Compte rendu de la partie B du TP3

Ali Gouarab

15 Novembre 2025

Structure des dossiers et du code

Pour le TP3, j'ai organisé mon travail dans un dossier principal `tp_3` contenant plusieurs sous-dossiers : `src`, `include`, `bin` et `correction_chatgpt`. Le dossier `src` contient les fichiers sources pour les deux exercices, `arbre_generique.c` et `monArbre.c` pour la manipulation des arbres génériques, ainsi que `graphe_liste.c`, `monGraphe.c` et `benchmark_welsh_powell.c` pour la manipulation des graphes représentés par des listes chaînées et le benchmarking de l'algorithme de Welsh-Powell. Le dossier `include` contient les fichiers d'en-tête `arbre_generique.h` et `arbre_liste.h` regroupant les structures de données et les prototypes de fonctions correspondants. Le dossier `bin` est réservé aux exécutables générés après compilation, et `correction_chatgpt` contient des fichiers de correction en C et Python (`welshpowell.c` et `welshpowell.py`).

1 Implémentation des structures avec des listes chaînées

Conversion des arbres binaires en arbres génériques

Un **arbre générique** est une structure hiérarchique composée de **nœuds**, où chaque nœud peut avoir un nombre arbitraire d'enfants. Pour représenter efficacement cet arbre, nous utilisons la technique dite **first-child / next-sibling** :

- Chaque nœud contient une valeur (`val`) et un identifiant unique (`num`).
- Le pointeur `premierFils` pointe vers le premier enfant du nœud.
- Le pointeur `frèreSuivant` pointe vers le frère suivant du nœud.

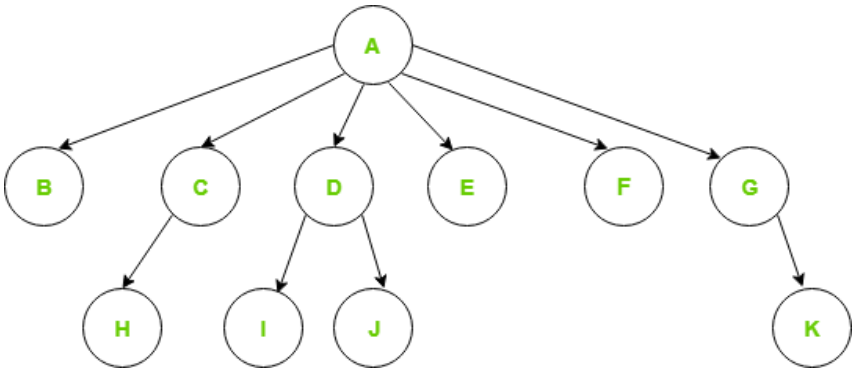

FIGURE 1 – Représentation d'un arbre générique

Modifications apportées de l'arbre binaire à l'arbre générique

Structure du nœud

- **Version arbre binaire :**

```c
typedef struct noeud {
    char val;
    int num;
    struct noeud *filsGauche;
    struct noeud *filsDroit;
} noeud;
```

Ici, chaque nœud possède exactement deux pointeurs de descendance : un vers le fils gauche et un vers le fils droit. Cette structure limite l'arbre à un maximum de deux enfants par nœud.

- **Version arbre générique :**

```c
typedef struct noeud {
    char val;
    int num;
    struct noeud *premierFils;
    struct noeud *frereSuivant;
} noeud;
```

Cette version utilise la représentation *first-child / next-sibling*. Chaque nœud peut ainsi avoir un nombre illimité d'enfants :

- `premierFils` pointe vers le premier enfant du nœud.
- `frereSuivant` relie les enfants entre eux sous forme de liste chaînée.

Fonction `rechercheNoeud()`

- **Version arbre binaire :**

```c
noeud *rechercheNoeud(noeud *n, int num_noeud) {
    if (n == NULL) return NULL;
    if (n->num == num_noeud) return n;

    tmpNoeud = rechercheNoeud(n->filsGauche, num_noeud);
    if (tmpNoeud != NULL) return tmpNoeud;

    return rechercheNoeud(n->filsDroit, num_noeud);
}
```

Dans un arbre binaire, la recherche s'effectue en explorant récursivement d'abord le fils gauche, puis le fils droit. La structure étant limitée à deux enfants, seuls deux appels récursifs sont nécessaires.

- **Version arbre générique :**

```c
noeud *rechercheNoeud(noeud *n, int num_noeud) {
    if (n == NULL) return NULL;
    if (n->num == num_noeud) return n;

    tmpNoeud = rechercheNoeud(n->premierFils, num_noeud);
    if (tmpNoeud != NULL) return tmpNoeud;

    return rechercheNoeud(n->frereSuivant, num_noeud);
}
```

Avec la représentation *first-child / next-sibling*, la recherche doit :

- explorer récursivement tous les enfants via `premierFils`,
- puis parcourir tous les frères via `frereSuivant`.

Ce mécanisme revient à parcourir une structure équivalente à une liste chaînée horizontale et verticale, permettant de gérer un nombre illimité d'enfants.

Fonction d'insertion

- **Version arbre binaire :**

```
void insererFG(noeud *noeudAInserer, noeud *arbre, int num_parent);
void insererFD(noeud *noeudAInserer, noeud *arbre, int num_parent);
```

Dans un arbre binaire, l'insertion est strictement limitée à deux positions possibles : `fils gauche` ou `fils droit`. Deux fonctions distinctes sont donc nécessaires, chacune insérant le nœud du côté approprié.

- **Version arbre générique :**

```
void ajouterEnfant(noeud *parent, noeud *enfant) {
    if (parent->premierFils == NULL) {
        parent->premierFils = enfant;
    } else {
        noeud *frere = parent->premierFils;
        while (frere->frereSuivant != NULL) {
            frere = frere->frereSuivant;
        }
        frere->frereSuivant = enfant;
    }
}
```

Dans un arbre générique, chaque nœud peut avoir un nombre illimité d'enfants. L'insertion consiste alors à :

- placer l'enfant comme `premierFils` s'il n'y en a pas encore,
- ou parcourir la liste chaînée des frères via `frereSuivant` pour l'ajouter en fin de chaîne.

Cette approche rend la structure extensible sans limite, contrairement à l'arbre binaire.

Parcours préfixe

- **Version arbre binaire :**

```
void parcoursPrefixe(noeud *n) {
    if (n == NULL) return;
    printf("%c(%d) ", n->val, n->num);
    parcoursPrefixe(n->filsGauche);
    parcoursPrefixe(n->filsDroit);
}
```

Dans un arbre binaire, le parcours préfixe suit l'ordre : **racine → fils gauche → fils droit**. Cet ordre est fixe car chaque nœud possède au maximum deux enfants.

- **Version arbre générique :**

```
void parcoursPrefixe(noeud *n) {
    if (n == NULL) return;
    printf("%c(%d) ", n->val, n->num);
    parcoursPrefixe(n->premierFils);
    parcoursPrefixe(n->frereSuivant);
}
```

Dans un arbre générique, la logique change :

- On visite d'abord le nœud courant (**racine**),
- puis on parcourt récursivement toute la chaîne des enfants via `premierFils`,

- et enfin la chaîne des frères via `frereSuivant`.

Ce schéma remplace les deux appels fixes `filsGauche` et `filsDroit` par une navigation libre dans une structure potentiellement infinie d'enfants et de frères.

Représentation d'un graphe à l'aide de listes chaînées

Un **graphe** est une structure de données composée d'un ensemble de **noeuds** (ou sommets) et de **lien(s)** (ou arêtes) entre ces noeuds. Chaque noeud peut être connecté à un ou plusieurs autres noeuds. Pour représenter un graphe quelconque, nous utilisons une structure permettant de stocker pour chaque noeud la liste de ses voisins. Cette représentation est particulièrement adaptée aux graphes clairsemés et permet d'ajouter ou de supprimer des arêtes facilement.

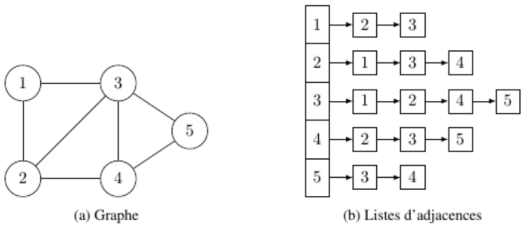

FIGURE 2 – Représentation d'un graphe à l'aide d'une liste d'adjacence

Adaptation de la structure du graphe vers une représentation en listes d'adjacence

Structures de données

- **Graphe avec matrice d'adjacence :**

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

Chaque sommet est représenté par un indice et l'existence d'une arête entre deux sommets est indiquée par l'entrée correspondante dans la matrice. Les couleurs et degrés sont stockés dans des tableaux séparés.

- **Graphe avec liste d'adjacence :**

```
typedef struct voisin {
    struct noeud *noeud;
    struct voisin *suivant;
} voisin;

typedef struct noeud {
    char *nom;
    voisin *voisins;
} noeud;
```

Ici, chaque sommet est un nœud contenant son nom et une liste chaînée de voisins. Cette structure permet de représenter efficacement des graphes creux et de parcourir rapidement les voisins d'un sommet.

Création d'un graphe

- **Graphe avec matrice d'adjacence :**

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

Chaque sommet et ses relations avec les autres sommets sont représentés par une matrice carrée. L'espace mémoire est proportionnel à n^2.

- **Graphe avec liste d'adjacence :**

```
noeud *creerNoeud(const char *nom) {
    n->nom = (char *)malloc(strlen(nom) + 1);
    strcpy(n->nom, nom);
    n->voisins = NULL;
}
```

Chaque sommet est initialisé avec son nom et une liste chaînée vide pour ses voisins. Cette approche est plus économe en mémoire pour les graphes peu denses.

Ajout d'une arête

- **Graphe avec matrice d'adjacence :**

```
void ajouterArete(Graphe *g, int sommet1, int sommet2) {
    g->adjacence[sommet1][sommet2] = 1;
    g->adjacence[sommet2][sommet1] = 1;
    g->degres[sommet1]++;
    g->degres[sommet2]++;
}
```

L'arête est ajoutée en modifiant directement la matrice. Chaque mise à jour des degrés est immédiate, et l'accès reste en temps constant $O(1)$.

- **Graphe avec liste d'adjacence :**

```
void ajouterArete(noeud *n1, noeud *n2) {

    voisin *v1 = (voisin *)malloc(sizeof(voisin));
    v1->noeud = n2;
    v1->suivant = n1->voisins;
    n1->voisins = v1;

    voisin *v2 = (voisin *)malloc(sizeof(voisin));
    v2->noeud = n1;
    v2->suivant = n2->voisins;
    n2->voisins = v2;
}
```

Chaque arête est représentée par deux éléments dans les listes chaînées des deux sommets. L'insertion se fait en tête de liste, ce qui est très efficace en $O(1)$, même pour des graphes de grande taille.

Affichage du graphe

- **Graphe avec matrice d'adjacence :**

```c
void afficherGraphe(Graphe *g) {
    for (int i = 0; i < g->nbSommets; i++) {
        for (int j = 0; j < g->nbSommets; j++) {
            printf("%2d ", g->adjacence[i][j]);
        }
    }
}
```

L'affichage parcourt la matrice entière et imprime chaque valeur. Cela permet de visualiser rapidement toutes les arêtes, mais nécessite $O(n^2)$ opérations pour un graphe de n sommets.

- **Graphe avec liste d'adjacence :**

```c
void afficherVoisins(noeud *n) {
    printf("%s est voisin de: ", n->nom);
    voisin *v = n->voisins;
    while (v != NULL) {
        printf("%s ", v->noeud->nom);
        v = v->suivant;
    }
}

void afficherGraphe(noeud **graphe, int nb_noeuds) {
    for (int i = 0; i < nb_noeuds; i++) {
        afficherVoisins(graphe[i]);
    }
}
```

Chaque sommet parcourt sa propre liste de voisins. Cette méthode est plus efficace pour les graphes clairsemés, car elle ne visite que les arêtes existantes, réduisant ainsi le coût à $O(n + m)$ pour n sommets et m arêtes.

Calcul du degré

- **Graphe avec matrice d'adjacence :**

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

Le degré de chaque sommet est incrémenté à chaque ajout d'arête, ce qui permet un accès direct en temps constant $O(1)$.

- **Graphe avec liste d'adjacence :**

```c
int degre(noeud *n) {
    int deg = 0;
    voisin *v = n->voisins;
    while (v != NULL) {
        deg++;
        v = v->suivant;
    }
    return deg;
}
```

Ici, le degré est calculé dynamiquement en parcourant la liste des voisins. Le coût est proportionnel au nombre de voisins $O(d)$, où d est le degré du sommet.

Vérification d'adjacence

- **Graphe avec matrice d'adjacence :**

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

Cette fonction parcourt toute la ligne correspondant au sommet dans la matrice pour vérifier si un voisin a déjà la couleur donnée, coût $O(n)$.

- **Graphe avec liste d'adjacence :**

```
int couleurUtiliseeParVoisin(noeud *n, int couleur) {
    voisin *v = n->voisins;
    while (v != NULL) {
        if (v->noeud->couleur == couleur) {
            return 1;
        }
        v = v->suivant;
    }
    return 0;
}
```

Ici, seuls les voisins existants sont parcourus, ce qui réduit le coût à $O(d)$ où d est le degré du sommet.

2 Correction des codes de coloration

Code en C

Dans le code en C, plusieurs problèmes ont été identifiés :

1. **Mauvaise initialisation du tableau de couleurs :** Le code original :

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

initialise seulement `color[0]` à `-1`, alors que nous souhaitons que toutes les couleurs soient à `-1`. La correction consiste à initialiser explicitement toutes les cases :

```
for (i = 0; i < graph->vertices; i++) {
    color[i] = -1;
}
```

2. **Problème d'incrémentation des couleurs :** Dans le code original, on incrémente toujours la couleur courante :

```
else {
    current_color++;
    color[vertex] = current_color;
}
```

Ainsi, lorsqu'un sommet a un voisin avec la même couleur, on passe directement à la couleur suivante, alors qu'il serait parfois possible d'utiliser une couleur inférieure.

3. **Solution proposée :** Le code suivant montre l'implémentation corrigée de l'algorithme de Welsh-Powell, avec initialisation correcte des couleurs et vérification systématique des voisins avant l'attribution de la couleur.

```
void welshPowell(struct Graph* graph) {
    int degree[MAX_VERTICES] = {0};
    int i, j, current_color;

    for (i = 0; i < graph->vertices; i++) {
        for (j = 0; j < graph->vertices; j++) {
            if (graph->adjMatrix[i][j] == 1)
                degree[i]++;
        }
    }

    int sorted_vertices[MAX_VERTICES];

    for (i = 0; i < graph->vertices; i++)
        sorted_vertices[i] = i;

    for (i = 0; i < graph->vertices - 1; i++) {
        for (j = 0; j < graph->vertices - i - 1; j++) {
            if (degree[sorted_vertices[j]] < degree[sorted_vertices[j + 1]]) {
                int temp = sorted_vertices[j];
                sorted_vertices[j] = sorted_vertices[j + 1];
                sorted_vertices[j + 1] = temp;
            }
        }
    }

    int color[MAX_VERTICES];

    for (i = 0; i < graph->vertices; i++)
        color[i] = -1;

    current_color = 0;

    for (i = 0; i < graph->vertices; i++) {
        int vertex = sorted_vertices[i];

        if (color[vertex] == -1) {

            color[vertex] = current_color;

            for (j = i + 1; j < graph->vertices; j++) {
                int v = sorted_vertices[j];

                if (color[v] == -1) {
                    if (graph->adjMatrix[vertex][v] == 0) {
                        int can_color = 1;
                        int k = 0;
                        while (k < graph->vertices && can_color) {
                            if (graph->adjMatrix[v][k] == 1 && color[k] ==
                                current_color) {
                                can_color = 0;
                            }
                            k++;
                        }
                        if (can_color) {
                            color[v] = current_color;
                        }
                    }
                }
            }
            current_color++;
        }
    }

    int num_colors = current_color;

    printf("Graph colored using %d colors:\n", num_colors);
    for (i = 0; i < graph->vertices; i++) {
        printf("Vertex %d: Color %d\n", i, color[i]);
    }
}
```

Ce code trie les sommets par degré décroissant, puis colorie chaque sommet en choisissant la plus petite couleur possible tout en respectant les contraintes de voisinage, garantissant ainsi un résultat cohérent et minimal.

Code en Python

Dans ce code Python, plusieurs problèmes ont été identifiés :

1. La création de la liste des voisins non coloriés est incorrecte :

```
uncolored_adjacent = [adj_vertex for adj_vertex in range(self.vertices)
                      if self.adjMatrix[vertex][adj_vertex] == 1 and
                          color[adj_vertex] == -1]
```

   Cette ligne stocke uniquement les voisins du sommet courant qui ne sont pas encore coloriés.

2. Lors de la coloration avec ce bloc :

```
for adj_vertex in uncolored_adjacent:
    if not any(self.adjMatrix[adj_vertex][v] == 1 and color[v] ==
        current_color for v in range(self.vertices)):
        color[vertex] = current_color
        break
```

   on ne vérifie pas l'état des autres sommets adjacents déjà colorés. Comme la liste ne contient que les sommets non coloriés, il est possible de colorier le sommet courant avec une couleur déjà utilisée par un voisin existant, ce qui constitue une erreur.

   En résumé, le problème principal vient du fait que les sommets déjà coloriés adjacents au sommet courant ne sont jamais pris en compte dans la vérification. Pour corriger cela, il faut s'assurer que la couleur choisie pour un sommet n'est pas utilisée par aucun de ses voisins, qu'ils soient colorés ou non.

3. **Solution proposée :**

```
import networkx as nx
import matplotlib.pyplot as plt

class GraphWelshPowell:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adjMatrix = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v):
        self.adjMatrix[u][v] = 1
        self.adjMatrix[v][u] = 1

    def welsh_powell(self):
        degree = []
        for row in self.adjMatrix:
            degree.append(sum(row))

        sorted_vertices = sorted(range(self.vertices), key=lambda x:
            degree[x], reverse=True)

        color = [-1] * self.vertices
        current_color = 0

        for i in range(self.vertices):
            vertex = sorted_vertices[i]

            if color[vertex] == -1:
                color[vertex] = current_color

                for j in range(i + 1, self.vertices):
                    v = sorted_vertices[j]

                    if color[v] == -1:
                        if self.adjMatrix[vertex][v] == 0:
                            can_color = True
                            k = 0
                            while k < self.vertices and can_color:
                                if self.adjMatrix[v][k] == 1 and color[k]
                                    == current_color:
                                    can_color = False
                                k += 1

                            if can_color:
                                color[v] = current_color

                current_color += 1

        num_colors = current_color

        print("Graph colored using {} colors:".format(num_colors))
        for i in range(self.vertices):
            print("Vertex {}: Color {}".format(i, color[i]))

        G_viz = nx.Graph()
        for i in range(self.vertices):
            for j in range(i+1, self.vertices):
                if self.adjMatrix[i][j] == 1:
                    G_viz.add_edge(i, j)

        color_map = ['red', 'blue', 'green', 'yellow', 'orange', 'purple',
            'pink', 'brown']
        node_colors = []
        for i in range(self.vertices):
            node_colors.append(color_map[color[i] % len(color_map)])

        plt.figure(figsize=(8, 6))
        nx.draw(G_viz, with_labels=True, node_color=node_colors,
            node_size=800, font_size=14,
                font_weight='bold', edge_color='gray', width=2)
        plt.title("Welsh-Powell : {} couleurs".format(num_colors),
            fontsize=14)
        plt.show()

        return num_colors
```

3 Analyse de la complexité

La fonction utilise `rand()` afin de générer le nombre d'arêtes. La fonction `rand()` fournit un nombre aléatoire, puis nous appliquons le modulo 2 afin d'obtenir uniquement les valeurs 0 ou 1. En effet, pour un modulo n, le reste est compris entre 0 et $n-1$.
Voici notre fonction de génération de graphe :

```
Graphe *genererGraphe(int nb_sommets) {
    Graphe *graphe = creerGraphe(nb_sommets);
    for (int i = 0; i < nb_sommets; i++) {
        for (int j = i + 1; j < nb_sommets; j++) {
            if (rand() % 2) {
                ajouterArete(graphe, i, j);
            }
        }
    }
    return graphe;
}
```

Le fonctionnement de cette fonction se fait en trois étapes :

1. Création du graphe avec la fonction `creerGraphe()` , qui initialise notamment la matrice d'adjacence avec des zéros.
2. Ajout des arêtes grâce à la fonction `ajouterArete()`. Une arête n'est ajoutée que si `rand() % 2` renvoie 1, ce qui correspond à une chance sur 2. La double boucle `for` permet de tester chaque paire de sommets sans répéter les boucles.
3. Retour du graphe généré.

La probabilité d'ajout d'une arête étant fixée arbitrairement à 50%, la densité du graphe peut être élevée. Pour un graphe peu dense, on pourrait par exemple utiliser 10% de chance d'ajout.
Un exemple d'utilisation serait :

```
Graphe *g = genererGraphe(10);
```

Exemples d'utilisation

Pour analyser le temps de réponse de l'algorithme de Welsh-Powell sous Windows, nous utilisons la commande `Measure-Command` de PowerShell. Le programme prend en argument le nombre de sommets du graphe. Par exemple, pour exécuter notre programme avec 1000 sommets et mesurer son temps d'exécution :

```
PS C:\Users\ali-d\Desktop\C\tp_3> Measure-Command { .\bin\benchmark.exe 1000 }
```

FIGURE 3 – Utilisation de la commande Measure-Command

Résultats pour différents nombres de sommets

Avec 10 sommets :

```
PS C:\Users\ali-d\Desktop\C\tp_3> Measure-Command { .\bin\benchmark.exe 10 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 7
Ticks             : 76057
TotalDays         : 8,80289351851852E-08
TotalHours        : 2,11269444444444E-06
TotalMinutes      : 0,000126761666666667
TotalSeconds      : 0,0076057
TotalMilliseconds : 7,6057
```

FIGURE 4 – Welsh-Powell avec 10 sommets

Avec 100 sommets :

```
PS C:\Users\ali-d\Desktop\C\tp_3> Measure-Command { .\bin\benchmark.exe 100 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 7
Ticks             : 76167
TotalDays         : 8,815625E-08
TotalHours        : 2,11575E-06
TotalMinutes      : 0,000126945
TotalSeconds      : 0,0076167
TotalMilliseconds : 7,6167
```

FIGURE 5 – Welsh-Powell avec 100 sommets

Avec 1000 sommets :

```
PS C:\Users\ali-d\Desktop\C\tp_3> Measure-Command { .\bin\benchmark.exe 1000 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 66
Ticks             : 669161
TotalDays         : 7,74491898148148E-07
TotalHours        : 1,85878055555556E-05
TotalMinutes      : 0,00111526833333333
TotalSeconds      : 0,0669161
TotalMilliseconds : 66,9161
```

FIGURE 6 – Welsh-Powell avec 1000 sommets

Avec 2000 sommets :

```
PS C:\Users\ali-d\Desktop\C\tp_3> Measure-Command { .\bin\benchmark.exe 2000 }


Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 334
Ticks             : 3345632
TotalDays         : 3,87225925925926E-06
TotalHours        : 9,29342222222222E-05
TotalMinutes      : 0,00557605333333333
TotalSeconds      : 0,3345632
TotalMilliseconds : 334,5632
```

FIGURE 7 – Welsh-Powell avec 2000 sommets

Je vais par la suite seulement récapituler les autres tests dans mon tableau pour éviter la redondance. En récapitulant cela dans un tableau nous obtenons :

Nombre de sommets	Temps d'exécution
10	0.007 s
100	0.007 s
1000	0.065 s
2000	0.336 s
5000	3.540 s
6000	5.729 s
7000	8.972 s
8000	12.376 s
9000	19.397 s
10000	27.021 s
11000	36.429 s
12000	49.009 s
13000	59.866 s
14000	75.520 s
15000	89.558 s
16000	108.334 s
17000	126.728 s
18000	152.731 s
19000	180.598 s
20000	203.022 s

TABLE 1 – Récapitulatif des temps de calcul pour différents nombres de sommets

Courbe de complexité

La courbe obtenue suit approximativement $O(N^3)$, ce qui indique que la complexité de l'algorithme de Welsh-Powell est polynomiale par rapport au nombre de sommets du graphe.

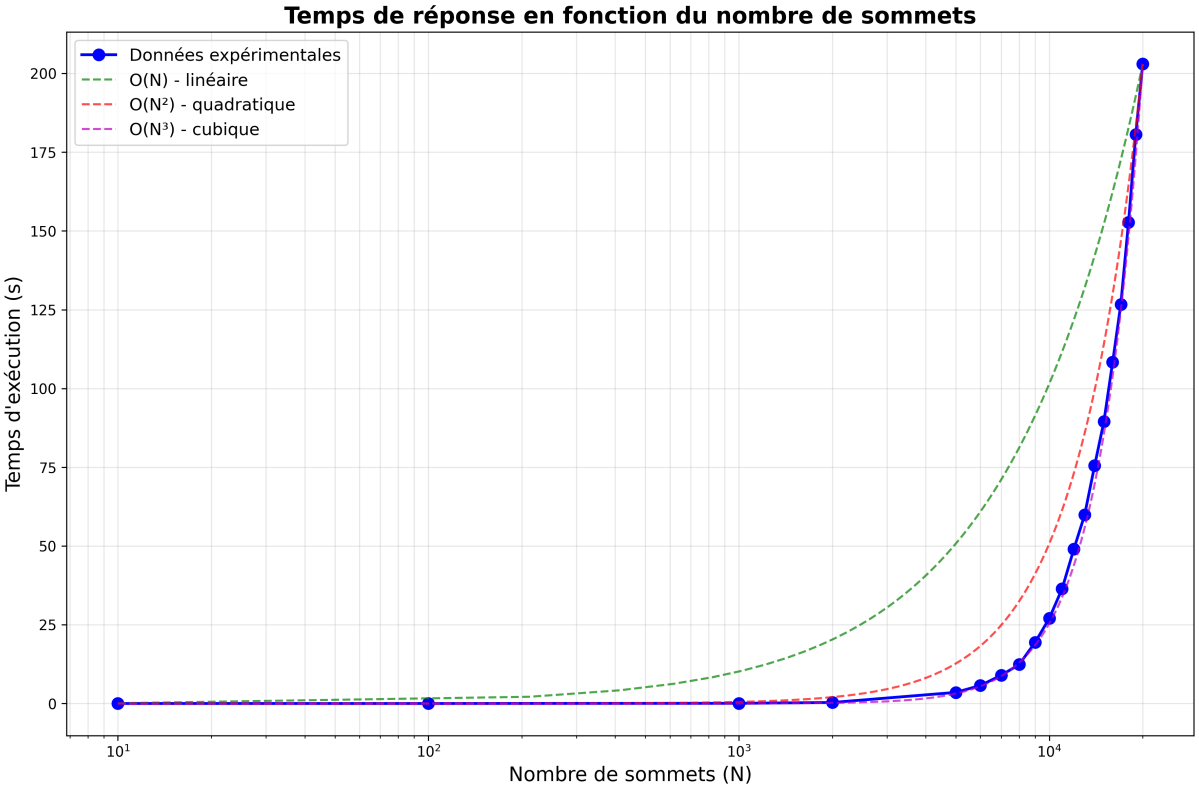

FIGURE 8 – Courbe de complexité