République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Rapport de Travaux Pratiques

Module : TPRO

TP1 : Le Plus Court Cycle dans un Graphe Non Orienté

Réalisé par le binôme :

ATTIA Oussama Abderraouf
SRAICH Imene

Groupe : SIL 1

Année Universitaire : 2025 - 2026

Table des matières

1 Introduction **2**

2 Approche Algorithmique **2**
2.1 Principe 2
2.2 Structure de Données 2

3 Implémentation et Code **2**
3.1 Détection de cycle (BFS_cycle) 2
3.2 Fonction Principale 3

4 Étude de la Complexité **3**
4.1 Complexité Temporelle 3
4.2 Complexité Spatiale 3

5 Interface Graphique avec CONIO2 **3**
5.1 Fonctionnalités utilisées 4

6 Exemple d'Exécution **4**
6.1 Graphe Testé 4
6.2 Affichage Console 4

7 Conclusion **4**

1 Introduction

Ce TP a pour objectif de concevoir et implémenter un algorithme capable de trouver la longueur du plus court cycle (aussi appelé la *maille* du graphe) dans un graphe non orienté.

Nous avons utilisé le langage **C** pour l'implémentation logique et la bibliothèque `conio2.h` pour créer une interface graphique utilisateur (GUI) conviviale dans la console.

2 Approche Algorithmique

Pour trouver le plus court cycle dans un graphe non pondéré, la méthode la plus efficace consiste à utiliser un parcours en largeur (**BFS - Breadth-First Search**).

2.1 Principe

Un cycle est détecté lors d'un parcours BFS si l'on rencontre un sommet déjà visité (ou en cours de visite) qui n'est pas le parent direct du sommet actuel.

Puisque nous cherchons le cycle le plus court **globalement**, un seul BFS à partir d'un sommet arbitraire ne suffit pas (car le plus court cycle pourrait ne pas être accessible rapidement depuis ce sommet). L'algorithme procède donc ainsi :

1. Lancer un BFS à partir de **chaque sommet** s du graphe.
2. Calculer la longueur du cycle détecté : $Long = distance[v] + distance[w] + 1$.
3. Garder la valeur minimale trouvée parmi tous les BFS.

2.2 Structure de Données

Nous représentons le graphe sous forme de **listes d'adjacence** pour optimiser l'espace mémoire.

```c
typedef struct Node {
    int val;            // Sommet voisin
    struct Node* next; // Pointeur vers le voisin suivant
} Node;
```

3 Implémentation et Code

Voici les parties clés de notre programme.

3.1 Détection de cycle (BFS_cycle)

Cette fonction effectue un parcours BFS depuis une source donnée. Nous utilisons trois états pour les sommets :

— `NON_VISITE (0)` : Sommet jamais atteint.
— `EN_COURS (1)` : Sommet dans la file d'attente.
— `VISITE (2)` : Sommet traité.

```c
// Extrait de la boucle BFS
if (etat[w] == NON_VISITE) {
    etat[w] = EN_COURS;
    distance[w] = distance[v] + 1;
    parent[w] = v;
    queue[tail++] = w;
}
else if (etat[w] == EN_COURS && parent[v] != w) {
    // Cycle detecte
    int cycle_length = distance[v] + distance[w] + 1;
    if (cycle_length < best)
        best = cycle_length;
}
```

3.2 Fonction Principale

La fonction `shortestCycle` appelle simplement `BFS_cycle` pour chaque nœud i allant de 0 à $N-1$ et retourne le minimum global.

4 Étude de la Complexité

4.1 Complexité Temporelle

— Un parcours BFS classique sur un graphe représenté par liste d'adjacence coûte $O(S+A)$, où S est le nombre de sommets et A le nombre d'arêtes.

— Dans notre solution, nous exécutons ce BFS à partir de chaque sommet (boucle de 0 à $S-1$).

La complexité totale est donc :

$$Complexit = O(S \times (S+A))$$

Dans le pire des cas (graphe dense où $A \approx S^2$), cela approche $O(S^3)$. Pour des graphes clairsemés, c'est très efficace.

4.2 Complexité Spatiale

Nous stockons le graphe ($O(S+A)$) et quelques tableaux auxiliaires pour le BFS (`etat, distance, parent`) de taille S.

$$Espace = O(S+A)$$

5 Interface Graphique avec CONIO2

Pour améliorer l'expérience utilisateur et répondre aux exigences du TP, nous avons utilisé la bibliothèque `conio2.h`. Elle nous permet de manipuler la console comme une interface graphique simple.

5.1 Fonctionnalités utilisées

— **Couleurs :** Utilisation de `textcolor()` et `textbackground()` pour différencier les sections (bleu pour le fond, jaune pour les titres, rouge pour le résultat).
— **Cadres :** Dessin de boîtes avec les caractères ASCII étendus (codes 201, 187, 205, etc.) pour encadrer le titre et les résultats.
— **Positionnement :** La fonction `gotoxy(x, y)` place le texte précisément à l'écran pour afficher proprement la liste d'adjacence et la complexité.

6 Exemple d'Exécution

6.1 Graphe Testé

Le programme crée le graphe suivant (6 sommets) :
— Cycle externe : $0 - 1 - 2 - 3 - 4 - 5 - 0$ (longueur 6)
— Cordes ajoutées : $1 - 4$ et $2 - 4$
Cela crée plusieurs cycles, notamment :
— Cycle $1 - 2 - 4 - 1$ (Longueur 3)
— Cycle $2 - 3 - 4 - 2$ (Longueur 3)

6.2 Affichage Console

Lors de l'exécution, l'écran affiche :

1. Le cadre bleu avec le titre du TP.
2. La structure du graphe (liste des voisins pour chaque sommet).
3. La complexité théorique calculée.
4. Une invite "Appuyez sur une touche...".
5. **Résultat Final :** Une boîte rouge affichant :

RESULTAT : Plus court cycle = 3 aretes

7 Conclusion

Ce TP nous a permis de manipuler les algorithmes de parcours de graphes dans un contexte pratique. L'utilisation de BFS itératif est une méthode robuste pour trouver la maille d'un graphe non pondéré.

L'intégration de la bibliothèque `conio2` a ajouté une dimension professionnelle à l'application, permettant une présentation claire et structurée des résultats, dépassant le simple affichage séquentiel standard.