Problem D – unum modo platea

Gruppe: Alexander Lontke und Nico Hasler

Problemstellung

In the Country of Via, the cities are connected by roads that can be used in both directions. However, this has been the cause of many accidents since the lanes are not separated: The drivers frequently look at their smartphones while driving, causing them to collide with the oncoming traffic. To alleviate the problem, the politicians of Via came up with the magnificent idea to have one-way roads only, i.e., the existing roads arealtered such that each can be only used in one of two possible directions. They call this “one-way-ification”. The mayors do not want too many one-way roads to lead to their cities because this can cause traffic jam within the city: they demand that the smallest integer d be found such that there is a ‘one-way-ification’ in which for every city, the number of one-way roads leading to it is at most d.

Modellieren als Graphenproblem

Städte	→	Knoten
Straßen im Problem	→	Ungerichtete Kanten
Straßen in der Lösung	→	Gerichtete Kanten

Der Input

1.Zeile: Integer n ($1 \leq n \leq 500$), wobei n die Anzahl der Städte ist

2.Zeile: Integer m ($0 \leq m \leq 2{,}5*10^3$), wobei m die Anzahl der Straßen ist

2+m.Zeilen: Zwei Integers a und b ($1 \leq a,b \leq n$, $a \neq b$), wobei die Zeile die Straße zwischen Stadt a und b beschreibt

(höchstens eine Straße zwischen zwei Städten)

4

5

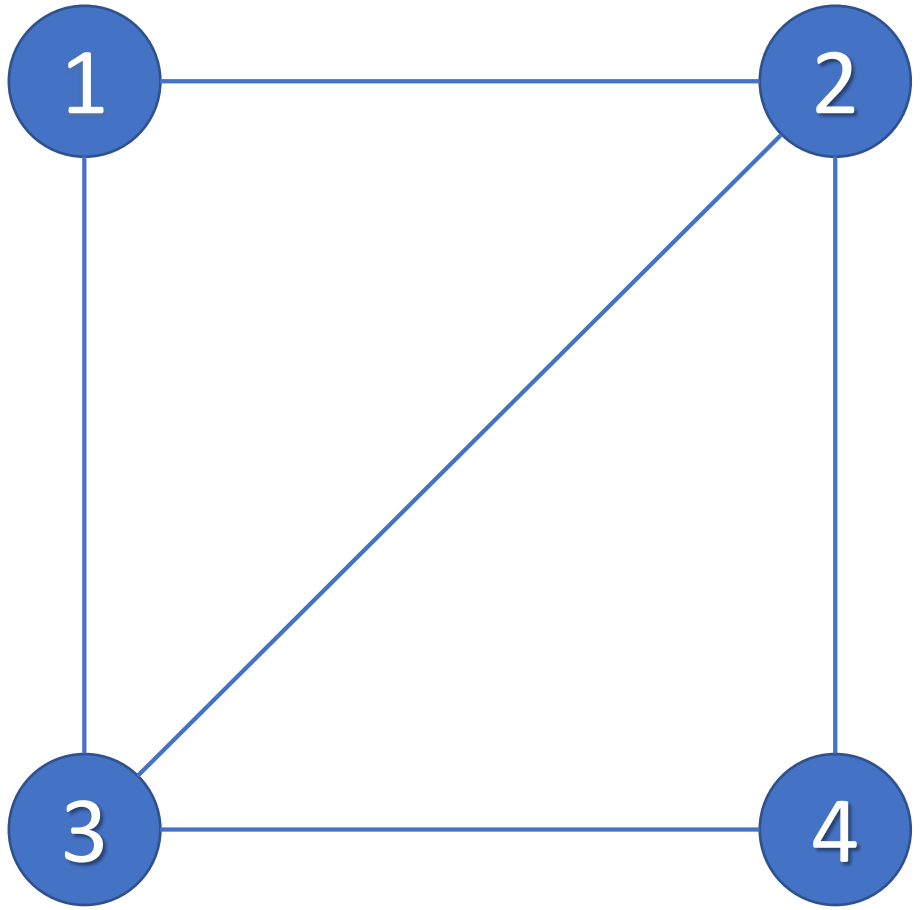
1 2

1 3

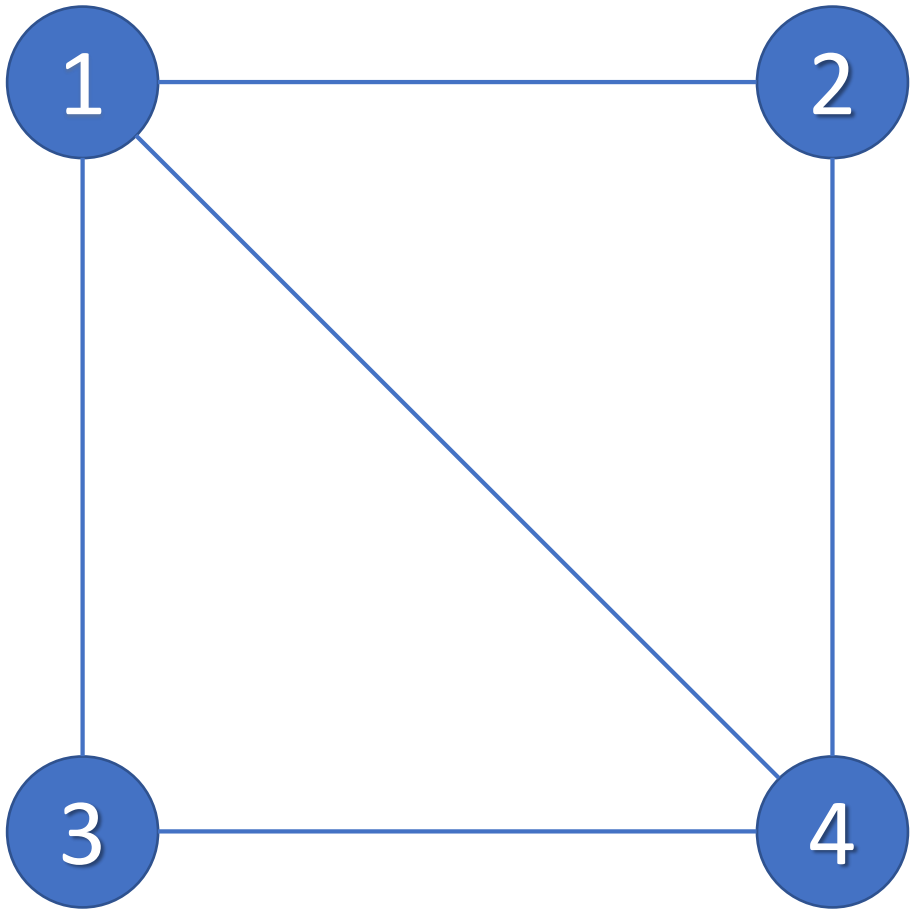
2 3

2 4

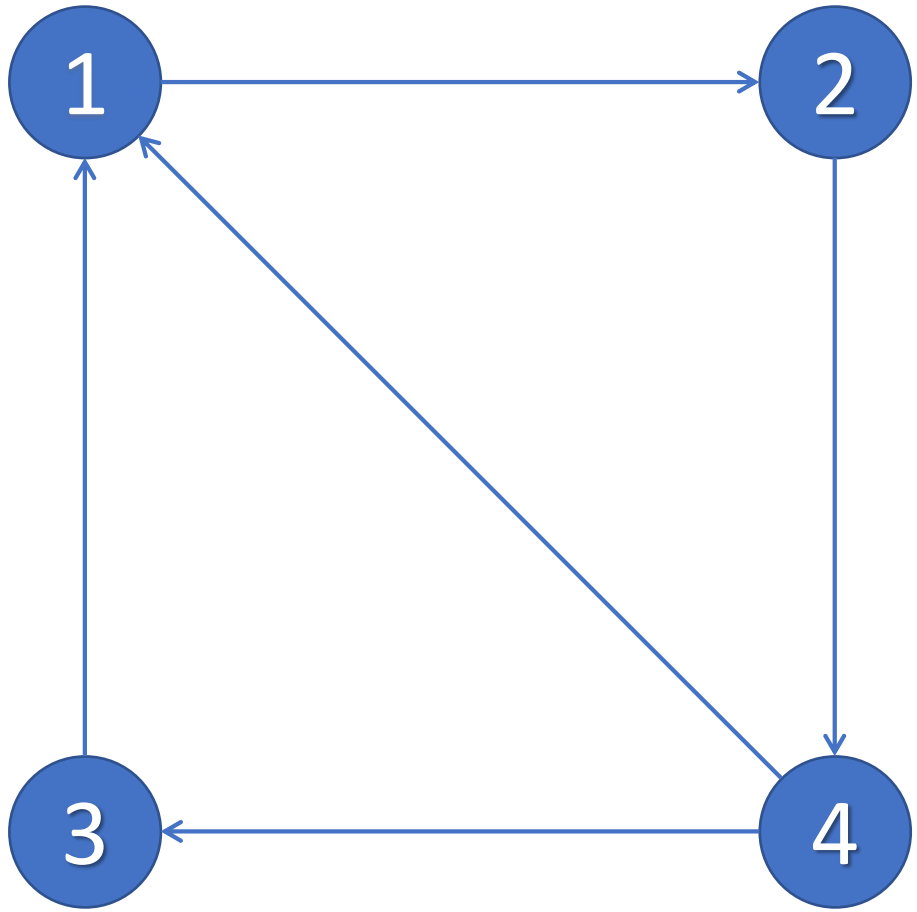
3 4



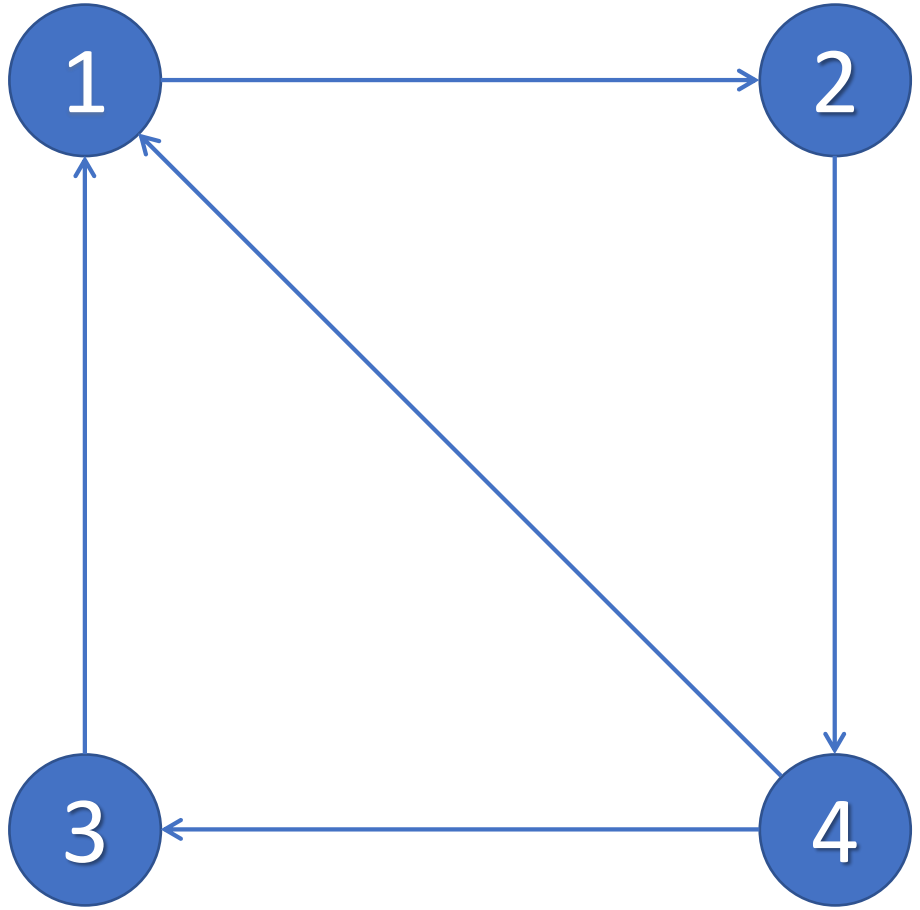
Problem Graph



Lösungs Graph

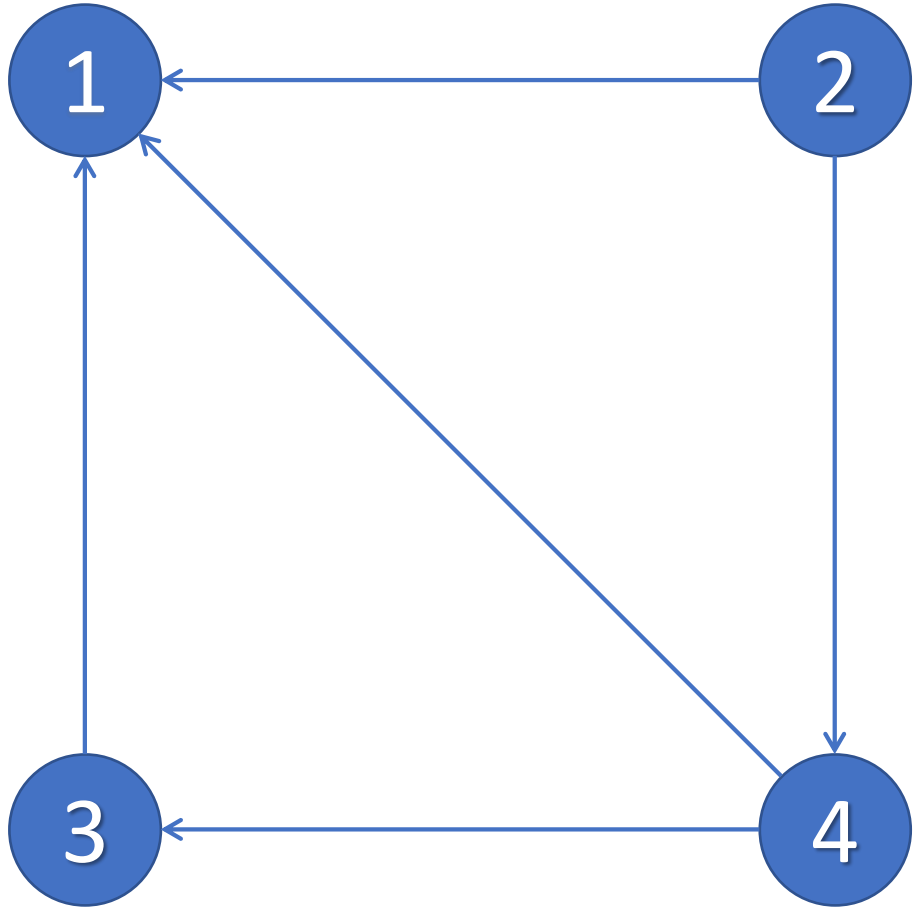


d = 2, da:
Zu dem Knoten 1 zwei gerichtete Kanten führen.



d = 3, da:
Zu dem Knoten 1 drei gerichtete Kanten führen.

d ist also die größte Anzahl an gerichteten Kanten, die im Graphen zu einem Knoten führen.
Diese Anzahl wollen wir so klein wie möglich halten.

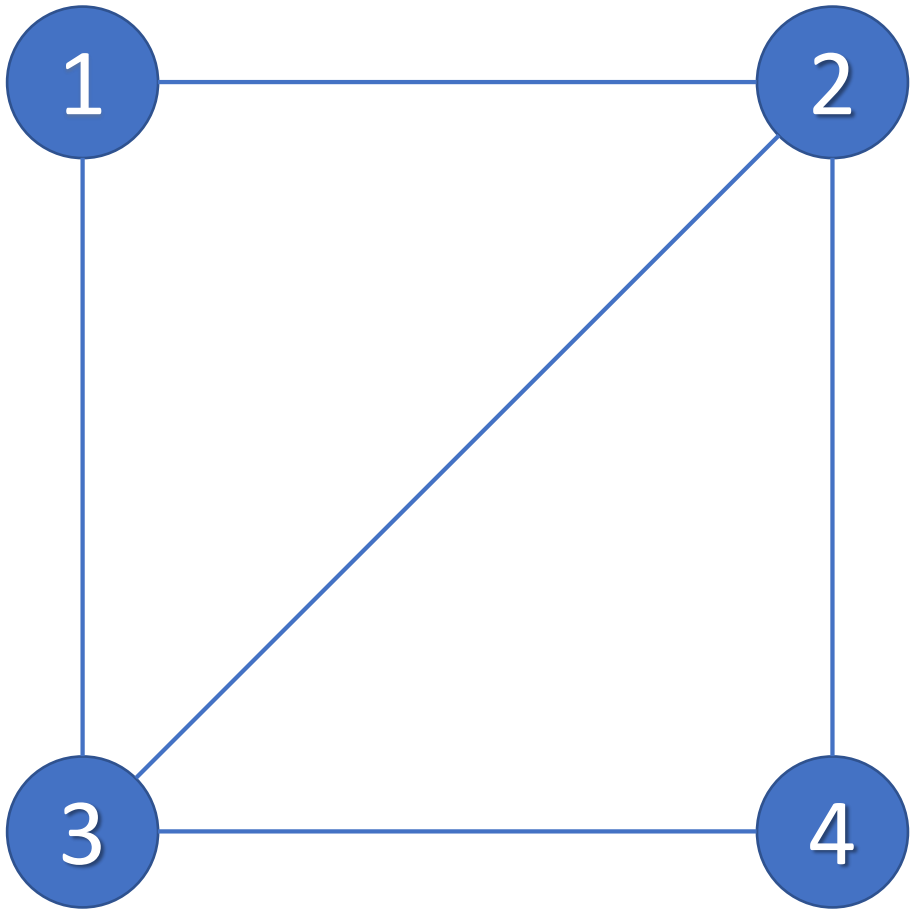


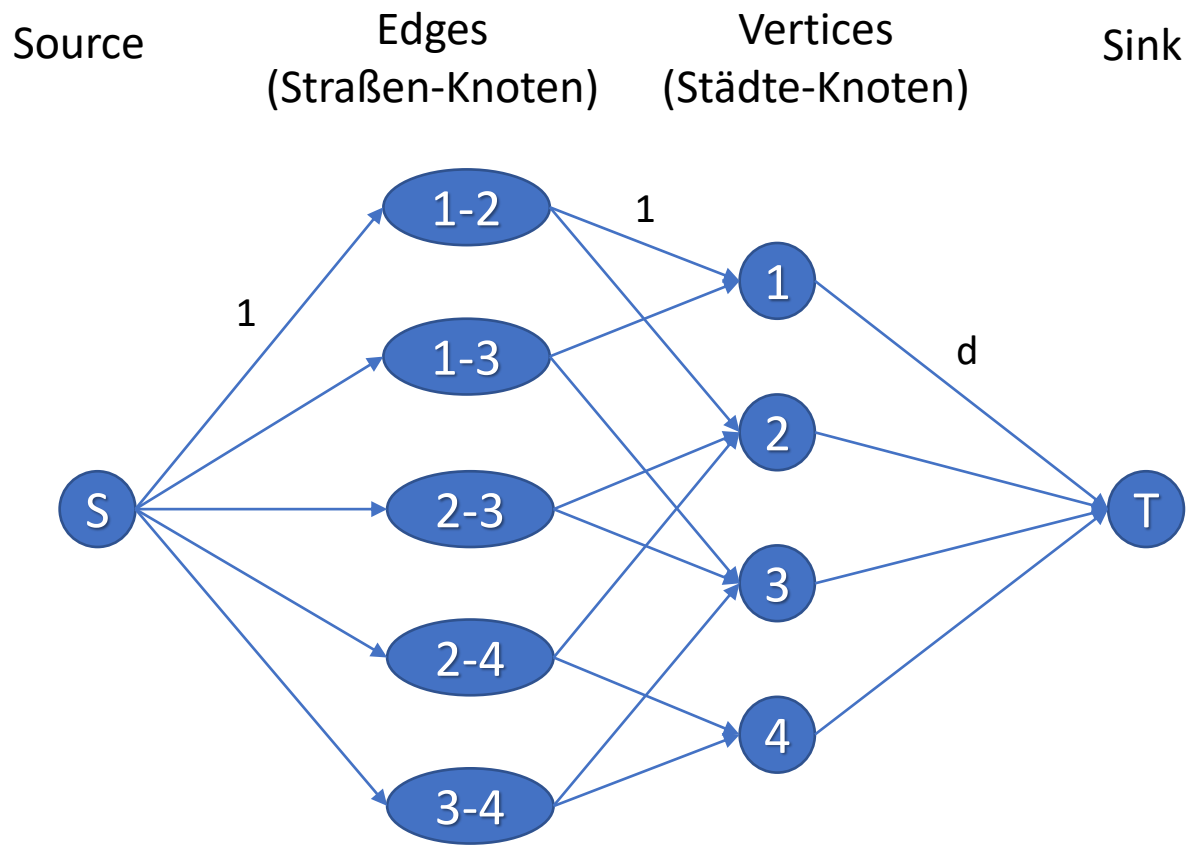
1.Lösungsansatz: Brute-Force

- Jede Orientierung ausprobieren
- Führt zur einer Laufzeit von $O(2^{|E|}*|E|)$
- Bei maximal $2,5*10^3$ viele Straßen ist das zu lange

2.Lösungsansatz: FlowGraph

- Idee: FlowGraph erstellen und damit den Kanten eine optimale Orientierung geben.
- Wie kommt man zu einem FlowGraphen?
- Wie gibt man den Kanten eine Orientierung?
- Wie wird d ermittelt?


1
2
3
4

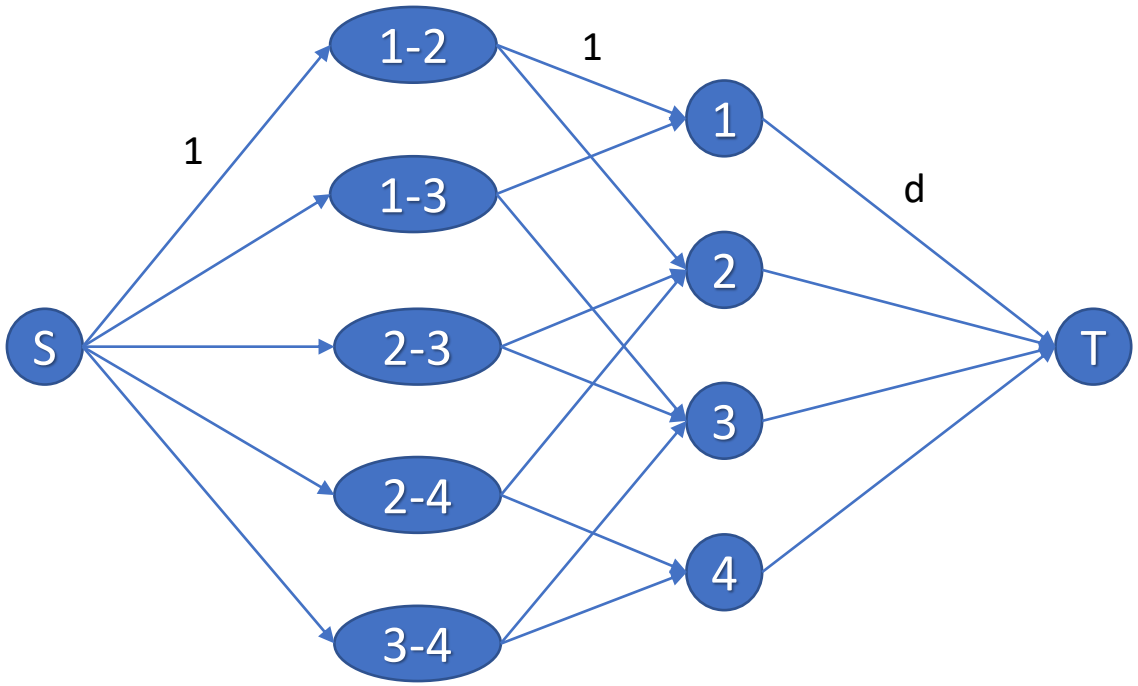

Source
Edges
(Straßen-Knoten)
Vertices
(Städte-Knoten)
Sink
1-2
1-3
2-3
2-4
3-4
1
2
3
4
S
T
1
1
d

Wie gibt man den Kanten eine Orientierung?

-Straßen-Knoten besitzen Zufluss von 1 und Abfluss von 2

Beispiel:
-Source zur Straßen-Knote 1-2 mit Flow 1
-Besitzt Kanten die zu Knoten 1 und 2 reichen
-Zufluss von 1-2 ist nur 1

→ Orientierung

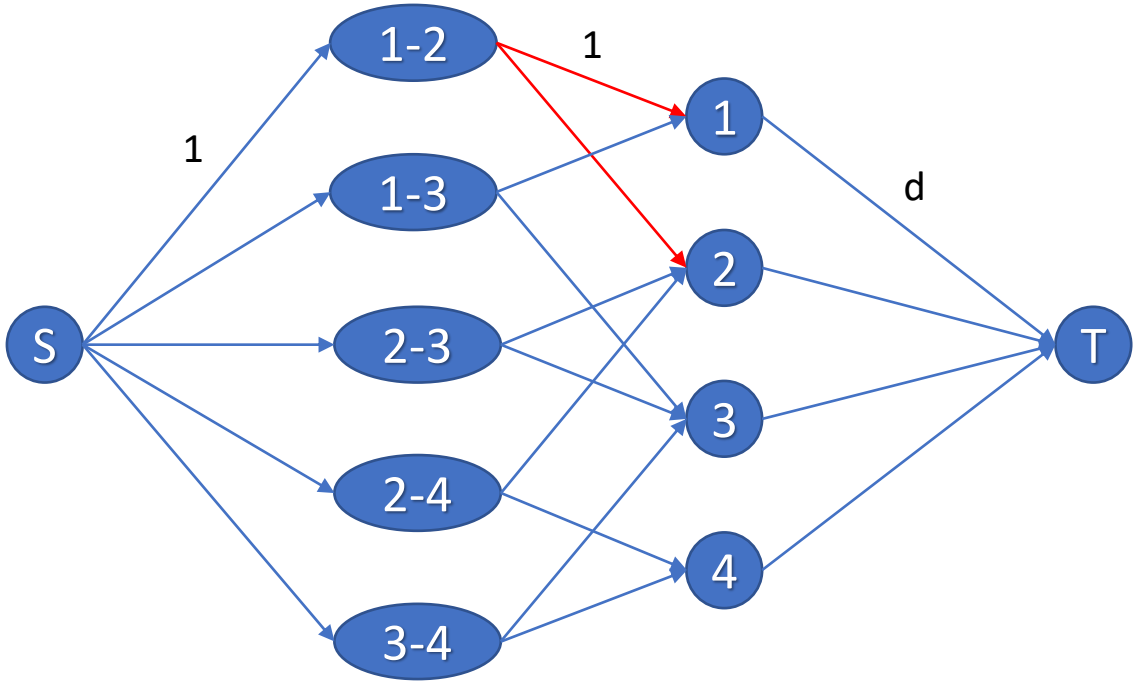


Wie gibt man den Kanten eine Orientierung?

-Straßen-Knoten besitzen Zufluss von 1 und Abfluss von 2

Beispiel:
-Source zur Straßen-Knote 1-2 mit Flow 1
-Besitzt Kanten die zu Knoten 1 und 2 reichen
-Zufluss von 1-2 ist nur 1

→ Orientierung



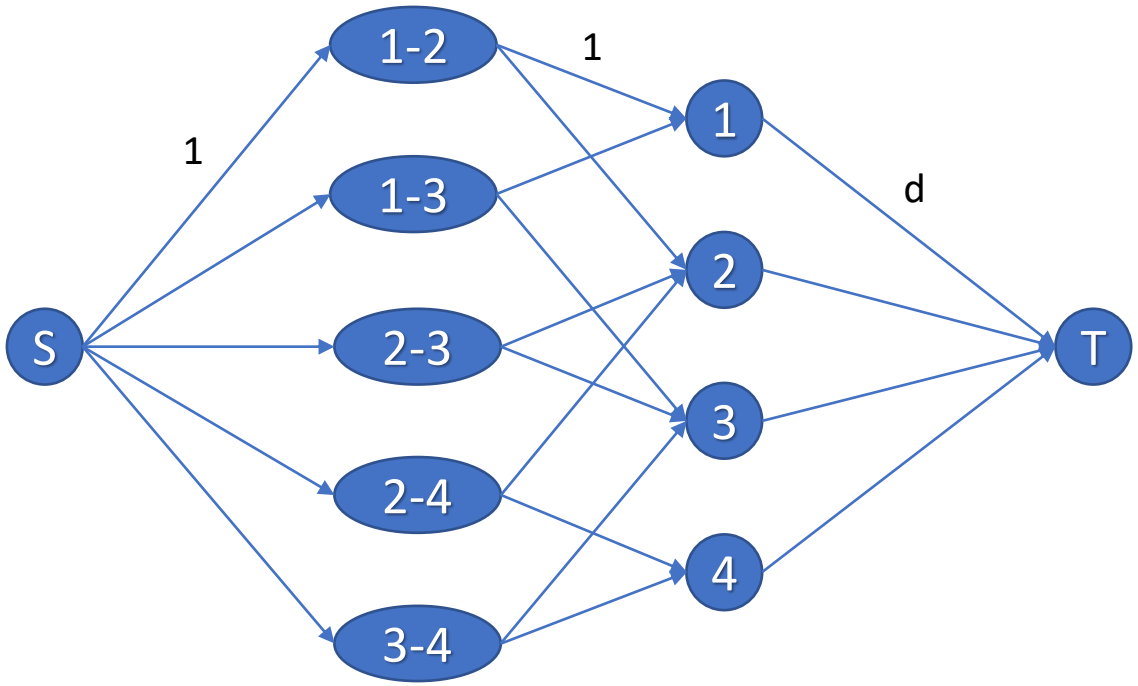
Wie gibt man den Kanten eine Orientierung?

-Kanten von Städte-Knoten zum Sink-Knoten besitzen variable Kapazität.

-MaxFlow führt zur Orientierung

-Grund: Es führt nur eine Kante zu einer Stadt

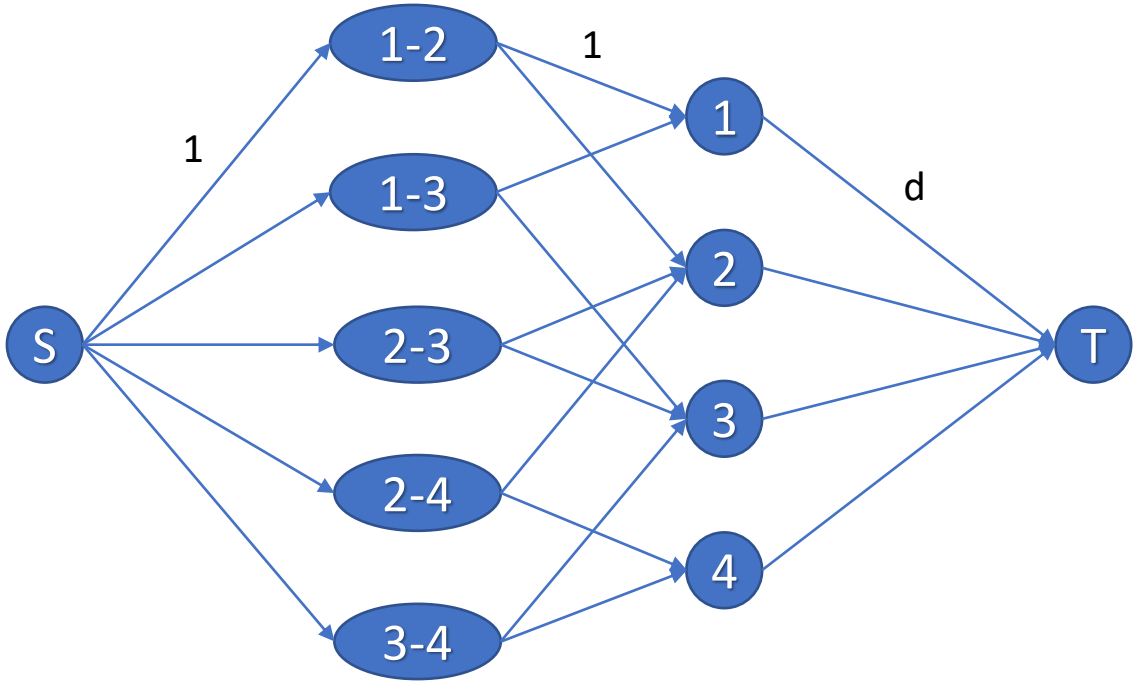
→ MaxFlow berechnen



Wie wird d ermittelt?

-Zufluss eines Stadt-Knotens v = Anzahl der Straßen u für die gilt, dass (u,v) ∈ E` und (u,v).flow = 1.
-Maximal n-1
-Beschränkung von d duch Abfluss.
-Ist d zu klein, besteht die Möglichkeit, dass eine Straße keine Orientierung zugeteilt werden kann →MaxFlow < m

Nun kann man über d iterieren, bis MaxFlow gleich der Anzahl der Straßen ist.



Wie berechnen wir unseren MaxFlow?
→ Edmons-Karp

PseudoCode:

```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```

→Alle flows mit 0 initialisieren

→Gf = Residualgraph

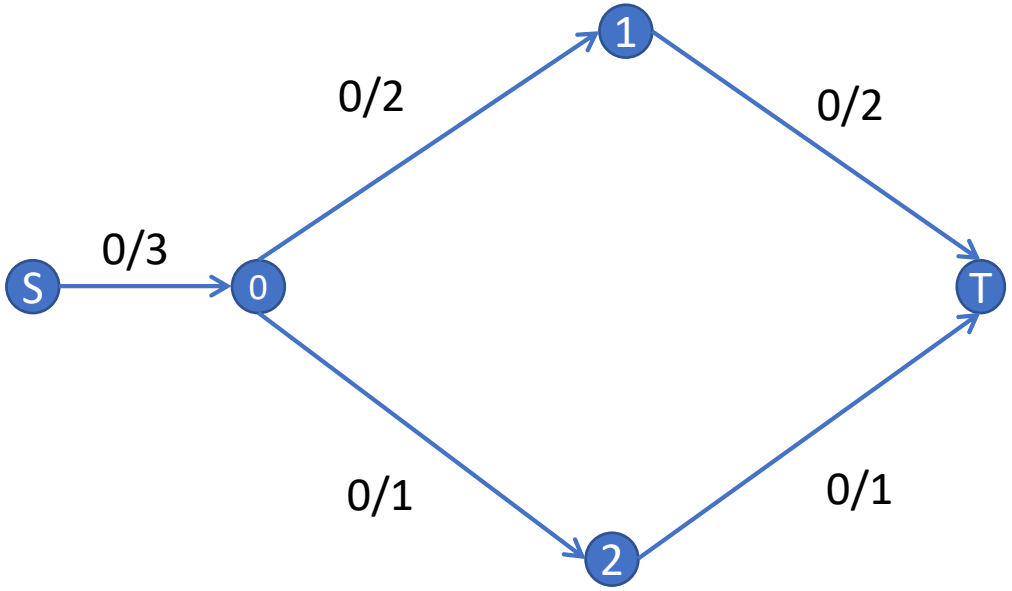
→Kürzesten Weg mit Hilfe von BFS finden

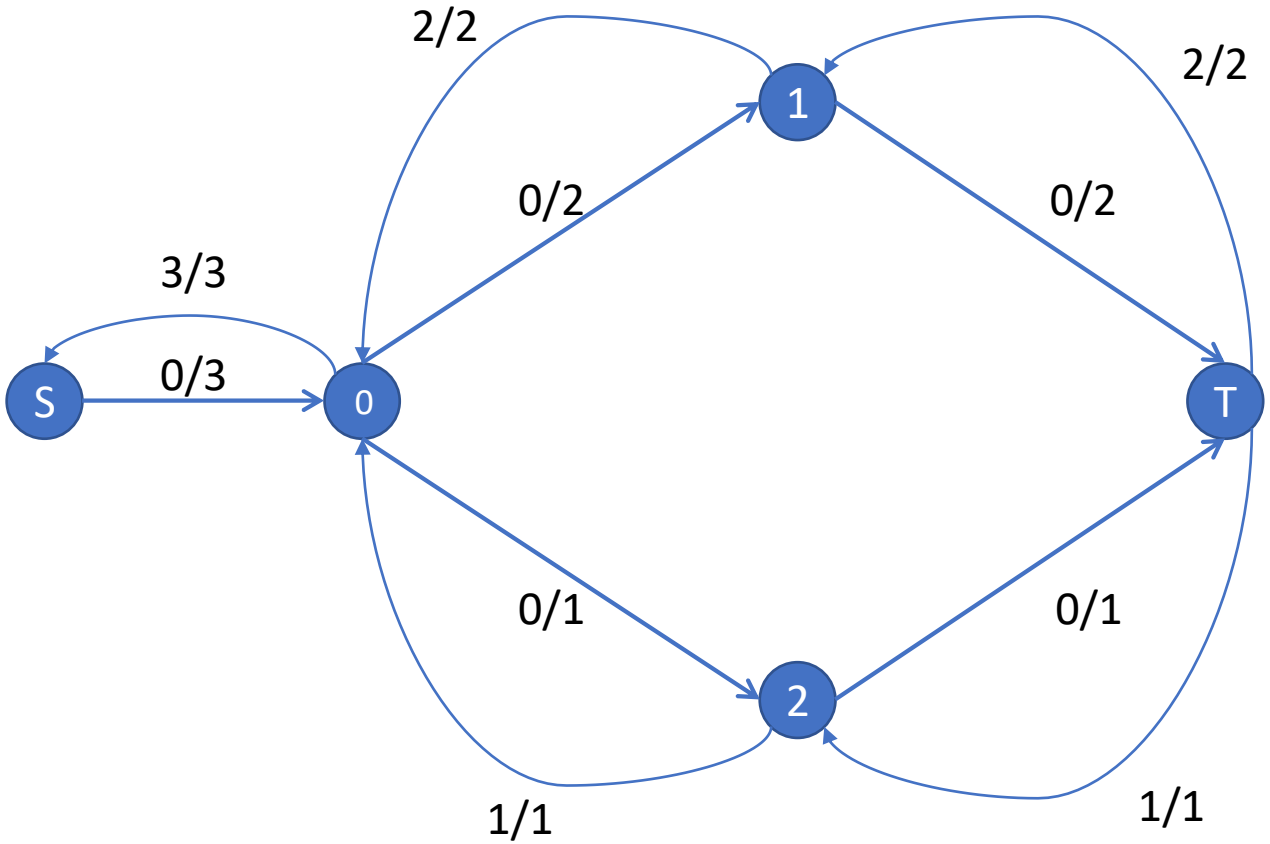
→Minimum flow vom kürzesten Weg

→Minimum auf die Kanten addieren
und von den Residualkanten abziehen

→ f zurückgeben

Wie erstellt man ein Residualgraphen?





Wir berechnen wir unseren MaxFlow?
→ Edmons-Karp

PseudoCode:

```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```

→Alle flows mit 0 initialisieren

→Gf = Residualgraph

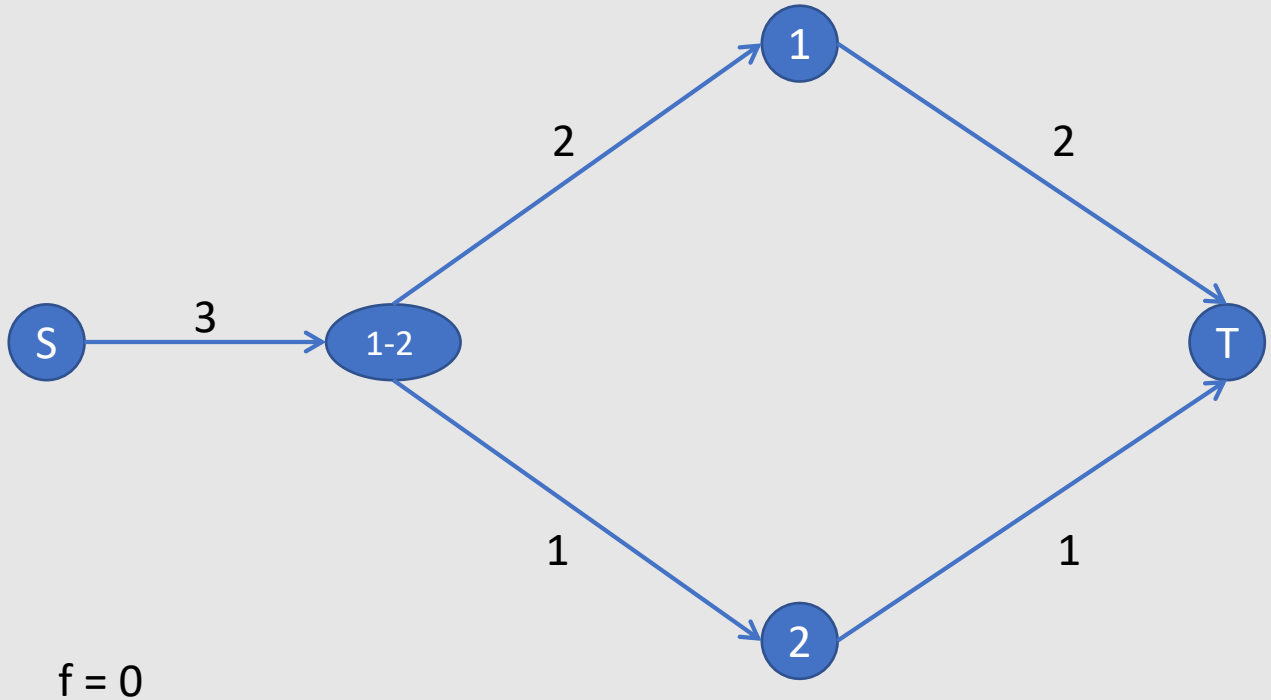
→Kürzesten Weg mit Hilfe von BFS finden

→Minimum flow vom kürzesten Weg

→Minimum auf die Kanten addieren
und von den Residualkanten abziehen

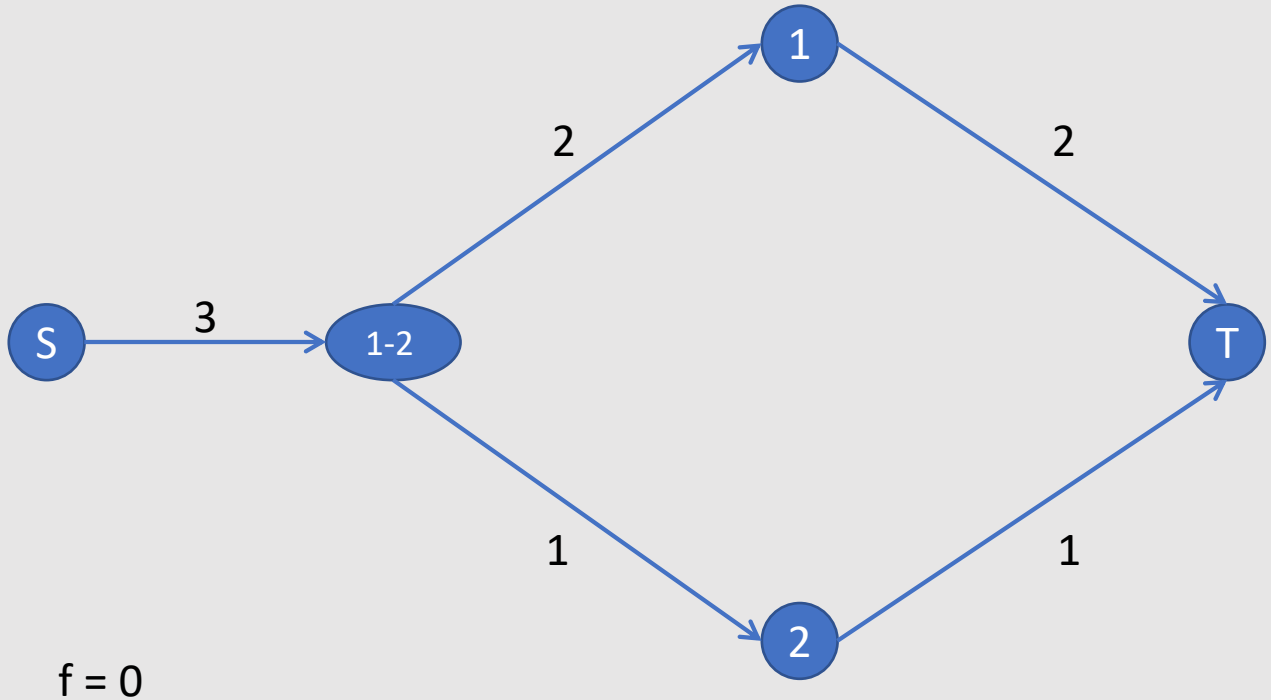
→ f zurückgeben

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f



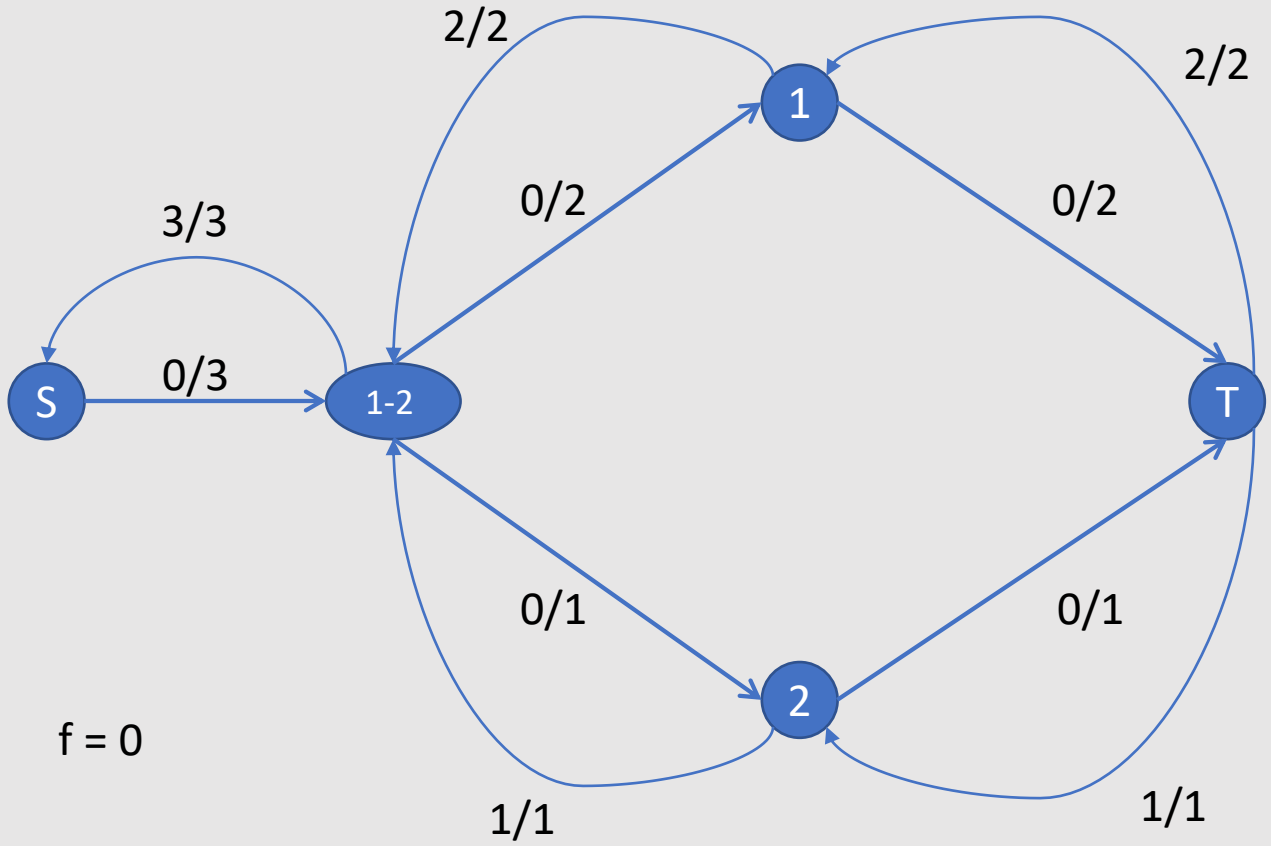
f = 0

```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```

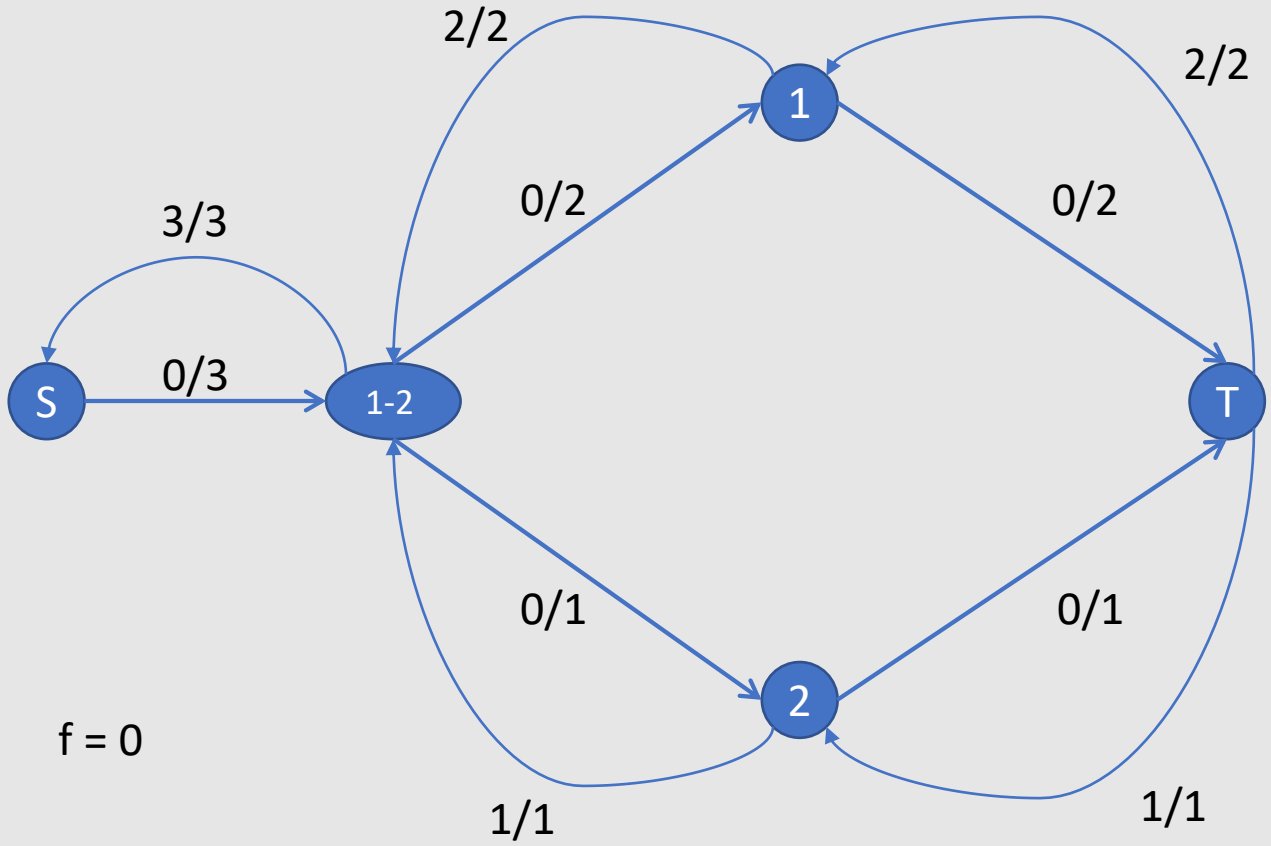


f = 0

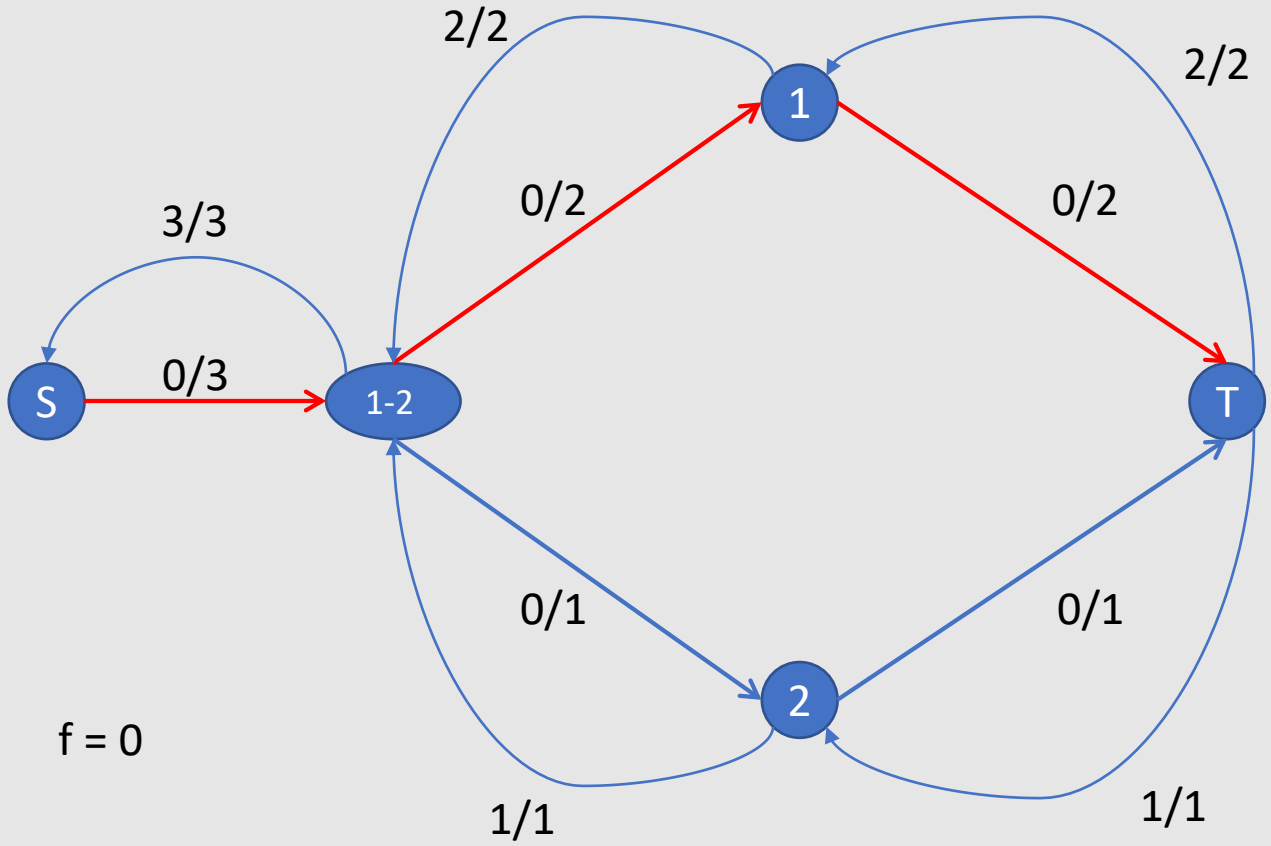
```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```



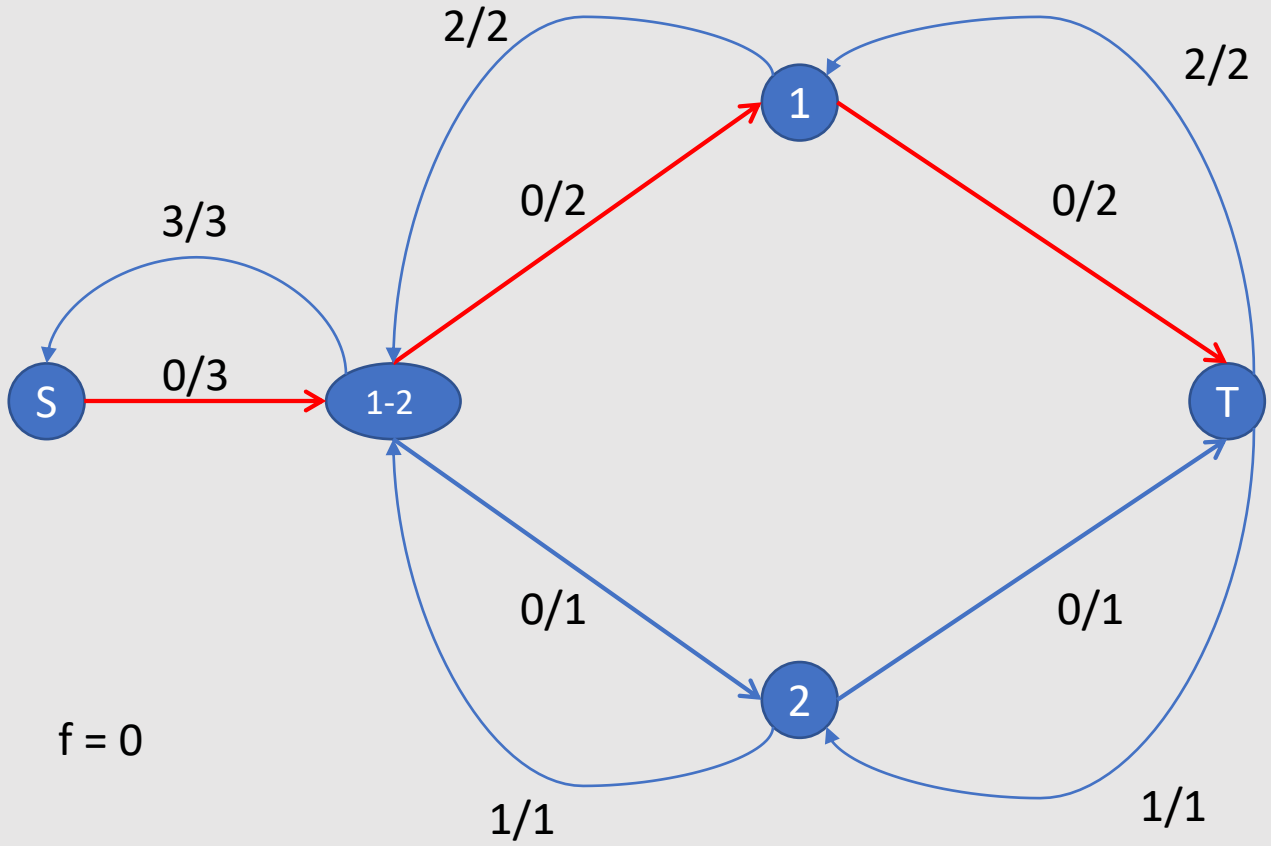
```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```



foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

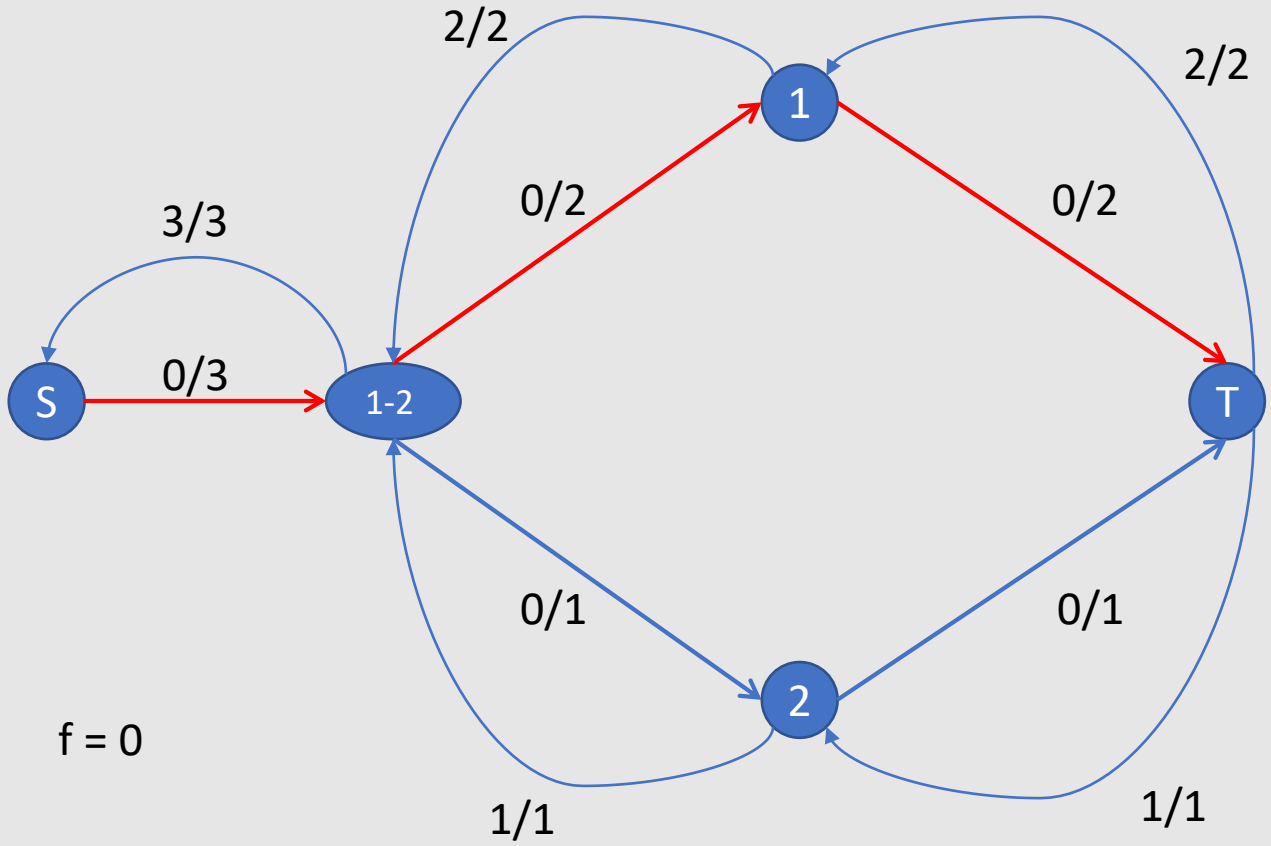


foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f



foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

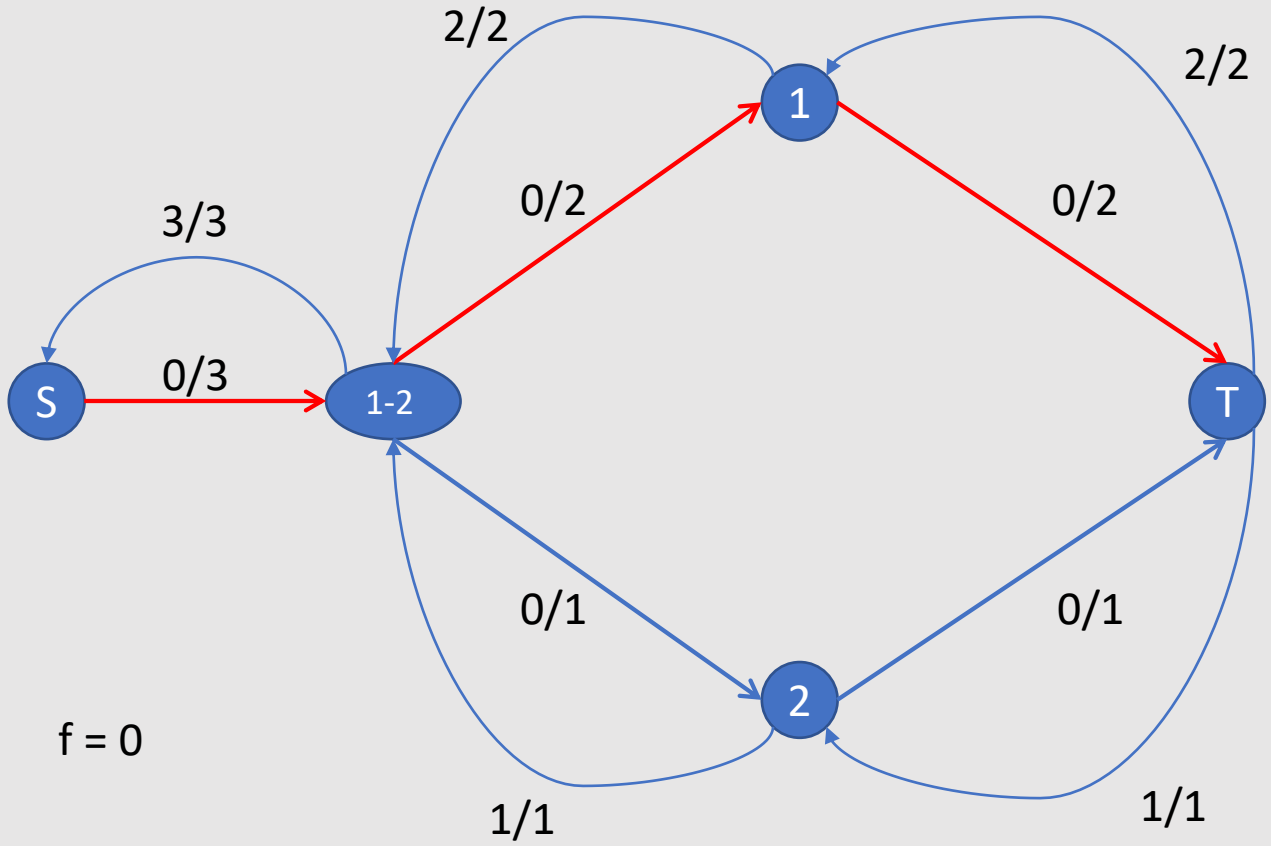
Δw = min(2,3) = 2



f = 0

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

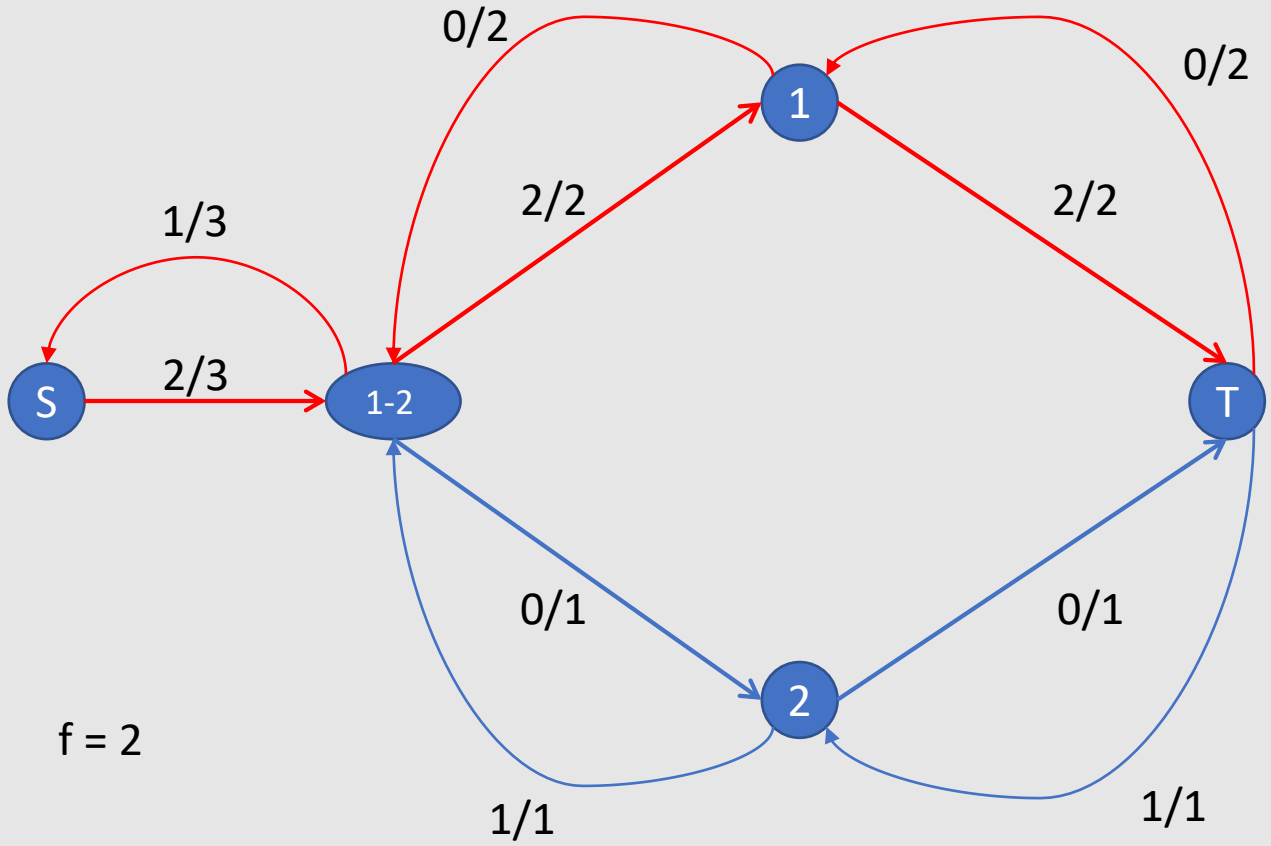
Δw = min(2,3) = 2



f = 0

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

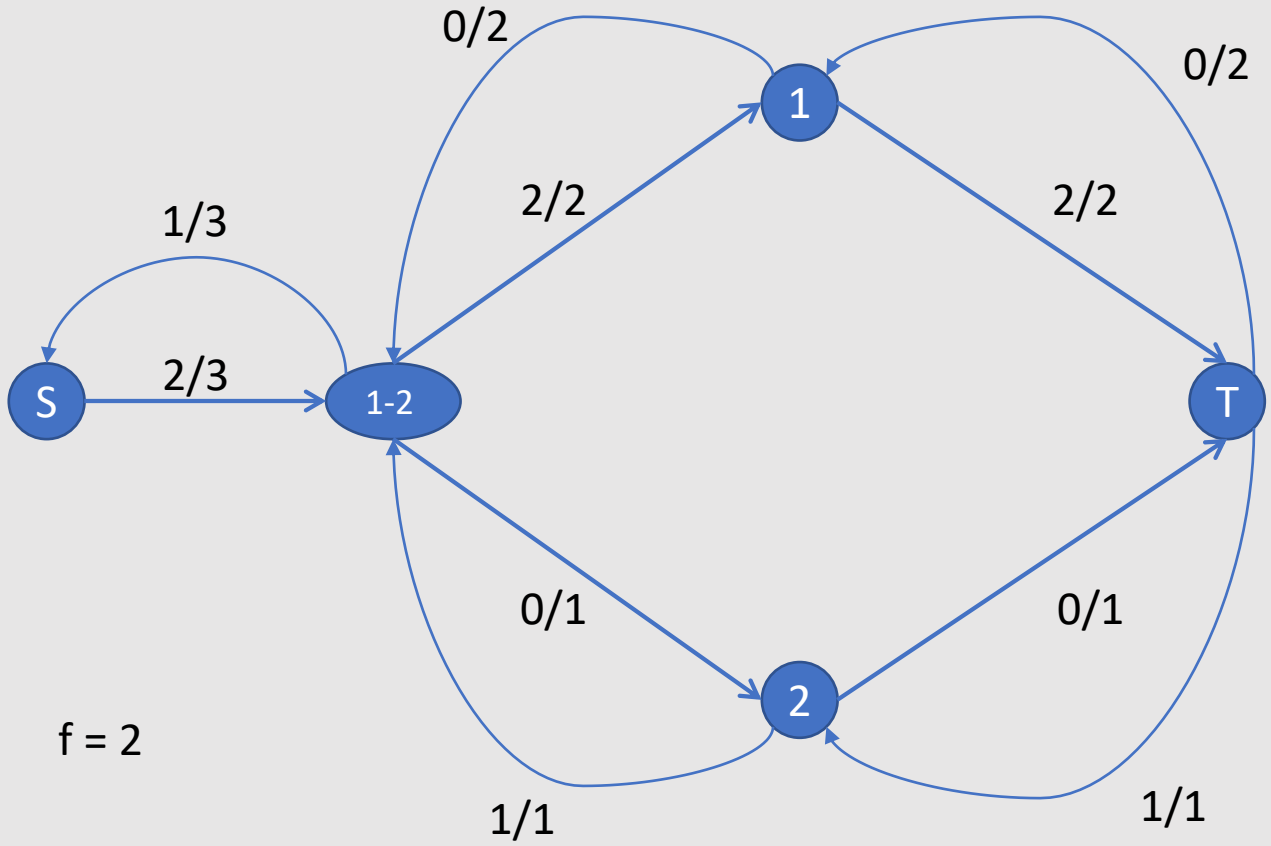
$\Delta w = \min(2,3) = 2$



f = 2

```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```

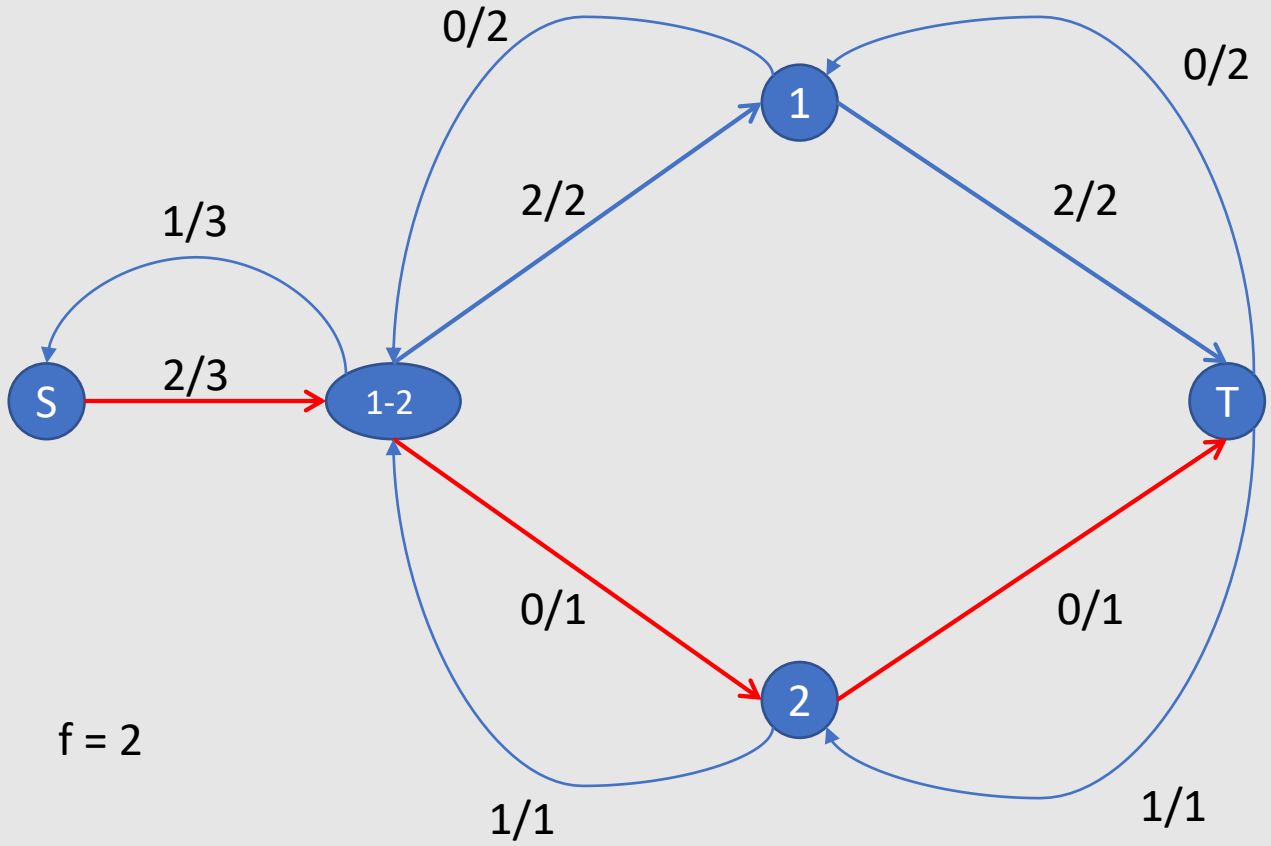
Δw = min(2,3) = 2



f = 2

```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```

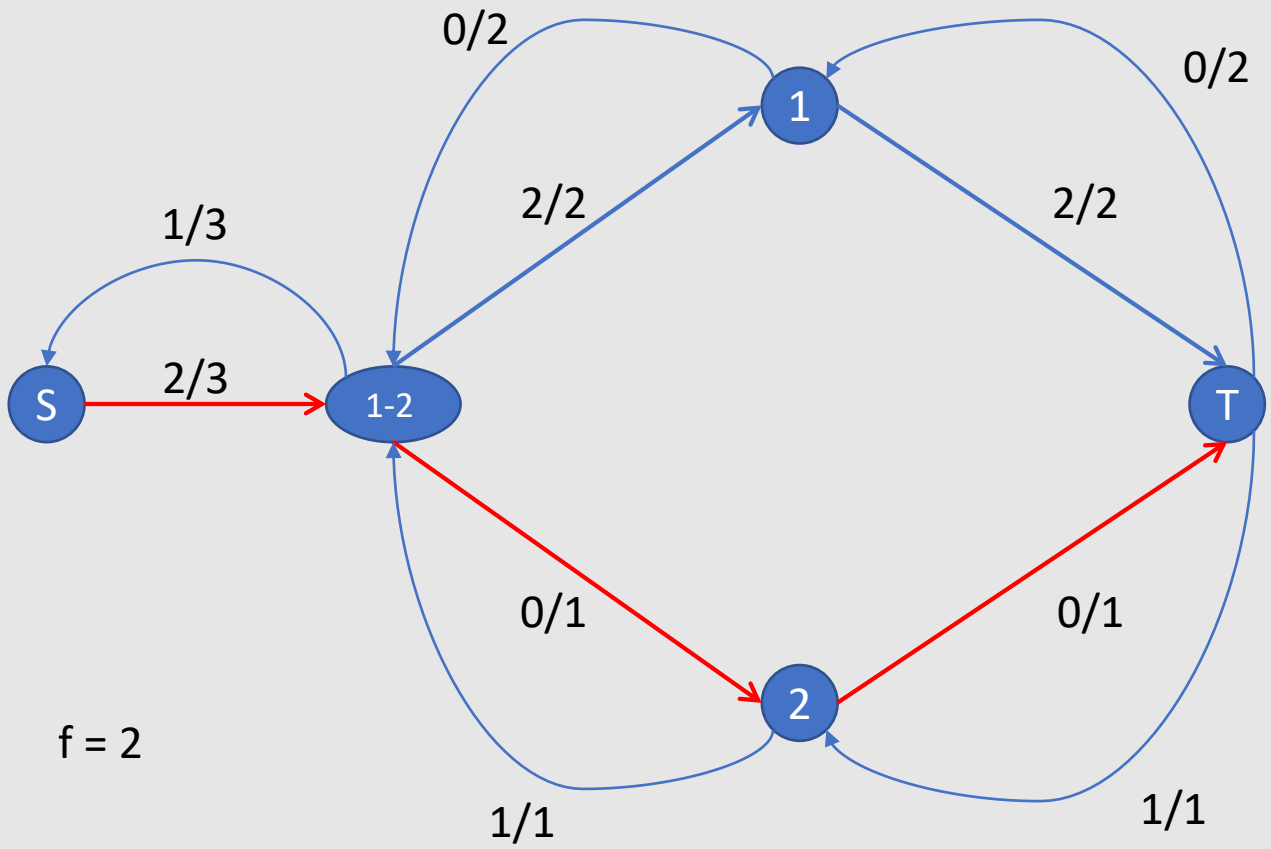
Δw = min(2,3) = 2



f = 2

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

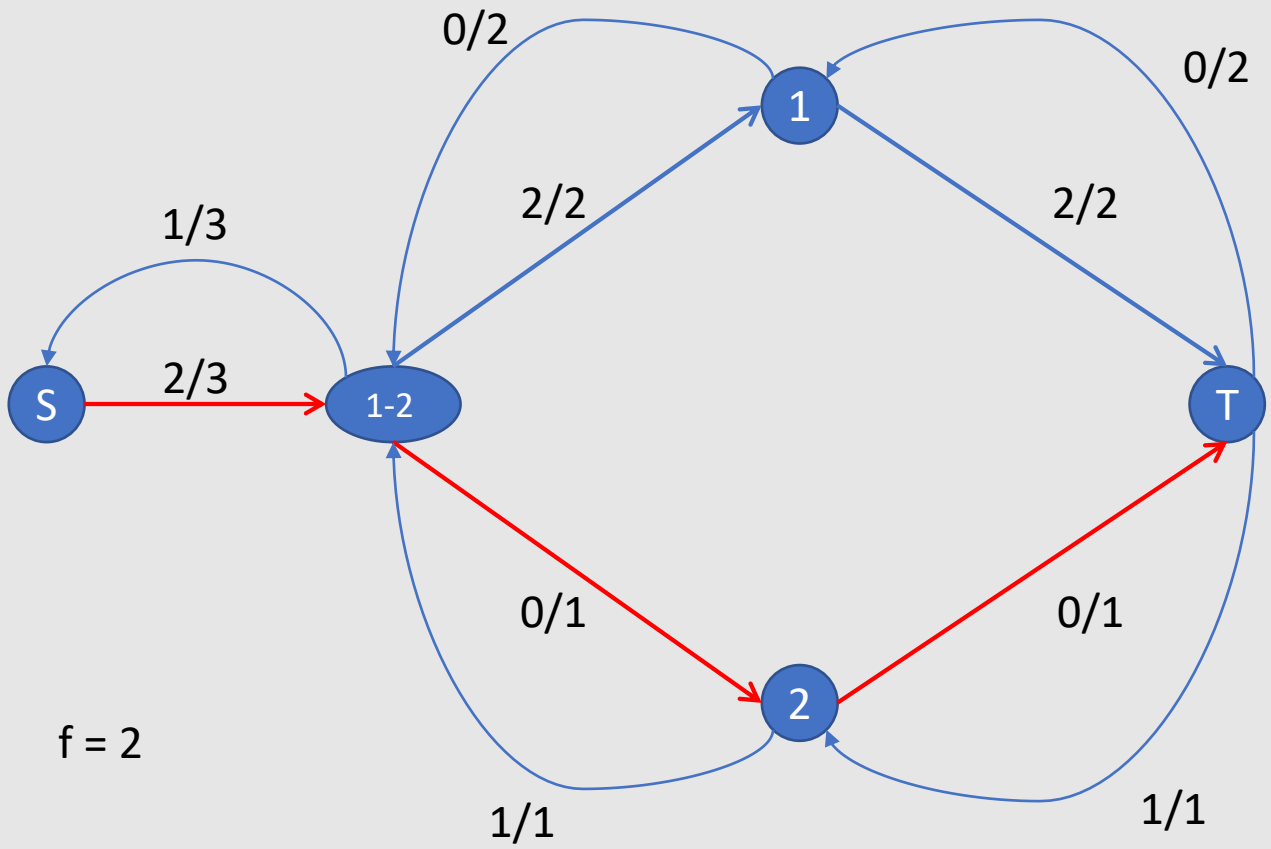
Δw = min(2,3) = 2



f = 2

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

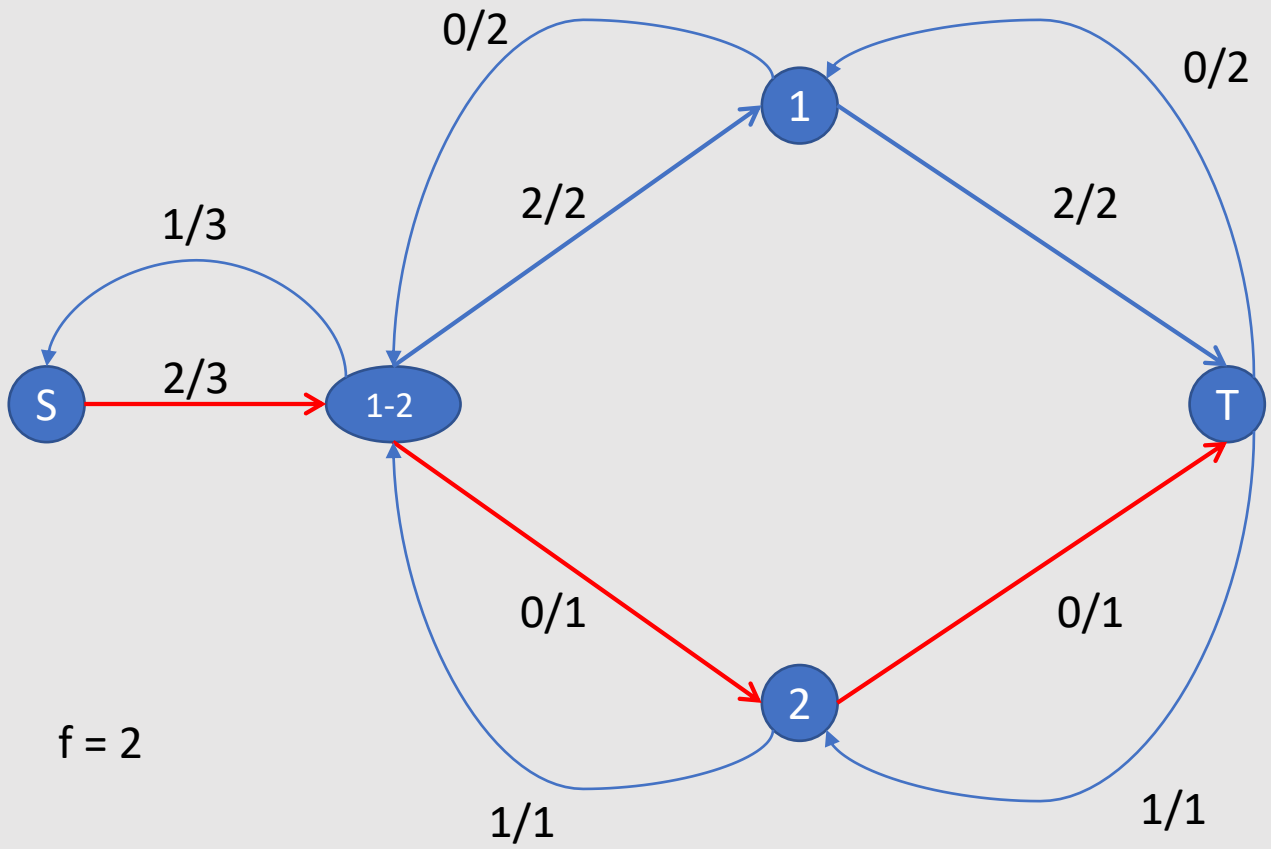
Δw = min(2,3) = 1



f = 2

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

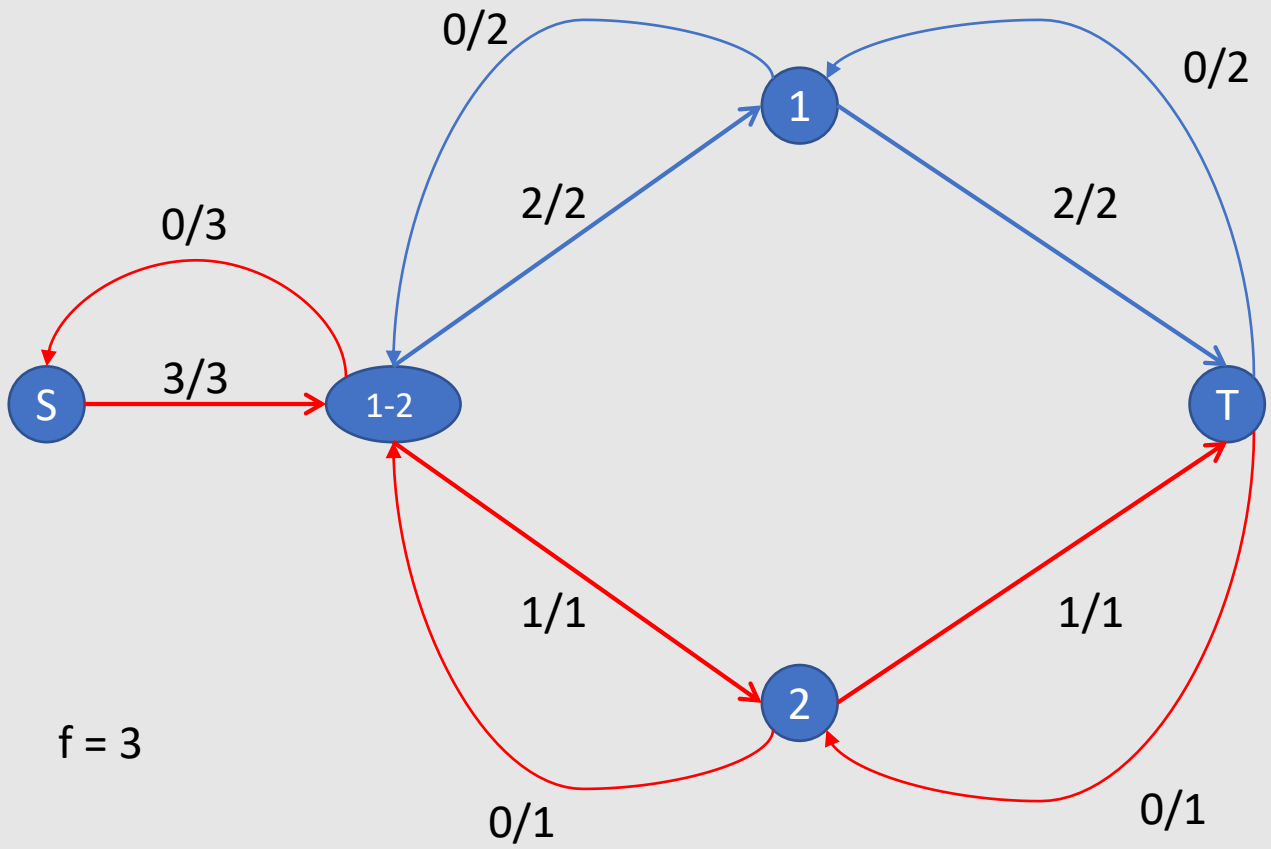
Δw = min(2,3) = 1



f = 2

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

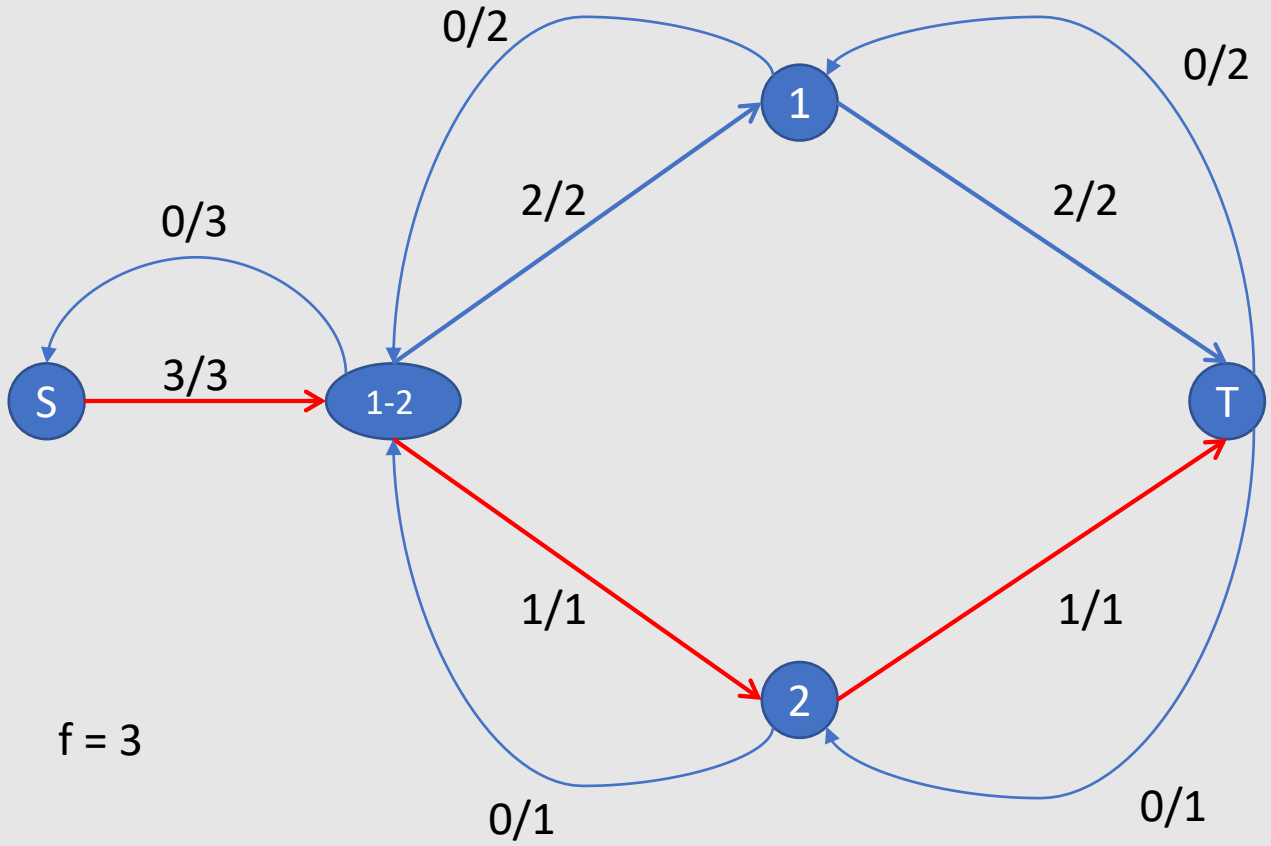
Δw = min(2,3) = 1



f = 3

```
foreach uv ∈ E do
    f_uv = 0
while G_f enthält s-t-Weg do
    W = kürzester s-t-Weg in G_f
    Δ_W = min_{uv∈W} c_f(uv)
    foreach uv ∈ W do
        if uv ∈ E then
            f_uv = f_uv + Δ_W
        else
            f_vu = f_vu − Δ_W
return f
```

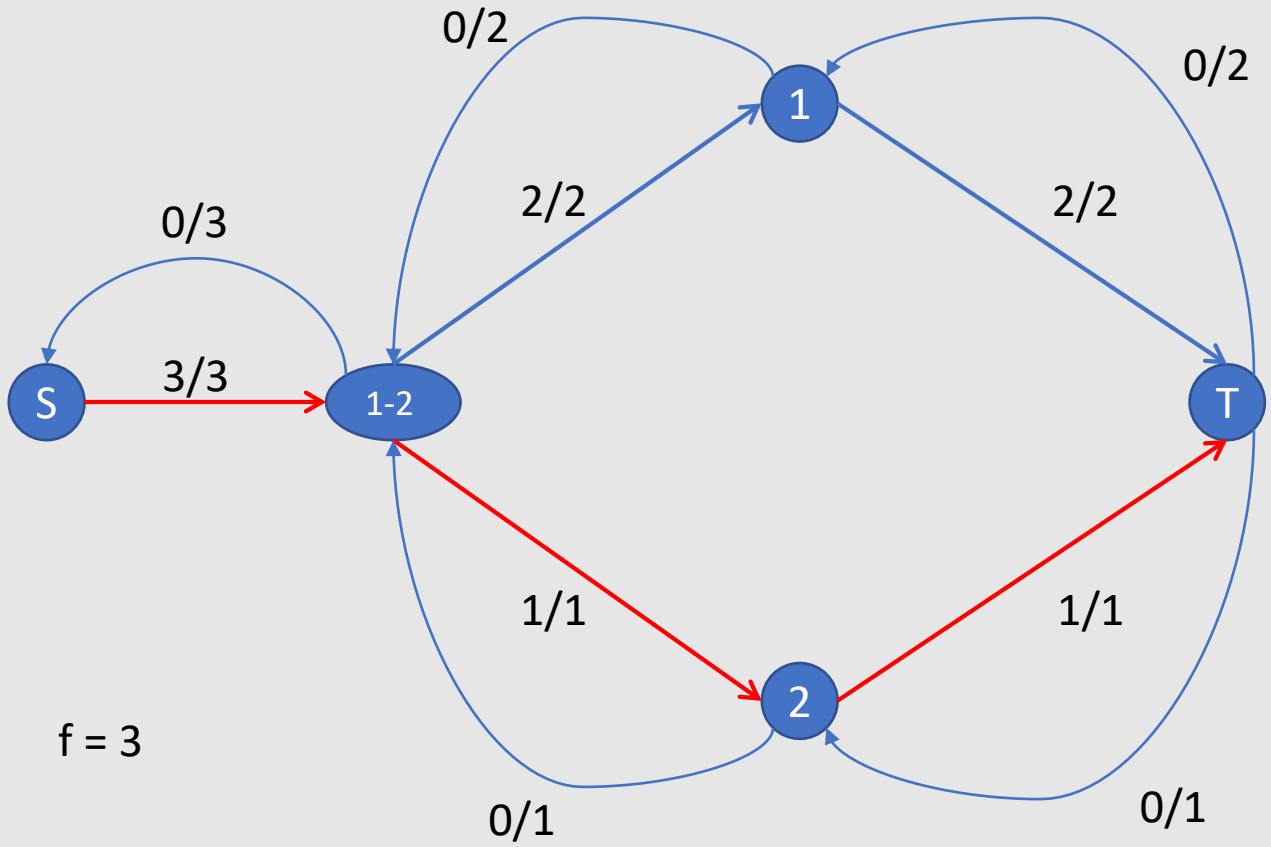
Δw = min(2,3) = 1



f = 3

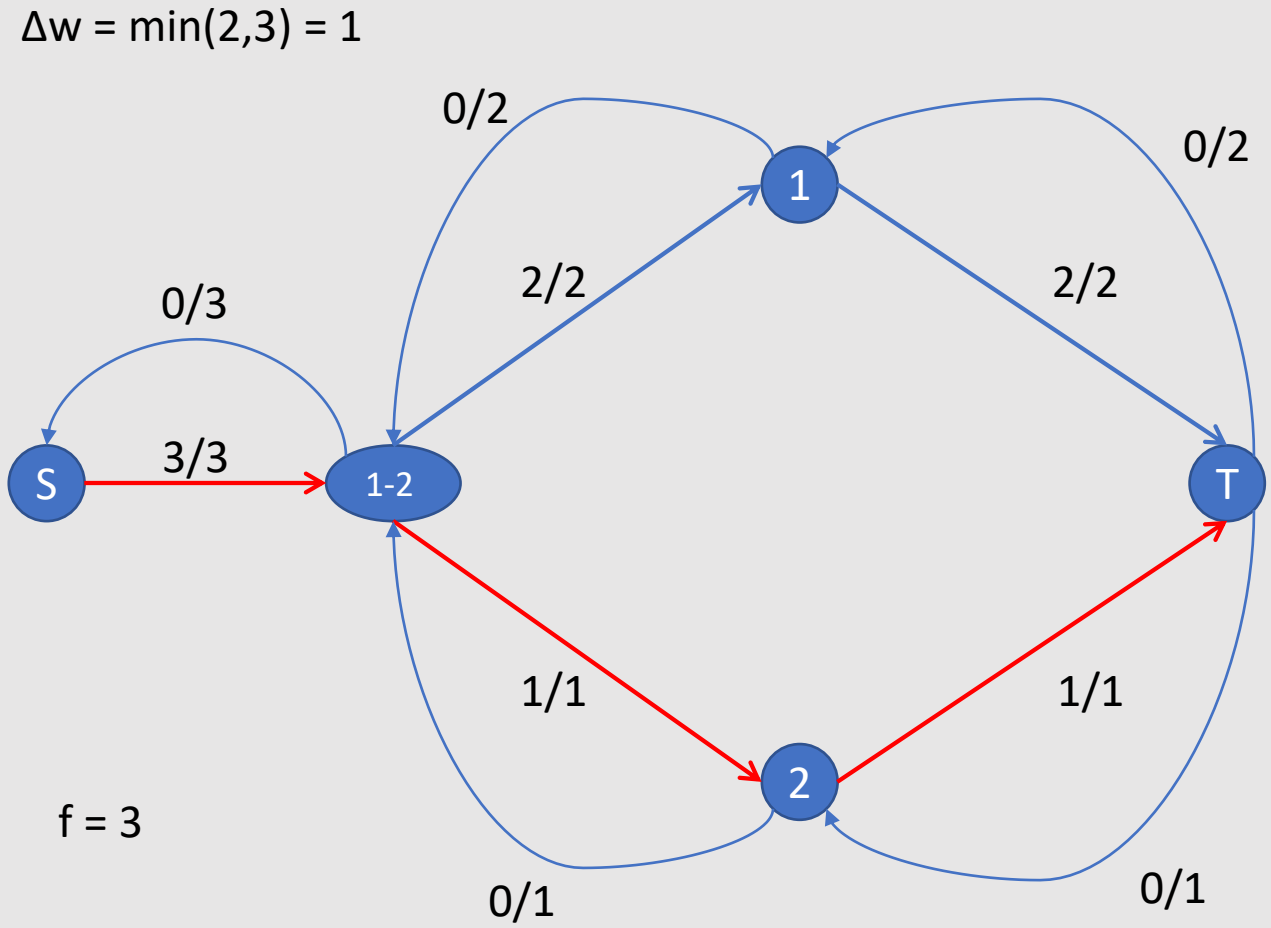
foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f

$\Delta w = \min(2,3) = 1$



f = 3

foreach $uv \in E$ **do**
  $f_{uv} = 0$
while G_f enthält s-t-Weg **do**
  $W =$ kürzester s-t-Weg in G_f
  $\Delta_W = \min_{uv \in W} c_f(uv)$
  foreach $uv \in W$ **do**
    if $uv \in E$ **then**
      $f_{uv} = f_{uv} + \Delta_W$
    else
      $f_{vu} = f_{vu} - \Delta_W$
return f



MaxFlow ist in diesem Graphen 3

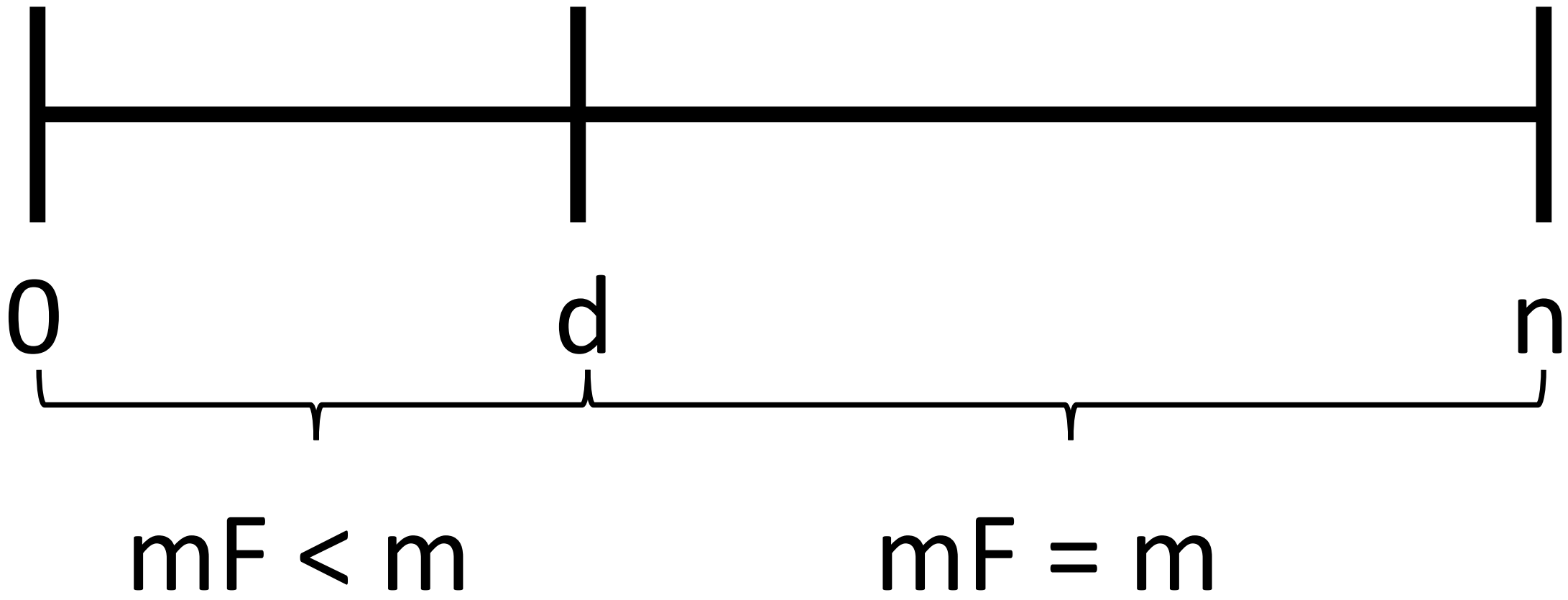
Fazit Lösungsansatz 2

- Die Anzahl der Kanten und Knoten des FlowGraphen G`(V`,E`) entspricht:
  - |V`| = n + m + 2
  - |E`| = 3m + n
- Edmonds-Karp läuft in O(E*(V`+E`))
- Jedoch muss man die Kapazitäten der letzten Kanten immer wieder neu initialisieren, höchstens so oft wie es Städte gibt.
- Daher ist die Laufzeit O(V * (E*(V`+E`))), da wir über d iterieren
- Mit Notation aus dem Input: O(n*(m*((n+m+2)+(3m+n)))) = $O(m^2n + n^2m)$

Lösungsansatz 3: Edmonds Karp mit Binary Search

- Idee: Die Suche nach passender Kapazität mit Hilfe von Binary Search schneller durchführen
- Laufzeit von Binary Search: O(log(n))
- Damit würde unser Programm die Laufzeit O(log(V)*(E*(V`+E`))
- Vom Input ausgehend: O(log(n)*(m*((n+m+2)+(3m+n))) = O(max{log(n)(m^2), log(n)(mn)})

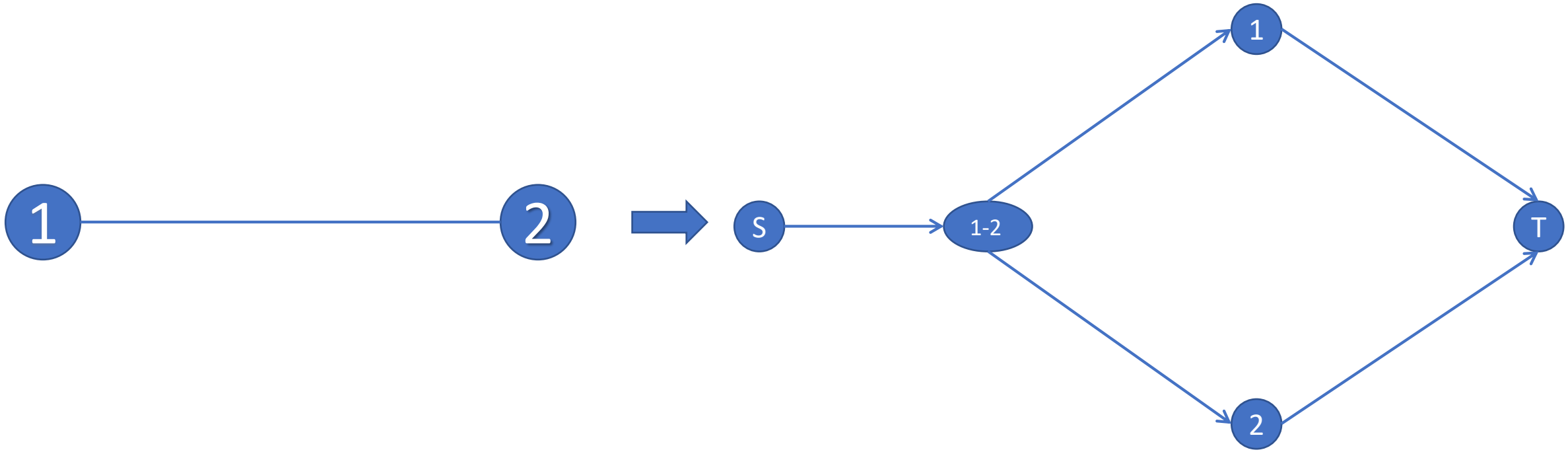
Warum BinarySearch?



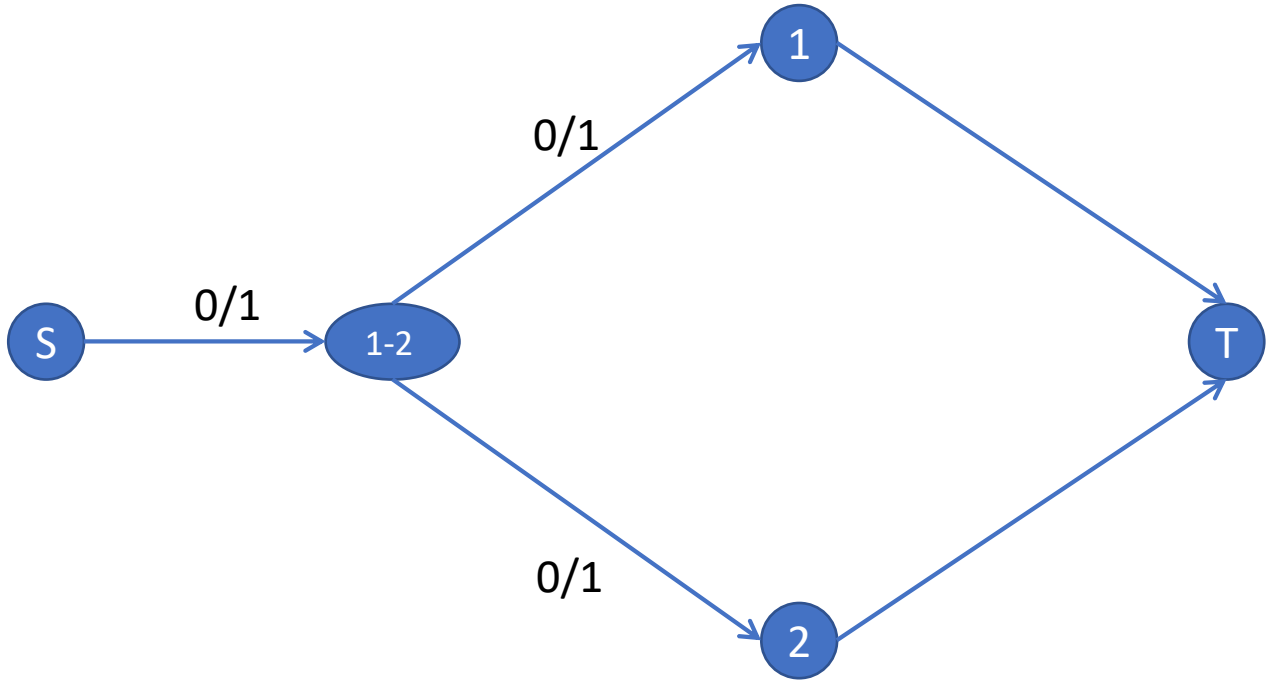
Wo fängt man bei Binary Search an?

- d ist maximal die Anzahl der Städte und minimal 0
- Daher ist low = 0 und high = Maximumgrad = |Städte|
- Der Anfang ist deswegen d = mid $= \frac{low+high}{2}$
- Wenn MaxFlow = |Straßen|, dann wird high = mid
- Sonst wird low = mid + 1
- Dies machen wir solange low >= high

Step by Step

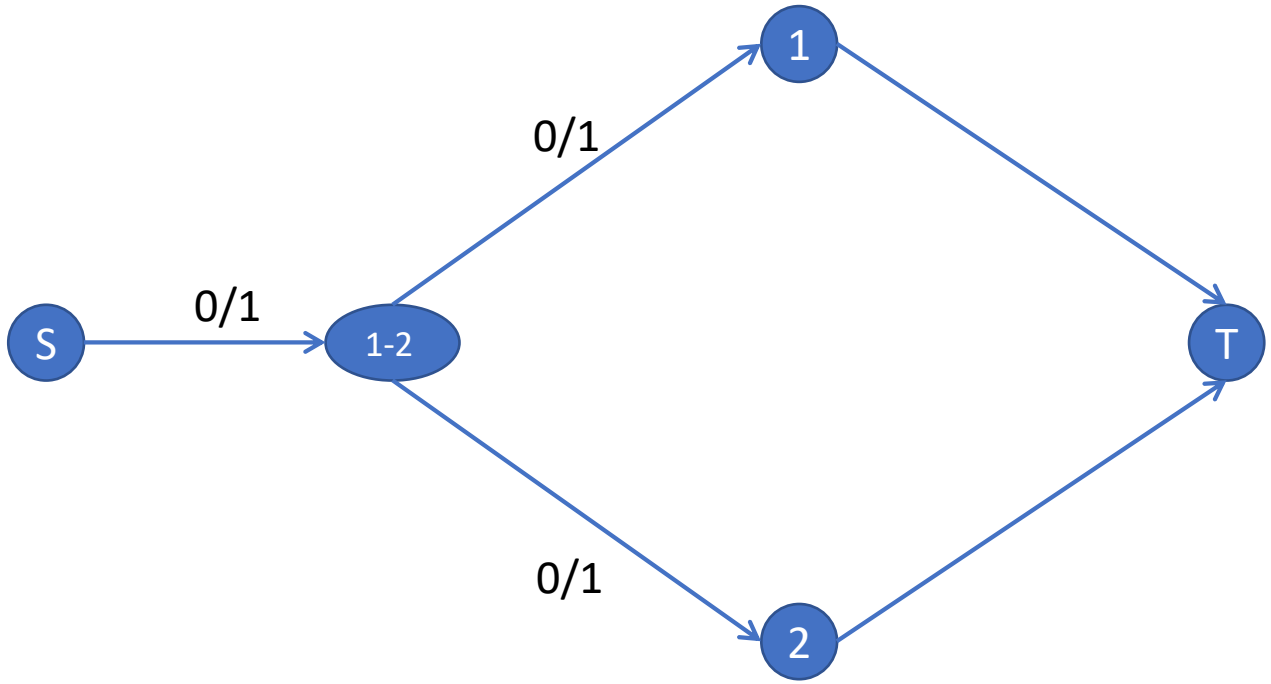

1
2
S
1-2
1
2
T

n = |Städte| = 2
m = | Straßen| = 1



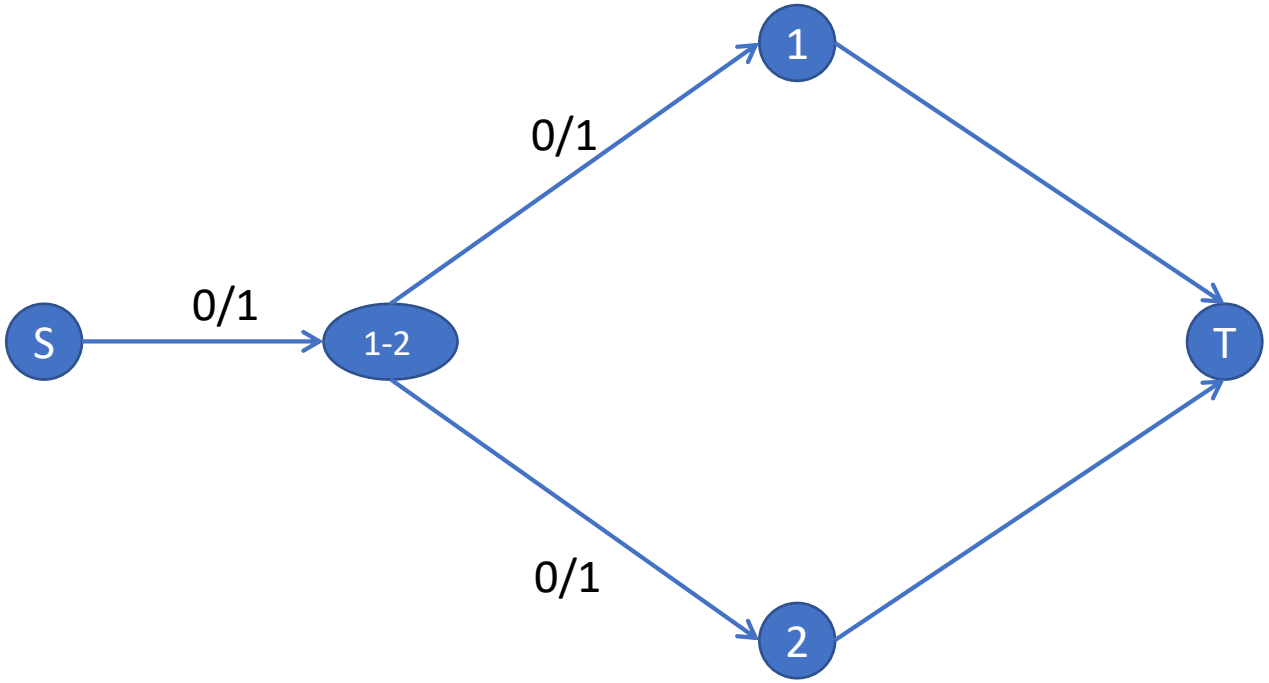
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2



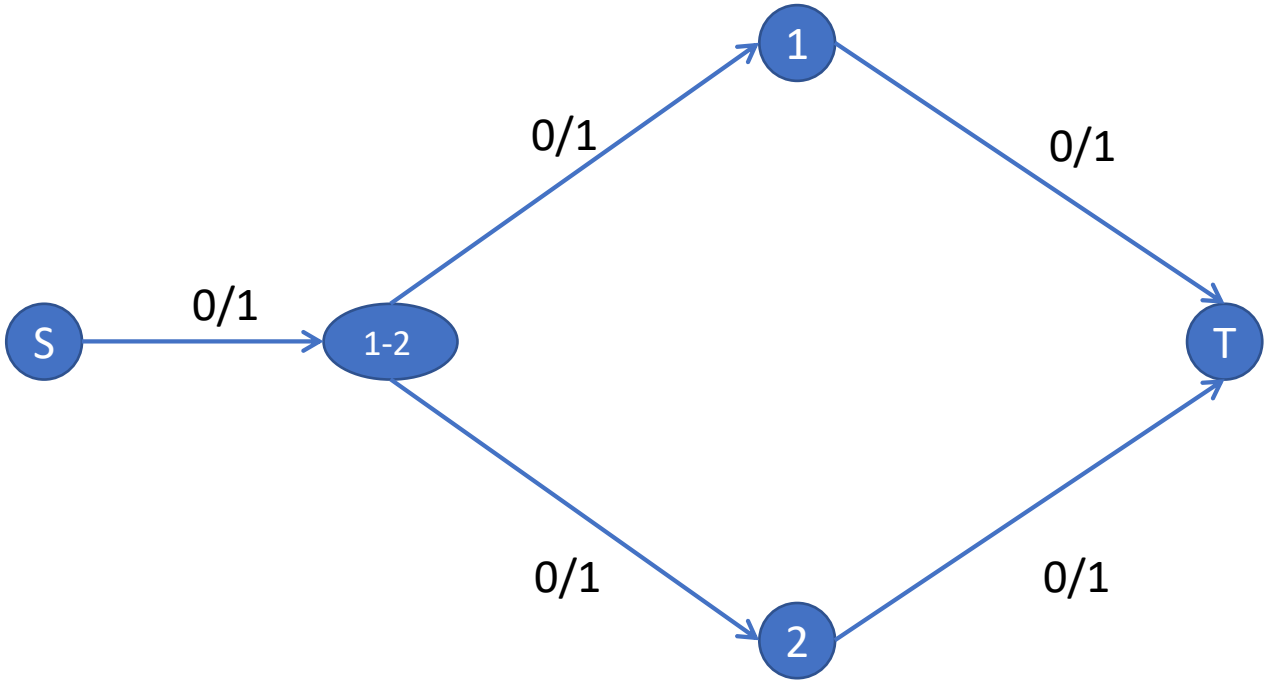
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2
while(low < high):



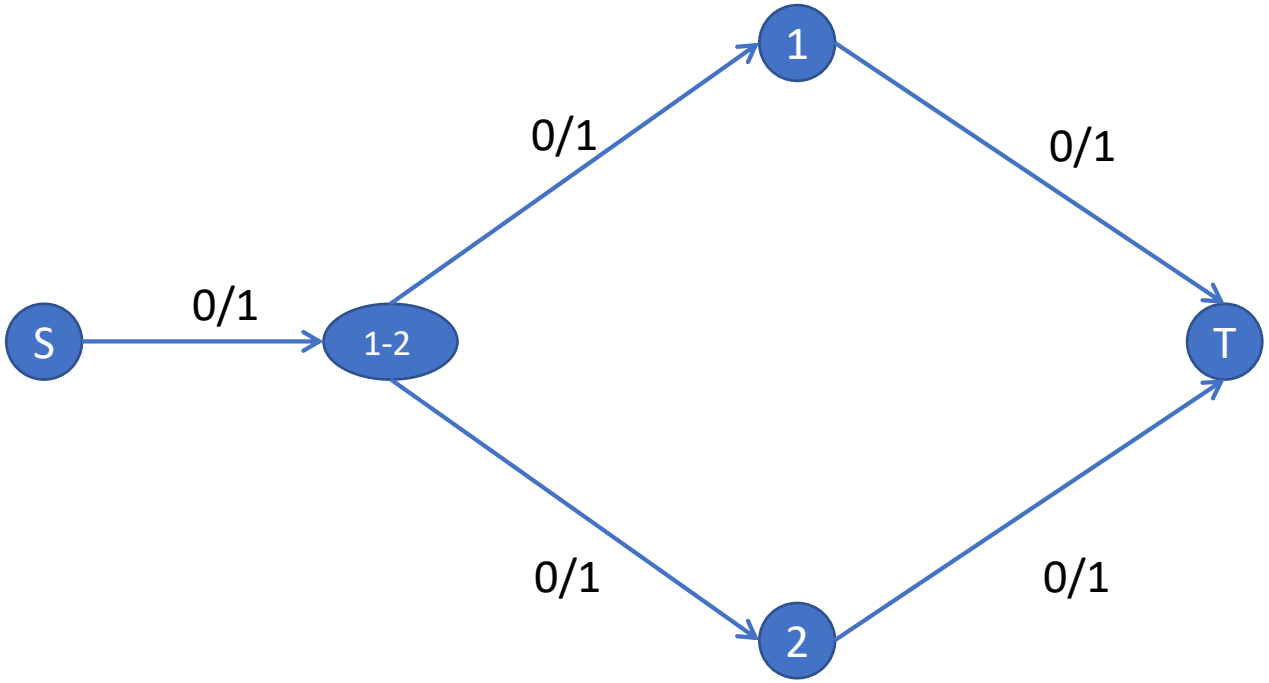
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2
while(low < high):
mid = $\frac{low+high}{2}$ = 1
forall edges e of Graph from
Stadt-Knoten to Sink-Knoten do:
e.setCapacity(mid)

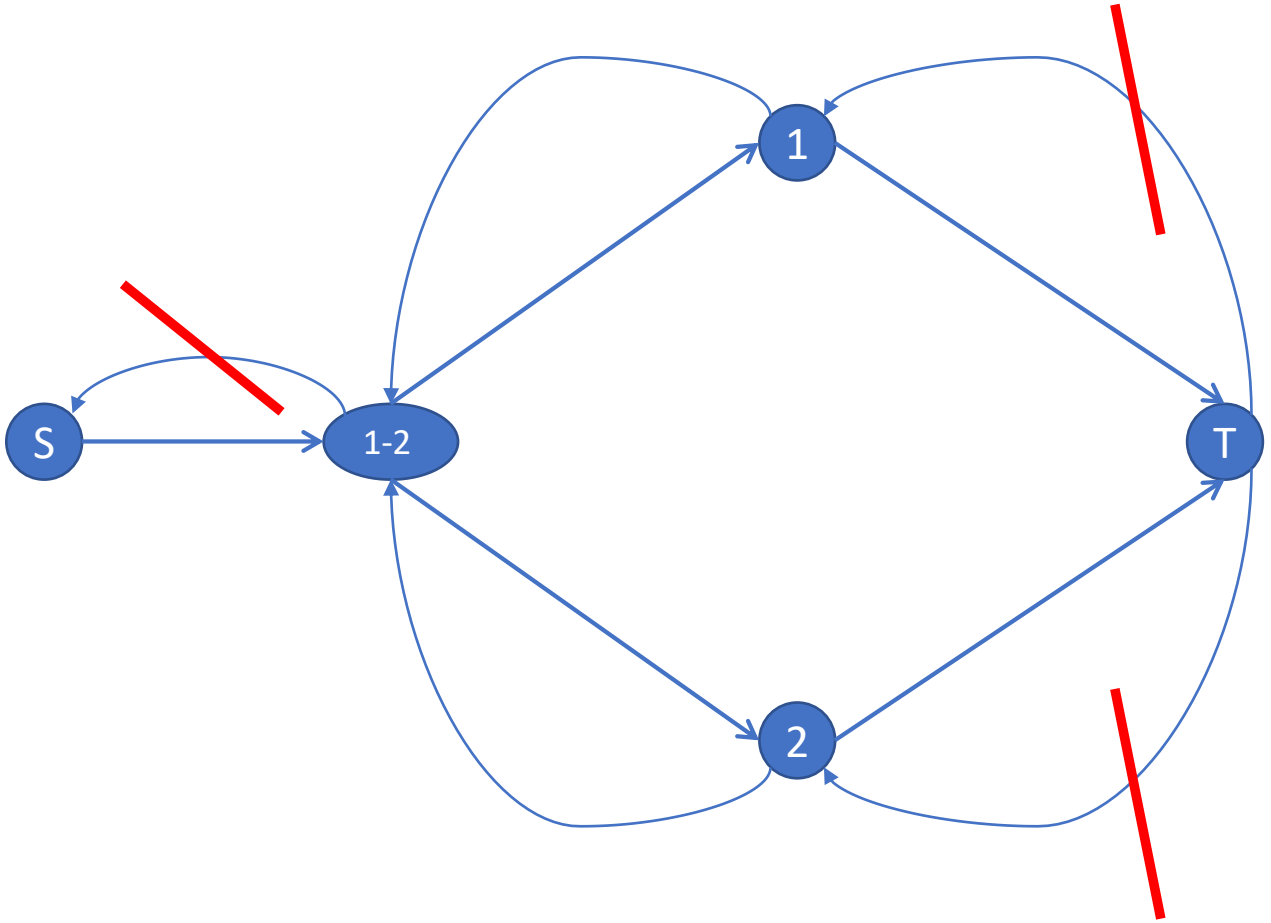


n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2
while(low < high):
mid = $\frac{low+high}{2}$ = 1
forall edges e of Graph from
Stadt-Knoten to Sink-Knoten do:
e.setCapacity(mid)
maxFlow = graph.edmondsKarp()



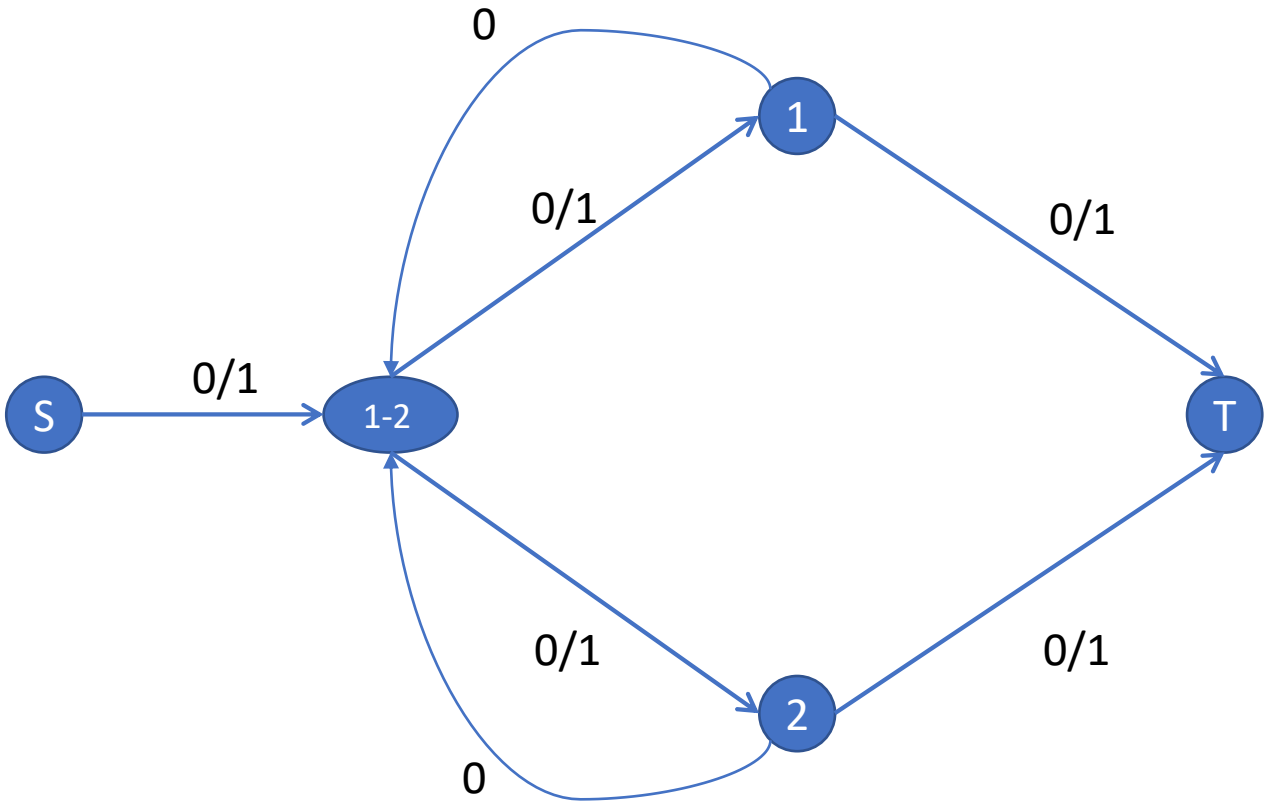
Einschub:



n = |Städte| = 2
m = | Straßen| = 1

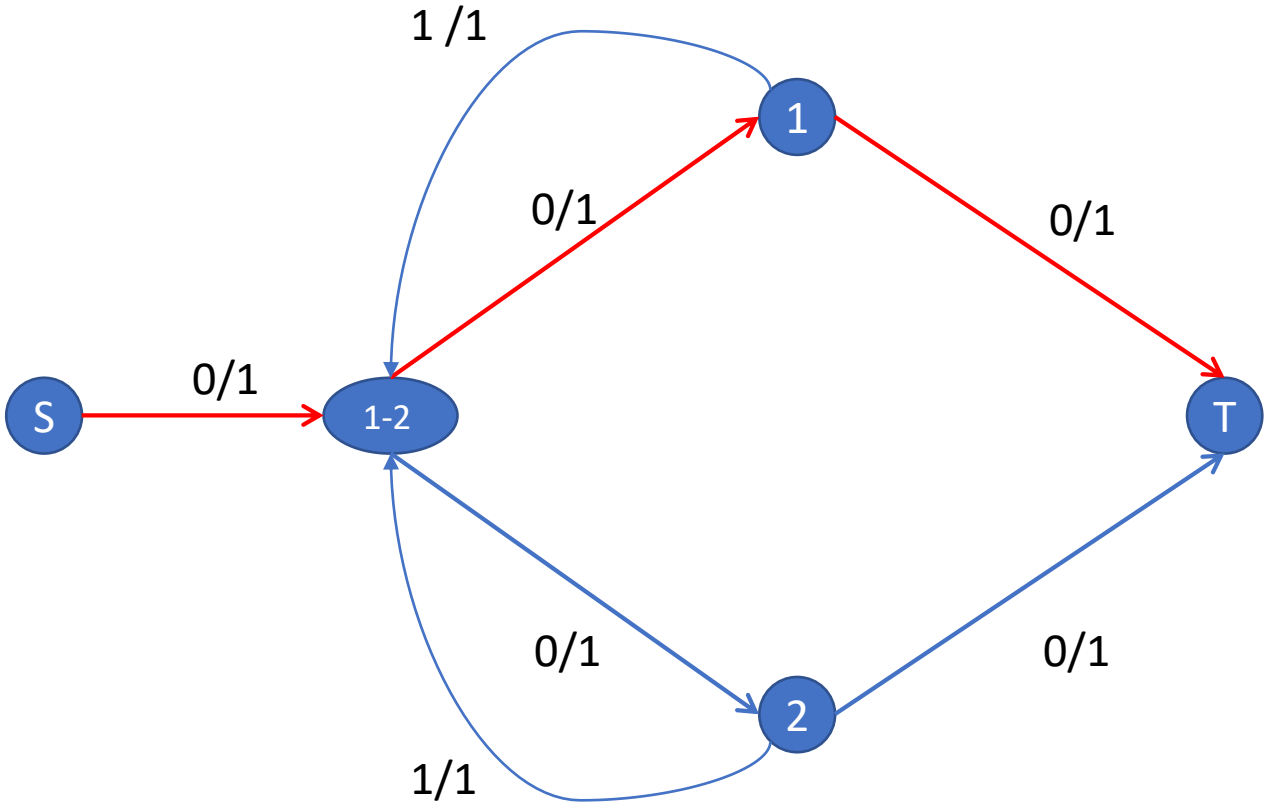
low = 0
high = n = 2
while(low < high):
mid = $\frac{low+high}{2}$ = 1
forall edges e of Graph from
Stadt-Knoten to Sink-Knoten do:
e.setCapacity(mid)
maxFlow = graph.edmondsKarp()

Residualgraph:



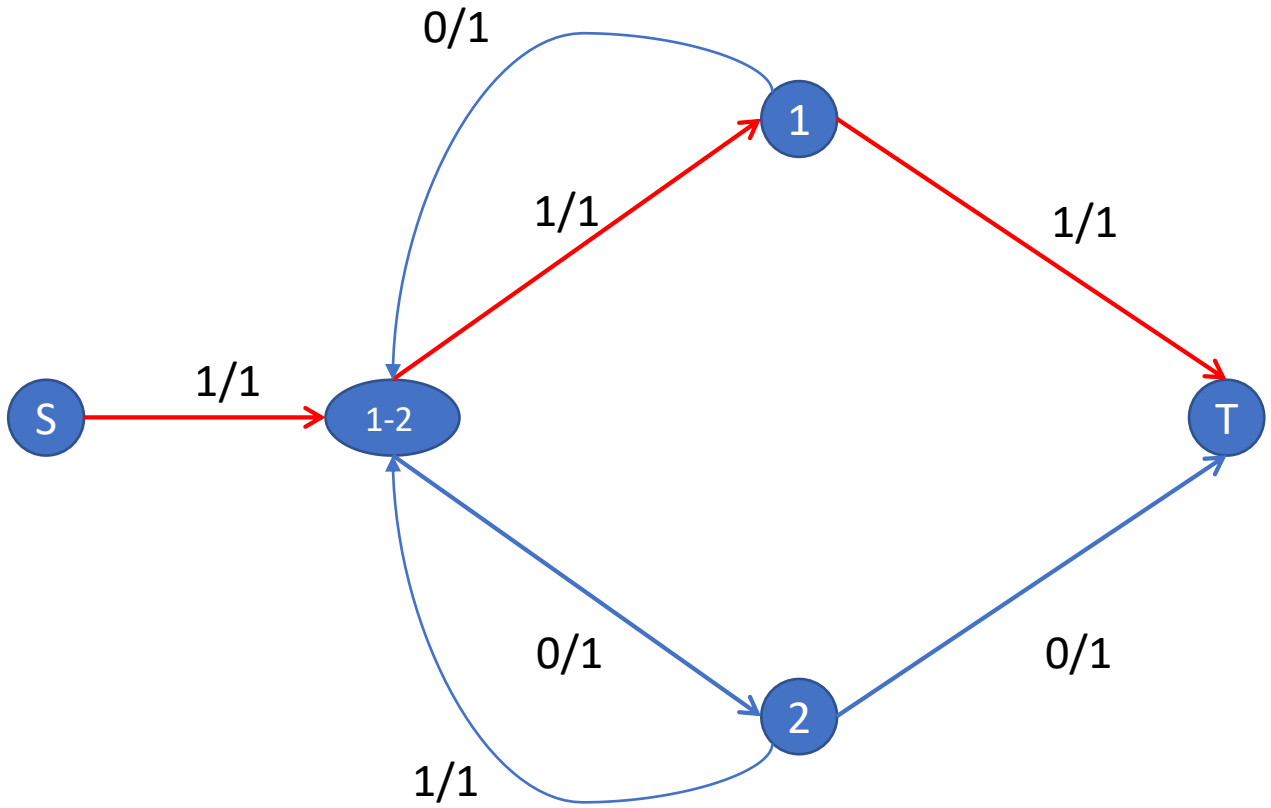
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2
while(low < high):
mid = $\frac{low+high}{2}$ = 1
forall edges e of Graph from
Stadt-Knoten to Sink-Knoten do:
e.setCapacity(mid)
maxFlow = graph.edmondsKarp()



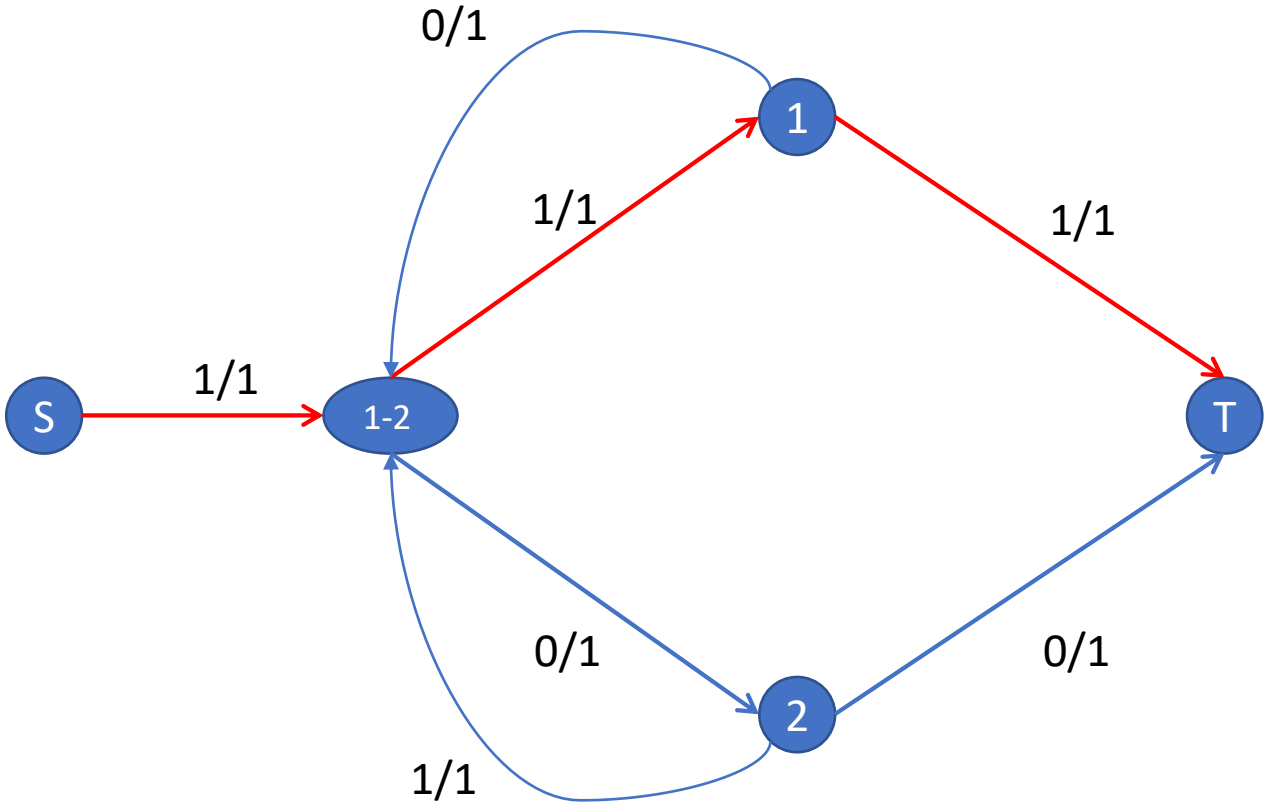
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2
while(low < high):
mid = $\frac{low+high}{2}$ = 1
forall edges e of Graph from
Stadt-Knoten to Sink-Knoten do:
e.setCapacity(mid)
maxFlow = graph.edmondsKarp()



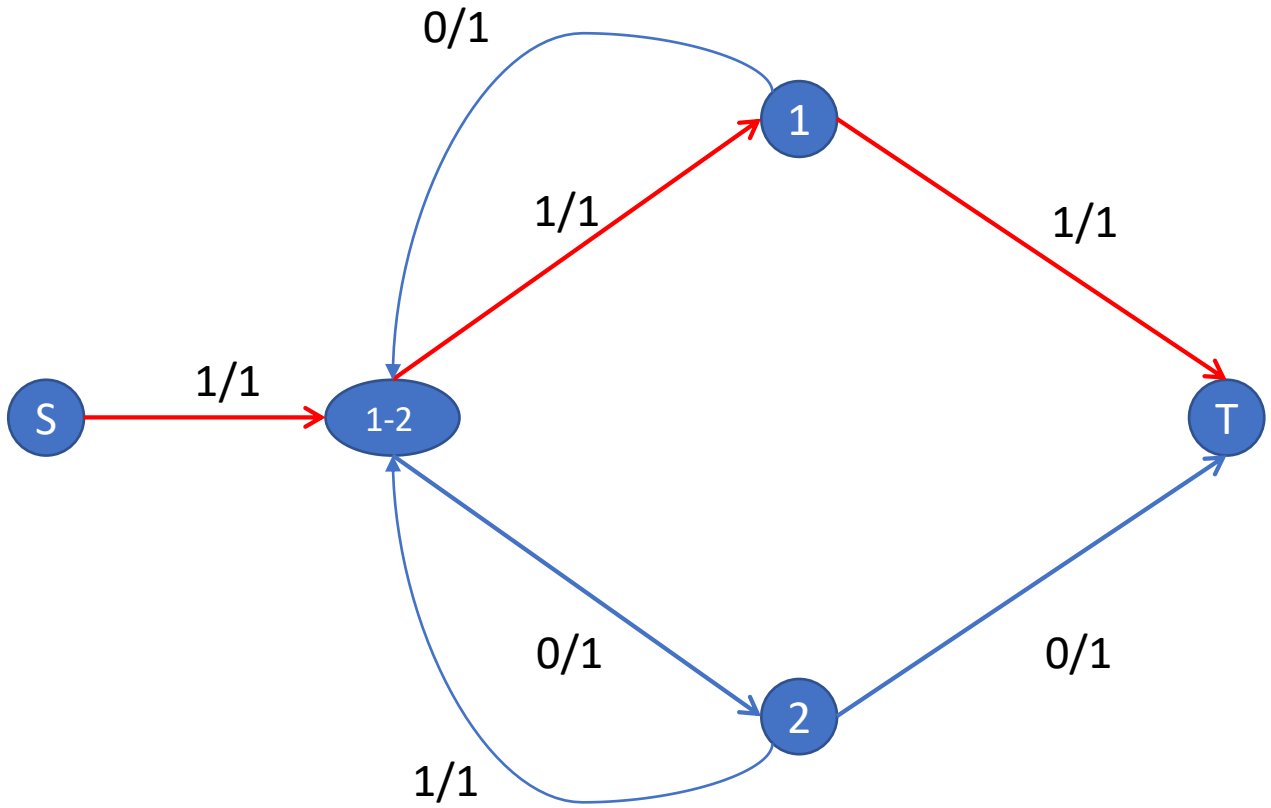
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2
while(low < high):
mid = $\frac{low+high}{2}$ = 1
forall edges e of Graph from
Stadt-Knoten to Sink-Knoten do:
e.setCapacity(mid)
maxFlow = graph.edmondsKarp() = 1



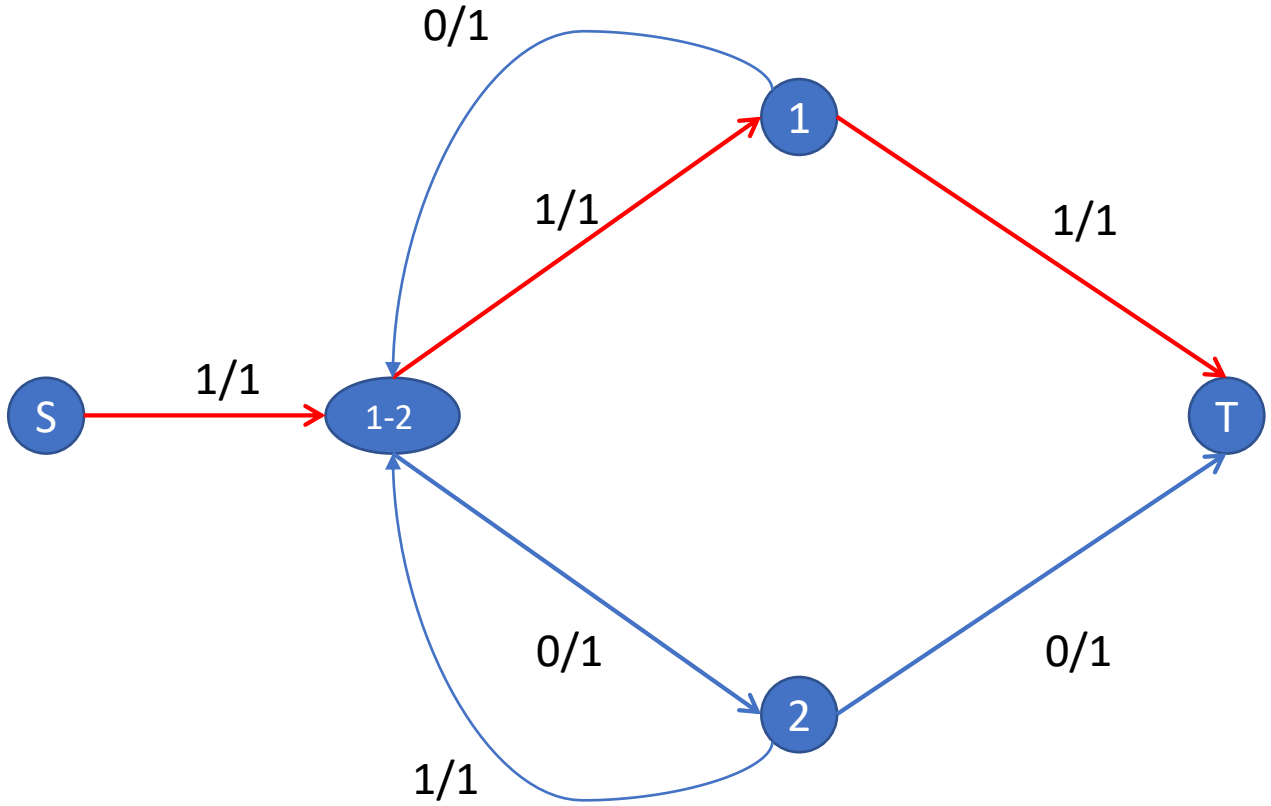
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = n = 2
while(low < high):
    mid = $\frac{low+high}{2}$ = 1
    forall edges e of Graph from
    Stadt-Knoten to Sink-Knoten do:
        e.setCapacity(mid)
    maxFlow = graph.edmondsKarp() = 1
    if(maxFlow == m):
        high = mid
    else:
        low = mid + 1



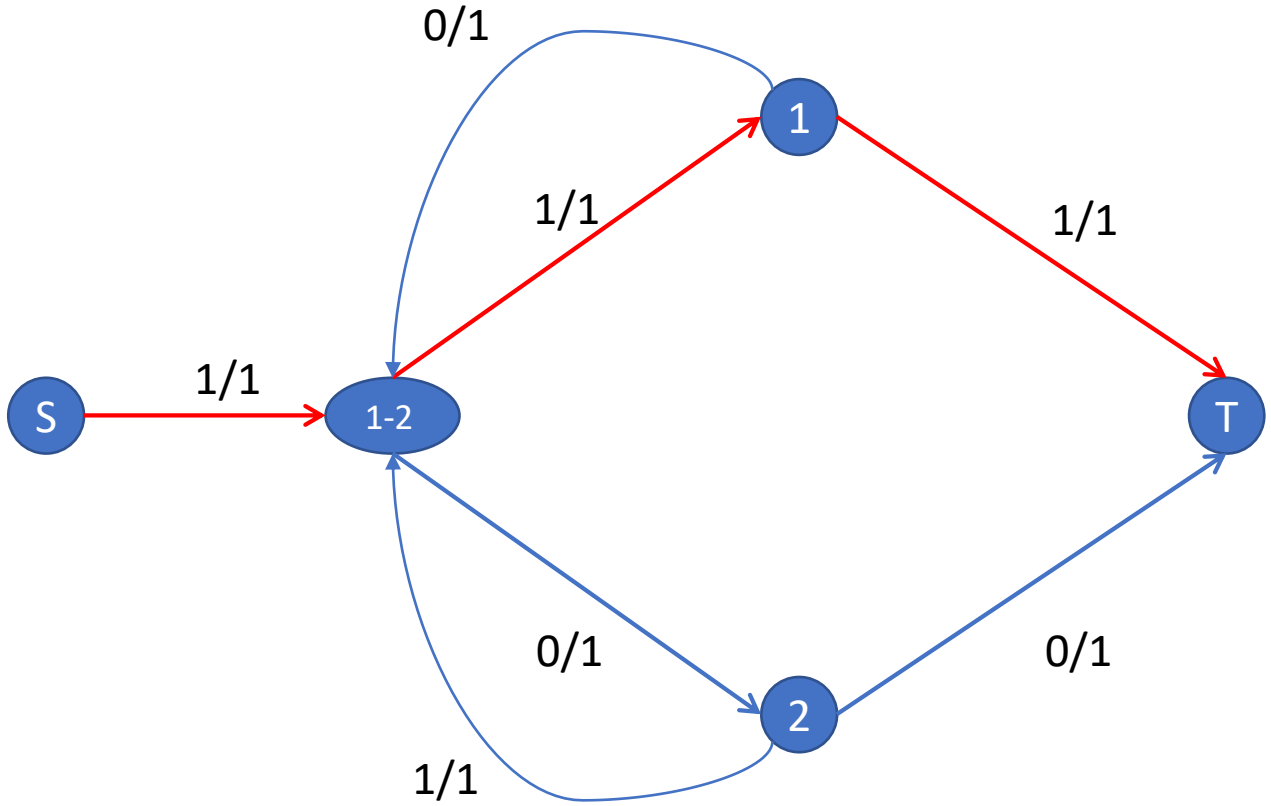
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = 1
while(low < high):
    mid = $\frac{low+high}{2}$ = 1
    forall edges e of Graph from
    Stadt-Knoten to Sink-Knoten do:
        e.setCapacity(mid)
    maxFlow = graph.edmondsKarp() = 1
    if(maxFlow == m):
        high = mid
    else:
        low = mid + 1



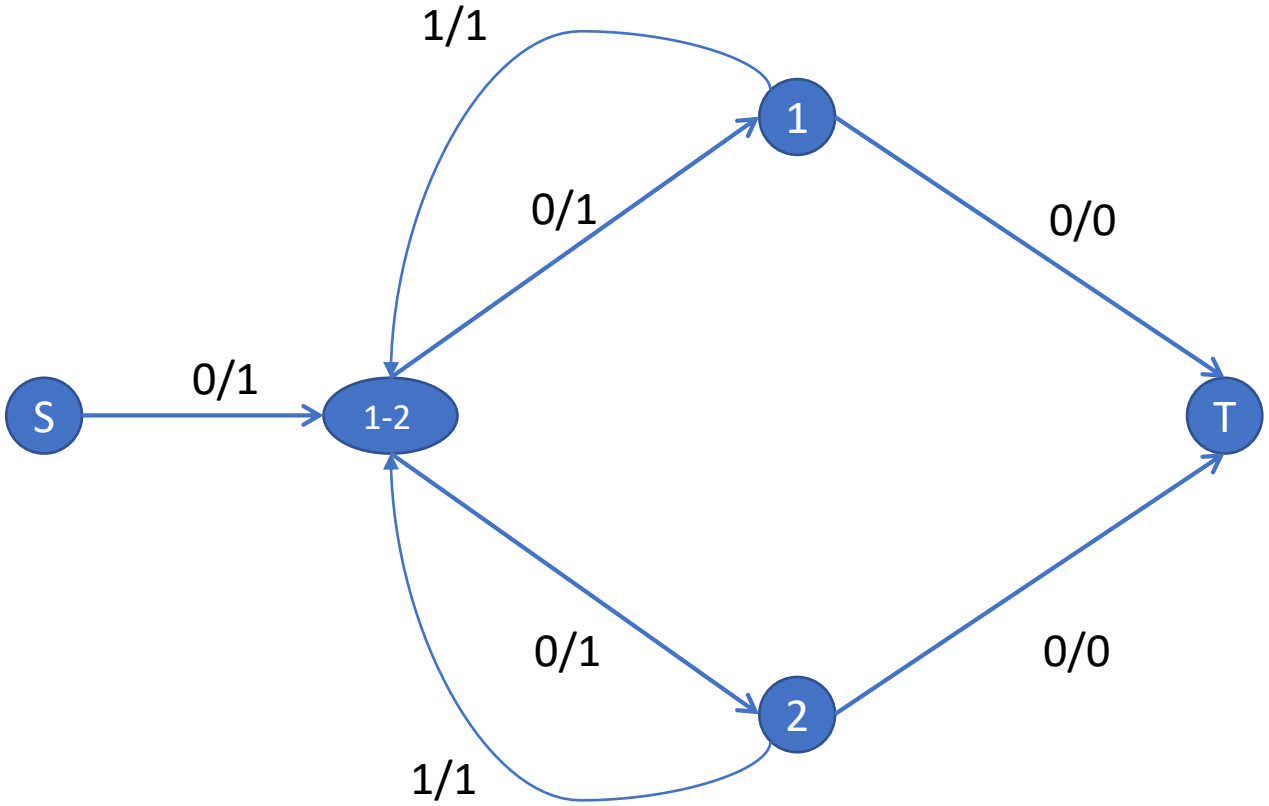
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = 1
while(low < high):
    mid = $\frac{low+high}{2}$ = 0
    forall edges e of Graph from
    Stadt-Knoten to Sink-Knoten do:
        e.setCapacity(mid)
    maxFlow = graph.edmondsKarp() = 1
    if(maxFlow == m):
        high = mid
    else:
        low = mid + 1



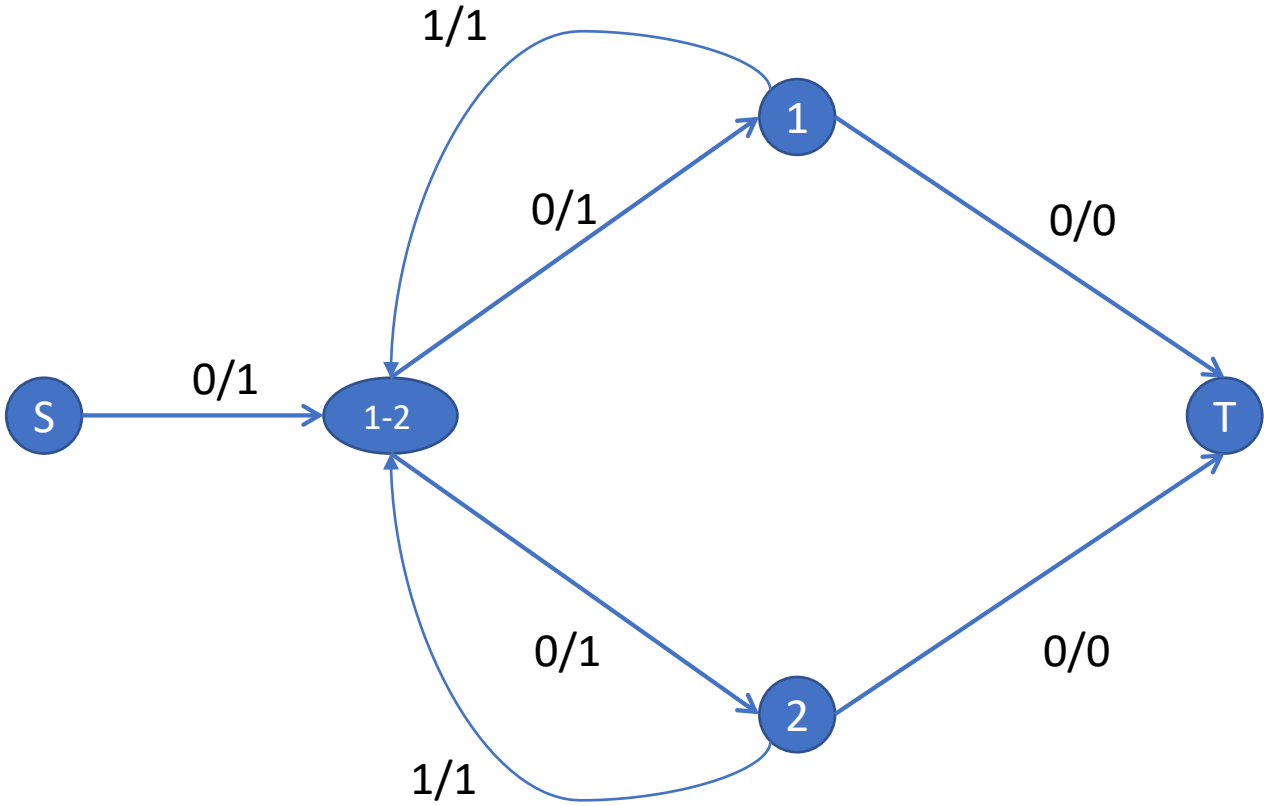
n = |Städte| = 2
m = | Straßen| = 1

low = 0
high = 1
while(low < high):
    mid = $\frac{low+high}{2}$ = 0
    forall edges e of Graph from
    Stadt-Knoten to Sink-Knoten do:
        e.setCapacity(mid)
    maxFlow = graph.edmondsKarp() = 1
    if(maxFlow == m):
        high = mid
    else:
        low = mid + 1



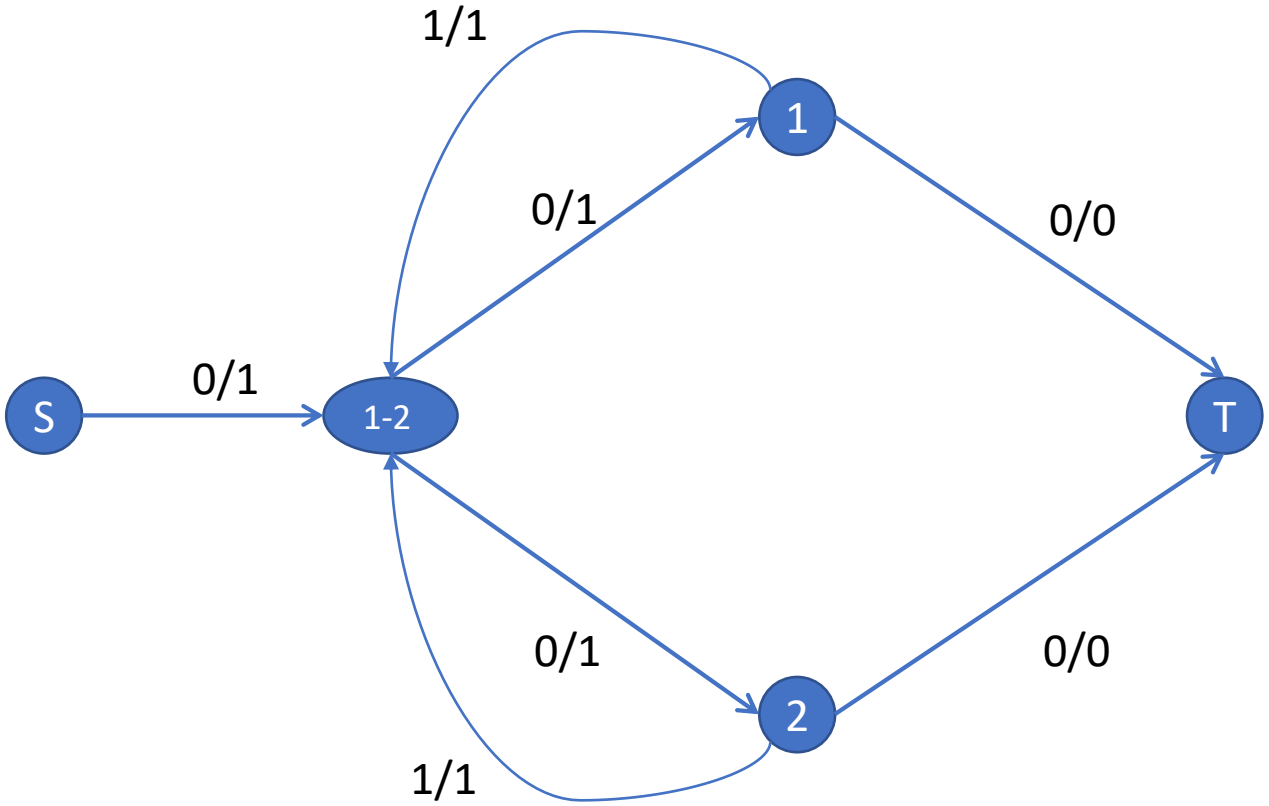
n = |Städte| = 2
m = | Straßen| = 1

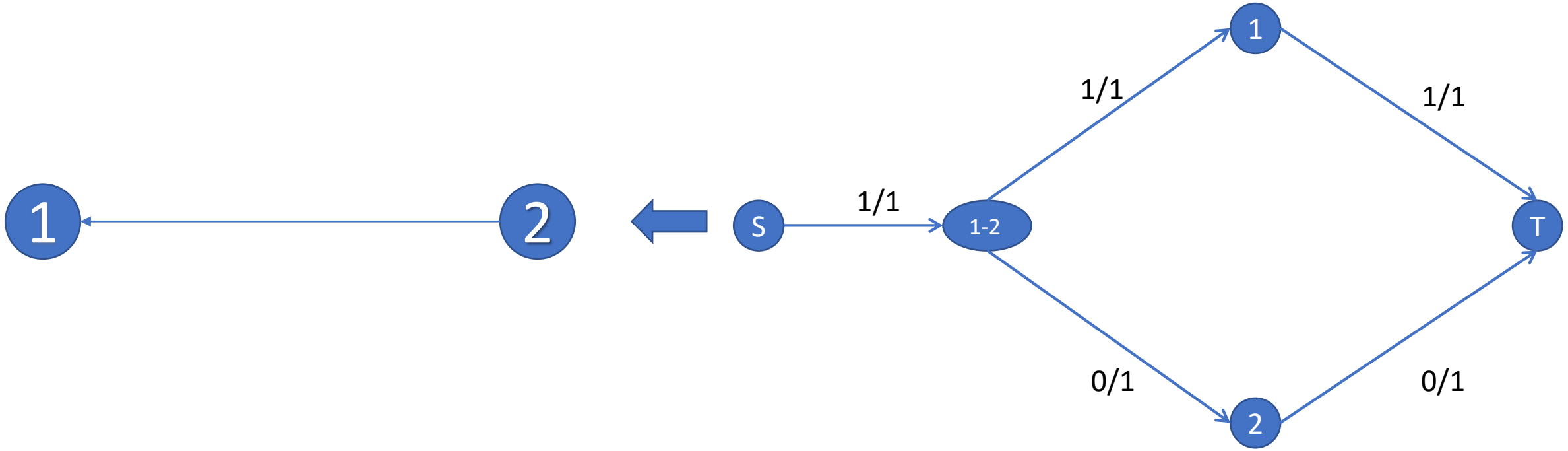
low = 0
high = 1
while(low < high):
    mid = $\frac{low+high}{2}$ = 0
    forall edges e of Graph from
    Stadt-Knoten to Sink-Knoten do:
        e.setCapacity(mid)
    maxFlow = graph.edmondsKarp() = 0
    if(maxFlow == m):
        high = mid
    else:
        low = mid + 1



n = |Städte| = 2
m = | Straßen| = 1

low = 1
high = 1
while(low < high):
    mid = $\frac{low+high}{2}$ = 0
    forall edges e of Graph from
    Stadt-Knoten to Sink-Knoten do:
        e.setCapacity(mid)
    maxFlow = graph.edmondsKarp() = 0
    if(maxFlow == m):
        high = mid
    else:
        low = mid + 1





d ist in diesem Beispiel 1, da der kleineste Integer von maximalen Indegrees gleich 1 ist.

Implementier-Tipps

- Graphen Implementierung durch Adjazenzlisten effizient gestalten
- BFS findet kürzeste Wege in ungewichteten Graphen
- BFS in Edmonds-Karp kann abgebrochen werden sobald man beim Sink ankommt
- Residualkanten müssen nur zwischen Straßen-Knoten und Städte-Knoten gespeichert werden
- Residualgraph direkt aus dem Input erstellen und pro Iteration von d nur die Kapazitäten anpassen
- Warum ResidualFlow irrelevant sind?
  - Wenn wir eine Residualkante (v,u) auf einem S-T-Weg benutzen dann wird die Kapazität von u,v` aufgebraucht, somit ist es nicht mehr möglich, dass wir die Residualkante auf einem S-T-Weg noch einmal nutzen, da die Kapazität v-Sink aufgebraucht ist und u,v`

```
Edge[][] edges = new Edge[V][];

// TODO Store which roads connect to which cities in a map (Key: index of city)
HashMap<Integer,ArrayList<Integer>> citiesToRoads = new HashMap<>();
for (int i = 1; i < n+1; i++) {
    citiesToRoads.put(i,new ArrayList<>());
}
// Our source connects to each "road"
edges[source] = new Edge[m];
// iterating through our edges
for (int j = n + 1; j < V-1; j++) {
    // TODO Add edges with capacity 1 from source to roads
    // e.g. edges[0][?] = new Edge(0,j, 1,false);

    // TODO Read in all roads and crate 2 edges for each (edges connect to each adjacent city)
    String[] destinations = br.readLine().split( regex: " ");
    edges[j] = new Edge[2];

    // TODO store which roads connect to a city in citiesToRoads (to create residual Edges from cities to roads)
}
br.close();

//Iterating through all cities
for (int i=1; i<n+1; i++){
    ArrayList<Integer> roads = citiesToRoads.get(i);
    // Every "city" has one Edge to the sink
    // and roads.size() residual Edges for adjacent roads
    edges[i] = new Edge[1 + roads.size()];
    // TODO for each city create residual edges for each connecting road
}

Graph graph = new Graph(edges,n,m);
```

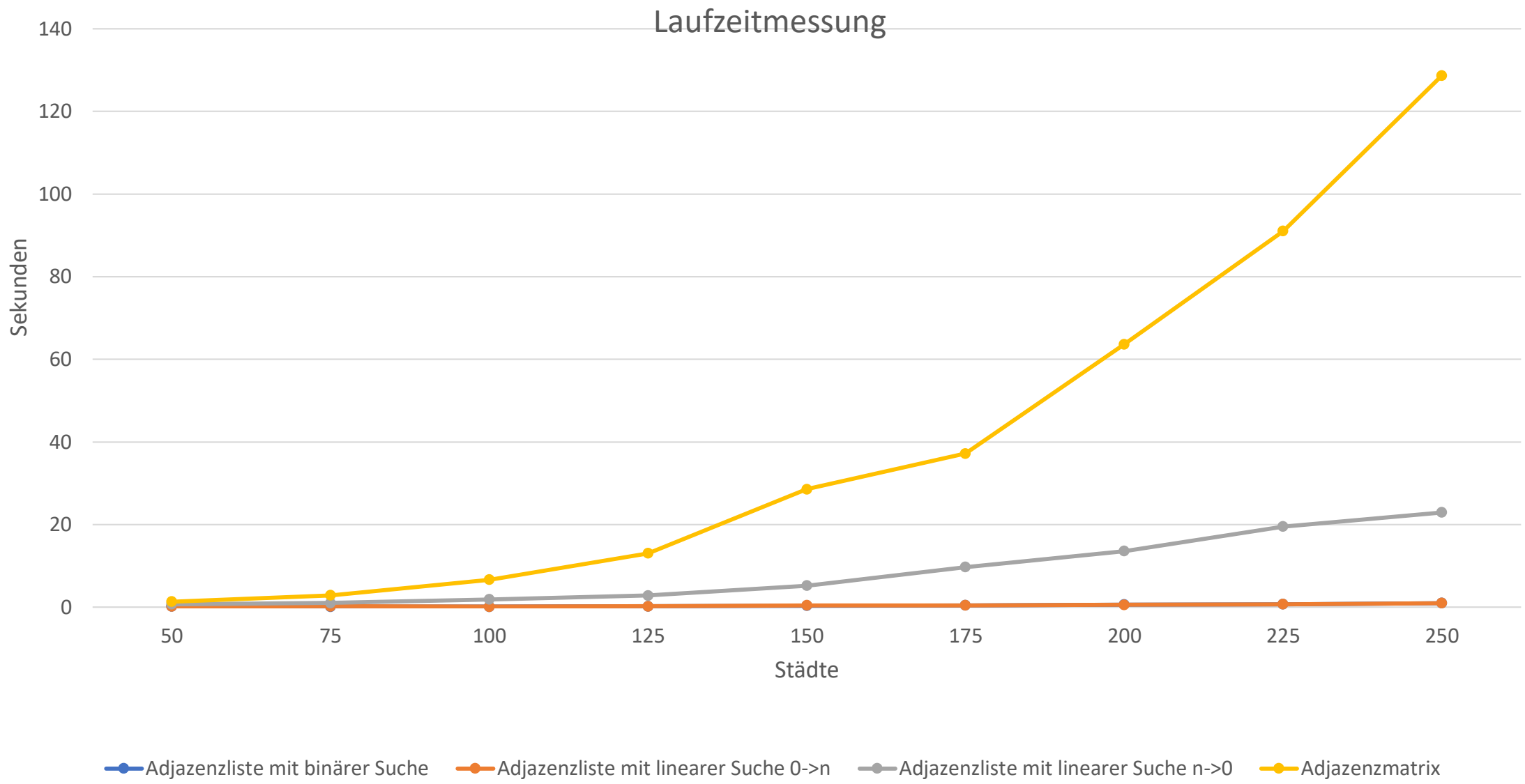
```
public static class Edge{
    int capacity;
    int flow;
    boolean res;
    int dest;
    int start;
```

```
// Our source connects to each "road"
edges[0] = new Edge[m];
```

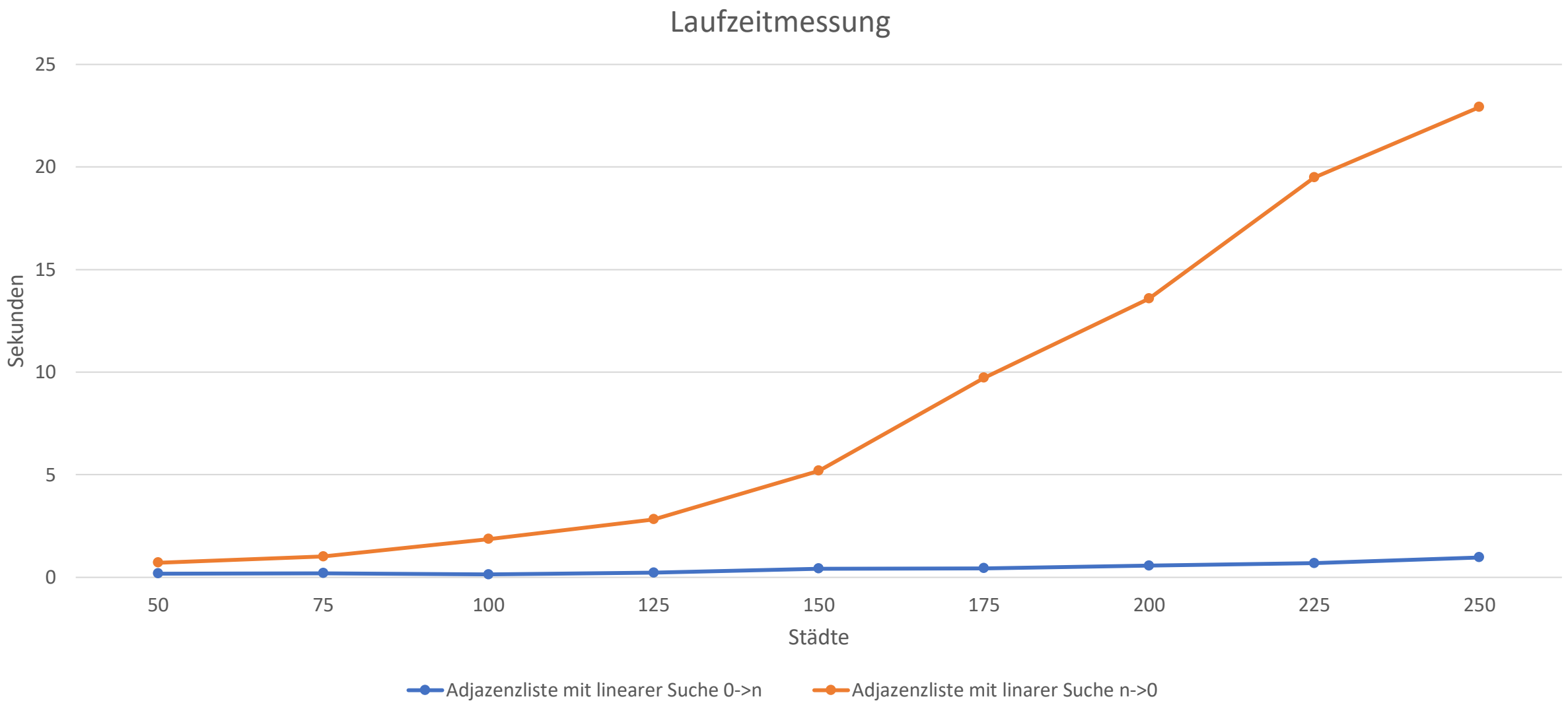
Laufzeit-Messungen

- Adjazenzmatrix vs Adjazenzliste
- BinarySearch vs Iteration (d=n, d=0)
- n = 25 + 25 bis 250
- m = 475 bis 2375
- d = 10
- Jede Stadt von 1 bis n/2 hat 19 Straßen die die Stadt mit einer Stadt aus (n/2,n) verbindet

Adjazenzliste vs. Adjazenzmatrix



Lineare Suche (n->0) vs. Lineare Suche (0->n)



Binäre Suche