Projet Graphes pour l'optimisation

Algorithme de Dinic pour le problème du flot maximum

Rapport Final

Vendredi 23 Mai 2025

Alexis Evaristo
Imane Mahmoud

Table des matières

1 Introduction

Ce projet a pour objectif de concevoir et d'implémenter une solution efficace au problème du flot maximum en réseau, en s'appuyant sur l'algorithme de Dinic. Cet algorithme repose sur l'exploitation successive de chemins de niveau dans des graphes d'écart.

Le problème du flot maximum consiste, à partir d'un réseau orienté avec capacités sur les arcs, à déterminer le débit maximal pouvant être transféré d'une source **S** vers un puits **P** sans violer les contraintes de capacité. Cette problématique se retrouve dans de nombreux domaines tels que l'optimisation des réseaux de transport, la logistique, ou encore les télécommunications.

Notre approche a été structurée en plusieurs étapes :

- Analyse théorique de l'algorithme de Dinic et de la notion de graphe d'écart,
- Choix de la structure de données pour représenter le réseau et optimiser les accès aux successeurs,
- Décomposition de l'algorithme en modules fonctionnels, chacun chargé d'une tâche précise (construction du graphe, recherche de chemin, mise à jour du flot),
- L'implémentation a été d'abord validée sur les réseaux R1 et R2 au format DIMACS, avant d'être testée sur des instances de plus grande taille pour évaluer ses performances.

Le projet impose également une réflexion sur les coûts mémoire et coûts de traitement, en comparant différentes représentations des graphes (matrice d'adjacence, listes de successeurs, etc.). Cette analyse a guidé la conception de notre solution afin de garantir un compromis optimal entre efficacité et simplicité d'implémentation.

Ce rapport présente dans un premier temps le contexte et la description du sujet, suivi de l'analyse des structures de données retenues. Ensuite, la conception de l'algorithme est détaillée, accompagnée d'un mode d'emploi de l'application. Enfin, nous présentons les résultats obtenus sur les exemples fournis ainsi que notre bilan personnel sur le projet.

2 Description de l'algorithme de DINIC

2.1 Algorithme de recherche de plus court chemin en nombre d'arcs de s à p

Pour déterminer un chemin entre une source s et un puits p en minimisant le nombre d'arcs traversés, le parcours en largeur constitue l'approche la plus adaptée. En explorant systématiquement les sommets par niveaux croissants de distance depuis la source, il garantit que le premier chemin trouvé vers le puits sera le plus court en nombre d'arcs.
À l'inverse, un parcours en profondeur explore prioritairement des chemins longs sans garantir leur minimalité, ce qui compromettrait la structure du graphe de couches et entraînerait des recherches redondantes et inefficaces. Cela ralentirait considérablement l'algorithme en augmentant inutilement le nombre d'opérations nécessaires avant d'atteindre le flot maximal.

De plus, d'autres algorithmes classiques de recherche de chemin, comme Dijkstra ou Bellman-Ford, ne sont pas appropriés dans ce contexte. Dijkstra est conçu pour minimiser la somme des poids associés aux arcs, ce qui n'est pas notre objectif ici : dans l'algorithme de Dinic, seuls l'existence d'une capacité résiduelle positive et le nombre d'arcs comptent, sans prise en compte d'un poids de coût. Quant à Bellman-Ford,s'il permet de traiter des poids négatifs, il présente une complexité beaucoup plus élevée ($O(n \times m)$) et n'apporte aucun avantage dans un graphe sans poids négatifs.

Dans ce cadre précis, le parcours en largeur offre la meilleure solution : il est à la fois optimal pour trouver rapidement un chemin court en nombre d'arcs et parfaitement adapté à la dynamique de l'algorithme de Dinic, où l'efficacité dans la construction du graphe de couches est essentielle pour accélérer la recherche des chemins augmentants.

Ainsi, le recours au parcours en largeur est justifié non seulement par la recherche d'optimalité des chemins, mais aussi par la volonté d'assurer la meilleure efficacité algorithmique possible.

2.2 Explications de l'algorithme de recherche en largeur (BFS)

Le parcours en largeur est une méthode d'exploration systématique d'un graphe. À partir d'un sommet donné (ici, la source), on visite d'abord tous ses voisins directs avant de progresser vers les voisins des niveaux supérieurs.

Dans l'algorithme de Dinic, le parcours en largeur sert à attribuer à chaque sommet une distance minimale depuis la source, exprimée en nombre d'arcs traversés. Chaque sommet est ainsi marqué par son niveau, permettant de construire un graphe de couches, où l'on ne conserve que les arcs reliant des sommets de niveaux consécutifs.

Ce choix du parcours en largeur est fondamental :
- Il garantit l'identification rapide des chemins les plus courts entre la source et le puits.
- Il réduit efficacement le nombre d'arcs considérés, car seuls les arcs respectant les niveaux successifs sont utilisés.

— Il accélère la recherche des chemins augmentants, favorisant une progression plus rapide du flot vers l'optimalité.

Ainsi, le parcours en largeur est spécifiquement adapté aux besoins de Dinic : il structure efficacement le graphe pour exploiter uniquement les chemins courts, assurant à la fois rapidité et efficacité dans le calcul du flot maximum.

2.3 Application de l'algorithme de DINIC sur un exemple

Le réseau R1 se compose de cinq sommets numérotés de 1 à 5 et de six arcs orientés entre eux, chacun associé à une capacité maximale.
Le graphe suivant représente ce réseau, où le flot initial est nul sur tous les arcs, et où les capacités maximales sont indiquées à proximité des arcs :

Les arcs et leurs capacités sont les suivants :

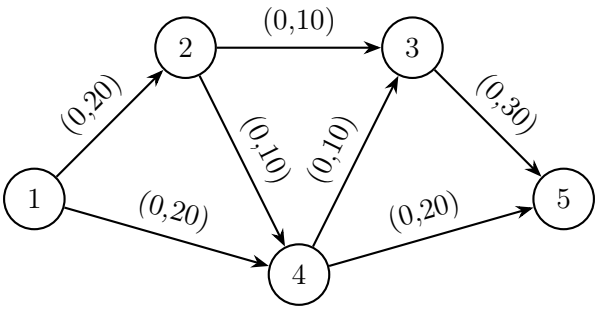

Figure 1 : Réseau R1 avec le flot initial nul.

À partir de cette configuration, nous allons dérouler pas à pas l'algorithme de Dinic, en appliquant successivement :
— Le parcours en largeur pour identifier les couches du graphe de couches.
— La recherche des chemins augmentants en utilisant le parcours en largeur.
— L'augmentation des flots,
— Et la mise à jour du graphe d'écart.
L'objectif est d'atteindre le flot maximal possible entre la source 1 et le puits 5.

On effectue un parcours en largeur à partir du sommet 1 :
— Niveau 0 : 1
— Niveau 1 : 2, 4
— Niveau 2 : 3, 5
Pour déterminer le chemin augmentant, je construis un tableau des pères lors du parcours en largeur. À partir du puits (sommet d'indice 5), je remonte en suivant les pères successifs jusqu'à atteindre la source, ce qui me permet de reconstituer le chemin améliorant.

Pour aller de la source au puit, le chemin le plus court en nombre d'arc est donc le chemin 1-4-5. Ces deux arcs ont une capacité de 20, il vient alors qu'il est possible d'augemter le flot de 20 à prtir de ces deux acrs.
Mise à jour du graphe :

Il suffit maintenant de mettre à jour le graphe d'écart :

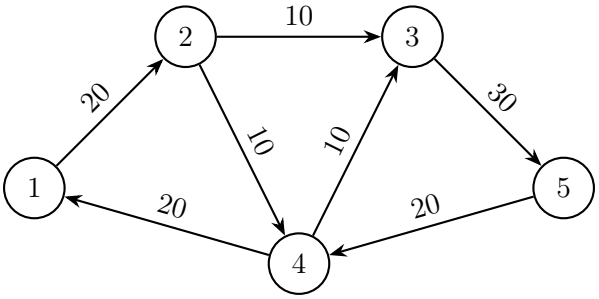

Figure 2 : Mise à jour du graphe d'écart avec $\phi = 20$.

Cela termine la première itération de l'algorithme de DINIC. Le flot et le graphe d'écart sont mis à jour et la nouvelle itération de l'algorithme est lancée. On effectue à nouveau un parcours en largeur sur le graphe d'écart :

— Niveau 0 : 1
— Niveau 1 : 2
— Niveau 2 : 3, 4
— Niveau 3 : 5

Il vient que le chemin le plus court en nombre d'arc est le chemin 1-2-3-5. Il est donc possible d'augmenter le flot de 10.

Mise à jour du graphe d'écart :

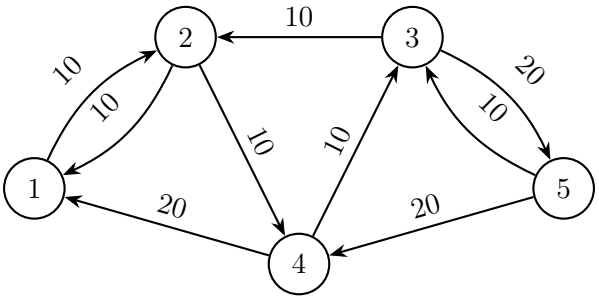

Figure 3 : Mise à jour du graphe d'écart.

Nous effectuons à nouveau un parcours en largeur sur le graphe d'écart, et nous obtenons ainsi les niveaux suivants :

— Niveau 0 : 1
— Niveau 1 : 2
— Niveau 2 : 4
— Niveau 3 : 3
— Niveau 4 : 5

Le chemin augmentant est 1-2-4-3-5, il augmente le flow de 10.

Mise à jour du graphe d'écart :

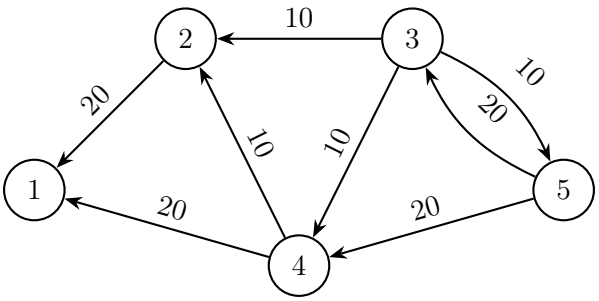

Figure 4 : Mise à jour du graphe d'écart.

Nous effectuons à nouveau un parcours en largeur sur le graphe d'écart. Cependant les arcs $(1 \to 4)$, $(4 \to 5)$, $(4 \to 3)$, $(1 \to 2)$ et $(2 \to 3)$ sont saturés. Il est donc impossible d'atteindre 5 depuis 1 en respectant les niveaux (aucun chemin complet $1 \to ... \to 5$ n'existe). Il n'y a aucun chemin d'augmentation disponible. L'algorithme de Dinic s'arrête ici avec un flux max de 40 .

Nous obtenons finalement le réseau suivant avec un flot maximal de 40 :

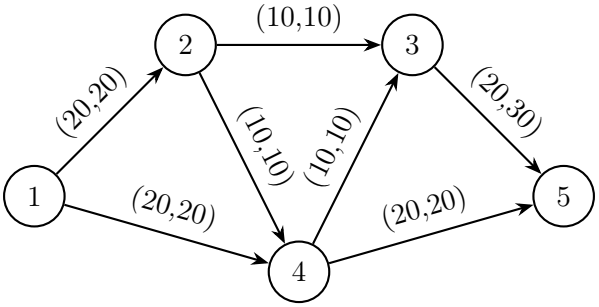

Figure 5 : Réseau R1 avec le flot maximal $\phi = 40$.

Capacités des arcs :
— $(1 \to 2)$ saturé (capacité résiduelle 0), le flot vaut 20
— $(2 \to 4)$ saturé (capacité résiduelle 0), le flot vaut 10
— $(4 \to 3)$ saturé (capacité résiduelle 0), le flot vaut 10
— $(3 \to 5)$ passe de 20 à 10 de capacité résiduelle, la flot vaut 20

3 Analyse des structures de données pour implémenter le réseau et le graphe d'écart associé

3.1 Coût mémoire

Dans l'étude des structures de données pour représenter un graphe orienté, il est important de comparer leur consommation mémoire, en particulier pour des graphes dits "creux", c'est-à-dire avec peu d'arcs par rapport au nombre de sommets.

Trois structures principales sont envisagées :

— La matrice d'incidence représente le graphe à l'aide d'une matrice de taille n×m, où n est le nombre de sommets et m le nombre d'arcs. Chaque colonne correspond à un arc, et contient une valeur positive (+1) pour le sommet de départ, et une valeur négative (-1) pour le sommet d'arrivée.Cette structure consomme une mémoire proportionnelle à O(n×m), ce qui peut devenir très coûteux pour des graphes de grande taille, même s'ils sont creux.

— Les tableaux sommet-successeurs utilisent deux tableaux, l'un listant les successeurs de chaque sommet, et l'autre indiquant le début de la liste pour chaque sommet. Cette structure nécessite un espace proportionnel à O(n+m), où m est le nombre d'arcs, ce qui est beaucoup plus adapté aux graphes creux.

— La liste de successeurs repose sur une liste chaînée associée à chaque sommet, contenant uniquement les arcs effectivement présents. Elle présente également un coût mémoire en O(n+m), mais légèrement supérieur aux tableaux à cause de l'ajout de pointeurs pour chaîner les éléments.

Ainsi, bien que les tableaux sommet-successeurs et les listes de successeurs présentent tous deux une complexité mémoire en O(n+m), l'implémentation par tableaux est légèrement plus avantageuse en pratique. En effet, elle évite le surcoût mémoire induit par les pointeurs nécessaires au chaînage des listes, ce qui la rend plus compacte pour des graphes creux.

En conclusion, la structure utilisant des tableaux sommet-successeurs est la plus avantageuse du point de vue du coût mémoire. Elle permet de représenter efficacement un graphe creux en limitant l'espace utilisé, tout en assurant un accès direct aux successeurs sans gaspiller de mémoire.

3.2 Coût de traitement

3.2.1 Coût liés à l'accès des successeurs

L'efficacité de l'accès aux successeurs d'un sommet dépend fortement de la structure de données choisie pour représenter le graphe.

— Matrice d'incidence : Cette structure associe à chaque arc une colonne et à chaque sommet une ligne. Pour savoir quels sont les successeurs d'un sommet donné, il faut parcourir toutes les colonnes de la matrice afin d'identifier celles dont la ligne correspondant au sommet contient une valeur indiquant un arc sortant (par convention, +1). Il faut ensuite vérifier, pour chaque arc identifié, vers quel sommet il pointe. Ce processus impose de parcourir toutes les colonnes, soit un coût en O(m), indépendamment du nombre réel de successeurs. Cette approche est donc peu efficace, notamment pour les graphes creux, où peu d'arcs sont présents.

— Tableaux sommet-successeurs : Cette structure permet un accès direct et rapide aux successeurs. Chaque sommet est associé à une sous-liste continue dans un

tableau, ce qui permet de parcourir uniquement les arcs effectivement présents. Le coût d'accès est proportionnel au degré du sommet, soit O(degre(v)). Le degré(v) désigne le nombre de successeurs directs du sommet v (arcs sortants partant de v).

— Listes de successeurs : De manière similaire aux tableaux, la liste chaînée permet un accès proportionnel au degré du sommet, soit O(degre(v)) également. La différence pratique réside uniquement dans la nécessité de suivre des pointeurs d'un nœud à l'autre, ce qui peut introduire un surcoût mineur en temps d'accès mémoire, sans changer la complexité théorique.

Pour conclure le tableau sommet-successeurs et la liste de successeurs présentent un coût de traitement quasi identique en termes de complexité O(degre(v), rendant les deux structures équivalentes d'un point de vue théorique.

Toutefois, en pratique, le tableau présente un très léger avantage en termes de rapidité d'accès, du fait de la continuité des données en mémoire, réduisant ainsi les sauts nécessaires.

3.2.2 Coût liés à l'ajout et à la suppression d'arcs

Lors de l'évolution d'un graphe, certaines opérations nécessitent d'ajouter ou de supprimer dynamiquement des arcs. L'impact de ces opérations dépend fortement de la structure de données utilisée :

— Matrice d'incidence : Cette structure associe à chaque arc une colonne et à chaque sommet une ligne. Pour savoir quels sont les successeurs d'un sommet donné, il faut parcourir toutes les colonnes de la matrice afin d'identifier celles dont la ligne correspondant au sommet contient une valeur indiquant un arc sortant (par convention, +1). Il faut ensuite vérifier, pour chaque arc identifié, vers quel sommet il pointe. Ce processus impose de parcourir toutes les colonnes, soit un coût en O(m), indépendamment du nombre réel de successeurs. Cette approche est donc peu efficace, notamment pour les graphes creux, où peu d'arcs sont présents.

— Tableaux sommet-successeurs : Dans cette structure, les arcs sont stockés dans des tableaux continus. Ajouter un arc nécessiterait d'insérer un nouvel élément dans le tableau, ce qui peut obliger à décaler de nombreux éléments et entraîner un coût important (O(m) dans le pire cas). Supprimer un arc impose aussi de décaler les éléments suivants. Les ajouts et suppressions sont donc très coûteux en tableaux. Cette structure est donc peu adaptée à des modifications dynamiques fréquentes.

— Listes de successeurs : Avec les listes chaînées, l'ajout d'un nouvel arc ou la suppression d'un arc existant est beaucoup plus souple. L'insertion ou la suppression d'un élément dans une liste chaînée se fait en O(1), en ajustant simplement quelques pointeurs. Les listes conviennent ainsi naturellement aux graphes amenés à évoluer.

En résumé, du point de vue de la facilité de modification du graphe, les listes chaînées apparaissent comme la structure la plus adaptée, car elles permettent d'ajouter ou de supprimer un arc en temps constant grâce à une simple manipulation de pointeurs. La matrice d'incidence, bien que conceptuellement simple, est rigide et coûteuse en cas de modifications fréquentes. Enfin, les tableaux sommet-successeurs sont peu adaptés aux ajouts et suppressions, car toute modification nécessite potentiellement le déplacement de nombreux éléments, ce qui entraîne un surcoût important en temps de traitement.

Face à ces difficultés, notamment dans les tableaux sommet-successeurs, il devient nécessaire d'adapter la définition du graphe d'écart afin de s'affranchir des opérations

coûteuses d'ajout et de suppression d'arcs. Pour ce faire une adaptation de la gestion du graphe d'écart est utilisée :

- Tous les arcs potentiels (directs et retours) sont présents dès la création du graphe.
- Lors de l'exécution de l'algorithme, seule la capacité résiduelle des arcs est modifiée.
- Un arc est considéré comme utilisable si sa capacité résiduelle est strictement positive, sinon il est ignoré lors des parcours.

Ainsi, aucun ajout ni suppression physique d'arc n'est nécessaire pendant le traitement, ce qui stabilise la structure du graphe et optimise les parcours. Cette méthode est mise en œuvre dans l'algorithme de Dinic, qui exploite ce modèle fixe pour garantir l'efficacité et la simplicité de la gestion du graphe d'écart tout au long de son déroulement.

3.3 Choix de la structure de données

Pour représenter un réseau ou un graphe d'écart, nous utilisons une structure de données basée sur une liste de successeurs. Cette structure consiste en un tableau de taille n (où n est le nombre de sommets du graphe), indexé par les numéros de sommets. Chaque case du tableau contient un pointeur vers une liste chaînée, qui regroupe l'ensemble des successeurs (ou voisins directs) du sommet correspondant. Concrètement, pour un sommet donné, la liste chaînée associée énumère tous les sommets vers lesquels il existe une arête sortante. Cette organisation permet un accès direct aux successeurs d'un sommet et offre une représentation compacte, notamment pour les graphes creux. L'ajout d'une arête consiste simplement à insérer un nouveau nœud en tête de la liste chaînée appropriée, garantissant ainsi une complexité en temps constant.

Pour illustrer cela, nous prenons pour exemple le réseau de la section 1 :

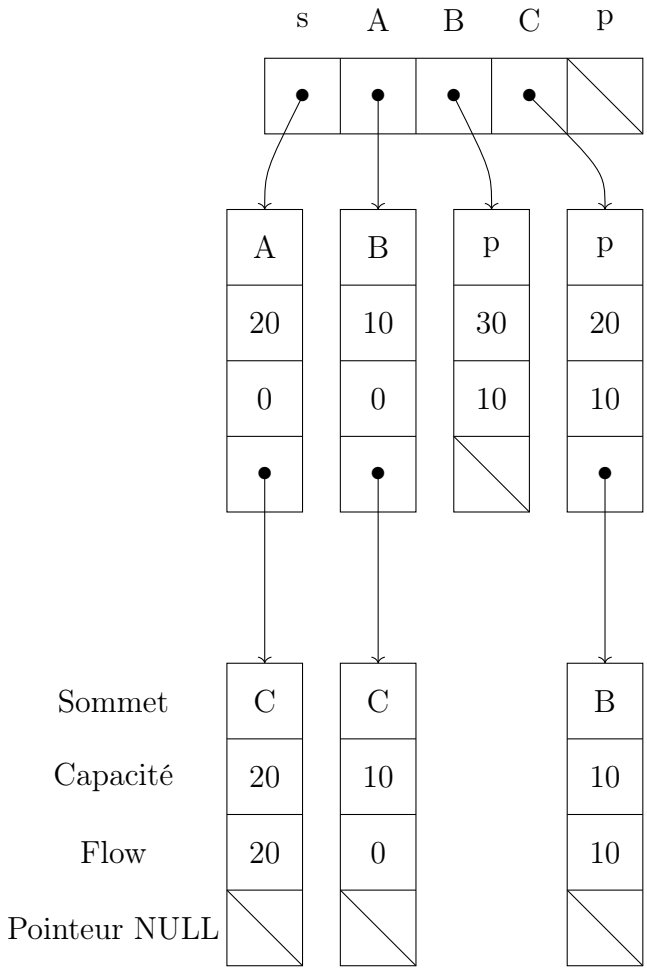

Figure 6 : Représentation par liste de successeurs.

Représentons maintenant son graphe d'écart associé, à noté que l'on utilise la même stuture de donnée que précédemment ainsi pour les listes chainée nous avons besoin de seulement 3 informations le sommet – capacité – successeur. *Remarque : afin de réutiliser la même stucture de donnéee on stockera -1 dans la case dédié au flow, elle n'est pas représentée sur la figure suivante. De même nous nous gardons le droit d'ajouter une tête aux chainées lors que lors implémentation en C, elles ne sont pas représentées ici.*

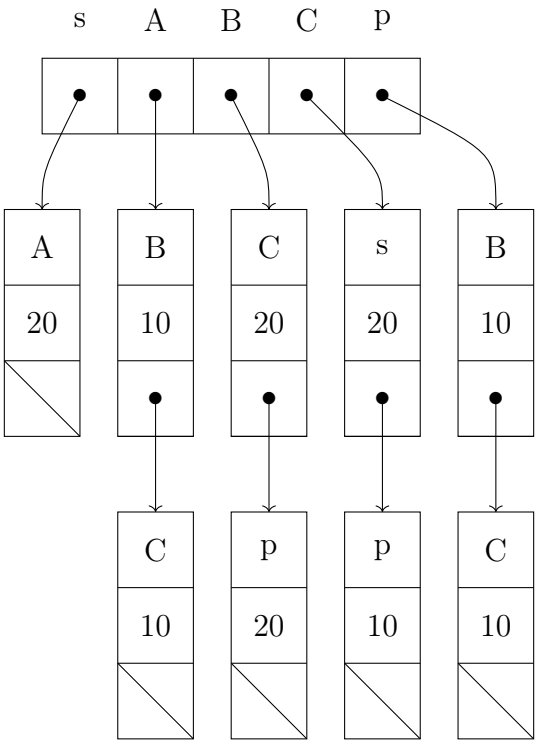

Figure 7 : Représentation par liste de successeurs du graphe d'écart.

4 Décomposition de l'algorithme de DINIC

4.1 Stucture de données pour représenter un chemin de s à p

La structure de données choisie pour représenter un chemin de s à p est une liste de successeurs. Cette structure permet de stocker le chemin de s à p de la façon la plus efficace possible en terme de mêmoire et de traitement. La structure comportera une tête avec les champs **nb** et **tête** qui sont respectivement le nombre de maillon de la chaine et un pointeur vers le premier maillon de la chaine. Les maillons seront eux constitier de deux champs **sommet** et **suivant**. Le dernier maillon de la chaine sera un maillon dont le champ suivant sera **NULL** (le pointeur NULL). *Remarque : Dans la suite de l'implémentation des procédures, le champ nb ne n'est pas utilisé mais nous tenons à le garder à la fois pour l'esthétisme et dans le cas de modifications future nécessitant de connaitre rapidement le nombre de successeurs.*

4.2 Pseudo-code des prodédures de l'algorithme de Dinic

Ici sont regroupées les procédures et leur pseudo-code de l'algorithme de Dinic tout en tennant compte de la structure de données choisie.

4.2.1 builRG

Procédure 1: buildRG(G : réseau)

```
   Result: RG : graphe d'écart
 1 n ← taille(G);
 2 initialiser pointeur NULL Q;
 3 initialiser RG comme un graphe vide de n sommets;
 4 for i ← 1 to n do
 5 |   Q ← G[i];
 6 |   while Q ≠ NULL do
 7 |   |   if Q.capacité - Q.flow > 0 then
 8 |   |   |   initialiser maillon m avec sommet =
   |   |   |    Q.sommet, capacité = Q.capacité - Q.flow et
   |   |   |    suivant = RG[i];
 9 |   |   |   RG[i] ← m;
10 |   |   end
11 |   |   if Q.flow > 0 then
12 |   |   |   initialiser maillon m avec sommet = i, capacité
   |   |   |    = Q.flow et suivant = RG[Q.sommet];
13 |   |   |   RG[Q.sommet] ← m;
14 |   |   end
15 |   |   Q ← Q.suivant
16 |   end
17 end
18 Retourner RG
```

4.2.2 shortestPath

Procédure 2: shortestPath(RG : graphe, s : source, p : puits)

```
   Result: L : liste de successeurs
 1 n ← taille(RG);
 2 initialiser tableau niveau de taille n à -1;
 3 initialiser tableau Pére de taille n à 0;
 4 niveau[s] ← 0;
 5 Pére[s] ← s;
 6 créer une file File;
 7 enfiler(File, s);
 8 while File ≠ NULL do
 9 |   u ← defiler(File);
10 |   initialiser pointeur NULL Q;
11 |   Q ← G[u];
12 |   while Q ≠ NULL do
13 |   |   if niveau[Q.sommet] = -1 then
14 |   |   |   niveau[Q.sommet] ← niveau[u] + 1;
15 |   |   |   Pére[Q.sommet] ← u;
16 |   |   |   enfiler(File, Q.sommet);
17 |   |   end
18 |   |   Q ← Q.suivant
19 |   end
20 end
21 // Construction de la liste de successeurs initialiser liste
     chainée L;
22 sommetPére ← p;
23 initialiser maillon m avec sommet = p et suivant =
     NULL;
24 initialisation liste chainee L;
25 L.tête ← m;
26 while sommetPére != s do
27 |   initialiser maillon m avec sommet = sommetPére et
   |     suivant = L.tête;
28 |   m.suivant ← L.tête;
29 |   L.tête ← m;
30 |   sommetPére ← Pére[sommetPére];
31 end
32 Retourner L
```

4.2.3 minCapa

Procédure 3: minCapa(G : réseau, L : liste chainées de s à p)

```
   Result: k : capacité minimale
 1 minCapacité ← infinie;
 2 initialiser pointeur NULL P1;
 3 initialiser pointeur NULL P2;
 4 P1 ← L.tête;
 5 P2 ← P1.suivant;
 6 while P2 ≠ NULL do
 7 |   capa ← 0;
 8 |   initialiser pointeur NULL Q;
 9 |   Q ← G[P1.sommet];
10 |   while Q ≠ NULL and Q.sommet != P2.sommet do
11 |   |   Q ← Q.suivant;
12 |   end
13 |   capa ← |Q.capacité - Q.flow|;
14 |   if capa < minCapacité then
15 |   |   minCapacité ← capa;
16 |   end
17 |   P1 ← P2;
18 |   P2 ← P2.suivant;
19 end
20 Retourner minCapacité
```

4.2.4 updateFlowInRG

Procédure 4: updateFlowInRG(RG : graphe d'écart, chemin : liste chainée de s à p, k : capacité minimale)

```
1  n = taille RG;
2  initialiser pointeur Q = chemin.tête;
3  initialiser pointeur NULL P;
4  initialiser pointeur NULL K;
5  sommet ← Q.sommet;
6  while Q.suivant ≠ NULL do
7  |  Q ← Q.suivant;
8  |  P ← RG.tab[sommet].tête;
9  |  while P.sommet ≠ Q.sommet do
10 |  |  P ← P.suivant;
11 |  end
12 |  if P.capacité == P.flow then
13 |  |  P.flow ← P.flow - k;
14 |  |  ajout_tete(RG)(RG.tab[P.sommet],
   |  |   sommet,P.capacité,k);
15 |  end
16 |  else
17 |  |  if P.capacité - P.flow > 0 then
18 |  |  |  P.flow ← P.flow - k;
19 |  |  |  K ← RG.tab[P.sommet].tête;
20 |  |  |  while K.sommet != sommet do
21 |  |  |  |  K ← K.suivant;
22 |  |  |  end
23 |  |  |  K.flow ← K.flow + k;
24 |  |  |  ajout_tete(RG)(RG.tab[K.sommet],
   |  |  |   sommet,k);
25 |  |  end
26 |  end
27 |  // On regarde s'il faut retirer l'arc if
   |   clear_maillon(RG.tab[sommet],P);
28 |   then P.flow == 0
29 |  |  s
30 |  end
31 |  ommet ← Q.sommet;
32 end
```

4.2.5 updateFlowInNet

Pour cet algorithme pour partons du principe que si un arc (u,v) appartient à RG et si (v,u) appartient à G, alors le flow de (v,u) dans G correspond au poids de (u,v).

Procédure 5: updateFlowInNet(G : réseau, RG : graphe d'écart)

```
1  n ← taille(G);
2  initialiser pointeur NULL Q;
3  initialiser pointeur NULL K;
4  for i ← 1 to n do
5  |  Q ← G[i];
6  |  while Q ≠ NULL do
7  |  |  K ← RG[Q.suivant];
8  |  |  Ref ← False;
9  |  |  while K ≠ NULL et non Ref do
10 |  |  |  if K.sommet == i then
11 |  |  |  |  Ref ← True;
12 |  |  |  end
13 |  |  |  K ← K.suivant;
14 |  |  end
15 |  |  if Ref then
16 |  |  |  K.flow ← Q.flow;
17 |  |  end
18 |  |  Q ← Q.suivant;
19 |  end
20 end
```

4.2.6 Pseudo-code de l'algorithme de Dinic

Procédure 6: Main(s : source, p : puits)

```
1  initialiser réseau NULL G;
2  G ← buildGraph();
3  initialiser graphe d'écart NULL RG;
4  RG ← buildRG(G);
5  initialiser liste chainée L;
6  L ← shortestPath(RG, s, p);
7  flot ← 0;
8  while L ≠ NULL do
9  |  k ← minCapa(RG, L);
10 |  updateFlowInRG(G, L, k);
11 |  flot ← flot + k;
12 |  L ← shortestPath(RG, s, p);
13 end
14 updateFlowInNet(G,RG);
15 ouvrirFichier(resultat.txt);
16 écrireFichier("flot maximal = ", flot);
17 écrireFichier(retour à la ligne);
18 initialiser pointeur NULL Q;
19 n ← taille(G);
20 for i ← 1 to n do
21 |  Q ← G[i];
22 |  while Q ≠ NULL do
23 |  |  écrireFichier(G[i], " -> ", Q.sommet, " : ",
   |  |   Q.flow");
24 |  |  écrireFichier(retour à la ligne);
25 |  |  Q ← Q.suivant;
26 |  end
27 end
28 fermerFichier(resultat.txt);
```

5 Mode d'emploie

5.1 Compilation

Voici les commandes de compilation de tout le code. Nous utilisons ici le compilateur GNU Compiler Collection.

Pour la compilation des différents fichiers objets :

```
gcc -Wall -c liste-chainee.c
gcc -Wall -c liste-successeurs.c
gcc -Wall -c buildGraph.c
gcc -Wall -c buildRG.c
gcc -Wall -c shortestPath.c
gcc -Wall -c minCapa.c
gcc -Wall -c updateFlowInRG.c
gcc -Wall -c updateFlowInNet.c
gcc -Wall -c main.c
```

Pour la compilation du programme principal :

```
gcc -Wall -o Dinic main.o liste-chainee.o
liste-successeurs.o buildGraph.o buildRG.o
shortestPath.o minCapa.o updateFlowInRG.o
updateFlowInNet.o
```

5.2 Compilation avancée

Afin de s'assurer que le code ne comporte pas de fuite de mémoire nous utilisons valgrind. Il suffit de rajouter lors de chaque compilation l'attribut "-g". Il est ensuite possible de lancer une analyse complète du code avec la commande :

```
valgrind --leak-check=full --track-origins=yes ./Dini
```

```
buildGraph.c  buildRG.h        liste-successeurs.c  minCapa.c       shortestPath.h     updateFlowInRG.c
buildGraph.h  liste-chainee.c  liste-successeurs.h  minCapa.h       updateFlowInNet.c  updateFlowInRG.h
buildRG.c     liste-chainee.h  main.c               shortestPath.c  updateFlowInNet.h
aevarist@florine16:~/Bureau/Projet_Graphe/code_valgrind$ gcc -Wall -c -g liste-chainee.c
gcc -Wall -c -g liste-successeurs.c
gcc -Wall -c -g buildGraph.c
gcc -Wall -c -g buildRG.c
gcc -Wall -c -g shortestPath.c
gcc -Wall -c -g minCapa.c
gcc -Wall -c -g updateFlowInRG.c
gcc -Wall -c -g updateFlowInNet.c
gcc -Wall -c -g main.c
aevarist@florine16:~/Bureau/Projet_Graphe/code_valgrind$  gcc -g liste-chainee.o liste-successeurs.o buildGraph.o
buildRG.o shortestPath.o minCapa.o updateFlowInRG.o updateFlowInNet.o main.o -o Dinic
aevarist@florine16:~/Bureau/Projet_Graphe/code_valgrind$  valgrind --leak-check=full --track-origins=yes ./Dinic
==27759== Memcheck, a memory error detector
==27759== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==27759== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==27759== Command: ./Dinic
==27759==
Usage: ./Dinic <fichier.dimacs> [affichage_etapes]
==27759==
==27759== HEAP SUMMARY:
==27759==     in use at exit: 0 bytes in 0 blocks
==27759==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==27759==
==27759== All heap blocks were freed -- no leaks are possible
==27759==
==27759== For lists of detected and suppressed errors, rerun with: -s
==27759== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
aevarist@florine16:~/Bureau/Projet_Graphe/code_valgrind$
```

FIGURE 1 – Utilisation de valgrind

5.3 Description des entrées et sorties

Nous utilisons ici comme entrée des fichiers au format DIMACS, ils contiennent les graphes que l'on souhaite traiter (Une bref description de leur fonctionnement est décrite dans le sujet). Le programme produit un fichier .txt en sortie. Celui-ci contient sur la première ligne le flow optimal obtenu, les autres lignes indiquent pour chaque arcs le flow calculé.

Exemple : ./Dinic net1.txt

Il est aussi possible de donner un second argument au programme, si l'on précise « true » ou « 1 » cela active un affichage de toutes les étapes de façon détaillée lors de l'exécution du programme.

Exemple : ./Dinic net1.txt true ou ./Dinic net1.txt 1

La sortie est un fichier .txt assez simple contenant le flow optimal du graphe et les flots pour chaque arc.

6 Description détaillée d'exemples

6.1 Graphe R1

Exemple R1 : Ce premier cas contient un petit réseau composé de 5 sommets et 6 arcs, destiné à valider le fonctionnement élémentaire de notre implémentation. L'exécution de notre programme donne un flot final de 40, ce qui correspond parfaitement au résultat attendu détaillé dans notre rapport (section 2.3).

La décomposition du flot se fait comme ci-dessous :

```
Flow final dans le réseau : 40
Arc : 1 -> 4 ; Flow : 20
Arc : 1 -> 2 ; Flow : 20
Arc : 2 -> 4 ; Flow : 10
Arc : 2 -> 3 ; Flow : 10
Arc : 3 -> 5 ; Flow : 20
Arc : 4 -> 5 ; Flow : 20
Arc : 4 -> 3 ; Flow : 10
```

FIGURE 2 – Résultats R1

6.2 Graphe G 100 300

Exemple G 100 300 : Ce fichier représente un graphe de taille moyenne, utilisé pour tester l'efficacité de notre algorithme sur des données plus réalistes. Le graphe comprend 102 sommets et 290 arcs, comme indiqué dans l'en-tête DIMACS (p 102 290).
L'exécution de notre programme a permis d'obtenir un flot maximal de 9 860 177, réparti sur l'ensemble des chemins possibles de la source (s = 1) au puits (t = 102). Un extrait des résultats montre les flux suivants depuis le sommet source :

```
Flow final dans le réseau : 9860177
Arc : 1 -> 2 ; Flow : 81247
Arc : 1 -> 3 ; Flow : 71011
Arc : 1 -> 4 ; Flow : 1219979
Arc : 1 -> 5 ; Flow : 1225786
Arc : 1 -> 6 ; Flow : 250097
Arc : 1 -> 7 ; Flow : 688836
```

FIGURE 3 – Résultats de l'exécution sur G 100 300

6.3 Graphe G 900 2700

Exemple G 900 2700 : Ce test a été réalisé sur un graphe de grande taille contenant 902 sommets et 2700 arcs. L'objectif est ici de tester l'efficacité de notre implémentation de l'algorithme de Dinic. Le flot maximal obtenu est bien 28 258 807.
Le flot est réparti efficacement sur les arcs depuis la source (s = 1) jusqu'au puits (t = 902). Voici un extrait des flux sortants :

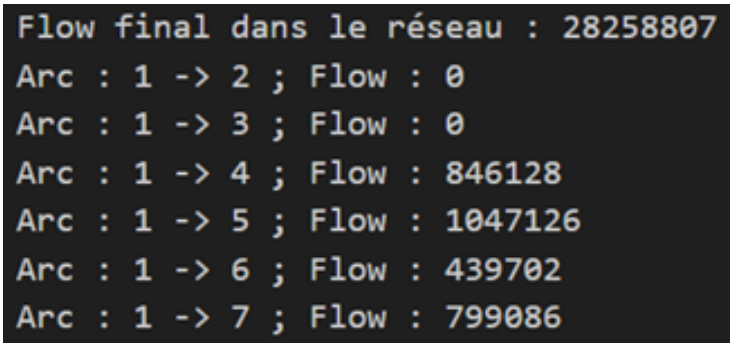

FIGURE 4 – Résultats pour l'exécution G 900 2700

6.4 Graphe G 2500 7500

Exemple G 2500 7500 : Ce cas est le plus volumineux traité dans notre série de tests. Le graphe contient 2502 sommets et 7500 arcs, ce qui permet de tester les limites de performance de notre implémentation de l'algorithme de Dinic. Nous obtenons bien un flot maximal de 42 791 871.
Comme dans les autres cas, notre programme a su construire efficacement des chemins augmentant à travers le réseau. Exemple des flux sortants du sommet source :

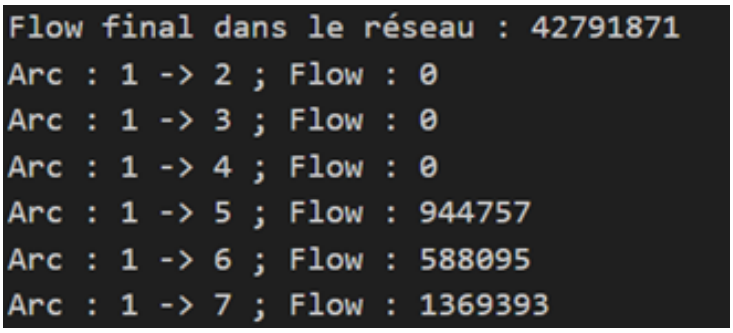

FIGURE 5 – Résultats de l'exécution sur G 2500 7500

Ces résultats ont été obtenus sans erreurs, et confirment que notre implémentation gère efficacement de grands réseaux. Le temps de calcul reste raisonnable (voir l'Annexe avec le tableau des temps), démontrant la pertinence des choix de structures de données (notamment l'utilisation de listes de successeurs pour optimiser les accès mémoire et éviter les suppressions d'arcs coûteuses).

7 Conclusions et améliorations possibles

Ce projet avait pour objectif de résoudre le problème du flot maximum en réseau, en s'appuyant sur l'algorithme de Dinic, reconnu pour son efficacité sur les graphes creux. L'ensemble du pseudocode a été rigoureusement traduit en langage C, avec un souci de modularité : chaque fonction a été isolée dans un fichier source spécifique, accompagné d'un fichier d'en-tête dédié, ce qui a permis de construire une architecture claire, lisible et facilement maintenable. Les premiers tests ont été réalisés sur des instances simples comme R1 et R2, afin de valider la justesse de notre implémentation. Par la suite, nous avons étendu nos expérimentations à des graphes de plus grande taille tels que G 100 300, G 900 2700 et G 2500 7500, afin d'évaluer la robustesse, la stabilité et la scalabilité de notre programme. Les résultats obtenus se sont révélés conformes aux attentes théoriques, avec des flots maximaux corrects et des temps d'exécution très satisfaisants, même pour les plus grandes instances.

Toutefois, certaines pistes d'amélioration peuvent être envisagées pour enrichir notre travail. Il serait notamment pertinent d'ajouter un module de visualisation graphique permettant de suivre l'évolution du graphe d'écart et la répartition du flot à chaque itération. Cette fonctionnalité offrirait une meilleure lisibilité du fonctionnement de l'algorithme et permettrait une interprétation plus intuitive des résultats, notamment dans un but pédagogique ou d'analyse plus poussée.

De plus, nous avons tenu à conserver, dans l'implémentation des listes chaînées, une cellule de tête explicite, même si celle-ci n'est pas indispensable au bon fonctionnement des algorithmes actuels. Ce choix, volontairement structurel, s'inscrit dans une logique de conception modulaire. En effet, la présence d'une tête rend l'initialisation, l'ajout et la suppression d'éléments plus cohérents et homogènes, notamment en évitant de devoir gérer des cas particuliers pour les premières insertions. De plus, cela facilite la maintenance du code, le débogage et l'extension future de la structure, par exemple pour intégrer des métriques sur la liste (taille dynamique, pointeur de queue, etc.) ou pour implémenter des variantes comme les listes doublement chaînées. Ce compromis entre performance immédiate et robustesse architecturale nous semble pertinent dans le cadre d'un projet évolutif.

Enfin, il est possible d'optimiser la compilation du code en utilisant le flag -02. Le programme a été compilé avec les options suivantes :

```
gcc -Wall -O2 -march=native -flto -c *.c
gcc -Wall -O2 -march=native -flto -o Dinic *.o
```

Ces options permettent :

- `-O2` : des optimisations standards (propagation de constantes, déroulage de boucles, etc.).
- `-march=native` : une optimisation adaptée à l'architecture CPU locale.
- `-flto` : des optimisations à l'édition de liens (Link Time Optimization).

Nous fournissons en annexe une analyse détaillées des temps d'exécution pour chacune des instances test.

8 Bilans personnels

Alexis Evaristo :

Ce projet m'a permis de concrétiser mes connaissances théoriques en théorie des graphes à travers une implémentation complète en langage C. J'ai particulièrement apprécié le défi d'organiser rigoureusement les structures de données et l'architecture modulaire du projet, en veillant à la clarté et à la cohérence entre les différents fichiers source et en-tête.

Au-delà du code, j'ai également pris plaisir à structurer un dépôt Git propre et complet, avec une documentation disponible en français et en anglais, ce qui m'a permis de consolider mes bonnes pratiques en développement collaboratif et en versionnage.

Enfin, la rédaction du rapport en LaTeX, un outil que je découvrais pour l'occasion, m'a offert une nouvelle compétence très précieuse. Ce travail d'édition m'a permis de me préparer efficacement à mon futur stage en laboratoire de recherche, en développant mon autonomie et ma rigueur scientifique. Ce projet a donc été pour moi une expérience aussi formatrice que motivante, en parfaite cohérence avec mes objectifs académiques et professionnels.

Enfin, chat gpt m'a permis de gagner du temps dans la mise en page LaTeX, en générant du code pour les tableaux ou en corrigeant des formulations.

Imane Mahmoudi :

Dans ce projet, je me suis principalement investi dans la phase de modélisation et de structuration de l'algorithme. Cela m'a permis de consolider ma compréhension du problème du flot maximum, notamment en analysant les mécanismes internes de l'algorithme de Dinic et la construction du graphe d'écart. J'ai participé à la réflexion autour des structures de données à adopter, en évaluant les avantages et les limites des différentes représentations possibles (matrice, liste de successeurs, tableaux chaînés) pour garantir à la fois performance et clarté. Ce travail m'a beaucoup appris sur l'importance de la conception en amont dans un projet algorithmique, et m'a donné une méthodologie que je pourrai réutiliser dans d'autres projets complexes. Finalement ce projet m'a permis de mieux comprendre les exigences propres aux problèmes de graphes, et surtout l'importance de bien structurer et modéliser les données en amont pour pouvoir les résoudre de manière efficace.

Pour m'accompagner dans ce projet, j'ai utilisé ChatGPT comme une aide à la rédaction, à la clarification de concepts, et à l'organisation de mes idées. Je m'en suis servi notamment pour reformuler le rapport de manière plus fluide et plus professionnelle, ainsi que pour structurer la présentation des résultats (par exemple, la rédaction des bilans, des tableaux de performances ou des conclusions). ChatGPT m'a également aidé à mieux comprendre certains aspects théoriques de l'algorithme de Dinic, comme la gestion du graphe d'écart ou le rôle des parcours en largeur, en les expliquant avec des exemples simples.

Auto évalution : 01 = 1, 02 = 1, 03 = 1, 04 = 1, 05 = 0.5 Total : 10 points, note finale : 19/20.

9 Annexe

- *Voici le lien vers le Github du projet :* https://github.com/AlexGit31/Projet-Graphe

- *Tableau des temps obtenus :*

Fichier	Temps total (real)	Temps utilisateur (user)	Temps système (sys)
`nets1.txt`	0,019 s	0,000 s	0,003 s
`net2.txt`	0,006 s	0,000 s	0,003 s
`G_100_300`	0,013 s	0,008 s	0,000 s
`G_900_2700`	0,180 s	0,163 s	0,013 s
`G_2500_7500`	1,619 s	1,564 s	0,049 s

TABLE 1 – Temps d'exécution mesurés pour différentes instances

Sur les plus petits graphes, les résultats sont quasi instantanés : nets1.txt s'exécute en 0,019 seconde, net2.txt en 0,006 seconde et G 100 300 en 0,013 seconde. Pour des graphes plus volumineux, les temps restent très raisonnables : G 900 2700 est traité en 0,180 seconde, et même le fichier le plus lourd, G 2500 7500, est résolu en seulement 1,619 seconde.

On note que la majorité du temps est consacrée à l'exécution utilisateur (par exemple 1,564 seconde sur les 1,619 secondes pour G 2500 7500), ce qui montre une bonne gestion mémoire et une faible charge système (seulement 0,049 seconde dans ce cas). Ces résultats confirment que notre programme est à la fois rapide et capable de gérer efficacement des graphes allant jusqu'à plusieurs milliers de sommets.

- Temps avec optimisation :

Fichier	Taille estimée	Temps total	User	System	CPU
`net1.txt`	Petit graphe	0,007s	0,00s	0,00s	58%
`G_100_300.max`	100 sommets, 300 arêtes	0,007s	0,00s	0,00s	65%
`G_900_2700.max`	900 sommets, 2700 arêtes	0,071s	0,06s	0,01s	95%
`G_2500_7500.max`	2500 sommets, 7500 arêtes	0,567s	0,52s	0,04s	98%

TABLE 2 – Temps avec optimisation `-O2`

— Le temps d'exécution croît de manière raisonnablement proportionnelle à la taille du graphe, ce qui montre une bonne scalabilité.
— L'utilisation du CPU est très élevée (jusqu'à 98%), ce qui indique que le programme est bien optimisé et tire parti des ressources disponibles.
— Le programme reste très rapide, même pour des graphes de plusieurs milliers de sommets.
— Ces résultats valident le choix des options de compilation et la qualité de l'implémentation.

L'utilisation des optimisations classiques de GCC combinée à des options spécifiques à l'architecture (`-march=native`) et à l'optimisation au moment du linkage (`-flto`) permet de réduire significativement le temps d'exécution. L'algorithme de Dinic s'est montré efficace et stable sur des graphes de tailles variées. Une comparaison future avec une version non optimisée pourrait être envisagée, ainsi qu'une exploration de la *profil-guided optimization (PGO)* pour améliorer encore davantage les performances sur des cas typiques.