Projet d'algorithmique
Recherche d'un flot maximum dans un réseau

Samy Braik

19 novembre 2023

1 Cadre d'études

Nous étudierons les réseaux de flot, c'est-à-dire les graphes orientés où chaque arc possède une capacité et peut recevoir un flot.
Certaines contraintes s'appliquent aux flots; sur chaque arc un flot ne peut pas excéder la capacité et la somme des flots entrant un sommet du graphe est égale à la somme des flots le quittant.
Ainsi, plus formellement, en considérant $G = (V, E)$ un graphe orienté pondéré avec V la liste de ses sommets, E la liste de ses arcs et $c : E \to \mathbb{R}_+$ qui définie les capacités des arcs.
Un flot est une fonction $f : V^2 \to \mathbb{R}^+$ tel que :

- $\forall (u, v) \in E,\ f(u, v) \leq c(u, v)$
- $\forall u \in V \setminus \{s, t\}, n \in \mathbb{N},\ \sum_{i=1}^{n} f(u, v_i) - \sum_{i=1}^{n} f(v_i, u) = 0$ tel que $\forall i \in \{1, ..., n\}$, $(u, v_i), (v_i, u) \in E^2$

On désigne un sommet s, qui n'est l'arrivée d'aucun arc, comme la source et un noeud t, qui n'est le départ d'aucun arc, comme le puits.
Le problème que l'on cherche alors à résoudre est de maximiser le flot quittant s (qui est par définition le même flot entrant t) en satisfaisant les contraintes précédemment définies.
Nous considérons uniquement les graphes où V et E ne sont pas vide.

2 Algorithme d'Edmonds-Karp

Jack Edmonds et Richard Karp publie un article en avril 1972 dans le *Journal of the Association for Computing Machinery*. Dans cet article, J.Edmonds et R.Karps donne un algorithme qui améliore celui proposé par Ford et Fulkerson en 1956 pour déterminer le flot maximum d'un graphe.
Avant de donner la proposition d'algorithme, il faut définir quelques notions.
En considérant le même cadre défini dans la première partie, avec f un flot quelconque, on définit un chemin augmentant comme un chemin entre s et t tel que :

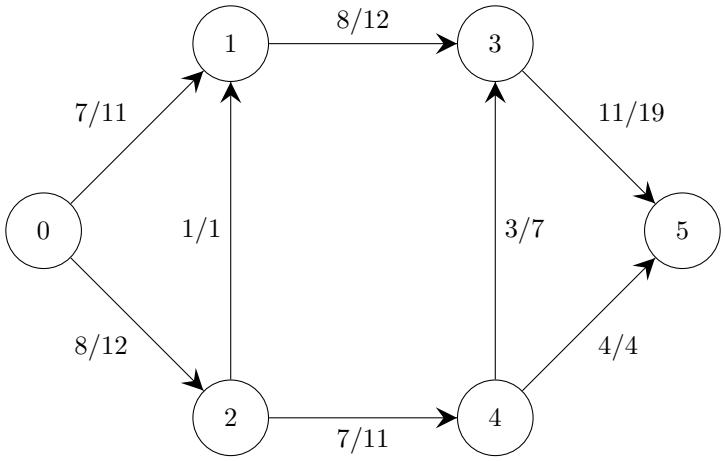
- Si $(u_i, u_{i+1}) \in A$ et $(u_{i+1}, u_i) \notin A$ alors $c(u_i, u_{i+1}) - f(u_i, u_{i+1}) > 0$
- Si $(u_i, u_{i+1}) \notin A$ et $(u_{i+1}, u_i) \in A$ alors $f(u_{i+1}u_i) > 0$

Avec $u_i \in V,$ pour tout $0 \leq i \leq n - 1,\ n \in \mathbb{N}$
De plus, on définit le graphe résiduel G_f, comme le graphe partageant les mêmes sommets et arcs que G, mais incorporant également les arcs inversés, où sur chacun de ses arcs on retrouve la capacité résiduelle représentée par $r : V^2 \to \mathbb{R}$ définie par

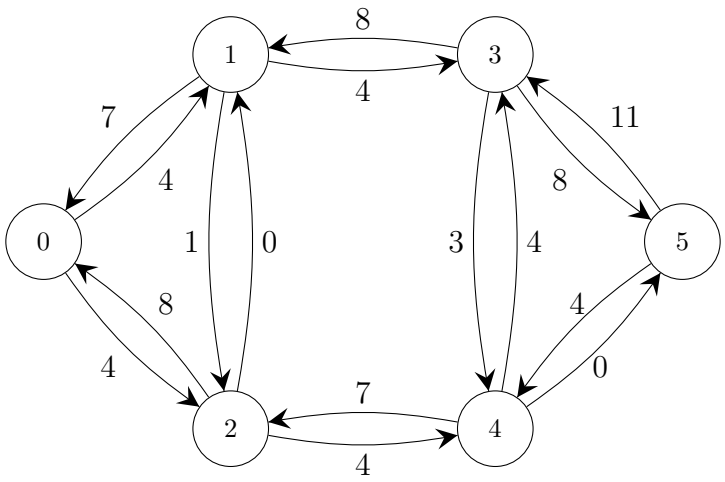
$$r(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{si } (u, v) \in E \\ f(v, u) & \text{si } (u, v) \notin E \end{cases}$$

Pour illuster ces notions on se donne le graphe suivant :
$G = (\{0, 1, 2, 3, 4, 5\}, \{(0, 1), (0, 2), (1, 3), (2, 1), (2, 4), (3, 5), (4, 3), (4, 5)\})$

Graphe G avec sur chaque arc un couple flot/capacité



Graphe résiduel G_f de G avec sur chaque arc la capacité résiduelle



On peut alors identifier un chemin augmentant comme un chemin de s à t dans le graphe résiduel où chaque arc emprunté est pondéré par une valeur strictement positive.
Ainsi, J.Edmonds et R.Karp propose l'algorithme suivant :

- On commence par initialiser la valeur du flot à 0.
- On détermine un plus court chemin augmentant en utilisant un parcours en largeur et on remplace les flots le long des arcs empruntés par la capacité résiduelle minimale rencontrée dans le parcours.
- On itère jusqu'à ce qu'il n'y ai plus de chemin augmentant.

La différence dans l'algorithme d'Edmonds-Karp par rapport à celui de Ford-Fulkerson est dans le choix du chemin augmentant. Ici on utilise un parcours en largeur pour déterminer un chemin de s à t dans le graphe résiduel, là où Ford-Fulkerson ne propose pas de manière de trouver un chemin augmentant.

3 Complexité

Soit $G = (V, E)$ un graphe orienté, s la source et t le puits.
L'algorithme d'Edmonds-Karp s'exécute en temps $O(|V| \cdot |E|^2)$.

Soit $(u, v) \in V^2 \setminus \{s, t\}$ et $d : V^2 \to \mathbb{R}_+$ définie par $d(a, b) =$ la distance entre le sommet a et b dans le graphe résiduel. On désigne par $|V|$ le nombre de sommets du graphe et $|E|$ le nombre d'arcs. De plus, on dit qu'un arc est saturé si sa capacité est égale à 0 dans le graphe résiduel.
Supposons que (u, v) soit l'arc saturé après une augmentation initiale du flot. Par définition, on a que $d(s, v) = d(s, u) + d(u, v) = d(s, u) + 1$.
Supposons alors une seconde augmentation du flot telle que (v, u) soit l'arc saturé et appelons d' la distance dans le nouveau graphe résiduel. On a donc, sachant que l'on choisit le chemin le plus court, $d'(s, v) \geq d(s, v) \implies d'(s, v) \geq d(s, u) + 1$ et de la même manière que précédemment, $d'(s, u) = d'(s, v) + 1$ d'où $d'(s, u) \geq d(s, u) + 2$.
Ainsi, lorsque l'on augmente le flot en passant par u, la longueur du plus court chemin entre la source s et u augmente d'au moins 2. Or, la longueur d'un plus court chemin entre s et u est borné par $|V|$ puisque dans le pire des cas le chemin le plus court pour joindre deux arcs est de parcourir tous les arcs du graphe. Par conséquent l'arc (u, v) peut être saturé au maximum $\frac{|V|}{2}$ fois.
D'autre part, il y a au maximum $2|E|$ arcs dans le graphe résiduel, ainsi il y a potentiellement $2|E|$ arcs pouvant être saturée.
Par conséquent, en utilisant le résultat précèdent, le nombre de chemin augmentant est borné par $\frac{|V|}{2} \cdot 2|E| = |V| \cdot |E|$.
Par hypothèse, $\exists K > 0$ tel que $|V| \leq K \cdot |E|$ puisque $|V| \neq 0$ et $|E| \neq 0$.
Ainsi, on effectue le parcours en largeur, qui se fait en $O(|V| + |E|) = O(|E|)$, un nombre $O(|V| \cdot |E|)$ de fois.
On obtient alors la complexité finale de l'algorithme qui est en $O(|V| \cdot |E|^2)$.

4 Preuve de l'algorithme

Soit $G = (V, E)$ un graphe orienté, s la source et t le puits.
Pour faire la preuve de l'algorithme, on utilise le théorème de flot-max/coupe-min qui nous dit que le flot est maximum si et seulement si le graphe résiduel ne contient plus de chemin augmentant.
En effet, si il existe un chemin augmentant alors le flot peut être augmenté par la valeur de la capacité résiduelle minimale rencontrée, ce qui implique que le flot n'est pas maximum donc par contraposée on a la sens direct.
Pour le sens indirect, on définit la notion de coupe c'est-à-dire une partition (A, B) de V tel que $s \in A$ et $t \in B$. Fixons alors un flot f et supposons qu'il n'existe plus de chemin augmentant, c'est-à-dire que pour tout chemin entre s et t au moins un arc est saturé. On définit alors une coupe (C, D) de tel sorte que tous les arcs (u, v) tel que $u \in C$ et $v \in D$ soient saturés. Augmenter le flot impliquerait que $f(u, v) > c(u, v)$ pour un certain $u \in C$ et $v \in D$ ce qui n'est pas possible. Donc le flot est maximum.

Au début de l'algorithme, on initialise le flot à 0.
A chaque itération de l'algorithme, il y au moins un arc appartenant à un chemin de s à t, dans le graphe résiduel, qui se retrouve saturé de tel sorte à ce que le flot augmente. Or dans la section précédente, on a vu qu'un arc ne pouvait être saturé qu'un nombre fini de fois et que ceci implique que le nombre de chemin augmentant est fini. Par conséquent l'algorithme termine lorsqu'il n'y a plus de chemin augmentant.
Ainsi, par le théorème de flot-max/coupe-min, lorsque l'algorithme termine le flot est bien maximal.

5 Implémentation de l'algorithme

```
import List
import Queue
import numpy as np


class Adj_list(List.Linked_list):
    def print(self):
        print("Liste d'adjacence = ",end='')
        current=self.head
        while current:
            edge=current.data
            print(edge.head.name, end=' ; ')
            current=current.next
        print()


class Node:
    def __init__(self,index=0,name=''):
        self.index = index
        self.name = name
        self.adj = Adj_list()


class Edge:

    def __init__(self,n):
        self.head = n


class Graph:
    def __init__(self,l=[],edg=[]):
        self.size = len(l)
        self.nodes=list()
        self._weight = np.zeros((self.size,self.size),float)
        self._flow = np.zeros((self.size,self.size),float)
        for i in range(len(l)):
            n=Node(i,l[i])
            self.nodes.append(n)
        for i in range(len(edg)):
            self.add_edge(edg[i][0],edg[i][1])
```

```
            self._weight[edg[i][0]][edg[i][1]] = edg[i][2]


    def add_edge(self,i,j):
        if i<len(self.nodes) and j<len(self.nodes):
            e=Edge(self.nodes[j])
            self.nodes[i].adj.append(e)

    def print_adj_list(self):
        for n in self.nodes:
            print('Noeud :', n.index,n.name )
            n.adj.print()


    #On implemente un parcours en largeur qui va de la source jusqu'au puit
    #Si un chemin existe et a une capacité résiduelle minimum strictement
    #positive renvoyer True sinon False
    def BFS(self,s,t,p):
        n=self.size
        c=[0]*n
        f=Queue.Queue()
        p[s]=-1
        f.enqueue(s)
        c[s]=1
        while not f.is_empty():
            u=f.dequeue()
            current=self.nodes[u].adj.head
            while current :
                v=current.data.head.index
                if not c[v] and abs(self._weight[u][v]-self._flow[u][v])>0:
                    c[v]=1
                    p[v]=u
                    if v == t:
                        return True
                    f.enqueue(v)
                current=current.next
            c[u]=2
        return False


    def EdmondsKarp(self,source,puit):
        #Construction du graphe résiduel
        l = [i for i in range(self.size)]
```

```
        residual_graph = Graph(l,[(u,v,abs(self._weight[u][v]-self._flow[u][v]))
        for u in l for v in l if u!=v])
        p = [-1]*self.size #Garde en mémoire les noeuds déjà visité
        maxFlow = 0 #Initialisation du flot à 0

        #Tant qu'il y a un chemin entre la source et le puit qui augmente le flot
        while residual_graph.BFS(source,puit,p):
            currentFlow = float("Inf")
            s = puit
            while s!=source:
                currentFlow = min (currentFlow, residual_graph._weight[p[s]][s])
                s = p[s]

            maxFlow += currentFlow

            y = puit
            while y!=source:
                x = p[y]
                residual_graph._weight[x][y] -= currentFlow
                residual_graph._weight[y][x] += currentFlow
                self._flow[x][y]+=currentFlow
                y=p[y]
        return maxFlow
```

Bibliographie

[1] L.R. Ford Jr et D.R. Fulkerson (1956), *Maximum flow through a network.*

[2] Jack Edmonds et Richard Karp (1972), *Theoritical Improvements in Algorithmic Efficiency for Network Flow Problems.*

[3] Ciryl Nicaud (2019), *Flots dans les graphes, Notes de cours d'algorithmique avancée.*

[4] Evripidis Bampis (2023), *Intelligence artificielle et recherche d'outils : Problèmes de flots.*

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein (2001), *Introduction to algorithms.*