



Résolution du problème du flot maximum

Auteurs

Cédric Audie
Marie Dalenc
Julien Lahoz
Adrien Martinelli

Encadrant

Rodolphe Giroudeau

30 janvier 2025

Table des matières

1	Définitions générales	2
1.1	Réseau résiduel	2
1.2	Chemin et flot améliorants	2
2	Différents algorithmes et exemple	2
2.1	L'algorithme de Ford-Fulkerson	2
2.1.1	Fonctionnement général	2
2.1.2	Algorithme	2
2.1.3	Complexité	3
2.1.4	Avantages/inconvénients	3
2.1.5	Application sur l'exemple	3
2.2	L'algorithme d'Edmonds-Karp	4
2.2.1	Fonctionnement général	4
2.2.2	Algorithme	4
2.2.3	Complexité	4
2.2.4	Avantages/inconvénients	4
2.2.5	Application sur l'exemple	4
2.3	L'algorithme de Dinic	5
2.3.1	Fonctionnement général	5
2.3.2	Algorithme	5
2.3.3	Complexité	5
2.3.4	Avantages/inconvénients	5
2.3.5	Application sur l'exemple	5
3	Programmation linéaire	7
3.1	Matrice totalement unimodulaires	7
3.2	Solveurs de programmation linéaire	7
4	Création des instances	8
4.1	La classe RandomFlowNetwork	8
4.2	Tests sur les graphes générés	8
4.3	Représentation des instances	10
4.4	Les instances choisies	10

1 Définitions générales

1.1 Réseau résiduel

Soit G un graphe avec s , p et c respectivement la source, le puits et la fonction de capacité associés au graphe G . Notons $N = (G, s, p, c)$ un réseau de transport dans G avec un flot f .

Le réseau résiduel de N et d'un flot f , noté N_f , est construit de la manière suivante :

Pour chaque arc xy de G :

- Si $f(xy) < c(xy)$, on crée un arc xy dans G' avec la quantité restante disponible de la capacité : $c'(xy) = c(xy) - f(xy)$.
- Si $f(xy) > 0$ avec $x \neq s$ et $y \neq p$, on crée un arc yx dans G' avec la capacité qu'on peut ré-aiguiller : $c'(yx) = f(xy)$.

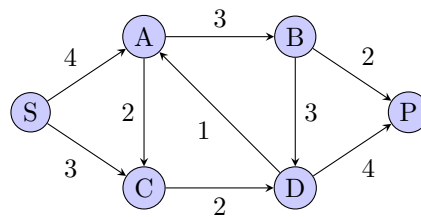
Ceci nous donne $N_f = (G', s, p, c')$.

1.2 Chemin et flot améliorants

- Un chemin améliorant pour N et f est un chemin de s à p dans le réseau résiduel N_f .
- Soit $C = x_0, x_1, \dots, x_k$ un chemin améliorant dans N_f , le flot améliorant correspondant est : $f'(xy) = \begin{cases} \gamma & \text{si } xy \text{ est un arc de } C \\ 0 & \text{sinon.} \end{cases}$ avec $\gamma = \min\{c'(x_i x_{i+1}); i \in \llbracket 0; k-1 \rrbracket\}$

2 Différents algorithmes et exemple

Nous prendrons l'exemple suivant pour illustrer les différents algorithmes :



2.1 L'algorithme de Ford-Fulkerson

2.1.1 Fonctionnement général

L'algorithme de Ford-Fulkerson est le plus connu d'entre tous. Celui-ci consiste à trouver un chemin améliorant entre la source et le puits afin d'augmenter la valeur du flot. Pour cela nous aurons besoin du graphe résiduel de G et f qui nous permettra de savoir les capacités restantes disponibles.

2.1.2 Algorithme

Pour tout arc xy du graphe G : on initialise la valeur du flot de xy à 0.
Tant qu'il existe un chemin améliorant C dans le graphe résiduel de G et f :

- Calculer le flot améliorant f' correspondant.

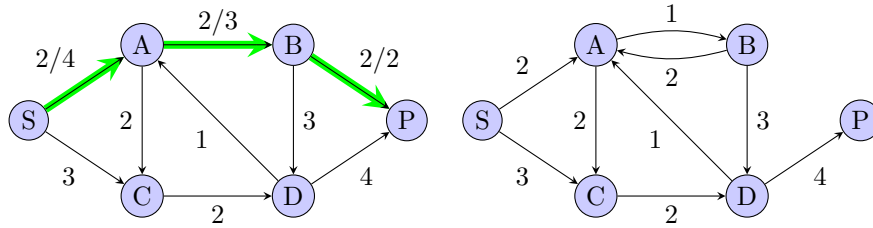
— Augmenter f par f'
Retourner f .

2.1.3 Complexité

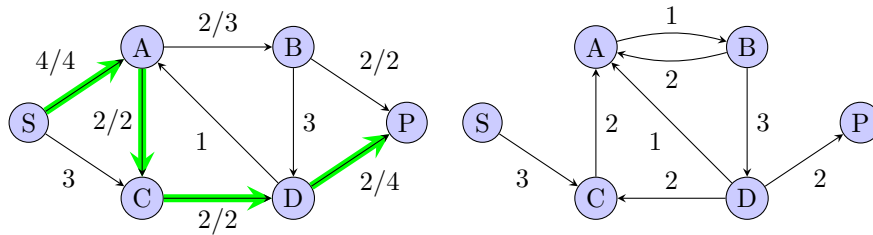
2.1.4 Avantages/inconvénients

2.1.5 Application sur l'exemple

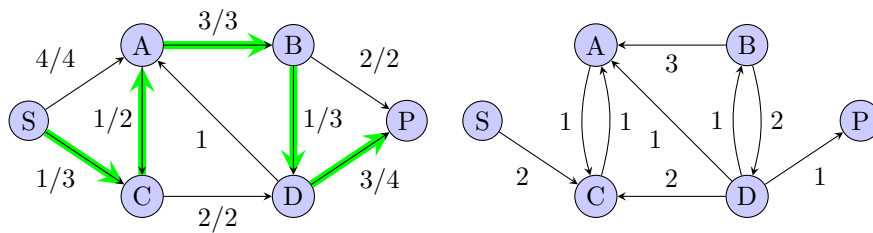
Tout d'abord, on a le chemin améliorant : $SABP$ avec une valeur de flot améliorant égal à 2.
On obtient donc à droite le réseau résiduel N_f :



On voit qu'il y a le chemin améliorant $SACDP$ avec un flot améliorant égal à 2.
On obtient à droite le nouveau réseau résiduel N_f :



On voit qu'il y a le chemin améliorant $SCABDP$ avec un flot améliorant égal à 1.
On obtient à droite le nouveau réseau résiduel N_f :



Dans ce dernier réseau résiduel, si nous partons de la source S nous ne pouvons atteindre que les sommets A et C . Comme nous ne pouvons plus atteindre le puits P , cela signifie que le flot obtenu est maximal.

Finalement la coupe minimale de ce réseau est l'ensemble $\{S, A, C\}$ et son flot maximal est de valeur 5.

2.2 L'algorithme d'Edmonds-Karp

2.2.1 Fonctionnement général

L'algorithme d'Edmonds-Karp est une spécificité de celui de Ford-Fulkerson. Celui-ci consiste à trouver le plus court chemin améliorant entre la source et le puits afin d'augmenter la valeur du flot. Pour le trouver, il suffira d'utiliser un parcours en largeur.

2.2.2 Algorithme

Pour tout arc xy du graphe G : on initialise la valeur du flot de xy à 0.
 Tant qu'il existe un plus court chemin améliorant C dans le graphe résiduel de G et f :
 — Calculer le flot améliorant f' correspondant.
 — Augmenter f par f'
 Retourner f .

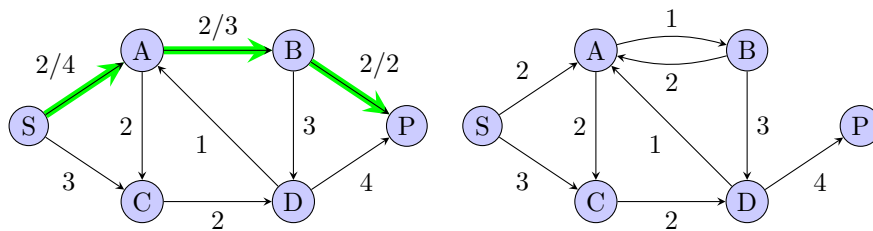
2.2.3 Complexité

2.2.4 Avantages/inconvénients

2.2.5 Application sur l'exemple

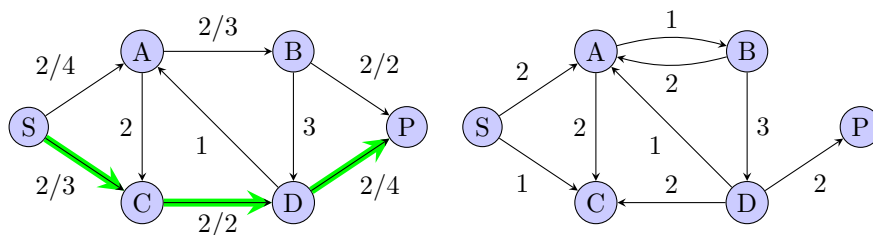
Tout d'abord, on a le chemin améliorant : $SABP$ de taille 4 avec une valeur de flot améliorant égal à 2.

On obtient donc à droite le réseau résiduel N_f :



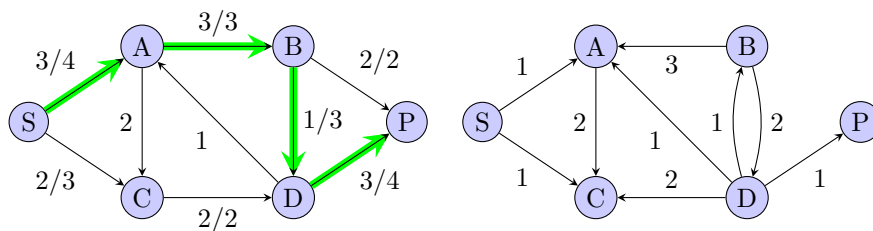
On voit qu'il y a le chemin améliorant $SACDP$ de taille 5 avec un flot améliorant égal à 2. Regardons si on trouve un chemin améliorant plus petit, le chemin $SCDP$ est un autre chemin améliorant de taille 4 avec un flot améliorant égal à 2. C'est ce chemin que nous choisissons.

On obtient à droite le nouveau réseau résiduel N_f :



Il n'y a plus de chemin améliorant de taille 4 allant de S à P . On va choisir un chemin améliorant de taille 5 : $SABDP$ avec un flot améliorant égal à 1.

Nous obtenons à droite le nouveau réseau résiduel N_f :



Dans ce dernier réseau résiduel, si nous partons de la source S nous ne pouvons atteindre que les sommets A et C . Comme nous ne pouvons plus atteindre le puits P , cela signifie que le flot obtenu est maximal.

Finalement la coupe minimale de ce réseau est l'ensemble $\{S, A, C\}$ et son flot maximal est de valeur 5.

2.3 L'algorithme de Dinic

2.3.1 Fonctionnement général

L'algorithme de Dinic est semblable à celui d'Edmonds-Karp. Comme lui, il utilise des plus courts chemins améliorants entre la source et le puits afin d'augmenter la valeur du flot.

Pour les trouver, il faudra renommer les sommets en fonction de leur distance par rapport à la source et garder les arcs qui relient un sommet à un sommet de distance immédiatement supérieure. Ceci nous donnera le réseau de niveau N_L obtenu à partir du réseau résiduel N_f .

Nous aurons également besoin du flot bloquant. Il est défini comme suit : un flot est bloquant si $\forall C$ chemin entre la source et le puits, $\exists xy$ un arc dans C où $f(xy) = c(xy)$.

2.3.2 Algorithme

Pour tout arc xy du graphe G : on initialise la valeur du flot de xy à 0.

Tant qu'il existe un plus court chemin améliorant C dans le graphe résiduel de G et f :

- On détermine le réseau de niveau N_L de N_f .
- Calculer le flot bloquant f' correspondant.
- Augmenter f par f'

Retourner f .

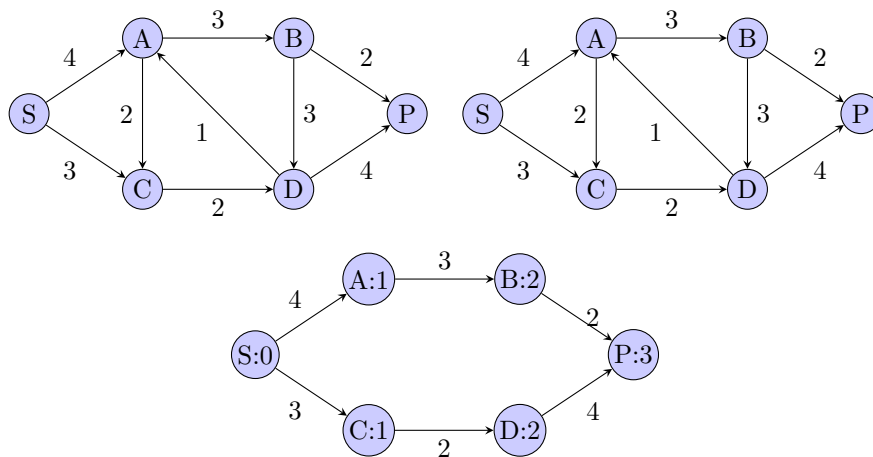
2.3.3 Complexité

2.3.4 Avantages/inconvénients

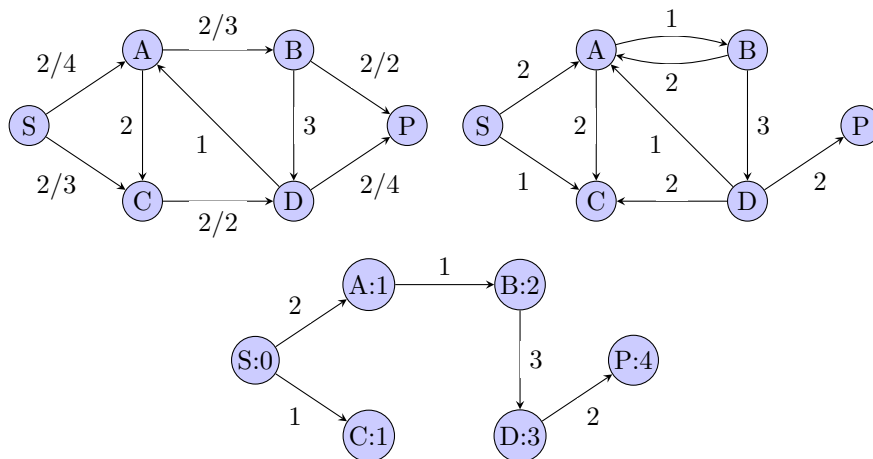
2.3.5 Application sur l'exemple

Au début le flot est nul, le réseau résiduel N_f à droite est donc le même que le réseau de transport N .

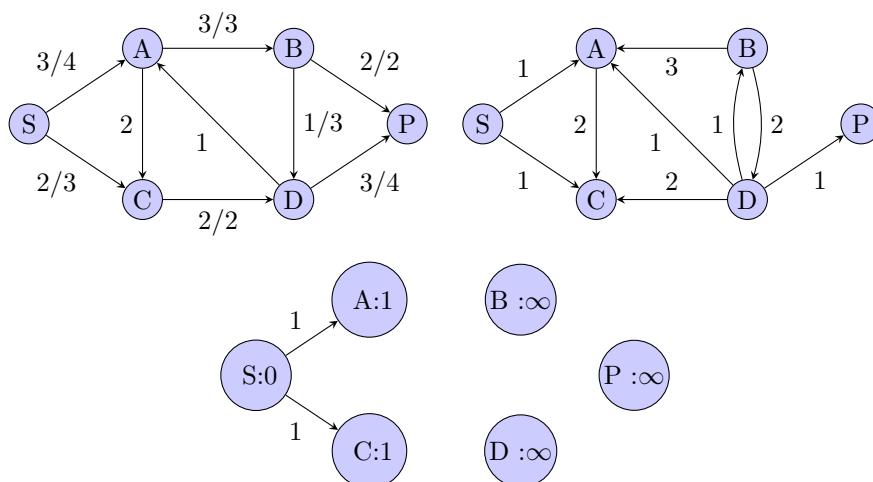
Nous avons aussi en dessous, le réseau de niveau N_L :



On a donc un chemin améliorant $SABP$ de valeur 2 ainsi qu'un chemin améliorant $SCDP$ de valeur 2.



Nous avons un chemin améliorant $SABDP$ de valeur 1.



Le puits P n'est plus atteignable, l'algorithme se termine.
Finalement la coupe minimale de ce réseau est l'ensemble $\{S, A, C\}$ et son flot maximal est de valeur 5.

3 Programmation linéaire

Nous avons également comparé nos algorithmes avec les meilleurs solveurs de programmation linéaire. Pour cela nous avons utilisé la modélisation standard du problème de flot maximum :

$$\begin{aligned} \max \quad & z = \sum_{v:(s,v) \in E} f(s, v) \\ \left\{ \begin{array}{ll} \sum_{u:(u,v) \in E} f(u, v) - \sum_{w:(v,w) \in E} f(v, w) = 0 & \forall v \in V \setminus \{s, t\} \\ f(u, v) \leq c(u, v) & \forall (u, v) \in E \\ f(u, v) \geq 0 & \forall (u, v) \in E \end{array} \right. \end{aligned}$$

La fonction à maximiser est la somme des flots sortants du sommet source s , par conservation du flot, c'est aussi égal à la somme des flots entrants dans le sommet puits. La première contrainte assure la conservation du flot pour chaque sommet, sauf pour la source et le puits. La deuxième contrainte assure que le flot sur chaque arête ne dépasse pas sa capacité et la dernière contrainte assure que le flot est positif ou nul.

Une remarque importante est que les variables de flot $f(u, v)$ sont continues, donc à priori, rien n'assure que la solution optimale soit entière.

3.1 Matrice totalement unimodulaires

Définition Soit M une matrice, on dit qu'elle est totalement unimodulaire si pour toute matrice carrée N de M , le déterminant de N est égal à 0, 1 ou -1 .

Théorème (Hoffman & Kruskal, 1956) Une matrice entière est totalement unimodulaire si et seulement si le polyèdre défini par $\{x : Ax \leq b, x \geq 0\}$ est un polyèdre entier pour tout vecteur b entier.

Ce que nous dit ce théorème, c'est que tout problème linéaire dont la matrice de contraintes est totalement unimodulaire et dont le vecteur de contraintes est entier, a une solution entière.

Théorème La matrice du problème de flot maximum est totalement unimodulaire.

Et comme le vecteur de contraintes est entier (car les capacités sont entières), la solution optimale est entière.

L'intérêt de ce théorème est que, bien que la programmation linéaire en nombres entiers soit NP-complet, dans le cas du flot maximum, on peut utiliser un solveur de programmation linéaire classique pour obtenir une solution entière.

3.2 Solveurs de programmation linéaire

Nous avons utilisé deux solveurs de programmation linéaire : **Gurobi** et **GLOP**, ces deux solveurs sont très performants et permettent de résoudre des problèmes de grande taille rapidement.

4 Création des instances

Afin de pouvoir tester et comparer nos algorithmes, nous avons eu besoin d'instances de graphe de tailles et de formes différentes afin de pouvoir comparer comment se comportent nos algorithmes. Comme nous n'avons pas trouvé d'instances de graphes suffisamment grande en ligne, nous avons décidé de créer nos propres instances. Cette partie détaille ce processus de création et présente les différentes instances que nous avons pu créer ainsi que les choix que nous avons fait.

4.1 La classe `RandomFlowNetwork`

Afin de créer un grand nombre d'instances, nous avons commencé par générer des graphes de manière aléatoire. Pour cela nous avons créé une classe `RandomFlowNetwork`.

Cette classe permet de générer un réseau de flux orienté en plusieurs étapes :

1. Elle crée un graphe vide et commence par lui ajouter l'ensemble de ses sommets, chacun identifié par un nom ou un numéro. Le nombre de sommet n créé est contrôlé à l'avance.
2. Ensuite, elle établit des arêtes orientées entre ces sommets. Ces connexions sont ajoutées de manière aléatoire, tout en respectant certaines contraintes :
 - Soit on fixe à l'avance un nombre précis de connexions.
 - Soit on utilise une densité, c'est-à-dire une proportion du nombre maximal de connexions possibles.

Ici dans le cas de nos instances, cette dernière méthode est utilisée. Pour expliquer plus clairement comment cela fonctionne, pour un graphe $G = (E, V)$ on pose $d \in [0, 1]$ la densité, et ensuite $\forall x, y \in V$ la probabilité d'apparition de l'arête xy est d . On essaie donc de créer le graphe complet avec une probabilité d d'apparition sur chaque arête.

Nous avons ainsi pu faire varier la densité de nos graphes entre 0.1 et 0.9.

Chaque arête reçoit finalement une capacité aléatoire, c'est-à-dire une valeur représentant sa "capacité" ou "poids".

3. Une fois les arêtes ajoutées, deux sommets distincts sont choisis au hasard : un sommet source (point de départ) et un sommet puits (point d'arrivée).
4. Les arêtes qui partent directement de la source et celles qui arrivent directement au puits sont récupérées et stockées à part.
5. Finalement, le graphe est retourné avec :
 - Le sommet source ;
 - Le sommet puits ;
 - Les arêtes entre les sommets.

En résumé, cette classe construit un graphe orienté aléatoire, puis désigne deux extrémités pour simuler un réseau de flux. Le problème est le suivant : comment construire des graphes pertinents afin de tester les algorithmes ?

4.2 Tests sur les graphes générés

Les graphes étant générés aléatoirement, on peut obtenir des graphes non connexes, des sommets non reliés à la source ou au puits, etc. Tout cela fausserait les instances. En effet, si le graphe n'est pas connexe, cela pourrait fausser les résultats : un graphe de taille 1000 pourrait se résumer à un de taille 100, ce que l'on ne souhaite pas. On pourrait même tomber sur des graphes où nos algorithmes ne convergeraient pas, par exemple si la source et le puits ne sont

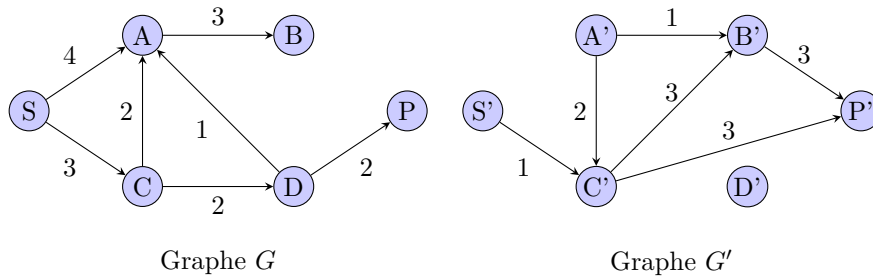
pas reliés.

On va donc devoir créer un ensemble de tests sur nos graphes afin de s'assurer qu'ils soient bien formés et propices à être utilisés comme instances. Cela pourra prendre du temps car nous générerons les graphes aléatoirement et n'en gardons qu'une petite partie, mais ce n'est pas très grave, car le jeu d'instances n'a besoin d'être créé qu'une seule fois. Il n'est pas nécessaire que le code qui permet sa création soit bien optimisé : on le lance un soir et on récupère les instances le lendemain dans le pire des cas.

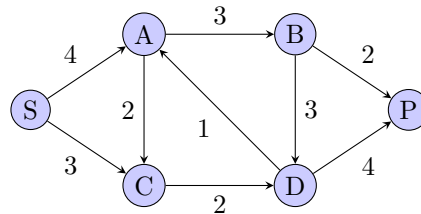
Nous avons donc implémenté une fonction **IsConnected** qui permet de vérifier les trois critères suivants sur le graphe :

1. Depuis la source, on peut atteindre n'importe quel sommet du graphe.
2. Depuis n'importe quel sommet du graphe, on peut atteindre le puits.
3. Le graphe est connexe.

Cela nous permet de nous assurer que le graphe que nous avons généré est bien formé. En effet, l'algorithme convergera car le premier point nous assure qu'on peut créer un flot sur ce graphe (la source est bien connectée au puits). Les points 2 et 3 nous permettent d'être sûrs que sur un graphe de taille n , chaque sommet aura bien son importance et sera bien pris en compte par le calcul. Le graphe ne peut donc pas se résumer à une version plus petite de lui-même.



Ainsi le graphe G ne sera pas pris car on ne peut pas accéder au puit depuis les sommets A et B . Le graphe G' quand à lui ne sera pas pris car il n'est pas connexe (le sommet D' est isolé), aussi on ne peut pas accéder au sommet A' depuis la source S' . En revanche, le graphe suivant est tout à fait valide :



Nous avons ainsi pu créer un jeu d'instances complet. Afin de généraliser davantage et de mieux comparer les résultats, plusieurs instances similaires (ayant la même taille et la même densité) sont créées. Pour connaître l'efficacité d'un algorithme sur une instance de ce type, on peut ainsi faire la moyenne des performances sur l'ensemble des instances de même taille et même densité.

4.3 Représentation des instances

Nous avons décidé de représenter les instances sous forme de fichiers texte. Chaque instance est représentée par un fichier contenant les informations suivantes :

- Le numéro de la source ;
- Le numéro du puits ;
- La liste des arcs du graphe, sous la forme : $x \ y \ c(xy)$ où x et y sont les numéros des sommets de l'arc, et $c(xy)$ est la capacité de l'arc.

Le nombre de sommets ainsi que la densité du graphe sont indiqués dans le nom du fichier. Par exemple, voici les premières lignes du fichier `inst30_0,1_1`, où 30 représente le nombre de sommets, 0,1 la densité, et 1 le numéro de l'instance (car plusieurs instances sont créées pour un même nombre de sommets et une même densité).

```
22
4
17 21 9
24 25 8
12 27 55
...
```

4.4 Les instances choisies

Nous avons créé des instances pour toutes les tailles multiples de 10 entre 30 et 200. Pour chaque taille, nous avons utilisé trois densités : 0,1, 0,5 et 0,9. Pour chaque densité, trois instances différentes ont été générées. Cela représente au total plus de 150 instances pour tester nos programmes. Nous avons choisi de rester sur des tailles raisonnables afin que tous les algorithmes, même les moins efficaces, puissent toujours converger et ainsi être comparés. En effet, on constate que pour une taille de 200 et une densité de 0,9, on a en moyenne :

$$\frac{200 \times (200-1)}{2} \times 2 \times 0,9 = 35820 \text{ arêtes}$$

Le $\frac{200 \times (200-1)}{2} \times 2$ correspondant à toutes les arêtes possibles. En effet $\frac{n \times (n-1)}{2}$ pour le graphe complet, $\times 2$ car les arêtes sont orientées. On multiplie par la densité pour savoir le nombre d'arête en moyenne, dans notre exemple la densité est de 0,9.