TIPE: Gestions des flux de spectateurs autour du Stade de France par la théorie des graphes.

Définition d'itinéraires sécurisés pour évacuer le Stade de France lors des Jeux Olympiques.

Noé VINCENT

Comment utiliser la théorie des graphes pour définir les itinéraires piétons aux alentours du stade de France ?

En cherchant à :

- maintenir la sécurité des spectateurs.
- le plus efficacement.

Plan

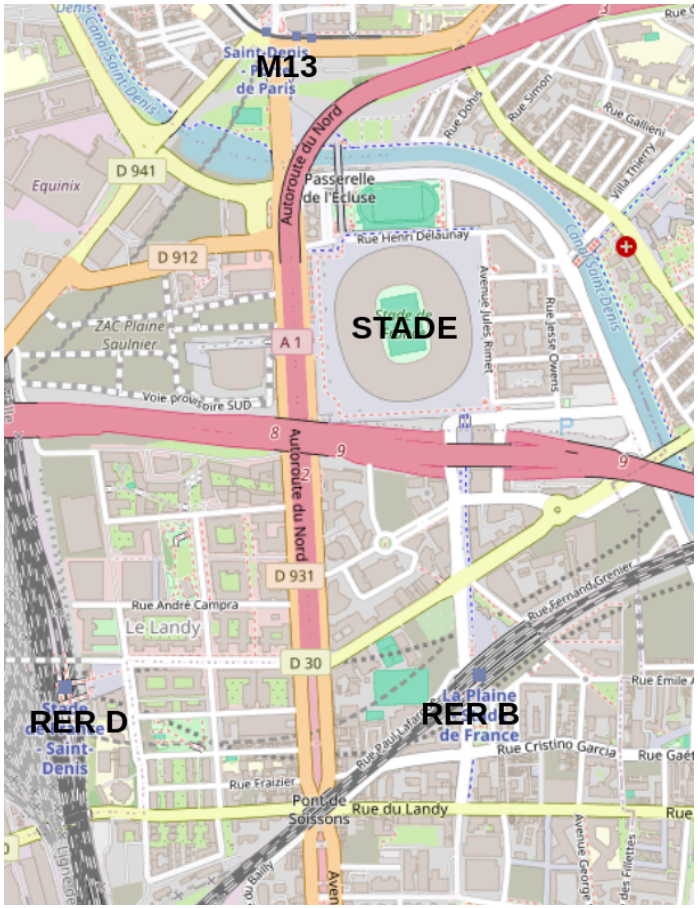
1. Analyse de la situation
2. Modélisation: Graphe de capacité
3. Une méthode naïve
4. Une méthode optimale: le flot maximal
5. Analyse des résultats
6. Annexe

1. Analyse de la situation

Situation géographique

Le stade de France: 81 500 spectateurs
3 stations de transport en commun aux alentours:

- Saint-Denis Porte de Paris (M13)
- La Plaine - Stade de France (RER B)
- Stade de France - Saint-Denis (RER D)

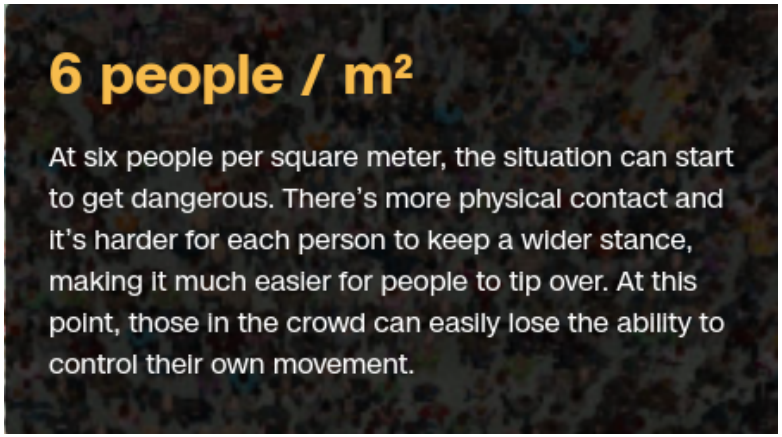


Carte des alentours du Stade de France - @OSM

Critère de sécurité:

À partir de 6 personnes/m², -> Potentiel danger
On place la limite à 5 personnes/m²

Helbing & Mukerji -> trop grande densité:
l'une des causes du drame de la Love Parade.



Issu de: These are the warning signs that a crowd is dangerously dense - @CNN

2. Modélisation: Graphe de capacité

Graphe de capacité

Soit $G_c = (V, E, C)$ un graphe non orienté pondéré par:

$C : E \to \mathbb{N}$ la capacité de chaque arête.

Le graphe des capacités.

Modélisation de la capacité

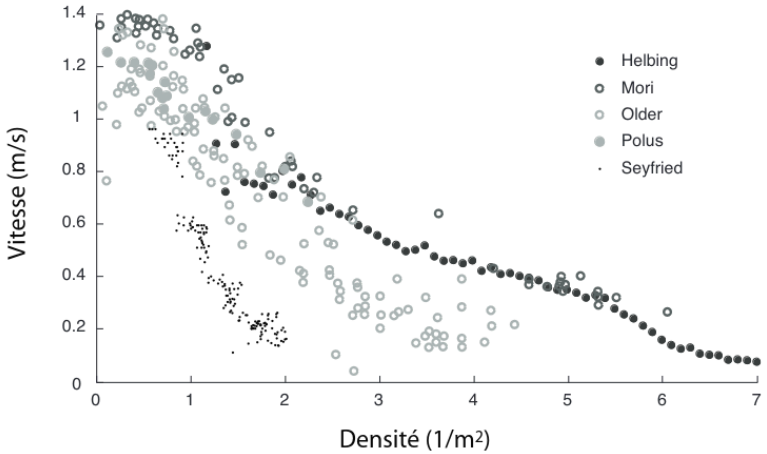
Modélisation de piétons dans le pire des cas :
$\delta = 5pers./m^2$

Capacité d'une rue: débit maximal en pers./s

$c = \delta * w * v$

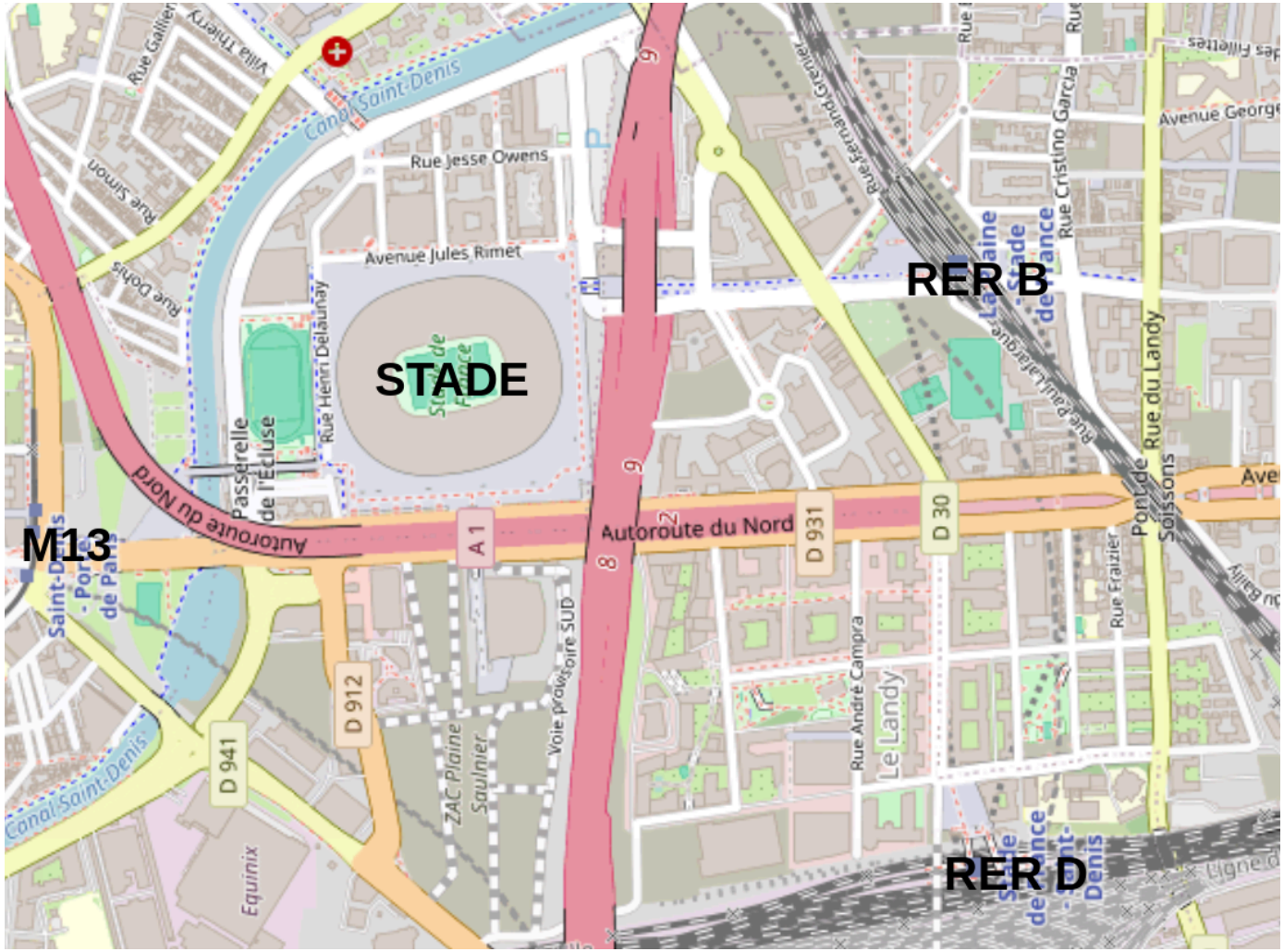
- w la largeur de la rue (approximée)
- v la vitesse de la foule
- δ la densité de la foule

$v = 0,4m/s$



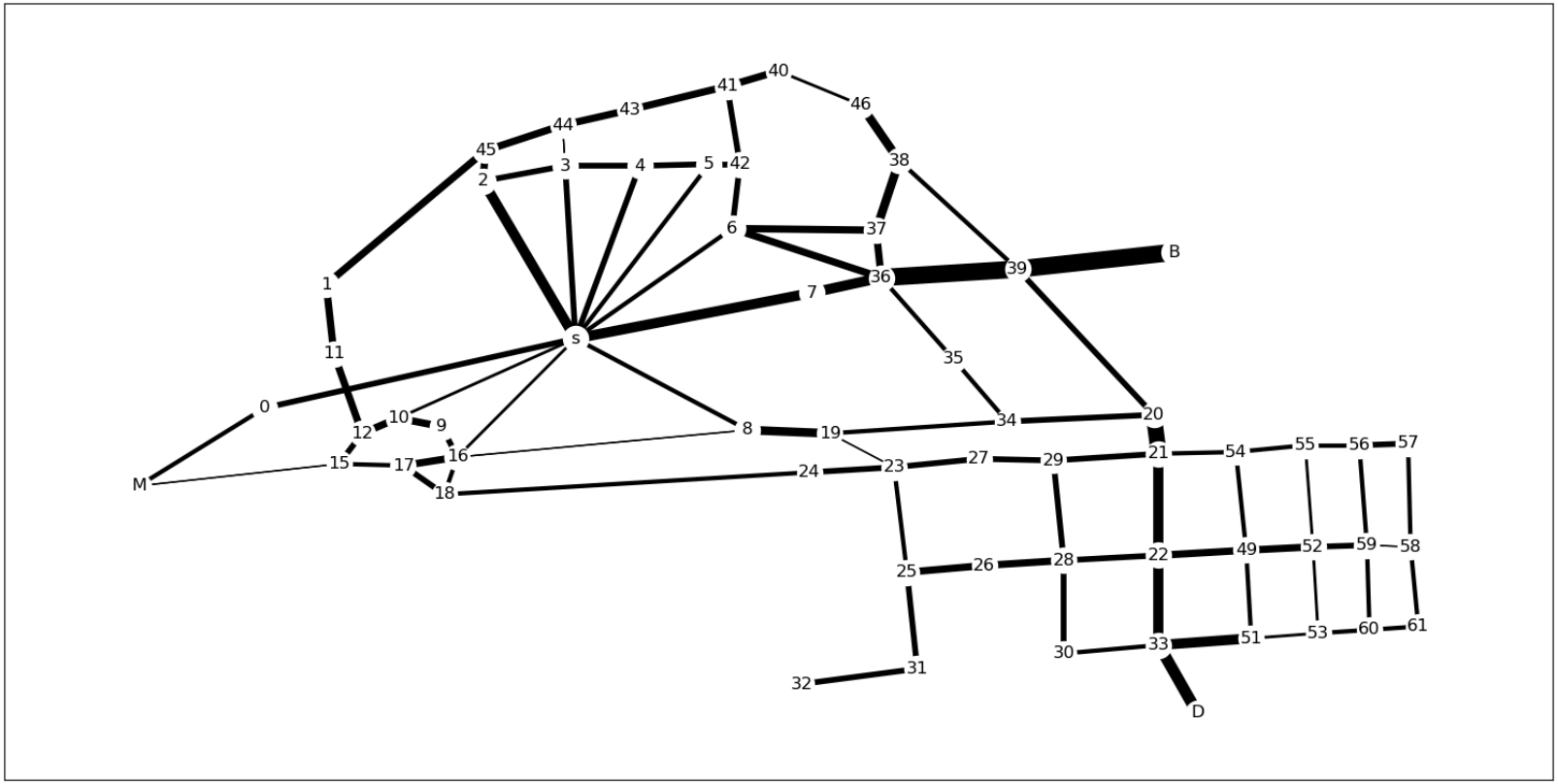
Étude expérimentale et modélisation des déplacements collectifs de piétons @Mehdi Moussaid

Graphe de capacité



Carte de la zone @OSM

Graphe de capacité



Graphe représentant la zone

3. Une méthode naïve

$\rightarrow$ les chemins de plus grande capacité entre le stade et les stations

Capacité d'un chemin

$p \subset E$
$C(p) = min\{C(e), e \in p\}$

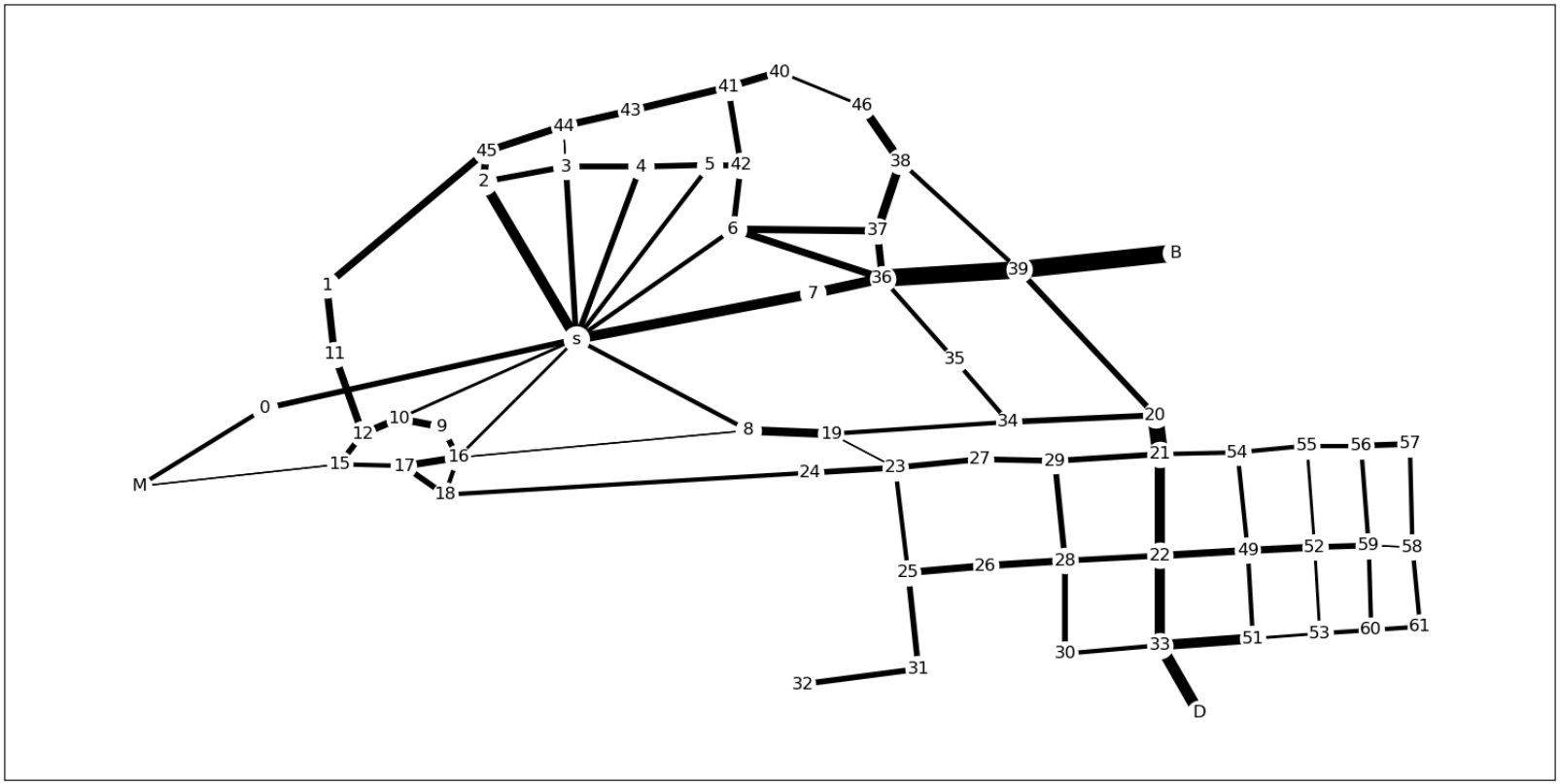
L'algorithme de Dijkstra

Chemin de poids minimal dans un graphe.

Algorithme Widest Path

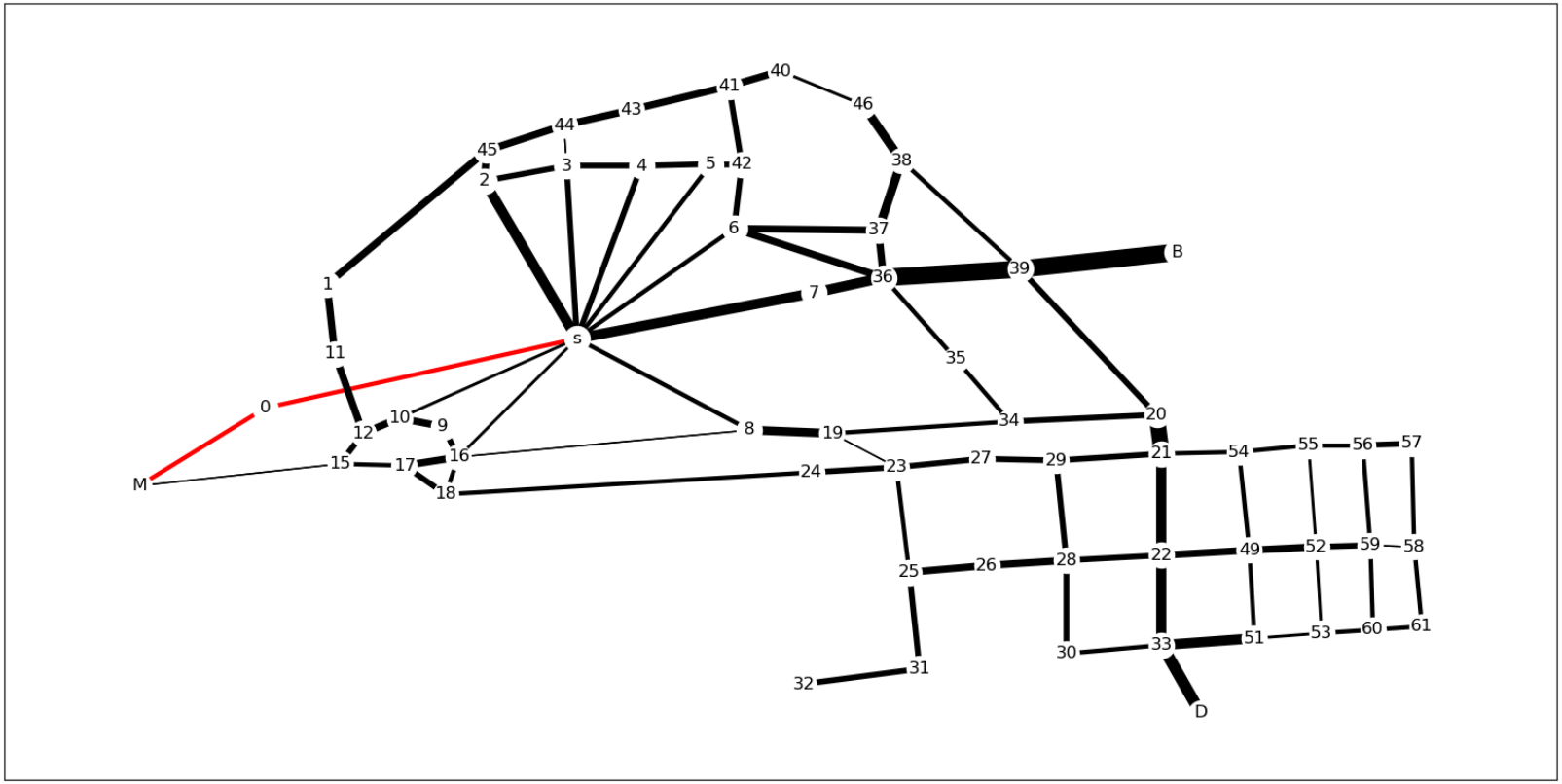
Algorithme de Dijkstra modifié $\rightarrow$ chemin de capacité maximale

Proposition de solution



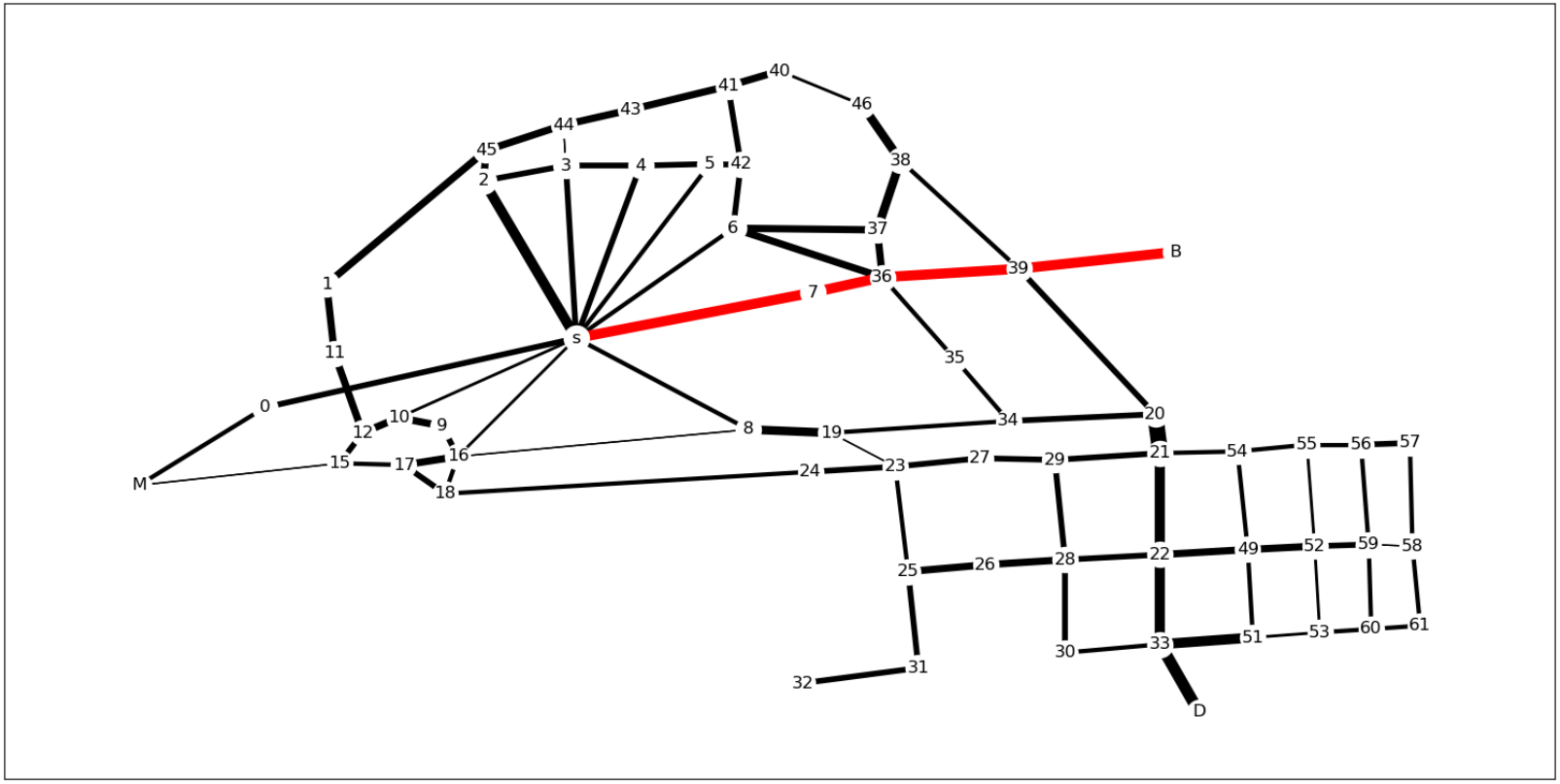
Proposition de solution

Chemin le plus large entre le stade et le métro, largeur : 3



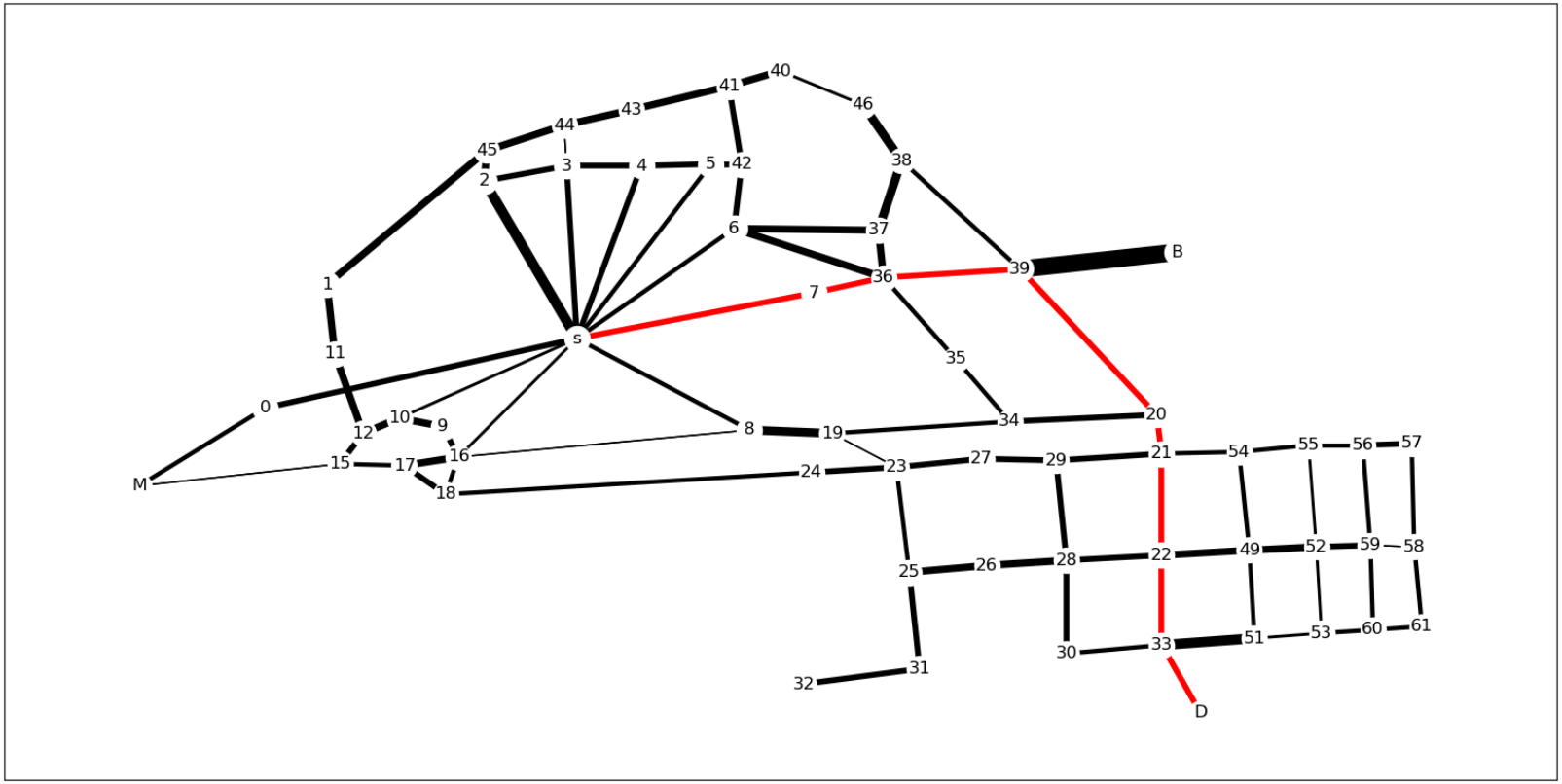
Proposition de solution

Chemin le plus large entre le stade et le RER B, largeur 7



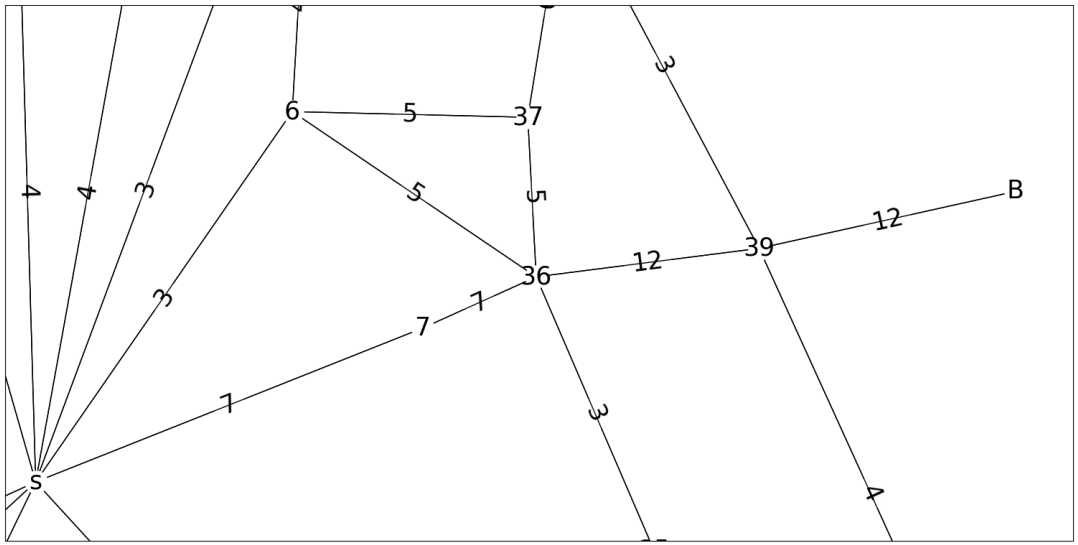
Proposition de solution

Chemin le plus large entre le stade et le RER D, largeur 4

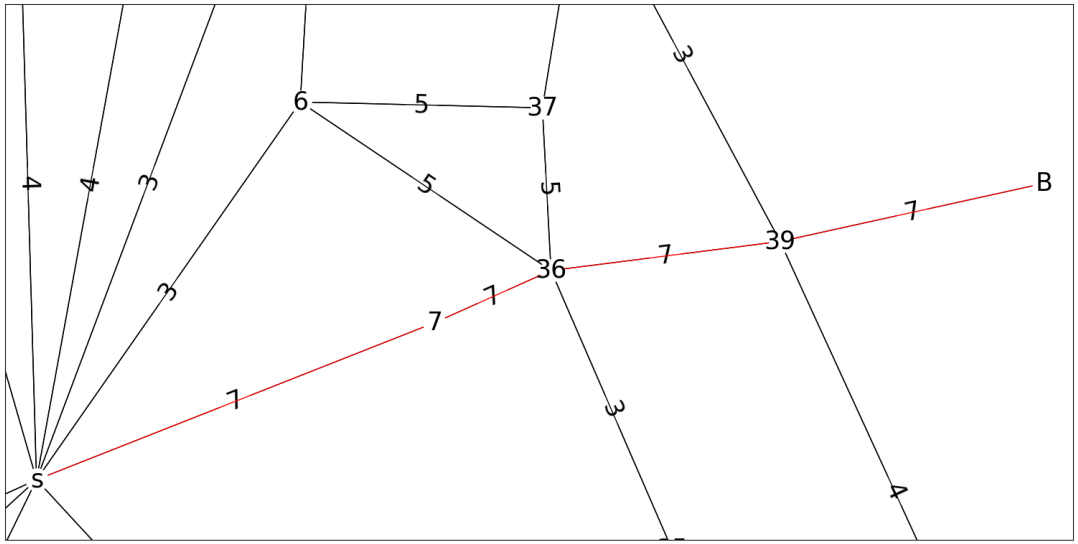


Analyse de la solution

Goulot d'étranglement

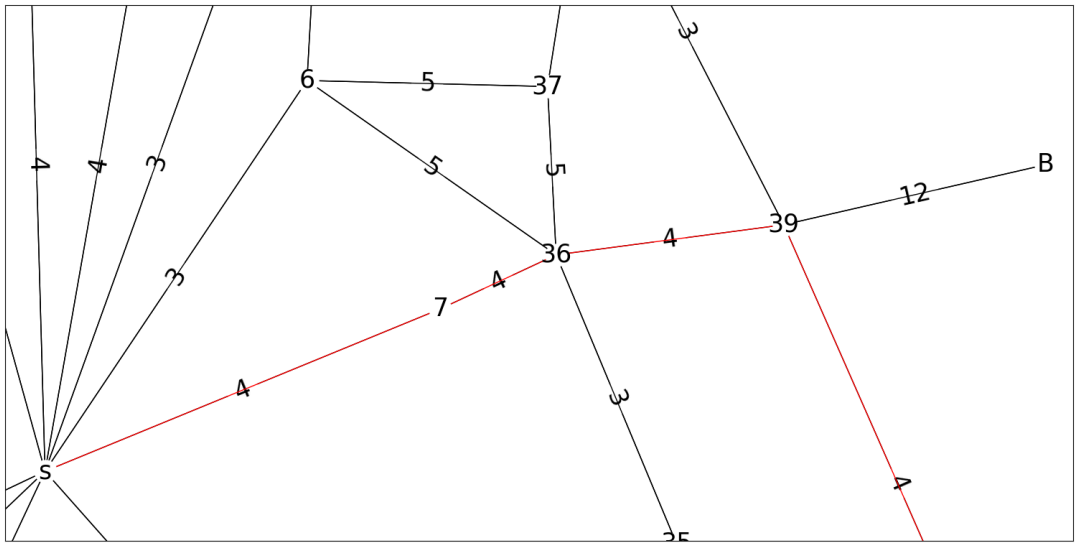


Zoom sur le sud-est du graphe

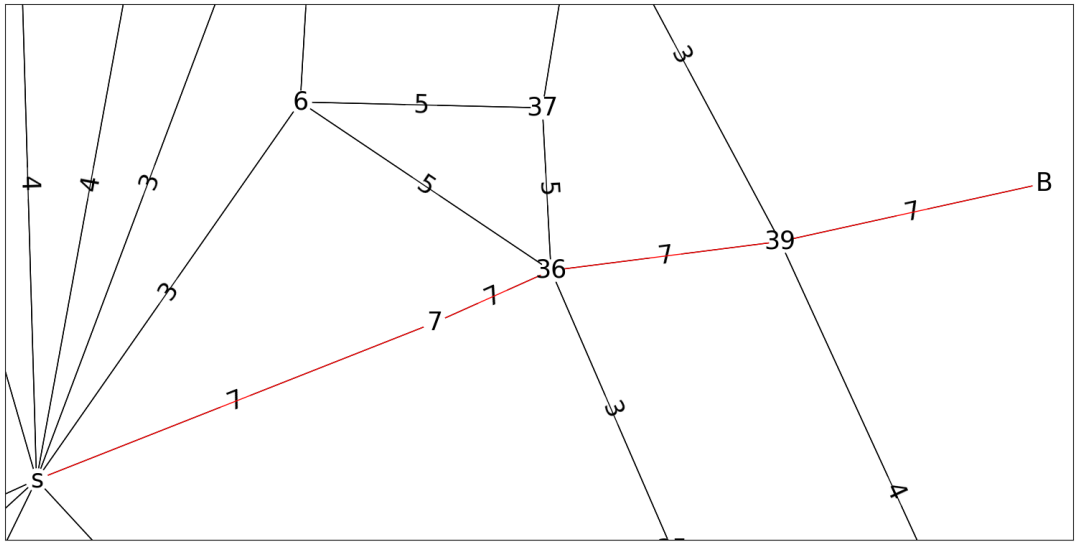


Analyse de la solution

Cannibalisme



Zoom sur le sud-est du graphe



Analyse de la solution

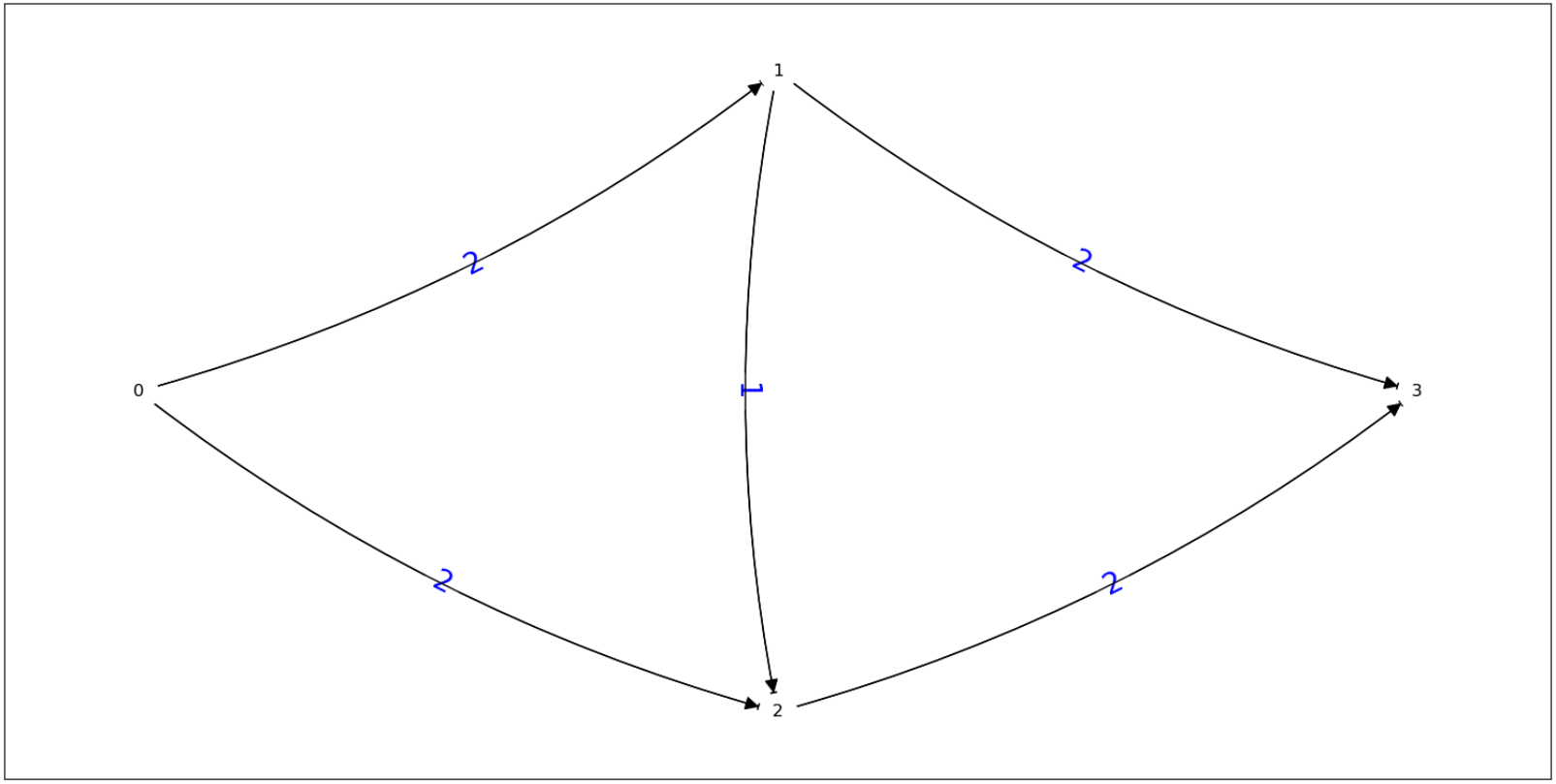
Largeur théorique : $3 + 4 + 7 = 14$
Débit théorique : $28 pers/s$

Largeur réelle: 10
Débit réel : $20 pers/s$
Temps d'évacuation : 1h08

4. Une méthode optimale : Le flot maximal

Algorithme d'Edmond-Karp

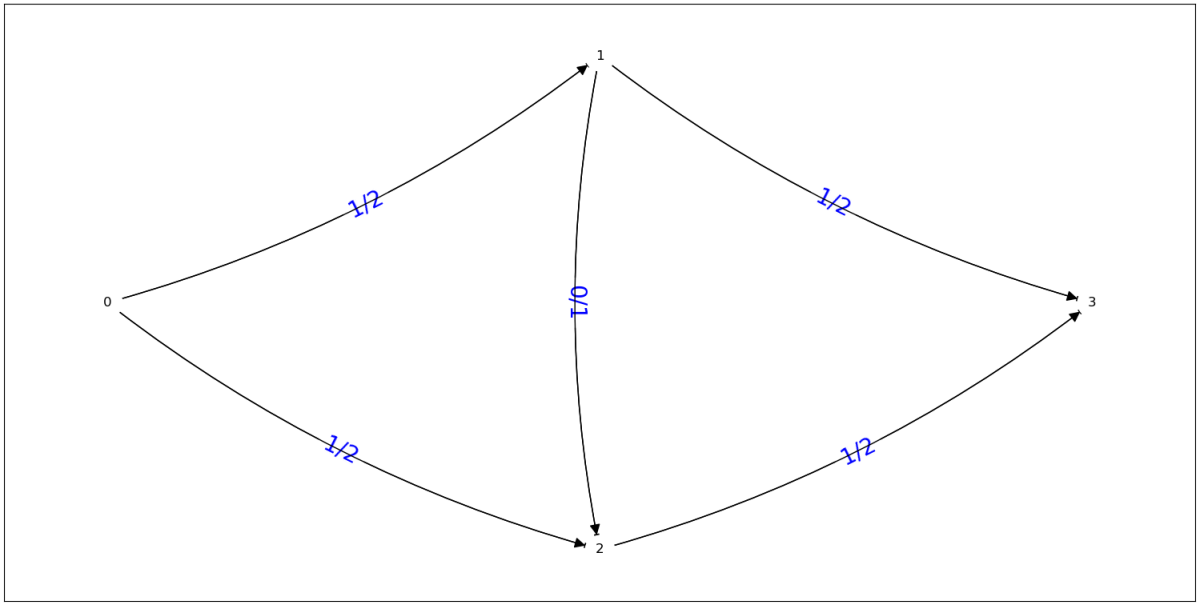
Graphe d'exemple



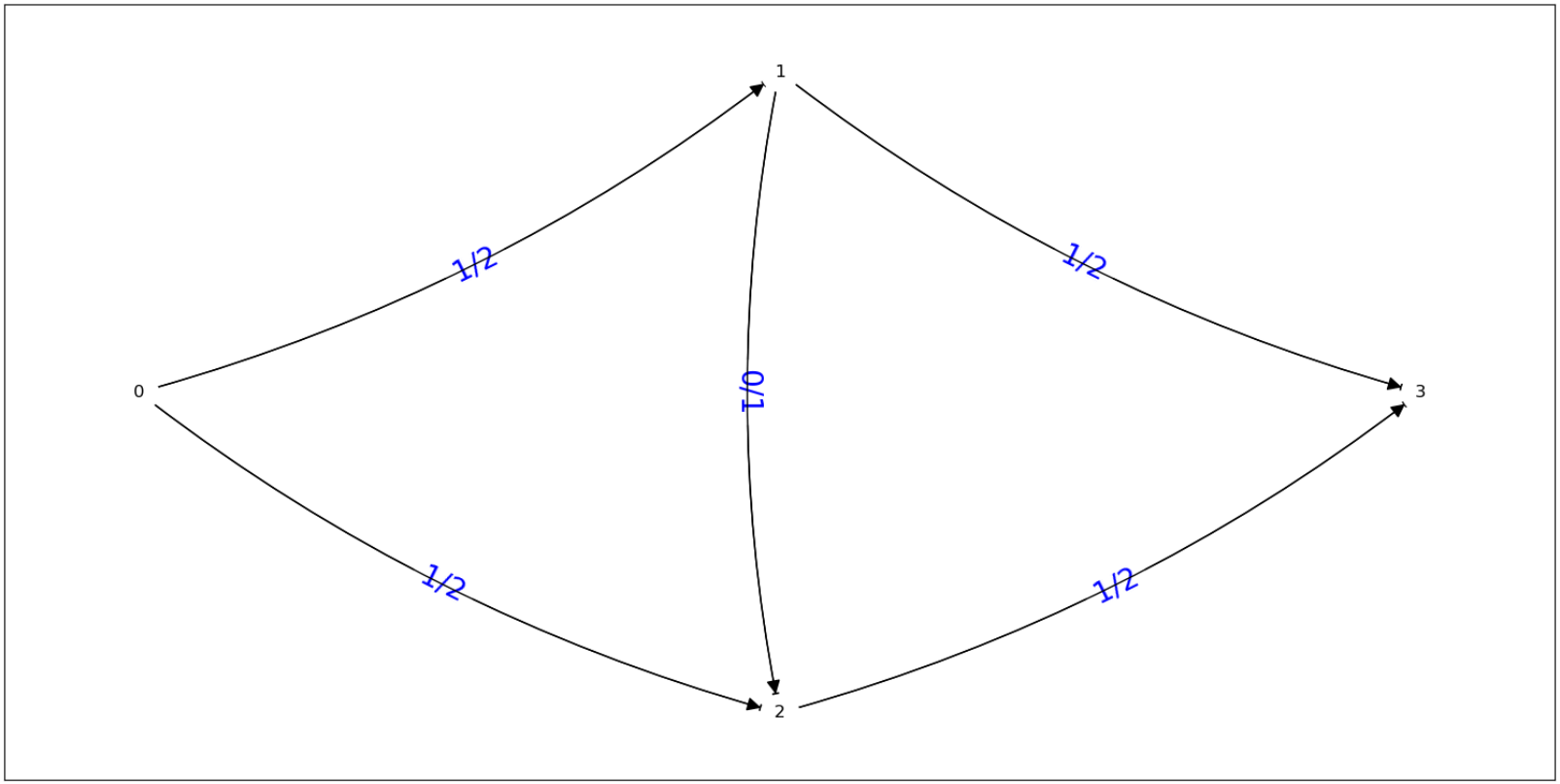
Graphe de flot

Soit $\varphi = (V, E, \phi, s, t)$ un graphe orienté pondéré par $\phi : E \to \mathbb{N}$ le flot passant dans chaque arètes.

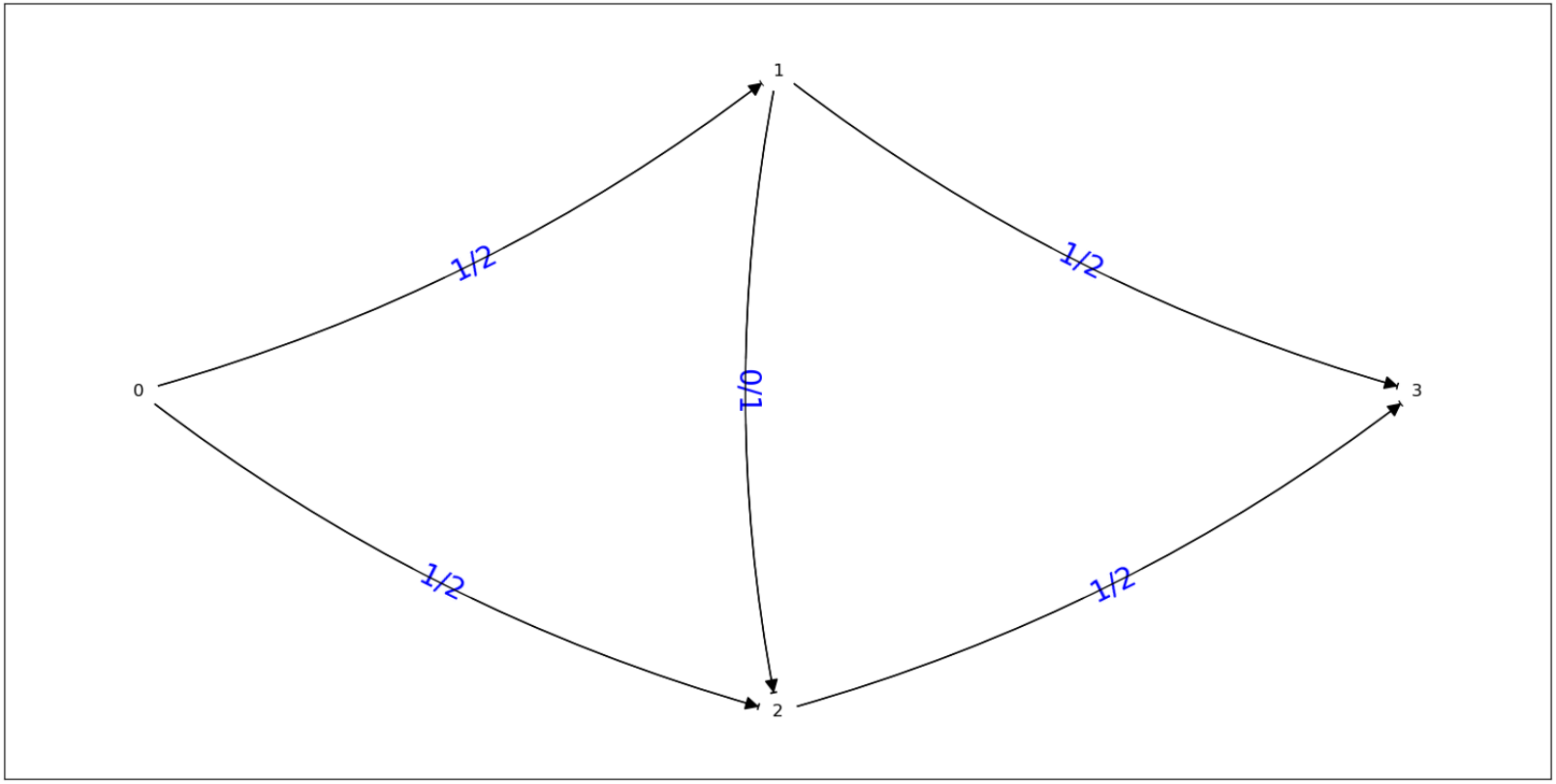
Le graphe de flot avec s : la source, t : le puit



Propriétés des flots: conservation du flot

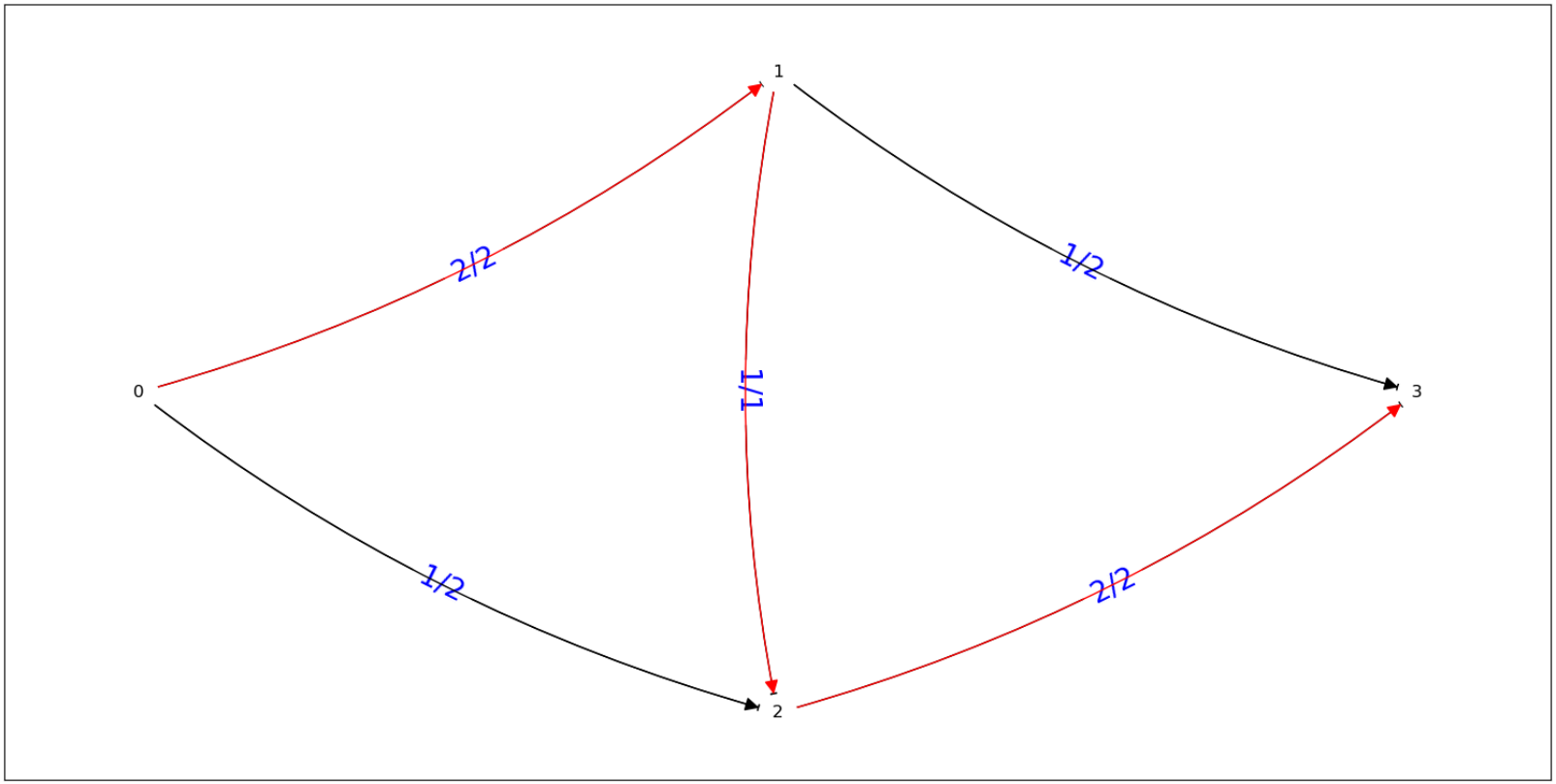


Propriétés des flots: valeur du flot



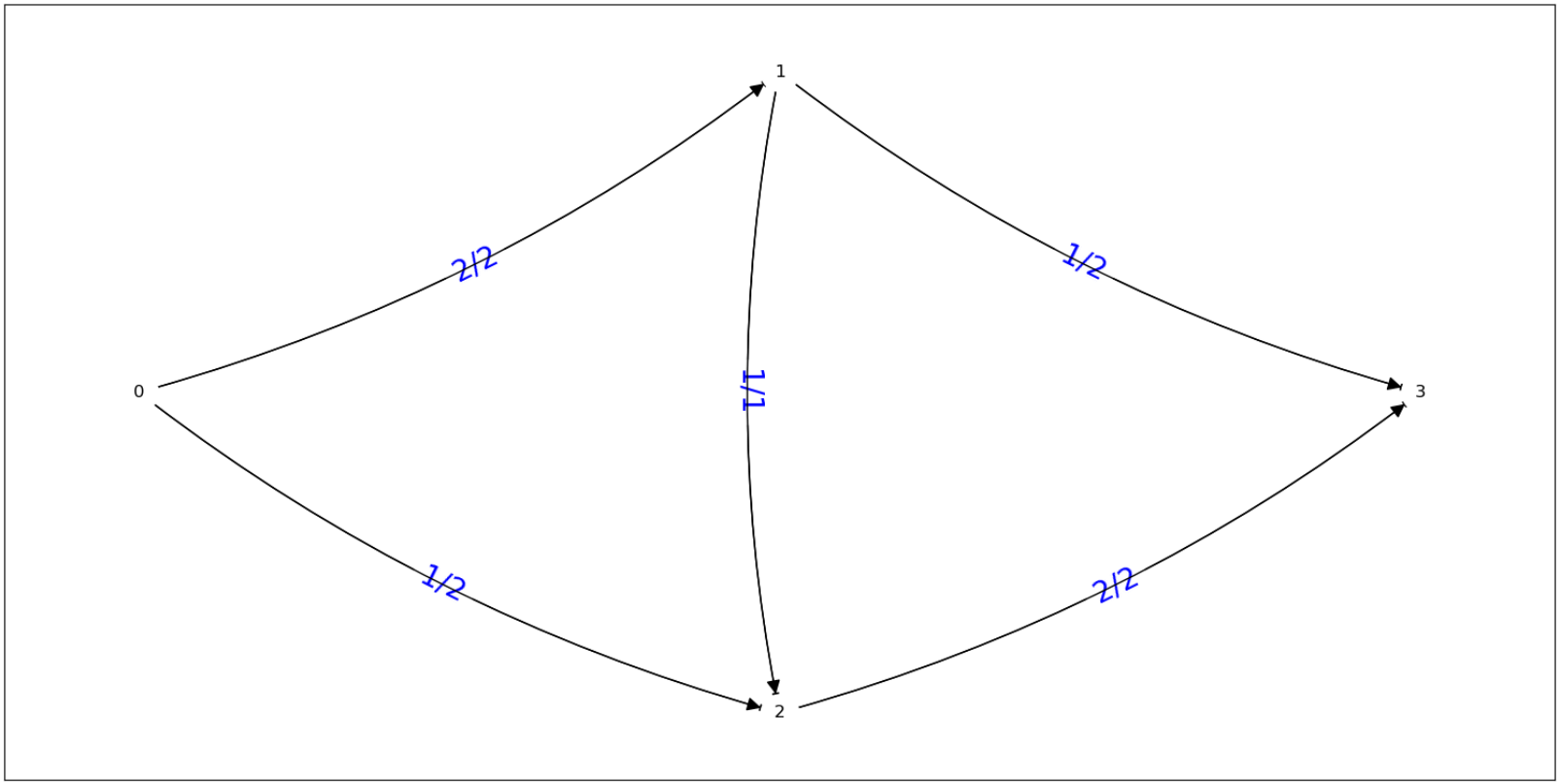
$V_\phi =$ le flot total sortant de s

Chemin augmentant

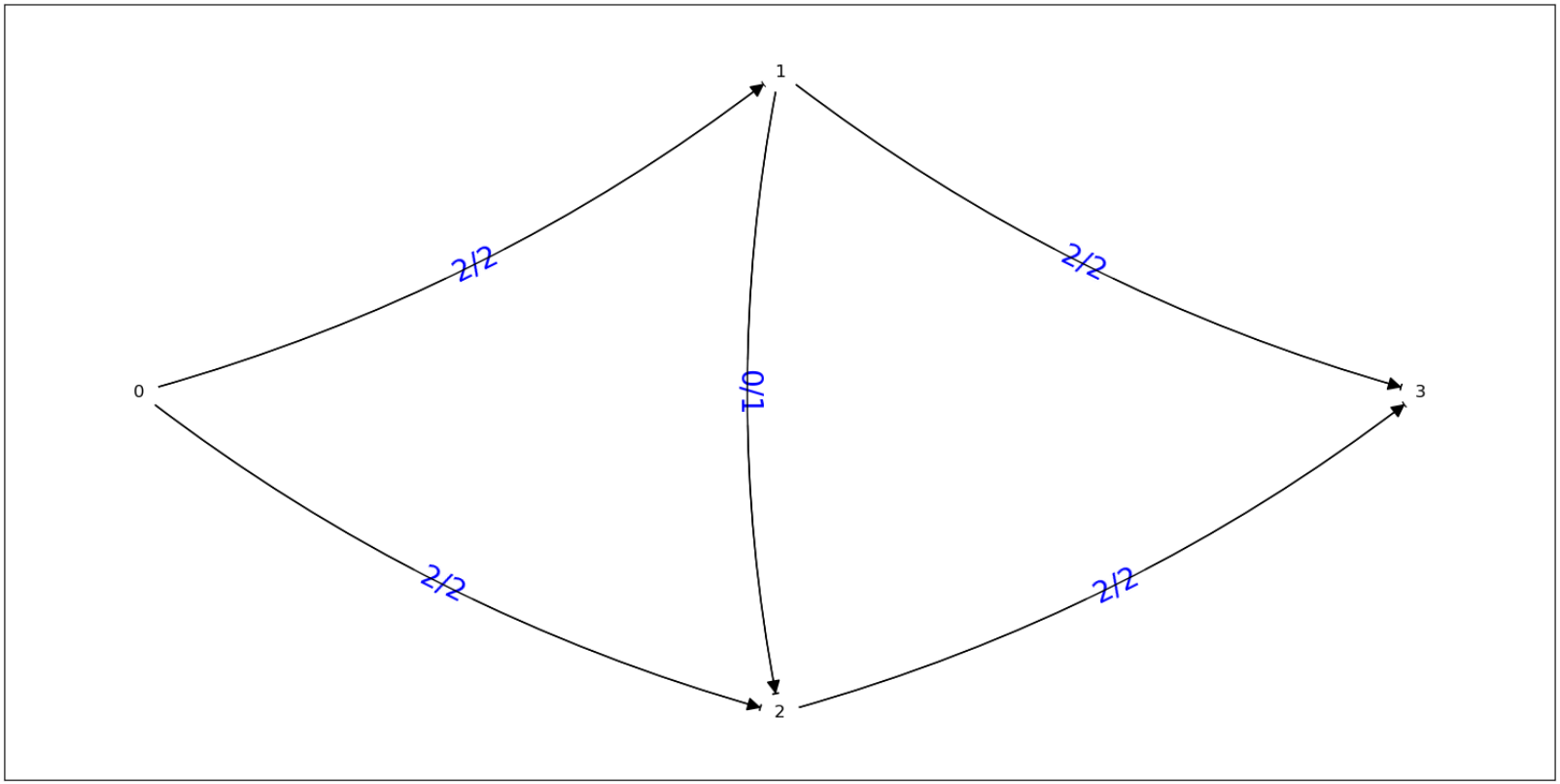


$$(p, d_\phi), p \subset E, d_\phi \in \mathbb{N}$$

Flot saturé

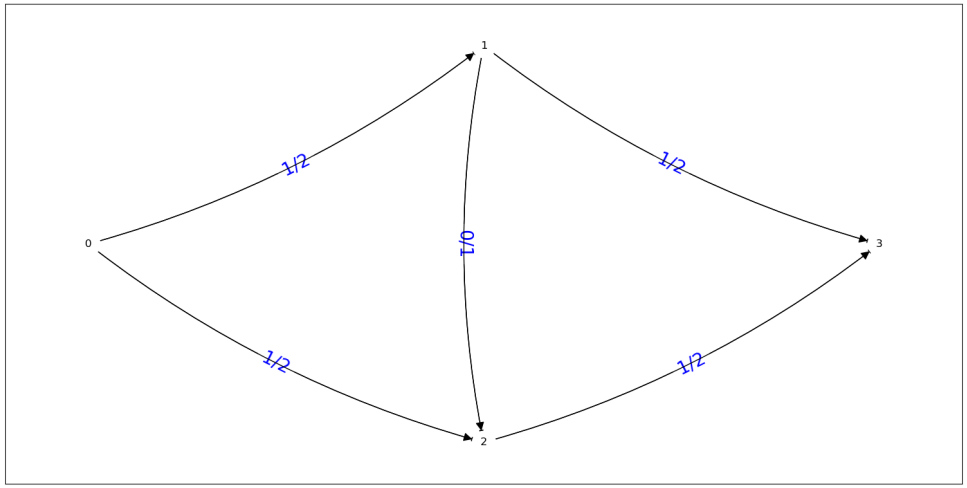


Flot maximal

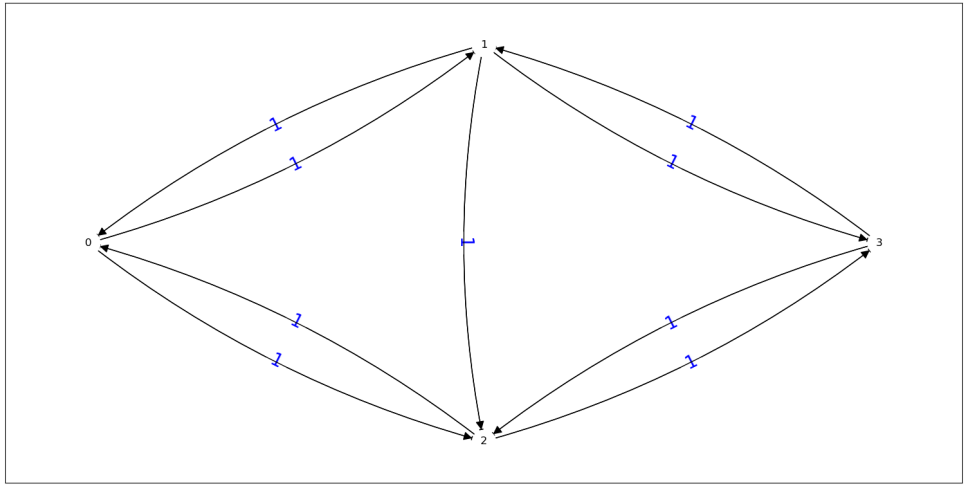


Graphe des augmentations

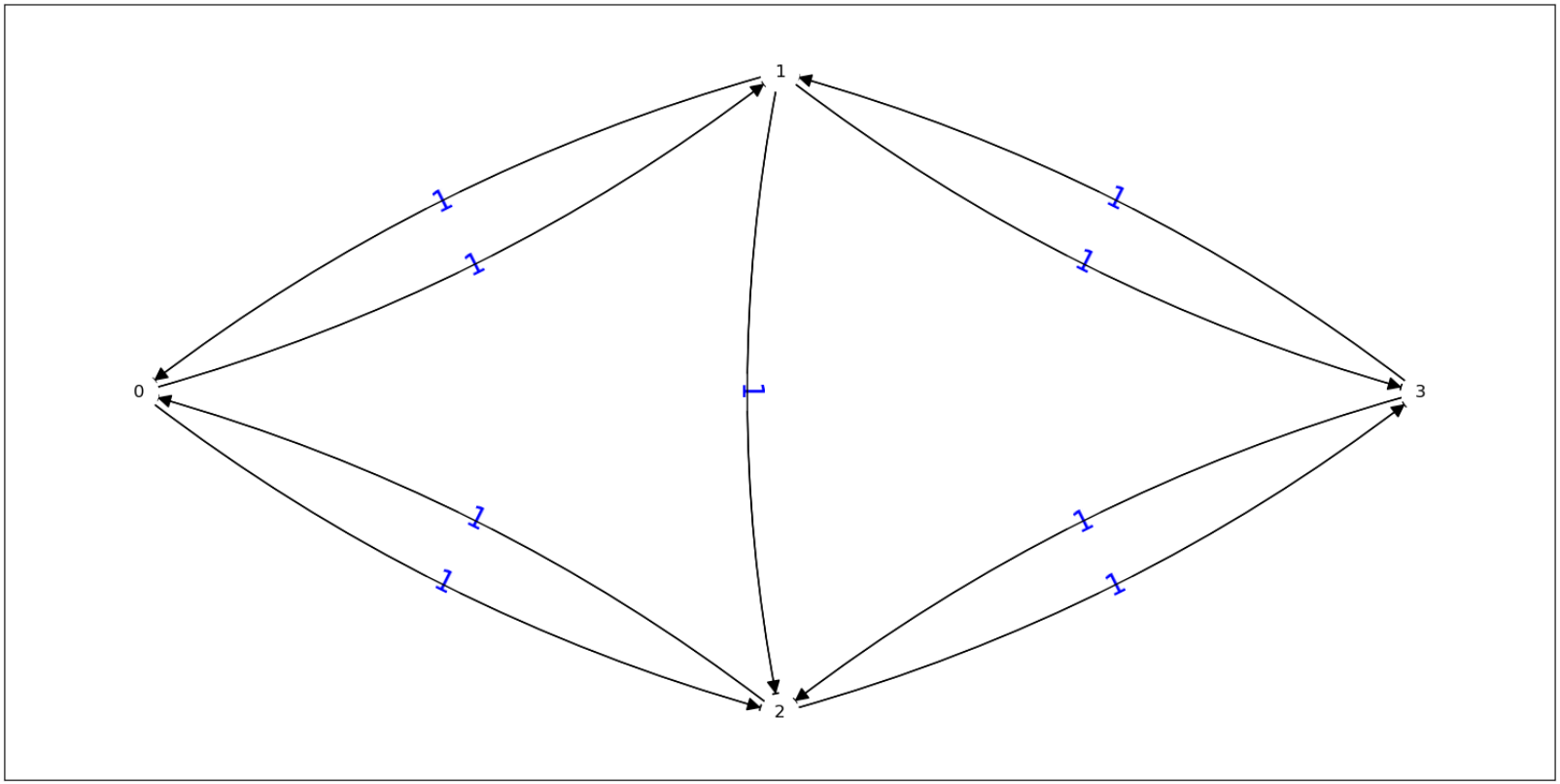
Soit $G_a = (V, E, C_r)$ un graphe orienté pondéré.



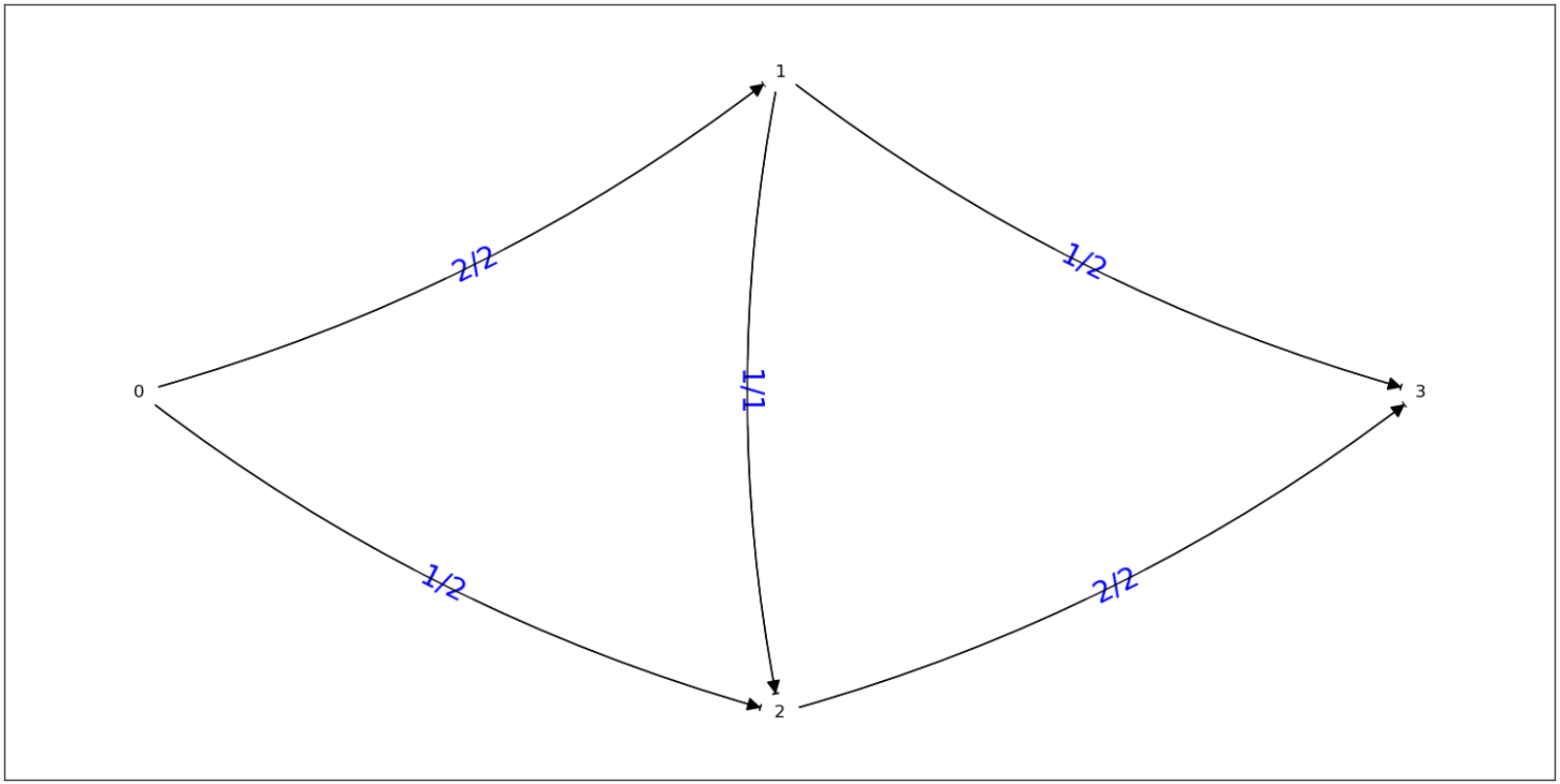
$C_r : E \to \mathbb{N}$ la capacité restante de chaque arête.



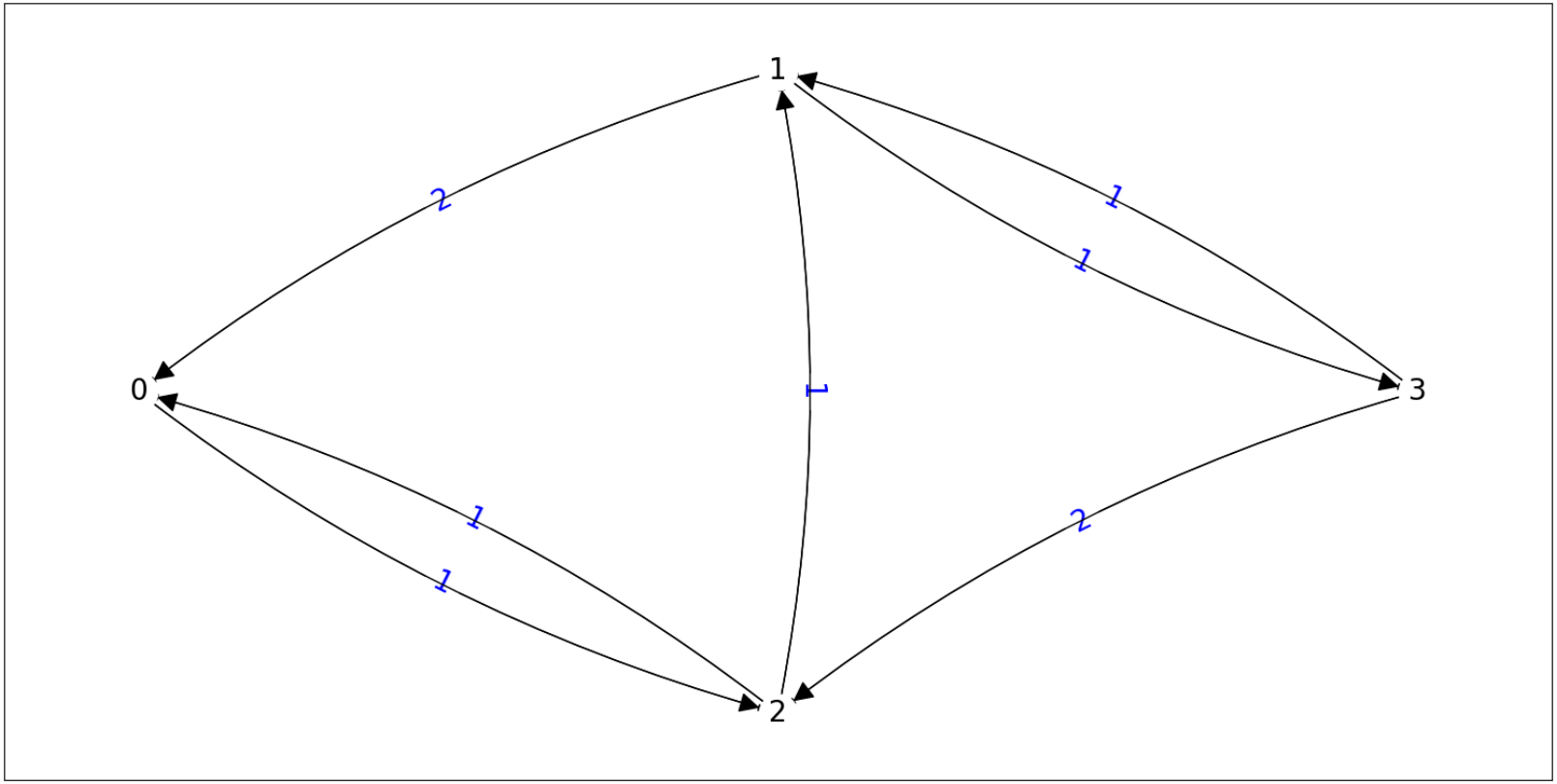
Arcs avant, Arcs arrières



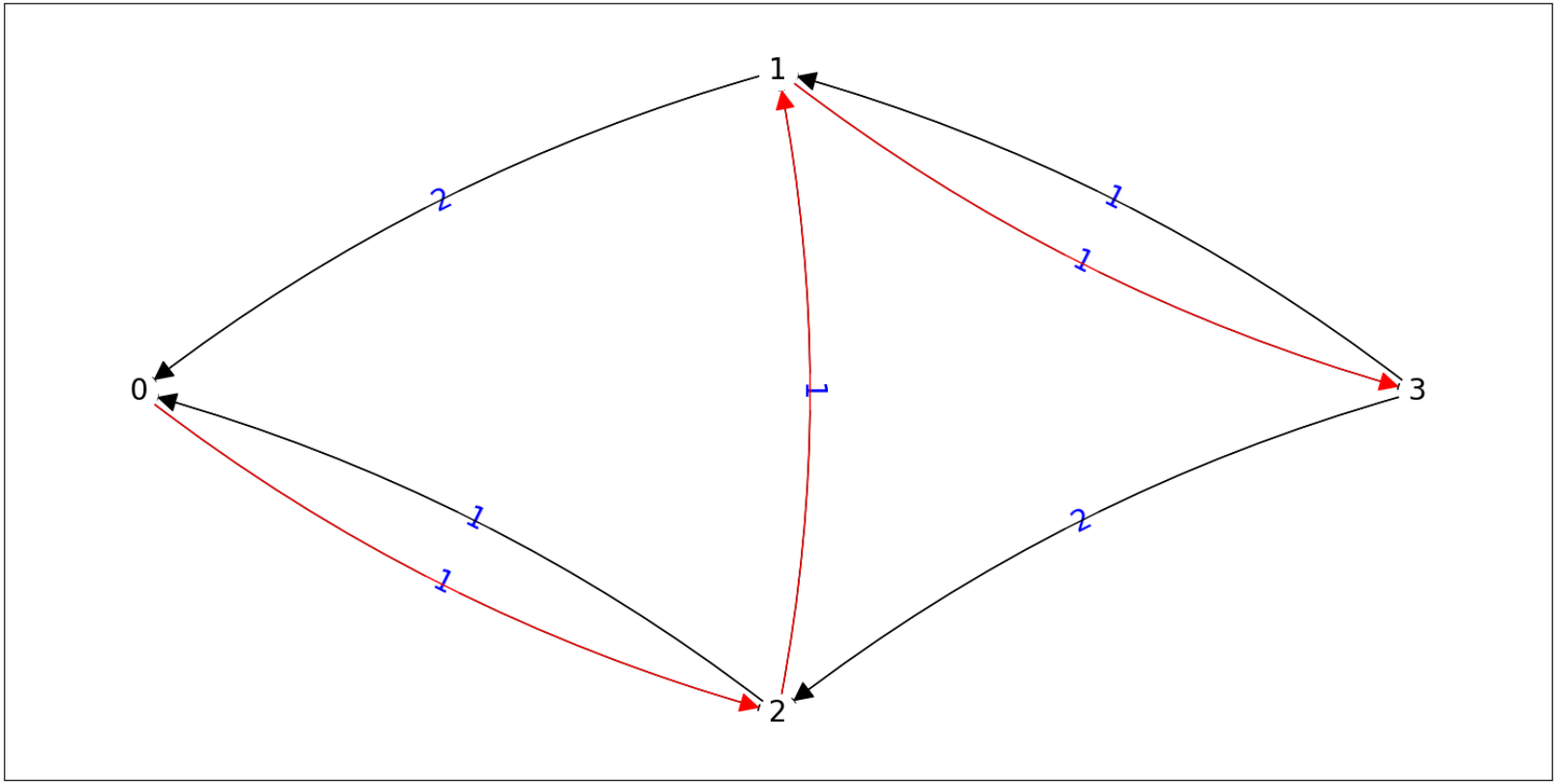
Flot saturé



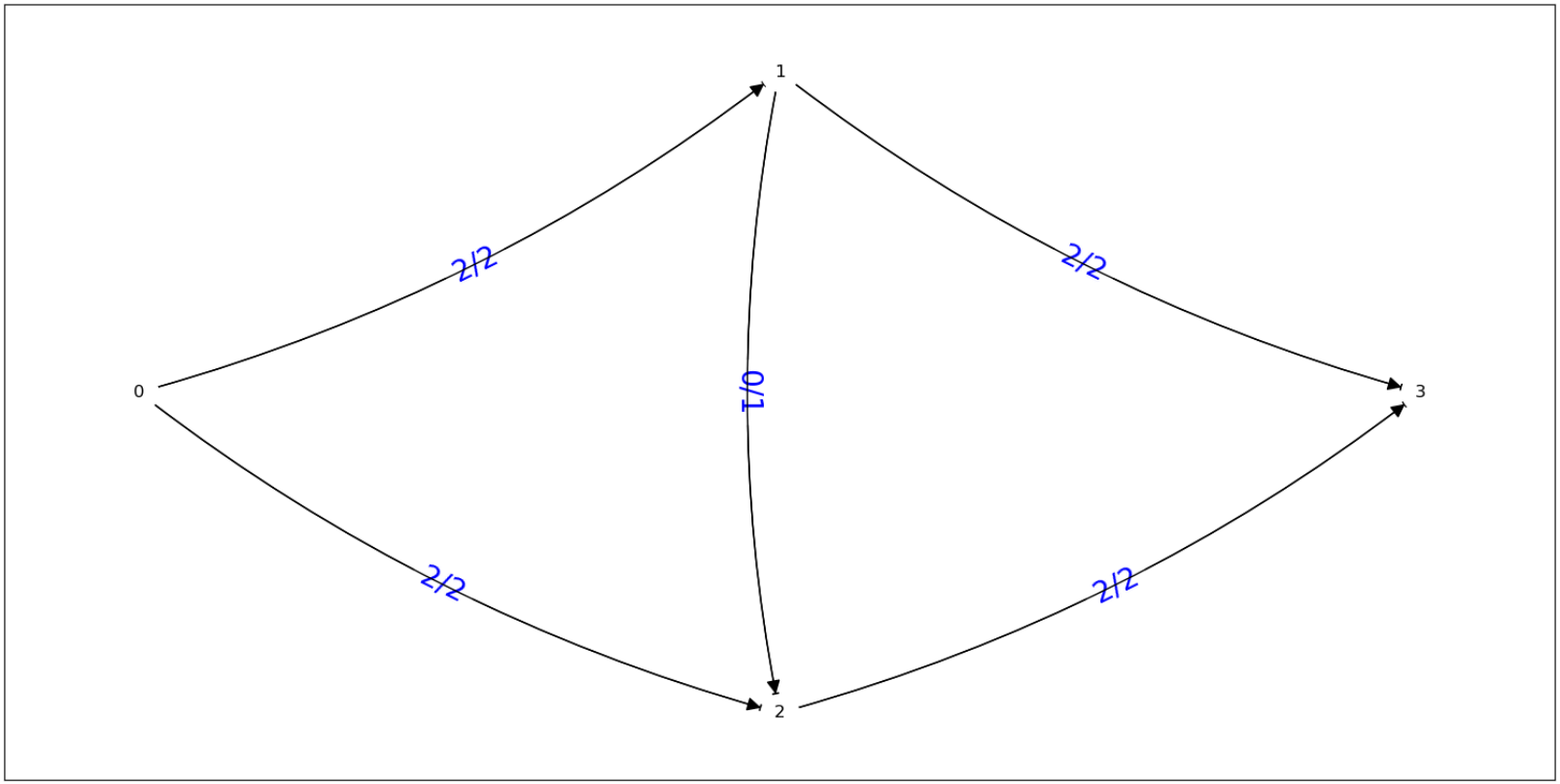
Nouveau graphe des augmentations



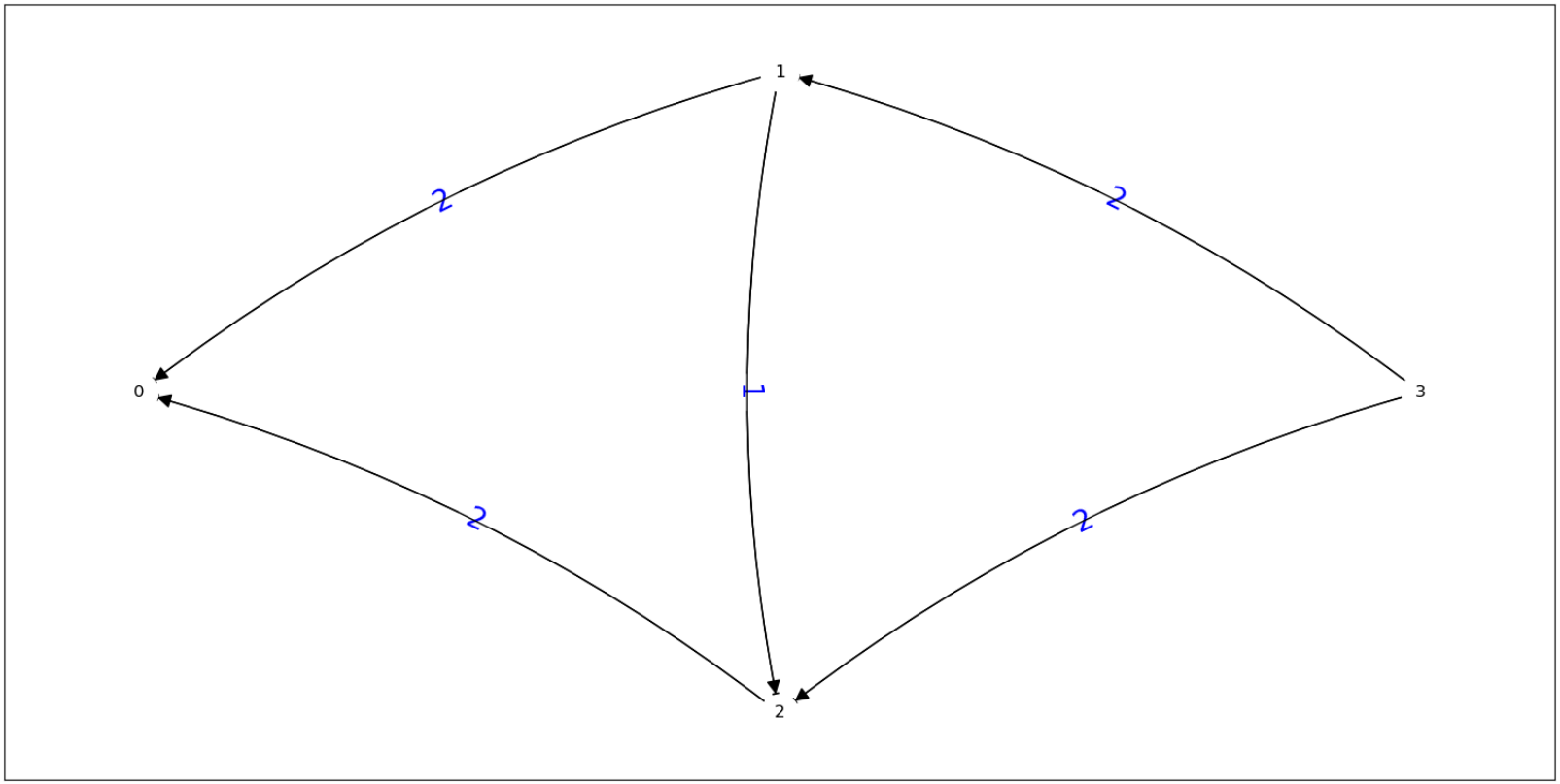
Nouveau chemin augmentant



Flot maximal



Graphe final des augmentations



Algorithme d'Edmond-Karp

$EK : G_c \rightarrow \varphi_{max}$

Objectif

Maximiser V_ϕ

Fonctionnement

Trouver des chemins augmentants dans G_a afin d'augmenter le flux.

```
Soit E-K(G_c = (V,E,C)):

  Phi <- Vide;
  G_a <- (V,
          E,
          C_r(u,v) = C_r(v,u)  = C(u,v));

  (P_a, dphi) = Chemin_augmentant(G_a);

  Tant que P_a existe :
    Mettre à jour G_a avec P_a, dphi;
    Mettre à jour Phi avec P_a, dphi;
    (P_a, dphi) = Chemin_augmentant(G_a);

  Renvoyer Phi
```

Pseudo-Code de l'algorithme d'Edmond Karp

- G_c le graphe des capacités

Algorithme d'Edmond-Karp

Recherche du chemin augmentant

Parcours en largeur : plus court chemin en nombre d'arc

-> Compléxité en $O(|V| * |E|^2)$

Renvoie $(p, d\phi)$ un chemin augmentant

Mise à jour de φ

$\forall e = (u, v) \in p$

si e est un arc avant:

$\phi(u, v) \leftarrow \phi(u, v) + d\phi$

si $e = (v, u)$ est un arc arrière :

$\phi(u, v) \leftarrow \phi(u, v) - d\phi$

Mise à jour de G_a

$\forall e = (u, v) \in p, C_r(u, v) \leftarrow C_r(u, v) - d\phi$

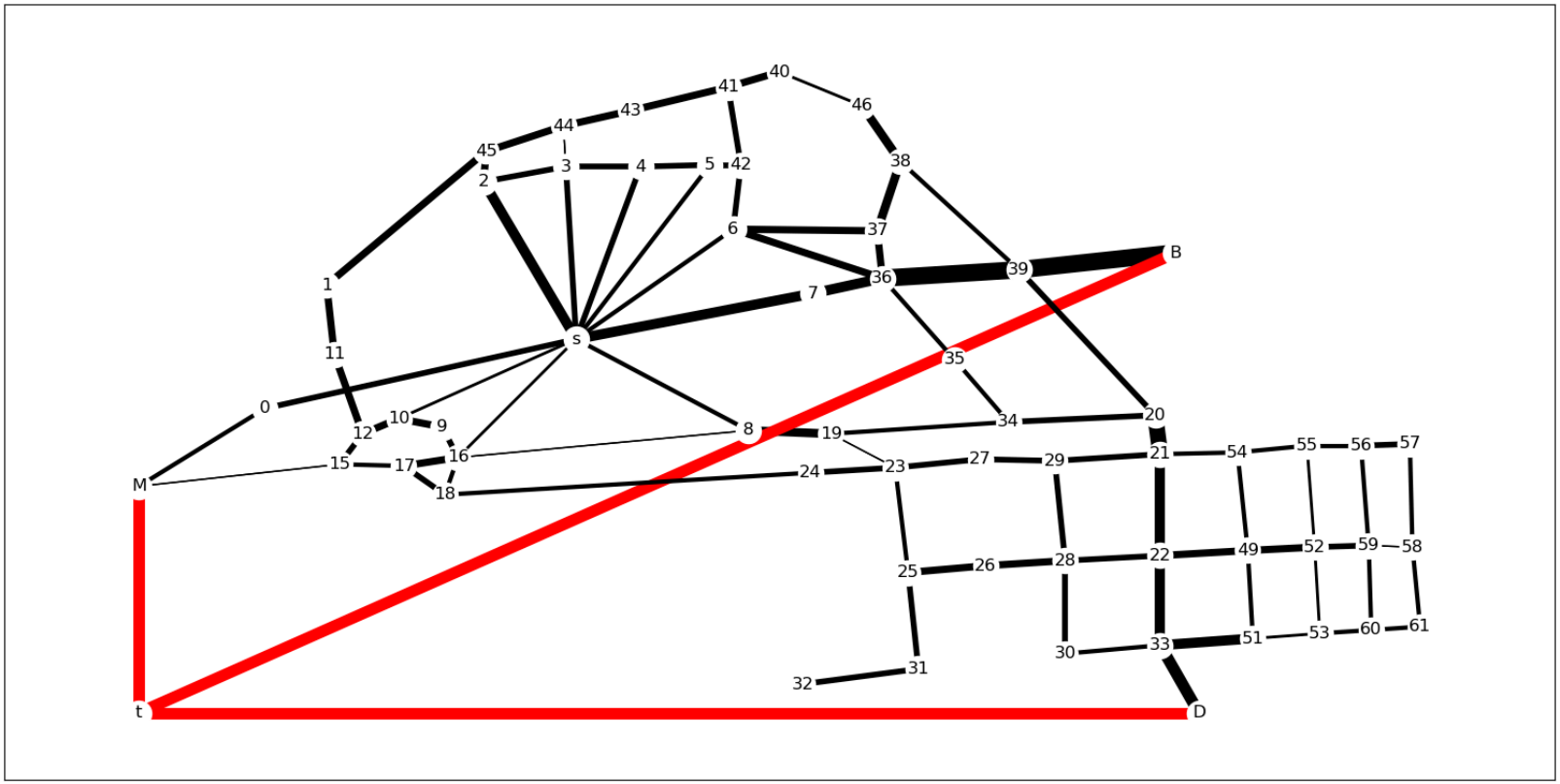
$\forall e = (u, v) \in p, C_r(v, u) \leftarrow C_r(v, u) + d\phi$

Dans le cas du Stade de France

Capacité proportionnelle à la largeur de la rue.
La source : le Stade de France
Le puit: un noeud fictif relié par des arêtes de capacité maximale aux stations de transport en commun.
Objectif:

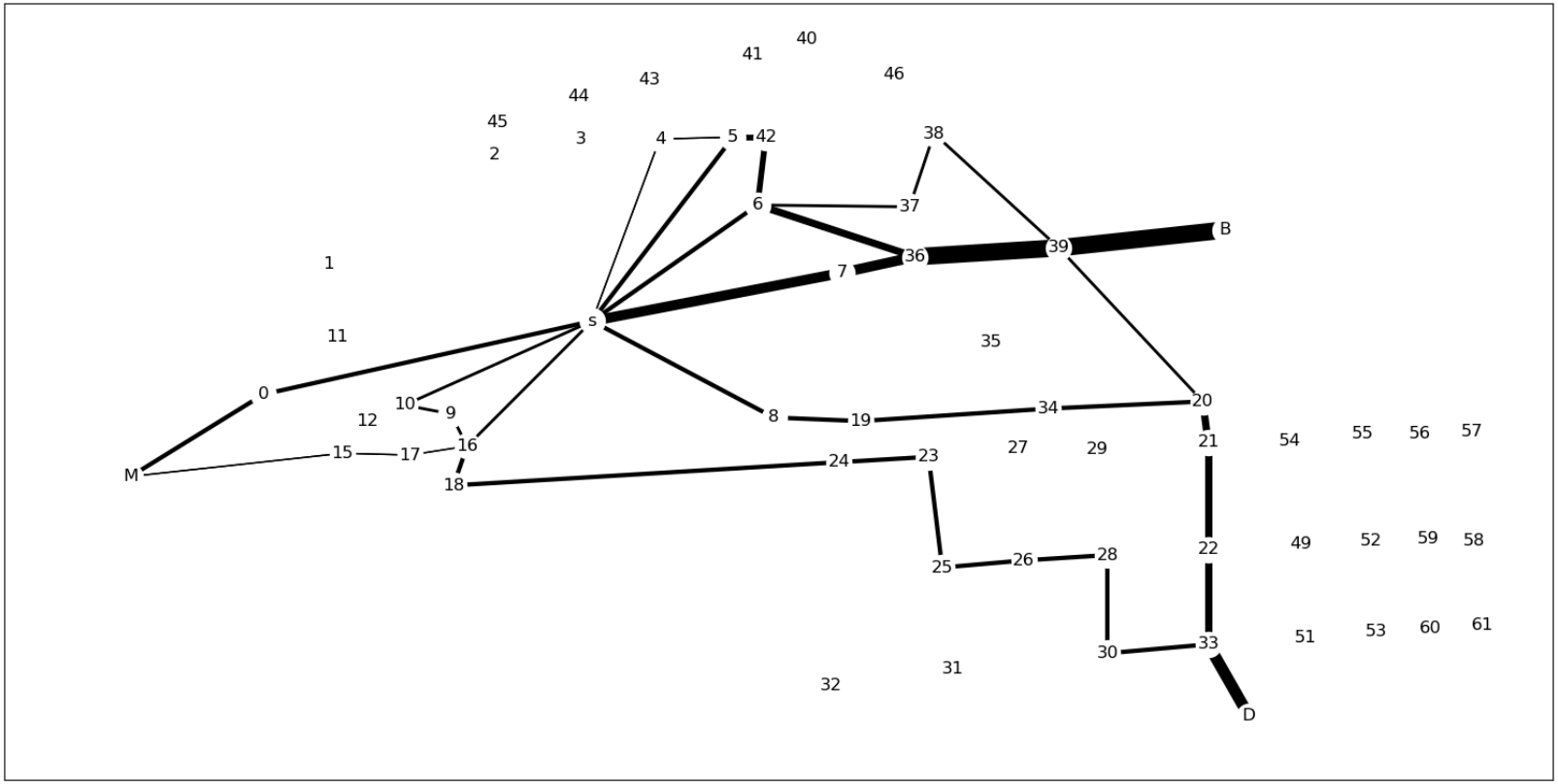
- Le flot (un débit de personne) maximal, pour évacuer le plus efficacement la foule.

Graphe de capacité



Résultats expérimentaux

Valeur du flot : 23, débit : $46 pers/s$, temps :$30 min$
(WP : $10, 20 pers/s$)

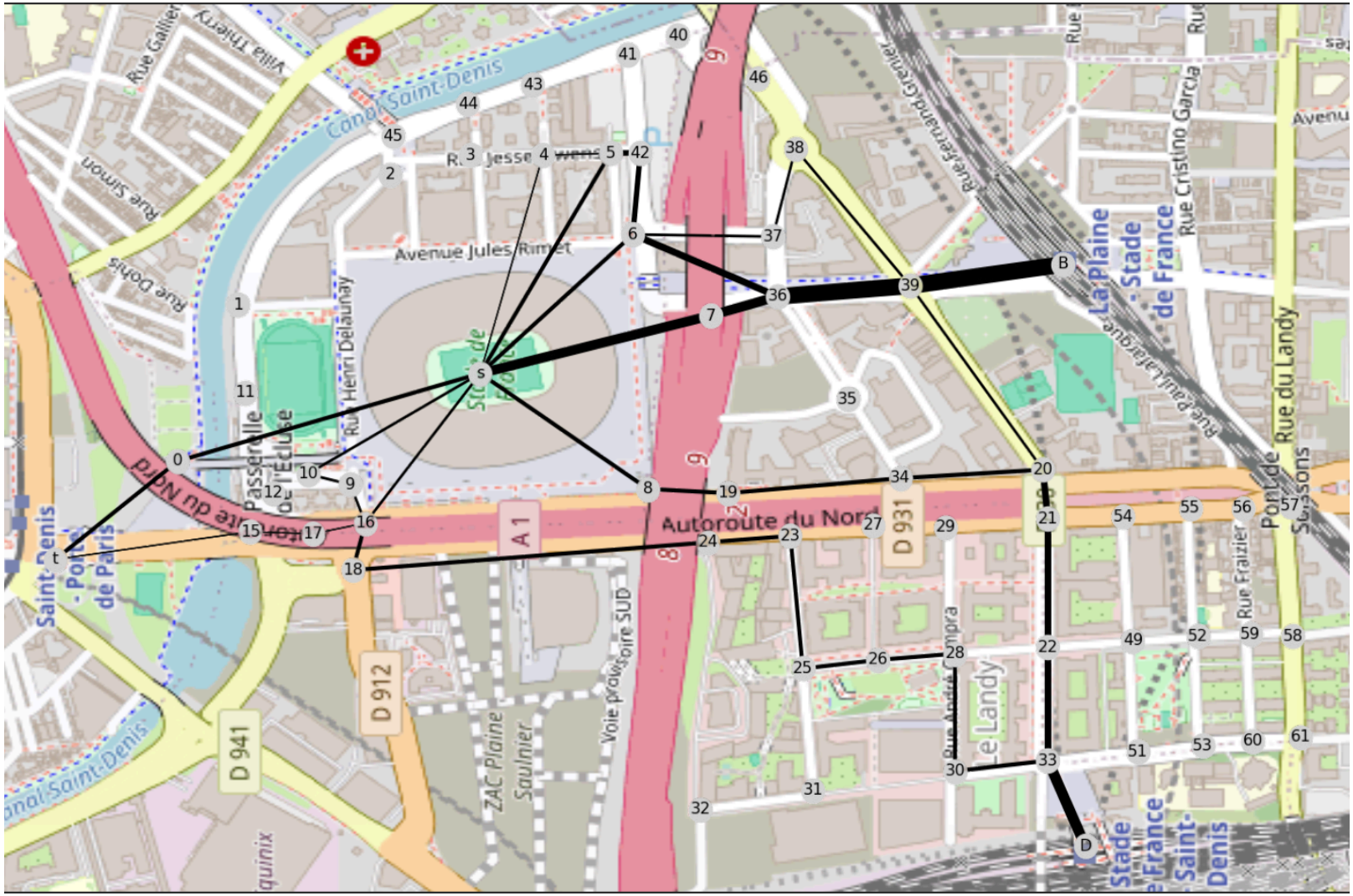


5. Analyse des résultats

Comparaison W-P v. E-K

- Pas de cannibalisme.
- Pas de goulot détranglement.
- Performance:
  E-K 2 fois plus rapide pour le même niveau de sécurité

Critique du résultat



6. Annexe

Idée de preuve pour la correction et la complexité d'Edmond-Karp

Correction: équivalence entre absence de chemin augmentant et flot maximal atteint.

Complexité : si (u,v) est critique à deux moments distincts alors, u s'est éloigné de s d'au moins 2 entre ces deux moments, or $\delta(s,u) \leq |V|$ donc (u,v) n'est critique qu'au plus $\frac{|V|-2}{2}$ fois. D'où nombre d'itération majoré par $|E| * |V|$ or chaque itération a une complexité en $O(|E|)$: parcours en largeur + màj φ et màj G_a.
$\rightarrow O(|V| * |E|^2\ )$

Code - Imports et Classe de graphe:

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

```python
class graph:
    def __init__(self,n) -> None:
        self.size = n
        self.matrix = [[0 for _ in range(n)] for _ in range (n)]
        self.adj = [[]for _ in range(n)]
        self.s = -1
        self.t = -1

    def weight(self,x,y):
        return self.matrix[x][y]

    def add_edge(self,x,y,w):
        # directed edges
        assert(w!=0)
        if self.weight(x,y) == 0:
            self.adj[x].append(y)
        self.matrix[x][y] += w
```

```python
class graph:
    def update_edge(self,x,y,v):
        if v == 0:
            self.matrix[x][y] = v
            self.adj[x].remove(y)
        else:
            if v>0:
                self.matrix[x][y] = v
            else:
                self.matrix[y][x] = -v

    def edges(self,x):
        return [(y, self.weight(x,y)) for y in self.adj[x]]

    def custom_copy(self):
        g = graph(self.size)
        g.matrix = [t.copy() for t in self.matrix]
        g.adj = [t.copy() for t in self.adj]
        g.s = self.s
        g.t = self.t
        return g
```

Code - Classe de tas max:

```python
class max_heap:
    """
    Une implementation personnelle d'un tas max stockant des tuples (int, any)
    basé sur un arbre linéarisé dans un tableau (ici une liste).

    A custom implementation of max heap to store tuples (int, any)
    based on a linear tree (a list).
    """
    def __init__(self) -> None:
        self.data = []
        self.size = 0


    def switch(self,i1,i2):
        assert(0 <= i1 < self.size)
        assert(0 <= i2 < self.size)
        tmp = self.data[i1]
        self.data[i1] = self.data[i2]
        self.data[i2] = tmp


    def sift_up (self,index):
        k = index
        k_prime = int((k-1)/2)
        p = self.data[k][0]
        while self.data[k_prime][0] < p:
            self.switch(k,k_prime)
            k = k_prime
            k_prime = int((k-1)/2)
```

```python
class max_heap:
    def sift_down(self,index):
        k = index
        k_prime = k
        if 2*k+1 < self.size and self.data[2*k+1][0] > self.data[k_prime][0]:
            k_prime = 2*k+1
        if 2*k+2 < self.size and self.data[2*k+2][0] > self.data[k_prime][0]:
            k_prime = 2*k+2
        while k!=k_prime:
            self.switch(k,k_prime)
            k = k_prime
            if 2*k+1 < self.size and self.data[2*k+1][0] > self.data[k_prime][0]:
                k_prime = 2*k+1
            if 2*k+2 < self.size and self.data[2*k+2][0] > self.data[k_prime][0]:
                k_prime = 2*k+2


    def push(self, p,v):
        self.data.append((p,v))
        self.size += 1
        self.sift_up(self.size-1)

    def pop (self):
        if self.size > 1:
            self.switch(0, self.size-1)
        res = self.data.pop()
        self.size-=1
        self.sift_down(0)
        return res
```

Code - Widest-Path, reconstruction du chemin:

```python
def rebuild_path(partial, start,end):
    """
    Partial: le tableau des prédecesseurs (any list) (içi int list)
    Partial: the predecessor array (any list) (her an int list)
    """
    path = [end]
    current = end
    while current !=  start :
        next = partial[current][0]
        path.append(next)
        current = next
    path.reverse()
    return path
```

```python
def widest_path (g, start, end):
    """
    Version adaptée de Dijkstra au probleme considéré :
    - chemin le plus large: chemin dont le minimum des capacités
    de ses arètes est maximal.

    Adapted implementation of Dijkstra's algorithm to solve the
    widest-path problem: looking for the path with the biggest
    minimum capacity along its way.
    """
    node =  start
    bag = max_heap()
    bag.push(float("inf"),node)
    partial = [(-1,0) for _ in range(g.size)]
    partial[node] = (float("inf"), node)
    treated = set()
    while end not in treated:
        (capa,node) = bag.pop()
        treated.add(node)
        edges = g.edges(node)
        for (y,w) in edges:
            new_capa = min(capa,w)
            if y not in treated and ( partial[y][1] == 0  or new_capa > partial[y][1]):
                bag.push(new_capa,y)
                partial[y] = (node,new_capa)
                # garder node permet de garder le meilleur prédecesseur

    widest_path = rebuild_path ( partial, start, end)
    return (widest_path, partial[end][1])
```

Code - Edmond-Karp et fonctions associées:

```python
def init_ag(g,sinks) :
    """
    Construit le graphe des augmentations
    Builds the augmenting graph

    On ajoutera des arcs de capacités infinies entre t (un puits virtuel)
    et les differents points de sorties réels

    We will add to ag edges of infinite capacity between t
    (a virtual sink node) and the different real sink nodes
    """
    if sinks != None:
        n = g.size
        ag = g.custom_copy()
        ag.matrix = [row+[0] for row in ag.matrix]
        ag.matrix.append([0 for _ in range(n +1)])
        ag.adj.append([])
        ag.size = n+1
        ag.t = n
        ag.s = g.s
        for s in sinks:
            ag.add_edge (s,ag.t, float("inf"))
    else:
        ag = g.custom_copy()
    return ag
```

```python
def update_ag(ag,ap,dphi):
    n = len(ap)

    for i in range(1,n):
        x = ap[i-1]
        y = ap[i]
        ag.update_edge(x, y, ag.weight(x,y)-dphi)
        ag.update_edge(y, x, ag.weight(y,x)+dphi)

def init_flow(ag):
    f = graph(ag.size)
    f.s = ag.s
    return f

def update_flow(f,ap,dphi):
    n = len(ap)
    for i in range(1,n):
        x = ap[i-1]
        y = ap[i]
        f_inverse = f.weight(y,x)
        diff = dphi - f_inverse
        f.add_edge(x,y,diff)
```

Code - Edmond-Karp et fonctions associées:

```python
def find_path(ag):
    """
    Cherche et (parfois) trouve le chemin augmentant
    le plus court pour le flot dans ag
    Renvoie (b,ap,dphi)
    -b; booleen : indiquant si un chemin augmentant
        pour le flot à été trouvé
    -ap: edge list: le chemin
    -dphi: int : la variation de flot

    Looks for and (sometime) finds the shortest augmenting
    path for the flow in the augmenting graph
    Returns (b,ap,dphi):
    - b: boolean : indicating whether or not
        the augmenting path has been found.
    - ap: edge (int) list : the path itself
    - dphi: int :  the flow variation

    """
```

```python
    bag = q.Queue()
    bag.put(ag.s)
    vus = set()
    vus.add(ag.s)
    partial = [(-1,float('inf')) for _ in range(ag.size)]
    while not bag.empty():
        current = bag.get()
        dphi = partial[current][1]
        aretes = ag.edges(current)
        for y,w in aretes:
            if y not in vus:
                vus.add(y)
                bag.put(y)
                partial[y] = (current,min(w,dphi))
    if partial[ag.t][0] != -1 :
        return (True,rebuild_path(partial,ag.s,ag.t),partial[ag.t][1])
    else:
        return (False, None, 0)
```

Code - Edmond-Karp et fonctions associées:

```
def edmond_karp(g, sinks):
    ag = init_ag(g,sinks)
    (found,ap,dphi) = find_path(ag)
    f = init_flow(ag)
    #show_graph_with_labels(np.matrix(ag.matrix),rho)
    while found:
        update_flow(f,ap,dphi)
        update_ag(ag,ap,dphi)

        (found,ap,dphi) = find_path(ag)
    return f
```

Code - Graphe de capacité

Où DATA est un tableau contenant N triplets (d, f, c)

- $d, f \in V$ les extremités d'une arète
- $c \in \mathbb{N}$ sa largeur.

Et Vertices un set contenant les somemts de G.

```python
def main():
    """
    rho et sigma créent une bijection entre [1;N] et V
    qui ne sont pas des entiers consecutifs (s,M,D,B)

    rho and sigma create a bijection between [1;N] and V as
    these are not consecutive integers.
    """
    g = graph(N)
    global sigma
    sigma = {}
    i = 0
    for v in Vertices:
        sigma[v] = i
        i+=1
    for i in range(len(DATA)):
        x,y,w = DATA[i]
        rx = sigma[x]
        ry = sigma[y]
        g.add_edge(rx,ry,w)
        g.add_edge(ry,rx,w)
    g.s = sigma["s"]
    g.t = None
    global rho
    rho = {v:k for (k,v) in sigma.items()}
```

Code - Affichage des graphes

Avec Positions un tableau associant à chaque noeud une position sur l'image.

```
def find(i, pos):
    if i not in rho:
        return (0,i)
    else:
        i = rho[i]
        for j in range(len(pos)):
            if i == pos[j][0]:
                return (pos[j][1], -pos[j][2])
        return (0,i)

def show_graph_wo_sink(adjacency_matrix, mylabels,path = [],w = 0):

    G = nx.from_numpy_array(adjacency_matrix, parallel_edges=True, create_using=nx.Graph)
    N = len(adjacency_matrix)
    ly= {i:find(i,Positions) for i in range(N)}
    path_edges = {(path[i-1],path[i]) for i in range(1,len(path))}
    couleurs = ['k' if ((u,v) not in path_edges) and ((v,u) not in path_edges) else "red" for u,v in G.edges()]

    for u,v in path_edges:
        G[u][v]['weight'] = w
    poids = [G[u][v]['weight'] if G[u][v]['weight']!= float('inf') else 8 for u,v in G.edges()]
    nx.draw_networkx_edges(G,pos=ly)
    nx.draw_networkx(G, node_size=300, labels=mylabels, with_labels=True,
                     pos=ly, edge_color = couleurs,node_color = 'lightgrey', width = poids )
    plt.show()
```

Code - Affichage des graphes

```
def show_graph_w_sink(adjacency_matrix, mylabels,path = [],w = 0):

    G = nx.from_numpy_array(adjacency_matrix, parallel_edges=True, create_using=nx.Graph)
    N = len(adjacency_matrix)
    ly= {i:find(i,Positions) for i in range(N)}
    ly[N-1] = (40,-418)
    mylabels[N-1] = "t"
    path_edges = {(path[i-1],path[i]) for i in range(1,len(path))}
    couleurs = ['k' if u!= N-1 and v!= N-1 and ((u,v) not in path_edges)
                and ((v,u) not in path_edges) else "red" for u,v in G.edges()]

    for u,v in path_edges:
        G[u][v]['weight'] = w
    poids = [G[u][v]['weight'] if G[u][v]['weight']!= float('inf') else 8 for u,v in G.edges()]
    nx.draw_networkx_edges(G,pos=ly)
    nx.draw_networkx(G, node_size=300, labels=mylabels, with_labels=True,
                     pos=ly, edge_color = couleurs,node_color = 'lightgrey', width = poids )
    plt.show()
```