Projet Graphes

Aurélien Teilhet, Bintang Alifseptiawan

Mai 2025

Sommaire

1 Introduction

On se propose de résoudre un problème de flot maximum à l'aide de l'algorithme de Dinic, une méthode efficace et bien adaptée aux graphes comportant un grand nombre d'arêtes.
Nous partons d'un flot initial nul et augmentons progressivement ce flot jusqu'à atteindre une situation où aucun chemin admissible du sommet source vers le puits n'existe dans le graphe de niveaux.
L'objectif est donc de déterminer la valeur maximale du flot pouvant transiter entre la source et le puits, tout en respectant les capacités imposées sur les arêtes.

2 Algorithme de DINIC

Le principe de l'algorithme de DINIC repose sur la recherche du plus court chemin dans le graphe d'écart créé depuis le graphe de départ.
Par la suite, on détermine l'amélioration possible de flot sur ce plus court chemin puis on itère ces 2 opérations jusqu'à ce qu'il n'existe plus de chemin entre la source et le puits.
On se propose d'utiliser l'algorithme de BFS (Breadth-First Search) afin de résoudre le problème du plus court chemin, en raison de la nature particulière du graphe d'écart : les arêtes y sont toutes de poids identique.

2.1 Déroulement de l'algorithme

On part du graphe de départ, on construit le graphe d'écart puis on effectue l'algorithme pour trouver le plus court chemin.

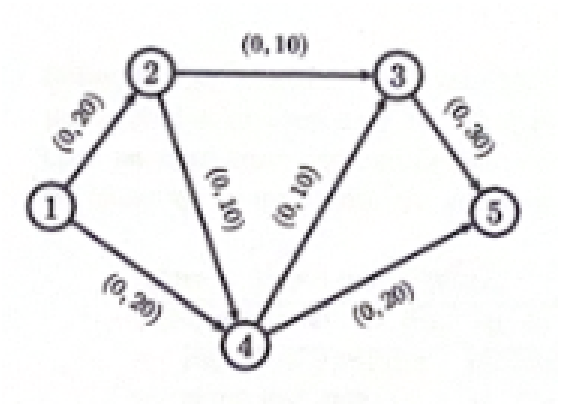

Figure 1: Réseau R1 de départ

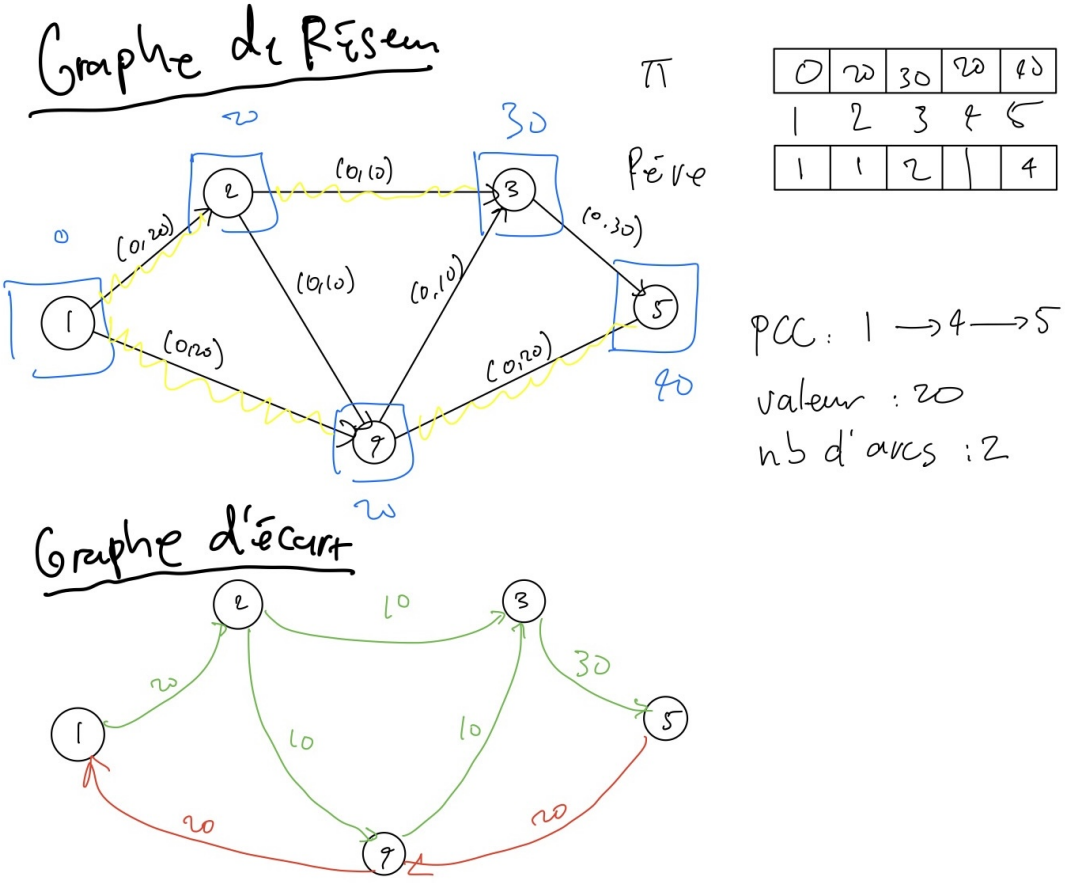

Figure 2: Résultat 1ère itération

On obtient le plus court chemin avec 3 arcs et une valeur de 20, donc on augmente dans le graphe de départ le flot de 20 sur ce plus court chemin.
On reconstruit le graphe d'écart à partir du graphe et on applique l'algorithme de Dijkstra sur ce graphe.
Pour les futures itérations, on ne détaillera plus le processus, restant le même.

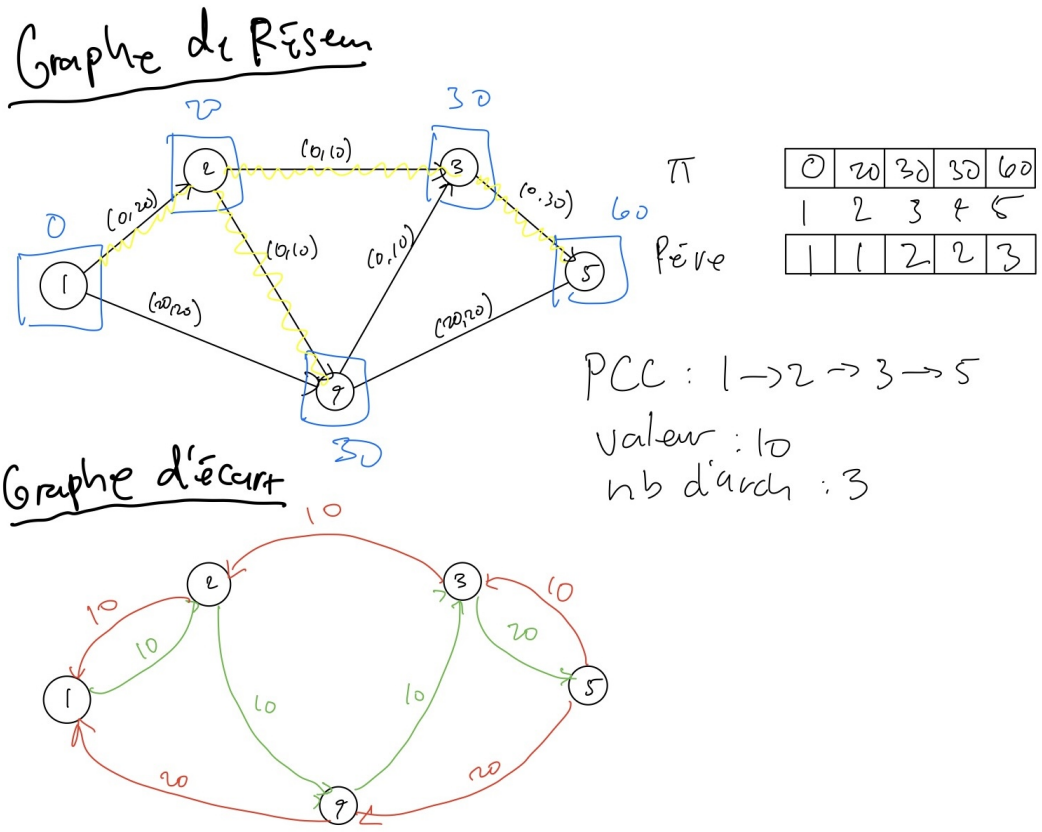

Figure 3: Résultat 2ème itération

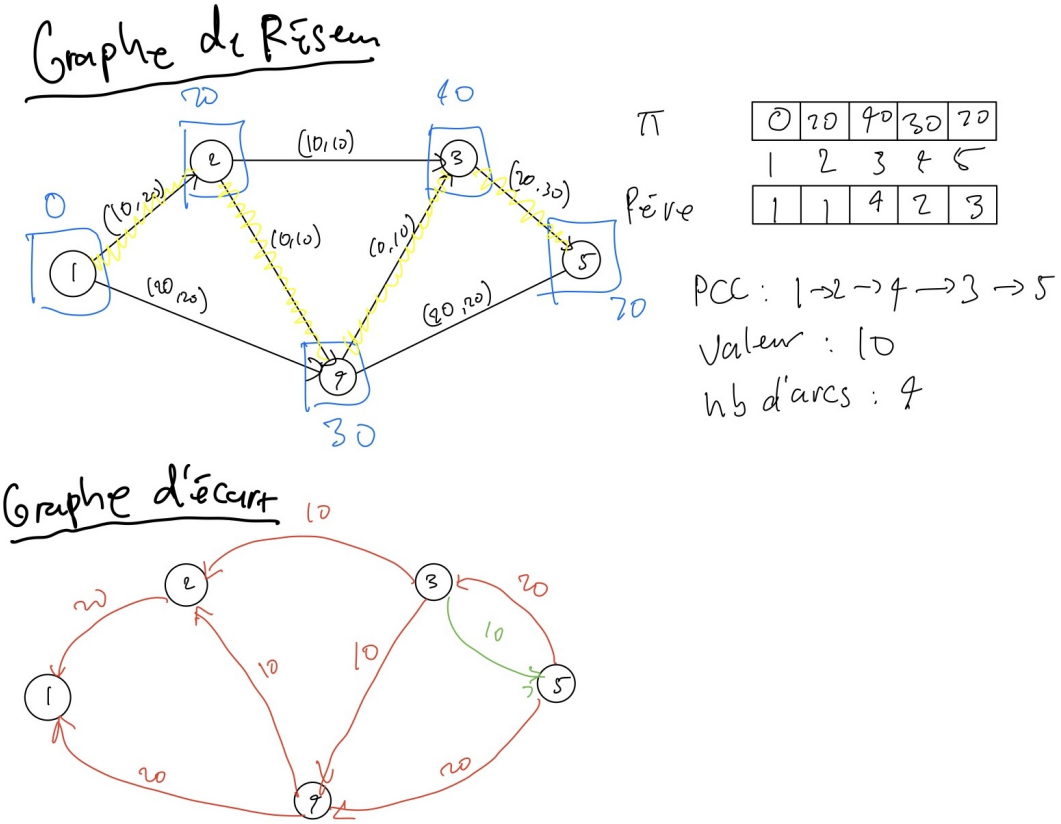

Figure 4: Résultat 3ème itération

On observe qu'il n'existe plus de chemin entre la source (sommet 1) et le puits (sommet 5) dans le graphe d'écart, donc on arrête l'algorithme.

On obtient un flot maximal d'une valeur de 40 et le graphe final suivant:

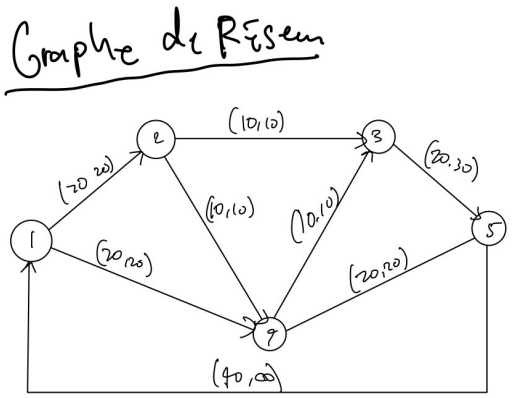

Figure 5: Résultat 4ème itération

3 Analyse des structures de données

Tous d'abord, nous allons commencer par analsyser la meilleur de strcuture de données pour le de graphe orienté.

3.1 Coût mémoire

Dans un graphe, on a n sommets et m arêtes. Pour un graphe creux, où le nombre d'arêtes m est beaucoup plus petit que le nombre maximum d'arêtes possible n(n-1)/2, on veut que la structure de données utilise la mémoire de manière efficace.

- Matrice d'incidence - O(n*m) : Une matrice d'incidence est un tableau 2D avec n lignes (pour les sommets) et m colonnes (pour les arêtes). Pour un graphe creux, cela peut être très inefficace en termes d'espace.
- Table des successeurs - O(m+n) : Dans cette représentation, on a un tableau de taille n pour les sommets, et chaque sommet pointe vers un tableau de ses successeurs. Pour un graphe creux, cela reste relativement efficace.
- Liste de successeurs - O(m+n) : Chaque sommet détient une liste de ses successeurs, typiquement une liste chaînée ou un tableau dynamique. Cette représentation est plus efficace en mémoire pour un graphe creux, car on ne stocke que les arêtes qui existent réellement.

Du point de vue mémoire, la table des successeurs et la liste de successeurs sont plus avantageuses.

3.2 Coût de traitement

Le coût de traitement dans ce contexte est lié à la complexité temporelle d'accès aux successeurs d'un sommet donné. Dans des algorithmes de flot maximum comme celui de DINIC, cela est crucial car, à chaque étape, on doit accéder rapidement aux voisins (successeurs) d'un sommet.

- Matrice d'incidence - O(n*m) : Pour trouver les voisins d'un sommet, il nous faut parcourir toute la ligne correspondante à ce sommet. Cela peut être lent.
- Table des successeurs - O(d)(d est le nombre d'arêtes sortantes du sommet) : Comme chaque sommet pointe vers un tableau de ses successeurs, l'accès aux voisins est rapide (temps constant pour accéder au tableau). Mais si on doit trouver un successeur spécifique, cela peut être coûteux.

- Liste de successeurs - O(d) : Chaque sommet pointe vers une liste (soit une liste chaînée, soit un tableau dynamique). L'accès aux voisins est relativement facile car chaque liste ne contient que les arêtes existantes.

En termes de coût de traitement, la Représentation par tableaux et la Représentation par listes de successeurs sont assez efficaces pour accéder aux voisins, avec un coût d'accès de O(d) dans les deux cas. Cependant, la Représentation par listes de successeurs reste légèrement plus flexible et efficace en pratique pour les graphes creux.

3.3 Impact des ajouts/suppressions

Lorsque l'on travaille avec des graphes dynamiques, notamment dans des algorithmes comme celui de DINIC qui modifient le graphe (ajoutant ou supprimant des arêtes pendant l'exécution), la manière dont la structure de données gère ces opérations devient importante.

- Matrice d'incidence - O(n) : Ajouter ou supprimer une arête nécessite de modifier la matrice, ce qui implique de mettre à jour des lignes et des colonnes spécifiques. Cette opération est relativement lente car on doit peut-être parcourir toute la matrice.

- Table des successeurs : Ajouter ou supprimer une arête nécessite d'ajuster le tableau des successeurs pour le sommet correspondant. Pour les graphes creux, cette opération est relativement rapide. Dans le cas où les structures est ordonnées, table des surccesseurs coût O(log d).

- Liste de successeurs : La liste des successeurs de chaque sommet doit être modifiée lorsque des arêtes sont ajoutées ou supprimées. Dans le cas d'une liste chaînée, ajouter une arête est une opération O(1) car on ajoute en tête. Cependant, la suppression d'une arête est une opération O(d) car il faut chercher l'arête dans la liste.

En termes de gestion des ajouts et suppressions d'arêtes, la Représentation par listes de successeurs est la plus efficace. La représentation matricielle serait la plus lente dans ce cas, en particulier pour les graphes dynamiques.

3.4 Implémentation du réseau et du graphe d'écart avec la représentation par listes de successeurs

3.4.1 Représentation du réseau (Graphe original)

Le réseau est un graphe dans lequel on a des sommets reliés par des arêtes. Chaque arête a une capacité et un flot courant associé. L'idée est de modéliser ce graphe de

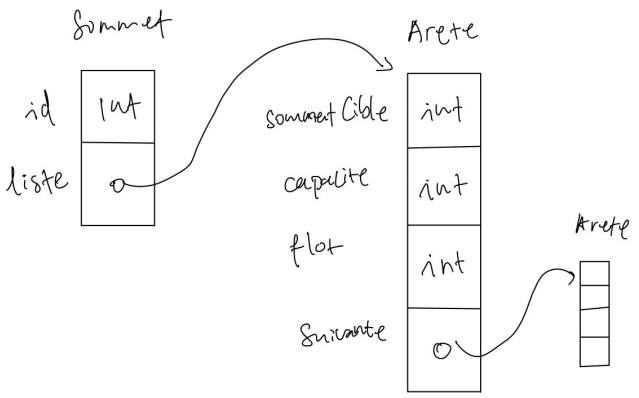

Figure 6: Illustration de la graphe de réseau

manière à pouvoir ajouter, supprimer ou modifier les arêtes pendant l'algorithme de flot maximal. Type de structure :

- Le graphe est représenté par un **tableau de pointeurs vers les sommets**.
- Chaque sommet contient une **liste chaînée d'arêtes sortantes**.

3.4.2 Représentation du graphe d'écart (résiduel)

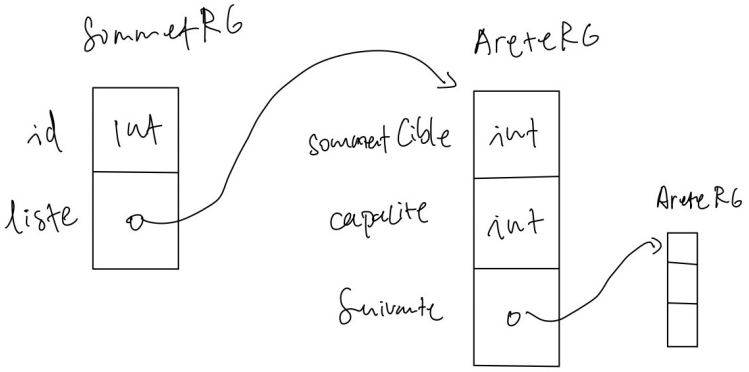

Figure 7: Illustration de la graphe d'écart

Même principe que pour le graphe original, sauf que les arêtes peuvent exister dans les deux sens (direct et retour) et changent à chaque mise à jour.

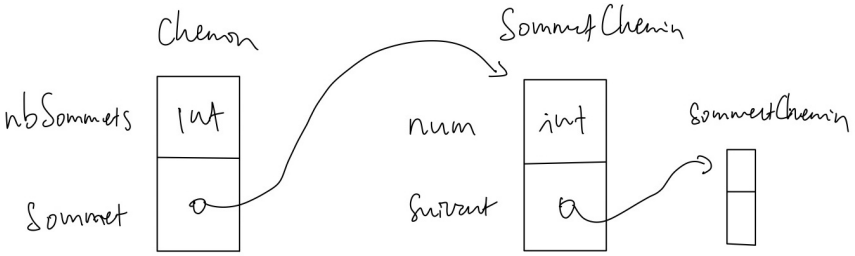

Figure 8: Illustration du chemin augmentant

3.4.3 Représentation du chemin augmentant (liste de sommets)

Un chemin est une liste chaînée simple des identifiants de sommets du chemin trouvé entre la source et le puits dans le graphe résiduel.

4 Décomposition de l'algorithme de DINIC

4.1 Structure de données choisie

Nous retenons la présentation par listes de successeurs pour représenter le réseau et le graphe d'écart associé.

Nous utiliserons la structure suivante pour représentation un chemin:

```

struct Sommet {
    int num;         // sommet destination
    Sommet* suivant;     // pointeur vers le successeur
}

struct Chemin {
    int nbSommets;
    Sommet* sommet;
}
```

4.2 Procédures

```
Action buildRG(grapheDepart,  grapheEcart)
D: grapheDepart: Graphe
R: grapheEcart: Graphe
Initialiser grapheEcart avec les memes sommets que grapheDepart
Pour chaque sommet s de grapheDepart:
    Pour chaque successeur succ de s:
        Ajouter un successeur de s dans grapheEcart
        avec poids = capacite - flux
        Ajouter un predecesseur de s dans grapheEcart
        avec poids = flux
```

```
Fonction shortestPast(grapheOriente, s, p)
D: grapheOriente: Graphe
    s, p: Sommet
R: cheminMin : Chemin
Initialiser Mark[x] = Faux, d[x]=infini, Pere[x] = -1
pour tout x sommet de grapheOriente
d[s] = 0
Pere[s] = s

```

```
Tant qu'il existe x non marque avec d[x] minimal:
    Marquer x
    Pour chaque voisin y de x:
        Si d[y] > d[x] + 1:
            d[y] = d[x] + 1
            Pere[y] = x
Reconstituer le chemin de s a p avec Pere dans cheminMin
Retourner cheminMin
```

```
Fonction minCapa(graphe, chemin)
D: graphe: Graphe
    chemin: Chemin
R: minCap: Entier
cheminTmp = chemin->sommet
minCap = la capacite du 1er sommet du chemin
Tant que cheminTmp != null:
    minCap = min(minCap, la capacite de cheminTmp)
    cheminTmp = cheminTmp->suivant
Retourner minCap
```

```
Action updateFlowInRG(grapheEcart, chemin, k)
D: chemin: Chemin
    k: Entier
D/R: grapheEcart: Graphe
Initialiser Mark[x] = Faux
pour tout sommet x de grapheEcart
Pour chaque sommet s de grapheEcart:
    Pour chaque successeur succ de s:
        Si Mark[succ] = Faux:
            Augmente le poids de k sur l'arete qui va
            du predecesseur au successeur de grapeEcart
            Diminiue le poids de k sur l'arete qui va
            du successeur au predecesseur de grapheEcart
            Mark[succ] = Vrai
```

```
Action updateFlowInNet(grapheEcart, reseau)
D: grapheEcart: Graphe
D/R: reseau: Graphe
Pour chaque sommet s de reseau:
    Lire le poids du successeur de s vers s
    Le flot du sommet s devient egal a cette valeur
```

4.3 Algorithme principal

```
Fonction main(fichier)
D: fichier: Fichier DIMACS
chemin: Chemin
reseau = buildGraph(fichier)

grapheEcart = buildRG(reseau)

flotTotal := 0

Tant que chemin = shortestPath(grapheEcart, s, p) existe :
    // trouver la capacite minimale sur ce chemin de s a p
    k = minCapa(grapheEcart, chemin)

    updateFlowInRG(grapheEcart, chemin, k)
    flotTotal = flotTotal + k

updateFlowInNet(grapheEcart, reseau)
ecrireResultats(flotTotal, reseau)

Retourner 1
```

5 Mode d'emploi du programme

5.1 Objectif du programme

Ce programme implémente l'algorithme de Ford-Fulkerson à l'aide d'un parcours en largeur (BFS) pour déterminer le flot maximum entre une sourec et un puits dans un graphe orienté pondéré par des capacités. Il s'appuie sur :

- Un réseau d'origine (le graphe initial);
- Un graphe résiduel (qui évolue au fil des mises à jour des flots);
- Une recherche de chemins augementants (untilisés pour augementer le flot total).

5.2 Structure des fichiers du projet

Dans le réndu du programme, il y a des des fichers en format .c et .h. Il existe également les fichers supports. Voici la définition de chaque fichier :

Fichier	**Rôle**
main.c	Point d'entrée : lit le graphe, exécute l'algorithme, affiche le résultat.
graphe.c/h	Gestion du graphe original (lecture DIMACS, affichage, libération mémoire).
grapheRG.c/h	Construction et gestion du graphe résiduel.
chemin.c/h	Recherche de chemins augmentants (BFS), mise à jour des flots.
constante.h	Contient la constante globale `MAX_SOMMETS`.
README.md	Description du projet et instructions.
data/	Fichiers de test au format DIMACS.

5.3 Déroulement de l'algorithme

L'algorithme implémenté est une version du **Ford-Fulkerson** utilisant un **parcours en largeur (BFS)** pour identifier les *chemins augmentants* dans un **graphe résiduel**. L'objectif est de maximiser le flux circulant entre une source et un puits, sous contraintes de capacités sur les arcs.

Le programme effectue les étapes suivantes :

1. **Lecture du graphe :** Le graphe est lu depuis un fichier au format **DIMACS**, contenant la liste des sommets, des arêtes et les informations sur la source et le puits. Cette étape est réalisée par la fonction `buildGraphe()`.

2. **Construction du graphe résiduel :** À partir du graphe original, un graphe résiduel est créé avec les capacités restantes sur chaque arc. Si une arête $(u \to v)$ a une capacité c et un flot actuel f, deux arêtes sont ajoutées dans le graphe résiduel :

   - $(u \to v)$ avec capacité $c - f$ (capacité directe),
   - $(v \to u)$ avec capacité f (capacité retour).

   Cette étape est effectuée par `buildRG()`.

3. **Recherche d'un chemin augmentant :** À l'aide d'un **BFS**, le programme recherche un chemin de la source au puits dans le graphe résiduel, ne passant que par des arcs de capacité strictement positive. Si un tel chemin existe, il est retourné sous forme d'une liste chaînée. Fonction : `shortestPathRG()`.

4. **Calcul de la capacité minimale du chemin :** La plus petite capacité rencontrée le long du chemin déterminera combien d'unités de flux peuvent être ajoutées dans ce cycle. Fonction : `minCapa()`.

5. **Mise à jour du graphe résiduel :** Le graphe résiduel est modifié en fonction du flux injecté k :

   - L'arête directe $(u \to v)$ voit sa capacité réduite de k.
   - L'arête retour $(v \to u)$ est augmentée de k (ou créée si absente).

   Fonction : `updateFlowInRG()`.

6. **Mise à jour du graphe original :** Le graphe de base est modifié pour refléter le nouveau flot :

   - L'arête $(u \to v)$ voit son flot augmenté de k.
   - Si le flot passe en sens inverse, l'arête $(v \to u)$ est décrémentée.

   Fonction : `updateFlowInNet()`.

7. **Répétition jusqu'à saturation :** Les étapes 2 à 6 sont répétées jusqu'à ce qu'aucun chemin augmentant ne soit trouvé. Cela signifie que le flot maximum a été atteint.

8. **Affichage du résultat :** Le flot total est affiché à la fin du programme. Exemple : `Flot maximum trouvé : 42`

5.4 Fonctionnalités principales

Le programme repose sur une architecture modulaire, où chaque fonctionnalité est implémentée dans un fichier source dédié. Les fonctions principales sont regroupées selon leur rôle dans l'algorithme de flot maximum.

Fonction	Description
`buildGraphe()`	Lit un fichier DIMACS et construit le graphe original avec les sommets, les arêtes et les capacités.
`buildRG()`	Construit le graphe résiduel à partir du graphe original et des flots actuels.
`shortestPathRG()`	Recherche un chemin augmentant (de la source au puits) dans le graphe résiduel en utilisant un BFS.
`minCapa()`	Calcule la capacité minimale disponible sur un chemin augmentant, utilisée pour déterminer le flot injecté.
`updateFlowInRG()`	Met à jour le graphe résiduel en fonction du flot injecté sur un chemin augmentant.
`updateFlowInNet()`	Met à jour le graphe original en modifiant les flots selon le chemin utilisé.
`afficherGraphe()`	Affiche le graphe original avec les arêtes, capacités et flots actuels (utile pour le débogage).
`afficherGrapheRG()`	Affiche le graphe résiduel et ses capacités restantes.
`libererGraphe()`	Libère la mémoire allouée au graphe original à la fin de l'exécution.
`libererGrapheRG()`	Libère la mémoire du graphe résiduel.
`libererChemin()`	Libère la mémoire d'un chemin augmentant après traitement.

5.5 Nettoyage mémoire automatique

Le programme alloue dynamiquement de la mémoire pour représenter les sommets, arêtes, graphes et chemins. Afin de garantir une exécution propre et sans fuite mémoire, un nettoyage systématique est effectué à la fin de l'algorithme.

Les fonctions suivantes sont appelées pour libérer l'ensemble des structures utilisées :

Fonction	Description
`libererGraphe()`	Libère la mémoire allouée pour le graphe original, en supprimant chaque arête de chaque sommet, puis les sommets eux-mêmes.
`libererGrapheRG()`	Libère le graphe résiduel de la même manière que pour le graphe original.
`libererChemin()`	Libère la mémoire d'un chemin augmentant (liste chaînée de sommets) après chaque itération.

Ce nettoyage est essentiel, notamment dans des cas où :

- l'algorithme parcourt de nombreux chemins (boucle principale) ;
- des erreurs empêchent l'atteinte de la fin du programme (ex. fichier manquant) ;
- le programme est testé avec des graphes de grande taille.

L'appel à ces fonctions est intégré dans la fonction `main()`, ce qui garantit une libération complète de la mémoire à la fin de l'exécution, même en cas d'arrêt prématuré.

6 Exemples traités et résultats obtenus

6.1 Format attendu du fichier d'entrée (DIMACS)

Le programme attend un fichier texte décrivant un graphe orienté au format **DIMACS**, standard en algorithmique des graphes. Chaque ligne du fichier commence par une lettre indiquant le type d'information :

- `p <nb_sommets> <nb_arêtes>` : définit le nombre total de sommets et d'arêtes. Cette ligne est obligatoire et doit apparaître en premier.
- `n <id> s` : indique que le sommet d'identifiant `id` est la source (`s`).
- `n <id> t` : indique que le sommet `id` est le puits (`t`).
- `a <u> <v> <capa>` : ajoute une arête orientée de `u` vers `v` avec une capacité entière strictement positive.
- `c ...` : lignes de commentaire, ignorées par le programme.

Exemple : extrait du fichier `net1.txt`

```
c Pb de flot max
p 5 7
n 1 s
n 5 t
a 1 2 20
a 1 4 20
a 2 3 10
a 2 4 10
a 3 5 30
a 4 3 10
a 4 5 20
```

—

6.2 Compilation du programme

Le programme est écrit en C, organisé en modules. Il nécessite un compilateur standard compatible GCC.

Pour compiler tous les fichiers manuellement, utiliser :

```
gcc main.c graphe.c grapheRG.c chemin.c -o main -Wall
```

- `-Wall` : active les avertissements du compilateur.
- `-o main` : nom de l'exécutable généré.

Conseil : ajouter `-O2` ou `-O3` pour optimiser les performances sur de grands graphes.

—

6.3 Exécution du programme

L'exécutable se lance avec un seul argument : le chemin vers un fichier DIMACS.

```
./main data/net1.txt
```

Si aucun argument n'est donné, un message d'erreur s'affiche :

```
Usage: fournir un fichier DIMACS en parametre
```

En cas d'erreur de format, le programme s'arrête et affiche un message explicite.

—

6.4 Exemples traités et résultats obtenus

Petit graphe
C'est l'exécution du programme pour l'exemple de données de `net1.txt`

- Description : Graphe à 5 sommets et 7 arêtes.
- Source : 1, Puits : 5
- Résultat affiché : `Flot maximum trouvé :  30`

Graphe intermédiaire
C'est l'exécution du programme pour l'exemple de données de `net2.txt`

- Description : Graphe à 6 sommets et 8 arêtes.
- Résultat affiché : `Flot maximum trouvé :  10`

Cas intensifs
C'est l'exécution du programme pour l'exemple de données de `G_100_300.max`, `G_900_2700.max`, `G_2500_7500.max`

- Fichiers utilisés pour des tests de performance.

- Temps d'exécution variable selon la taille :
  - `G_100_300.max` : < 1 seconde
  - `G_900_2700.max` : quelques secondes
  - `G_2500_7500.max` : plusieurs dizaines de secondes
- Le programme a pu calculer le flot maximum dans tous les cas sans erreur.

—

6.5 Conseils d'utilisation

- Toujours commencer les identifiants de sommets à `1`, jamais à `0`.
- Vérifier la syntaxe des fichiers DIMACS, notamment :
  - Une seule ligne `p` et une source/puits bien définies.
  - Aucune capacité nulle ou négative.
- Pour tester la stabilité mémoire :

```
valgrind ./main data/net2.txt
```

- Pour optimiser les performances :

```
gcc -O2 main.c graphe.c grapheRG.c chemin.c -o main
```

- Adapter la constante `MAX_SOMMETS` dans `constante.h` si nécessaire.

—

6.6 Remarques finales

Ce programme peut servir de base pour :

- Étudier l'efficacité de différents algorithmes de flot.
- Comparer BFS avec DFS pour les chemins augmentants.

La modularité du code permet de remplacer ou améliorer certains composants facilement (par exemple, stratégie de parcours, structure de données, lecture de formats différents).

7 Auto-évaluation

Dans le cadre de ce projet, nous avons veillé à répondre de manière complète et rigoureuse à l'ensemble des objectifs pédagogiques fixés. Nous proposons ci-dessous une auto-évaluation argumentée.

(O1) Rédaction du cahier des charges, présentation de l'analyse du sujet et de la méthode

Aurélien: 1 point & Bintang : 1 point

- Le cahier des charges a été fidèlement restitué dans l'introduction.
- L'analyse du sujet a été détaillée avec clarté, en présentant les étapes principales de la méthode : construction du graphe, génération du graphe résiduel, recherche de chemins augmentants, mise à jour des flots.
- Le choix de l'algorithme basé sur un parcours en largeur (BFS) dans le graphe résiduel a été justifié.

(O2) Analyse des structures de données

Aurélien: 1 point & Bintang : 1 point

- Toutes les structures de données utilisées ont été décrites dans le rapport : sommets, arêtes, graphes originaux et résiduels, chemins augmentants.
- Les choix de représentation (listes chaînées) ont été justifiés en réspectant le cours.

(O3) Rédaction des algorithmes

Aurélien: 1 point & Bintang : 1 point

- L'ensemble des algorithmes clés (buildGraphe, buildRG, shortestPath, minCapa, updateFlowInRG, updateFlowInNet) a été rédigé clairement et structuré dans le rapport.
- Chaque algorithme est accompagné d'une explication des structures de données mobilisées.
- Le pseudocode présenté correspond fidèlement aux fonctions C implémentées.

(O4) Résultats expérimentaux

Aurélien: 1 point & Bintang : 1 point

- Les exemples `net1.txt` et `net2.txt` ont été traités correctement avec un affichage du flot maximum attendu.
- Les trois grandes instances (G_100_300, G_900_2700, G_2500_7500) ont été exécutées avec succès, avec des résultats cohérents et stables.
- L'affichage final comporte à la fois le flot total et le flot sur chaque arc.

(O5) Développement et programmation

Aurélien: 1 point & Bintang : 1 point

- Le code source est proprement structuré en plusieurs modules (graphe, graphe résiduel, chemin, etc.).
- Chaque fonction est accompagnée d'un en-tête de documentation et des commentaires ont été ajoutés pour faciliter la lecture.
- Aucune sortie inutile n'est réalisée dans les calculs ; l'entrée du fichier est fournie via la ligne de commande ; un `Makefile` est disponible ; un fichier `README.md` explique le fonctionnement.

8 Conclusion

Ce projet a permis de concevoir et d'implémenter un programme complet en langage C pour résoudre le problème du **flot maximum** dans un graphe orienté, en utilisant l'algorithme de **Ford-Fulkerson** basé sur une recherche en largeur (`BFS`).

Travail réalisé :

- Lecture et construction du graphe original à partir de fichiers au format DIMACS.
- Construction dynamique du graphe résiduel à chaque itération.
- Recherche de chemins augmentants par BFS.
- Mise à jour des flots dans le graphe original et le graphe résiduel.
- Calcul automatique du flot maximum.
- Gestion propre de la mémoire dynamique (allocation et libération).
- Documentation technique du code et rédaction d'un mode d'emploi complet.
- Tests sur plusieurs fichiers d'exemples (simples et volumineux).

Limites actuelles :

- Le programme utilise une stratégie simple (BFS) pour les chemins augmentants, ce qui peut ralentir le traitement de très grands graphes.
- Aucun affichage visuel des graphes (via Graphviz ou autre).
- Pas d'interface interactive ou de paramétrage dynamique (ex. sélection de source/puits).
- Aucun module de vérification automatique de validité du fichier DIMACS.

Améliorations possibles :

- **Améliorer les messages d'erreur** : ajouter des vérifications pour détecter les erreurs de format dans le fichier d'entrée (ex. ligne mal écrite, capacité manquante).
- **Ajouter des commentaires dans le code** : pour mieux comprendre chaque fonction et faciliter la relecture par d'autres utilisateurs.

- **Permettre d'afficher le graphe** : écrire une fonction qui imprime le graphe sous forme lisible (par exemple : `Sommet 1 → Sommet 2 (capacité 10)`).
- **Afficher les chemins utilisés** : montrer les chemins trouvés pendant le calcul du flot, avec les valeurs de flux.
- **Proposer un menu interactif** : par exemple, après lancement du programme, proposer à l'utilisateur d'entrer un fichier, d'afficher le graphe, ou de lancer le calcul.
- **Mesurer le temps d'exécution** : utiliser la bibliothèque standard `time.h` pour afficher le temps que met l'algorithme à s'exécuter.
- **Créer un Makefile** : pour automatiser la compilation du projet au lieu d'écrire la commande manuellement.
- **Tester d'autres fichiers** : essayer le programme sur des graphes plus gros ou créés soi-même pour voir comment il réagit.

Ce projet constitue ainsi une base solide pour le traitement des problèmes de flots dans les graphes. Il est extensible à d'autres variantes du problème (flot de coût minimal, flot avec contraintes multiples) et peut servir de socle pour un projet de recherche ou d'application industrielle.