Network Flow

CS31005: Algorithms-II
Autumn 2020
IIT Kharagpur

Network Flow

- Models the flow of items through a network
- Example
  - Transporting goods through the road/rail/air network
  - Flow of fluids (oil, water,..) through pumping stations and pipelines
  - Packet transfer in computer networks
  - Many others in a variety of fields…
- Has many different versions with wide practical applicability
- We will study the maximum flow problem

The Maximum Flow Problem

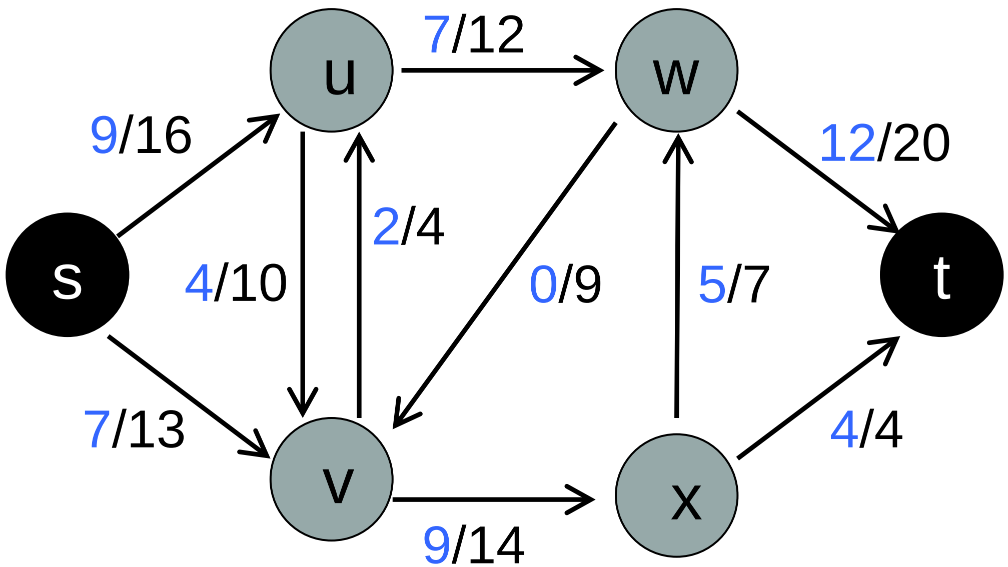
- Input: a directed graph $G = (V, E)$ with
  - Each edge $(u,v) \in E$ has a capacity $c(u, v) \geq 0$
  - Two distinguished vertices s (source) and t (sink)
- Output: Flow in G, a function $f: E \rightarrow \mathbb{R}$ such that
  - $0 \leq f(u,v) \leq c(u,v)$ for each (u,v) in E (capacity constraint)
  - $\sum_{u \in V, (u, v) \in E} f(u,v) = \sum_{w \in V, (v, w) \in E} f(v, w)$ for all v in $V \setminus \{s, t\}$ (flow conservation constraint)
- Easy to see that this means total flow leaving s must be the total flow entering t
- Flow satisfying the two constraints is called a feasible flow

- Value of the flow in the network

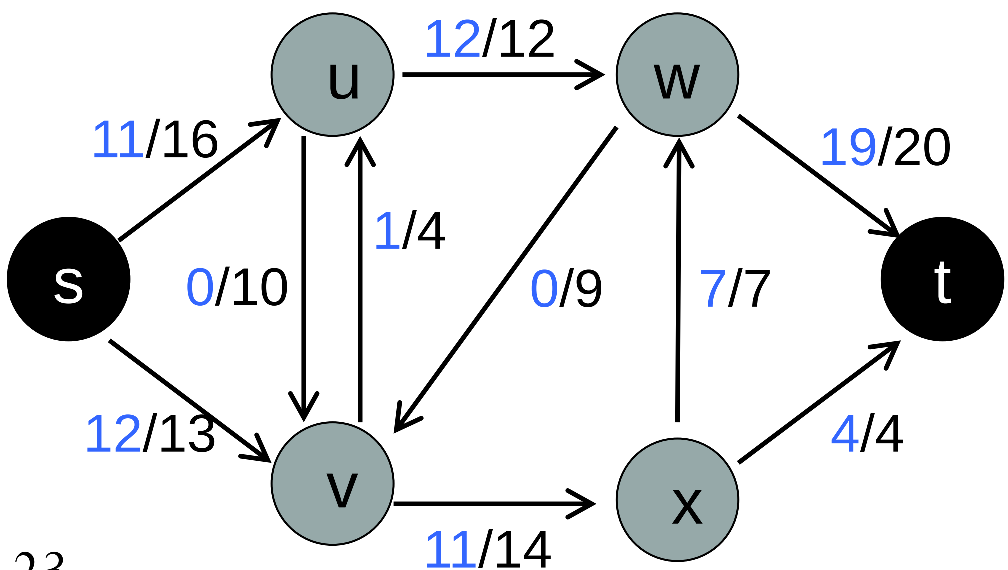
$$|f| = \sum_{u \in V, (s, u) \in E} f(s,u) = \sum_{u \in V, (u, t) \in E} f(u, t)$$

- Maximum Flow Problem: Find a feasible flow f such that the |f| is maximum among all possible feasible flows
- The assigned flow values on edges can model amount of goods in a transportation network, oil in a pipeline network, packets in a computer network along road/pipeline/link etc. to maximize the total amount of items moved from a source to a destination

Example



A feasible flow with $|f| = 16$



A maximum flow with $|f| = 23$

Algorithms for Maximum Flow

- Follows two broad approaches
  - The Ford-Fulkerson Method
    - Originally proposed by Ford and Fulkerson in 1956
    - Actually defines a method, the original paper did not specify any particular implementation of some steps
    - Many algorithms proposed later following the method, with specific implementations of steps
  - Preflow-Push Method
    - Presented by Andrew Goldberg and Robert Tarjan in 1986 (ACM STOC, later detailed journal version in JACM in 1988)
    - A totally different approach from the Ford-Fulkerson methods

Ford-Fulkerson Method

- Before starting the algorithm, we first give an equivalent modelling of the problem by
  - Extending the domain of capacity c and flow f to $V \times V$ (instead of keeping to E only)
  - Modifying the constraints appropriately

- Capacity c: $V \times V \rightarrow R$ such that $c(u,v) = 0$ if (u,v) not in E
- Flow $f : V \times V \rightarrow R$ satisfying:
  - Capacity constraint: For all u, v $\in V$, $f(u,v) \leq c(u,v)$
  - Skew symmetry: For all u, v $\in V$, $f(u,v) = -f(v,u)$
  - Flow conservation: For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u,v) = 0$

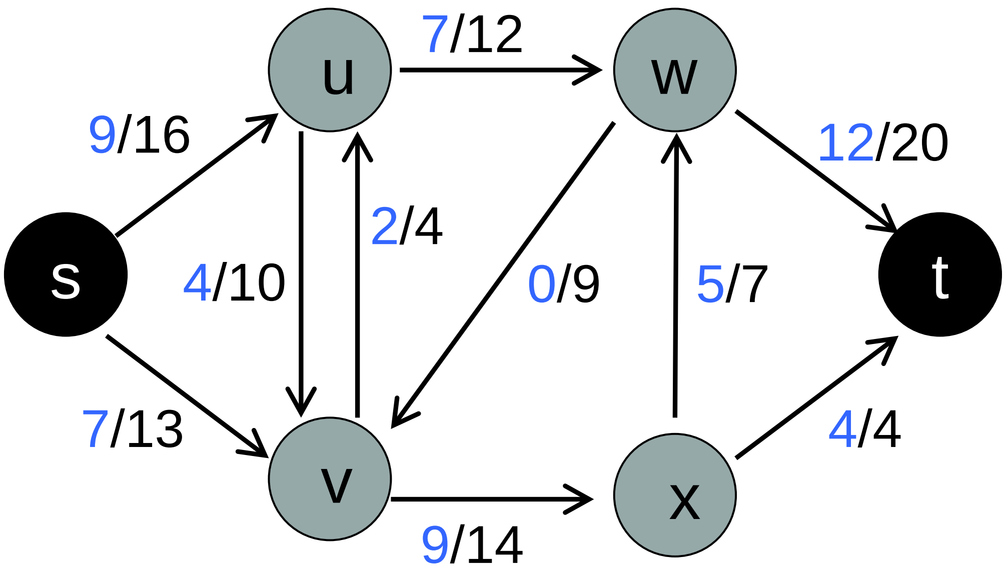
The value of the flow f is defined to be $|f| = \sum_{v \in V} f(s,v)$

The maximum flow problem is to find the flow with maximum value (same as before)

- What does this mean? Consider different possibilities for a pair (u,v)
  - None of the edges (u,v) or (v,u) exist
    - So $c(u,v) = c(v,u) = 0$
    - So $f(u,v) = f(v,u)$ must be 0 as otherwise capacity constraint and skew symmetry are violated
  - Only one of the edges exist (say (u,v))
    - So $c(u,v) \geq 0$ and $c(v,u) = 0$
    - If $f(u,v) = 0$, then $f(v,u) = 0$ (skew symmetry)
    - If $f(u,v) > 0$, then $f(v,u) < 0$ (skew symmetry)
    - If $f(u,v) < 0$ then $f(v,u) > 0$ (skew symmetry), But this violates capacity constraint for (v,u). So $f(u,v)$ cannot be negative

- Both the edges (u,v) and (v,u) exist
  - So c(u,v) ≥ 0 and c(v,u) ≥ 0
  - So seems like both f(u,v) and f(v,u) can be positive (by capacity constraint)
  - But that would break skew symmetry, so both cannot be positive
  - The way to think about it is to consider the "net flow"
    - If you ship 20 units from A to B and ship 5 units from B to A, the net flow into B is not 20, it is 20 – 5 = 15. Similarly the net flow into A is not 5, but (-20) + 5 = -15, indicating it is actually an outflow
- In general, for any two vertices u, v, if f(u,v) > 0, then f(v,u) must be < 0 (skew symmetry)

Example



$f(s, u) = 9, \quad f(u, s) = -9$

$f(s, v) = 7, \quad f(v, s) = -7$

$f(u,w) = 7, \quad f(w,u) = -7$

$f(u,v) = 4 - 2 = 2$

$f(v, u) = 2 - 4 = -2$

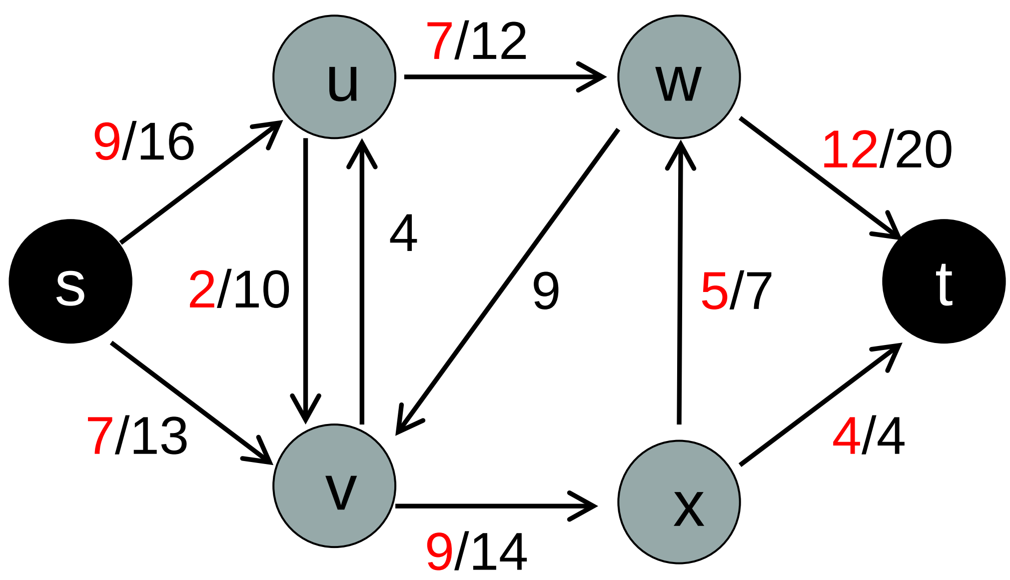
$f(v,x) = 9, \quad f(x,v) = -9$

$f(w,v) = 0, \quad f(v, w) = 0$

$f(u, x) = 0, \quad f(x, u) = 0$

similar for other pairs in $V \times V$

- With our new definition of flow, we will represent the graph to show f values on edges in red (not necessarily actual shipments)
- Also, we will only show positive f values on the edges of the graph
  - So for edges (v,u) and (w,v), we do not show the f values because f(v,u) = – 2 and f(w,v) = 0



- Did we lose anything from the earlier model?
  - For edges (u,v) and (v,u) (i.e for the case when edges exist in both direction between a pair of vertices), we are now representing only the net flow, not how exactly the net flow is achieved
    - For example, the net flow of 2 from u to v could have been achieved in different ways like "ship 6 units from u to v and 4 units from v to u", "ship 2 units from u to v and 0 units from v to u",.....
- So this model is not exactly equivalent to the model we had,
  - For the earlier model, actual shipments are the flow f
  - but ok as in practice as no need to ship in both directions
- If you have edge only in one direction, f will show the actual shipment

Residual Network

- Let f be a flow in a flow network $G = (V, E)$ with source s and sink t.
- Residual capacity of (u,v) = amount of additional flow that can be pushed from a node u to node v before exceeding the capacity c(u,v)

$$c_f(u, v) = c(u, v) - f(u, v)$$

- The residual graph of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

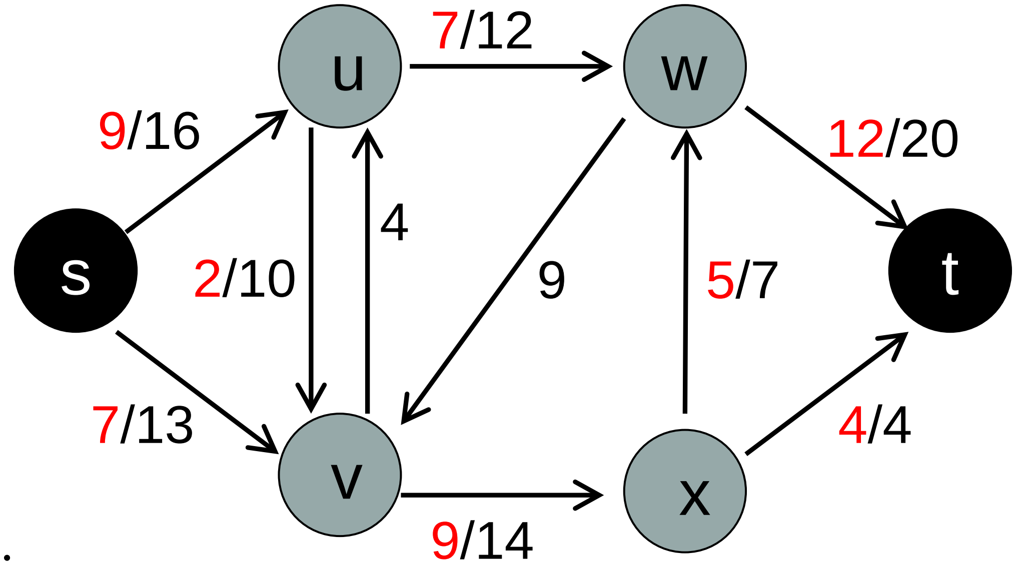
Edges of the residual graph are called residual edges, with capacity c_f

- Augmenting path: a simple path from source s to sink t in the residual graph G_f
- Residual capacity of an augmenting path p

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$$

$c_f(p)$ gives the maximum amount by which the flow on each edge in the path p can be increased

Example



- Residual capacities:

  $c_f(s,u) = 16 - 9 = 7$, $\qquad c_f(u,s) = 0 - (-9) = 9$

  $c_f(s,v) = 13 - 7 = 6$, $\qquad c_f(v,s) = 0 - (-7) = 7$

  $c_f(u,v) = 10 - 2 = 8$, $\qquad c_f(v, u) = 4 - (-2) = 6$

  $c_f(u,w) = 12 - 7 = 5$, $\qquad c_f(w, u) = 0 - (-7) = 5$

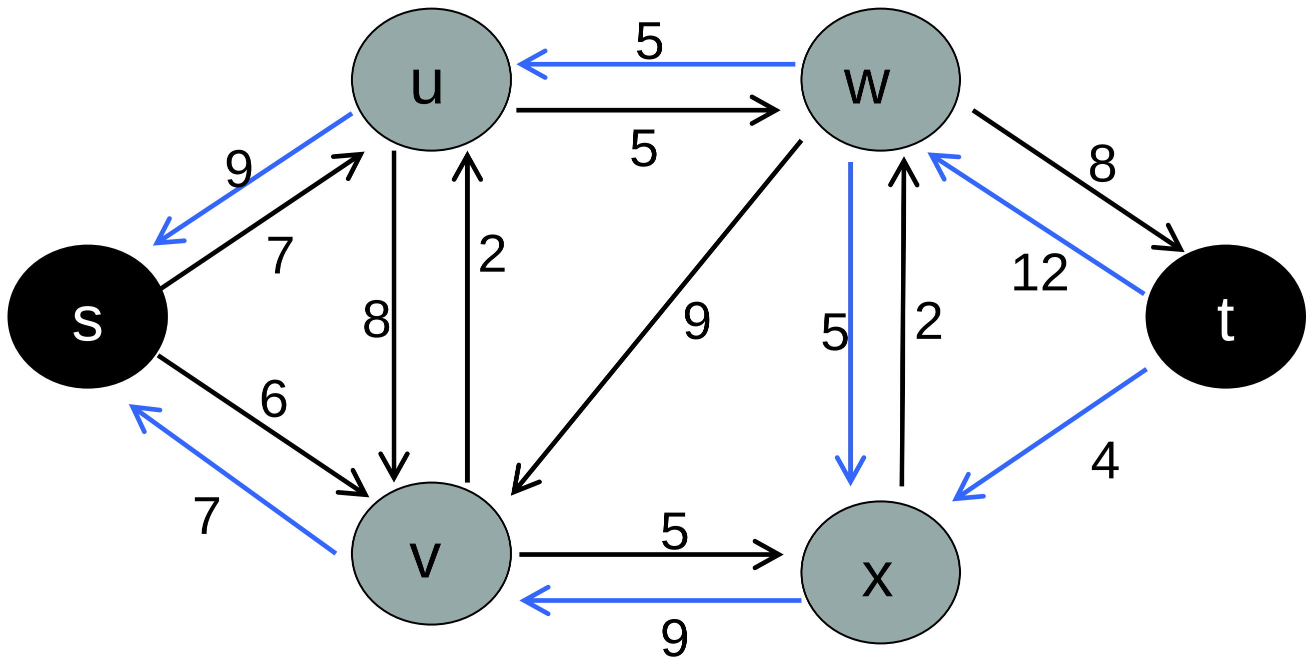
  $c_f(w,v) = 9 - 0 = 9$, $\qquad c_f(v, w) = 0 - 0 = 0$

  $c_f(x,t) = 4 - 4 = 0$, $\qquad c_f(t,x) = 0 - 0 = 0$

  and so on for the other pairs

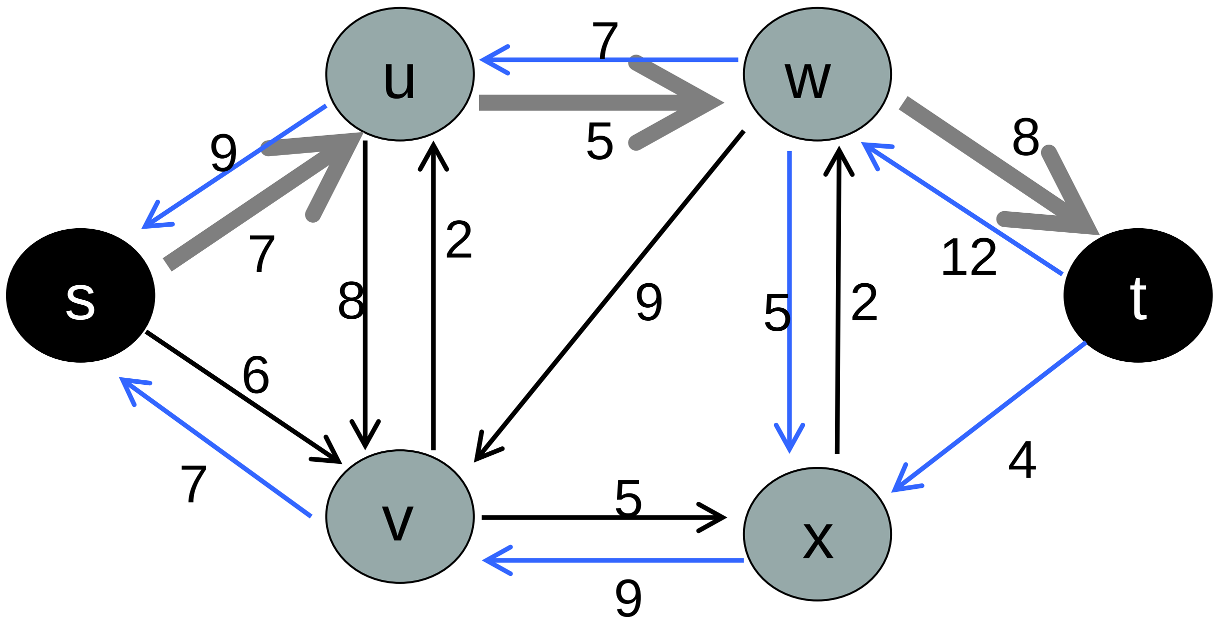
- For any a, b in V, $c_f(a,b) = 0$ if neither (a,b) nor (b,a) is an edge (as c and f are both 0 for such pairs), so we do not look at them

- Residual Graph (edges with 0 residual capacity are never shown)



- Note that residual graph may have edges where G did not (shown in color blue)
- It also may NOT have edges where G has one, ex. (x,t)
  - The residual capacity of the edge is 0
  - Such edges are called saturated

- Augmenting Path – path from s to t



- One path shown in bold grey, <s,u,w,t> with residual capacity = min(7, 5, 8) = 5
  - We can increase ("augment") the flow on each edge of the path by 5 to get a new feasible flow with higher value

Ford-Fulkerson Algorithm

1. Start with a feasible flow f (usually f=0 for all (u,v))
2. Create the residual graph G_f
3. Find an augmenting path p in G_f
4. Augment the flow in G
5. Repeat 2-4 until there is no augmenting path

```
FORD-FULKERSON-METHOD(G, s, t)
initialize flow f to 0
while there exists an augmenting path p
      do augment flow f along p
return f
```

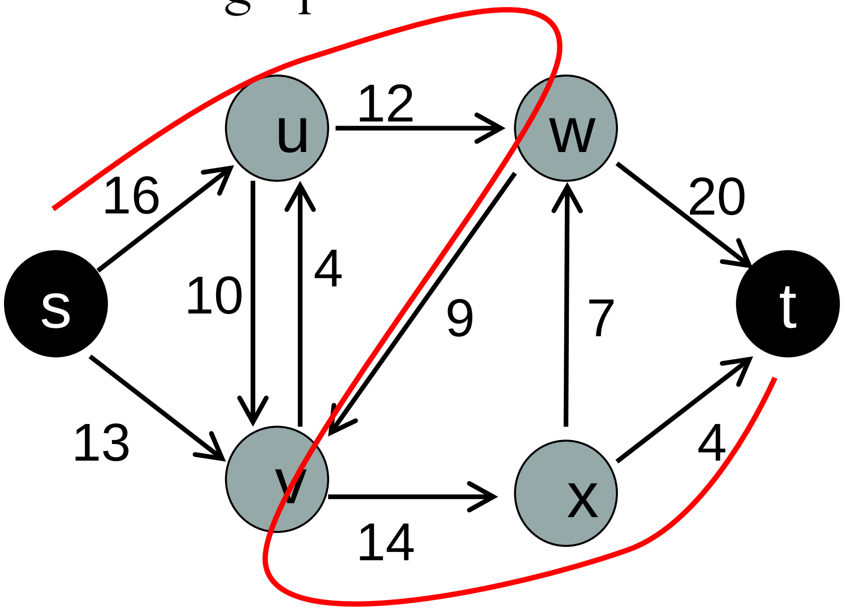
- Augmenting the flow along path p with residual capacity c

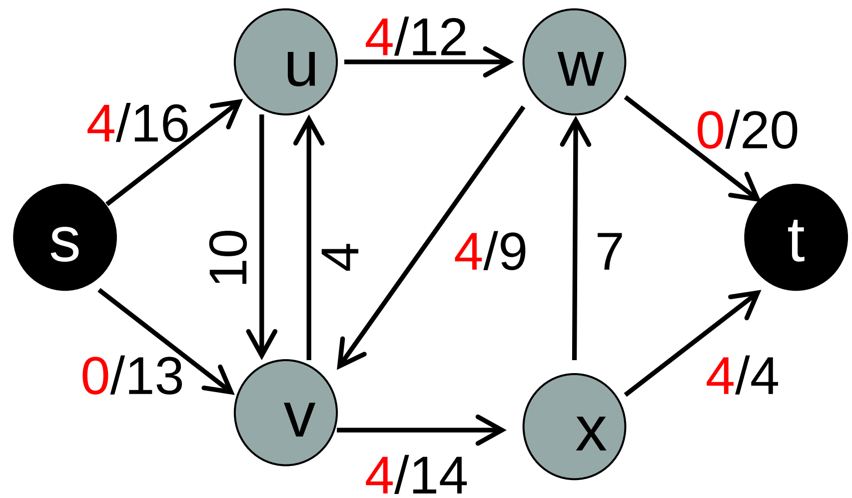
```
FORD-FULKERSON(G, s, t)
for each edge (u, v) ∈ E[G]
     do f[u, v] ← 0
        f[v, u] ← 0
while there exists a path p from s to t in the residual network G_f
     do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
        for each edge (u, v) in p
             do f[u, v] ← f[u, v] + c_f(p)
                f[v, u] ← −f[u, v]
```

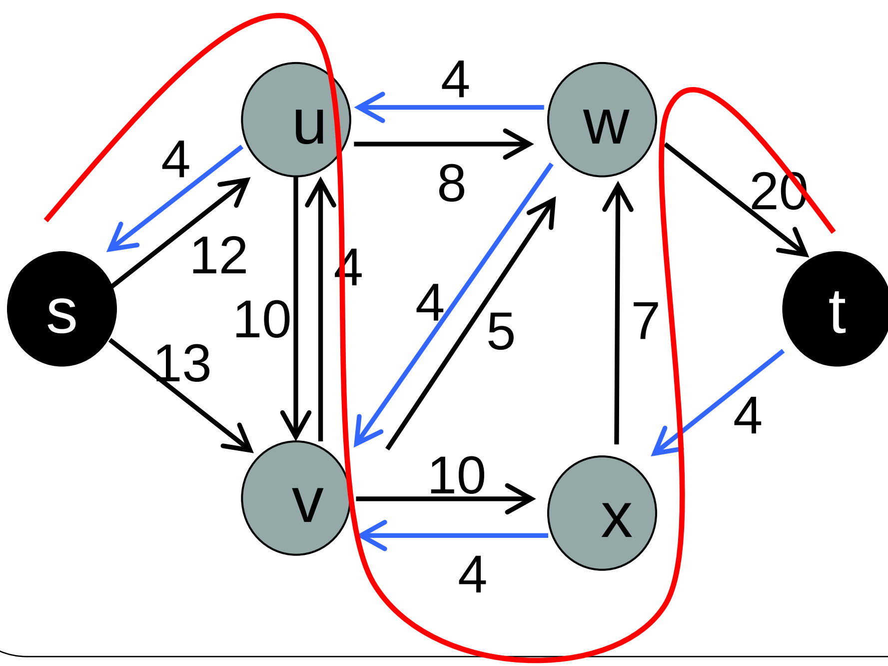
  - Note that either (u,v) or (v,u) must be an edge in G (or (u.v) cannot be in G_f)
  - If (u,v) is an edge, this increases f (u,v)
  - If (u,v) is not an edge, this actually decreases f(v,u)

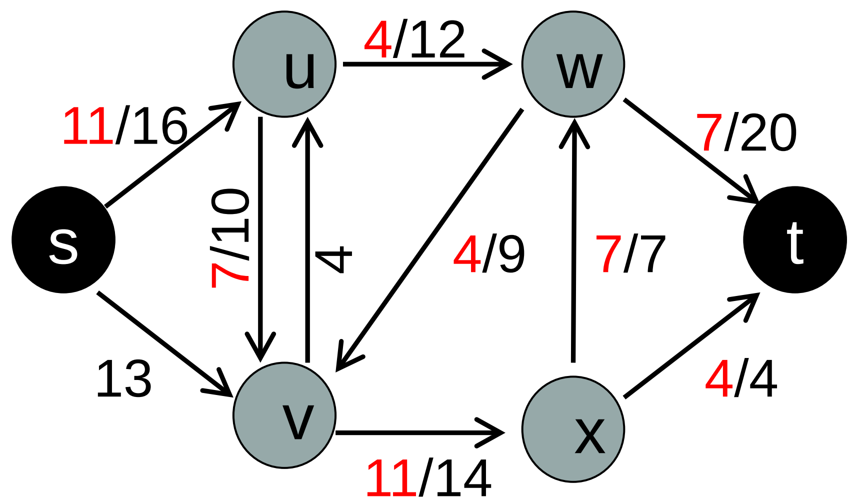
Residual graph

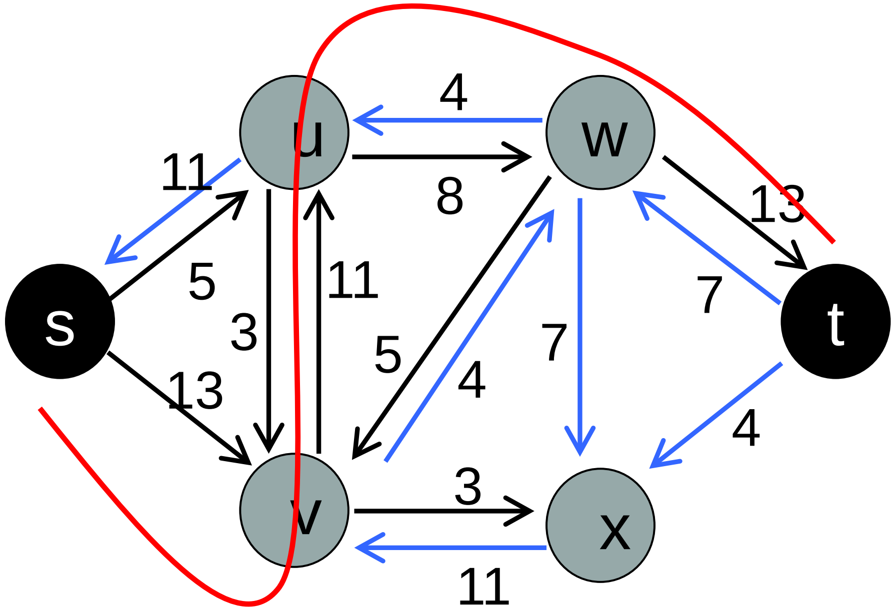
Flow Assignment

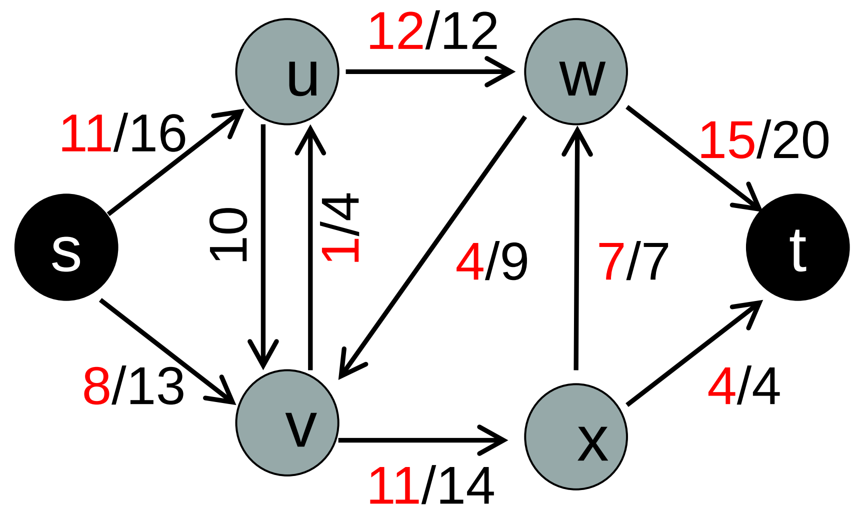


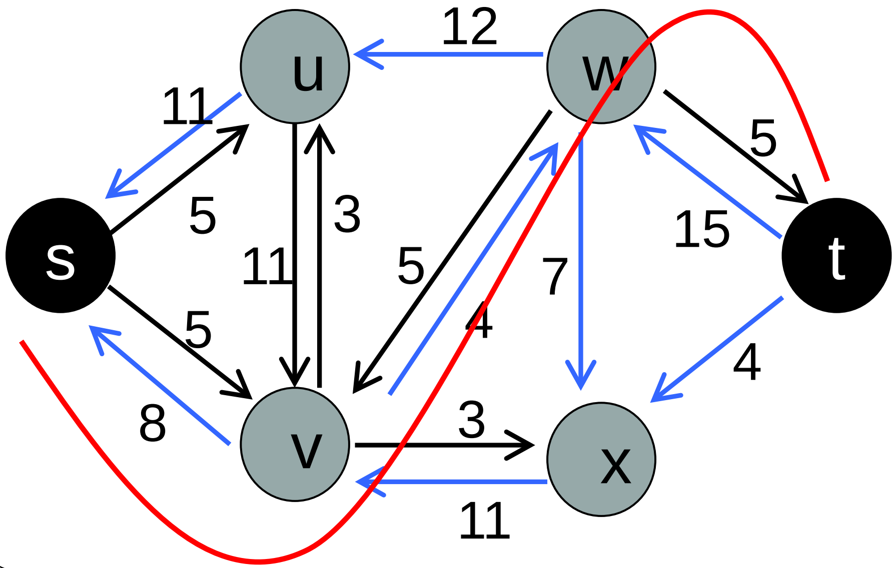


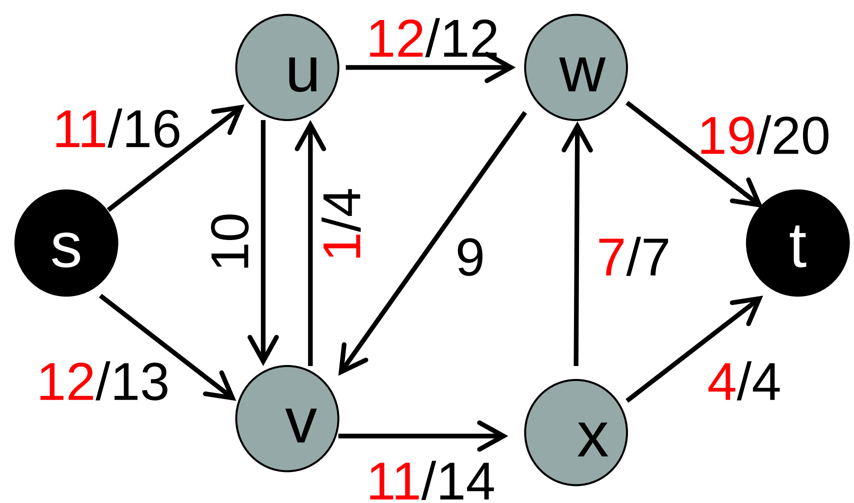


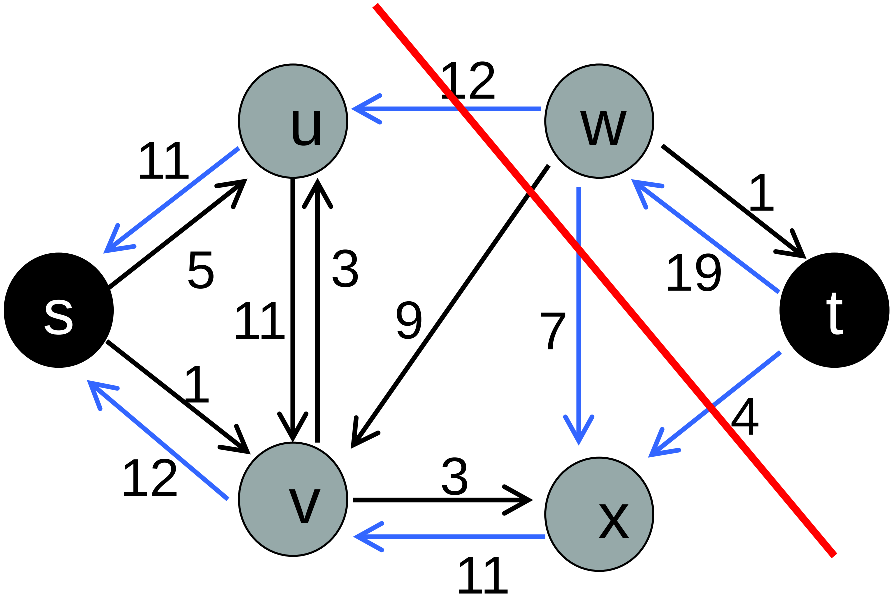



s
u
w
t
v
x
11
5
13
3
11
4
8
13
5
4
7
7
4
3
11


s
u
w
t
v
x
12/12
11/16
15/20
10
1/4
4/9
7/7
8/13
11/14
4/4


s
u
w
t
v
x
12
11
5
5
8
11
3
5
4
7
5
15
4
3
11


s
u
w
t
v
x
12/12
11/16
19/20
10
1/4
9
7/7
12/13
11/14
4/4

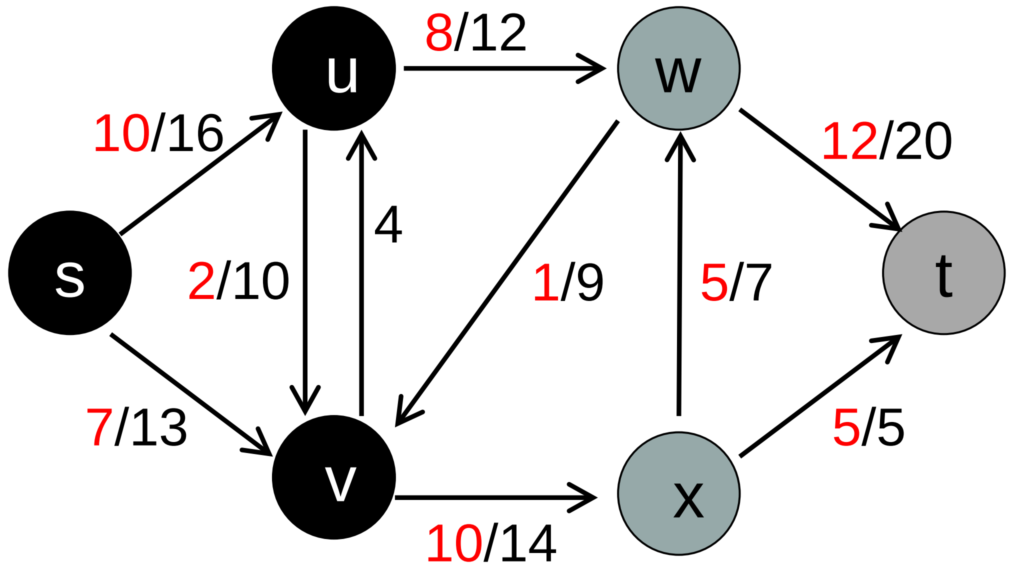


No augmenting path in the residual graph, so stop

Maximum Flow $|f| = 23$

Proof of Correctness

- We first need some definitions
  - A cut (S, T) of a flow network G = (V, E) is a partition of V into S and T = V – S, such that $s \in S$ and $t \in T$
  - If f is a flow then the net flow across the cut (S, T) , f(S, T), is the sum of the flows (f) of all <u>pairs</u> (u,v) with u in S and v in T
  - The capacity of the cut (S, T), c(S, T), is the sum of the capacities of all <u>edges</u> (u,v) with u in S and V in T
    - Of course, $f(S, T) \leq c(S, T)$
  - A minimum cut of a network is a cut whose capacity is minimum over all possible cuts of the network



- Consider the cut (S={s, u, v}, T={w, x, t})
- $f(S,T) = f(u,w) + f(v,w) + f(v,x)$

  $= 8 + (-1) + 10 = 17$
- $c(S,T) = c(u,w) + c(v,x) = 12 + 14 = 26$

Lemma 1: Let f be a flow in a network G with source s and sink t, and let (S, T) be a cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof:

$$
\begin{aligned}
f(S, T) &= f(S,V) - f(S,S) \\
&= f(S,V) \\
&= f(s, V) + f(S - s, V) \\
&= f(s,V) \\
&= |f|
\end{aligned}
$$

Lemma 1 implies that the net flow across *any* cut is the same (= value of flow).

Corollary 2: The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G, and hence by the capacity of the minimum cut.

Theorem 3 (Max-flow min-cut theorem): If f is a flow in a flow network G = (V, E) with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths
3. $|f|$ = capacity of the minimum cut

Proof:

1 implies 2 is obvious, as otherwise $|f|$ can be increased by increasing the flow along the augmenting path

2 implies 3:

Suppose that G_f has no augmenting paths. Let

$S = \{v \in V$: there exists a path from s to v in $G_f\}$ and

$T = V - S$.

Then (S,T) is a cut as s is in S and t is not in S as there is no path from s to t in G_f.

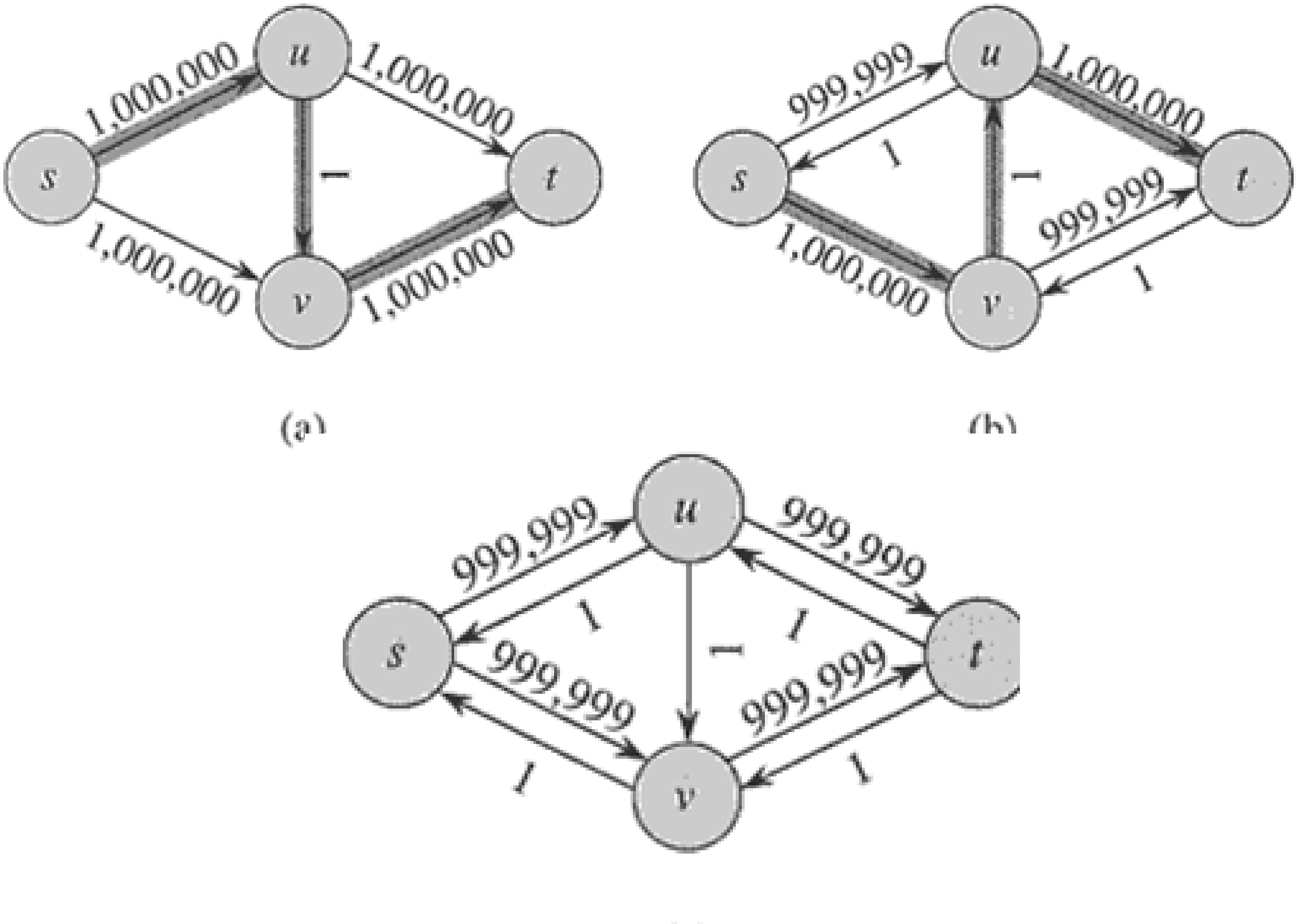
For any $u \in S$ and $v \in T$, we have $f(u,v) = c(u,v)$ as otherwise (u,v) is in G_f, which would mean v is in S, which is a contradiction. Therefore, by Lemma 1, $|f| = f(S,T) = c(S,T)$

3 implies 1: By corollary 2, $|f| \le c(S,T)$ for all cuts (S,T). Then, $|f| = c(S,T)$ implies $|f|$ is a maximum flow.

Time Complexity

- Original Ford-Fulkerson algorithm does not specify how to find an augmenting path
  - Can find in any order
- Assume all capacities are integer
- Let $f^* =$ maximum flow
- Lines 1-3 (Initialization) takes $O(|E|)$ time
- No. of times the while loop (no. of times an augmenting path is found) is executed is bounded above by $|f^*|$
  - As $|f|$ increases by at least 1 in each augmentation
- Each iteration of the while loop takes $O(|E|)$ time
- So worst case time complexity $O(|E||f^*|)$
  - This is not polynomial, it is pseudo-polynomial

- This bound is tight



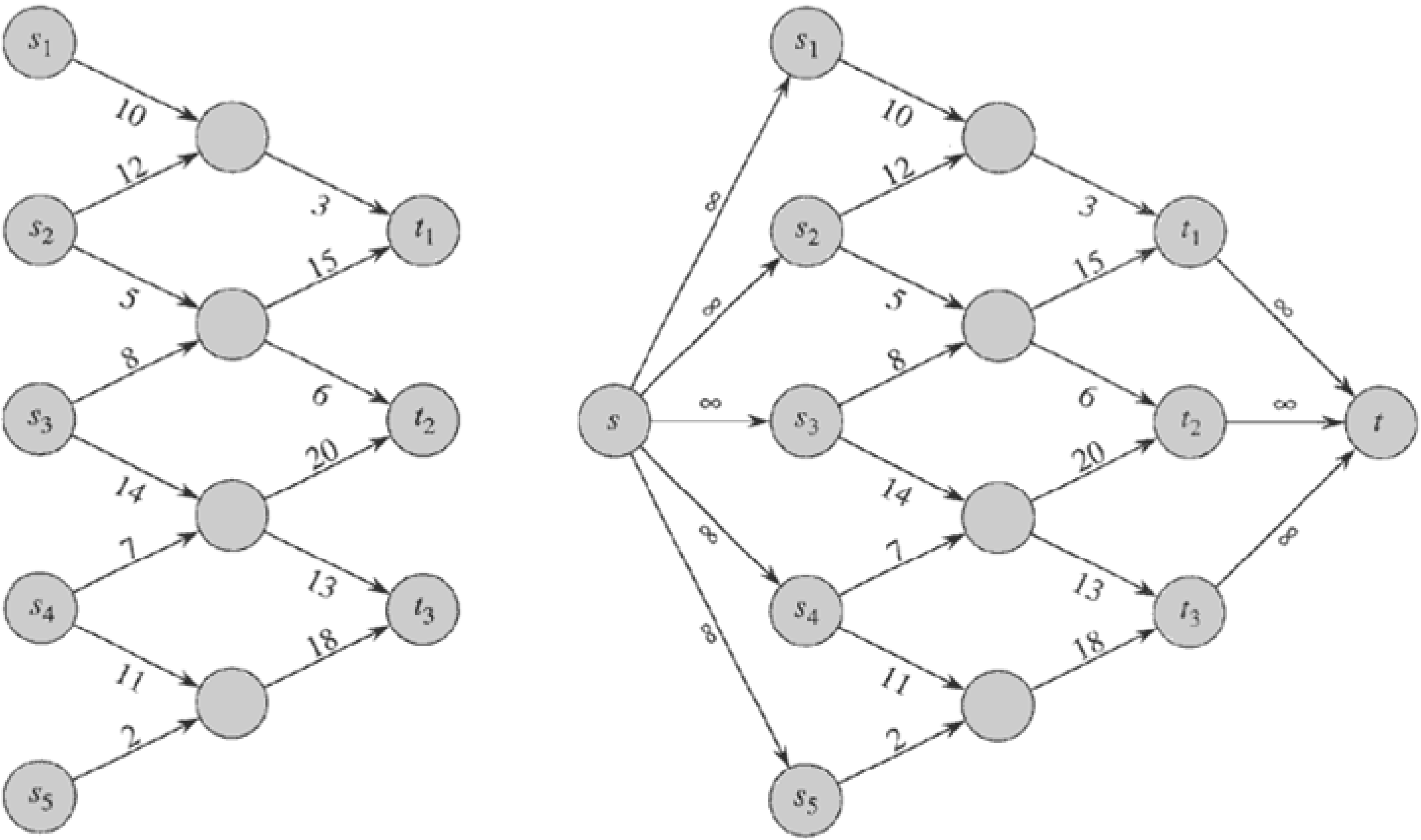
(c)

Edmonds-Karp Algorithm

- Proposed in 1972
- Almost same as Ford-Fulkerson
- Main difference: Uses BFS to find augmenting paths in residual graph instead of DFS
- You can prove that
  - If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation
  - The total number of flow augmentations performed by the Edmonds-Karp algorithm is $O(VE)$
- This gives time complexity of Edmonds-Karp as $O(VE^2)$, as BFS can be done in $O(E)$

What if there are multiple sources and sink?

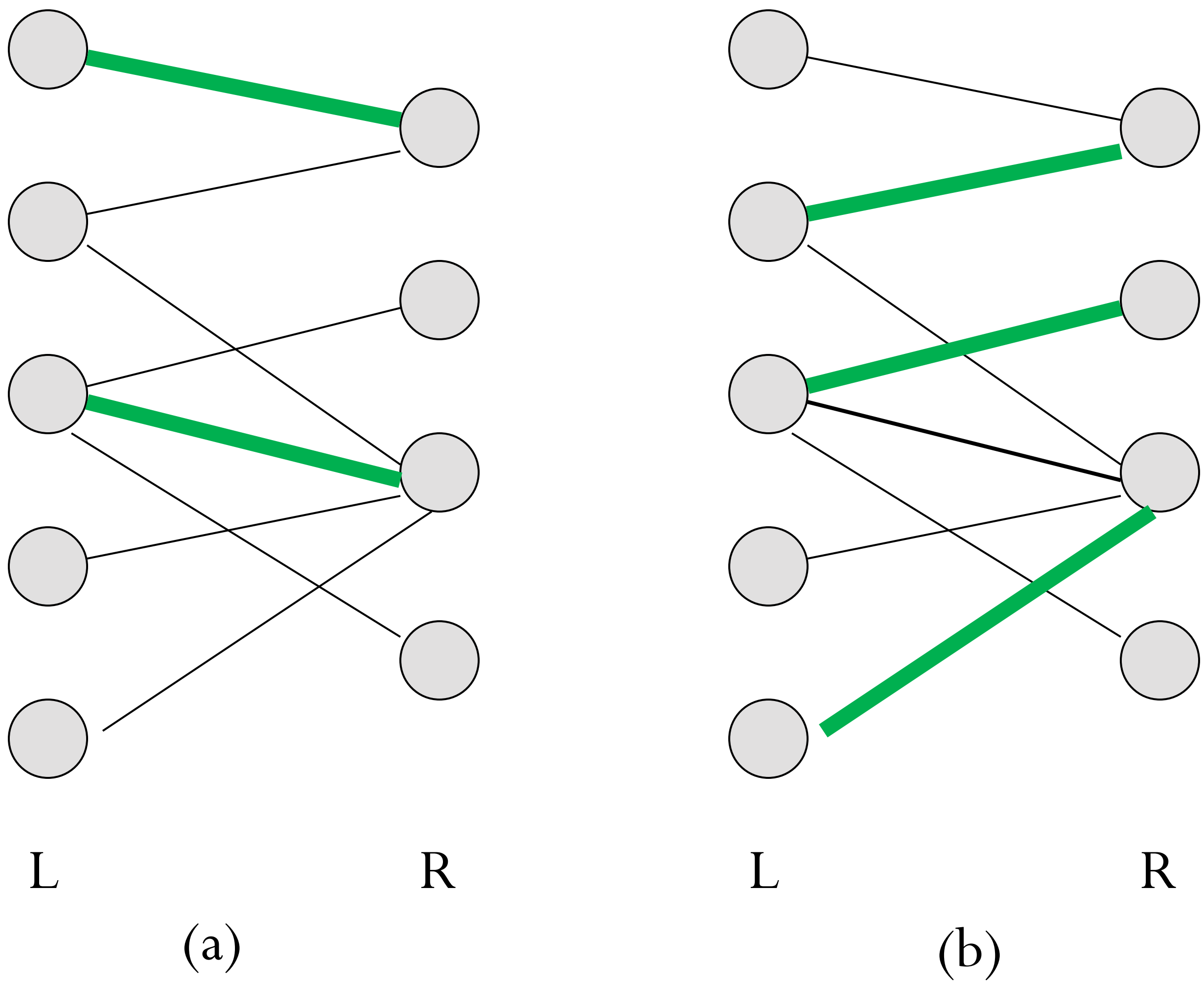
- Suppose there are multiple sources $s_1, s_2, s_3,..s_p$ and multiple sinks $t_1, t_2, t_3, ...t_q$
- How do we maximize the sum of the flows from all the sources to all the sinks?
- Can easily use the standard maximum flow problem
  - Add a "supersource" s with edge (s, s_j) from s to all sources s_j with capacity ∞
  - Add a "supersink" t with edge (t_j, t) from all sinks t_j to t with capacity ∞
  - Solve the maximum flow problem with s as source and t as sink



(a) (b)

Application: Maximum Cardinality Bipartite Matching

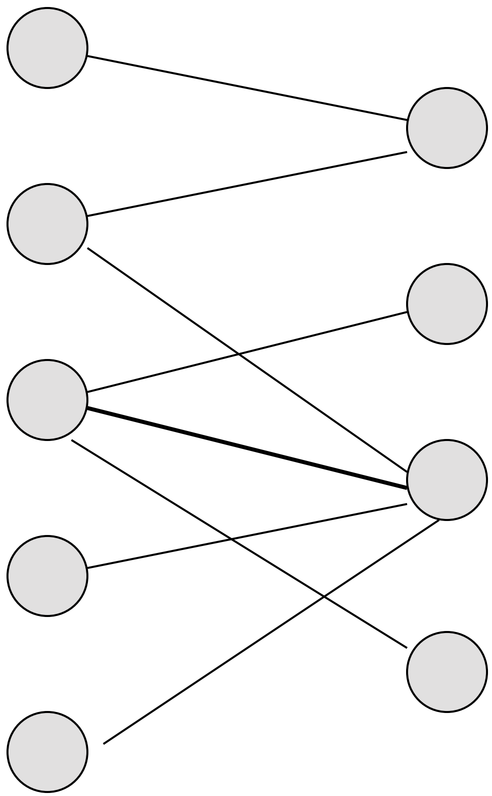
- Bipartite Graph: an undirected graph $G = (V, E)$ such that the vertex set can be partitioned $V = L \cup R$ where L and R are disjoint and there is no edge between two vertices in L or two vertices in R
- A matching in an undirected graph $G = (V, E)$ is a subset of edges $M \subseteq E$, such that for all vertices $v \in V$, at most one edge of M is incident on v.
- A maximum cardinality matching is a matching with maximum number of edges among all possible matchings
  - Also simply called maximum matching for unweighted graphs



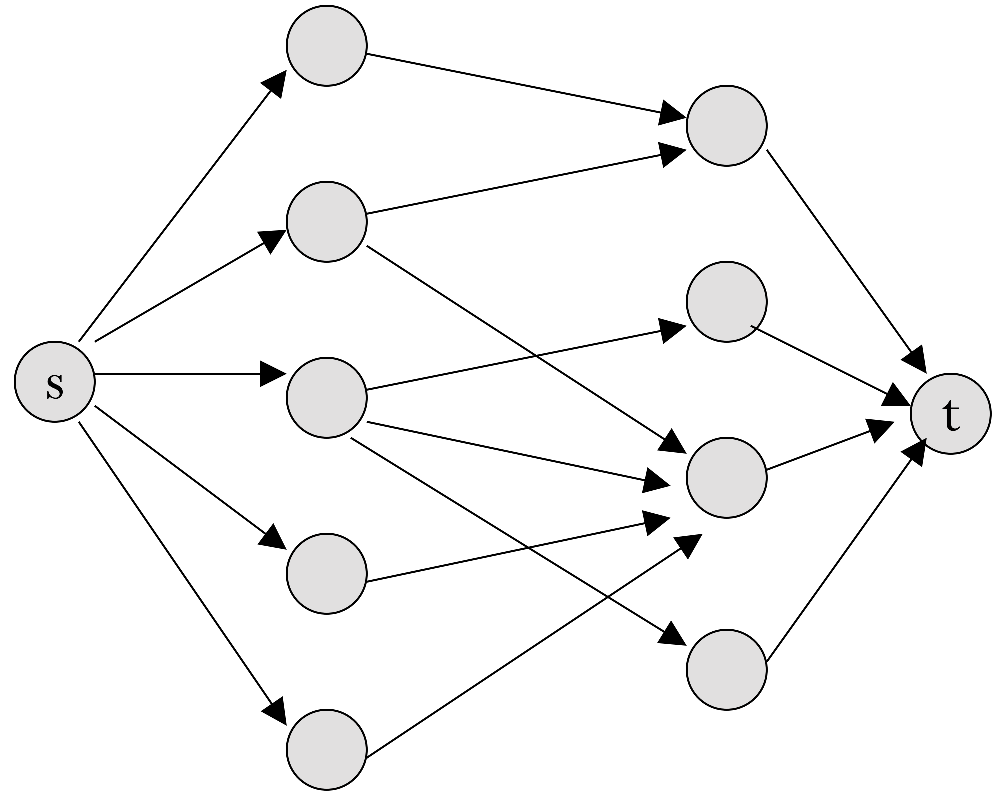
(a)A matching with cardinality 2

(b) A maximum matching with cardinality 3

- Given the undirected bipartite graph G = (V, E) with partitions L and R, create a flow network G' = (V', E') as follows
  - Add two new vertices s, t. So V' = V ∪ {s, t}
  - For each node u in L, add a directed edge (s,u) with capacity 1 to E'
  - For each node v in R, add a directed edge (v,t) with capacity 1 to E'
  - For each edge (u,v) in E with u in L and v in R, add a directed edge (u, v) with capacity 1 to E'



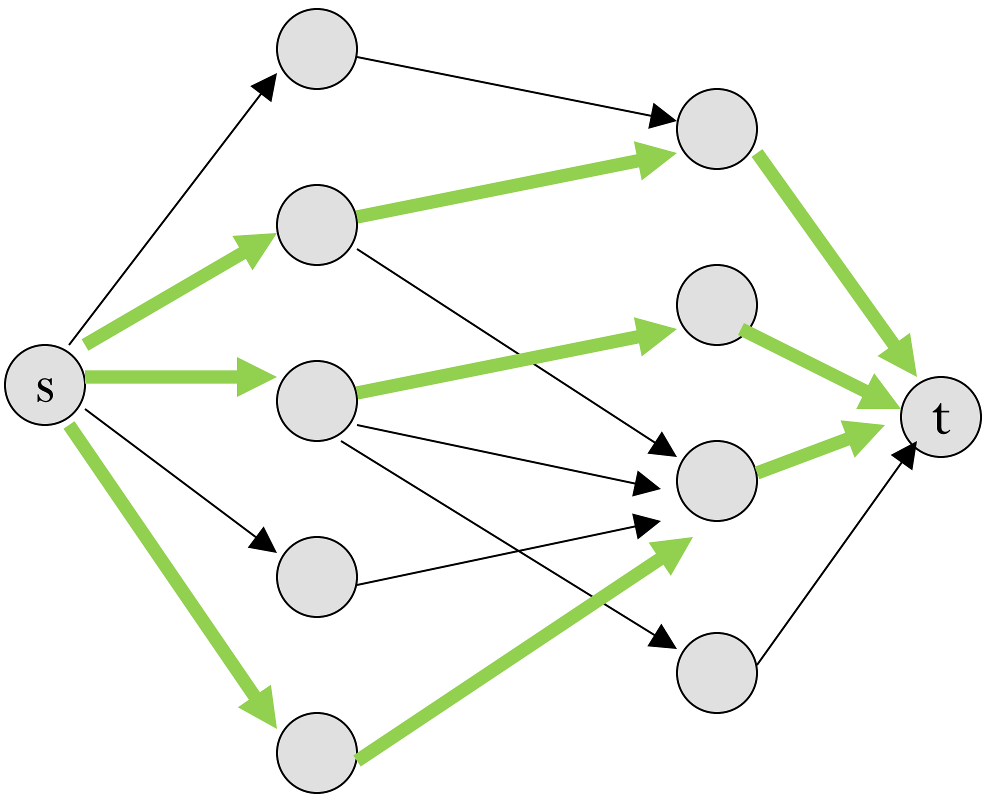
L R



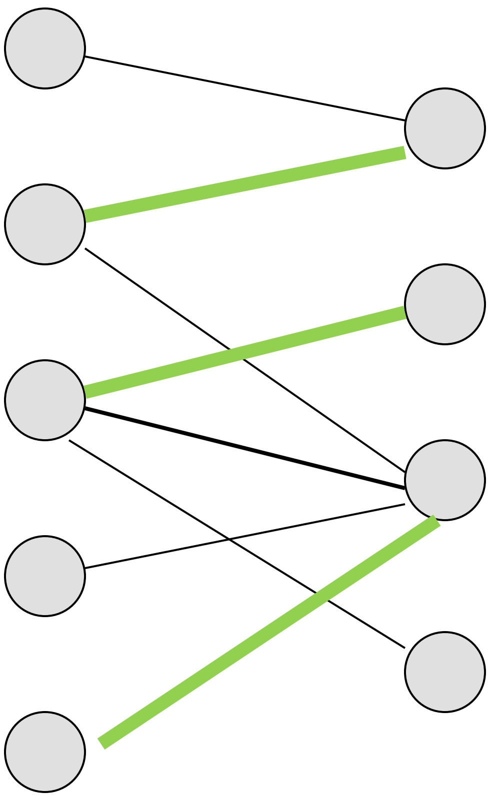
L R

All capacities are 1

- Now solve the maximum flow problem from s to t in G’
- The edges of G with corresponding edges in G’ with flow = 1 correspond to the maximum matching



Maximum flow found



Corresponding Maximum Matching

Application: Edge Connectivity

- Given an undirected graph G = (V, E), edge connectivity of G is the minimum number of edges that have to be removed to disconnect the graph
  - A graph is called k-edge-connected if its edge connectivity is at least k
- Problem: Find the edge connectivity of a given undirected graph
- Important practical problem in various forms for different types of network design
  - Example: to avoid disruption in a computer network, need to ensure that a small number of link failures cannot disconnect the network

- We will use the maximum flow problem
- We know that the maximum flow is equal to the capacity of the minimum (S,T) cut
- So if we set all capacities to 1, the maximum flow value gives the minimum number of edges that goes across any cut (S,T), and so, the minimum number of edges that needs to be removed so that there is no path from s to t
- But the flow network is a directed graph, we need to solve it for an undirected graph
  - Easy. Maximum flow algorithms work on undirected graphs simply by converting it first to a directed graph, with each undirected edge replaced by two directed edges

- We also need to consider disconnection of any two vertices, not just two specified ones like s and t
  - So (u,v)-cuts for any two vertices u and v
  - Simple solution:
    - For each pair of vertices (u,v), set s=u, t=v and find the minimum cut size by solving the maximum flow problem
    - Take the minimum over all (u,v) pairs
    - Time complexity = no. of distinct pairs × max-flow time

      $= O(|V|^2) \times O(|V||E|^2)$ (using Edmonds-Karp)

      $= O(|V|^3|E|^2)$
    - Can do better, no need to consider all pairs

Input: Connected graph G = (V, E)

choose any vertex p in V

min_size = |E|

for all vertices q ≠ p do

find maxflow M in directed graph G' = (V, E')

where E' = { (u,v), (v,u) | (u,v) in E }

s = p, t = q, and all capacities = 1

min_size = min (min_size, M)

edge connectivity of G = min_size

Why is it sufficient to just find edge-connnectivity between a fixed *p* and all other vertices (and not between all pairs of vertices)?

Time Complexity = $(|V|^2 |E|^2)$ (using Edmonds-Karp)

Preflow-Push Method

- Also called Push-Relabel method as it is based on two basic operations, push and relabel
- Main difference from Ford-Fulkerson based algorithms
  - Do not need to maintain the flow-conservation property throughout the execution
    - Total inflow at a vertex can be greater than total outflow from it in intermediate steps
    - But in the final solution, they must be the same as before

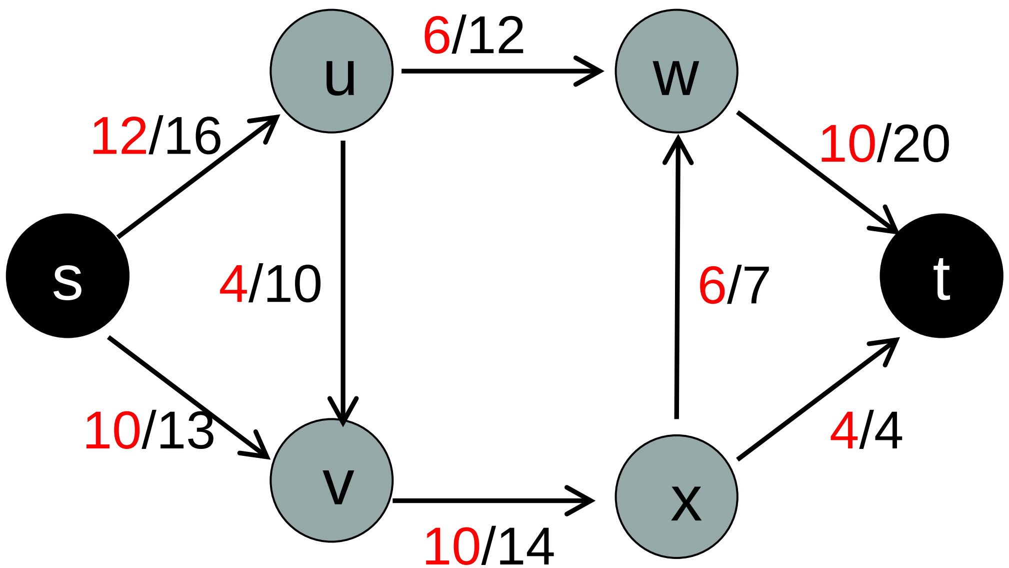
- Constraints satisfied by $f : V \times V \rightarrow R$ in intermediate steps of preflow-push:
  - Capacity constraint : For all $u, v \in V$, $f(u,v) \leq c(u,v)$

    (same as before)
  - Skew symmetry : For all $u, v \in V$, $f(u,v) = -f(v,u)$

    (same as before)
  - Flow constraint: For all $v \in V - \{s\}$, $\sum_{u \in V} f(u,v) \geq 0$

    (Relaxed, allows net flow into v to be greater than 0)
- Excess flow into v, $e(v)$ = net flow into v = $\sum_{v \in V} f(u,v)$
- A vertex is called active or overflowing if $e(v) > 0$
- f is called a preflow

An Example Preflow

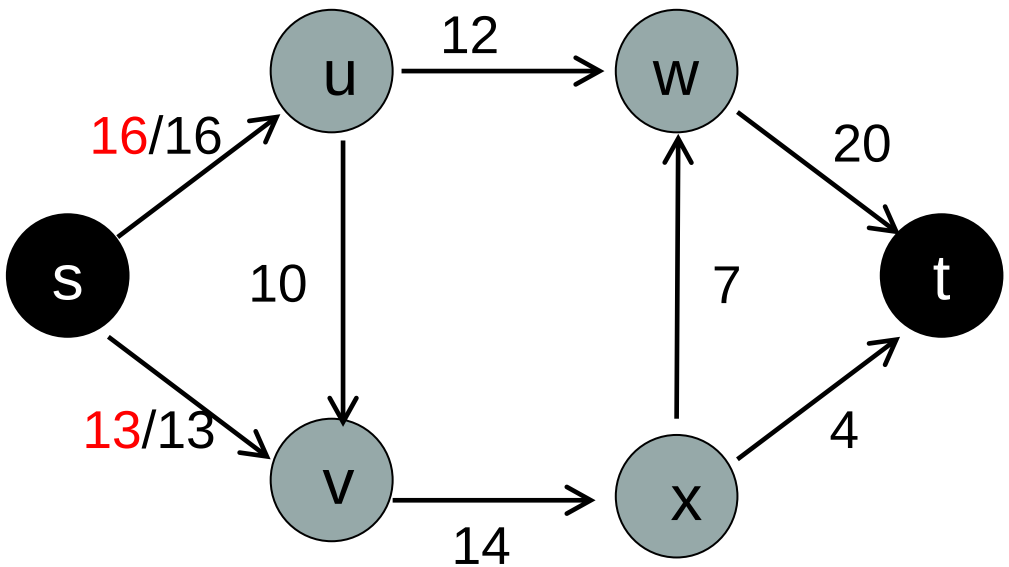


- e(u) = 2 (active)
- e(v) = 4 (active)
- e(w) = 2 (active)
- e(x) = 0

Basic Idea

- Think of the vertices at different heights
  - Initially s is at height |V| and all others at height 0
- Think that each vertex has an arbitrarily large temporary storage
- Flow is pushed only downhill, from a vertex with higher height to a vertex with lower height
- Start the algorithm by pushing as much flow as possible from s to all its outgoing edges (i.e., push up to capacity of each edge from s)
  - Initial preflow
- The flow pushed first gets stored in the storage of the vertices at the other end

Initial Preflow



- e(u) = 16 (active)
- e(v) = 13 (active)
- e(w) = 0
- e(x) = 0

- Any other vertex u pushes this flow along each edge whenever possible (if the vertex v at the other end of the edge is at a lower height, i.e, is downhill, and the edge (u,v) is not saturated)
  - PUSH operation
- What if no such vertex v is found?
  - All vertices at the other end of outgoing edges have height $\geq$ this node's height
  - In this case, increase vertex u's height by 1 + minimum height of any vertex at other end of an unsaturated edge
  - RELABEL operation

- Continue until flow cannot be pushed forward anymore
  - All edges across the minimum cut get saturated
- But now you may have vertices with excess flow left in them
- Push this flow back towards s
  - RELABEL to heights greater than |V|
  - Eventually all excess flows go out through s (whose height always stays at |V|)
- The final flow satisfies the flow conservation constraint at each vertex
- So two types of operation, PUSH and RELABEL
  - This is why preflow-push method is also called the push-relabel method

The Height Function

- The same notion of residual capacity c_f and residual graph G_f as before is also used here
- Given a preflow f, a function $h: V \rightarrow N$ is a height function if it satisfies the following properties:
  - $h(s) = |V|$
  - $h(t) = 0$
  - $h(u) \leq h(v) + 1$ for any residual edge $(u,v) \in E_f$

- It is usually called the distance function, as it gives a lower bound on the distance from u to t in G_f
  - The text uses the term height to relate to downhill-uphill analogy, so let us use it also
- Note that the definition implies that given any preflow f, for any two vertices u, v, if $h(u) > h(v) + 1$, then (u,v) is not an edge in the residual graph G_f

PUSH Operation

- PUSH(u,v)

  Precondition:

  $e(u) > 0$ (i.e., u is active)

  $c_f(u,v) > 0$

  $h(u) = h(v) + 1$

  Action:

  Let $d_f(u,v) = \min(e(u), c_f(u,v))$

  Push $d_f(u,v)$ amount of flow from u to v

- PUSH is saturating if $c_f(u,v) = 0$ after the PUSH, otherwise non-saturating

PUSH(u, v)

▷ **Applies when**: u is overflowing, $c_f(u, v) > 0$, and $h[u] = h[v] + 1$.
▷ **Action:** Push $d_f(u, v) = \min(e[u], c_f(u, v))$ units of flow from u to v.
$d_f(u, v) \leftarrow \min(e[u], c_f(u, v))$
$f[u, v] \leftarrow f[u, v] + d_f(u, v)$
$f[v, u] \leftarrow -f[u, v]$
$e[u] \leftarrow e[u] - d_f(u, v)$
$e[v] \leftarrow e[v] + d_f(u, v)$

RELABEL Operation

- RELABEL(u)

  Precondition:

  $e(u) > 0$ (i.e., u is active)

  $h(u) \leq h(v)$ for all edges $(u,v) \in E_f$

  Action:

  $h(u) = 1 + \min\{h(v) \mid (u,v) \in E_f\}$

- Note that h(u) never decreases for any vertex u

RELABEL(u)

▷ **Applies when:** u is overflowing and for all $v \in V$ such that $(u, v) \in E_f$,
we have $h[u] \leq h[v]$.

▷ **Action:** Increase the height of u.

$h[u] \leftarrow 1 + \min\{h[v] : (u, v) \in E_f\}$

An Important Property

For any active vertex u, either a PUSH or a RELABEL operation must be applicable

- Why?
  - If PUSH operation is not applicable, then for all residual edges $(u,v) \in E_f$, $h(u) < h(v) + 1$
    - Note that $h(u)$ cannot be $>$ than $h(v) + 1$ by defn. of h
  - So $h(u) \leq h(v)$
  - But then a RELABEL operation is applicable to u

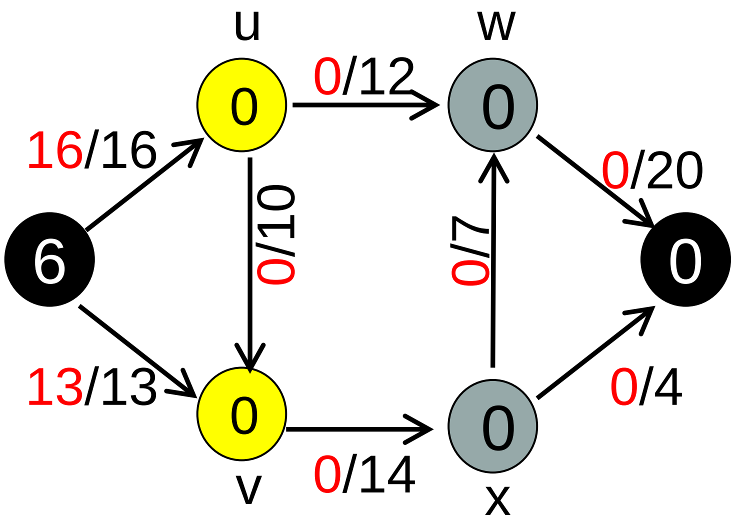
Generic Preflow-Push Algorithm

```
INITIALIZE-PREFLOW(G, s)
for each vertex u ∈ V[G]
      do h[u] ← 0
         e[u] ← 0
for each edge (u, v) ∈ E[G]
      do f[u, v] ← 0
         f[v, u] ← 0
h[s] ← |V[G]|
for each vertex u ∈ Adj[s]
      do f[s, u] ← c(s, u)
         f[u, s] ← −c(s, u)
         e[u] ← c(s, u)
         e[s] ← e[s] − c(s, u)
```

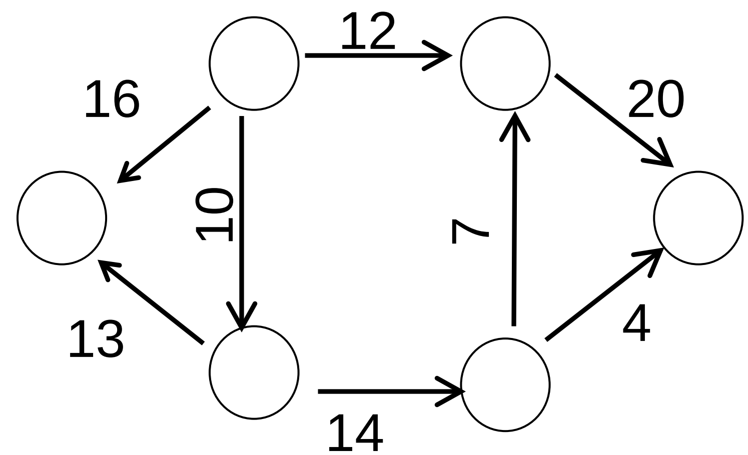
```
GENERIC-PUSH-RELABEL(G)
INITIALIZE-PREFLOW(G, s)
while there exists an applicable push or relabel operation
      do select an applicable push or relabel operation and perform it
```

Example

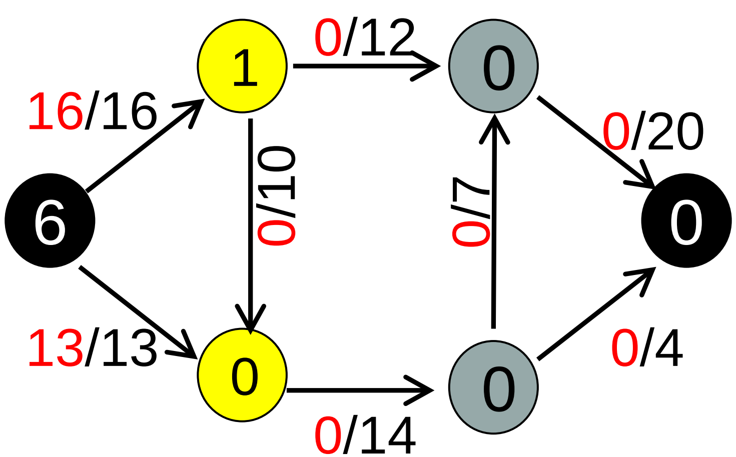
Initial Preflow

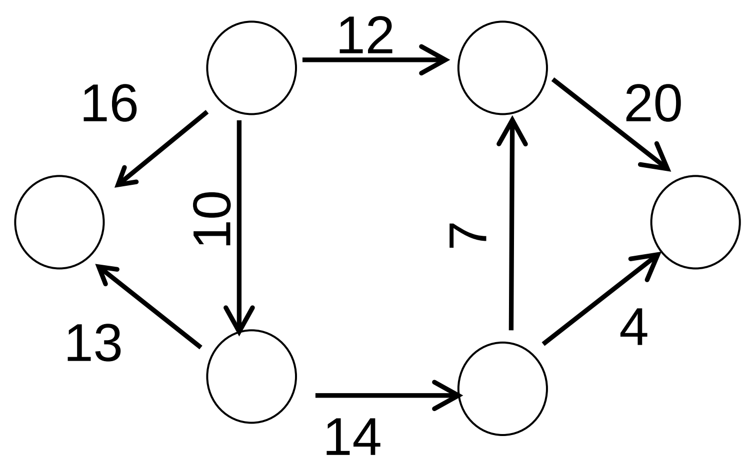


Residual Graph

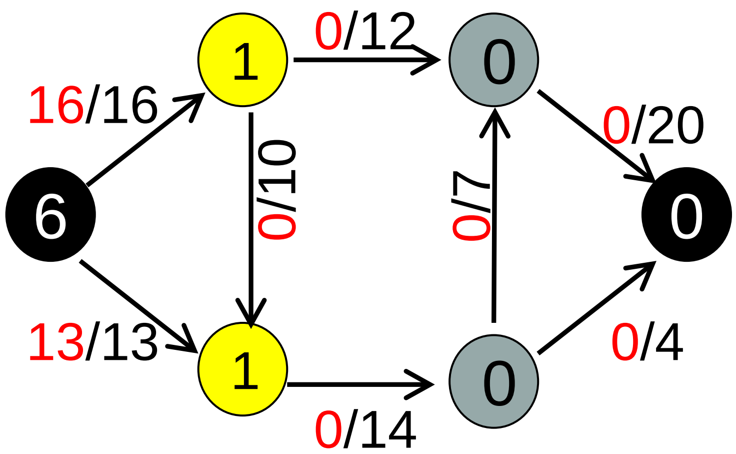


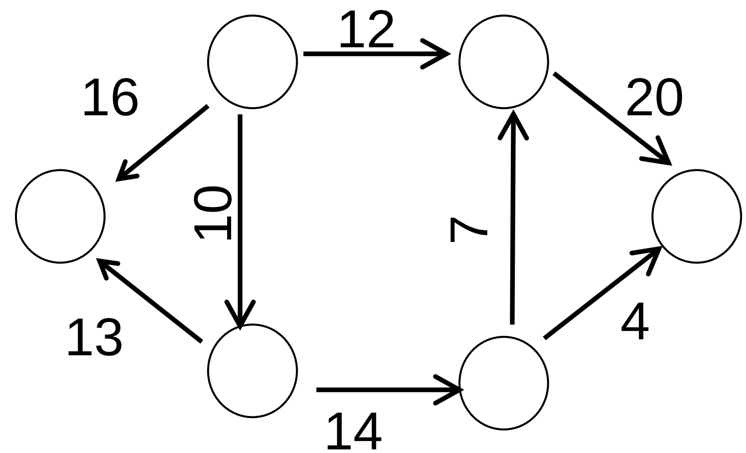
RELABEL(u)



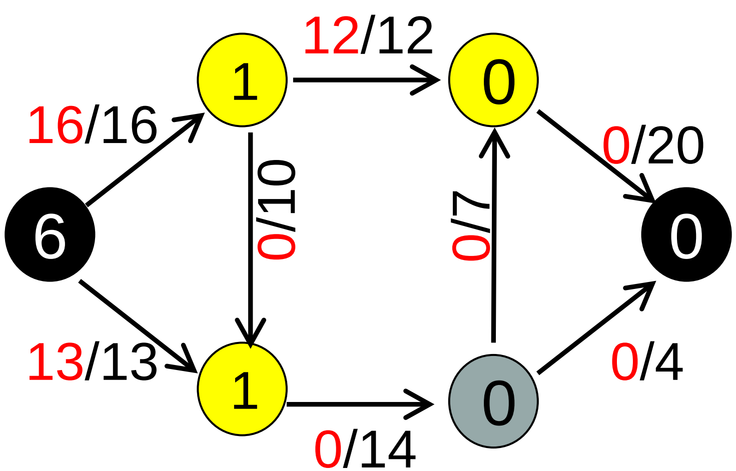


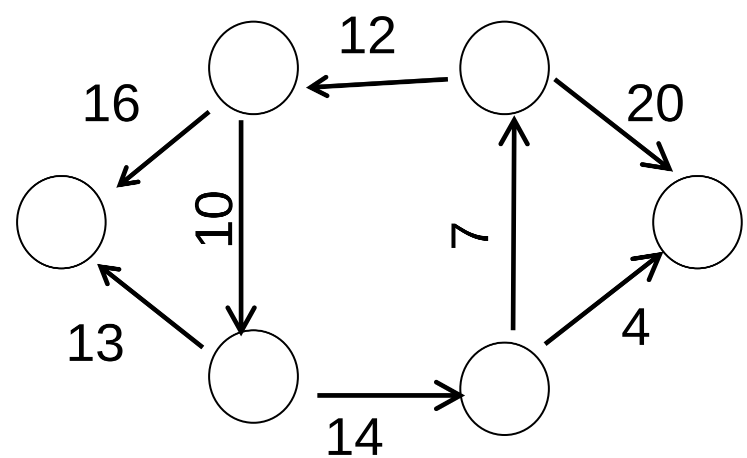
RELABEL(v)



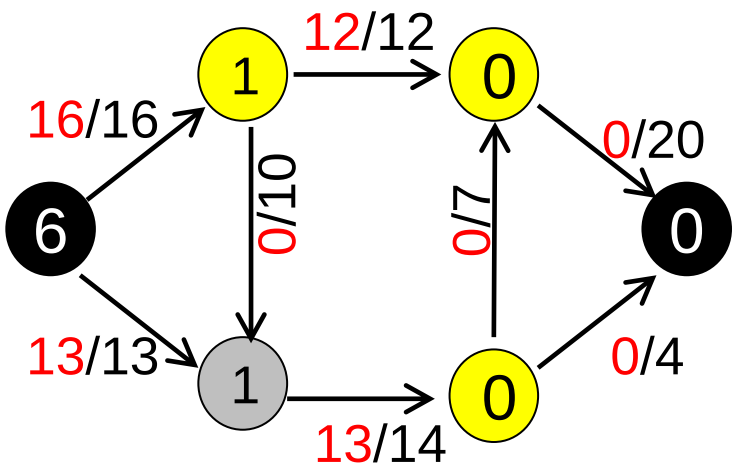


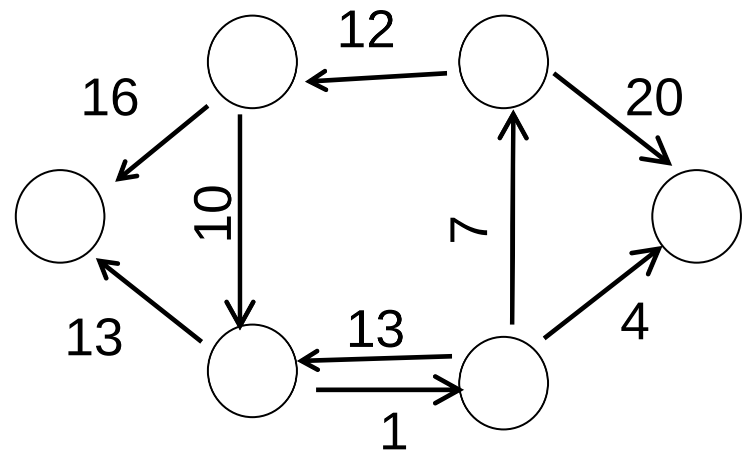
PUSH(u,w)



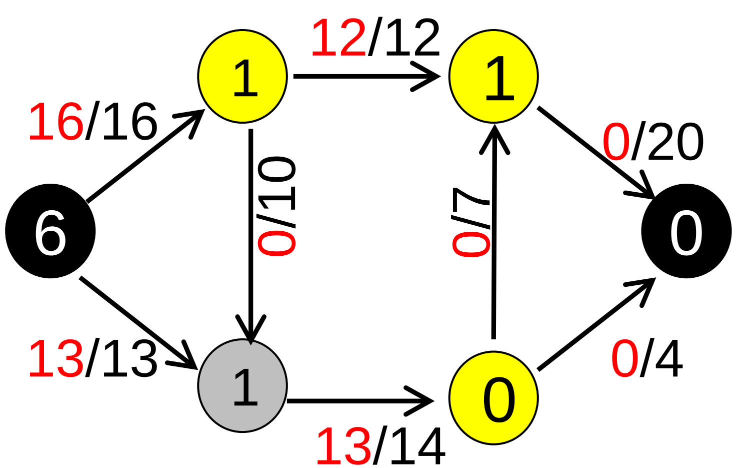


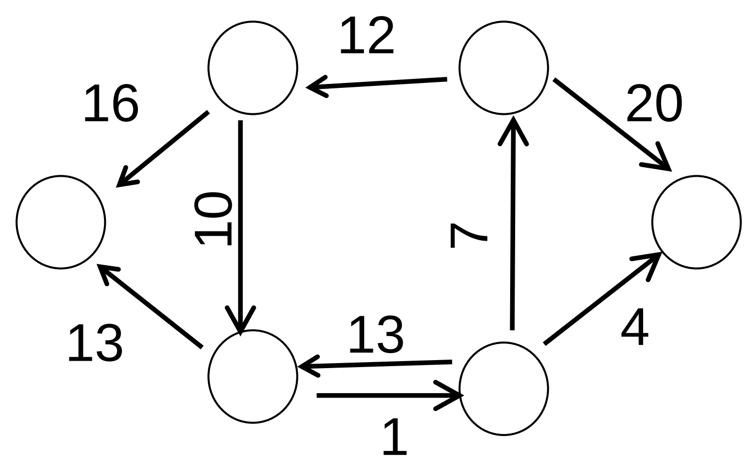
PUSH(v,x)



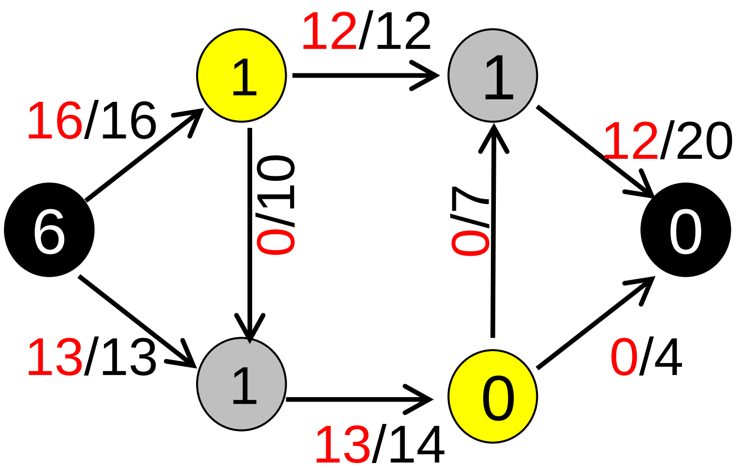


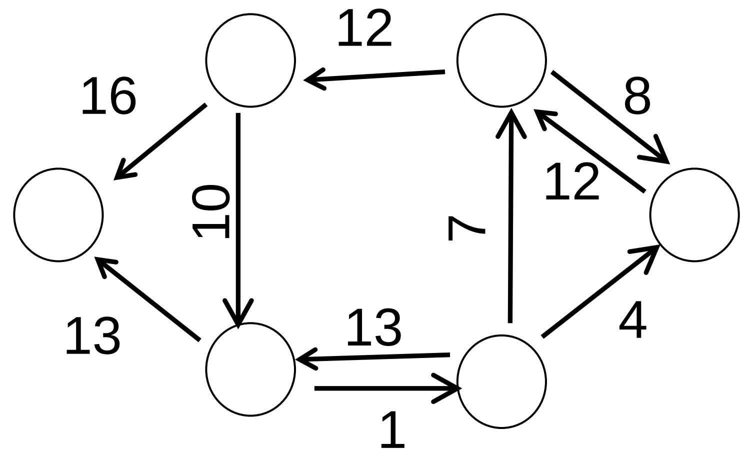
RELABEL(w)



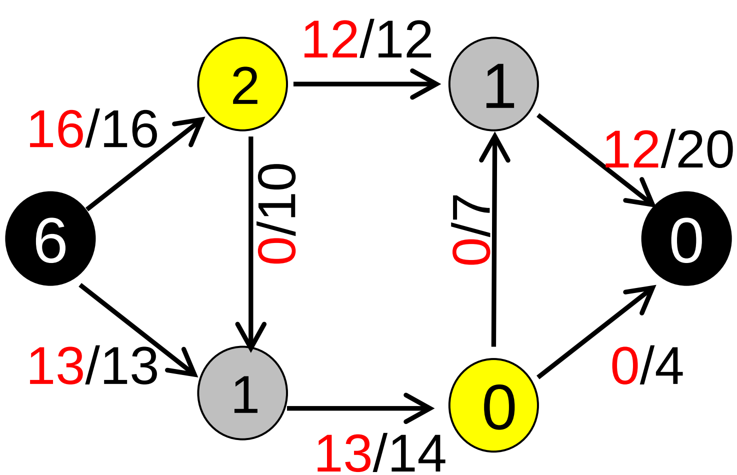


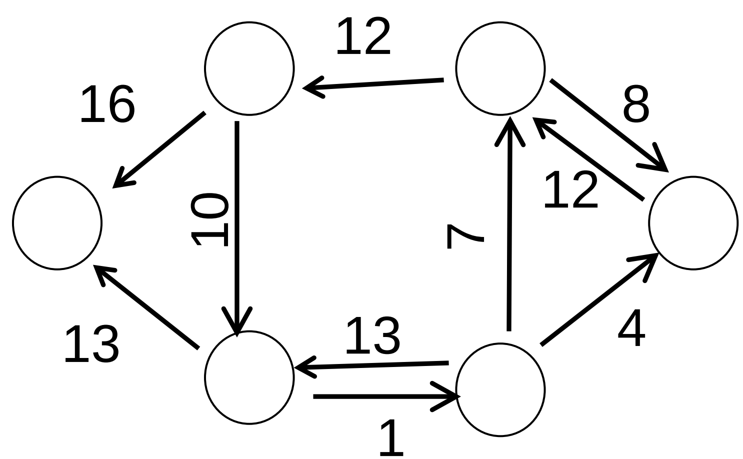
PUSH(w, t)



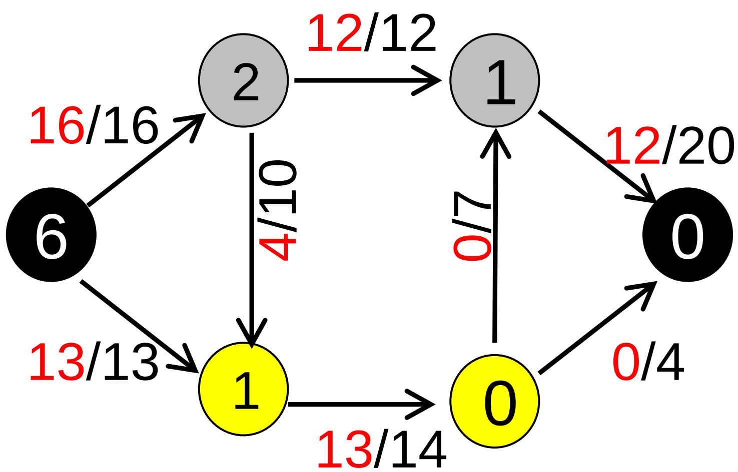


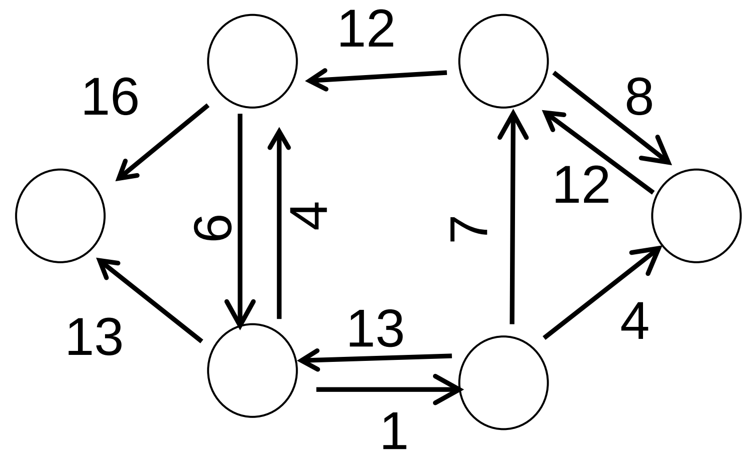
RELABEL(u)



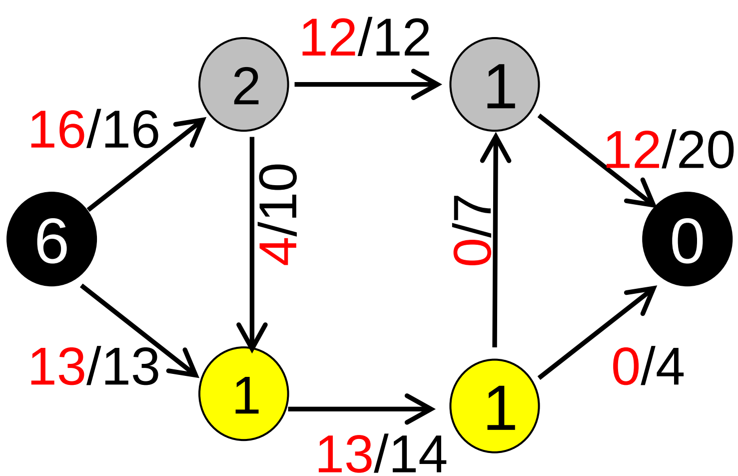


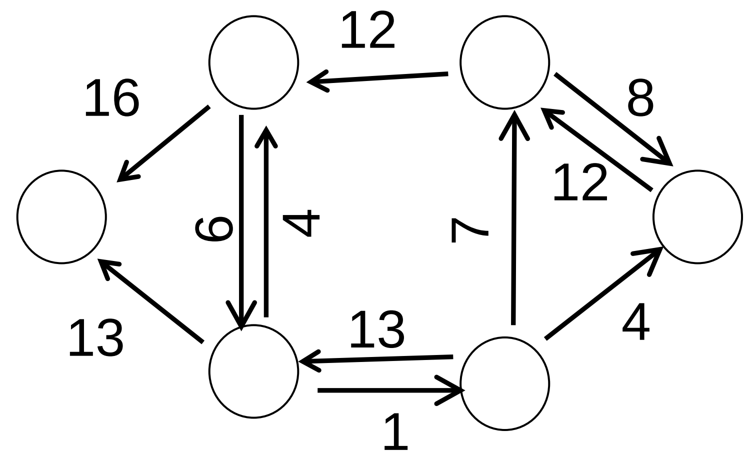
PUSH(u,v)



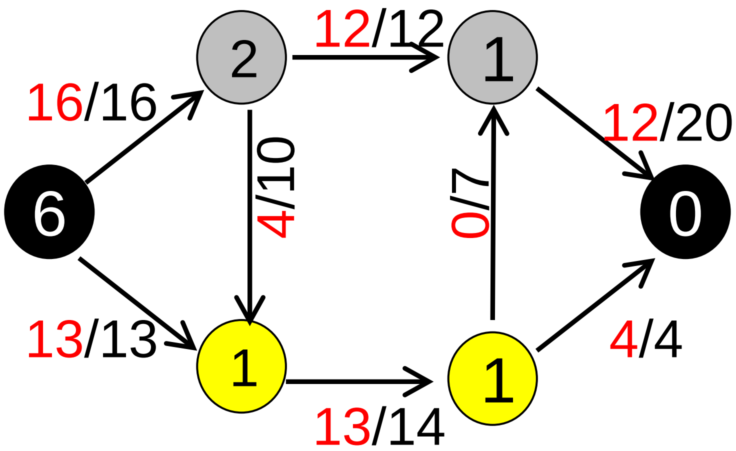


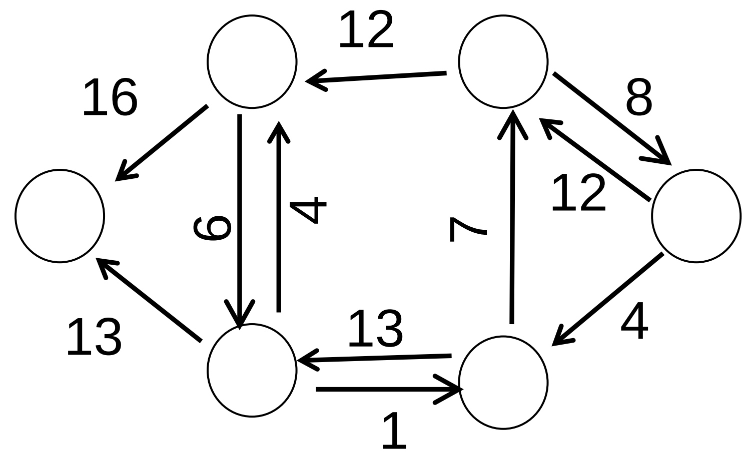
RELABEL(x)



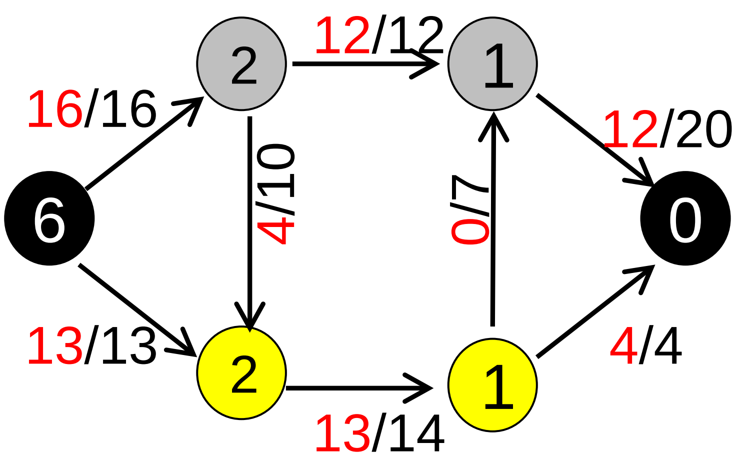


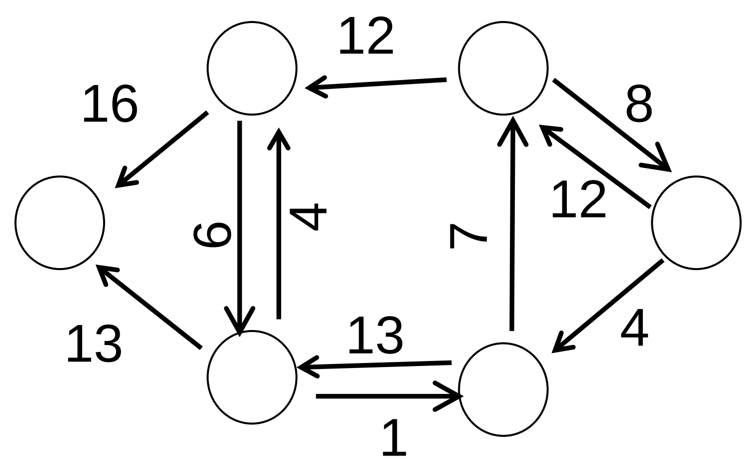
PUSH(x, t)



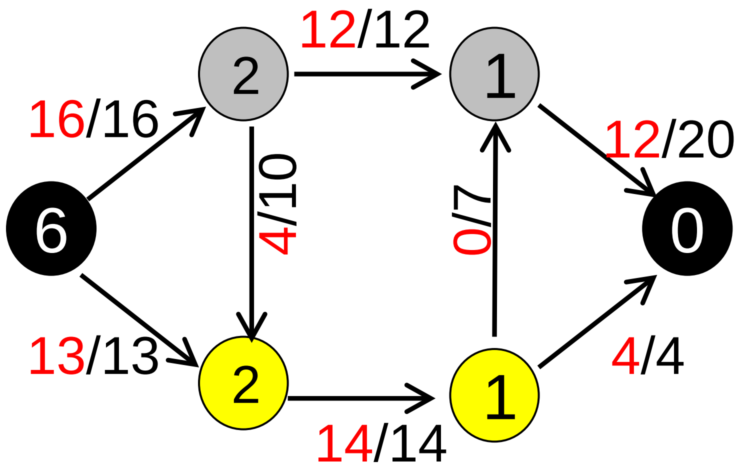


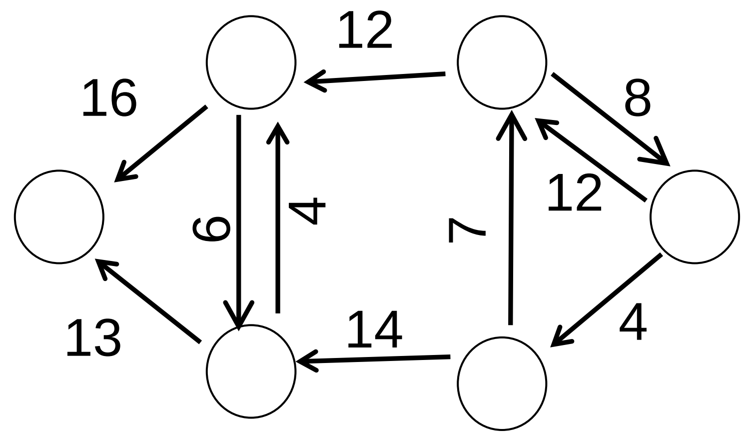
RELABEL(v)



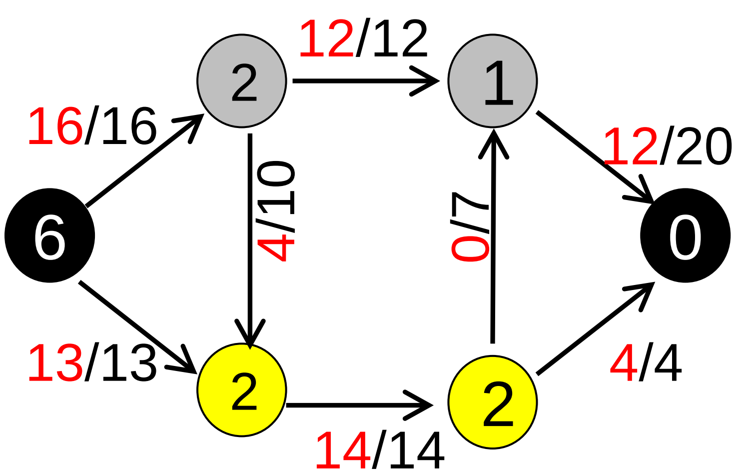


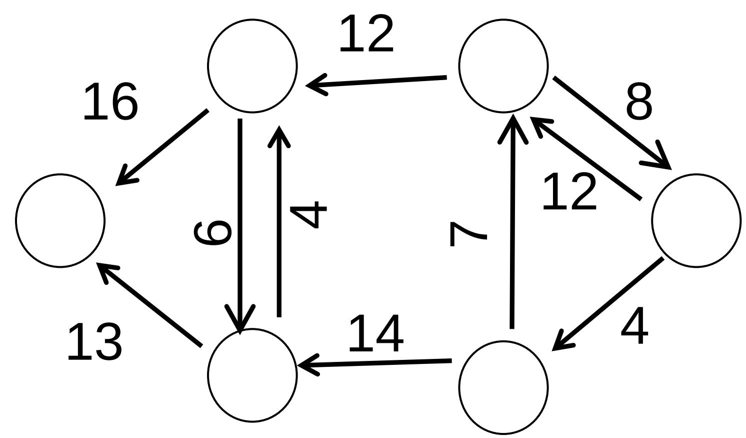
PUSH(v,x)



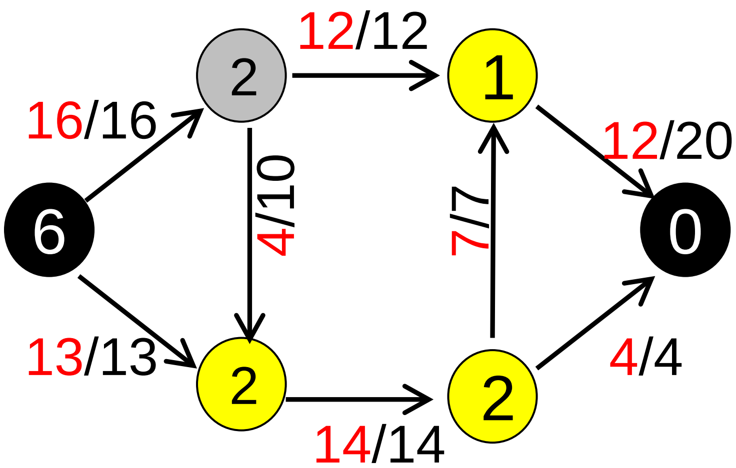


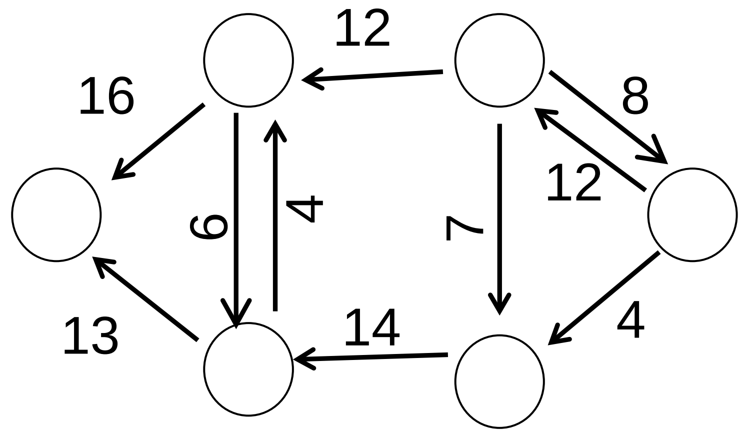
RELABEL(x)



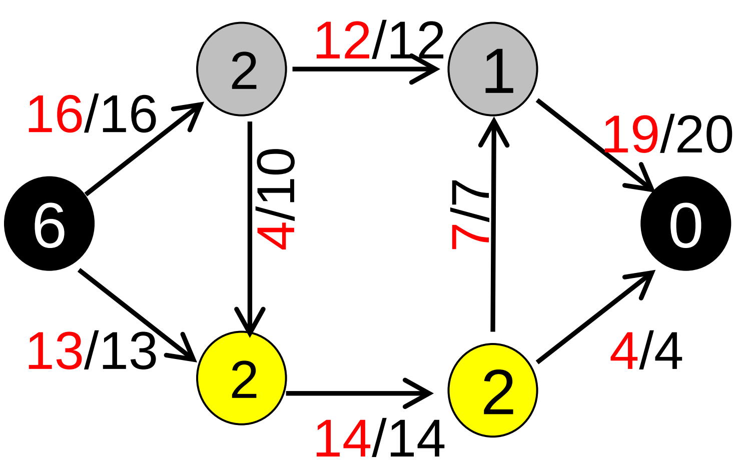


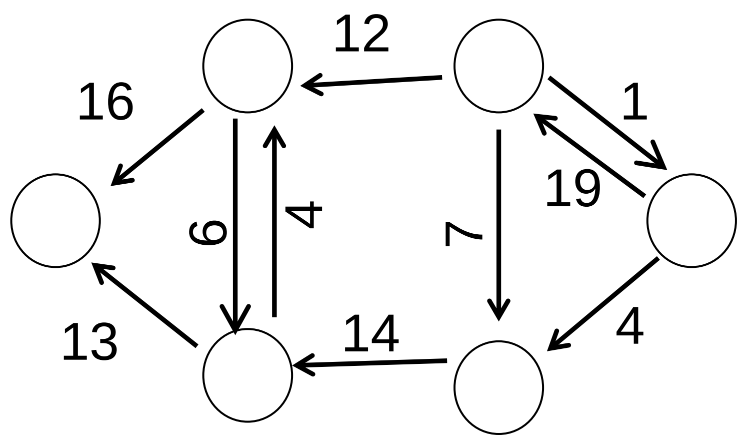
PUSH(x,w)



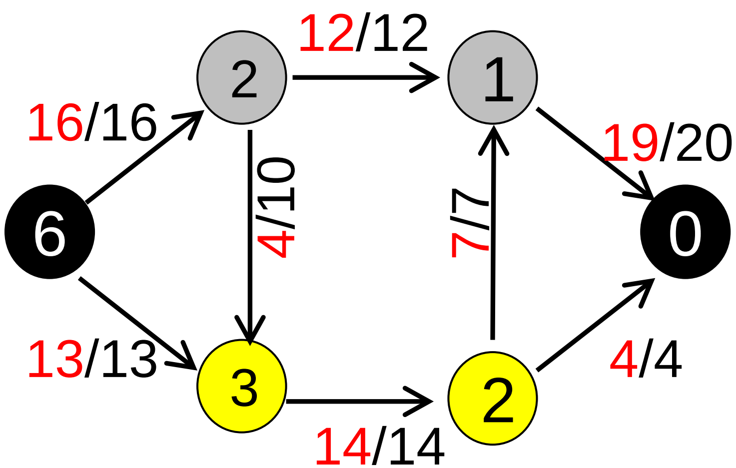


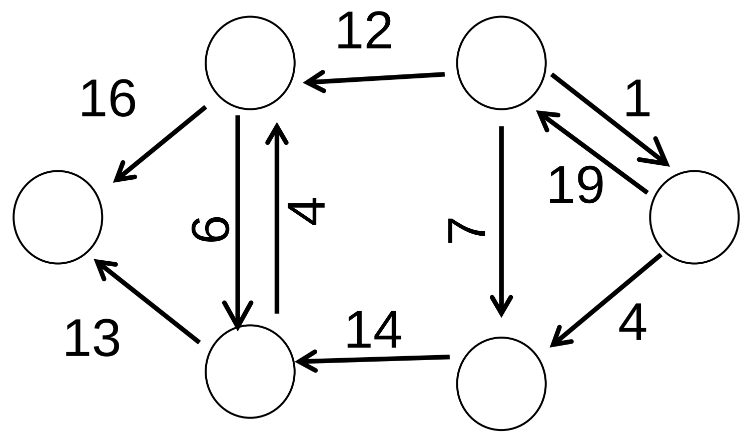
PUSH(w,t)



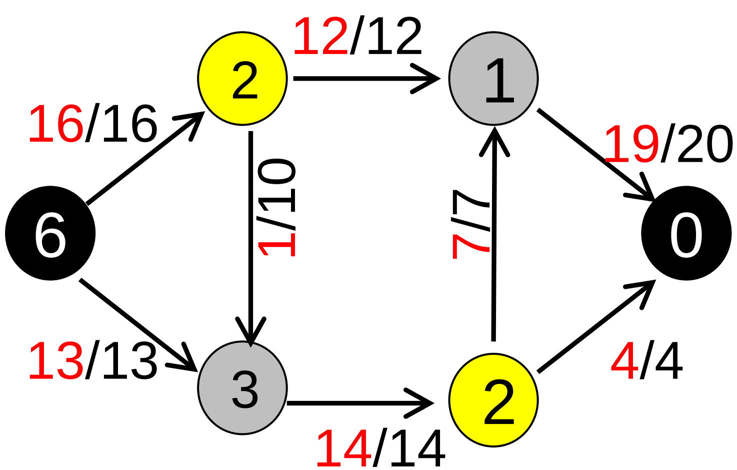


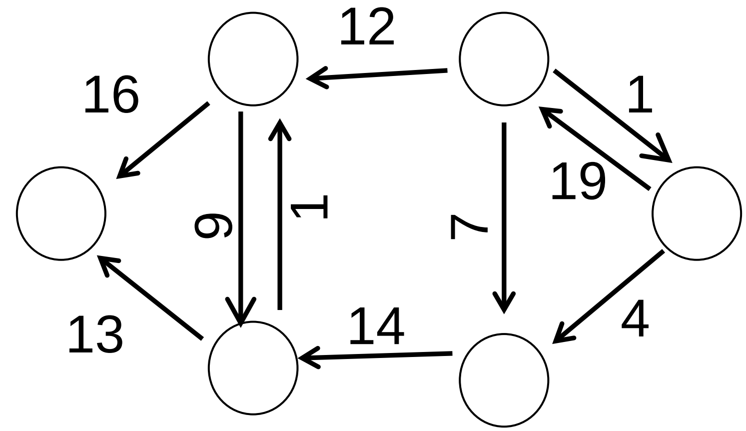
RELABEL(v)



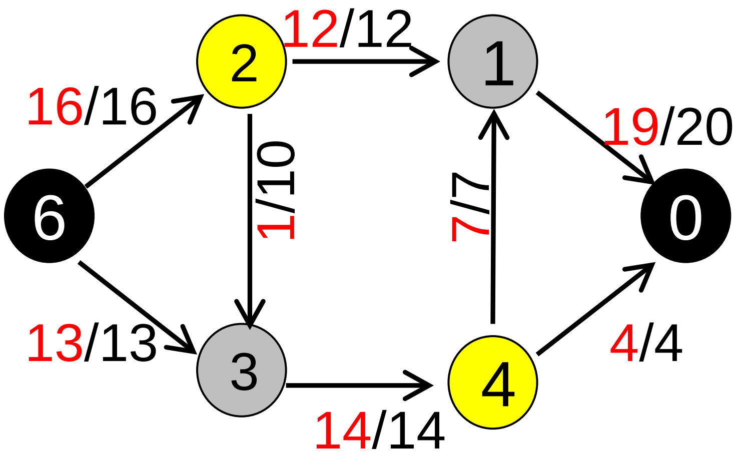


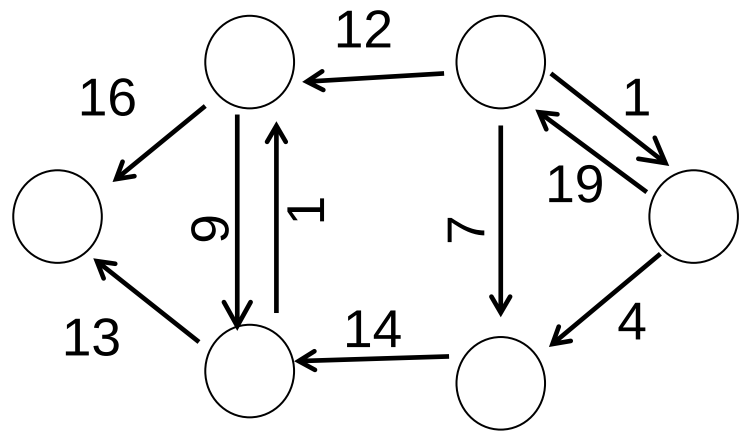
PUSH(v,u)



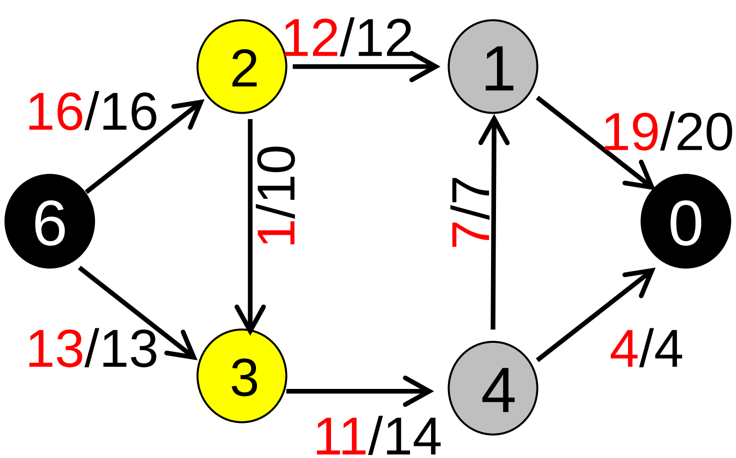


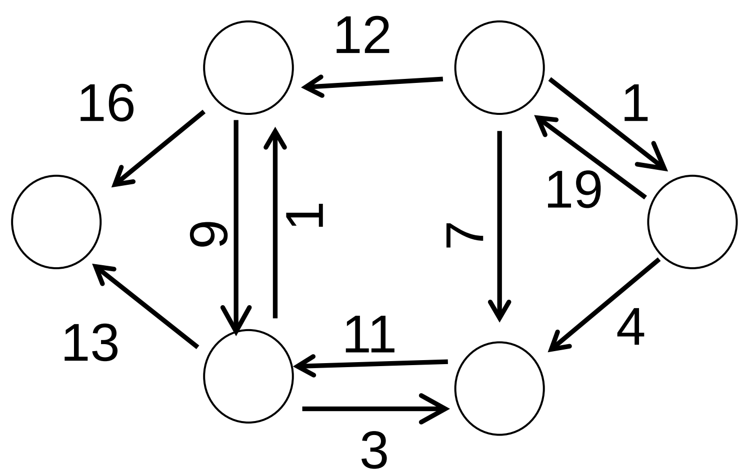
RELABEL(x)



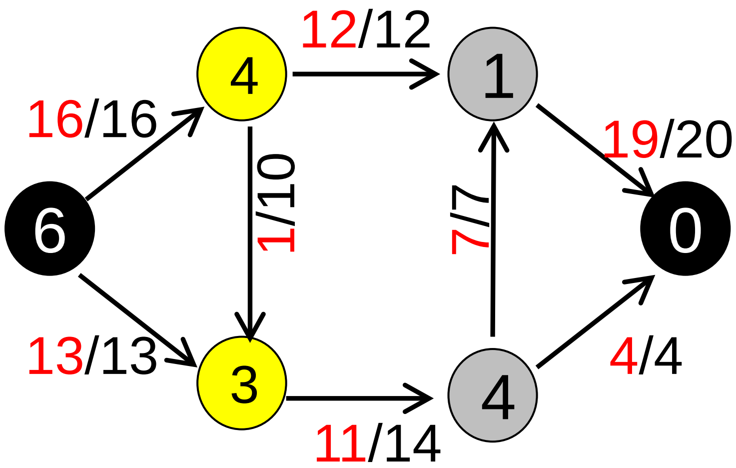


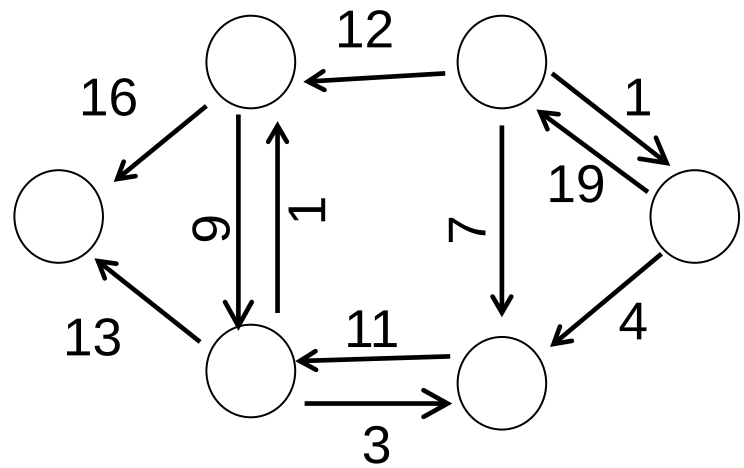
PUSH(x,v)



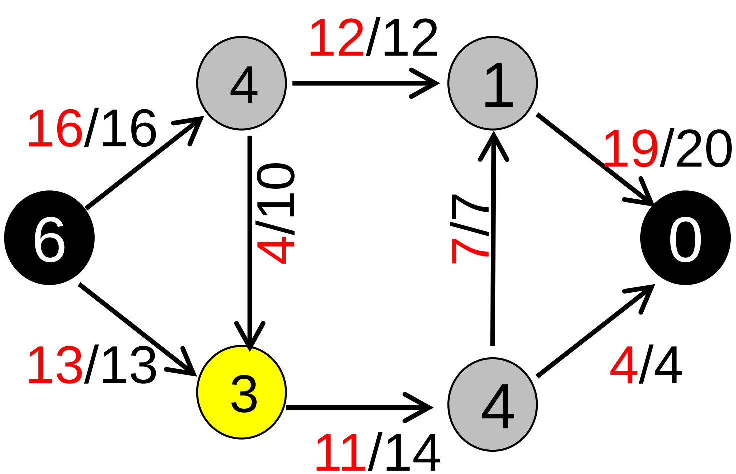


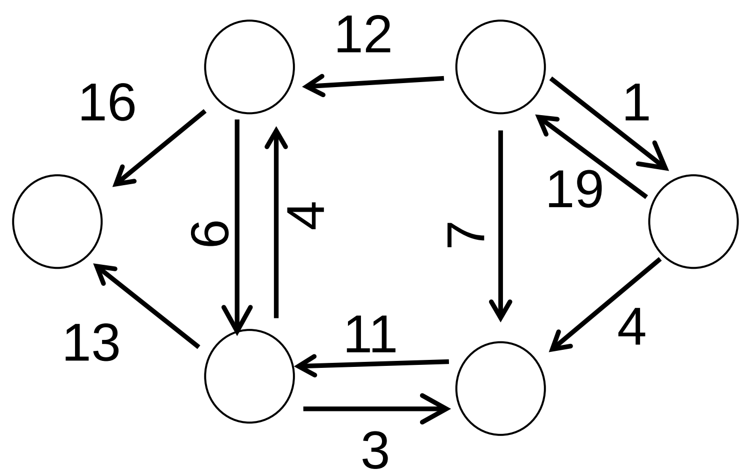
RELABEL(u)



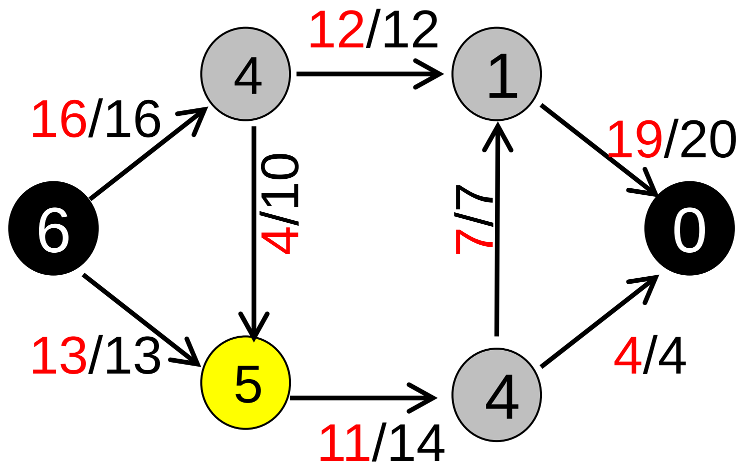


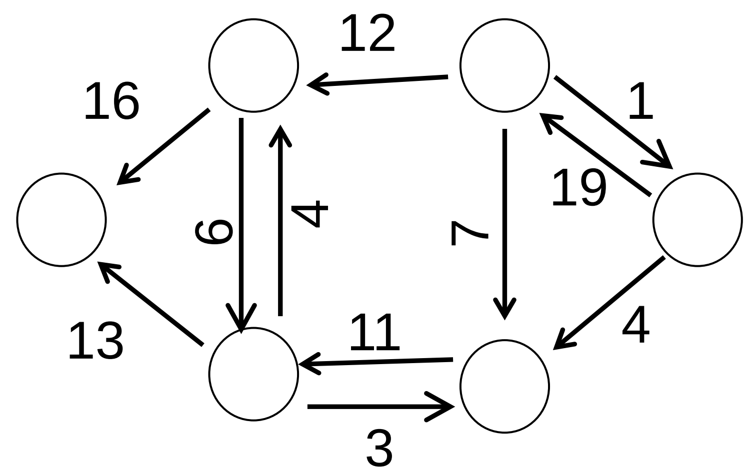
PUSH(u,v)



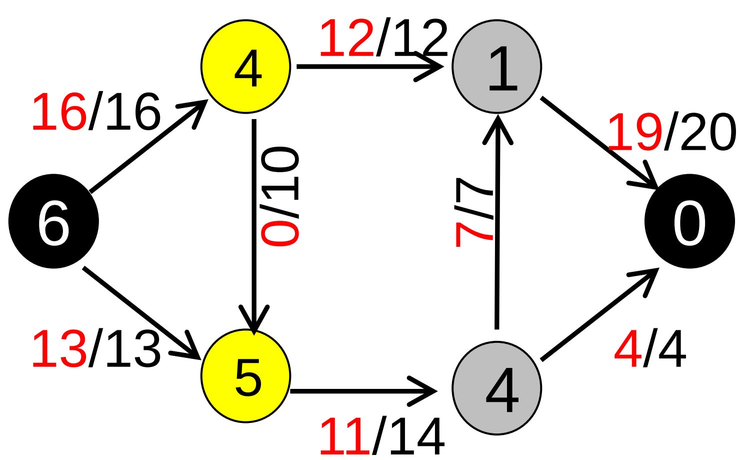


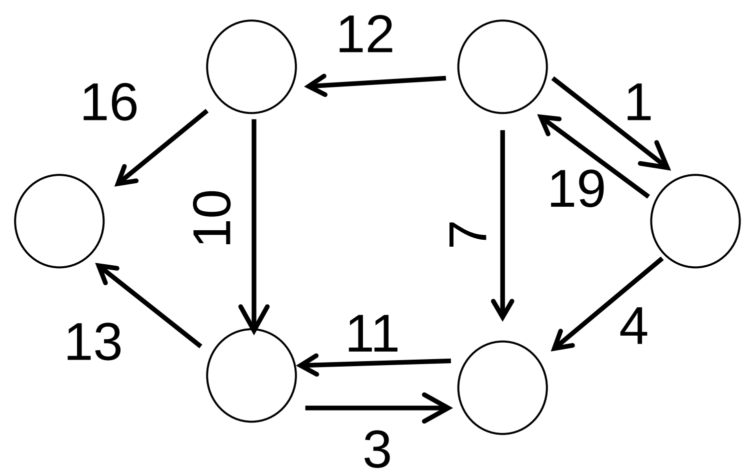
RELABEL(v)



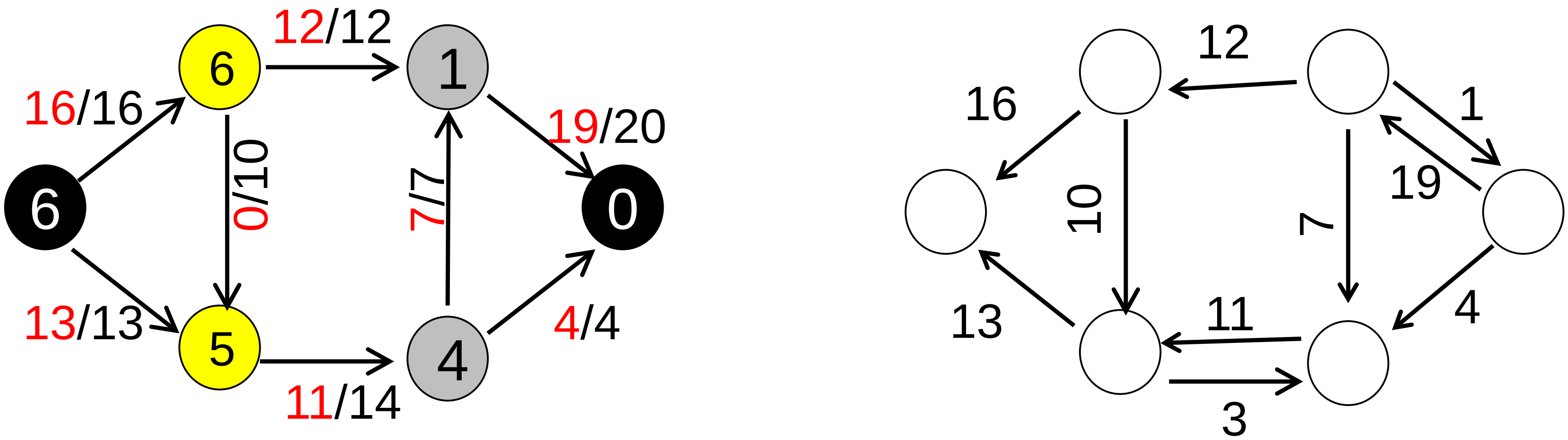


PUSH(v,u)

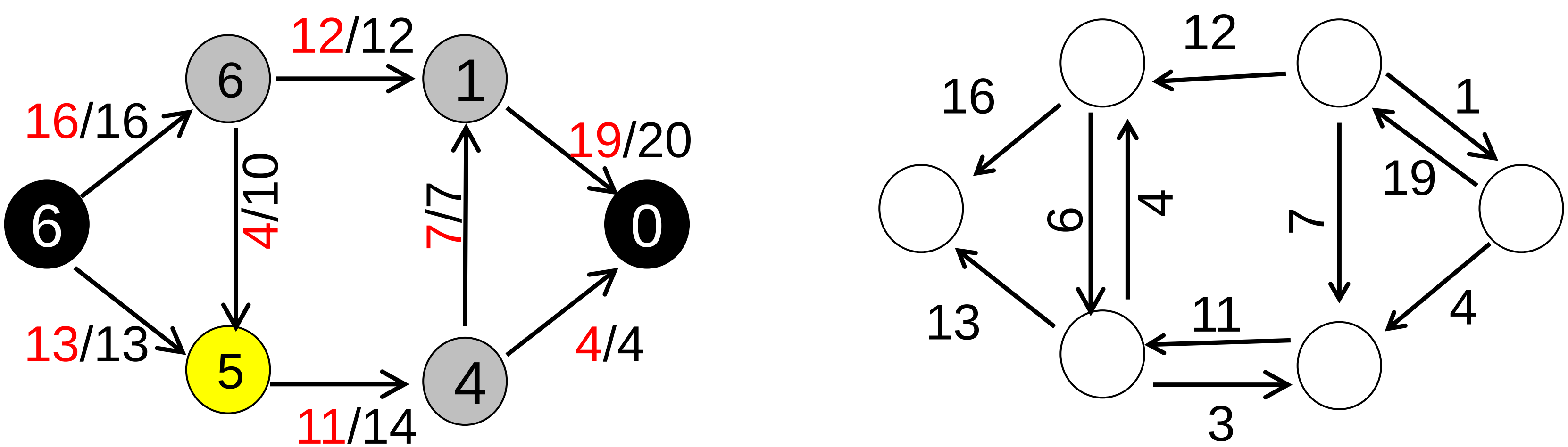




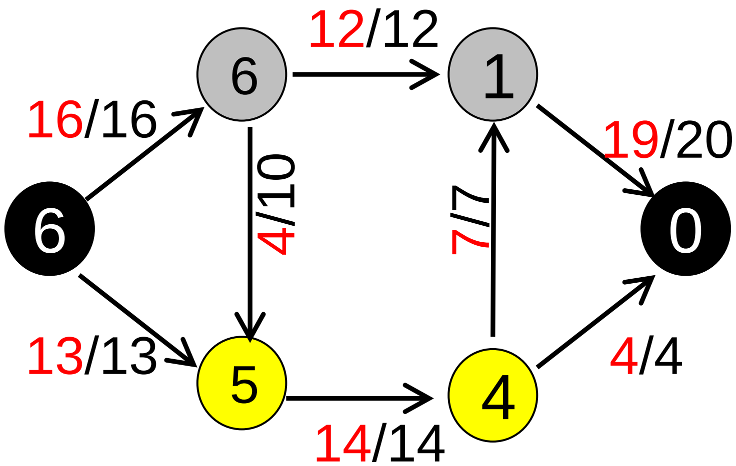
RELABEL(u)

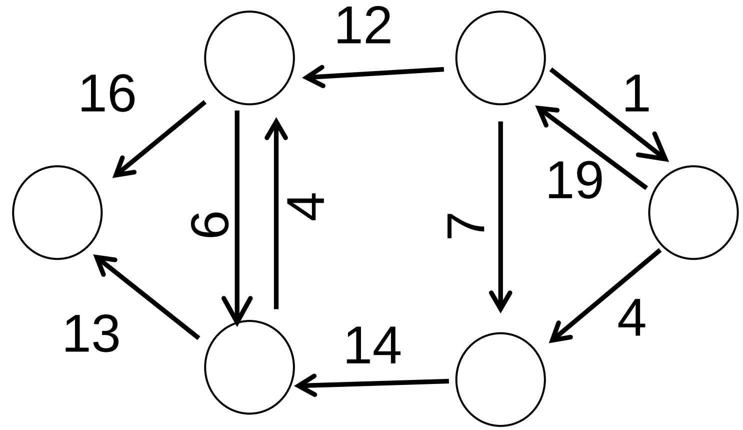


PUSH(u,v)

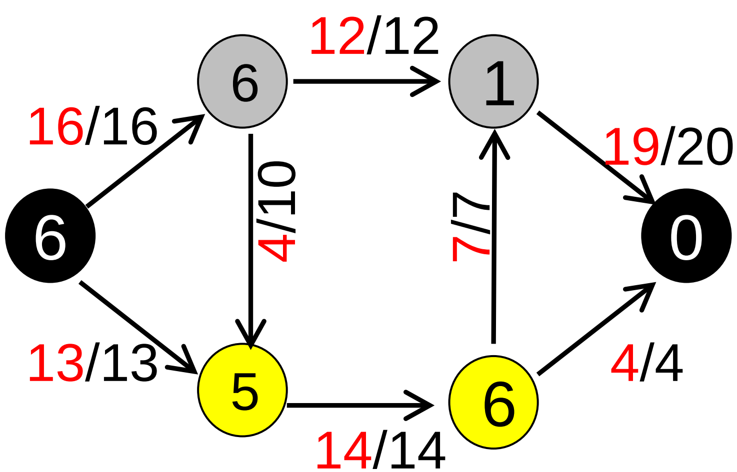


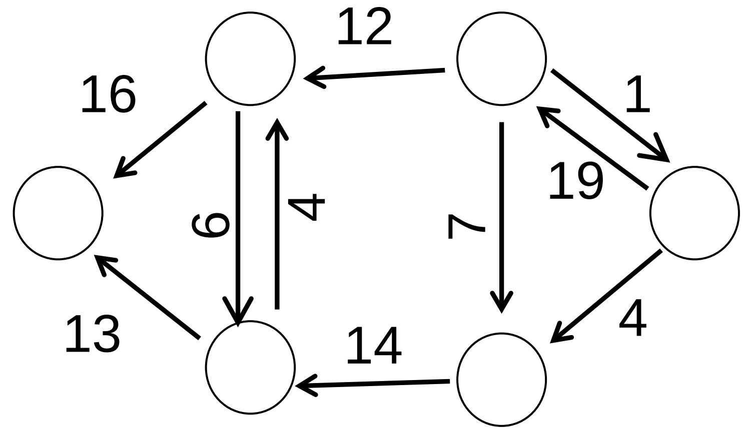
PUSH(u,x)



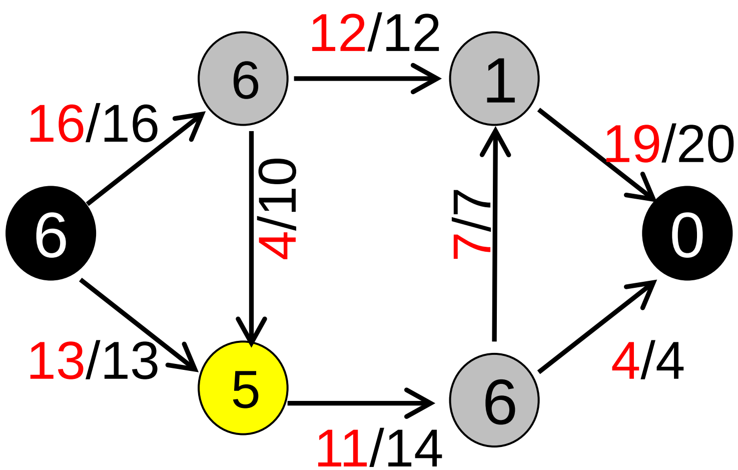


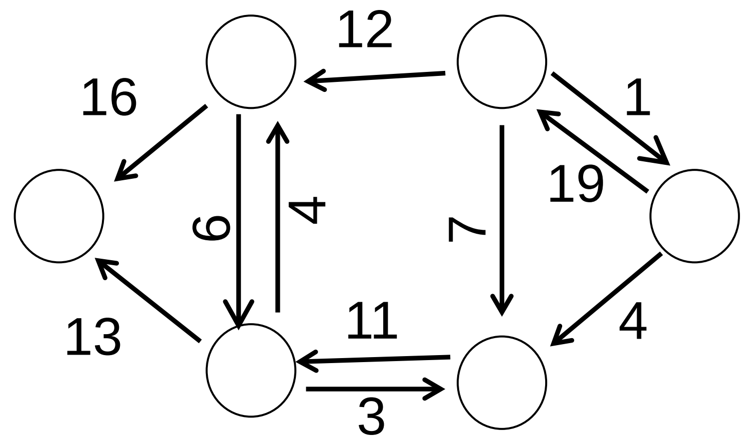
RELABEL(x)



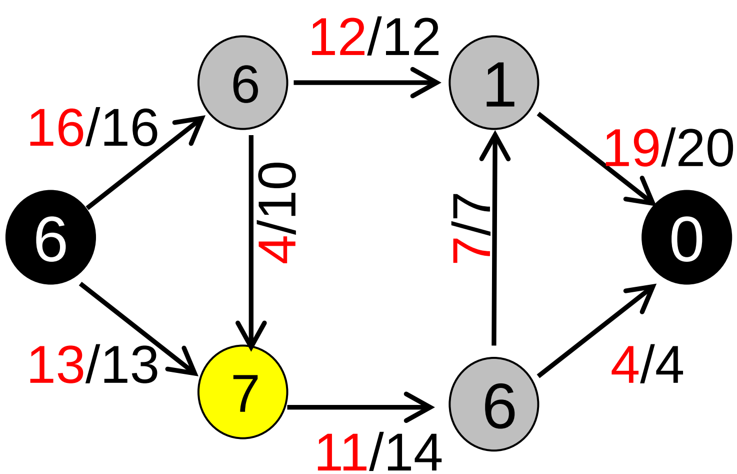


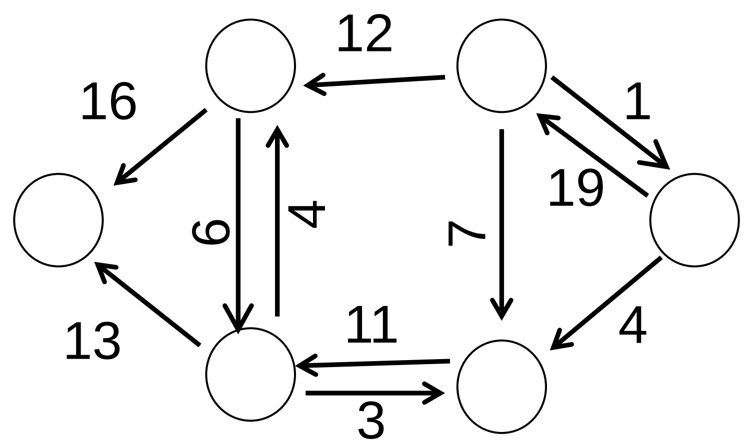
PUSH(x,v)



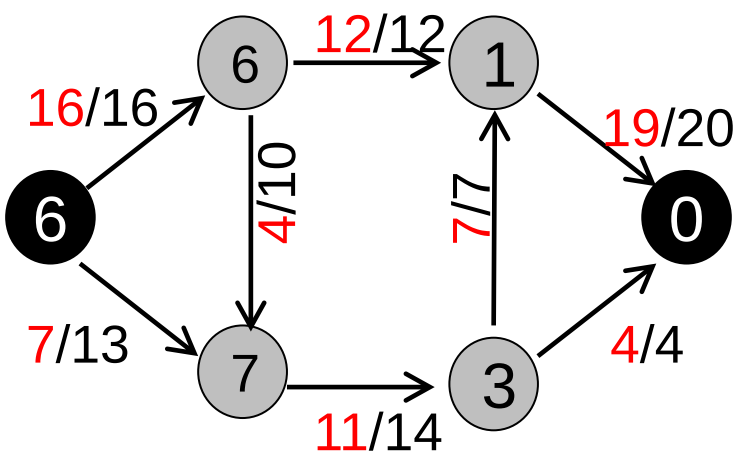


RELABEL(v)





PUSH(v,s)



No active node, so stop

Maximum flow $|f| = 23$

Proof of Correctness (Outline)

- Claim 1: Vertex heights never decrease
  - PUSH does not change h, and RELABEL only increases it
- Claim 2: PUSH(u,v) and RELABEL(u) maintain the properties of the height function
  - PUSH(u,v) pushes flow along $(u,v) \in E_f$, so there may be two possibilities:
    - It may add the edge (v,u) to E_f. Since PUSH(u,v) occurred, so $h(u) = h(v) + 1$ before the push. PUSH does not change h. So $h(v) = h(u) - 1 < h(u) + 1$ after the push, which satisfies the height function property for the edge (v,u)
    - It may remove the edge (u,v) from E_f. Then the constraint does not apply to (u,v) anyway (as height function properties apply only for edges in E_f)

- RELABEL(u) increases h(u)
  - Outgoing edges from u in G_f: Just before relabel, $h(u) \leq h(v)$ for any edge $(u,v) \in E_f$. Relabel increases h(u) to 1 + minimum of the h(v)'s. So $h(u) \leq h(v) + 1$ for any edge $(u,v) \in E_f$. This satisfies the height function property.
  - Incoming edges to u in G_f: For any edge $(w,u) \in E_f$, just before RELABEL, $h(w) \leq h(u) + 1$ (as the height function was satisfied before the operation). So just after RELABEL, $h(w) < h(u) + 1$ trivially as h(u) is increased.

- Claim 3: For a preflow f, there is no path from s to t in the residual graph G_f
  - Can show by contradiction
    - Assume that such a path p exists. By the property of the height function, for any edge $(u,v) \in E_f$, $h(u) \le h(v) + 1$. Applying this to successive vertices of the path p, it is easy to show that $h(s) \le h(t) + k$, where k is the length of the path. But that means h(s) cannot be $|V|$, as $h(t) = 0$ and $k < |V|$. This is a contradiction.

- Claim 4: PUSH operations maintains the properties of a preflow
  - Since PUSH increases flow from u to v by $d_f(u,v)$ = min(e(u), $c_f(u,v)$) amount, it cannot make e(u) negative or exceed the capacity c(u,v). So the preflow f after the PUSH satisfies the capacity constraint and the flow constraint. It obviously satisfies the skew symmetry constraint (see pseudocode). So if f is a preflow before the PUSH, it remains a preflow after the PUSH

Theorem: If the algorithm terminates, the preflow f at the end is a maximum flow.

Proof Outline:

- Initial f is a preflow.
- RELABEL operations do not affect flow, so a preflow remains a preflow
- PUSH operations also maintain preflows (Claim 4)
- Termination means for any vertx in V – {s,t} , PUSH and RELABEL are not applicable, which implies all vertices in V – {s,t} must have excess 0. So it is a flow, and it will not change (as no more PUSH and RELABEL can be done)
- We know that there is no path from s to t in G_f (Claim 3)
- So there is no augmenting path in the residual graph, so by max-flow min-cut theorem, f is a maximum flow.

- Are we done with correctness proof?

- No. We have proved “If” it terminates, f is a maximum flow
- We have not proved that it “does” terminate
  - What if there is always one or more vertices with excess > 0, and an infinite sequence of PUSH and RELABEL operations occur?
- So we have to prove that the algorithm terminates
  - We can prove termination by showing that the number of PUSH and the number of RELABEL operations are bounded

- We will omit this proof, will just note that the following can be proved:
  - At any time t during the execution of the algorithm, $h(u) \leq 2|V| - 1$
  - Then, the number of RELABEL operations is bounded by $(2|V| - 1)(|V| - 2) < 2|V|^2$
  - Number of saturating pushes is $< 2|V||E|$
  - Number of nonsaturating pushes is $< 4|V|^2(|V| + |E|)$
  - Therefore time complexity $= O(|V|^2E)$
    - Can implement each PUSH and RELABEL in $O(1)$ time

- Note that the algorithm we presented is "generic" in the sense that it can apply PUSH and RELABELs in any order
- There are different implementations that apply these operations in different specific orders to get better complexity
  - Relabel-to-front
  - FIFO
  - Highest-label
  - .....