Report homework 1
Network Dynamics and Learning

Matteo Merlo s287576

November 14, 2021

0 Introduction

This is the report of the first homework of the course Network Dynamics and Learning 2021/2022.
Topic: Connectivity and flows, Network flows optimization

1 Exercise 1

1.1 Point 1.a

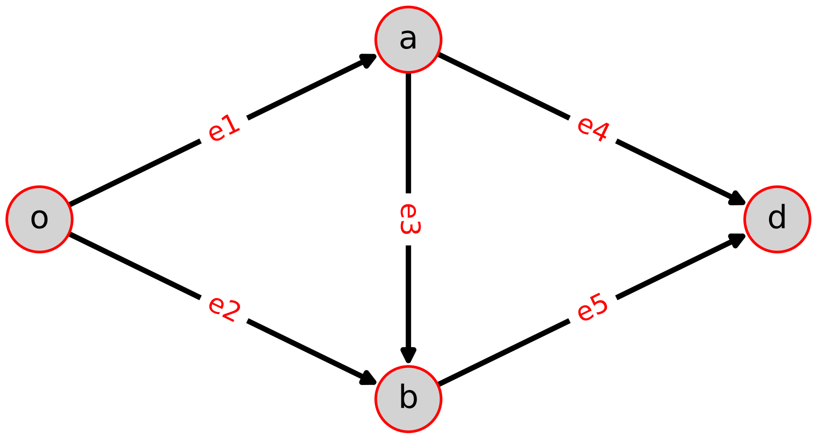


Figure 1: Graph exercise 1

In the graph in *Figure 1* I can assume that capacity $c \in N$. For no feasible unitary flow I have to remove at least s unit of capacity from the graph. This quantity is equal to min-cut capacity.

1.2 Point 1.b

Some capacities are added to the graph, so that the capacities on the edges C = (C_{e_1}, C_{e_2}, C_{e_3}, C_{e_4}, C_{e_5}) = (3, 2, 2, 3, 2).
The first step consists of determining all the possible cuts on the studied network:

1. U_1= {o}
2. U_2= {o,a}
3. U_3 = {o, b}

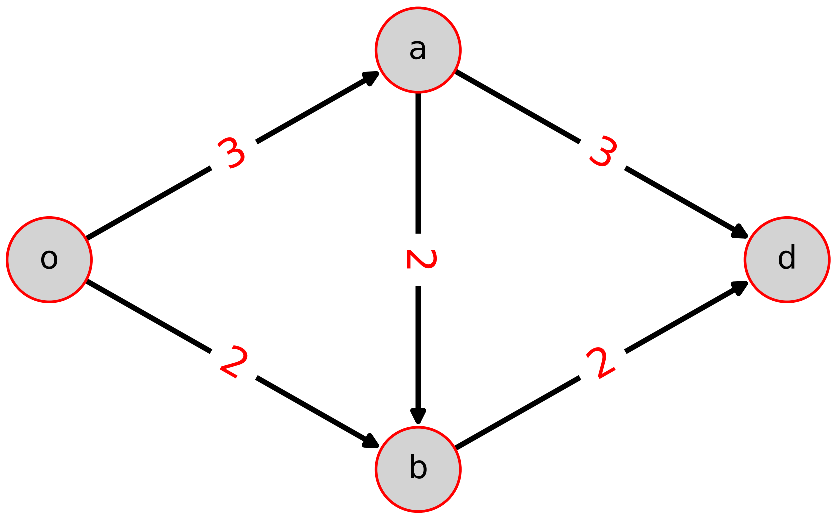


Figure 2: Weighted graph exercise 1

4. U_4 = {o, a, b}

Then, as second step, the total capacity on all these cuts must be determined:

1. $C_{U_1} = C_{e_1} + C_{e_2} = 3 + 2 = 5$
2. $C_{U_2} = C_{e_2} + C_{e_3} + C_{e_4} = 2 + 2 + 3 = 7$
3. $C_{U_3} = C_{e_1} + C_{e_5} = 3 + 2 = 5$
4. $C_{U_4} = C_{e_4} + C_{e_5} = 3 + 2 = 5$

As a result the min cut capacities C_{U_1}, C_{U_3} and C_{U_4} are the bottlenecks. Finally, by applying the already mentioned min-cut max-flow theorem, the maximum flow that can be sent from o to d $\tau_{o,d} = C_{U_1} = C_{U_3} = C_{U_4} = 5$. All the flow that pass through edge e_1, will pass also through edge e_4. The same is for edges e_2 and e_5. No flow at maximum-flow condition pass through the edge e_3. Since I would add 1 capacity this is not enough to change the min-cut in all bottleneck cited previously. So for example if I add the capacity on the first edge e_1, no more extra flow pass through edges e_3 and e_4. And there will be no change on the min-cut. No solution to increase the min cut are possible for one capacity added.

1.3 Point 1.c

In the graph, differently from the point before, if I add 2 more capacities the maximal throughput will change. The new maximal throughput obtained is 6. The added capacity can be distributed in 3 different way among the edges:

1. $(C_{e_1}, C_{e_2}, C_{e_3}, C_{e_4}, C_{e_5}) + (1, 0, 0, 1, 0) \Rightarrow (4, 2, 2, 4, 2)$
2. $(C_{e_1}, C_{e_2}, C_{e_3}, C_{e_4}, C_{e_5}) + (1, 0, 0, 0, 1) \Rightarrow (4, 2, 2, 3, 3)$
3. $(C_{e_1}, C_{e_2}, C_{e_3}, C_{e_4}, C_{e_5}) + (0, 1, 0, 0, 1) \Rightarrow (3, 3, 2, 3, 3)$

1.4 Point 1.d

In this case I add 4 more capacities in the graph. The new maximal throughput will be 7. The added capacity can be distributed in 6 different way among the edges:

1. $(\mathrm{C}_{e_1}, \mathrm{C}_{e_2}, \mathrm{C}_{e_3}, \mathrm{C}_{e_4}, \mathrm{C}_{e_5}) + (2, 0, 0, 2, 0) \Rightarrow (5, 2, 2, 5, 2)$
2. $(\mathrm{C}_{e_1}, \mathrm{C}_{e_2}, \mathrm{C}_{e_3}, \mathrm{C}_{e_4}, \mathrm{C}_{e_5}) + (2, 0, 0, 1, 1) \Rightarrow (5, 2, 2, 4, 3)$
3. $(\mathrm{C}_{e_1}, \mathrm{C}_{e_2}, \mathrm{C}_{e_3}, \mathrm{C}_{e_4}, \mathrm{C}_{e_5}) + (2, 0, 0, 0, 2) \Rightarrow (5, 2, 2, 3, 4)$
4. $(\mathrm{C}_{e_1}, \mathrm{C}_{e_2}, \mathrm{C}_{e_3}, \mathrm{C}_{e_4}, \mathrm{C}_{e_5}) + (1, 1, 0, 1, 1) \Rightarrow (4, 3, 2, 4, 3)$
5. $(\mathrm{C}_{e_1}, \mathrm{C}_{e_2}, \mathrm{C}_{e_3}, \mathrm{C}_{e_4}, \mathrm{C}_{e_5}) + (1, 1, 0, 0, 2) \Rightarrow (4, 3, 2, 3, 4)$
6. $(\mathrm{C}_{e_1}, \mathrm{C}_{e_2}, \mathrm{C}_{e_3}, \mathrm{C}_{e_4}, \mathrm{C}_{e_5}) + (0, 2, 0, 0, 2) \Rightarrow (3, 4, 2, 3, 4)$

In the code I created all possible combination where to put the capacity, then i remove the repeated combinations. Then I add and try iteratively the new mincut. If there is an increase in the maximal throughput, the best combinations are saved.

2 Exercise 2

2.1 Point 2.a

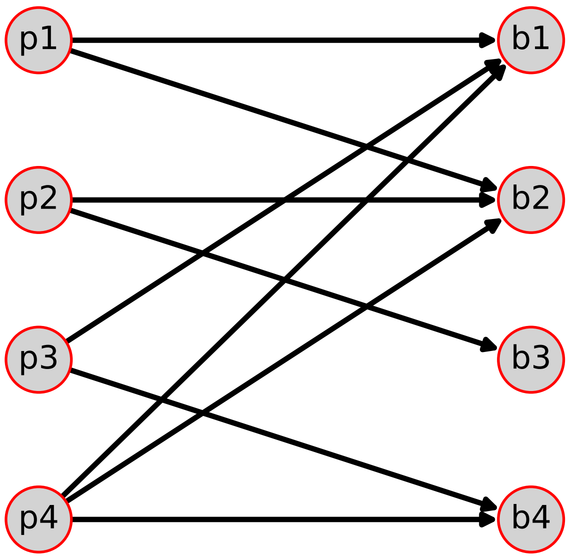


Figure 3: Bipartite graph exercise 2

The bipartite graph of exercise 2 is showed in *Figure 3* The graph show the relation between a set of people and a set of book. The relations are:

- $p_1 \Rightarrow (\ b_1,\ b_2\ )$
- $p_2 \Rightarrow (\ b_2,\ b_3\ )$
- $p_3 \Rightarrow (\ b_1,\ b_4\ )$
- $p_4 \Rightarrow (\ b_1,\ b_2,\ b_4\ )$

2.2 Point 2.b

Red edges in *Figure 4* are a possible perfect matching of graph of exercise 2. A matching is perfect if all nodes are matched. A perfect matching is also a minimum-size link cover and it is explained by Hall's theorem

Hall's Theorem: for a simple bipartite graph $G = (\ V,\ E\ )$ and $V_0 \subset V$, there exist V_0-perfect matching in G if and only if

$$|U| < |N_U|, \forall U \subseteq V_0$$

where

$$N_U = \sum_{i \in U}$$

is the neighborhood of U in G.

There is an analogy between perfect matching and maximal flow on this auxiliary network G. In particular, a V_0-perfect matching on G exists if and only if it there exists a flow with throughput $|V_0|$

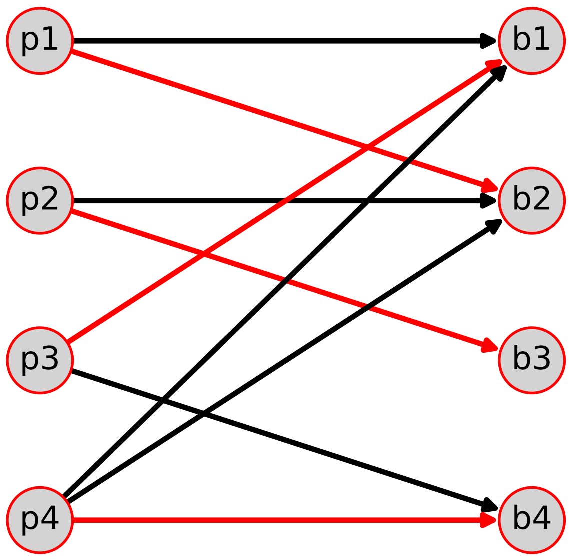


Figure 4: An example of perfect matching in the bipartite graph

on the network G. I can exploit Ford-Fulkerson algorithm to find the maximum flow that can be sent in. The associated V_0-perfect matching can be found by selecting all the edges such that flow f is equal 1. The total Max-flow found is 4 as expected.

2.3 Point 2.c

In the Library there are multiple copies of books, specifically the distribution of the number of copies is respectively *(2; 3; 2; 2)* . The weight are showed in *Figure 5* Since the only constraint is that each person can not take more copies of the same book, the problem could be solved as a Maximum-flow problem. The bipartite graph can be seen as the flux that flows from a source s to a destination d as depicted in *Figure 5*. On each edge the flow as explained in the previous point should be equal 1. The resulted max-flow is 8. And with this configuration not all edge are used at maximum possible flow, that means not all customers are satisfied and not all books are sold.

2.4 Point 2.d

Since not all customers are satisfied and not all books are sold, I implemented an algorithm to find out which edges are not optimal. I studied the inflow and the outflow of each node: a node is not optimal if the net flow from the difference between inflow and outflow is different from 0. The node that are not optimal are book b_1 and book b_3 . An extra edge should reach node b_3 from people p_1 or p_3 or p_4 , removing one from node b_1 to people p_1 or p_3 or p_4 . So with this new configuration, with the current constrains, the maximum possible flow will be 9.

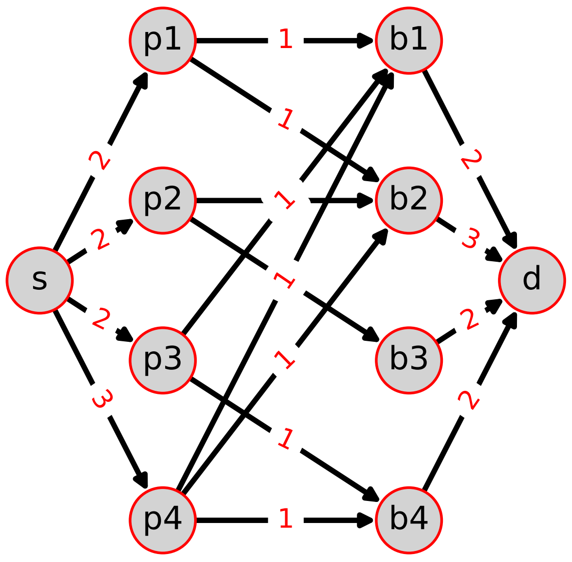


Figure 5: Weighted bipartite graph

3 Exercise 3

NOTE : the precision of results reported in this exercise is two decimals.

3.1 Point 3.a

The map of the city of Los Angeles is represent by the minimalist graph in *Figure 6* The graph is created by a node-edge incidence matrix read from *traffic.mat*. The shortest path, considering only travel time as the only cost on the edge, passes through nodes: [1, 2, 3, 9, 13, 17]. The path is highlighted in red in the *Figure 7*

3.2 Point 3.b

The maximum flow is 22448.

3.3 Point 3.c

For the external inflow I multiply node-link incidence matrix B with flow on each edge. The flow is get from *flow.mat*. The flow vector obtained respectively for each node is v = [16806, 8570, 19448, 4957, -746, 4768, 413, -2, -5671, 1169, -5, -7131, -380, -7412, -7810, -3430, -23544]. So the external inflow incoming from first node is 16806.

3.4 Point 3.d

Considering the flow from previous point, I would calculate the social optimum flow of the graph f^* from node 1 to 17. In the code I use cvxpy library to minimize cost the function of flow subject to the

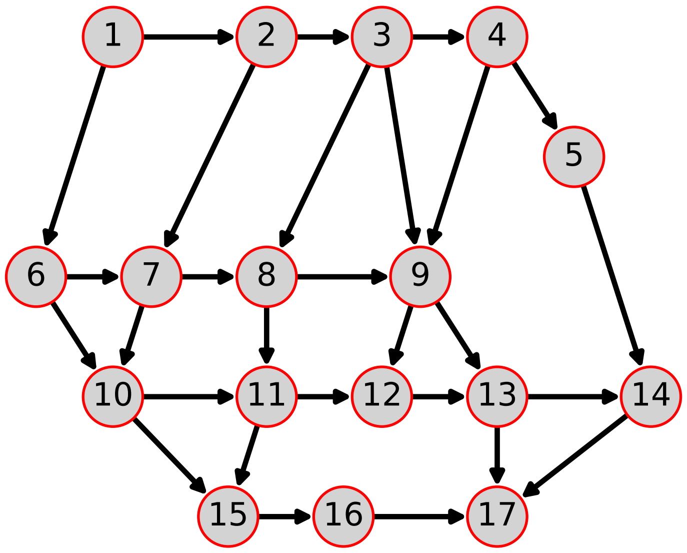


Figure 6: Map of Los Angeles in graph form

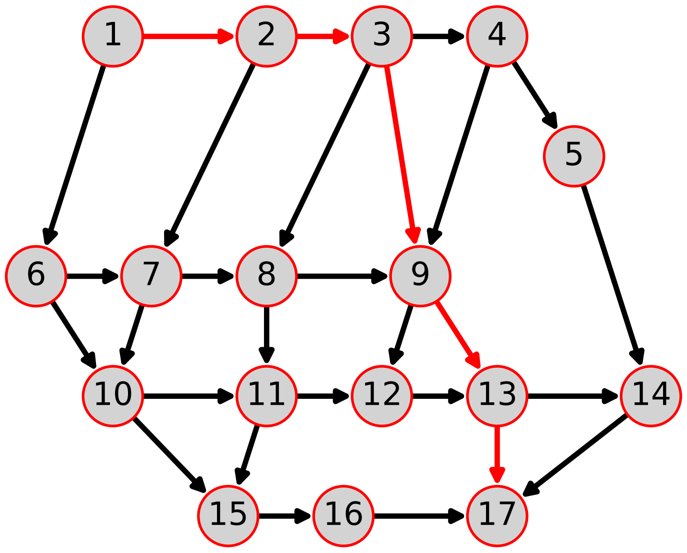


Figure 7: Shortest path in Los Angeles

flow constraints:

$$\sum_{e\in\varepsilon}\psi_e(f_e)=\sum_{e\in\varepsilon}f_e d_e(f_e)=\sum_{e\in\varepsilon}\frac{f_e l_e}{1-f_e/C_e}=\sum_{e\in\varepsilon}\left(\frac{C_e l_e}{1-f_e/C_e}-l_e C_e\right)$$

The social optimum vector f^* is: [6642.2, 6058.94, 3132.33, 3132.33, 10163.8, 4638.32, 3006.34, 2542.63, 3131.54, 583.26, 0.01, 2926.6, 0.00, 3132.33, 5525.48, 2854.27, 4886.45, 2215.24, 463.72, 2337.69, 3317.99, 5655.68, 2373.11, 0.00, 6414.12, 5505.43, 4886.45 4886.45]

The social optimum cost resulting from sum of all cost of the edge is 25943

The idea behind of system optimum is to model traffic flows in the network resulting from a centralized optimization.

3.5 Point 3.e

The wardrop equilibrium instead model the traffic as the outcome of selfish behaviors of users. Such behavior is modeled by assuming that users choose their route so as to minimize the delay they experience along it.

I would calculate the wardrop equilibrium flow of the graph f_0 from node 1 to 17. As previous point cvxpy library is used to minimize cost the function of flow subject to the flow constraints:

$$\sum_{e\in\varepsilon}\int_0^{f_e}d_e(s)ds=\sum_{e\in\varepsilon}-C_e l_e\log\left(1-\frac{f}{C_e}\right)$$

The result wardrop equilibrium flow f^0 is: [6715.65, 6715.65, 2367.41, 2367.41, 10090.35, 4645.39, 2803.84, 2283.56, 3418.48, 0.00, 176.83, 4171.41, 0.00, 2367.41, 5444.96, 2353.17, 4933.34, 1841.55, 697.11, 3036.49, 3050.28, 6086.77, 2586.51, 0.00, 6918.74, 4953.92, 4933.34, 4933.34]

Then I compute the wardrop equilibrium cost considering for each edge the relative delay:

$$\sum_{e\in\varepsilon}f_e^0 d_e(f_e^0)$$

where f_e is the flow at wardrop and delay d_e on edge e

The wardrop cost is 26292

So then I can calculate the Price of Anarchy:

$$PoA=\frac{\sum_{e\in\varepsilon}f_e^0 d_e(f_e^0)}{min\sum_{e\in\varepsilon}f_e^* d_e(f_e^*)}=\frac{26292}{25943}=1.0134$$

The PoA is not exactly equal one. This is why a uncoordinated selfish behaviour makes that every user uses a path with minimal delay. PoA equal one means that in their selfishness the drivers start to cooperate between them.

3.6 Point 3.f

In the graph tolls are introduce in such a way to align user optimum flows with social optimum flows and reduce the inefficiency due to uncoordinated behaviour of the agents.

The idea of marginal tolls is that if a toll equal to $f_e^* {d_e}'(f_e^*)$ is added to each link e (where f^* denotes the social optimum flow distribution), then selfish users should pay not only for their delay, but also for the cost that make other users pay due to their choice. I Compute the Wardrop equilibrium with the link delays $d_e(f_e)$ replaced by $\omega_e = f_e^* {d_e}'(f_e^*)$.

$$\omega_e = f_e^* {d_e}'(f_e^*) = f_e^*\frac{C_e l_e}{(C_e-f_e^*)^2}$$

where (f_e^*) is the flow at social optimum

So the delay on edge e is given by

$$d_e(f_e)+\omega_e$$

The new cost function is:

$$\sum_{e \in \varepsilon} \int_0^{f_e} d_e(s) + \omega_e ds = \sum_{e \in \varepsilon} \left(f\omega_e - C_e l_e \log \left(1 - \frac{f}{C_e} \right) \right)$$

The new wardrop equilibrium (f^*_ω) vector is: [6642.97, 6059.08, 3132.47, 3132.47, 10163.03, 4638.26, 3006.33, 2542.34, 3131.49, 583.9, 0.00, 2926.6, 0.00, 3132.47, 5524.77, 2854.23, 4886.37, 2215.83, 463.99, 2337.45, 3318.22, 5655.67, 2373.04, 0.00, 6414.12, 5505.51, 4886.37, 4886.37]

The new wardrop equilibrium cost is 25943. The new PoA is now equal almost 1, which means that the wardrop is really near to ideal the system optimum.

3.7 Point 3.g

The cost be the total additional delay compared to the total delay in free flow be given by:

$$c_e(f_e) = f_e(d_e(f_e) - l_e) = f_e d_e(f_e) - f_e l_e = \frac{C_e l_e}{1 - f_e/C_e} - l_e C_e - f_e l_e$$

The new system optimum $f^{(\omega^*)}$ is: [6653.3, 5774.66, 3419.72, 3419.71, 10152.7, 4642.78, 3105.84, 2662.18, 3009.08, 878.63, 0.01, 2354.94, 0.01, 3419.71, 5509.92, 3043.69, 4881.81, 2415.57, 443.66, 2008.05, 3487.35, 5495.4, 2203.78, 0.00, 6300.7, 5623.49, 4881.81, 4881.81].
The system optimum cost is 15095.
The new wardrop equlibrium with tolls $f^{(\omega^0)}$ constructed with f^* is: [6653.3, 5774.66, 3419.72, 3419.71, 10152.7, 4642.78, 3105.84, 2662.18, 3009.08, 878.63, 0.01, 2354.94, 0.01, 3419.71, 5509.92, 3043.69, 4881.81, 2415.57, 443.66, 2008.05, 3487.35, 5495.4, 2203.78, 0.00, 6300.7, 5623.49, 4881.81, 4881.81].
Then I compute the wardrop equilibrium cost in the same way as point E, the result is 15095. As the previous point the PoA is really near one, to the ideal system.

4 Collaborations

The results, especially for exercise 3, are compared with:

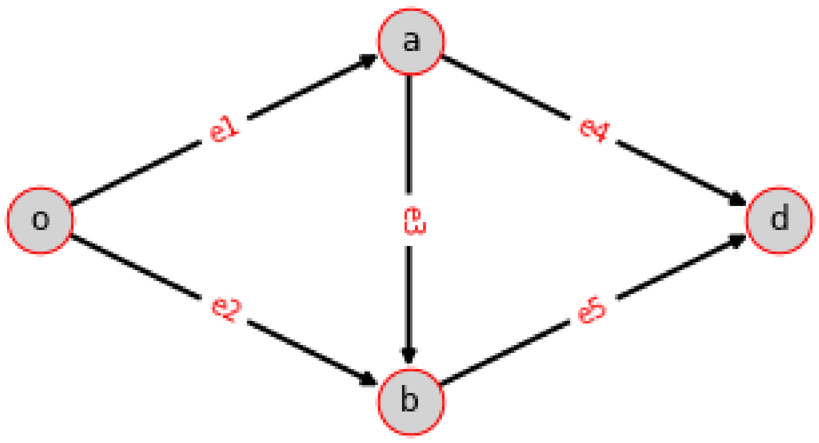
- Matteo Giardino
- Andrea Tampellini
- Dario Padovano

Homework1.1

November 14, 2021

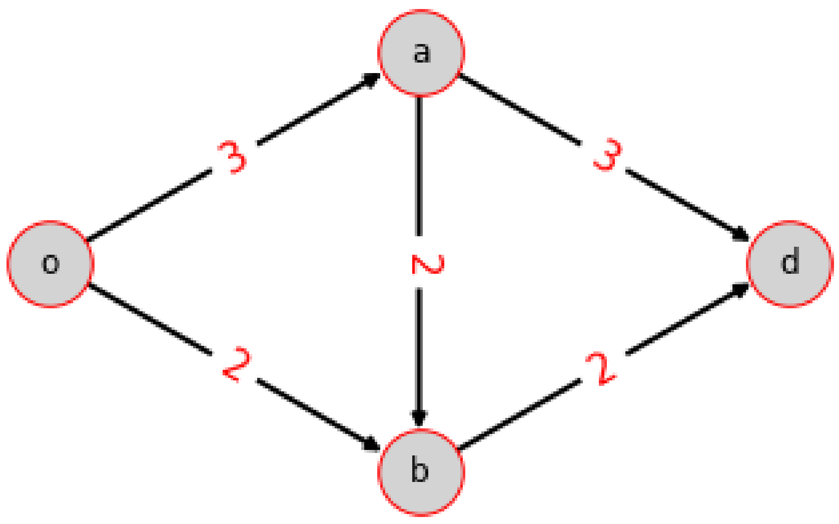
1 Homework 1 - Exercise 1

[9]:
```python
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from networkx.algorithms.flow import edmonds_karp
%matplotlib inline

# creation graph
G = nx.DiGraph()
G.add_edges_from([("o","a"), ("o","b"), ("a","b"), ("a","d"), ("b","d")])

# draw Graph
fig, ax = plt.subplots(figsize=(6,3))

pos = {"o":[-1,0], "a":[0,1], "b":[0,-1], "d":[1,0]}
labels = {("o","a"): "e1", ("o","b"): "e2", ("a","b"): "e3", ("a","d"): "e4",␣
↪("b","d"): "e5"}

nx.draw_networkx_edge_labels(G,  pos, edge_labels = labels, font_size = 10,␣
↪font_color='red', ax = ax)

nx.draw(G, pos, node_size = 600, font_size=12, node_color='lightgray',␣
↪with_labels=True, width=2, edge_color = 'black', edgecolors='red', ax=ax)

plt.savefig("plot1.1.svg", format="svg")
```



B) Where should 1 unit of additional capacity be allocated in order to maximize the feasible throughput from o to d

```
[10]: # add capacities
G["o"]["a"]['capacity'] = 3
G["o"]["b"]['capacity'] = 2
G["a"]["b"]['capacity'] = 2
G["a"]["d"]['capacity'] = 3
G["b"]["d"]['capacity'] = 2

#draw Graph
fig, ax = plt.subplots(figsize=(6,3.5))

nx.draw_networkx_edge_labels(G,pos,edge_labels={("o","a"): '3', ("o","b"): '2',␣
 ↪("a","b"): '2', ("a","d"): '3', ("b","d"):'2'}, font_size = 15,␣
 ↪font_color='red', ax = ax)
nx.draw(G, pos, node_size = 1000, font_size=12, node_color='lightgray',␣
 ↪with_labels=True, width=2, edge_color = 'black', edgecolors='red', ax=ax)
plt.savefig("plot1.2.svg", format="svg")
```



C) Where should 2 units of additional capacity be allocated in order to maximize the feasible throughput from o to d? Compute all the optimal capacity allocations for this case and the optimal throughput.

[16]:
```
edges = nx.get_edge_attributes(G, "capacity")
print("list of edges: ", edges)

#compute all possible combinationon how to distribute the capacities among all␣
↪edges
allCombination = list()

for a in range(len(edges)):
    for b in range(len(edges)):
        tmp = np.zeros(len(edges), dtype = "int32")
        tmp[a] += 1
        tmp[b] += 1
        allCombination.append(tmp)

#remove duplicates
allCombination = np.unique(allCombination, axis= 0 )

optimalCombinationsList = list()
maxFlowFound = 0

# current max-flow of the Graph
flow_value, flow_dict  = nx.algorithms.flow.maximum_flow(G,"o","d")

for combination in allCombination:
    for e_dict, e_value in edges.items():
        for i in range(len(edges)):

            # add the capacities to all edges
            pos = list(edges.keys()).index(e_dict)
            G[e_dict[0]][e_dict[1]]['capacity'] = int(e_value) +␣
↪combination[pos]

            # calculate new max-flow of graph with modified capacities
            new_flow_value, new_flow_dict  = nx.algorithms.flow.
↪maximum_flow(G,"o","d")

    #update list of best combination
    if(maxFlowFound <= new_flow_value and new_flow_value > flow_value):
        maxFlowFound = new_flow_value
        optimalCombinationsList.append(combination)

    G[e_dict[0]][e_dict[1]]['capacity'] = int(e_value)

print(maxFlowFound)

optimalCombinationsList
```

```
list of edges:  {('o', 'a'): 7, ('o', 'b'): 2, ('a', 'b'): 2, ('a', 'd'): 3,
('b', 'd'): 2}
7
```

[16]:
```
[array([0, 0, 0, 0, 2]), array([0, 0, 0, 1, 1]), array([0, 0, 0, 2, 0])]
```

D) Where should 4 units of additional capacity be allocated in order to maximize the feasible throughput from o to d? Compute all the optimal capacity allocations for this case. Among the optimal allocations, select the allocation that maximizes the sum of the cut capacities.

[14]:
```
#compute all possible combinationon how to distribute the capacities among all␣
↪edges
allCombination = list()

for a in range(len(edges)):
    for b in range(len(edges)):
        for c in range(len(edges)):
            for d in range(len(edges)):
                tmp = np.zeros(len(edges), dtype = "int32")
                tmp[a] += 1
                tmp[b] += 1
                tmp[c] += 1
                tmp[d] += 1
                allCombination.append(tmp)

#remove duplicates
```

```
allCombination = np.unique(allCombination, axis= 0 )
len(allCombination)
```

[14]: 70

[15]:
```
edges = nx.get_edge_attributes(G, "capacity")
print("list of edges: ", edges)

optimalCombinationsList = list()
maxFlowFound = 0

# current max-flow of the Graph
flow_value, flow_dict  = nx.algorithms.flow.maximum_flow(G,"o","d")

for combination in allCombination:
    for e_dict, e_value in edges.items():
        for i in range(len(edges)):

            # add the capacities to all edges
            pos = list(edges.keys()).index(e_dict)
            G[e_dict[0]][e_dict[1]]['capacity'] = int(e_value) +␣
 ↪combination[pos]

            # calculate new max-flow of graph with modified capacities
            new_flow_value, new_flow_dict  = nx.algorithms.flow.
 ↪maximum_flow(G,"o","d")

    #update list of best combination on best Max-Flow
    if(maxFlowFound <= new_flow_value and new_flow_value > flow_value and␣
 ↪new_flow_value == 7):
        maxFlowFound = new_flow_value
        optimalCombinationsList.append(combination)

    G[e_dict[0]][e_dict[1]]['capacity'] = int(e_value)

print(maxFlowFound)
optimalCombinationsList
```

```
list of edges:  {('o', 'a'): 3, ('o', 'b'): 2, ('a', 'b'): 2, ('a', 'd'): 3,
('b', 'd'): 2}
7
```

[15]:
```
[array([0, 2, 0, 0, 2]),
 array([1, 1, 0, 0, 2]),
 array([1, 1, 0, 1, 1]),
 array([2, 0, 0, 0, 2]),
 array([2, 0, 0, 1, 1]),
array([2, 0, 0, 2, 0])]
```

```
[ ]:
```

Homework1.2

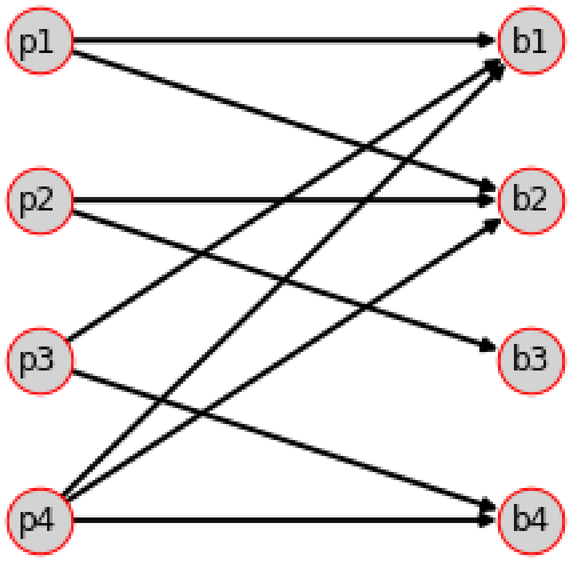
November 14, 2021

1 Homework 1 - Exercise 2

[1]:
```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from networkx.algorithms.flow import edmonds_karp
%matplotlib inline
```

A) Represent the interest pattern by using a simple bipartite graph.

[2]:
```
#G bipartite graph
G = nx.DiGraph()
G.add_edges_from([("p1","b1"), ("p1","b2"), ("p2","b2"), ("p2","b3"),␣
 ↪("p3","b1"), ("p3","b4"), ("p4","b1"), ("p4","b2"), ("p4","b4")])

#Draw the Bipartite Graph
fig, ax = plt.subplots(figsize=(4,4))

pos = {"p1":[0,2], "p2":[0,1], "p3":[0,0], "p4":[0,-1], "b1":[1,2], "b2":[1,1],␣
 ↪"b3":[1,0], "b4":[1,-1]}
nx.draw(G, pos, node_size = 600, font_size=12, node_color='lightgray',␣
 ↪with_labels=True, width=2, edge_color = 'black', edgecolors='red', ax=ax)

plt.savefig("plot2.1.svg", format="svg")
```

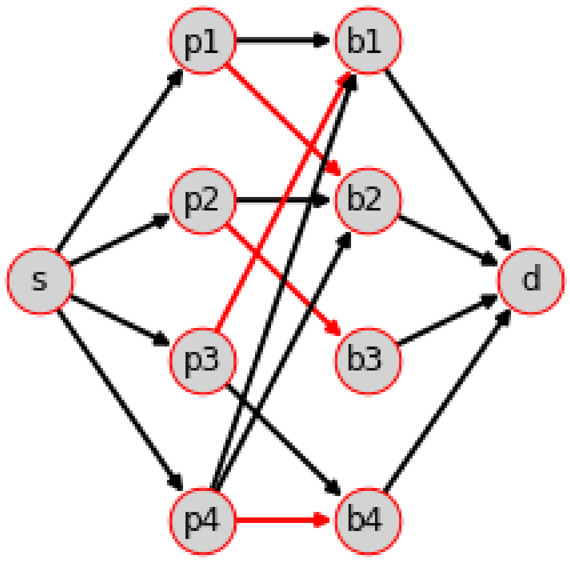


B) Exploit max-flow problems to establish whether there exists a perfect matching that assigns to every person a book of interest. If a perfect matching exists, find at least a perfect matching.

[3]:
```
# add source and destionation to bipartite graph
dG = nx.DiGraph()
dG.add_edges_from([("p1","b1"), ("p1","b2"), ("p2","b2"), ("p2","b3"),␣
 ↪("p3","b1"), ("p3","b4"), ("p4","b1"), ("p4","b2"), ("p4","b4")])
dG.add_edges_from([("s","p1"), ("s","p2"), ("s","p3"), ("s","p4"), ("b1","d"),␣
 ↪("b2", "d"), ("b3","d"), ("b4","d")])

# unitary capacity
dG["p1"]["b1"]['capacity']=1
dG["p1"]["b2"]['capacity']=1
dG["p2"]["b2"]['capacity']=1
dG["p2"]["b3"]['capacity']=1
dG["p3"]["b1"]['capacity']=1
dG["p3"]["b4"]['capacity']=1
dG["p4"]["b1"]['capacity']=1
dG["p4"]["b2"]['capacity']=1
dG["p4"]["b4"]['capacity']=1
dG["s"]["p1"]['capacity']=1
dG["s"]["p2"]['capacity']=1
dG["s"]["p3"]['capacity']=1
dG["s"]["p4"]['capacity']=1
dG["b1"]["d"]['capacity']=1
dG["b2"]["d"]['capacity']=1
dG["b3"]["d"]['capacity']=1
```

```
dG["b4"]["d"]['capacity']=1

# Max-flow of Graph
print("Max flow: ", nx.algorithms.flow.maximum_flow(dG, "s", "d"))

# draw graph
fig, ax = plt.subplots(figsize=(4,4))

edge_colors =␣
 ↪["black","red","black","black","black","red","black","red","black","black","black","black",

pos = {"p1":[0,2], "p2":[0,1], "p3":[0,0], "p4":[0,-1], "b1":[1,2], "b2":[1,1],␣
 ↪"b3":[1,0], "b4":[1,-1], "s":[-1,0.5], "d":[2,0.5]}
nx.draw(dG, pos, node_size = 600, font_size=12, node_color='lightgray',␣
 ↪with_labels=True, width=2, edge_color = edge_colors, edgecolors='red', ax=ax)

plt.savefig("plot2.2.svg", format="svg")
```

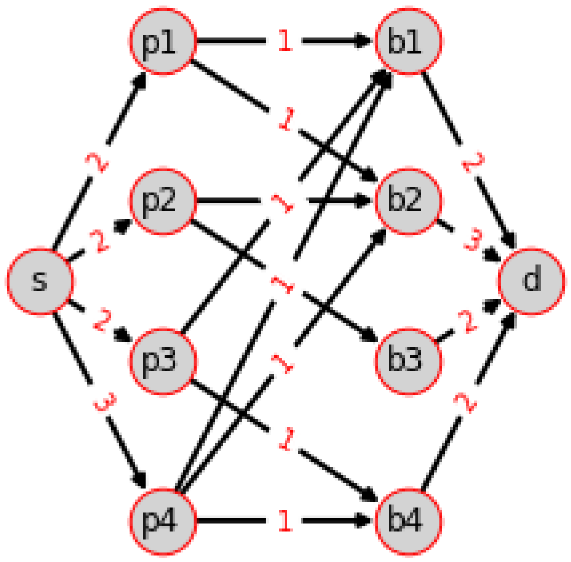
```
Max flow:  (4, {'p1': {'b1': 0, 'b2': 1}, 'b1': {'d': 1}, 'b2': {'d': 1}, 'p2':
{'b2': 0, 'b3': 1}, 'b3': {'d': 1}, 'p3': {'b1': 1, 'b4': 0}, 'b4': {'d': 1},
'p4': {'b1': 0, 'b2': 0, 'b4': 1}, 's': {'p1': 1, 'p2': 1, 'p3': 1, 'p4': 1},
'd': {}})
```



C) Assume now that there are multiple copies of book, specifically the distribution of the number of copies is (2; 3; 2; 2), and there is no constraint on the number of books that each person can take. The only constraint is that each person can not take more copies of the same book. Use the analogy with max-flow problems to establish how many books of interest can be assigned in total.

[4]:
```
# weighted bipartite graph
G_weighted = nx.DiGraph()
G_weighted.add_edges_from([("p1","b1"), ("p1","b2"), ("p2","b2"), ("p2","b3"),␣
↪("p3","b1"), ("p3","b4"), ("p4","b1"), ("p4","b2"), ("p4","b4")])
G_weighted.add_edges_from([("s","p1"), ("s","p2"), ("s","p3"), ("s","p4"),␣
↪("b1","d"), ("b2", "d"), ("b3","d"), ("b4","d")])

# weighted capacity
G_weighted["p1"]["b1"]['capacity']=1
G_weighted["p1"]["b2"]['capacity']=1
G_weighted["p2"]["b2"]['capacity']=1
G_weighted["p2"]["b3"]['capacity']=1
G_weighted["p3"]["b1"]['capacity']=1
G_weighted["p3"]["b4"]['capacity']=1
G_weighted["p4"]["b1"]['capacity']=1
G_weighted["p4"]["b2"]['capacity']=1
G_weighted["p4"]["b4"]['capacity']=1
G_weighted["s"]["p1"]['capacity']=2
G_weighted["s"]["p2"]['capacity']=2
G_weighted["s"]["p3"]['capacity']=2
G_weighted["s"]["p4"]['capacity']=3
G_weighted["b1"]["d"]['capacity']=2
G_weighted["b2"]["d"]['capacity']=3
G_weighted["b3"]["d"]['capacity']=2
G_weighted["b4"]["d"]['capacity']=2

# Max-flow of Graph
print("Max flow: ", nx.algorithms.flow.maximum_flow(G_weighted,"s","d"))

# draw Graph
fig, ax = plt.subplots(figsize=(4,4))

edge_colors =␣
↪["black","black","black","black","black","black","black","black","black","black","black","b

pos = {"p1":[0,2], "p2":[0,1], "p3":[0,0], "p4":[0,-1], "b1":[2,2], "b2":[2,1],␣
↪"b3":[2,0], "b4":[2,-1], "s":[-1,0.5], "d":[3,0.5]}
labels = nx.get_edge_attributes(G_weighted, 'capacity')
nx.draw_networkx_edge_labels(G_weighted,pos,edge_labels=labels, font_size=10,␣
↪font_color='red')
nx.draw(dG, pos, node_size = 600, font_size=12, node_color='lightgray',␣
↪with_labels=True, width=2, edge_color = edge_colors, edgecolors='red', ax=ax)

plt.savefig("plot2.3.svg", format="svg")
```

```
Max flow:  (8, {'p1': {'b1': 0, 'b2': 1}, 'b1': {'d': 2}, 'b2': {'d': 3}, 'p2':
{'b2': 1, 'b3': 1}, 'b3': {'d': 1}, 'p3': {'b1': 1, 'b4': 1}, 'b4': {'d': 2},
'p4': {'b1': 1, 'b2': 1, 'b4': 1}, 's': {'p1': 1, 'p2': 2, 'p3': 2, 'p4': 3},
'd': {}})
```



D) Starting from point (c), suppose that the library can sell a copy of a book and buy a copy of another book. Which books should be sold and bought to maximize the number of assigned books?

[5]:
```python
nodes = G_weighted.nodes
edges = G_weighted.edges

for node_i in nodes:

    inflow = 0
    outflow = 0
    maxPossibleOutflow = 0

    # source node is not considered
    if node_i == "s": continue

    # inflow of destination node is the maximal possible flow of graph
    if node_i == "d":
         for u, v, data in G_weighted.in_edges(node_i, data=True):
            maxPossibleOutflow = sum(data.values()) + maxPossibleOutflow

    # calculate inflow of node_i
    for u, v, data in G_weighted.in_edges(node_i, data=True):
        inflow = sum(data.values()) + inflow
    # calculate outflow of node_i
    for u, v, data in G_weighted.out_edges(node_i, data=True):
        outflow = sum(data.values()) + outflow

    # netflow of node_i
    netflow = inflow - outflow

    # if some nodes has inflow greater or minor than outflow means that it has␣
 ↪not an optimized flow
    if netflow != 0 and node_i != "s" and node_i != "d":
        print(f"Node {node_i} not optimized, netflow is {netflow}")

print("Max possible outflow: ", maxPossibleOutflow)
```

```
Node b1 not optimized, netflow is 1
Node b3 not optimized, netflow is -1
Max possible outflow:  9
```

```
[ ]:
```

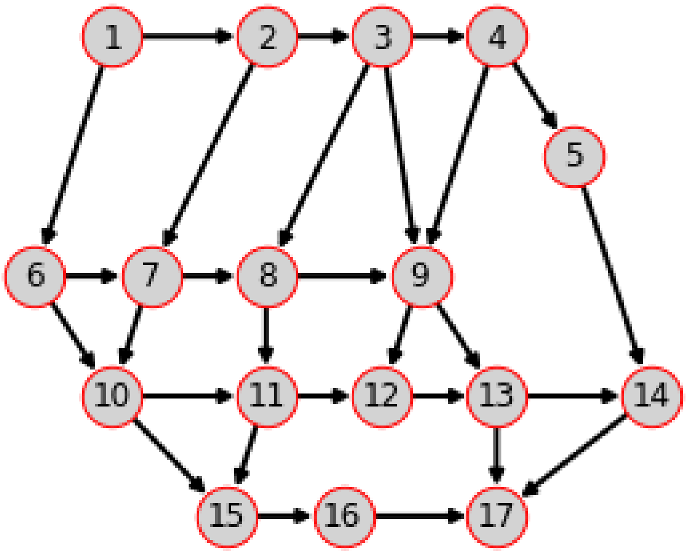
homework1.3

November 14, 2021

1 Homework 1 - Exercise 3

[114]:
```python
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import scipy.io
import scipy
import cvxpy as cp
%matplotlib inline

np.set_printoptions(precision=2, suppress= True)

file = scipy.io.loadmat('capacities.mat')
capacities = file.get('capacities')
capacities = capacities.reshape(28,)

file = scipy.io.loadmat('traveltime.mat')
traveltime = file.get('traveltime')
traveltime = traveltime.reshape(28,)

file = scipy.io.loadmat('flow.mat')
flow = file.get('flow')
flow = flow.reshape(28,)

file = scipy.io.loadmat('traffic.mat')
traffic = file.get('traffic')

# creation of Graph
G = nx.DiGraph()

for c in range(28):
    capac = capacities[c]
    travtime = traveltime[c]
    for r in range(17):
        if traffic[r][c]==1:
            i=r
        if traffic[r][c]==-1:
            j=r
    G.add_edges_from([(i+1,j+1)], capacity=capac, traveltime=travtime)

edges = G.edges()
print(edges)

#label = capacities.astype(str)
#zip_operator = zip(G.edges(), label)
# labels = dict(zip_operator)
# nx.draw_networkx_edge_labels(G,pos,edge_labels = labels, font_color='red')

#draw Graph
fig, ax = plt.subplots(figsize=(5,4))

pos = {1:[-3,1], 2:[-1,1], 3:[0.5,1], 4:[2,1], 5:[3,0], 6:[-4,-1], 7:[-2.5,-1],␣
↪8:[-1,-1], 9:[1,-1],
       10:[-3,-2], 11:[-1,-2], 12:[0.5,-2], 13:[2,-2], 14:[4,-2], 15:[-1.5,-3],␣
↪16:[0,-3], 17:[2,-3]}

nx.draw(G, pos, node_size = 500, font_size=12, node_color='lightgray',␣
↪with_labels=True, width=2, edge_color = 'black', edgecolors='red', ax=ax)

plt.savefig("plot3.1.svg")
```

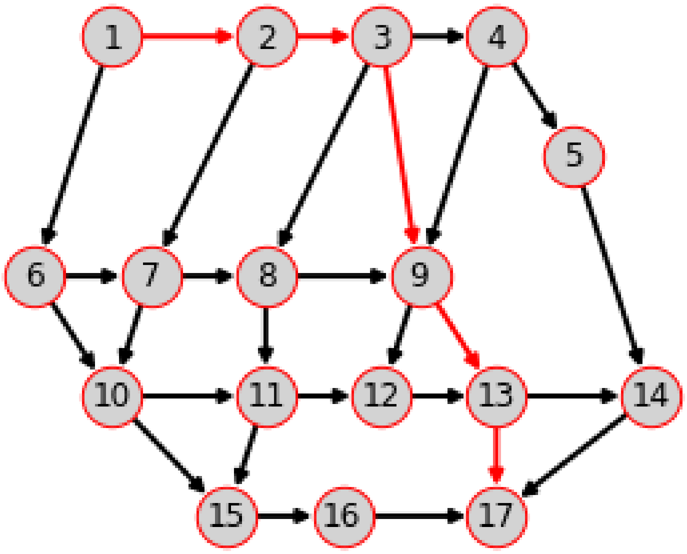
```
[(1, 2), (1, 6), (2, 3), (2, 7), (3, 4), (3, 8), (3, 9), (4, 5), (4, 9), (5,
14), (6, 7), (6, 10), (7, 8), (7, 10), (8, 9), (8, 11), (9, 13), (9, 12), (13,
14), (13, 17), (14, 17), (10, 11), (10, 15), (11, 12), (11, 15), (15, 16), (12,
13), (16, 17)]
```



A) Find the shortest path between node 1 and 17. This is equivalent to the fastest path (path with shortest traveling time) in an empty network.

[115]:
```
print("The shortest path from node 1 to node 17: ", nx.shortest_path(G,␣
 ↪source=1, target=17, weight='traveltime'))

fig, ax = plt.subplots(figsize=(5,4))

edge_colors =␣
 ↪["red","black","red","black","black","black","red","black","black","black",
              ␣
 ↪"black","black","black","black","black","black","red","black","black","red",
               "black","black","black","black","black","black","black", "black"]
nx.draw(G, pos, node_size = 500, font_size=12, node_color='lightgray',␣
 ↪with_labels=True, width=2, edge_color = edge_colors, edgecolors='red', ax=ax)

plt.savefig("plot3.2.svg")

# for path in nx.all_simple_paths(G, source=1, target=17):
#     print(path)
```

```
The shortest path from node 1 to node 17:  [1, 2, 3, 9, 13, 17]
```



[116]:
```
#node-link incidence matrix
B_matrix = traffic
n_edges = B_matrix.shape[1]
```

```
#unitary inflow from node 1 to node 17
nu = np.array([1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, -1]) # exogenous flow vector
l = np.array(traveltime)

# Construct the problem.
f = cp.Variable(n_edges)

objective = cp.Minimize(l.T @ f)
constraints = [B_matrix @ f == nu, f >=0]
prob = cp.Problem(objective, constraints)

# The optimal objective value is returned by `prob.solve()`.
cost_opt = prob.solve()

# The optimal value for f is stored in `f.value`.
print("Optimal f:", f.value)
```

```
Optimal f: [1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0.
 1. 0. 0. 0.]
```

B) Find the maximum flow between node 1 and 17.

[117]:
```
#nx.algorithms.flow.minimum_cut(G,1,17)
print("Maximum flow from node 1 to 17: ", nx.algorithms.flow.
 ↪maximum_flow(G,1,17) )
```

```
Maximum flow from node 1 to 17:  (22448, {1: {2: 8741, 6: 13707}, 2: {3: 8741,
7: 0}, 3: {4: 0, 8: 0, 9: 8741}, 4: {5: 0, 9: 0}, 5: {14: 0}, 6: {7: 4624, 10:
9083}, 7: {8: 4624, 10: 0}, 8: {9: 4624, 11: 0}, 9: {13: 6297, 12: 7068}, 13:
{14: 3835, 17: 10355}, 14: {17: 3835}, 10: {11: 825, 15: 8258}, 11: {12: 825,
15: 0}, 15: {16: 8258}, 12: {13: 7893}, 17: {}, 16: {17: 8258}})
```

C) Given the flow vector in *flow.mat*, compute the external inflow v satisfying $Bf = v$

[118]:
```
external_inflow = B_matrix @ flow
external_inflow
```

[118]:
```
array([ 16806,   8570,  19448,   4957,   -746,   4768,    413,     -2,
        -5671,   1169,     -5,  -7131,   -380,  -7412,  -7810,  -3430,
       -23544], dtype=int32)
```

D) Find the social optimum f^* with respect to the delays on the different links $f_e d_e$, for this minimize cost function

$$\sum_{e \in \varepsilon} \left(\frac{C_e l_e}{1 - f_e / C_e} - l_e C_e \right)$$

[119]:
```
# exogenous inflow vector
nu = np.array([16806,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, -16806])
f = cp.Variable(n_edges)

#cost function
func = cp.multiply(traveltime*capacities, cp.inv_pos(1 - cp.multiply(f,1/
 ↪capacities))) - traveltime*capacities
func = cp.sum(func)

# Construct the problem.
# Minimize cost function
objective = cp.Minimize(func)
constraints = [B_matrix @ f == nu, f >=0, f <= capacities]
prob = cp.Problem(objective, constraints)

# The optimal objective value is returned by `prob.solve()`.
cost_opt = prob.solve()

# The optimal value for f is stored in `f.value`.
opt_flow = f.value
print("Social optimal flow:", opt_flow)
print("Optimal cost:", cost_opt)
```

```
Social optimal flow: [ 6642.2   6058.94  3132.33  3132.33 10163.8   4638.32
3006.34  2542.63
  3131.54   583.26     0.01  2926.6      0.    3132.33  5525.48  2854.27
  4886.45  2215.24   463.72  2337.69  3317.99  5655.68  2373.11     0.
  6414.12  5505.43  4886.45  4886.45]
Optimal cost: 25943.62261121288
```

E) Find the Wardrop equilibrium $f^{(0)}$. For this use cost function:

$$\sum_{e \in \varepsilon} \int_0^{f_e} d_e(s) ds$$

[120]:
```
f = cp.Variable(n_edges)

#cost function
integral = - cp.multiply(traveltime * capacities, cp.log( 1 - (cp.multiply(f, 1/
 ↪capacities) )))
func2 = cp.sum(integral)

#minimize cost function
objective = cp.Minimize(func2)
constraints = [B_matrix @ f == nu, f >=0, f <= capacities]
prob = cp.Problem(objective, constraints)

cost_w = prob.solve()
```

```
print("Wardrop equilibrium flow:", f.value)

war_vect = f.value
```

```
Wardrop equilibrium flow: [ 6715.65  6715.65  2367.41  2367.41 10090.35  4645.39
2803.84  2283.56
  3418.48     0.     176.83  4171.41     0.    2367.41  5444.96  2353.17
  4933.34  1841.55   697.11  3036.49  3050.28  6086.77  2586.51     0.
  6918.74  4953.92  4933.34  4933.34]
```

Compute the cost of wardrop equilibrium with:

$$\sum_{e \in \varepsilon} f_e^0 d_e(f_e^0)$$

and Price of Anarchy $PoA(0) = \frac{wardropcost}{socialoptimumcost}$

[121]:
```
# cost, defined as \sum_e f_e d_e(f_e)
def cost(f):
    tot = []
    for i, value in enumerate(f):
        tot.append(((traveltime[i]*capacities[i]) / (1-(value/
 ↪capacities[i])))-traveltime[i]*capacities[i])
    return sum(tot)

cost_w = cost(war_vect)

print("Wardrop cost:", cost_w)

PoA = cost_w/cost_opt

print("The price of anarchy:", PoA)
```

```
Wardrop cost: 26292.963874629393
The price of anarchy: 1.013465400289377
```

F) Introduce tolls, such that the toll on link e is

$$\omega_e = f_e^* {d_e}'(f_e^*)$$

, where f_e^* is the flow at the system optimum. Now the delay on link e is given by

$$f\{e\}d\ \{e\} + \omega\_\{e\}$$

. compute the new Wardrop equilibrium \$ f^{(w)} \$.

[122]:
```
f = cp.Variable(n_edges)

#compute omega
omega = []
```

```
for i, value in enumerate(opt_flow):
    omega.append(value*((capacities[i]*traveltime[i])/
 ↪((capacities[i]-value)**2)))

func3 = cp.sum(cp.multiply(omega, f) - cp.multiply(capacities*traveltime, cp.
 ↪log(1-(cp.multiply(f, 1/capacities)))))

objective = cp.Minimize(func3)
constraints = [B_matrix @ f == nu, f >=0, f <= capacities]
prob = cp.Problem(objective, constraints)

result = prob.solve()

print("Wardrop equilibrium with tolls:", f.value)
```

```
Wardrop equilibrium with tolls: [ 6642.97  6059.08  3132.47  3132.47 10163.03
4638.26  3006.33  2542.34
  3131.49   583.9      0.    2926.6      0.    3132.47  5524.77  2854.23
  4886.37  2215.83   463.99  2337.45  3318.22  5655.67  2373.04     0.
  6414.12  5505.51  4886.37  4886.37]
```

[123]:
```
war_vect = f.value
#compute cost of wardrop equilibrium
cost_w = cost(war_vect)

print("Wardrop cost:", cost_w)

PoA = cost_w/cost_opt

print("The price of anarchy:", PoA)
```

```
Wardrop cost: 25943.62261534188
The price of anarchy: 1.000000000159153
```

G) Instead of the total delay, let the cost be the total additional delay compared to the total delay in free flow be given by $c_e f_e = f_e(d_e(f_e) - l_e)$ subject to the flow constraints. Compute the system optimum f^* for the costs given.

[124]:
```
f = cp.Variable(n_edges)

#cost function free flow delay
func4 = cp.sum(cp.multiply( cp.multiply(traveltime, capacities), cp.inv_pos(1 -␣
 ↪cp.multiply(f, 1/capacities) ) ) - cp.multiply(traveltime, capacities) - cp.
 ↪multiply(traveltime,f) )

objective = cp.Minimize(func4)
constraints = [B_matrix @ f == nu, f >=0, f <= capacities]
prob = cp.Problem(objective, constraints)
```

```

# The optimal objective value is returned by `prob.solve()`.
cost_opt = prob.solve()

# The optimal value for f is stored in `f.value`.
opt_flow = f.value
print("Social optimal flow:", opt_flow)
print("Optimal cost:", cost_opt)
```

```
Social optimal flow: [ 6653.3    5774.66   3419.72   3419.71 10152.7    4642.78
3105.84   2662.18
  3009.08    878.63      0.01   2354.94      0.01   3419.71   5509.92   3043.69
  4881.81   2415.57    443.66   2008.05   3487.35   5495.4    2203.78      0.
  6300.7    5623.49   4881.81   4881.81]
Optimal cost: 15095.51352460787
```

Construct tolls ω_e^* such that the new Wardrop equilibrium with the constructed tolls $f^{(\omega^*)}$ coincides with f^*

[125]:
```
w = cp.Variable(n_edges)
nu = np.array([16806, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16806])

# cost function with optimal flow
integral = - cp.multiply(capacities*traveltime, cp.log(1-(cp.multiply(opt_flow,␣
↪1/capacities)))) - cp.multiply(opt_flow,traveltime) + cp.multiply(w, opt_flow)
func5 = cp.sum(integral)

objective = cp.Minimize(func5)
constraints = [B_matrix @ w == nu, w>=0]
prob = cp.Problem(objective, constraints)

result_w = prob.solve()

print("Constructed tolls:", w.value)

constr_tolls = w.value
```

```
Constructed tolls: [16806.     0.     0.     0.     0.     0. 16806.     0.
0. 16806.
     0.     0.     0.     0.     0.     0.     0.     0. 16806.     0.
     0.     0.     0. 16806.     0.     0. 16806. 16806.]
```

Compute the new Wardrop equilibrium with the constructed tolls $f^{(\omega^*)}$ to verify your result.

[128]:
```
f = cp.Variable(n_edges)
nu = np.array([16806, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16806])

func = - cp.multiply(capacities*traveltime, cp.log(1-(cp.multiply(f, 1/
↪capacities)))) - cp.multiply(f,traveltime) + cp.multiply(constr_tolls, f)
```

```
objective = cp.Minimize(cp.sum(func))
constraints = [B_matrix @ f == nu, f >=0, f <= c, f==opt_flow]
prob = cp.Problem(objective, constraints)

result_w = prob.solve()

print("Wardrop equilibrium:", f.value)
```

```
Wardrop equilibrium: [ 6653.3   5774.66  3419.72  3419.71 10152.7   4642.78
3105.84  2662.18
  3009.08   878.63     0.01  2354.94     0.01  3419.71  5509.92  3043.69
  4881.81  2415.57   443.66  2008.05  3487.35  5495.4   2203.78     0.
  6300.7   5623.49  4881.81  4881.81]
```

Compute the cost of wardrop equilibrium with:

$$\sum_{e \in \varepsilon} f_e^0 d_e(f_e^0)$$

and Price of Anarchy

$$(PoA(0) = \frac{wardropcost}{socialoptimumcost}$$

```
[129]: def cost2(f):
    tot = []
    for i, value in enumerate(f):
        tot.append(((l[i]*c[i]) / (1-(value/c[i])))-l[i]*c[i]-l[i]*value)
    return sum(tot)

war_vect = f.value
cost_w = cost2(war_vect)

print("Wardrop cost:", cost_w)

PoA = cost_w/cost_opt

print("The price of anarchy:", PoA)
```

```
Wardrop cost: 15095.513347476217
The price of anarchy: 0.9999999882659406
```

```
[ ]:
```