Graphs Algorythms Algorythms

Uniform Cost Search (UCS)

```
UCS (S, T, G)                    # terminus (destination), start
    d[S] = 0                     # distances
    P[S] = S                     # previouses
    for v in V; v ≠ S:
        d[v] = ∞
        P[v] = -1
    found = false
    pq.push((S, 0))              # priority queue
    while pq not empty and not found:
        current = pq.pop()
        for neighbour in G.outbound(current):
            if d[current] + cost(current, neighbour) <
                < d[neighbour]:
                d[neighbour] = d[current] + cost(current, neighbour)
                P[neighbour] = current
                pq.push(neighbour, d[neighbour])
        if current == T:
            found = true
```

Bellman-Ford algorythm

- a.k.a. Bellman-Kalaba or Ford
- graph $G(V, E)$

```
algorythm (G, S)
    dist[S] = 0   <-- map of distances
    for v in V; v ≠ S:
        dist[v] = ∞
    changed = true
    while changed
        changed = false
        for (i, j) in E:
            if dist[j] > dist[i] + cost(i, j):
                dist[j] = dist[i] + cost(i, j)
                changed = true
```

- use a map of previous elements to traverse the graph
- to detect negative cycles: count how many times we iterate; if we iterate n times $\Rightarrow$ we have a neg. cycle
- on the k-th cycle we will find the min. cost walk of length k

Floyd - Warshall algorythm

- finds min cost walk between all vertices
- D ← distance matrix
- F ← first matrix; F[i][j] is the first vertex in the min cost walk between i and j

```
algorythm (G):
    for i in range(n):
        for j in range(n):
            if i == j
                D[i][j] = 0
                F[i][j] = j
            else
                if (i,j) in E
                    D[i][j] = cost(i,j)
                    F[i][j] = j
                else
                    D[i][j] = ∞
                    F[i][j] = NULL
```

```
for k in range(n):
    for i in range(n):
        for j in range(n):
            if D[i][j] > D[i][k] + D[k][j]
                D[i][j] = D[i][k] + D[k][j]
                F[i][j] = F[i][k]
```

Best First Search

- class of algorythms
- generalization of UCS
- the priority of a vertex is defined by a function $f(v)$
- ~~US~~ ~~UCS~~ UCS: $f(v)$ = cost of the walk from start to v
- Greedy: $f(v)$ = the heuristic
- A*: $f(v)$ = cost + heuristic

~~Kruskall's algorythm~~

Find minimum spanning subtree:

Kruskal

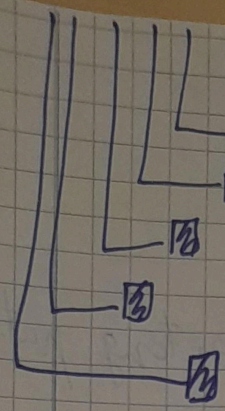
```
algorythm (G):
    for all v in V:
        Make Set (v)
    - sort all edges from E and put them in the queue Q

    while Q not empty:
        (u,v) = Q.pop()
        if Set(u) != Set(v)
            conjoin the two sets
            add (u,v) to the final set of edges
```

- starts with a forest
- keeps adding the minimum cost edge that does not form a cycle

Prim's algorythm

```
algorythm (G)
    -pick S a random vertex from V
    pq ← priority queue
    prev ← dictionary / map
    weights ← dictionary / map  -> weights[n] = cost(prev[n], n)
    edges = []
    for v in V:
        prev[v] = -1
        weights[v] = ∞
    visited ← ~~set~~ set
    for n in G.outbound(S):
        weights[n] = cost(S, n)
        prev[n] = S
        pq.push(n, weights[n])
    while pq not empty
        current = pq.pop()
        -if current not in visited
            edges add (prev[current], current)
            -for n in G.outbound(current)
                -if weights[n] > cost(current, n) and n not
                in visited:
                    weights[n] = cost(current, n)
                    prev[n] = current
```



Topological sorting

- add all vertices from a directed acyclic graph (DAG) in a list s.t. $\forall (u,v) \in E$, u is before v in the list

```
topological_sorting(G):
    sorted = []  <- empty list
    counts <- dictionary
    queue <- queue

    for v in V:
        counts[v] = g.in_degree(v)
        if counts[v] = 0:
            queue.push(v)

    while queue not empty
        current = queue.pop()
        sorted.add[current]
        for n in g.G.outbounds(current):
            counts[n] = counts[n]-1
            if counts[n]==0:
                queue.push(n)

    return sorted
```

Augmenting path

Normal & Regular graphs Max. matchings

- let M be a matching and AP an augmenting path

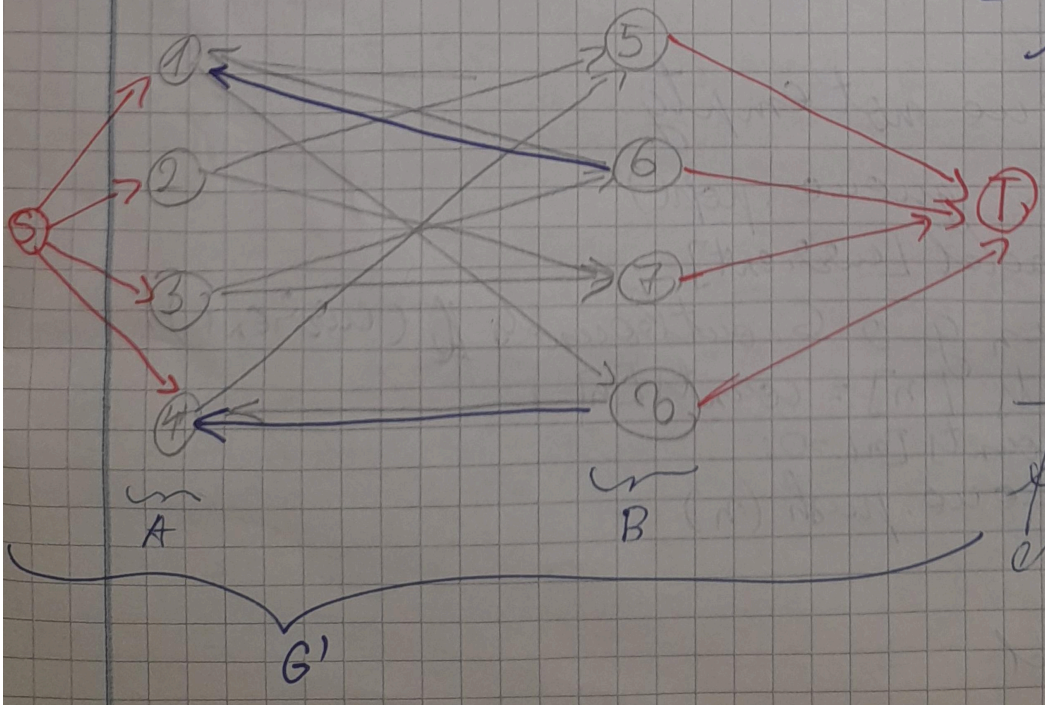
$M \oplus AP = M \setminus AP \cup AP \setminus M$

$\Longrightarrow$ another matching, with size increased by 1

```
max_matching (G):
    M ← matching containing one random edge
    repeat:
        • find augmenting path AP
        • if ∄ AP => M max. matching
          else
          M ← M ⊕ AP
    while M not max
```

Find longest AP in bipartite graph

- G(AUB, E) undirected bipartite graph



- S, T are help form an auxiliary directed graph G'

bfs_augm_path (G, M) ← matching of G

- construct G' a directed graph s.t. all edges in M are oriented B to A and all edges in E\M are oriented A to B; add S and T s.t.: $S \Rightarrow A$, $B \Rightarrow T$
- run BFS from S to T and find the longest path
- if not found $\Rightarrow \nexists$ AP

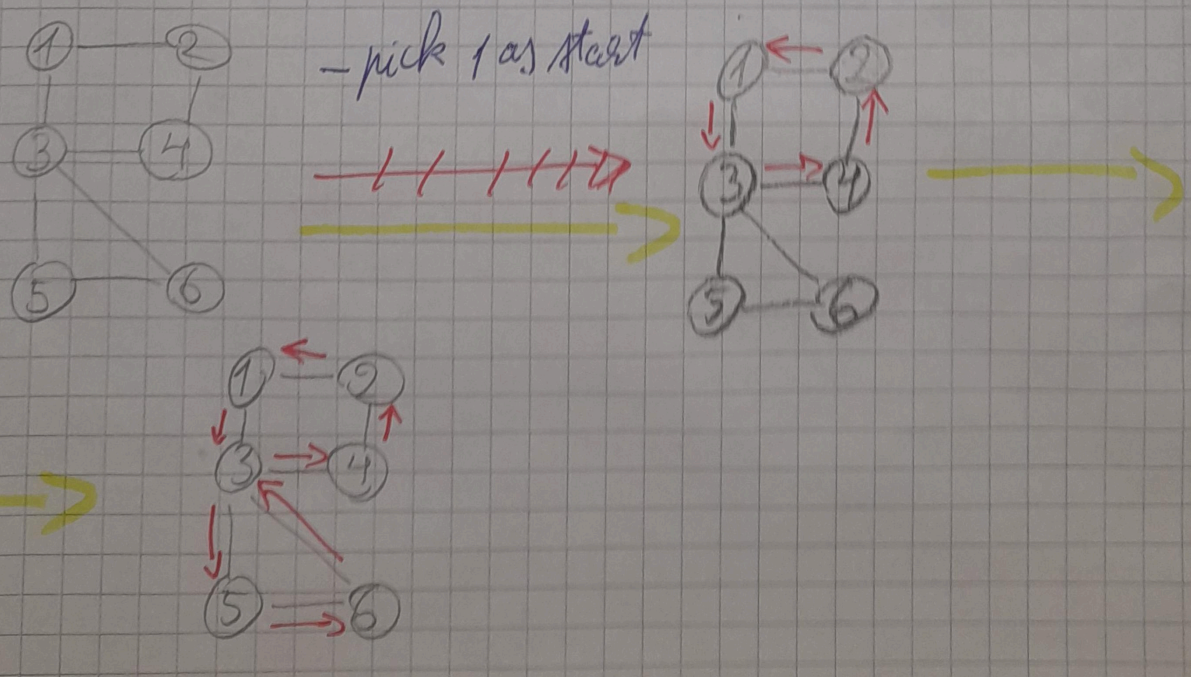
  else

  AP = * said longest path without S and T

Hierholzer algorythm to find an Eulerian circuit

1) start from random vertex
2) follow unused edges until we reach the initial vertex again
3) find a vertex with unused incident edges
4) from that vertex follow unused edges until you find it again
5) add the new circuit to the old one
6) repeat 3-5 until all edges are used

e.g.



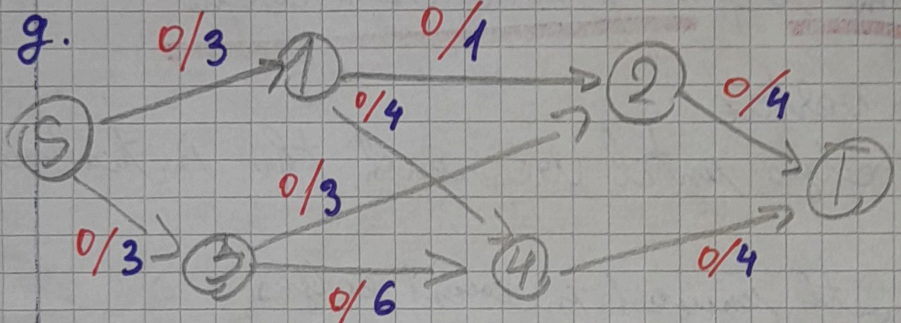
Ford-Fulkerson algorythm

- finds max flow

1) flow starts at 0 for all edges
2) compute N_f
3) find AP
4) if found, update N_f:
   a) add $c_f(AP)$ to the flow of the forward edges in N
   b) subtract -11- of the backward edges
   c) return to 2)

e.g.

N:

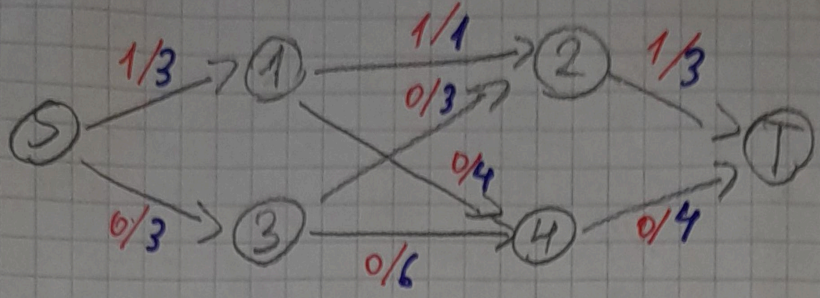


N_f: -11-

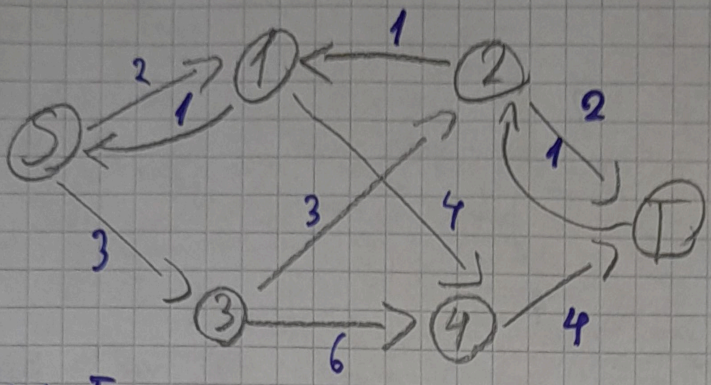
P: S, 1, 2, T

$c_f(P) = 1$

N:



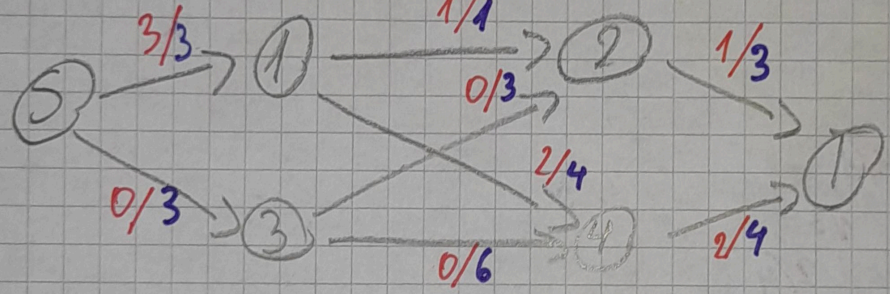
N_f:



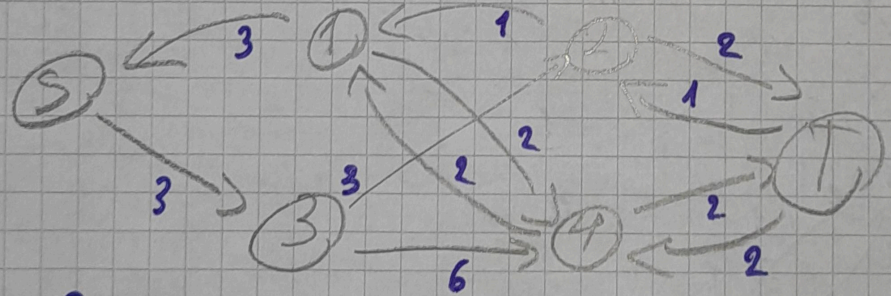
P_2: S, 1, 4, T

$C_f(P_2) = 2$

N:



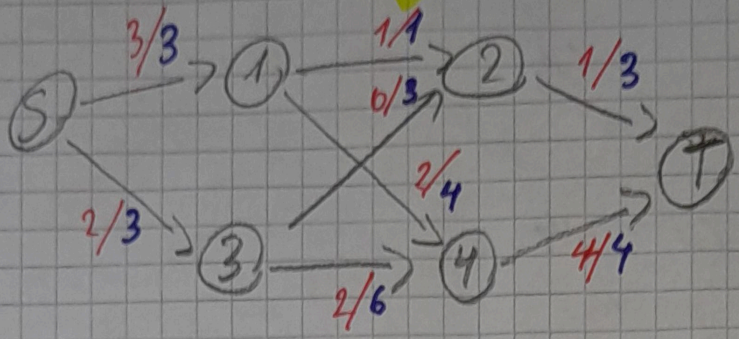
N_f:



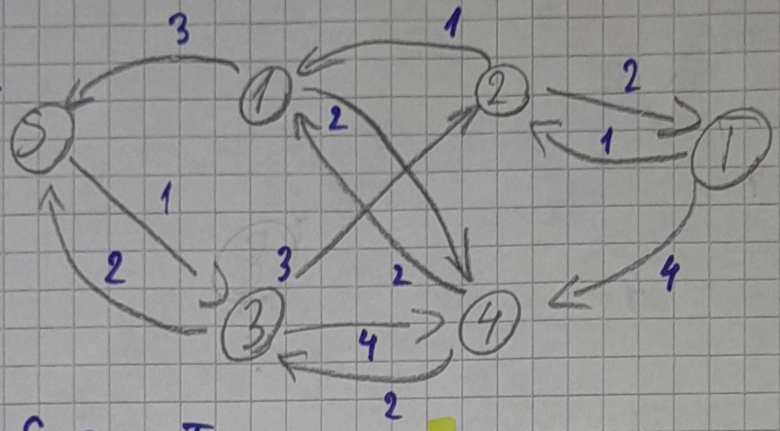
P_3: S, 3, 4, T

$C_f(P_3) = 2$

N:



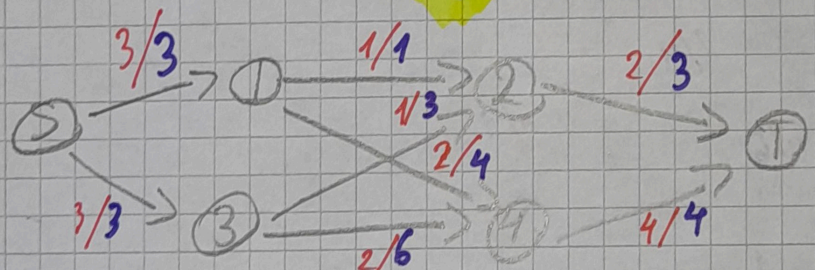
N_f:



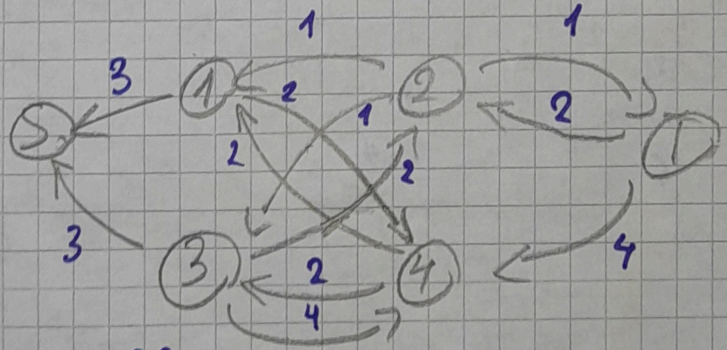
P_4: S, 3, 4, T

$C_f(P_4) = 1$

N:



N_E: N_f:



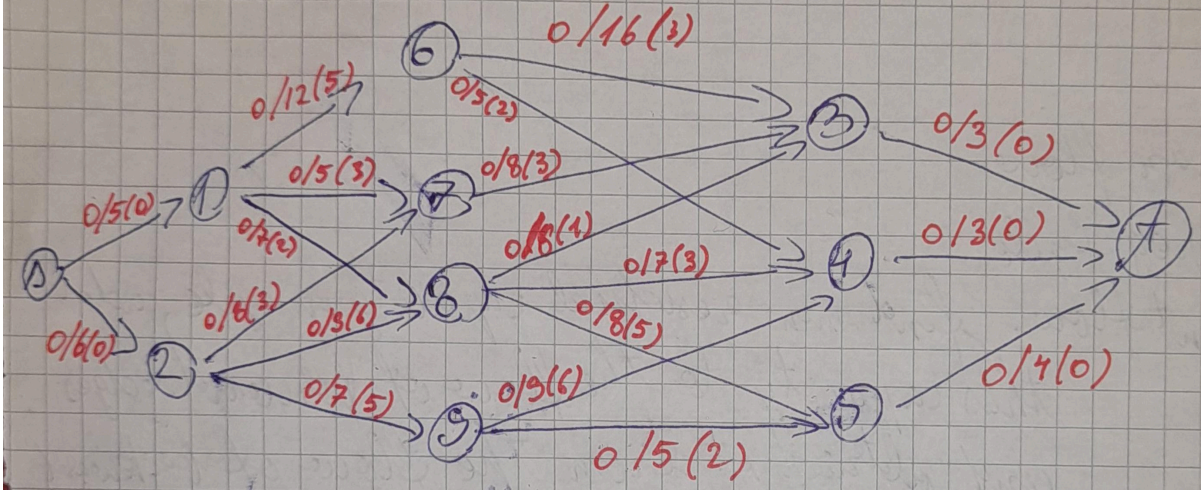
$\exists P_5 \Rightarrow$ max flow

Complexity:

- finding an AP with BFS: $O(n+m)$
- worst case for Ford-Fulkerson: $O(|f| \cdot (n+m))$
  - a lot of bottlenecks $\Rightarrow$ $C_f(P) = 1$ each time

Max flow of min. cost

notation: flow / capacity (cost)



- solution: find the min. cost path in the residual graph
- weights of edges in residual graph:
  - forward edges: the original cost if flow is 0, 0 otherwise
  - backward edges: cost 0 for all except those with minimum residual capacity, for which the weight is $(-1) \cdot$ (original weight)

weight of flow: $W(f) = \sum_{v_1, v_2 \in E,\ f(v_1, v_2) > 0} w(v_1, v_2)$

Min cost max flow

- transporting one unit of flow over an edge has the cost of said edge

$$W(f) = \sum_{(v_1, v_2) \in E} f(v_1, v_2) \cdot w(v_1, v_2)$$

~~We find a max. flow using Ford-Fulkerson, then we improve the cost~~

Weights in residual network:

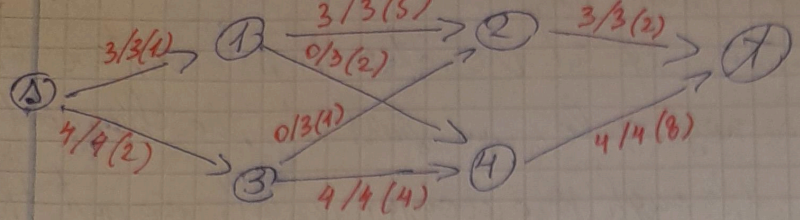
- forward ~~eg~~ edges: original cost
- backward ~~eg~~ edges: (-1)·(original cost)

Steps

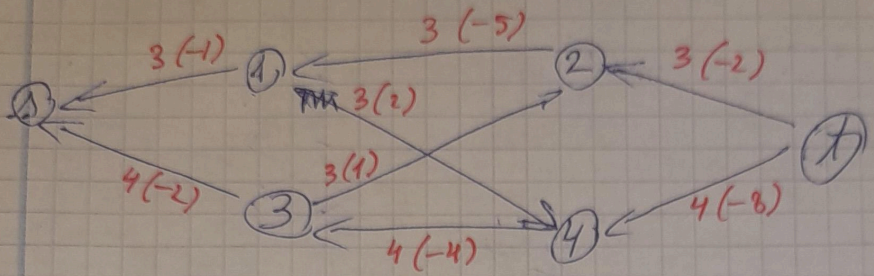
1) find max. flow
2) find neg. cycle
3) augment flow: find min. residual capacity in cycle, add this capacity to the flow of forward edges and subtract it from the flow of backward edges
4) if no cycle found => flow is optimal

e.g.

N:

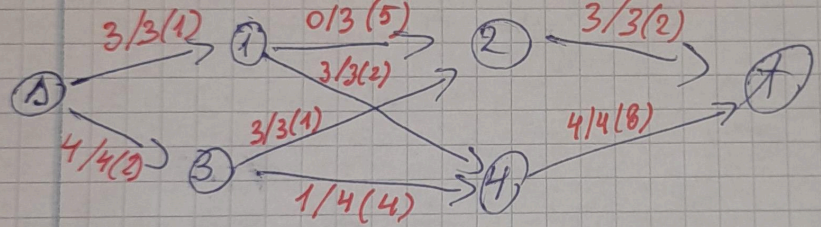


N_f:

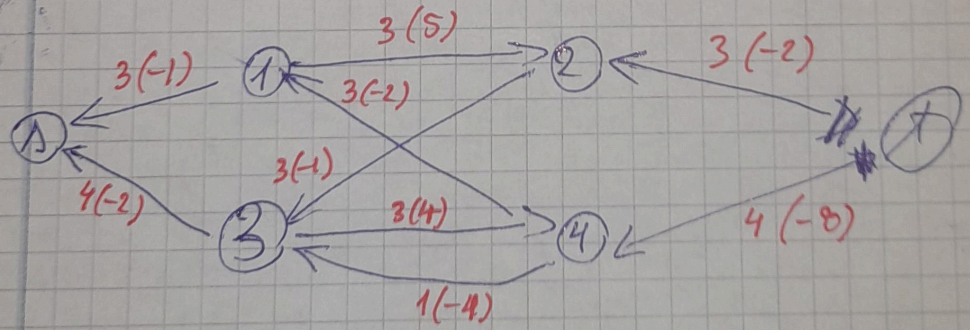


- negative cycle. 2, 1, 4, 3, 2
  min. capacity = 3

N:



N_f:



- no neg. cycles => optimal flow

Find neg. cycles using Bellman-Ford

- we run it n-times; if at iteration n ∃ a vertex that changes, we have a neg. cycle; we start at the changed vertex and go back through previouses until we get the cycle