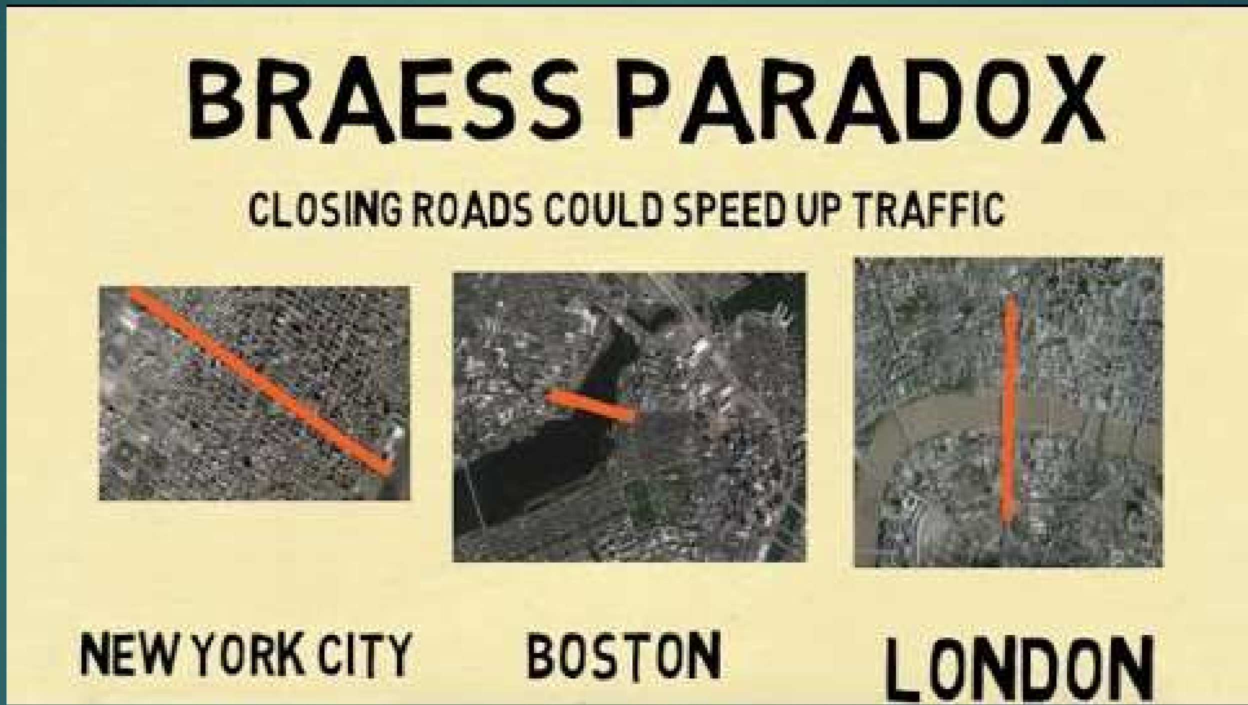
Modélisation du paradoxe de Braess sur flux routiers à l’aide de jeux de routage et de théorie des jeux

Table des matières

I) Introduction

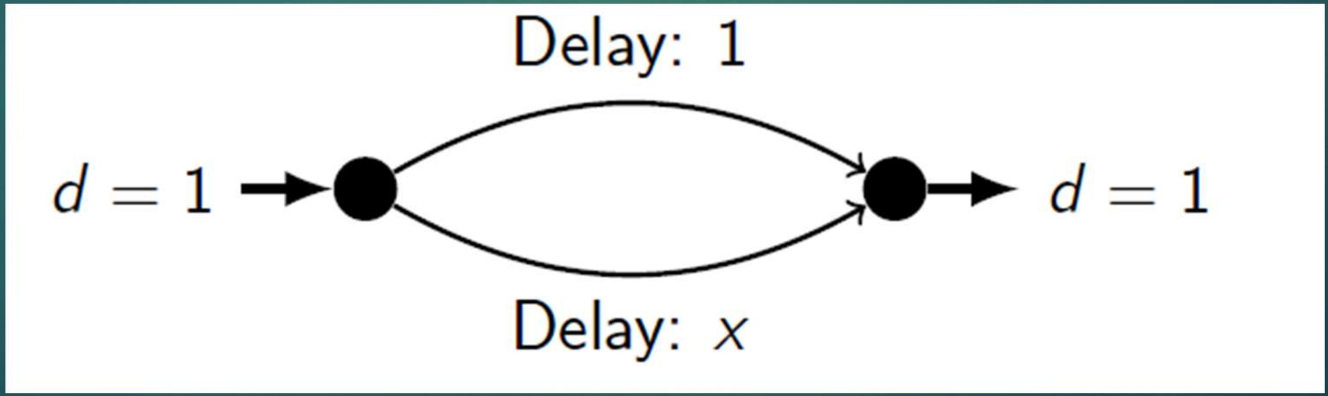


Source : https://youtu.be/8mlH9bnvWVE

II) Modélisation théorique

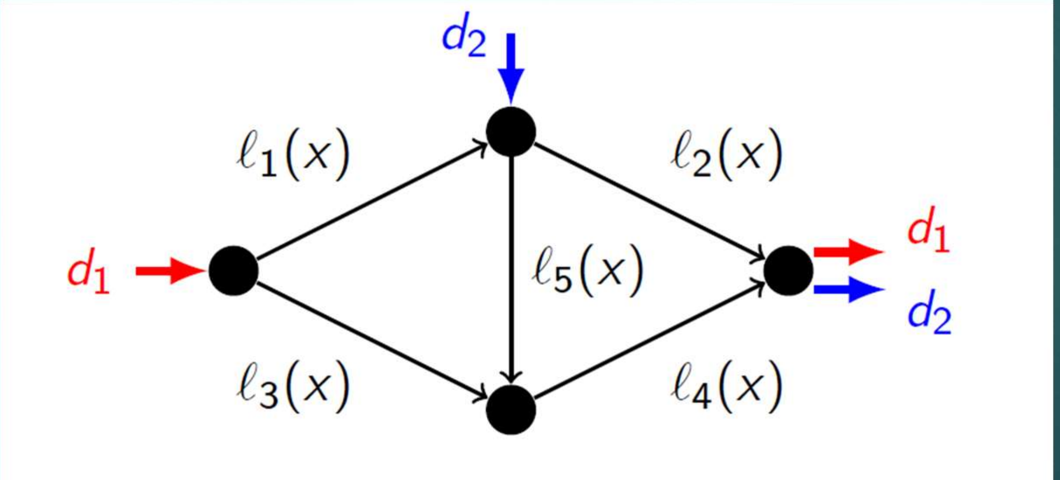
Notations

- *Graphe* : G = (N,A)
- *Paire origine-destination* : $K \subset N \times N,\ \forall\, k = (s_k, t_k) \in K,$
  d_k le débit
- *Fonction de latence* : coût de la traversée en fonction de la charge pour un arc donné
- *Instance* : $(\mathrm{G},\ (l_a)_{a \in A}, (d_k)_{k \in K})$



II) Modélisation théorique

- *Chemin* : P ⊂ A
- *Ensemble de chemins de* (s_k, t_k) : $\mathcal{P}_k$
- $\mathcal{P} = \bigcup_{k \in K} \mathcal{P}_k$



$$\mathcal{P} = \{\{1,2\};\{3,4\};\{1,5,4\};\{2\};\{5,4\}\}$$

II) Modélisation théorique

Définitions

- *Principe* : chemins moins coûteux choisis

- *Contraintes* : $\begin{cases} \sum_{P \in \mathcal{P}_k} x_p = d_k \\ x_p \geq 0 \end{cases}$

- *Equilibre de Wardrop (WE)* : flux réalisable x (donc qui satisfait les contraintes précédentes) avec $l_P(x) \leq l_Q(x)$, $\forall k \in K;\ P, Q\ \in \mathcal{P}_k;\ tel\ que\ x_P > 0$

➔ Les coûts de tous les chemins utilisés sont les mêmes

➔ le coût que paierait un utilisateur pour ajouter du trafic sur un chemin inutilisé serait au moins aussi cher que sur un chemin utilisé

II) Modélisation théorique

Théorèmes

- *Ensemble d'équilibres de Wardrop* : $\min_{x} \sum_{a \in A} \int_0^{x_a} l_a(y)\, dy$

  Sous les contraintes : $\begin{cases} \sum_{P \in \mathcal{P}_k} x_p = d_k \\ x_p \geq 0 \end{cases}$

- Pour une instance avec des fonctions de latence continues et croissantes, un équilibre de Wardrop existe et est essentiellement unique

II) Modélisation théorique

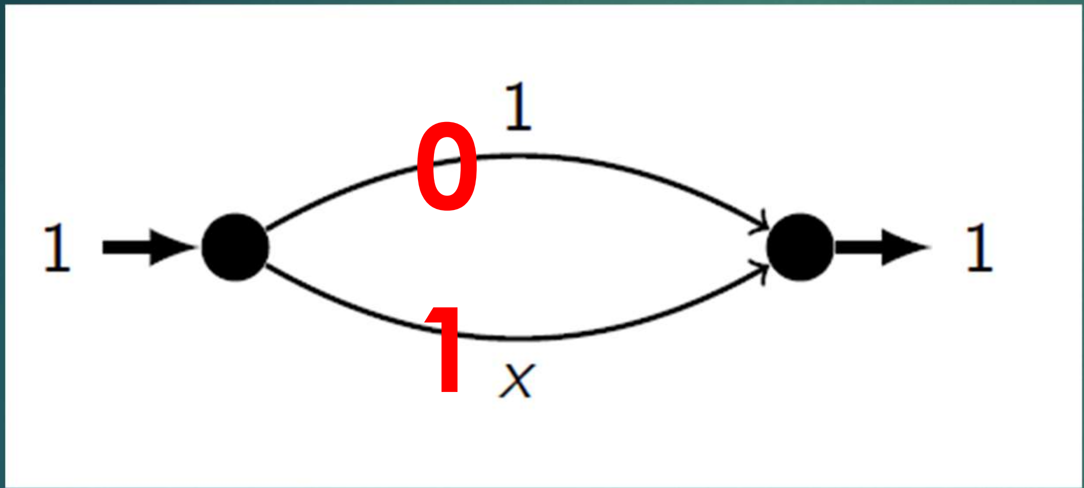
Définitions

- *Coût total* : pour un flux réalisable x, $C(x) = \begin{cases} \sum_{a \in A} x_a l_a(x_a) \\ \sum_{p \in \mathcal{P}} x_p l_p(x) \end{cases}$

- *Optimum social (SO)* : un flux réalisable qui minimise le coût total : $x^{SO} \in \arg\min_x C(x)$, sous les contraintes usuelles

II) Modélisation théorique

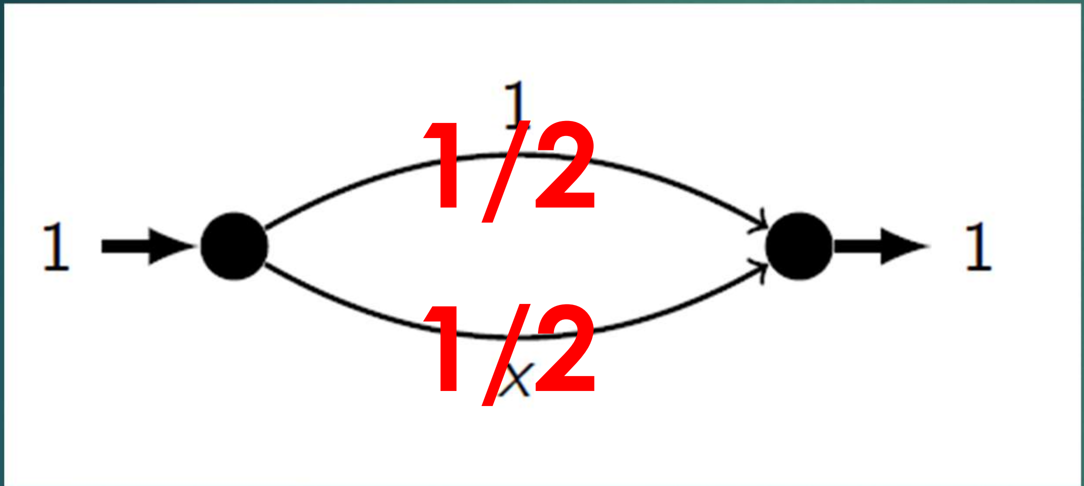
Différence entre optimum social et équilibre de Wardrop



Equilibre de Wardrop:
$C(x) = 1 \times 0 + 1 \times 1 = 1$
➔ **Situation réelle**

II) Modélisation théorique

Différence entre optimum social et équilibre de Wardrop



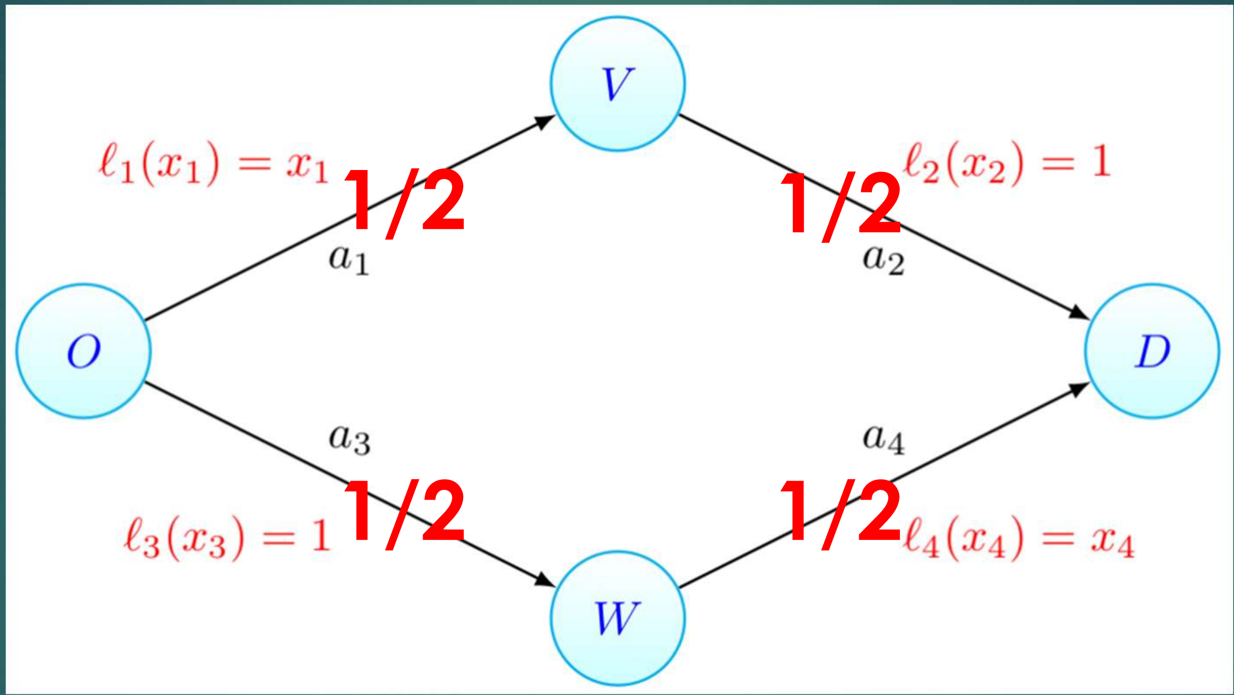
Optimum social:

$C(x) = 1 \times \frac{1}{2} + 1 \times \frac{1}{2} = \frac{3}{4}$

➔ **Situation idéale**

II) Modélisation théorique

Paradoxe de Braess – Situation initiale

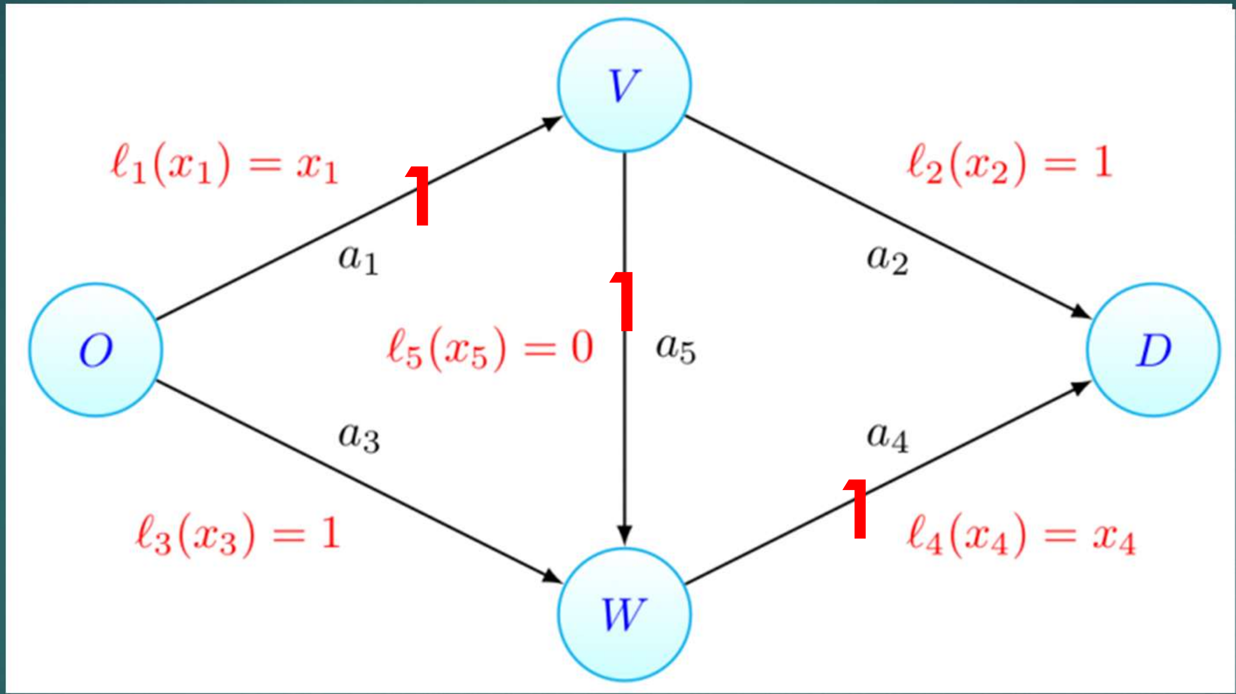


$$C^{WE} = \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times 1 + \frac{1}{2} \times 1 + \frac{1}{2} \times \frac{1}{2} = \frac{3}{2}$$

https://www.networkpages.nl/traffic-congestion-iv-braess-paradox/

II) Modélisation théorique

Paradoxe de Braess – ajout d'une route : le coût total augmente



$$C^{WE} = 1 + 0 + 1 = 2$$

https://www.networkpages.nl/traffic-congestion-iv-braess-paradox/

II) Modélisation théorique

Définitions

- *Prix de l'anarchie (POA) :* $POA = \max_{instances} \frac{C^{WE}}{C^{SO}}$
- *Réseau à fonctions de latence affines :* $POA \leq \frac{4}{3}$
- *Réseau à fonctions de latence polynomiales de degré p :* $POA = \ \Omega(\frac{p}{\ln})$

III) Implémentation

- Définition des structures de données nécessaires pour représenter le réseau de transport, y compris les listes des arêtes suivantes et les paires origine-destination
- Définition de la fonction de latence qui calcule la valeur associée au coût d'une arête pour une demande donnée
- Implémentation des fonctions pour calculer les chemins faisables pour chaque paire origine-destination, ainsi que les charges des arêtes en fonction des charges des chemins (formules des diapositives 4 à 8)
- Définition d'une fonction pour calculer le coût d'un chemin

III) Implémentation

- Implémentation de l'algorithme de Wardrop
- Fonctions pour calculer l'optimum social, le coût total, le prix de l'anarchie (formules des diapositives 4 à 8 et 13)
- Usage des bibliothèques NetworkX et Matplotlib pour visualiser les charges des arêtes dans le réseau de transport, à la fois pour la distribution optimale et pour l'équilibre de Wardrop

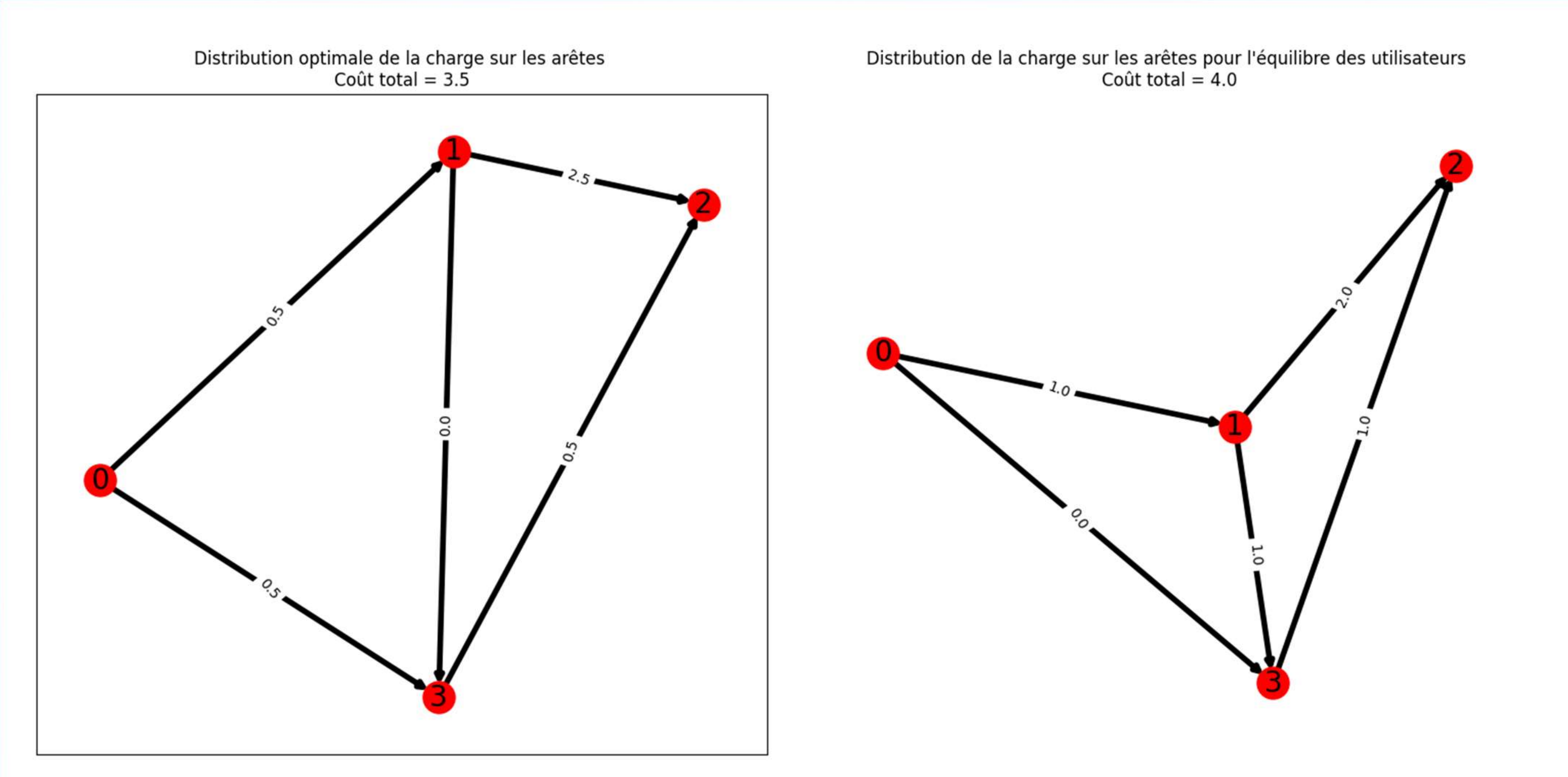
III) Implémentation

Approximation de l'équilibre de Wardrop

- Itération jusqu'à ce qu'une convergence soit atteinte, en ajustant les débits de chemin des utilisateurs
- Compare les coûts des chemins disponibles pour chaque paire origine-destination et permet aux utilisateurs de basculer vers un chemin moins coûteux s'il est disponible et s'ils ont suffisamment de débit
- Les débits des chemins sont mis à jour itérativement jusqu'à ce que les écarts de coûts entre les chemins utilisés et les moins coûteux soient suffisamment petits
- Affichage des débits des chemins et des charges des arêtes à la fin

III) Implémentation

Accord modèle théorique et implémentation python



III) Implémentation

Accord modèle théorique et implémentation python

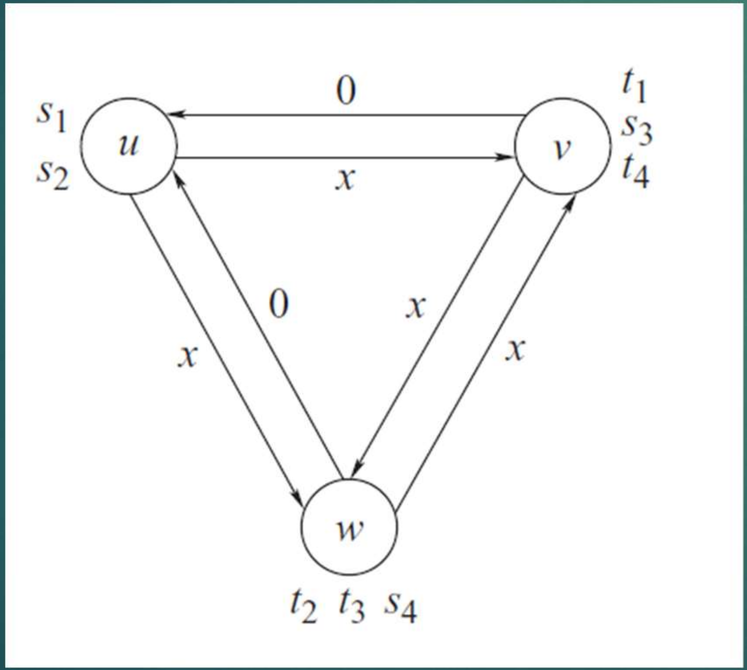
- *A gauche* – optimum social :
  - Pour $d_1 = 1$: ½ sur $x_{\{1,2\}}$ et ½ sur $x_{\{3,2\}}$
  - ➔ $l_0(x) = \frac{1}{4}$; $l_1(x) = \frac{1}{2}$; $l_2(x) = \frac{1}{2}$; $l_3(x) = \frac{1}{4}$
  - Pour $d_2 = 2$: 2 sur $x_{\{2\}}$
  - ➔ $l_1(x) = 2$
  - $C^{SO} = 3{,}5$

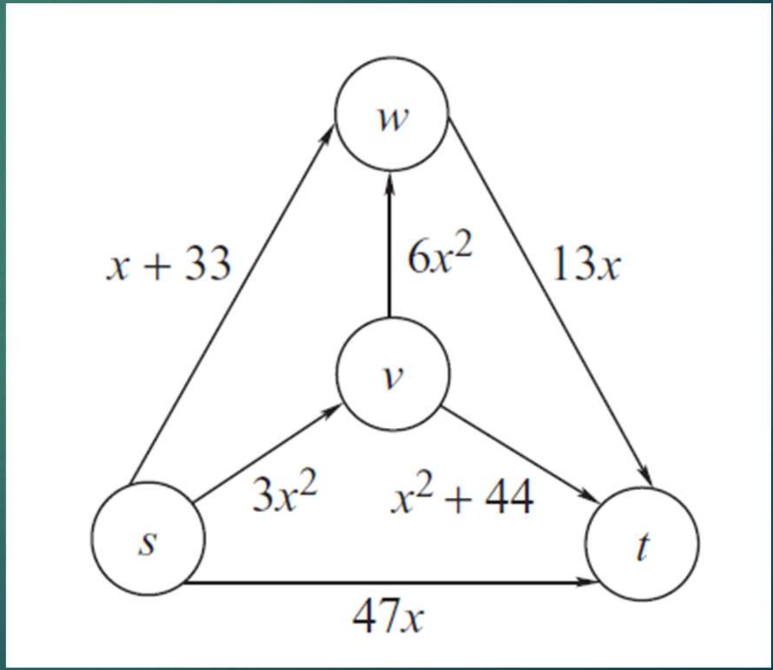
- *A droite* – équilibre de Wardrop :
  - Pour $d_1 = 1$: 1 sur $x_{\{1,3,2\}}$
  - ➔ $l_0(x) = 1$; $l_1(x) = 0$; $l_2(x) = 0$; $l_3(x) = 1$
  - Pour $d_2 = 2$: 2 sur $x_{\{2\}}$
  - ➔ $l_1(x) = 2$
  - $C^{WE} = 4$

IV) Limites du modèle

Limites du modèle

- Fonctions de latence affines
- Instance de Awerbuch-Azar-Epstein – défaut de l'algorithme d'approximation
- Instance de Roughgarden – sans équilibre



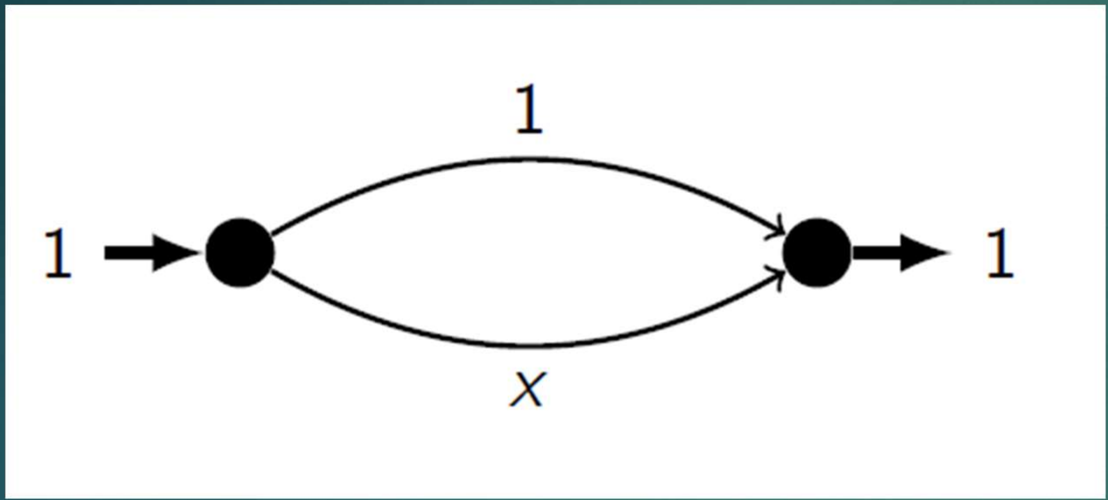


V) Solution envisageable

Taxe pigouvienne

- *Taxe pigouvienne :* $t_a = x_a{}^{SO} l_a{}'(x_a{}^{SO})$
  ➔ Création d'une restriction
- *Si :* $\forall\, a \in A,\ x \mapsto x l_a(x)$ *est convexe*, alors l'ensemble des optimums sociaux est un ensemble d'équilibre de Wardrop pour une instance avec les fonctions de latence :
  $$\overline{l_a}(x) = l_a(x) + x l_a{}'(x)$$
- Prise en compte taxe pigouvienne dans la fonction qui calcule le coût d'un chemin
- Affichage graphique de la variation de cette taxe

V) Solution envisageable



Optimum social :

$$C(x) = 1 \times \frac{1}{2} + \frac{1}{2} \times \frac{1}{2} = \frac{3}{4}$$

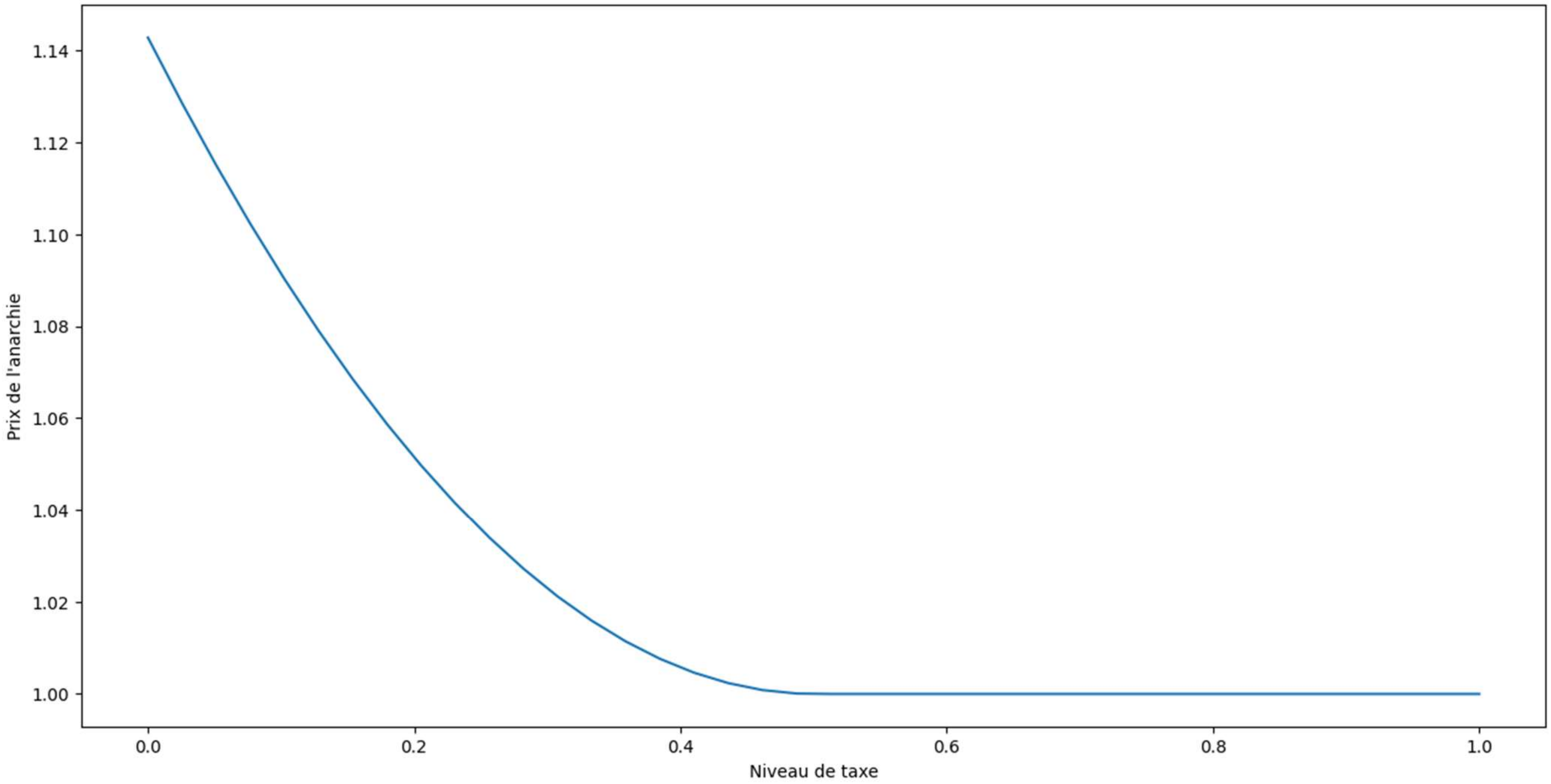
Taxes :

$$t_1 = 0$$
$$t_2 = x$$

Nouvelle instance avec $\overline{l_0}(x) = 1$ et $\overline{l_1}(x) = 2x$

Cout de l'équilibre de Wardrop : $C^{WE} = 1 \times \frac{1}{2} + 2 \times \frac{1}{2} \times \frac{1}{2} = 1$

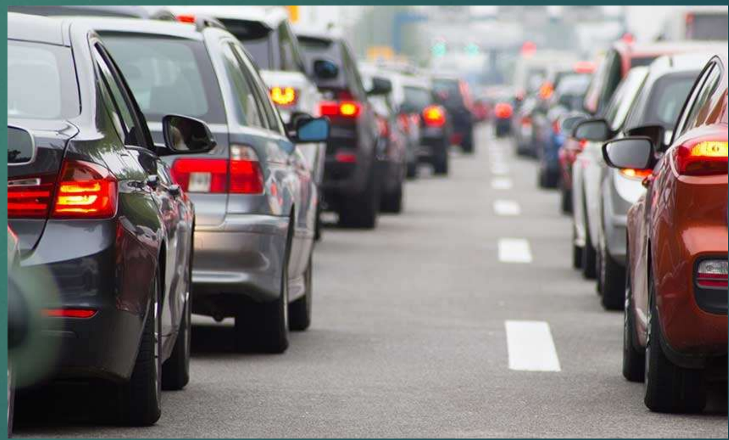
V) Solution envisageable



VI) Conclusion

Applications

- ❑ Cas de congestion – fluidification du trafic
- ❑ Travaux routiers – optimisation du réseau existant
- ❑ Renforcer l’usage des transports en commun



https://agirpourlatransition.ademe.fr/particuliers/bureau/deplacements/modifier-trafic-routier-necessite-ameliorer-qualite-lair

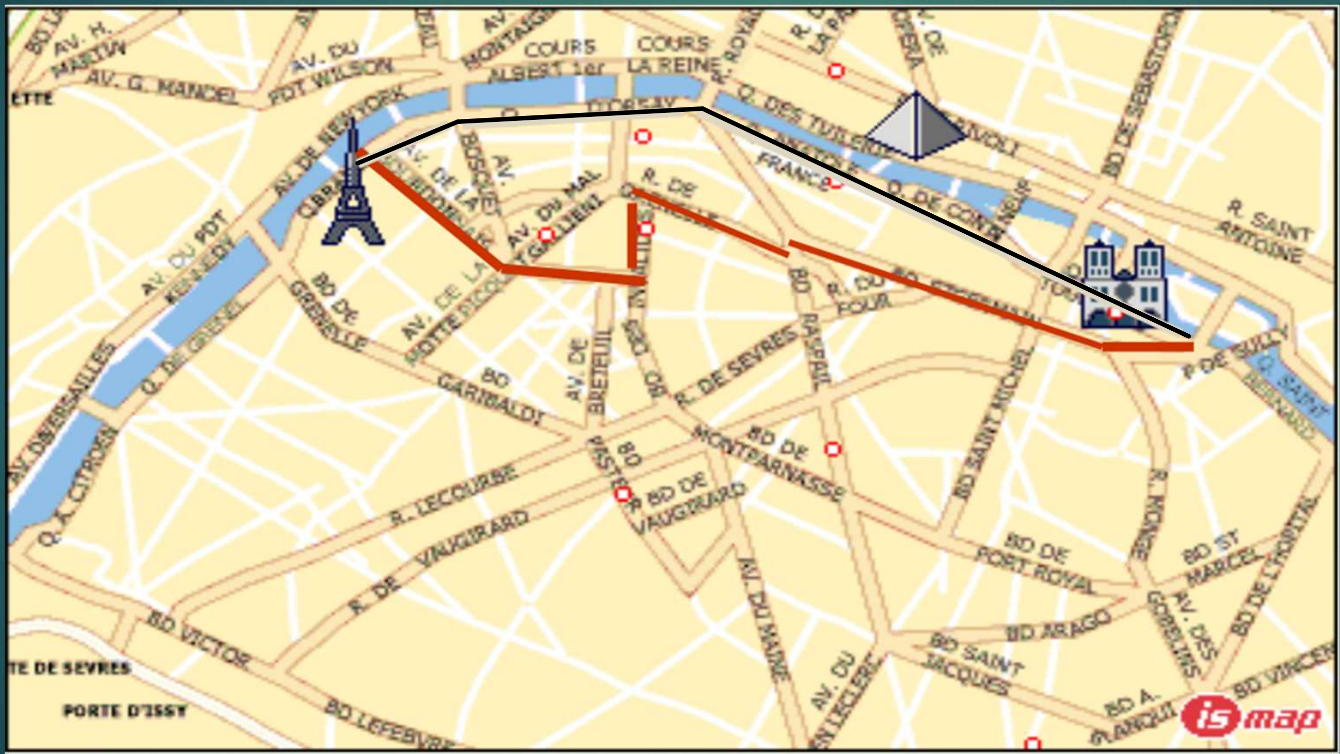
https://www.francebleu.fr/infos/transports/la-cts-renforce-son-protocole-sanitaire-a-partir-de-lundi-1637934418

N° 18335

VI) Conclusion

Ouverture

- ❑ Déploiement de la modélisation à l'échelle d'une ville
- ❑ Meilleure approximation de l'équilibre de Wardrop



https://educnet.enpc.fr/pluginfile.php/15132/mod_resource/content/0/04-b-wardrop.pdf

Merci beaucoup pour votre attention !

VII) Annexes

Principale différence entre équilibre de Wardrop et de Nash

- *L'équilibre de Nash* se concentre sur l'optimisation individuelle et locale des joueurs, obtenu en se basant uniquement sur les choix de chaque utilisateur

- *L'équilibre de Wardrop* vise à atteindre un équilibre global et équitable dans un réseau de transport, obtenu en cherchant à minimiser le temps de trajet global

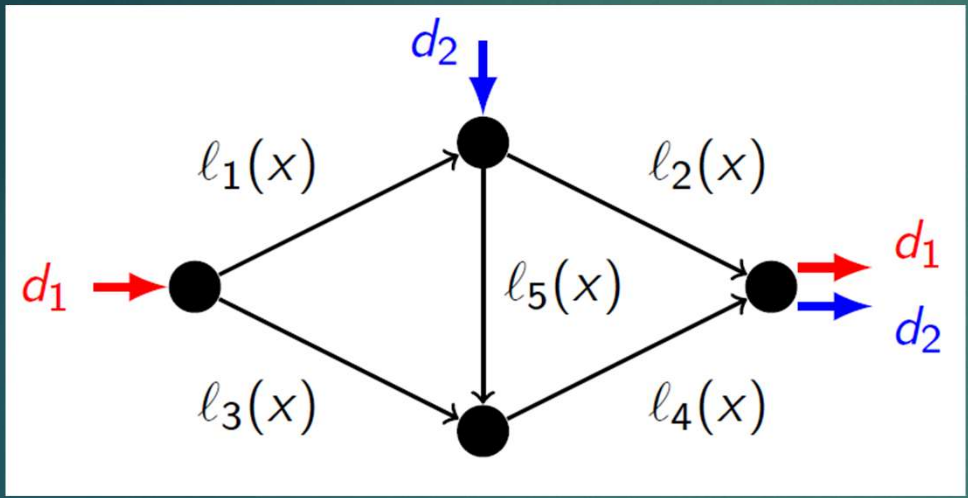
VII) Annexes

Définitions

- *Flux réalisable* : vecteur $(x_p)_{p\in\mathcal{P}}$ en accord avec la demande
  $\begin{cases} \sum_{P\in\mathcal{P}_k} x_p = d_k \\ x_p \geq 0 \end{cases}$
- *Flux par arc* : $x_a = \sum_{P:a\in\mathcal{P}} x_P$
- *Coût d'un chemin* : $l_p(x) = \sum_{a\in P} l_a(x_a)$
- *Principe* : chemins moins coûteux choisis
- *Equilibre de Wardrop (WE)* : flux réalisable x avec $l_P(x) \leq l_Q(x)$, pour $\forall k \in K;\ P, Q\ \in \mathcal{P}_k;\ tel\ que\ x_P > 0$

VII) Annexes

Exemple – Calcul d'un équilibre de Wardrop



1) Pas un équilibre : $l_{total} = 3{,}8$

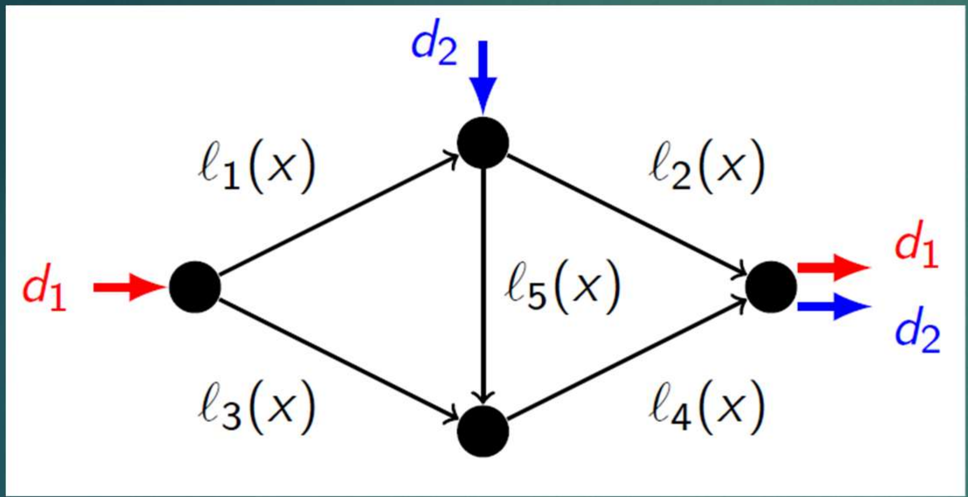
$$\begin{cases} x_{\{1,2\}} = 0{,}5\ ;\ x_{\{3,4\}} = 0{,}2\ ;\ x_{\{1,5,4\}} = 0{,}3 \\ x_{\{2\}} = 1{,}5\ ;\ x_{\{5,4\}} = 0{,}5 \end{cases}$$

2) Est un équilibre : $l_{total} = 3{,}4$

$$\begin{cases} x_{\{1,2\}} = 0{,}6\ ;\ x_{\{3,4\}} = 0\ ;\ x_{\{1,5,4\}} = 0{,}4 \\ x_{\{2\}} = 1{,}4\ ;\ x_{\{5,4\}} = 0{,}6 \end{cases}$$

VII) Annexes

Exemple – Calcul d'un équilibre de Wardrop



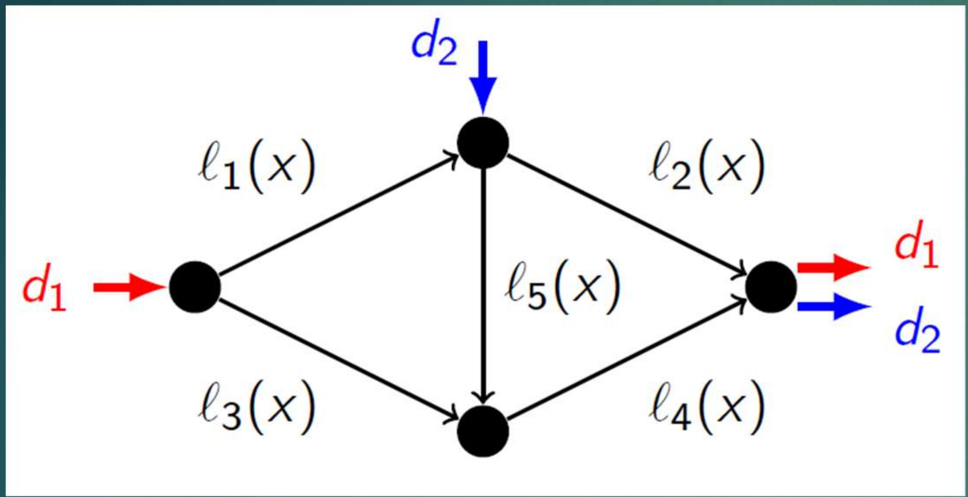
1) Pas un équilibre : $l_{total} = 3{,}8$

$$\begin{cases} x_{\{1,2\}} = 0{,}5 \; ; \; x_{\{3,4\}} = 0{,}2 \; ; \; x_{\{1,5,4\}} = 0{,}3 \\ x_{\{2\}} = 1{,}5 \; ; \; x_{\{5,4\}} = 0{,}5 \end{cases}$$

$$l_1 = 0{,}8$$
$$l_2 = 1$$
$$l_3 = 1$$
$$l_4 = 1$$
$$l_5 = 0$$

VII) Annexes

Exemple – Calcul d'un équilibre de Wardrop



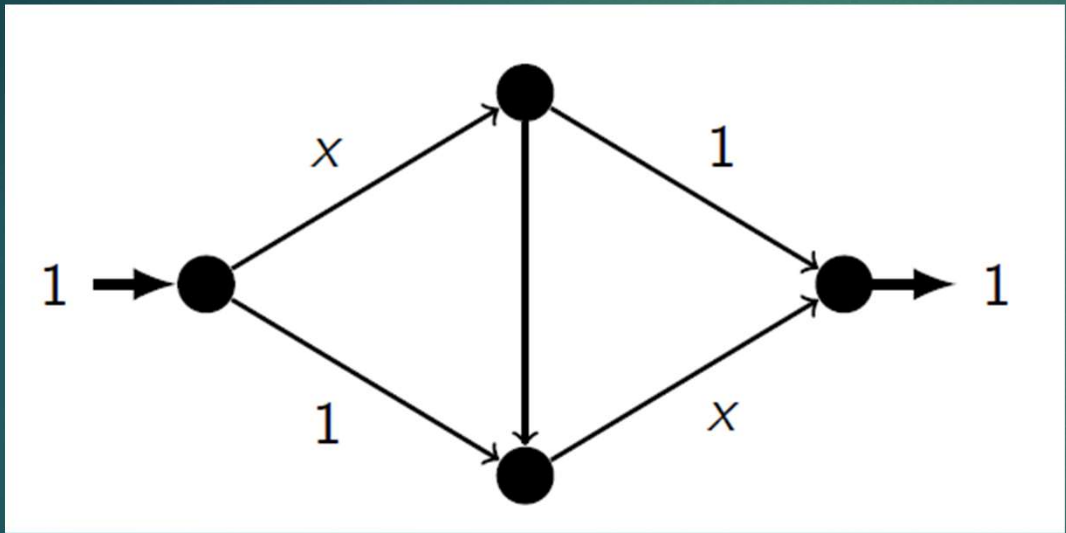
2) Est un équilibre : $l_{total} = 3{,}4$

$$\begin{cases} x_{\{1,2\}} = 0{,}6\ ;\ x_{\{3,4\}} = 0\ ;\ x_{\{1,5,4\}} = 0{,}4 \\ x_{\{2\}} = 1{,}4\ ;\ x_{\{5,4\}} = 0{,}6 \end{cases}$$

$$l_1 = 1$$
$$l_2 = 1$$
$$l_3 = 0{,}4$$
$$l_4 = 1$$
$$l_5 = 0$$

VII) Annexes

Exemple – Calcul de taxes pigouviennes sur l'instance de Braess



Optimum social :

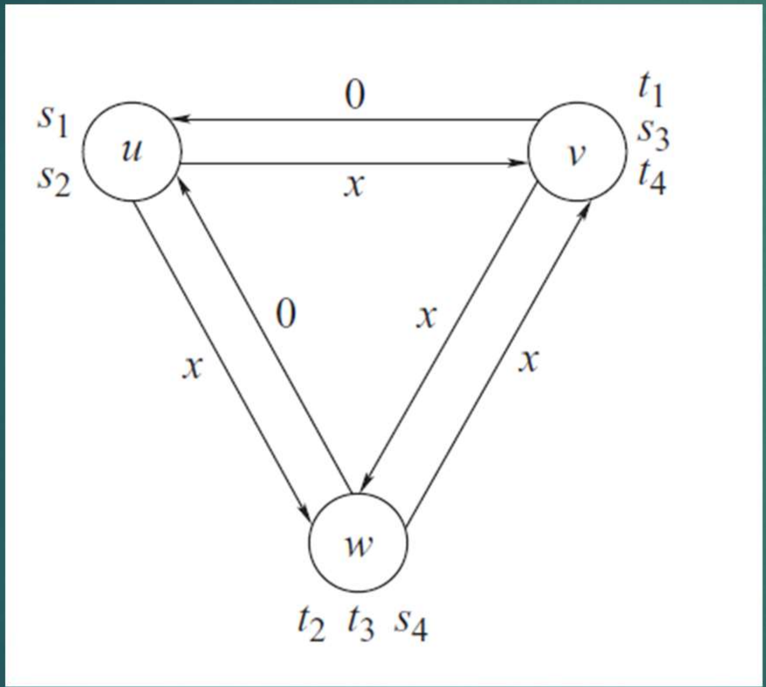
$$C(x) = 1 \times 1 + 1 \times 0 + 1 \times 1 = 2$$

Taxes :

$$t_1 = x$$
$$t_2 = 0$$
$$t_3 = 0$$
$$t_4 = x$$
$$t_5 = 0$$

VII) Annexes

Instance de Awerbuch-Azar-Epstein – défaut dans l'algorithme d'approximation



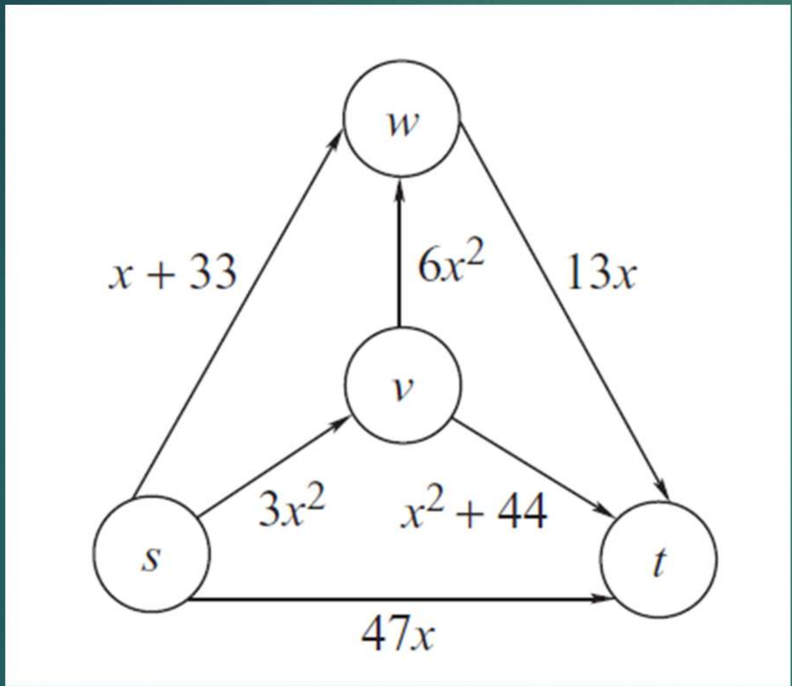
Algorithme python :
POA = 12,5

Alors qu'en réalité :
POA = 2,5

➔ approximations trop grandes dans Wardrop() et chemins_faisables()

VII) Annexes

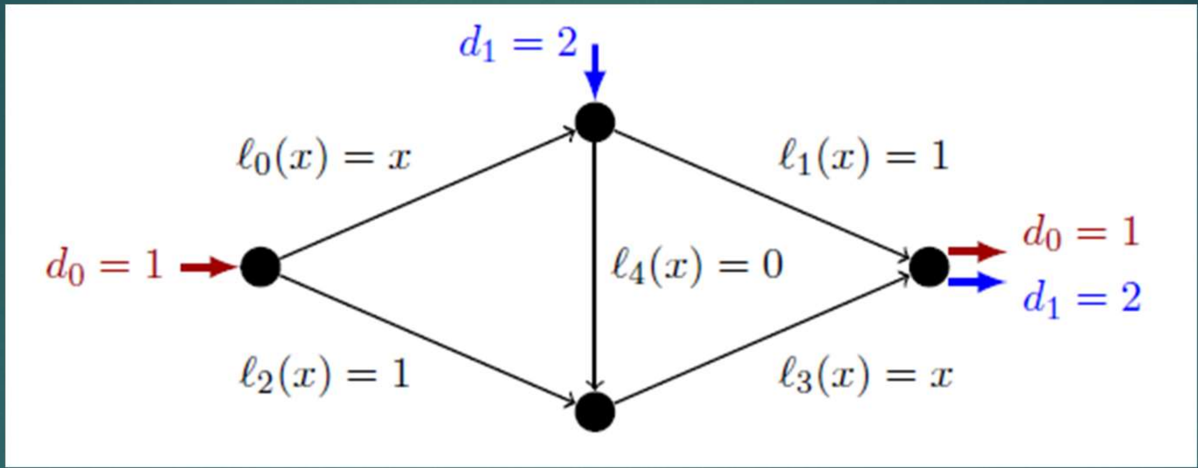
Instance de Roughgarden – sans équilibre



P1 = s → t
P2 = s → v → t
P3 = s → w → t
P4 = s → v → w → t
J1 = joueur 1, J2 = joueur 2

- J2 sur P1 ou P2 → J1 prend P4
- J2 sur P3 ou P4 → J1 prend P1
- J1 sur P4 → J2 prend P3
- J1 sur P1 → J1 prend P2

VII) Annexes



- *aretes_suivantes* : liste de listes, avec aretes_suivantes[i] contenant l'arête j si l'arête i arrive au départ de l'arête j

- *paires_od* : liste de dictionnaires qui pour une paire k contient :
  - « demande » : un entier correspondant à la demande
  - « origine » : une liste d'arêtes de même départ que k
  - « destination » : une liste d'arêtes de même destination que k

VII) Annexes

- *latence(a,x)* : renvoie la latence associée à l'arête a pour la demande x

- *chemins_faisables(k)* : renvoie tous les chemins faisables pour chaque paire k en ayant distribué la demande. On recherche une suite d'arêtes $(a_1, a_2, \dots, a_n)$ telle que :
  - a_1 soit dans la liste « origine »
  - a_n soit dans la liste « destination »
  - les liens soient connectés :

  $\forall\, i \in [\![1, n-1]\!], \qquad a_i \in aretes\_suivantes[a_i]$

VII) Annexes

- *chemins* : liste contenant tous les chemins pour toutes les paires OD

- *calcul_charge_aretes(charges_chemins)* : calcul du flux par arc en fonction du chargement des chemins de chemins

- *cout_chemin(chemin, charges_aretes, taxes_aretes)* : calcul du coût d'un chemin en fonction de son chargement et d'éventuelles taxes

- *Wardrop(taxes_aretes)* : approximation de l'équilibre de Wardrop

VII) Annexes

- *optimum_social()* : renvoie les flux associés aux chemins selon l'optimum social
- *cout(charges_chemins)* : renvoie sous forme de liste le coût par arc
- *cout_total(charges_chemins)* : somme totale des coûts des différentes arêtes
- *POA(taxes_aretes)* : calcul du prix de l'anarchie à l'aide du coût total de optimum_social() et du coût total de l'équilibre de Wardrop()

VII) Annexes

- *taxes_optimales()* : renvoie la liste des t_a
- *variation_taxe()*: trace le graphique du niveau de taxes en fonction du prix de l'anarchie

VII) Annexes


```python
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt


latences_changees = False

# Instance de Braess
aretes_suivantes = [[1,4], [], [3], [], [3]]
paires_od = [{"demande":1,"origine":[0,2],"destination":[1,3]},
             {"demande":2,"origine":[1,4],"destination":[1,3]}]

# Instance de Pigou
# aretes_suivantes = [[],[]]
# paires_od = [{"demande":1,"origine":[0,1],"destination":[0,1]}]
```

VII) Annexes


```
def latence(a,x):
    """Renvoie la valeur associée à la fonction de latence de
l'arête a pour une demande x,
    -1 si l'arête n'existe pas"""
    if a==0 or a==3:
        return x
    elif a==1 or a==2:
        return 1
    elif a==4:
        return 0
    else:
        return -1
```

VII) Annexes


```
# Instance de Awerbuch-Azar-Epstein
# POA de 25/2 contre 5/2, trop grosses approximations dans
Wardrop()
# aretes_suivantes = [[1,5], [2,4], [0,3], [2,4], [1,5], [0,3]]
# paires_od = [{"demande":1,"origine":[2,4],"destination":[2,5]},
#       {"demande":2,"origine":[1,5],"destination":[2,5]},
#       {"demande":3,"origine":[1,5],"destination":[1,3]},
#       {"demande":4,"origine":[2,4],"destination":[0,4]}]

# def latence(a,x):
#     """Renvoie la valeur associée à la fonction de latence de
l'arête a pour une demande x,
#       -1 si l'arête n'existe pas"""
#     if a==0 or a==1 or a==3 or a==4:
#         return x
#     elif a==2 or a==5:
#         return 0
#     else:
#         return -1
```

VII) Annexes


```
# # Instance de Roughgarden sans situation d'équilibre
# # Wardrop() boucle indéfiniment
# aretes_suivantes = [[], [2,3], [], [5], [5], []]
# paires_od =
[{"demande":1,"origine":[0,1,4],"destination":[0,2,5]},
#       {"demande":2,"origine":[0,1,4],"destination":[0,2,5]}]
```

VII) Annexes


```
# def latence(a,x):
#     """Renvoie la valeur associée à la fonction de latence de
l'arête a pour une demande x,
#       -1 si l'arête n'existe pas"""
#     if a==0:
#         return 47*x
#     elif a==1:
#         return 3*x*x
#     elif a==2:
#         return x*x+44
#     elif a==3:
#         return 6*x*x
#     elif a==4:
#         return x+33
#     elif a==5:
#         return 13*x
#     else:
        # return -1
```

VII) Annexes


```
# Ne fonctionne pas pour l'instance AAE car nous autorisons
seulement 2 stratégies
# 1 arête directe ou 2 arêtes, mais pas de cycle
def chemins_faisables(k):
    """Renvoie tous les chemins faisables pour chaque paire,
ceux-ci étant listés par leur indice, à prendre dans cet ordre"""
    origine,destination =
paires_od[k]["origine"],paires_od[k]["destination"]

    candidats = [[i] for i in
origine]

    # tous les chemins de la longueur considérée, démarrant dans
"origine"
    chemins_trouves = [[i] for i in origine if i in
destination]
    # tous les chemins faisables, commençant dans "origine" et
finissant dans "destination"
```

VII) Annexes


```
    chemins_trouves = [[i] for i in origine if i in
destination]
    # tous les chemins faisables, commençant dans "origine" et
finissant dans "destination"
    while candidats != []:
        candidats = [chemin+[j] for chemin in candidats for j in
aretes_suivantes[chemin[-1]] if not j in chemin]
        # tous les chemins candidats pour la taille de
aretes_suivantes
        chemins_trouves = chemins_trouves + [chemin for chemin in
candidats if chemin[-1] in destination]
        # ajout des "chemins_trouves" terminant dans
"destination"
    return chemins_trouves
```

VII) Annexes


```
chemins = [chemins_faisables(i) for i in range(len(paires_od))]
# Pour AAE :
# chemins = [[[2], [4,5]], [[5], [1,2]], [[1], [5,3]], [[4],
[2,0]]]


def calcul_charge_aretes(charges_chemins):
    """ Calcule la charge des liens en se basant sur la charge des
chemins,
    ceux-ci étant listés dans la liste chemins"""
    charge_aretes = [0]*len(aretes_suivantes)
    for k in range(len(paires_od)):
        for index_chemin in range(len(chemins[k])):
            for a in chemins[k][index_chemin]:
                charge_aretes[a] +=
   charges_chemins[k][index_chemin]
    return charge_aretes
```

VII) Annexes


```
def cout_chemin(chemin, charge_aretes, taxes_aretes):
    """Calcule le coût d'un chemin, en prenant en compte les
taxes pigouviennes ou non"""
    if latences_changees:
        epsilon = 0.000001
        return sum([latence(a,charge_aretes[a]) +
charge_aretes[a] *(latence(a,charge_aretes[a]+epsilon)-
latence(a,charge_aretes[a])) / epsilon
        for a in chemin])
    else:
         return
sum([latence(a,charge_aretes[a])+taxes_aretes[a] for a in
chemin])
```

VII) Annexes


```
def Wardrop(taxes_aretes):
    """Approximation de l'équilibre de Wardrop,
    renvoie charges_chemins, charge_aretes, et couts_chemins"""
    pas,delta_arret = 0.1, 0.000001
    # pas à chaque itération et précision pour l'arrêt
    charges_chemins = []
    # Initialisation des flux de chemins
    for k in range(len(paires_od)):
        taille_chemin = len(chemins[k])
        # nombre de chemins pour la paire de paires_od
actuellement séléctionnée
        charges_chemins =
charges_chemins+[[paires_od[k]["demande"]/taille_chemin]
*taille_chemin]
        # répartition égale de la demande parmi ces chemins
    delta_cout =[1]*len(paires_od)
    # Ecart entre les chemins utilisés les plus coûteux et les
moins coûteux pour chaque paire
```

VII) Annexes


```
# Phase de convergence
    while max(delta_cout) >
delta_arret:
        # Tant que l'on est trop loin de l'équilibre
        for k in range(len(paires_od)):
            # Chaque paire peut contenir des usagers qui doivent changer
de chemins
            charge_aretes = calcul_charge_aretes(charges_chemins)
            pcosts = [cout_chemin(chemin,charge_aretes,taxes_aretes) for
chemin in chemins[k]]
            # Coût actuel du chemin subi pour cette paire - celui le
moins coûteux
            index_chemin_moins_couteux =  pcosts.index(min(pcosts))
            for i in range(len(pcosts)):
                debit_a_transferer = min(pas*(pcosts[i]-min(pcosts)),
charges_chemins[k][i])
                charges_chemins[k][i] -= debit_a_transferer
                charges_chemins[k][index_chemin_moins_couteux] +=
debit_a_transferer
```

VII) Annexes


```
# Mise à jour des variables, en particulier de delta_cout
        charge_aretes = calcul_charge_aretes(charges_chemins)
        couts_chemins = []
        for k in range(len(paires_od)):
            couts_chemins.append([cout_chemin(chemin,charge_arete
s,taxes_aretes) for chemin in chemins[k]])
            couts_chemins_utilises_disponibles =
[couts_chemins[k][i]*(charges_chemins[k][i]>0) for i in
range(len(couts_chemins[k]))]
            # Coût parmi les chemins utilisés, 0 sinon
            delta_cout[k] =
max(couts_chemins_utilises_disponibles) - min(couts_chemins[k])
```

VII) Annexes


```
  # Affichage des résultats
    print("Charges des chemins = ",[[round(charges_chemins[k][a],3)
for a in range(len(charges_chemins[k]))] for k in
range(len(paires_od))])
    print("Coûts des chemins = ",[[round(couts_chemins[k][a],3) for
a in range(len(charges_chemins[k]))] for k in
range(len(paires_od))])
    charge_aretes = calcul_charge_aretes(charges_chemins)
    print("Charges des arêtes correspondantes = ", [round(x_A,3)
for x_A in calcul_charge_aretes(charges_chemins)], '\n')
    return charges_chemins
```

VII) Annexes


```
def optimum_social():
    global latences_changees
    latences_changees = True
    charges_chemins = Wardrop([0]*len(aretes_suivantes))
    latences_changees = False
    return charges_chemins

def cout(charges_chemins):
    charge_aretes = calcul_charge_aretes(charges_chemins)
    return [charge_aretes[a]*latence(a,charge_aretes[a]) for a in 
range(len(charge_aretes))]


def cout_total(charges_chemins):
    return sum(cout(charges_chemins))
```

VII) Annexes


```python
def POA(taxes_aretes):
    optimal = cout_total(optimum_social())
    equilibre = cout_total(Wardrop(taxes_aretes))
    return (equilibre, optimal, equilibre/optimal)


def variation_taxe():
    taxes = np.linspace(0,1,40)
    PoAs = []
    for tax in taxes:
        _,_,PoA = POA([0,0,0,0,tax])
        PoAs.append(PoA)
    plt.plot(taxes,PoAs)
    plt.xlabel("Niveau de taxe")
    plt.ylabel("Prix de l'anarchie")
    plt.show()
```

VII) Annexes


```
def taxes_optimales():
    epsilon = 0.000001
    # pour dériver
    x = calcul_charge_aretes(optimum_social())
    return [x[a] * (latence(a,x[a]+epsilon) - latence(a,x[a])) / 
epsilon
            for a in range(len(x))]


print(f"Chemins = {chemins} \n")
taxes_aretes = [0]*len(aretes_suivantes)
Wardrop(taxes_aretes)
variation_taxe()
```

VII) Annexes


```
# Affichage graphique

G = nx.DiGraph()

# Dessine charge_aretes avec la distribution de optimum_social()
subax1 = plt.subplot(121)
charges_chemins = optimum_social()
subax1.set_title(f"Distribution optimale de la charge sur les
arêtes \nCoût total = {round(cout_total(charges_chemins),1)}")

charges_chemins_brutes = calcul_charge_aretes(charges_chemins)
charge_aretes = [round(elt, 1) for elt in charges_chemins_brutes]

G.add_weighted_edges_from([(0,1,charge_aretes[0]),
(1,2,charge_aretes[1]), (0,3,charge_aretes[2]),
                          (3,2,charge_aretes[3]),
(1,3,charge_aretes[4])])
```

VII) Annexes

```
pos = nx.spring_layout(G, seed=7)
nx.draw_networkx_nodes(G, pos, node_size=500, node_color='r')
nx.draw_networkx_edges(G, pos, width=4)
nx.draw_networkx_labels(G, pos, font_size=20, font_family="sans-serif")
edge_labels = nx.get_edge_attributes(G, "weight")
nx.draw_networkx_edge_labels(G, pos, edge_labels)


# Dessine charge_aretes avec la distribution de Wardrop()
subax2 = plt.subplot(122)
charges_chemins = Wardrop([0]*len(aretes_suivantes))
subax2.set_title(f"Distribution de la charge sur les arêtes pour l'équilibre des utilisateurs \nCoût total = {round(cout_total(charges_chemins),1)}")
```

VII) Annexes


```
charges_chemins_brutes = calcul_charge_aretes(charges_chemins)
charge_aretes = [round(elt, 1) for elt in charges_chemins_brutes]
G.add_weighted_edges_from([(0,1,charge_aretes[0]),
(1,2,charge_aretes[1]), (0,3,charge_aretes[2]),
                                    (3,2,charge_aretes[3]),
(1,3,charge_aretes[4])])

pos = nx.spring_layout(G, seed=7)
nx.draw_networkx_nodes(G, pos, node_size=500, node_color='r')
nx.draw_networkx_edges(G, pos, width=4)
nx.draw_networkx_labels(G, pos, font_size=20, font_family="sans-serif")
edge_labels = nx.get_edge_attributes(G, "weight")
nx.draw_networkx_edge_labels(G, pos, edge_labels)

ax = plt.gca()
ax.margins(0.08)
plt.axis("off")
plt.tight_layout()
plt.show()
```