

# TPE 1 — Représentation des Graphes

## *Implémentation de la liste d'adjacence*

**Durée :** 2 à 3 heures  
**Rendu :** Graph.cpp + capture des tests passés  
**Travail :** Individuel ou en binôme

■ **Objectif** : Implémenter une classe **Graph** représentant un graphe non-orienté par liste d'adjacence. Votre code sera validé par une suite de **tests automatiques** avec 3 niveaux de difficulté.

### ■ Niveaux de validation

#### NIVEAU 1 — BASE (obligatoire — 12 pts)

Fonctions essentielles : constructeur, addEdge, display, order, size, degree

#### NIVEAU 2 — STANDARD (5 pts)

Fonctions intermédiaires : hasEdge, neighbors

#### NIVEAU 3 — AVANCÉ (3 pts) + BONUS

removeEdge (3 pts) + loadFromFile (BONUS +2 pts)

### ■ Rappel : Liste d'adjacence

#### ■ Concepts clés

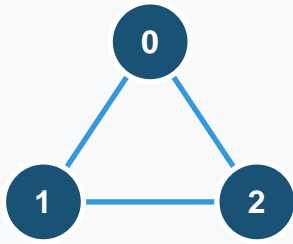
**Principe** : Chaque sommet stocke la liste de ses voisins.

**Graphe non-orienté** : Si  $(u,v)$  est une arête, alors  $v \in \text{voisins}(u)$  ET  $u \in \text{voisins}(v)$ .

**Degré  $d(v)$**  = nombre de voisins de  $v$  = taille de  $\text{adj}[v]$

**Formule fondamentale** :  $\sum \text{degrés} = 2 \times \text{nombre d'arêtes}$

### Exemple : Graphe Triangle



#### Liste d'adjacence :

0 → [ 1 , 2 ]

1 → [ 0 , 2 ]

2 → [ 0 , 1 ]

*Ordre = 3, Taille = 3*

*$\Sigma \text{ degrés} = 6 = 2 \times 3 \checkmark$*

#### ■■ NOTE IMPORTANTE — Gestion des doublons :

Dans ce TPE, **les arêtes multiples sont acceptées** (multi-graphe). Si vous appelez `addEdge( 0 , 1 )` deux fois, l'arête sera présente deux fois dans les listes.

**BONUS Niveau 3 :** Modifiez `addEdge( )` pour éviter les doublons en vérifiant avec `hasEdge( )` avant d'ajouter.

## ■ ■ Pièges courants à éviter

Ces erreurs sont commises par **plus de 50% des étudiants**. Lisez attentivement !

### Piège #1 : Oublier la symétrie (graphe non-orienté)

#### ■ FAUX :

```
adj[u].push_back(v); // Seulement une direction !
```

#### ■ CORRECT :

```
adj[u].push_back(v);  
adj[v].push_back(u); // Les DEUX directions !
```

### Piège #2 : Compter l'arête deux fois

#### ■ FAUX :

```
adj[u].push_back(v); n_edges++;  
adj[v].push_back(u); n_edges++; // Compte 2x !
```

#### ■ CORRECT :

```
adj[u].push_back(v);  
adj[v].push_back(u);  
n_edges++; // Une seule fois !
```

### Piège #3 : Virgule en trop dans display()

#### ■ FAUX :

```
for (int v : adj[u]) cout << v << ", ";  
// Affiche: 0 -> [1, 2, ] ← virgule en trop !
```

#### ■ CORRECT :

```
bool first = true;  
for (int v : adj[u]) {  
    if (!first) cout << ", ";  
    cout << v;  
    first = false;  
}
```

### Piège #4 : Ne pas vérifier les bornes

#### ■ FAUX :

```
int degree(int v) { return adj[v].size(); } // Crash si v invalide !
```

#### ■ CORRECT :

```
int degree(int v) {  
    if (v < 0 || v >= n_vertices) return -1;  
    return adj[v].size();  
}
```

## ■ Structure du projet

TPE1\_Graphes/

■■■ Graph.hpp	# Déclaration (fourni, NE PAS MODIFIER)
■■■ Graph.cpp	# Implémentation (À COMPLÉTER)
■■■ main.cpp	# Tests automatiques (fourni)
■■■ Makefile	# Compilation (fourni)
■■■ graphe_test.txt	# Fichier exemple pour loadFromFile()
■■■ README.md	# Instructions

## ■ Graph.hpp (fourni — ne pas modifier)

```
#ifndef GRAPH_HPP
#define GRAPH_HPP

#include <vector>
#include <list>
#include <string>
#include <iostream>

class Graph {
private:
    int n_vertices;           // Nombre de sommets
    int n_edges;              // Nombre d'arêtes
    std::vector<std::list<int>> adj; // Liste d'adjacence

public:
    // === NIVEAU 1 : BASE (12 pts) ===
    Graph(int n);             // Constructeur
    void addEdge(int u, int v); // Ajouter une arête
    void display() const;      // Afficher le graphe
    int order() const;         // Nombre de sommets
    int size() const;          // Nombre d'arêtes
    int degree(int v) const;    // Degré d'un sommet

    // === NIVEAU 2 : STANDARD (5 pts) ===
    bool hasEdge(int u, int v) const; // L'arête existe-t-elle ?
    std::vector<int> neighbors(int v) const; // Liste des voisins

    // === NIVEAU 3 : AVANCÉ (3 pts + bonus) ===
    void removeEdge(int u, int v); // Supprimer une arête
    bool loadFromFile(const std::string& filename); // BONUS +2pts
};

#endif
```

## ■ Graph.cpp (à compléter)

Les zones marquées **TODO** sont à compléter. Implémentez d'abord le **Niveau 1**, testez, puis passez aux niveaux suivants.

### Niveau 1 — Partie 1/2

```
#include "Graph.hpp"
#include <algorithm> // pour std::find
#include <fstream>   // pour fichiers (Niveau 3)

// =====
// NIVEAU 1 : BASE (12 points)
// =====

// Constructeur : crée un graphe avec n sommets (0 à n-1)
Graph::Graph(int n) : n_vertices(n), n_edges(0), adj(n) {
    // Déjà implémenté : adj(n) crée n listes vides
}

// Ajouter une arête non-orientée entre u et v
void Graph::addEdge(int u, int v) {
    // Vérification des bornes
    if (u < 0 || u >= n_vertices || v < 0 || v >= n_vertices) {
        std::cerr << "Erreur: sommet invalide" << std::endl;
        return;
    }

    // TODO 1: Ajouter v dans adj[u]

    // TODO 2: Ajouter u dans adj[v] (graphe NON-ORIENTÉ !)

    // TODO 3: Incrémenter n_edges (UNE SEULE FOIS !)
}
```

### Niveau 1 — Partie 2/2

```

// Afficher le graphe
void Graph::display() const {
    std::cout << "Graphe: " << order() << " sommets, "
               << size() << " aretes" << std::endl;

    for (int v = 0; v < n_vertices; v++) {
        std::cout << v << " -> [";

        // TODO 4: Afficher les voisins séparés par ", "
        // Astuce: utilisez un booléen "first" pour gérer la virgule

        std::cout << "]" << std::endl;
    }
}

// Retourner l'ordre (nombre de sommets)
int Graph::order() const {
    // TODO 5: Retourner n_vertices
    return 0;
}

// Retourner la taille (nombre d'arêtes)
int Graph::size() const {
    // TODO 6: Retourner n_edges
    return 0;
}

// Retourner le degré du sommet v
int Graph::degree(int v) const {
    if (v < 0 || v >= n_vertices) return -1;

    // TODO 7: Retourner la taille de adj[v]
    return 0;
}

```

## Niveau 2 — STANDARD

```
// =====
// NIVEAU 2 : STANDARD (5 points)
// =====

// Vérifier si l'arête (u,v) existe
bool Graph::hasEdge(int u, int v) const {
    if (u < 0 || u >= n_vertices || v < 0 || v >= n_vertices) {
        return false;
    }

    // TODO 8: Vérifier si v est dans adj[u]
    // Indice: std::find(adj[u].begin(), adj[u].end(), v) != adj[u].end()
    return false;
}

// Retourner la liste des voisins de v
std::vector<int> Graph::neighbors(int v) const {
    std::vector<int> result;
    if (v < 0 || v >= n_vertices) return result;

    // TODO 9: Copier adj[v] dans result
    // Indice: for (int voisin : adj[v]) result.push_back(voisin);

    return result;
}
```

## Niveau 3 — AVANCÉ + BONUS

```
// =====
// NIVEAU 3 : AVANCÉ (3 points + BONUS)
// =====

// Supprimer l'arête (u,v) - 3 points
void Graph::removeEdge(int u, int v) {
    if (!hasEdge(u, v)) return;

    // TODO 10: Retirer v de adj[u] et u de adj[v]
    // Indice: adj[u].remove(v);

    // TODO 11: Décrémenter n_edges
}

// BONUS (+2 points) : Charger depuis fichier
// Format: ligne 1 = n, puis chaque ligne = "u v"
bool Graph::loadFromFile(const std::string& filename) {
    // TODO 12 (BONUS): Ouvrir fichier, lire n, puis les arêtes
    // Voir graphe_test.txt pour le format
    return false;
}
```

## ■ Tests et validation

### Compilation et exécution

```
# Avec le Makefile (recommandé)
make clean
make
./test_graph

# Ou manuellement
g++ -std=c++11 -o test_graph Graph.cpp main.cpp
./test_graph
```

### ■ Sortie attendue (tous niveaux validés)

```
=====
TESTS TPE1 - THEORIE DES GRAPHS
=====

--- NIVEAU 1 : BASE ---
[TEST] Constructeur Graph(5)..... OK
[TEST] order() == 5..... OK
[TEST] size() == 0 (graphe vide)..... OK
[TEST] addEdge(0,1), addEdge(0,2)..... OK
[TEST] size() == 2..... OK
[TEST] degree(0) == 2..... OK
[...]
>>> NIVEAU 1 : 10/10 tests passés <<<

--- NIVEAU 2 : STANDARD ---
>>> NIVEAU 2 : 7/7 tests passés <<<

--- NIVEAU 3 : AVANCÉ ---
>>> NIVEAU 3 : 8/8 tests passés <<<

=====
RÉSULTAT FINAL : 25/25 tests passés
NOTE ESTIMÉE : 20/20
=====
```

## ■ Barème détaillé

Niveau	Tests	Points	Fonctions
NIVEAU 1	10 tests	12 pts	Graph(), addEdge(), display(), order(), size(), degree()
NIVEAU 2	7 tests	5 pts	hasEdge(), neighbors()
NIVEAU 3	8 tests	3 pts	removeEdge()
BONUS	—	+2 pts	loadFromFile()
TOTAL	25 tests	20 pts	(+ 2 bonus)

## ■ Checklist de rendu

Avant de soumettre, vérifiez chaque point :

- **make clean && make** compile sans erreur
- **./test\_graph** affiche au moins **10/10** (Niveau 1)
- Votre code est **indenté proprement**
- Votre **nom** est en commentaire en haut de Graph.cpp
- Vous avez créé le ZIP : **NOM\_Prenom\_TPE1.zip**
- Le ZIP contient : **Graph.cpp + capture d'écran** des tests
- La capture montre bien les tests passés (en couleur si possible)

### ■ Format de rendu :

- Fichier ZIP nommé `NOM_Prenom_TPE1.zip`
- Contenant : Graph.cpp + capture d'écran
- **Deadline** : Début de la Séance 2

## ■ Conseils pour réussir

1. **Procédez niveau par niveau** — Ne passez au suivant que quand tous les tests sont verts.
2. **Testez après chaque TODO** — Recompilez et relancez les tests fréquemment.
3. **Relisez les pièges courants** — 80% des erreurs sont listées page 2 !
4. **Dessinez sur papier** — Visualisez le graphe avant de coder.
5. **Utilisez graphe\_test.txt** — Pour tester loadFromFile() (bonus).