ESGC VERECHAGUINE A.K. | Année académique 2025-2026

Département Génie Informatique | Licence 2 (GI2)

Module : Théorie des Graphes | Enseignant : M. Alexandre

TPE 2 — Parcours en Largeur (BFS)

Implémentation de l'algorithme BFS et applications

Durée:	2-3h
Travail:	Individuel/Binôme
Rendu:	bfs.cpp + capture
Deadline:	Début Séance 3

■ Objectif

Implémenter l'algorithme de **Parcours en Largeur (BFS)** et ses applications. Le repository GitHub contient tout le code de base, les tests, et la documentation complète. Votre tâche : compléter le fichier **session2/tpe/src/bfs.cpp** avec les 7 fonctions demandées. **Consultez les slides de la Séance 2 (pages 6-20)** pour revoir l'algorithme BFS en détail.

■ **Repository GitHub :** https://github.com/shumpaga/graph-theory-course
■ **Dossier TPE2 :** session2/tpe/
■ **Documentation complète :** README.md, docs/ENONCE.md, docs/BAREME.md

■ Arborescence du projet

```
graph-theory-course/
■■■ session2/tpe/
■■■ README.md # Instructions complètes
■■■ Makefile # Compilation (make, make test)
■■■ src/
■ ■■■ graph.h/cpp # ✓ Classe Graph (fournie)
■ ■■■ queue.h/cpp # ✓ Classe Queue (fournie)
■ ■■■ bfs.h # ✓ Signatures (fourni)
■ ■■■ bfs.cpp # ■■ À COMPLÉTER (7 fonctions)
■ ■■■ main.cpp # ✓ Programme interactif
■■■ tests/
■ ■■■ test_bfs.cpp # ✓ 22 tests automatiques
■■■ data/
■ ■■■ graph_simple.txt # 5 sommets, 6 arêtes
■ ■■■ graph_disconnected.txt # 10 sommets, 3 composantes
■ ■■■ graph_bipartite.txt # 6 sommets, biparti
■ ■■■ graph_cotonou.txt # 15 sommets, réseau Gozem
■■■ docs/
■■■ ENONCE.md # Spécifications détaillées
■■■ BAREME.md # Barème complet (27 pts)
```

■ Niveaux de validation

Niveau	Points	Fonctions (dans src/bfs.cpp)
BASE	10 pts	bfs(), bfs_distances(), print_bfs_tree()

STANDARD	10 pts	shortest_path(), is_connected(), count_components()
BONUS	5 pts	is_bipartite()
CODE	+2 pts	Code très bien commenté
TOTAL	27 pts	Note ramenée sur 25

■ Rappel BFS (voir Séance 2, slides 6-20)

Principe : Explorer le graphe **niveau par niveau** avec une file FIFO (First In, First Out). BFS garantit le **plus court chemin** en nombre d'arêtes. **Structures clés :** Queue Q (file), visited[] (sommets visités), distance[] (distances minimales), parent[] (reconstruction chemins). **Complexité :** O(V+E) temps, O(V) espace.

■ Les 7 fonctions à implémenter

Ouvrez **src/bfs.cpp**. Chaque fonction contient des **TODO** numérotés avec instructions étape par étape. Consultez **docs/ENONCE.md** pour algorithmes détaillés.

#	Fonction	Description	Pts
1	`bfs(g, source)`	Parcours simple, affiche ordre de visite	3
2	`bfs_distances(g, source)`	Calcule distances minimales depuis source	3
3	`print_bfs_tree(g, source)`	Construit et affiche arbre BFS (parent → enfants)	4
4	`shortest_path(g, src, dest)`	Reconstruit plus court chemin entre deux sommets	3
5	`is_connected(g)`	Teste si le graphe est connexe	3
6	`count_components(g)`	Compte le nombre de composantes connexes	4
7	`is_bipartite(g)`	Teste si graphe est biparti (2-colorable)	5

■ Ce qui est fourni (NE PAS MODIFIER)

■ **graph.h/cpp** : Classe Graph complète avec liste d'adjacence
■ **queue.h/cpp** : Classe Queue FIFO (push, pop, front, empty)
■ **bfs.h** : Signatures des 7 fonctions BFS
■ **main.cpp** : Programme interactif avec menu pour tester
■ **tests/test_bfs.cpp** : 22 tests automatiques (8 BASE + 9 STANDARD + 5 BONUS)

■ Exemple de TODO dans bfs.cpp

```
void bfs(Graph& g, int source) {
// TODO 1 : Créer Queue Q et vector visited(V, false)
// TODO 2 : Marquer source visité : visited[source] = true
// TODO 3 : Enfiler source : Q.push(source)
// TODO 4 : Tant que Q non vide, défiler u, afficher u
// TODO 5 : Pour chaque voisin v non visité, marquer et enfiler
}
```

■ Pièges fréquents (60% des erreurs !)

Ces erreurs sont présentes dans **60% des copies**. Lisez attentivement !

#	Piège	Solution
1	Oublier de marquer visited AVANT enfiler	Toujours : visited[v] = true PUIS Q.push(v). Sinon le sommet peut être enfilé plusieurs fois !

2	Initialiser distance[] à 0 au lieu de -1	Utiliser vector int distance(V, -1) pour marquer "non atteignable". Seul source = 0.
3	Ne pas vérifier les bornes des sommets	Avant d'accéder à un sommet v : if (v < 0 || v >= V) return; sinon erreur segmentation.
4	Utiliser std::queue au lieu de Queue	Le TPE impose la classe Queue fournie (queue.h). Utiliser std::queue = -3 points de pénalité.
5	Ne pas gérer les sommets inatteignables	Si distance[v] reste à -1, le sommet n'est pas atteignable. Tester avant reconstruction chemin.

■■ **NOTE IMPORTANTE** — Utilisation de la classe Queue

Vous **DEVEZ** utiliser la classe **Queue** fournie dans queue.h/cpp. L'utilisation de **std::queue** de la STL entraînera une **pénalité de -3 points**. La classe Queue est déjà implémentée, testée, et prête à l'emploi avec les méthodes : push(x), pop(), front(), empty(). Consultez queue.h pour voir les signatures.

■ Format des fichiers graphes (data/)

Format : Ligne 1 = "V E directed", puis lignes "u v" (arêtes).
Exemple (graph_simple.txt) :

```
5 6 0     ← 5 sommets, 6 arêtes, non-orienté
0 1      ← Arête entre 0 et 1
0 2
1 3
2 3
2 4
3 4
```

Graphe résultant : Sommets {0,1,2,3,4} connectés selon les arêtes ci-dessus.

■ Graphes de test

Fichier	Contenu	Usage
graph_simple.txt	5 sommets, 6 arêtes, connexe	Tests BASE (fonctions 1-3)
graph_disconnected.txt	10 sommets, 8 arêtes, 3 composantes	Tests STANDARD (fonctions 5-6)
graph_bipartite.txt	6 sommets, 7 arêtes, 2-colorable	Test BONUS (fonction 7)
graph_cotonou.txt	15 stations réseau Gozem Cotonou	Validation complète (contexte réel)

■ Compilation et tests

```
# Compilation et tests automatiques
make clean
make test
./test_bfs

# Programme interactif (menu avec 7 fonctions)
make
./main data/graph_simple.txt

# Aide
make help
```

■ Sortie attendue des tests (22 tests)

```
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
TESTS TPE2 - BFS
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
--- NIVEAU 1 : BASE ---
✓ Test 2.1 - Graphe simple (depuis 0)
✓ Test 2.2 - Graphe simple (depuis 2)
...
>>> NIVEAU 1 : 8/8 tests passés <<<

--- NIVEAU 2 : STANDARD ---
>>> NIVEAU 2 : 9/9 tests passés <<<

--- NIVEAU 3 : BONUS ---
>>> NIVEAU 3 : 5/5 tests passés <<<

RÉSUMÉ : 22/22 tests - NOTE : 25/25 pts
```

■ Barème détaillé (voir docs/BAREME.md pour plus de détails)

Niveau	Fonction	Tests	Points	Critères principaux
BASE	bfs()	2	3	Queue utilisée, visited[] correct, ordre niveau par niveau
	bfs_distances()	3	3	distance[source]=0, autres=-1, calcul distance[v]=distance[u]+1
	print_bfs_tree()	3	4	Construction parent[], regroupement enfants, affichage correct
STANDARD	shortest_path()	3	3	Reconstruction avec parent[], gestion cas limites (source==dest)
	is_connected()	3	3	BFS depuis 0, comptage visités == V
	count_components()	3	4	BFS depuis chaque non-visité, comptage correct
BONUS	is_bipartite()	5	5	Coloration alternée 0/1, détection conflits, toutes composantes
CODE	Commentaires	—	+2	Explication algorithme, justification complexité, noms variables clairs

Total : 27 points (3+3+4+3+3+4+5+2) → Note ramenée sur 25 (note maximale = 20/20)

■ Checklist de rendu

■ Code compile sans erreur : make clean && make test
■ Tests automatiques passent (minimum 8/8 tests BASE)
■ Code bien commenté avec explications des étapes
■ Nom/Prénom en haut de bfs.cpp
■ ZIP créé : **NOM_Prenom_TPE2.zip**
■ ZIP contient : **bfs.cpp + capture_tests.png**
■ Capture montre les résultats de ./test_bfs

> **Format de rendu :** ZIP nommé NOM_Prenom_TPE2.zip contenant bfs.cpp + capture_tests.png
> **Deadline :** Début Séance 3 | **Soumission :** Via plateforme du cours

■ Conseils pour réussir

1. **Consultez README.md** — Documentation complète du projet
2. **Lisez docs/ENONCE.md** — Algorithmes détaillés pour chaque fonction
3. **Relisez les slides** — Séance 2 pages 6-20 (algorithme BFS) et 22-28 (applications)
4. **Procédez par niveau** — BASE (8 tests) → STANDARD (9 tests) → BONUS (5 tests)
5. **Testez fréquemment** — make test après chaque TODO complété
6. **Évitez les pièges** — Relire la liste des 5 pièges fréquents (page 3)

7. Programme interactif — Utilisez ./main pour tester manuellement

> **Bon courage !** Le code de base est sur GitHub. Les TODO dans src/bfs.cpp vous guident étape par étape. BFS est un algorithme fondamental — prenez le temps de bien le comprendre ! ■