Résolution du problème du flot maximum

Auteurs

Cédric Audie
Marie Dalenc
Julien Lahoz
Adrien Martinelli

Encadrant

Rodolphe Giroudeau

10 mai 2025

Résumé

Le problème du flot maximum est un problème d'optimisation classique en informatique et en recherche opérationnelle. Étant donné un réseau de flot, l'objectif est de déterminer la quantité maximale de flot qui peut être envoyée d'une source à un puits, tout en respectant les contraintes de capacité des arcs et de conservation du flot. Ce problème a de nombreuses applications pratiques, notamment dans la planification logistique, les réseaux de transport et les réseaux de communication.

Dans ce travail, nous comparons différentes méthodes de résolution du problème du flot maximum. Nous étudions les algorithmes classiques tels que Ford-Fulkerson, Edmonds-Karp, Poussage-Réétiquetage et Dinic, ainsi que des approches basées sur la programmation linéaire. Nous mettons en œuvre ces algorithmes et les testons sur un ensemble de graphes générés aléatoirement. Nous analysons les performances de chaque méthode en termes de complexité théorique et de temps d'exécution pratique.

Table des matières

1 Introduction

Le problème du flot maximum est l'un des problèmes fondamentaux en algorithmique et en recherche opérationnelle. Il consiste à trouver, dans un réseau orienté avec des capacités sur les arcs, le flot de valeur maximale entre une source et un puits, tout en respectant les contraintes de capacité et de conservation des flots.

Depuis plusieurs décennies, de nombreux algorithmes ont été développés pour résoudre ce problème, parmi lesquels on compte les méthodes classiques comme Ford-Fulkerson, Edmonds-Karp ou encore Dinic, mais aussi des approches basées sur la programmation linéaire. Ces algorithmes diffèrent à la fois par leur complexité théorique, leur comportement pratique et leur adaptabilité à différents types de graphes.

L'objectif de ce travail est de comparer ces différentes méthodes de résolution, à la fois sur le plan théorique et expérimental. Nous mettrons en œuvre ces algorithmes, puis les testerons sur des jeux d'essais variés pour évaluer leurs performances respectives.

Ce rapport est structuré comme suit : après une présentation des méthodes considérées, nous décrivons notre méthodologie expérimentale, puis nous analysons les résultats obtenus avant de conclure sur les perspectives ouvertes par ce travail.

1.1 Présentation détaillée du problème de flot maximum

Le problème du flot maximum se définit sur un graphe orienté $G = (V, E)$, où V est l'ensemble des sommets et E l'ensemble des arcs. Chaque arc $(u, v) \in E$ est associé à une capacité $c(u, v) \geq 0$, représentant la quantité maximale de flot qui peut circuler de u vers v. Si $(u, v) \notin E$, on suppose que $c(u, v) = 0$. On désigne par $s \in V$ la source et par $t \in V$ le puits, avec $s \neq t$.

Un *flot* est une fonction $f : V^2 \to \mathbb{R}$ qui satisfait les contraintes suivantes :

— **Capacité** : $\forall (u, v) \in V^2,\ 0 \leq f(u, v) \leq c(u, v)$;

— **Anti-symétrie** : $\forall (u, v) \in V^2,\ f(u, v) = -f(v, u)$;

— **Conservation** : $\forall v \in V \setminus \{s, t\},\ \sum_{(u,v)\in E} f(u, v) = \sum_{(v,z)\in E} f(v, z)$, c'est-à-dire que tout sommet intermédiaire conserve le flot (pas de perte ni de création de matière) ;

— **Valeur du flot** : la quantité totale envoyée par la source est $|f| = \sum_{v\in V} f(s, v)$, et c'est cette valeur que l'on cherche à maximiser.

On appelle *réseau de flot* un quadruplet (G, s, t, c), où :

— $G = (V, E)$ est un graphe orienté,

— $s \in V$ est la source,

— $t \in V$ est le puits ($s \neq t$),

— $c : V^2 \to \mathbb{R}^+$ est une fonction de capacité sur les arcs, et qui est nulle si l'arc n'existe pas.

Le problème du flot maximum consiste donc à trouver un flot f respectant les contraintes ci-dessus et maximisant $|f|$. Dans notre cas particulier, nous allons nous intéresser à des graphes orientés avec un flot entier et des capacités entières.

Applications : Le flot maximum possède de nombreuses applications concrètes : planification logistique, réseaux de transport (par exemple pour maximiser le trafic dans un réseau routier), routage dans les réseaux de communication, allocation de ressources, et même en algorithmique pour résoudre d'autres problèmes puisque le problème du flot maximum est P-COMPLET. C'est-à-dire qu'il se résout en temps polynomial, et que tout problème polynomial peut s'y réduire par une réduction en espace logarithmique.

1.2 Exemple

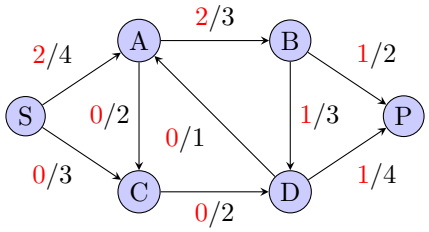


FIGURE 1 – Exemple de réseau de flot, avec en **rouge** un flot et en **noir** la capacité de chaque arc.

Dans cet exemple, le sommet A reçoit deux unités depuis la source et les envoies au sommet B. Le sommet B redistribue ce flot vers les sommets D et P. La valeur du flot est de 2, c'est la somme des flots sortants de la source S (et également la somme des flots entrants dans le puits).

2 Définitions générales

2.1 Réseau résiduel

Definition (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 flot 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 réaiguiller : $c'(yx) = f(xy)$.

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

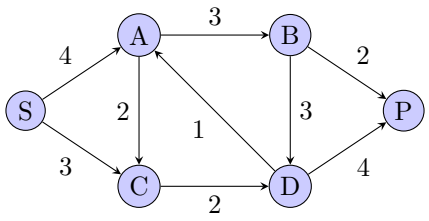
2.2 Chemin et flot améliorants

Definition (Chemin améliorant).

- Un chemin améliorant pour N et f est un chemin de s à p dans la réseau résiduel N_f.
- Soit $C = x_0, x_1, ..., x_k$ 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 $\gamma = \min\{c'(x_i x_{i+1}); i \in [\![0; k-1]\!]\}$

3 Différents algorithmes et exemple

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



3.1 L'algorithme de Ford-Fulkerson

3.1.1 Fonctionnement général

L'algorithme de Ford-Fulkerson est le plus connu parmi 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.

3.1.2 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'

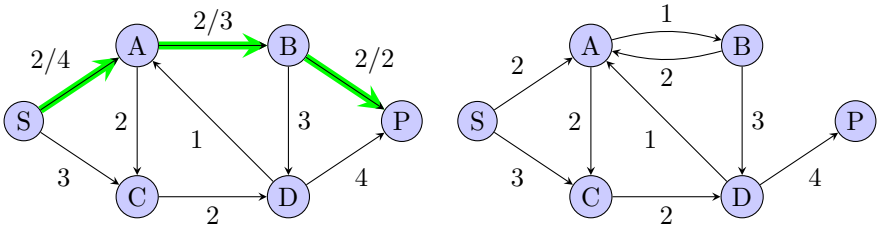
Retourner f.

3.1.3 Complexité

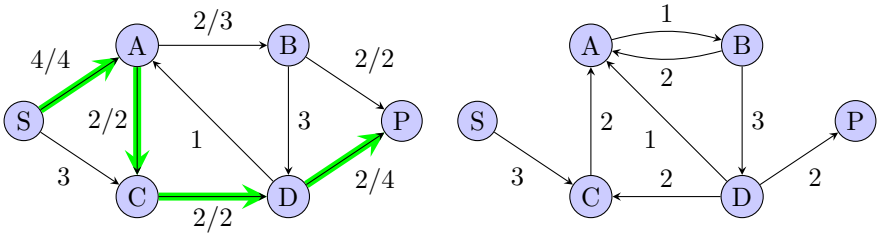
3.1.4 Avantages/inconvénients

3.1.5 Application sur l'exemple

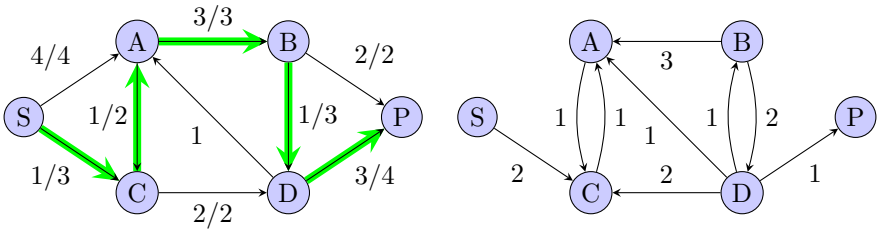
Tout d'abord, on a le chemin améliorant : $SABP$ avec une valeur de flot améliorant égal à 2.
On obtient donc à droite le réseau résiduel N_f :



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.

3.2 L'algorithme d'Edmonds-Karp

3.2.1 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 suffit d'utiliser un parcours en largeur.

3.2.2 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'

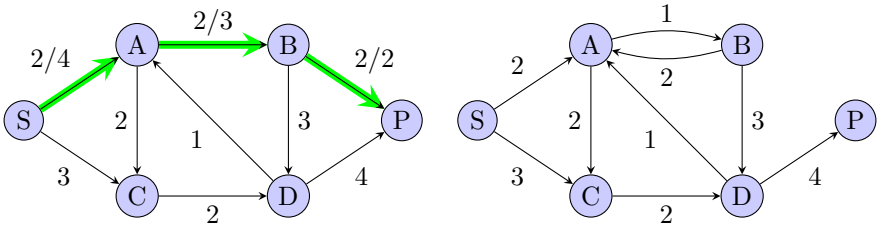
Retourner f.

3.2.3 Complexité

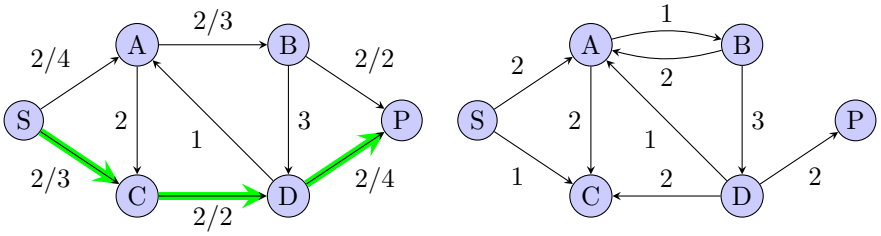
3.2.4 Avantages/inconvénients

3.2.5 Application sur l'exemple

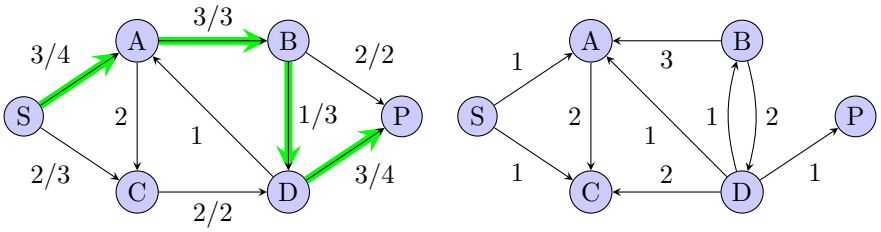
Tout d'abord, on a le chemin améliorant : $SABP$ de taille 3 avec une valeur de flot améliorant égal à 2.
On obtient donc à droite le réseau résiduel N_f :



On voit qu'il y a le chemin améliorant $SACDP$ de taille 4 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 3 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 3 allant de S à P. On va choisir un chemin améliorant de taille 4 : $SABDP$ 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.

3.3 L'algorithme de Dinic

3.3.1 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 étiqueter 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 $f(xy) = c(xy)$.

3.3.2 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'

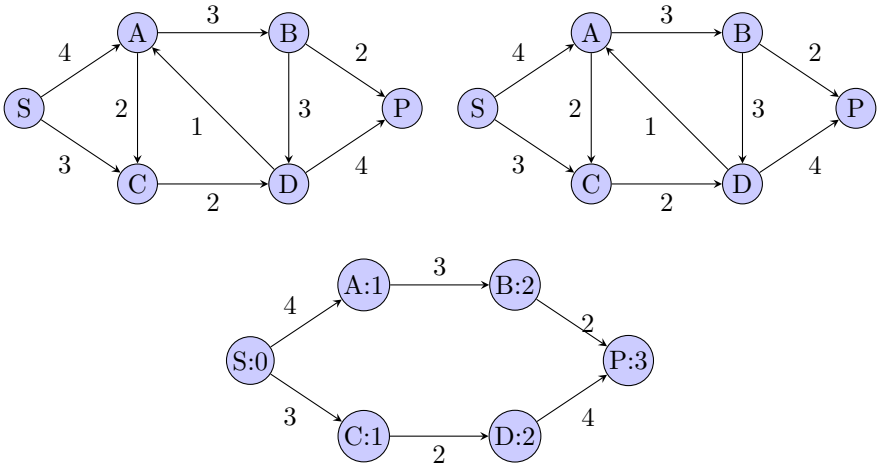
Retourner f.

3.3.3 Complexité

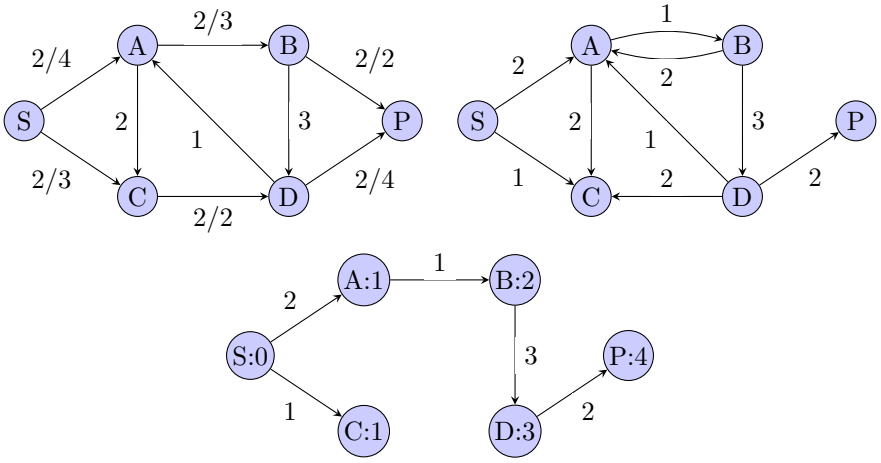
3.3.4 Avantages/inconvénients

3.3.5 Application sur l'exemple

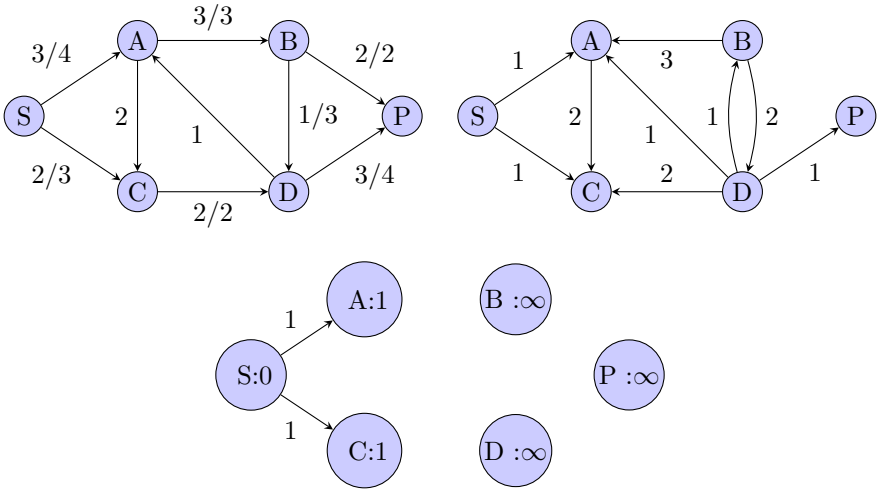
Au début le flot est nul, le réseau résiduel N_f à droite est donc le même que le réseau de flot N.
Nous avons aussi en dessous, le réseau de niveau N_L :



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.

4 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 :

$$\text{Maximiser} \quad \sum_{v:(s,v)\in E} f(s,v)$$

$$\text{Sous les contraintes} \quad \begin{cases} \sum_{u:(u,v)\in E} f(u,v) - \sum_{w:(v,w)\in E} f(v,w) = 0 & \forall v \in V \setminus \{s,t\} \\ f(u,v) \leq c(u,v) & \forall (u,v) \in E \\ f(u,v) \geq 0 & \forall (u,v) \in E \end{cases}$$

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 a priori, rien n'assure que la solution optimale soit entière.

4.1 Matrices totalement unimodulaires

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

Théorème (Hoffman & Kruskal, 1956[1]). Une matrice entière est totalement unimodulaire si et seulement si le polyhèdre défini par $\{x : Ax \leq b, x \geq 0\}$ est un polyèdre entier pour tout vecteur b entier.

Ce que nous apprend 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.

4.2 Solveurs de programmation linéaire

Nous avons utilisé deux solveurs de programmation linéaire : `Gurobi` et `SCIP`, ces deux solveurs sont très performants et permettent de résoudre des problèmes de grande taille rapidement. `Gurobi` est un solveur commercial, réputé pour être l'un des plus rapides du marché, tandis que `SCIP` est un solveur open-source qui est probablement le meilleur solveur open-source.

5 Création des instances

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

5.1 La classe RandomFlowNetwork

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

Cette classe permet de générer un réseau de flot 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 crée est controlé 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 derniere 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 arêtes.
   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 récupérée 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 flot. Le problème est le suivant : comment construire des graphes pertinents afin de tester les algorithmes ?

5.2 Tests sur les graphes générés

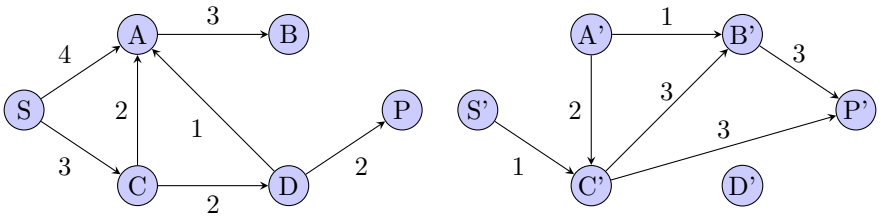
Les graphes étant générés aléatoirement, on peut obtenir des graphes non connexes, des sommets non reliés à la source ou au puits, etc. Tout cela fausseraient les instances. En effet, si le graphe n'est pas connexe, cela pourrait fausser les résultats : un graphe de taille 1000 pourrait se résumer à 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 utilisées comme instances. Cela pourra prendre du temps car nous générons les graphes aléatoirement et n'en conservons 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 :

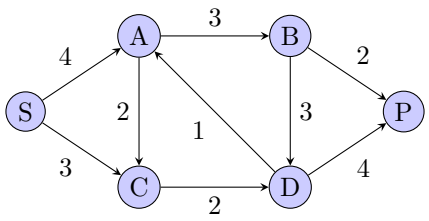
1. Depuis la source, on peut atteindre n'importe quel sommet du graphe.
2. Depuis n'importe quel sommet du graphe, on peut atteindre le puits.
3. Le graphe est connexe.

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.



Graphe G Graphe G'

Ainsi le graphe G ne sera pas pris car on ne peux 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 peux 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.

5.3 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
...
```

5.4 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, pour un graphe de taille de 200 sommets et une densité de 0,9, on obtient 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 orientés. On multiplie par la densité pour savoir le nombre d'arête en moyenne, dans notre exemple la densité est de $0,9$.

6 Méthodologie Expérimentale

7 Résultats et Discussion

8 Revue de la Littérature

9 Conclusion et Perspectives

Références

[1] A. J. Hoffman et J. B. Kruskal. *13. Integral Boundary Points of Convex Polyhedra*. Déc. 1957/1958, p. 223-246. DOI : 10.1515/9781400881987-014. URL : https://doi.org/10.1515/9781400881987-014.