre 3
Sommet 2 : degre 1
Sommet 3 : degre 2
Sommet 4 : degre 3
Sommet 5 : degre 3
Sommet 6 : degre 4
Sommet 7 : degre 6
Sommet 8 : degre 3
Sommet 9 : degre 4
Sommet 10 : degre 2

Ordre de traitement (par degre decroissant):
 0 7 6 9 1 4 5 8 3 10 2

Attribution des couleurs:
Sommet 0 -> Couleur 0
Sommet 5 -> Couleur 0
Sommet 10 -> Couleur 0
Sommet 2 -> Couleur 0
Sommet 7 -> Couleur 1
Sommet 1 -> Couleur 1
Sommet 4 -> Couleur 1
Sommet 6 -> Couleur 2
Sommet 9 -> Couleur 2
Sommet 3 -> Couleur 2
Sommet 8 -> Couleur 3

--- RESULTAT DE LA COLORATION ---
Nombre de couleurs utilisees : 4

Couleur 0 : 0 2 5 10
Couleur 1 : 1 4 7
Couleur 2 : 3 6 9
Couleur 3 : 8

```

FIGURE 2 – Résultat de l'exécution de l'algorithme de Welsh-Powell dans le terminal sur les pays européens

Visualisation du graphe (Graphviz)

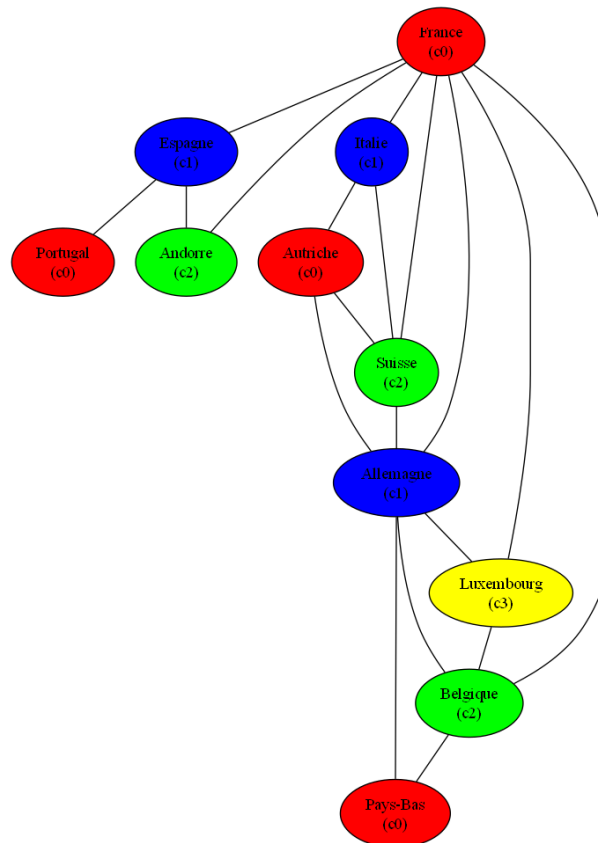


FIGURE 3 – Graphe des pays européens après coloration par l’algorithme de Welsh-Powell

2 Implémentation des parcours d’arbres binaires

Dans cette partie, nous implémentons et testons les parcours d’arbres binaires. Pour générer les arbres sur lesquels ces parcours seront appliqués, nous avons défini deux fonctions, `creerArbre()` pour créer des arbres parfaits et `creerArbreAleatoire()` pour créer des arbres de structure aléatoire. Ces fonctions permettent de générer facilement des arbres de différentes tailles et configurations, afin d’appliquer les trois parcours classiques préfixe, infixe et postfixe et d’analyser les résultats obtenus pour illustrer le fonctionnement de chaque parcours.

Création d’arbres binaires

Pour générer les arbres sur lesquels nous appliquerons les parcours, nous avons défini deux fonctions :

```
noeud *creerArbre(int hauteur, char valeurDepart) {
    if (hauteur == 0)
        return NULL;

    noeud *n = nouvNoeud(valeurDepart);

    n->filsGauche = creerArbre(hauteur - 1, valeurDepart + 1);
    n->filsDroit = creerArbre(hauteur - 1, valeurDepart + 1);

    return n;
}
```

Cette fonction crée un arbre binaire parfait de hauteur **hauteur**. Chaque niveau est complètement rempli et les valeurs des nœuds augmentent à chaque niveau. La récursion permet de construire l'arbre de manière naturelle en affectant récursivement les fils gauche et droit.

```
noeud *creerArbreAleatoire(int hauteur, char valeurDepart) {
    if (hauteur == 0)
        return NULL;

    noeud *n = nouvNoeud(valeurDepart);

    if (rand() % 2) {
        n->filsGauche = creerArbreAleatoire(hauteur - 1, valeurDepart + 1);
    }

    if (rand() % 2) {
        n->filsDroit = creerArbreAleatoire(hauteur - 1, valeurDepart + 1);
    }

    return n;
}
```

Cette fonction crée un arbre binaire de manière aléatoire. À chaque niveau, la présence d'un fils gauche ou droit est décidée aléatoirement grâce à l'expression `rand() % 2`, qui renvoie soit 0 soit 1, donnant ainsi une chance de 50/50 à chaque branche d'être créée. Cela permet de générer des arbres non équilibrés et de tester les parcours sur différentes structures.

Parcours d'arbres binaires

Nous avons implémenté les trois types de parcours classiques pour les arbres binaires : préfixe, infixe et postfixe. Chaque fonction est récursive et s'arrête lorsque le nœud courant est NULL.

```
void parcoursPrefixe(noeud *n) {
    if (n == NULL)
        return;

    printf("%c(%d) ", n->val, n->num);
    parcoursPrefixe(n->filsGauche);
    parcoursPrefixe(n->filsDroit);
}
```

Le parcours préfixe visite le nœud courant avant ses sous-arbres gauche et droit. On imprime donc la valeur du nœud, puis on explore récursivement le fils gauche et le fils droit.

```
void parcoursInfixe(noeud *n) {
    if (n == NULL)
        return;

    parcoursInfixe(n->filsGauche);
    printf("%c(%d) ", n->val, n->num);
    parcoursInfixe(n->filsDroit);
}
```

Le parcours infixe visite d'abord le sous-arbre gauche, puis le nœud courant, et enfin le sous-arbre droit. Ce parcours est souvent utilisé pour obtenir les valeurs des nœuds dans un ordre croissant pour les arbres binaires de recherche.

```

void parcoursPostfixe(noeud *n) {
    if (n == NULL)
        return;

    parcoursPostfixe(n->filsGauche);
    parcoursPostfixe(n->filsDroit);
    printf("%c(%d) ", n->val, n->num);
}

```

Le parcours postfixe explore d'abord les sous-arbres gauche et droit, puis visite le nœud courant. Ce type de parcours est utilisé lorsqu'on souhaite effectuer des opérations sur les sous-arbres avant de traiter le nœud parent.

Note importante

Les autres fonctions du programme sont similaires à celles du TP 0 implémentation d'arbres binaires.

Tests et Exemples d'exécution

Pour vérifier le bon fonctionnement des parcours d'arbres binaires, nous avons utilisé la fonction `main()` suivante :

```

srand(time(NULL));

printf("Creation d'un arbre aleatoire de hauteur 4...\n");
noeud *arbre = creerArbreAleatoire(4, 'A');

if (arbre == NULL) {
    printf("Erreur: arbre vide\n");
    return 1;
}

```

Ce bloc initialise le générateur de nombres aléatoires et crée un arbre binaire aléatoire de hauteur 4 (on va changer cette valeur par la suite pour les tests). Il vérifie ensuite que l'arbre n'est pas vide.

```

printf("Arbre cree avec succes!\n\n");
exporterArbreDot(arbre, "arbre_binaire.dot");

```

Affiche un message de succès et génère un fichier DOT pour visualiser l'arbre avec Graphviz.

```

printf("PARCOURS PREFIXE : \n");
parcoursPrefixe(arbre);
printf("\n\n");

printf("PARCOURS INFIXE : \n");
parcoursInfixe(arbre);
printf("\n\n");

printf("PARCOURS POSTFIXE : \n");
parcoursPostfixe(arbre);
printf("\n\n");

```

Ce bloc applique successivement les parcours préfixe, infixe et postfixe sur l'arbre généré et affiche les résultats dans le terminal.

```
return 0;
```

Termine l'exécution du programme.

Résultat après exécution du programme

Nous avons exécuté le programme 3 fois pour générer différents arbres aléatoires et observer les parcours préfixe, infixe et postfixe.

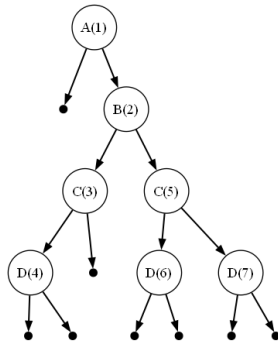


FIGURE 4 – Visualisation Graphviz de l'arbre (hauteur = 4)

```
Creation d'un arbre aleatoire de hauteur 4...
Arbre cree avec succes!

==> Fichier DOT genere: arbre_binaire.dot
    Pour visualiser: dot -Tpng arbre_binaire.dot -o arbre.png

PARCOURS PREFIXE :
A(1) B(2) C(3) D(4) C(5) D(6) D(7)

PARCOURS INFIXE :
A(1) D(4) C(3) B(2) D(6) C(5) D(7)

PARCOURS POSTFIXE :
D(4) C(3) D(6) D(7) C(5) B(2) A(1)
```

FIGURE 5 – Sortie terminal (hauteur = 4)

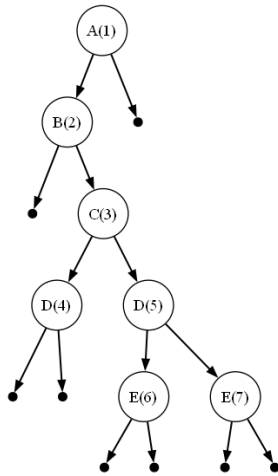


FIGURE 6 – Visualisation Graphviz de l'arbre (hauteur = 5)

```
Creation d'un arbre aleatoire de hauteur 5...
Arbre cree avec succes!

==> Fichier DOT genere: arbre_binaire.dot
    Pour visualiser: dot -Tpng arbre_binaire.dot -o arbre.png

PARCOURS PREFIXE :
A(1) B(2) C(3) D(4) D(5) E(6) E(7)

PARCOURS INFIXE :
B(2) D(4) C(3) E(6) D(5) E(7) A(1)

PARCOURS POSTFIXE :
D(4) E(6) E(7) D(5) C(3) B(2) A(1)
```

FIGURE 7 – Sortie terminal (hauteur = 5)

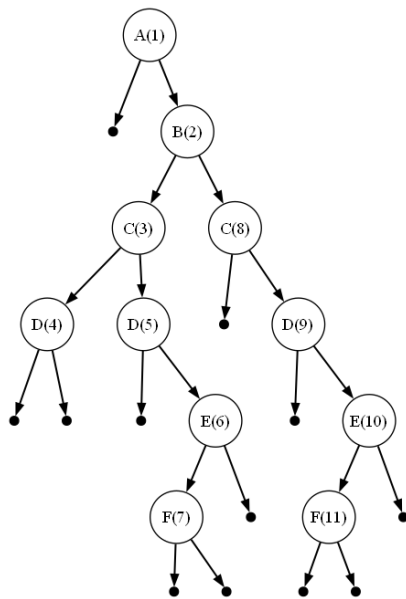


FIGURE 8 – Visualisation Graphviz de l'arbre (hauteur = 6)

```
Creation d'un arbre aleatoire de hauteur 6...
Arbre cree avec succes!

==> Fichier DOT genere: arbre_binaire.dot
    Pour visualiser: dot -Tpng arbre_binaire.dot -o arbre.png

PARCOURS PREFIXE :
A(1) B(2) C(3) D(4) D(5) E(6) F(7) C(8) D(9) E(10) F(11)

PARCOURS INFIXE :
A(1) D(4) C(3) D(5) F(7) E(6) B(2) C(8) D(9) F(11) E(10)

PARCOURS POSTFIXE :
D(4) F(7) E(6) D(5) C(3) F(11) E(10) D(9) C(8) B(2) A(1)
```

FIGURE 9 – Sortie terminal (hauteur = 6)

Note importante

Hauteur des arbres générés :

- ▶ Avec `creerArbreAleatoire()`, la hauteur de l'arbre peut varier entre **1** (seulement la racine si aucun fils n'est créé) et **6** (si tous les fils sont créés jusqu'au dernier niveau).
- ▶ Pour garantir un arbre complet de hauteur **6**, il faut utiliser `creerArbre(6, 'A')`, qui crée systématiquement tous les fils à chaque niveau.