Chapter 8: Network Flow

Algorithms

Flow networks | Residual networks and augmenting paths | Cuts | Max-flow min-cut | Ford-Fulkerson | Edmonds-Karp | Dinic | Push–relabel

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/algorithms/slides/Chapter_08_Network_Flow.html

Learning Outcomes

Modeling

Model a problem as a flow network with a source, a sink and capacitated edges; write down the capacity and conservation constraints.

Residual networks and cuts

Compute the residual network, an augmenting path and its bottleneck; find the capacity of an s–t cut and apply the max-flow min-cut theorem.

Algorithms

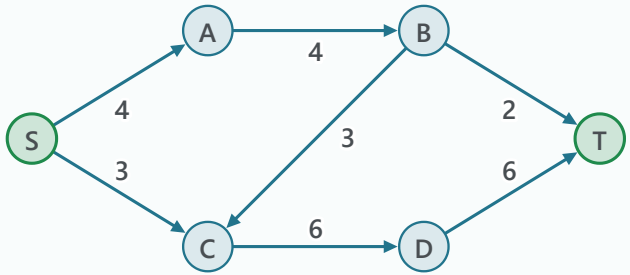
Run Ford-Fulkerson, Edmonds-Karp, Dinic and push–relabel step by step on an example network.

Analysis

Explain where the bounds $O(E \cdot |f^*|)$, $O(V \cdot E^2)$ and $O(V^2 \cdot E)$ come from; compare the algorithms and pick the right one.

The Maximum-Flow Problem

- A directed graph has a **source** vertex s and a **sink** vertex t.
- Each edge has a **capacity** c(u, v) ≥ 0: the most that can pass through it per unit time.
- Goal: find the **maximum flow** that can be sent from source to sink without exceeding any capacity.



Edge labels are capacities. The maximum flow in this network is 7.

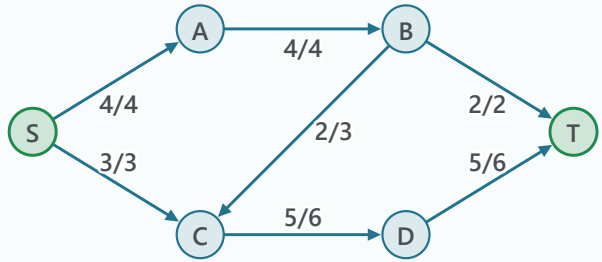
Water in a pipeline, traffic on roads and packets in a computer network are all flows. Each pipe (edge) carries at most its capacity; nothing accumulates at the junctions (vertices).

Flows and Their Constraints

A flow is a function that assigns a number f(u, v) to every edge:

1. **Capacity constraint:** $0 \le f(u, v) \le c(u, v)$.
2. **Flow conservation:** for every v other than s and t,
   flow in = flow out.

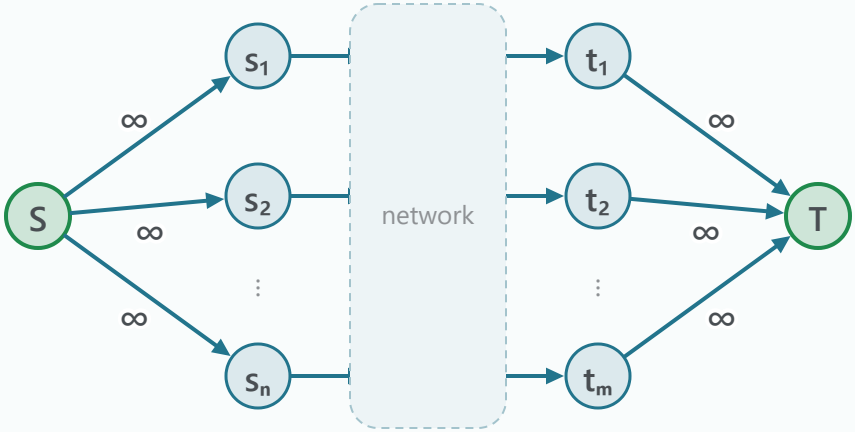
Value of the flow: $|f| = \Sigma f(s, v) - \Sigma f(v, s)$ (net flow out of the source = net flow into the sink).



Label: flow/capacity. 3 + 2 = 5 enters C and 5 leaves. |f| = 4 + 3 = 7.

An edge with **f = c** is **saturated**: no more flow can go through it.

Several Sources and Sinks

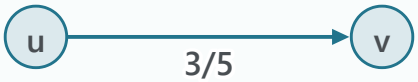


- With several sources (s_1 … s_n) and sinks (t_1 … t_m), add a new **super source** S and **super sink** T.
- Give the edges S → s_i and t_j → T **unlimited (∞) capacity**; the problem becomes a single-source, single-sink maximum flow.

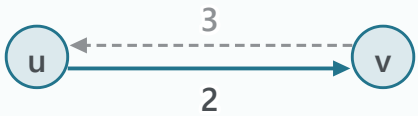
The Residual Network

- The residual network G_f shows how much flow **can still be sent** under the current flow f.
- **Forward edge:** $c_f(u, v) = c(u, v) - f(u, v)$: capacity minus flow; if the flow is below capacity, the leftover is positive.
- **Reverse edge:** $c_f(v, u) = f(u, v)$: flow already sent **can be taken back** (cancelled).
- Only edges with $c_f > 0$ belong to G_f; augmenting paths are searched for in this network.

Flow network: f/c = 3/5



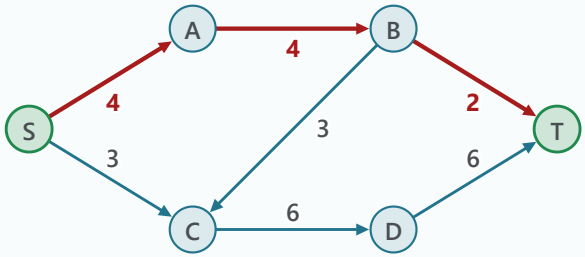
Residual network: 5 − 3 = 2 forward, 3 back



Solid arrow: forward (remaining) capacity · dashed arrow: reverse edge. Without reverse edges the algorithm cannot undo its first bad choice.

Augmenting Paths

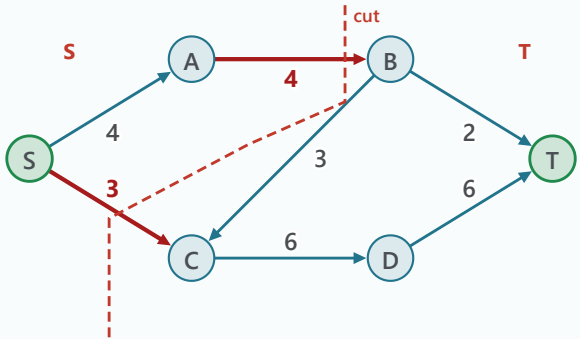
- A directed path **from source to sink** in the residual network; it carries **extra flow** from source to sink.
- Every edge on the path has positive residual capacity: it carries **less** flow than its capacity (or is a reverse edge).
- **Bottleneck**: $c_f(p) = \min c_f(u, v)$, $(u, v) \in p$. That much flow is added along the path.
- Usually found with **DFS** or **BFS**; when no augmenting path is left, the flow is maximum.



p = S → A → B → T: min(4, 4, 2) = **2**.

Cuts

- An **s–t cut** (S, T) splits the vertices in two with s ∈ S and t ∈ T; removing the cut edges disconnects the graph.
- **Cut capacity:** c(S, T) = Σ c(u, v), u ∈ S, v ∈ T: only edges going **from S to T** count.
- For any flow, the net flow across a cut equals |f|; hence **|f| ≤ c(S, T)**.
- **Minimum cut** (min-cut): a cut of minimum capacity (the *sum* of capacities, not the *number* of edges).



S = {S, A}: c(S, T) = 4 + 3 = **7** (minimum).

The Max-Flow Min-Cut Theorem

For a flow f, the following are **equivalent** (Ford and Fulkerson, 1956):

1. f is a maximum flow.
2. The residual network G_f has no augmenting path.
3. $|f| = c(S, T)$ for some cut (S, T).

Consequence: maximum flow value = minimum cut capacity. A min cut is found by taking S to be the **vertices reachable** from s in the final residual network.

S side (example network)	c(S, T)
{S}	**7**
{S, A}	**7**
{S, A, B}	8
{S, A, B, C, D}	8
{S, C}	10
{S, B, D}	18

The smallest of the 16 cuts is 7; since the flow found also has value 7, it is a maximum flow.

Network-Flow Algorithms

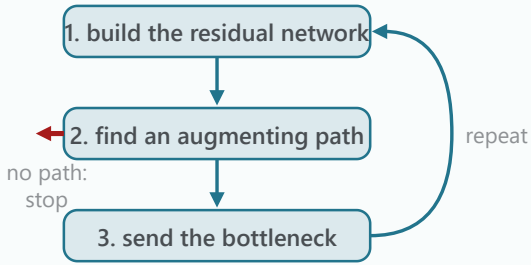
Goal: send as much flow as possible from source to sink under the constraints.

Algorithm	Approach	Time	Note
Ford-Fulkerson (1956)	any augmenting path (usually DFS)	$O(E \cdot \lvert f^* \rvert)$	simple; time depends on the flow value
Edmonds-Karp (1972)	shortest augmenting path (BFS)	$O(V \cdot E^2)$	the polynomial-time form of Ford-Fulkerson
Dinic (1970)	level graph + blocking flow	$O(V^2 \cdot E)$	faster than Edmonds-Karp; very efficient with unit capacities
Push–relabel (1988)	preflow + heights, local operations	$O(V^2 \cdot E)$	no path search; FIFO version $O(V^3)$

$\lvert f^* \rvert$: the value of a maximum flow. The Ford-Fulkerson bound holds for integer capacities.

The Ford-Fulkerson Method

- Developed by L. R. Ford Jr. and D. R. Fulkerson (1956).
- Finds the maximum flow between two vertices of a weighted (capacitated) directed graph: a graph problem that models the **most water** that can be sent from source to sink through pipes of given capacity.
- **Principles:** the network is a directed graph; each edge has a capacity; all flows start at **0**; augmenting paths are found until the flow is maximum.
- It is a **method**, since it does not say *how* to find the augmenting path: Ford-Fulkerson usually uses **DFS**, Edmonds-Karp uses **BFS**.



Ford-Fulkerson: The Steps

0. Set f(u, v) = 0 on every edge; the residual network = the original network.

The flow starts at zero, so each edge's residual capacity is its own capacity.

Ford-Fulkerson: The Steps

0. Set f(u, v) = 0 on every edge; the residual network = the original network.

The flow starts at zero, so each edge's residual capacity is its own capacity.

1. Find an **augmenting path** p from source to sink in the residual network (edges with $c_f > 0$).

The path is found with DFS (Ford-Fulkerson) or BFS (Edmonds-Karp).

Ford-Fulkerson: The Steps

0. Set f(u, v) = 0 on every edge; the residual network = the original network.

The flow starts at zero, so each edge's residual capacity is its own capacity.

1. Find an **augmenting path** p from source to sink in the residual network (edges with $c_f > 0$).

The path is found with DFS (Ford-Fulkerson) or BFS (Edmonds-Karp).

2. Compute the bottleneck: $\delta = \min c_f(u, v)$, $(u, v) \in p$.

The narrowest edge limits the path's flow.

Ford-Fulkerson: The Steps

0. Set f(u, v) = 0 on every edge; the residual network = the original network.

The flow starts at zero, so each edge's residual capacity is its own capacity.

1. Find an **augmenting path** p from source to sink in the residual network (edges with $c_f > 0$).

The path is found with DFS (Ford-Fulkerson) or BFS (Edmonds-Karp).

2. Compute the bottleneck: $\delta = \min c_f(u, v)$, $(u, v) \in p$.

The narrowest edge limits the path's flow.

3. Send δ units along the path: f(u, v) += δ, and for the reverse edge f(v, u) −= δ; |f| += δ.

Reverse edges are updated; at least one edge on the path becomes saturated.

Ford-Fulkerson: The Steps

0. Set $f(u, v) = 0$ on every edge; the residual network = the original network.

The flow starts at zero, so each edge's residual capacity is its own capacity.

1. Find an **augmenting path** p from source to sink in the residual network (edges with $c_f > 0$).

The path is found with DFS (Ford-Fulkerson) or BFS (Edmonds-Karp).

2. Compute the bottleneck: $\delta = \min c_f(u, v)$, $(u, v) \in p$.

The narrowest edge limits the path's flow.

3. Send δ units along the path: $f(u, v) \mathrel{+}= \delta$, and for the reverse edge $f(v, u) \mathrel{-}= \delta$; $|f| \mathrel{+}= \delta$.

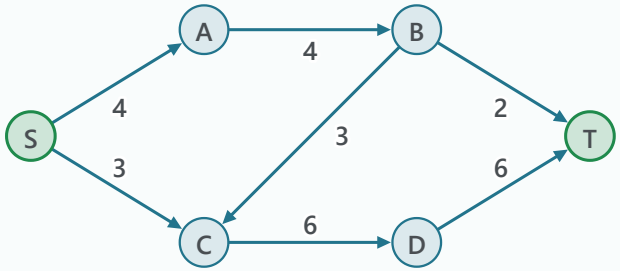
Reverse edges are updated; at least one edge on the path becomes saturated.

4. Repeat 1–3 until **no** augmenting path is left; $|f|$ is maximum.

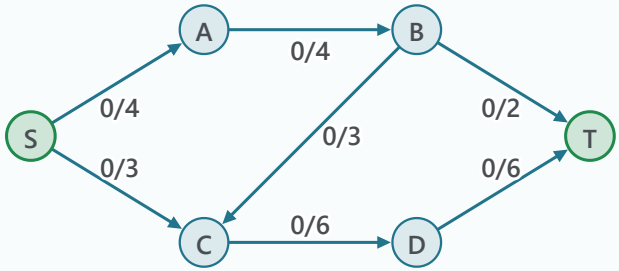
The loop ends when the sink is unreachable (no augmenting path is left); the max-flow min-cut theorem guarantees the result is optimal.

Ford-Fulkerson Example 1

Residual network G_f



Flow f (f/c)

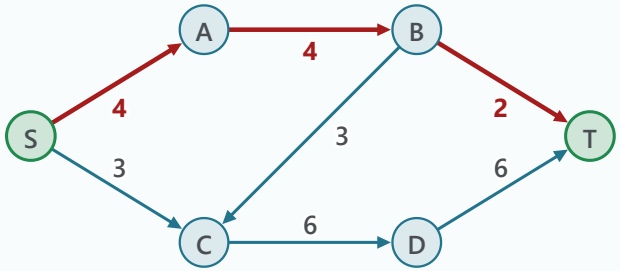


|f| = 0

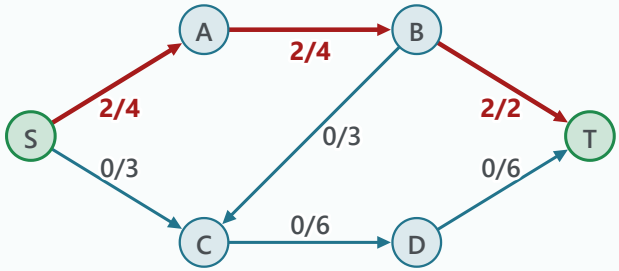
Start: all flows are 0, and the residual network is identical to the original.

Ford-Fulkerson Example 1

Residual network: path p



Flow: +2

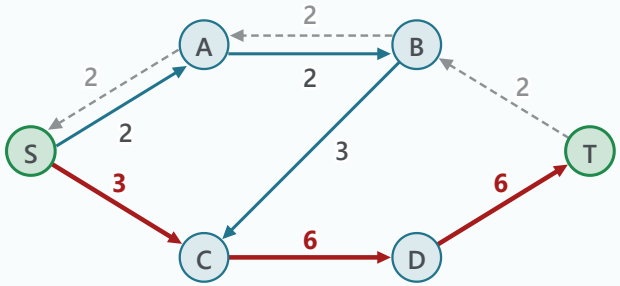


|f| = 2

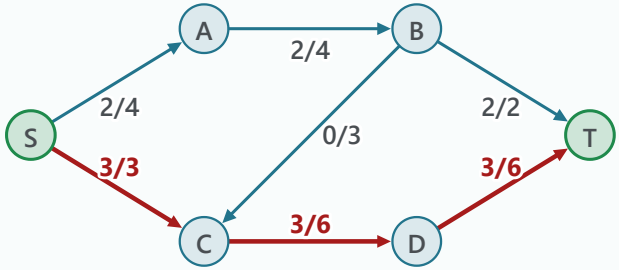
First augmenting path p = {S, A, B, T}: bottleneck **2** units. |f| = 2.

Ford-Fulkerson Example 1

Residual network: path p



Flow: +3

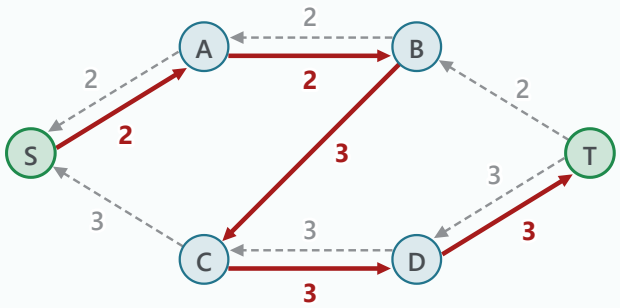


|f| = 5

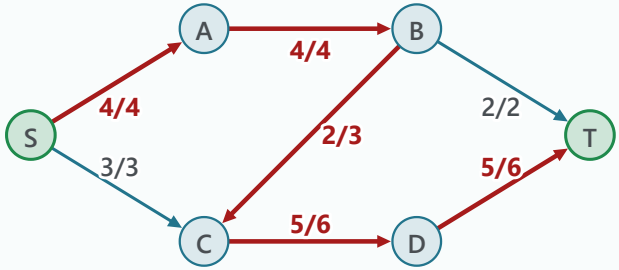
Second augmenting path p = {S, C, D, T}: bottleneck **3** units. |f| = 5.

Ford-Fulkerson Example 1

Residual network: path p



Flow: +2

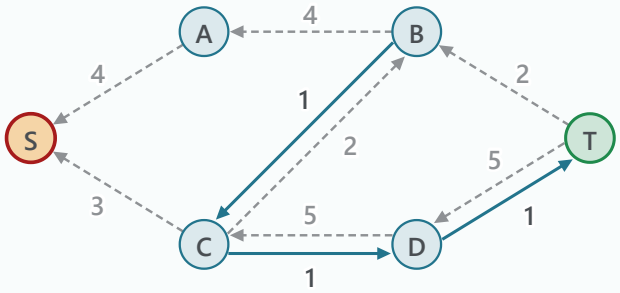


|f| = 7

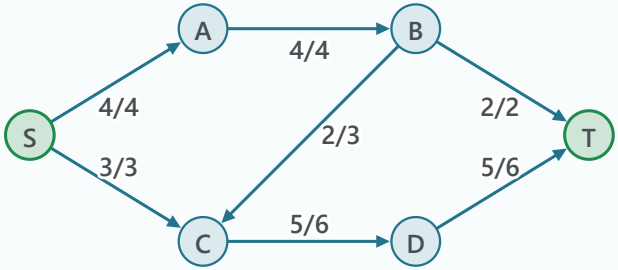
Third augmenting path p = {S, A, B, C, D, T}: bottleneck **2** units. |f| = 7.

Ford-Fulkerson Example 1

Final residual network



Maximum flow



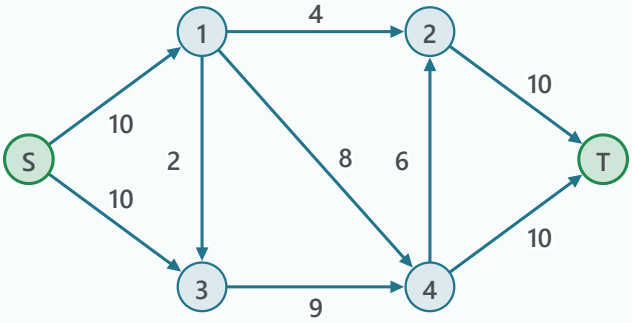
|f| = 7

Both edges out of S are saturated: in the residual network S cannot reach anything.

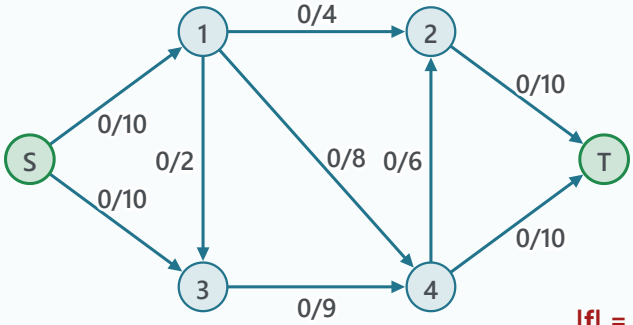
Maximum flow 7; min cut ({S}, the rest) = 4 + 3.

Ford-Fulkerson Example 2: Using a Reverse Edge

Residual network G_f



Flow f (f/c)

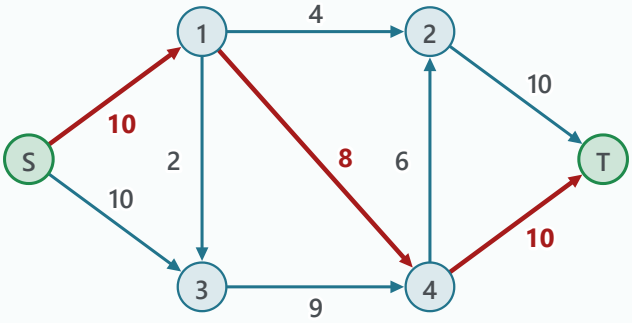


|f| = 0

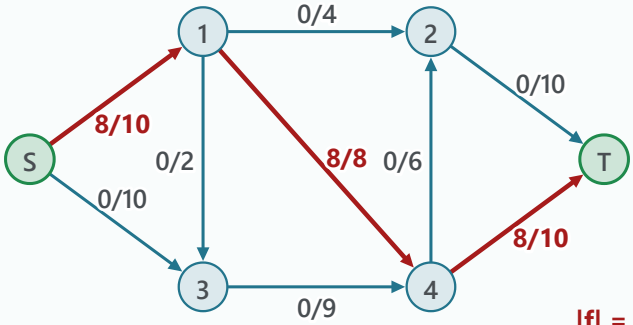
Source S, sink T. All flows start at 0.

Ford-Fulkerson Example 2: Using a Reverse Edge

Residual network: path p (+8)



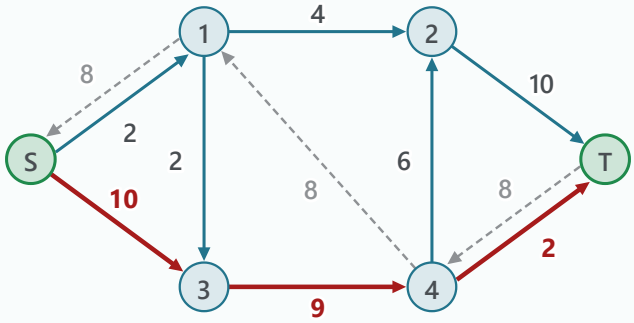
Flow f



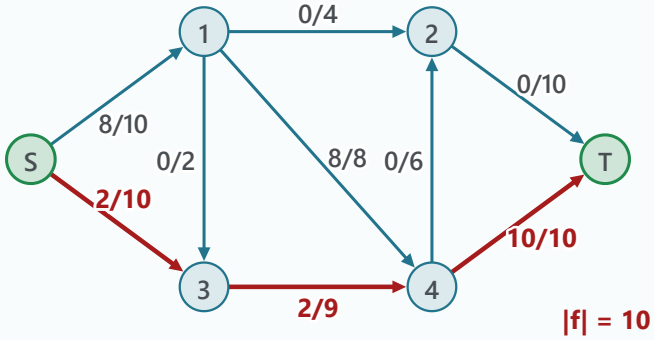
p = S → 1 → 4 → T: bottleneck 1 → 4 (8). |f| = 8.

Ford-Fulkerson Example 2: Using a Reverse Edge

Residual network: path p (+2)



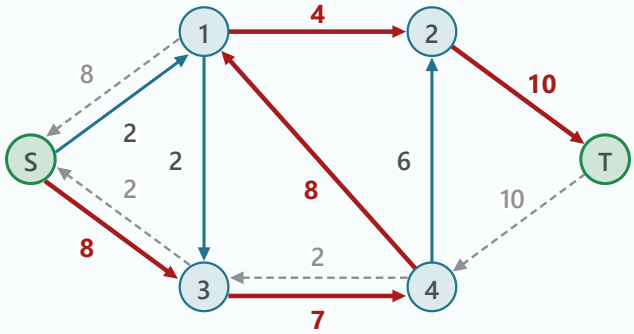
Flow f



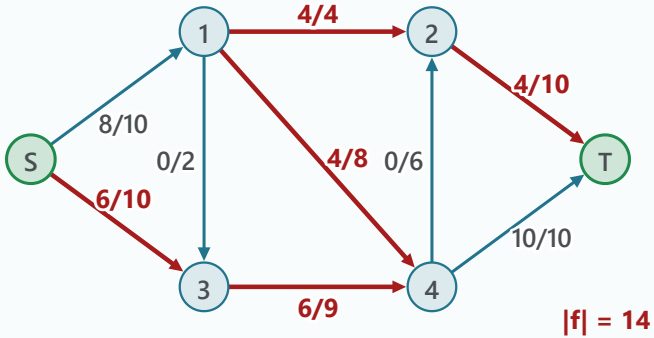
p = S → 3 → 4 → T: only 2 is left on 4 → T. |f| = 10.

Ford-Fulkerson Example 2: Using a Reverse Edge

Residual network: path p (+4)



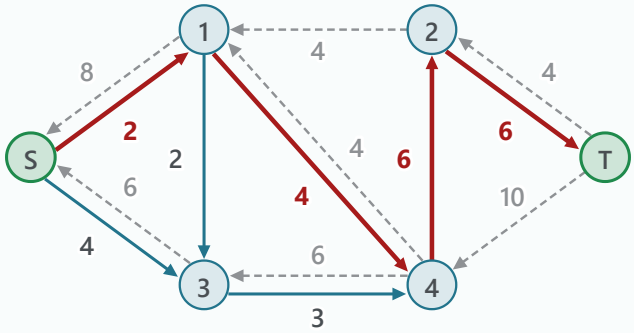
Flow f



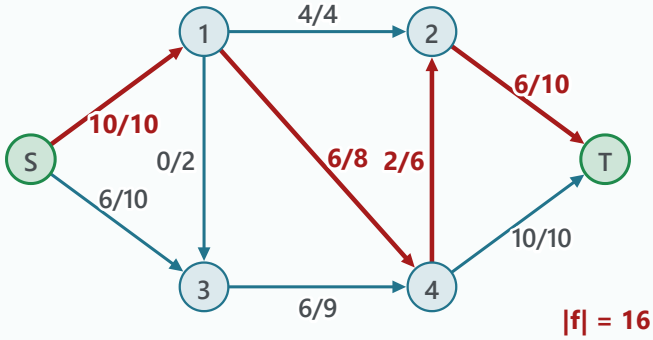
p = S → 3 → 4 → **1** → 2 → T: the **reverse edge 4 → 1** is used; the flow on 1 → 4 drops from 8 to 4, and the 4 units leaving 1 are rerouted to 1 → 2. |f| = 14.

Ford-Fulkerson Example 2: Using a Reverse Edge

Residual network: path p (+2)



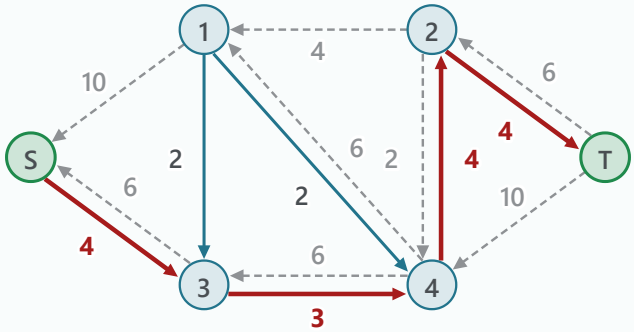
Flow f



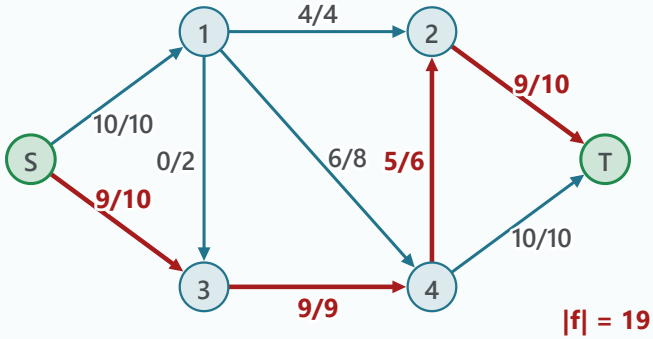
p = S → 1 → 4 → 2 → T: 2 units were left on S → 1. |f| = 16.

Ford-Fulkerson Example 2: Using a Reverse Edge

Residual network: path p (+3)



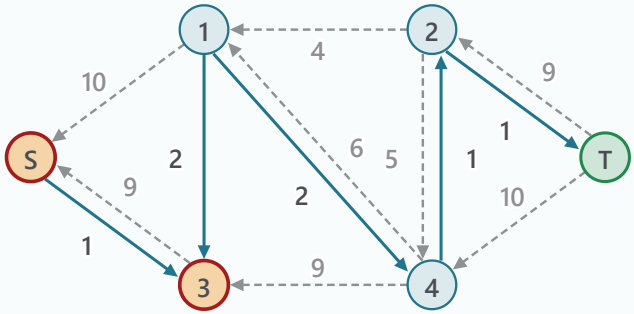
Flow f



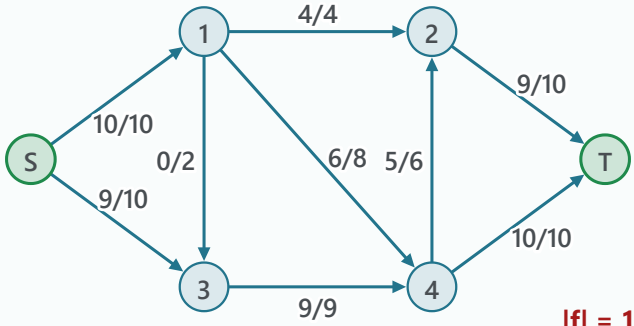
p = S → 3 → 4 → 2 → T: bottleneck 3 → 4 (9 − 6 = 3 left). |f| = 19.

Ford-Fulkerson Example 2: Using a Reverse Edge

Final residual network



Maximum flow



In the residual network S reaches only 3 → min cut ({S, 3}, the rest) = 10 + 9 = 19 = **|f*|**.

Why Are Reverse Edges Needed?

Forward edges only (greedy)

1. S → 1 → 4 → T: 8
2. S → 3 → 4 → T: 2
3. S → 1 → 2 → T: 2
4. S → 3 → 4 → 2 → T: 6

Total **18**. From S only 3 and 4 are reachable; 4's outgoing edges are full. Stuck!

Residual network (with reverse edges)

The first choice sent 8 units on 1 → 4. The reverse edge 4 → 1 lets us **undo** part of that decision: flow from 1 goes to 2, and flow from 3 goes to 4.

Result **19** = min cut. Whatever order the paths are chosen in, the residual network reaches the maximum.

A reverse edge does not really "send flow backward"; it **cancels** flow sent earlier: f(u, v) −= δ.

Ford-Fulkerson: Pseudocode

```
FORD_FULKERSON(G, source, sink):
  flow[u, v] = 0, for each (u, v) in E     // start at 0
  maxFlow = 0
  while an augmenting path exists do:
    path = DFS(G, source, sink, flow)            // edges with c − flow > 0
    delta = ∞                                    // smallest residual on path
    v = sink
    while v ≠ source do:
      u = path.parent[v]
      delta = min(delta, capacity[u, v] − flow[u, v])
      v = u
    v = sink                                     // update flows on the path
    while v ≠ source do:
      u = path.parent[v]
      flow[u, v] += delta
      flow[v, u] −= delta                        // reverse edge
      v = u
    maxFlow += delta
  return maxFlow, flow
```

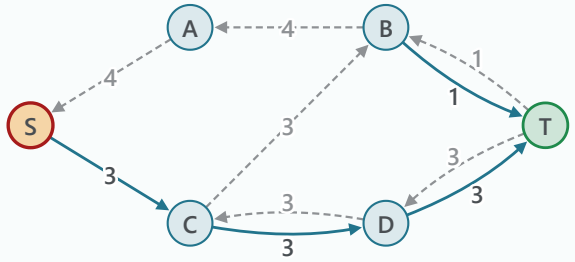
Java: Augmenting Paths with DFS

```java
boolean dfs(int u, int sink,
            boolean[] visited, int[] parent) {
  visited[u] = true;
  if (u == sink) return true;
  for (int v = 0; v < V; v++) {
    if (!visited[v] && res[u][v] > 0) {
      parent[v] = u;
      if (dfs(v, sink, visited, parent))
        return true;
    }
  }
  return false;   // no path to sink from here
}
```

- `res[u][v]`: the residual capacity matrix. It starts as the capacities; reverse edges start at 0.
- Returns `true` once the sink is reached; `parent[]` gives the path backward.
- Adding flow does `res[u][v] -= d` and `res[v][u] += d`: the reverse edge appears automatically.
- One DFS takes $O(V^2)$ on an adjacency matrix, $O(V + E)$ on an adjacency list.

DFS Paths: Dead Ends, Reverse Edges

```
boolean dfs(int u, int sink,
            boolean[] visited, int[] parent) {
  visited[u] = true;
  if (u == sink) return true;
  for (int v = 0; v < V; v++) {
    if (!visited[v] && res[u][v] > 0) {
      parent[v] = u;
      if (dfs(v, sink, visited, parent))
        return true;
    }
  }
  return false;   // no path to sink from here
}
```



Residual network after 2 augmentations (|f| = 4); dashed: reverse edges. `dfs(S, T, ...)` seeks path 3, trying S, A, B, C, D, T in order.

Step 1 / 8

VARIABLES

`u = –` `v = –`

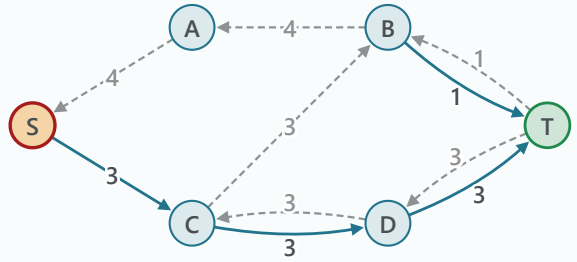
`visited = {}`

CALL STACK

empty

DFS Paths: Dead Ends, Reverse Edges

```
1  boolean dfs(int u, int sink,
2              boolean[] visited, int[] parent) {
3    visited[u] = true;
4    if (u == sink) return true;
5    for (int v = 0; v < V; v++) {
       if (!visited[v] && res[u][v] > 0) {
         parent[v] = u;
         if (dfs(v, sink, visited, parent))
9          return true;
10     }
11   }
12   return false;    // no path to sink from here
13 }
```



S → C has residual capacity 3 > 0 and C is unvisited: parent[C] = S, `dfs(C)`.

Step 2 / 8

VARIABLES

`u = S` `v = C`

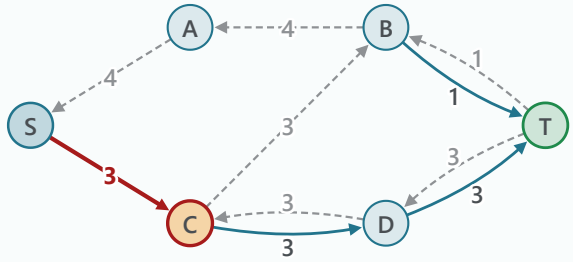
`visited = {S}`

CALL STACK

`dfs(S)`

DFS Paths: Dead Ends, Reverse Edges

```
1  boolean dfs(int u, int sink,
2              boolean[] visited, int[] parent) {
3    visited[u] = true;
4    if (u == sink) return true;
5    for (int v = 0; v < V; v++) {
▶      if (!visited[v] && res[u][v] > 0) {
▶        parent[v] = u;
▶        if (dfs(v, sink, visited, parent))
9          return true;
10     }
11   }
12   return false;    // no path to sink from here
13 }
```



C → B, a **reverse edge** (residual 3), B unvisited: parent[B] = C, `dfs(B)`.

Step 3 / 8

VARIABLES

`u = C` `v = B`

`visited = {S, C}`

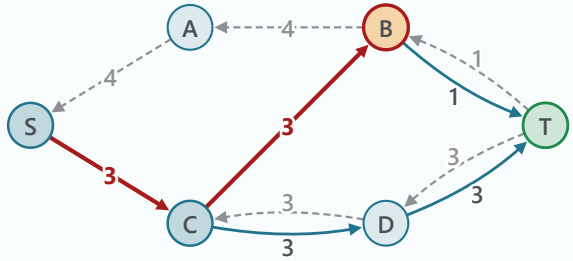
CALL STACK

`dfs(C)`

`dfs(S)`

DFS Paths: Dead Ends, Reverse Edges

```
1  boolean dfs(int u, int sink,
2              boolean[] visited, int[] parent) {
3    visited[u] = true;
4    if (u == sink) return true;
5    for (int v = 0; v < V; v++) {
       if (!visited[v] && res[u][v] > 0) {
         parent[v] = u;
         if (dfs(v, sink, visited, parent))
9          return true;
10     }
11   }
12   return false;    // no path to sink from here
13 }
```



B → A, a **reverse edge** (residual 4), A unvisited: parent[A] = B, `dfs(A)`.

Step 4 / 8

VARIABLES

`u = B` `v = A`

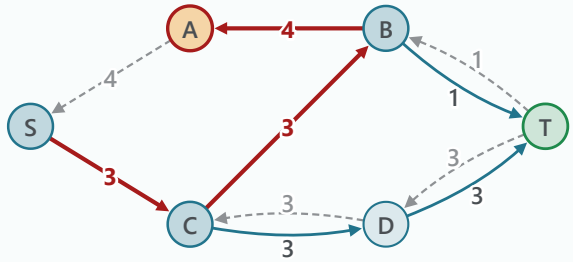
`visited = {S, B, C}`

CALL STACK

- `dfs(B)`
- `dfs(C)`
- `dfs(S)`

DFS Paths: Dead Ends, Reverse Edges

```
1  boolean dfs(int u, int sink,
2              boolean[] visited, int[] parent) {
3    visited[u] = true;
4    if (u == sink) return true;
5    for (int v = 0; v < V; v++) {
6      if (!visited[v] && res[u][v] > 0) {
7        parent[v] = u;
8        if (dfs(v, sink, visited, parent))
9          return true;
10     }
11   }
▶  return false;   // no path to sink from here
13 }
```



A has no usable neighbor: a **dead end**. Return `false`, back up.

Step 5 / 8

VARIABLES

`u = A` `v = –`

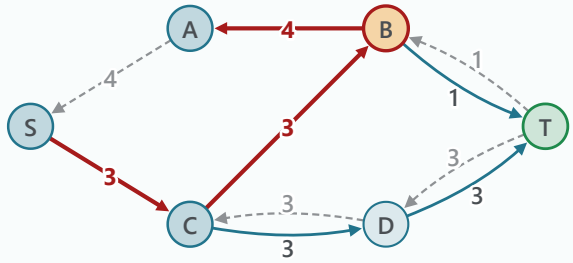
`visited = {S, A, B, C}`

CALL STACK

- `dfs(A)`
- `dfs(B)`
- `dfs(C)`
- `dfs(S)`

DFS Paths: Dead Ends, Reverse Edges

```
1  boolean dfs(int u, int sink,
2              boolean[] visited, int[] parent) {
3    visited[u] = true;
4    if (u == sink) return true;
5    for (int v = 0; v < V; v++) {
▶      if (!visited[v] && res[u][v] > 0) {
▶        parent[v] = u;
▶        if (dfs(v, sink, visited, parent))
9          return true;
10     }
11   }
12   return false;    // no path to sink from here
13 }
```



B → T has residual capacity 1 > 0 and T is unvisited: parent[T] = B, `dfs(T)`.

Step 6 / 8

VARIABLES

`u = B` `v = T`

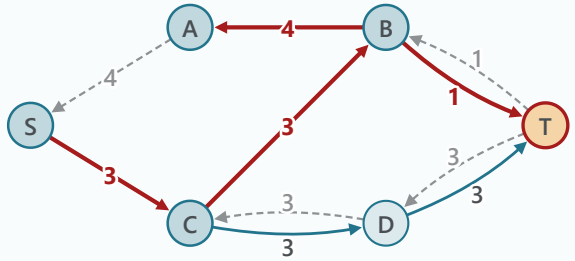
`visited = {S, A, B, C}`

CALL STACK

- `dfs(B)`
- `dfs(C)`
- `dfs(S)`

DFS Paths: Dead Ends, Reverse Edges

```
1  boolean dfs(int u, int sink,
2              boolean[] visited, int[] parent) {
▶   visited[u] = true;
▶   if (u == sink) return true;
5   for (int v = 0; v < V; v++) {
6     if (!visited[v] && res[u][v] > 0) {
7       parent[v] = u;
8       if (dfs(v, sink, visited, parent))
9         return true;
10    }
11  }
12  return false;    // no path to sink from here
13 }
```



u = T = sink: return `true`.

Step 7 / 8

VARIABLES

`u = T` `v = –`

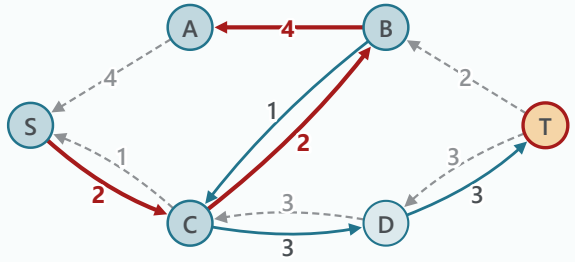
`visited = {S, A, B, C, T}`

CALL STACK

- `dfs(T)`
- `dfs(B)`
- `dfs(C)`
- `dfs(S)`

DFS Paths: Dead Ends, Reverse Edges

```
1  boolean dfs(int u, int sink,
2              boolean[] visited, int[] parent) {
3    visited[u] = true;
4    if (u == sink) return true;
5    for (int v = 0; v < V; v++) {
6      if (!visited[v] && res[u][v] > 0) {
7        parent[v] = u;
▶        if (dfs(v, sink, visited, parent))
▶          return true;
10     }
11   }
12   return false;    // no path to sink from here
13 }
```



`true` passes up every call. Path S → C → B → T, bottleneck 1; now |f| = 5 (updated residuals shown).

Step 8 / 8

VARIABLES

`u = S` `v = C`

`visited = {S, A, B, C, T}`

CALL STACK

empty

Ford-Fulkerson: Complexity Analysis

- The running time depends on the network's structure and its **capacities**.
- Finding one augmenting path (DFS/BFS) and updating the flow: **O(E)**.
- With integer capacities each augmenting path adds **at least 1 unit** → at most **|f*|** iterations.
- Total: **O(E · |f*|)**, where E is the number of edges and |f*| the maximum flow value.

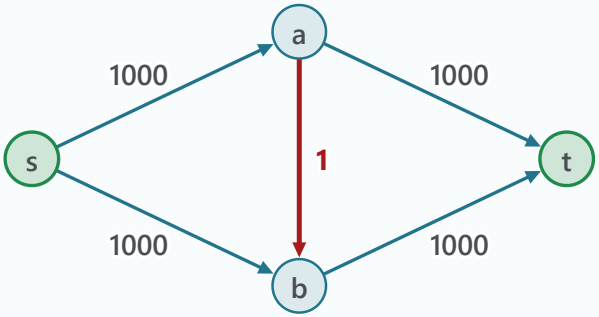
Integrality

If all capacities are integers, the flow stays integral and an integer-valued maximum flow is found. Matching problems use this property.

Caution

O(E · |f*|) is **not polynomial**: |f*| depends on the magnitude of the numbers. With irrational capacities the method may not even terminate.

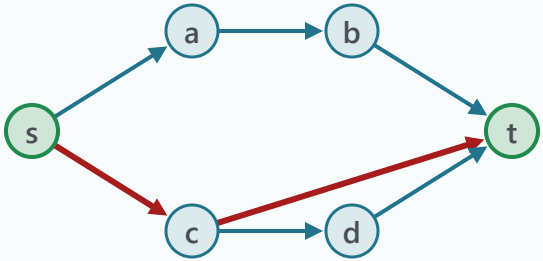
A Bad Choice of Paths: 2000 Augmentations



- Maximum flow 2000 (s → a → t and s → b → t).
- A bad choice uses the middle edge every time: s → a → b → t, then via the reverse edge s → b → a → t … Each path adds only **1 unit** → **2000 augmentations**.
- Edmonds-Karp (BFS) picks shortest paths: **2 augmentations** suffice.
- With capacity 10^9 the DFS version could take billions of steps; BFS removes this dependence.

The Edmonds-Karp Algorithm

- Finds the maximum flow between two vertices of a weighted directed graph; **a variant of the Ford-Fulkerson method**.
- Developed by Jack Edmonds and Richard Karp (1972; Dinic independently in 1970).
- Finds augmenting paths with **BFS** (breadth-first search): each time it picks the augmenting path with the **fewest edges**.
- Edmonds-Karp uses BFS to find an augmenting path; Dinic uses BFS to check whether more flow is possible and to build a **level graph**.



BFS: s → c → t (2 edges)

Steps: (1) Find a shortest augmenting path from source to sink with BFS. (2) Send the bottleneck along it and update the reverse edges. (3) Repeat until BFS cannot reach the sink.

Edmonds-Karp: Complexity $O(V \cdot E^2)$

- Each iteration is one BFS plus one path update: **O(E)**.
- Because BFS picks shortest paths, the residual distance from s to every vertex **never decreases**.
- Each augmentation makes at least one edge **critical** (saturated). For the same edge to become critical again its distance must grow by at least 2; since distances are ≤ V, each edge is critical at most ~V/2 times.
- Number of augmentations **$O(V \cdot E)$** → total **$O(V \cdot E) \cdot O(E) = O(V \cdot E^2)$**, with E edges and V vertices.

The time no longer depends on |f*|: however large the capacities, the algorithm finishes in polynomial time.

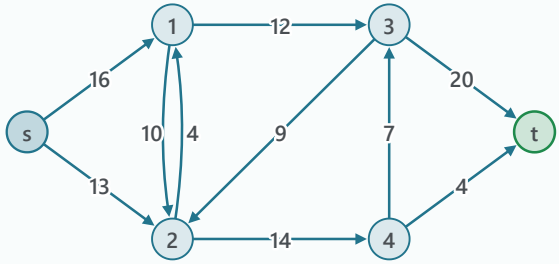
Java: Augmenting Paths with BFS

```java
boolean bfs(int src, int sink, int[] parent) {
  boolean[] visited = new boolean[V];
  Queue<Integer> queue = new LinkedList<>();
  queue.add(src);
  visited[src] = true;
  parent[src] = -1;
  while (!queue.isEmpty()) {
    int u = queue.poll();
    for (int v = 0; v < V; v++)
      if (!visited[v] && res[u][v] > 0) {
        queue.add(v);
        parent[v] = u;
        visited[v] = true;
      }
  }
  return visited[sink];
}
```

- The queue starts at the source; every unvisited neighbor with positive residual capacity is enqueued.
- `parent[v] = u` stores the BFS tree; if the sink is visited, the path found has the **fewest edges**.
- This function is the only difference from Ford-Fulkerson; the main loop is the same.

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
▶    boolean[] visited = new boolean[V];
▶    Queue<Integer> queue = new LinkedList<>();
▶    queue.add(src);
▶    visited[src] = true;
▶    parent[src] = -1;
7    while (!queue.isEmpty()) {
8      int u = queue.poll();
9      for (int v = 0; v < V; v++)
10       if (!visited[v] && res[u][v] > 0) {
11         queue.add(v);
12         parent[v] = u;
13         visited[v] = true;
14       }
15   }
16   return visited[sink];
17 }
```



Initial residual network (flows 0). s is marked and enqueued, parent[s] = −1; · = unassigned.

Step 1 / 8

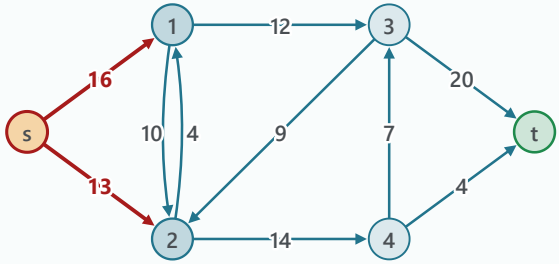
VARIABLES

`u = –` `queue = [s]`

`parent = [-1, ·, ·, ·, ·, ·]`

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
2    boolean[] visited = new boolean[V];
3    Queue<Integer> queue = new LinkedList<>();
4    queue.add(src);
5    visited[src] = true;
6    parent[src] = -1;
7    while (!queue.isEmpty()) {
▶      int u = queue.poll();
▶      for (int v = 0; v < V; v++)
▶        if (!visited[v] && res[u][v] > 0) {
▶          queue.add(v);
▶          parent[v] = u;
▶          visited[v] = true;
14       }
15   }
16   return visited[sink];
17 }
```



s dequeued: neighbors 1 (residual 16) and 2 (residual 13) enqueued, parent s.

Step 2 / 8

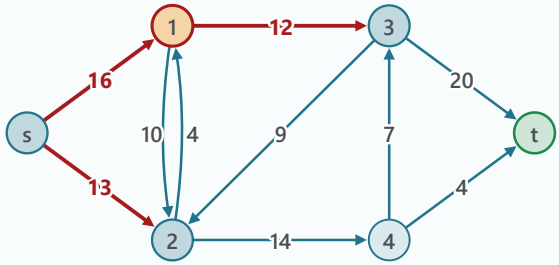
VARIABLES

`u = s` `queue = [1, 2]`

`parent = [-1, s, s, ·, ·, ·]`

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
2    boolean[] visited = new boolean[V];
3    Queue<Integer> queue = new LinkedList<>();
4    queue.add(src);
5    visited[src] = true;
6    parent[src] = -1;
7    while (!queue.isEmpty()) {
       int u = queue.poll();
       for (int v = 0; v < V; v++)
         if (!visited[v] && res[u][v] > 0) {
           queue.add(v);
           parent[v] = u;
           visited[v] = true;
14       }
15   }
16   return visited[sink];
17 }
```



queue 2 3 — red: BFS tree (parent)

1 is dequeued. Its unvisited neighbor 3 (residual 12) is enqueued with parent 1. 2 is already visited and is skipped.

Step 3 / 8

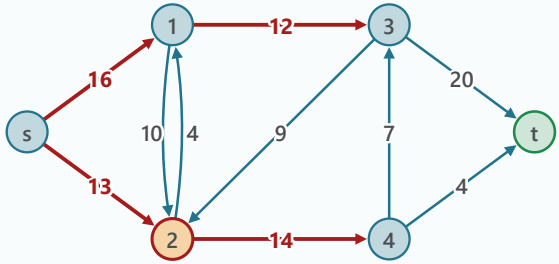
VARIABLES

`u = 1` `queue = [2, 3]`

`parent = [-1, s, s, 1, ·, ·]`

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
2    boolean[] visited = new boolean[V];
3    Queue<Integer> queue = new LinkedList<>();
4    queue.add(src);
5    visited[src] = true;
6    parent[src] = -1;
7    while (!queue.isEmpty()) {
▶      int u = queue.poll();
▶      for (int v = 0; v < V; v++)
▶        if (!visited[v] && res[u][v] > 0) {
▶          queue.add(v);
▶          parent[v] = u;
▶          visited[v] = true;
14       }
15   }
16   return visited[sink];
17 }
```



queue | 3 | 4 | | | red: BFS tree (parent)

2 is dequeued. Its unvisited neighbor 4 (residual 14) is enqueued with parent 2. 1 is already visited and is skipped.

Step 4 / 8

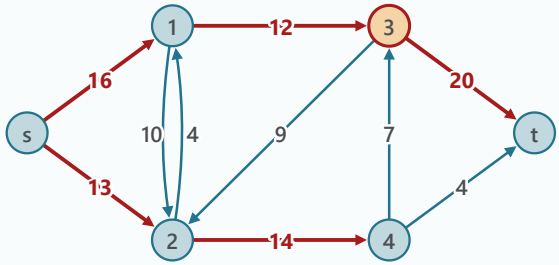
VARIABLES

`u = 2` `queue = [3, 4]`

`parent = [-1, s, s, 1, 2, ·]`

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
2    boolean[] visited = new boolean[V];
3    Queue<Integer> queue = new LinkedList<>();
4    queue.add(src);
5    visited[src] = true;
6    parent[src] = -1;
7    while (!queue.isEmpty()) {
       int u = queue.poll();
       for (int v = 0; v < V; v++)
         if (!visited[v] && res[u][v] > 0) {
           queue.add(v);
           parent[v] = u;
           visited[v] = true;
14       }
15   }
16   return visited[sink];
17 }
```



queue | 4 | t | | | red: BFS tree (parent)

3 dequeued: t (residual 20) enqueued, parent 3; 2 skipped. t is reached, but the loop runs until the queue is empty.

Step 5 / 8

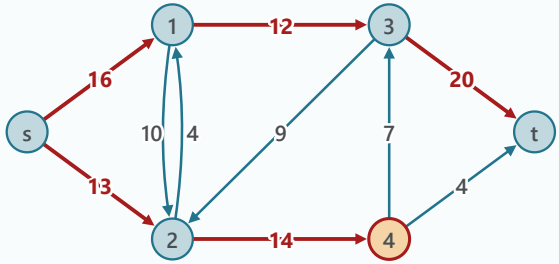
VARIABLES

`u = 3` `queue = [4, t]`

`parent = [-1, s, s, 1, 2, 3]`

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
2    boolean[] visited = new boolean[V];
3    Queue<Integer> queue = new LinkedList<>();
4    queue.add(src);
5    visited[src] = true;
6    parent[src] = -1;
7    while (!queue.isEmpty()) {
▶      int u = queue.poll();
▶      for (int v = 0; v < V; v++)
▶        if (!visited[v] && res[u][v] > 0) {
11         queue.add(v);
12         parent[v] = u;
13         visited[v] = true;
14       }
15   }
16   return visited[sink];
17 }
```



4 is dequeued. 3 and t are already visited and are skipped.

Step 6 / 8

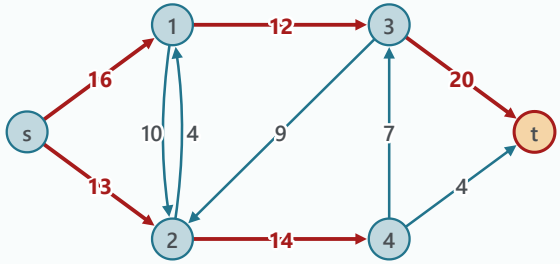
VARIABLES

`u = 4` `queue = [t]`

`parent = [-1, s, s, 1, 2, 3]`

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
2    boolean[] visited = new boolean[V];
3    Queue<Integer> queue = new LinkedList<>();
4    queue.add(src);
5    visited[src] = true;
6    parent[src] = -1;
7    while (!queue.isEmpty()) {
▶      int u = queue.poll();
▶      for (int v = 0; v < V; v++)
▶        if (!visited[v] && res[u][v] > 0) {
11         queue.add(v);
12         parent[v] = u;
13         visited[v] = true;
14       }
15   }
16   return visited[sink];
17 }
```



t is dequeued. It has no outgoing edge with positive residual capacity.

Step 7 / 8

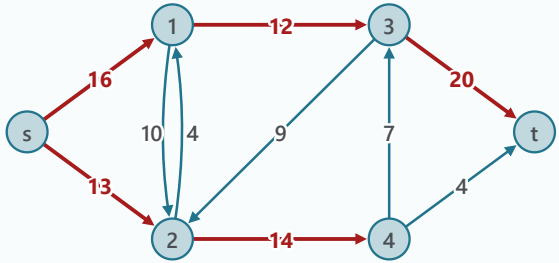
VARIABLES

`u = t` `queue = []`

`parent = [-1, s, s, 1, 2, 3]`

Fewest-Edge Path: `bfs(s, t, parent)`

```
1  boolean bfs(int src, int sink, int[] parent) {
2    boolean[] visited = new boolean[V];
3    Queue<Integer> queue = new LinkedList<>();
4    queue.add(src);
5    visited[src] = true;
6    parent[src] = -1;
7    while (!queue.isEmpty()) {
8      int u = queue.poll();
9      for (int v = 0; v < V; v++)
10       if (!visited[v] && res[u][v] > 0) {
11         queue.add(v);
12         parent[v] = u;
13         visited[v] = true;
14       }
15   }
▶    return visited[sink];
17 }
```



queue

red: BFS tree (parent)

Return `true`. Via parent[]: s → 1 → 3 → t, the **fewest edges** (3); bottleneck 12.

Step 8 / 8

VARIABLES

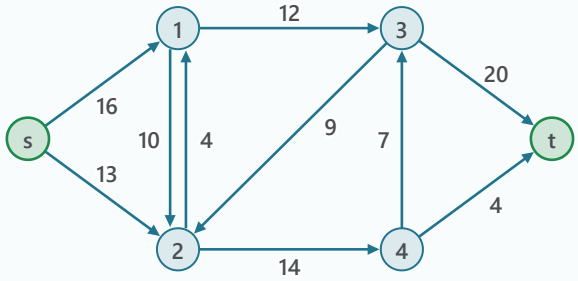
`u = –` `queue = []`

`parent = [-1, s, s, 1, 2, 3]`

`return = true`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
▶    int maxFlow = 0;
▶    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



Start: residual network = capacities, `maxFlow = 0`.

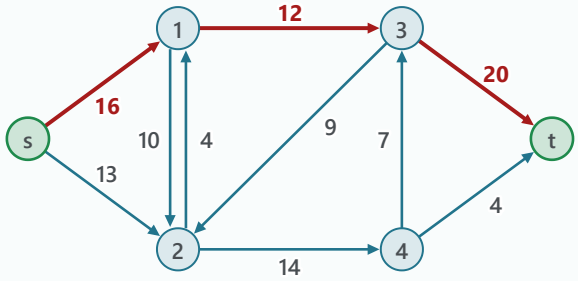
Step 1 / 14

VARIABLES

`maxFlow = 0` `path = –` `flow = –`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
▶    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



`bfs` found a shortest path: s → 1 → 3 → t (3 edges).

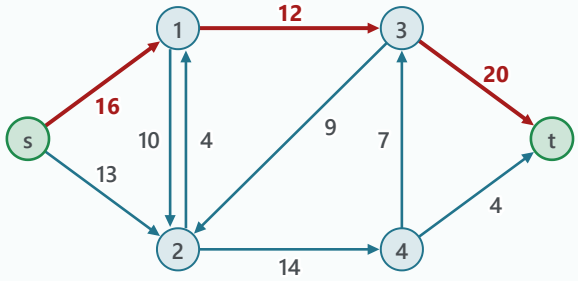
Step 2 / 14

VARIABLES

`maxFlow = 0` `path = s13t` `flow = –`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
▶      for (int v = t; v != s; v = parent[v])
▶        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



Walking back from the target: flow = min(16, 12, 20) = **12**.

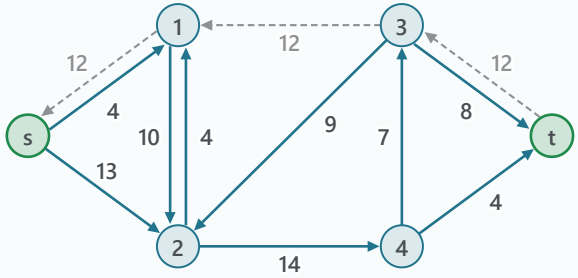
Step 3 / 14

VARIABLES

`maxFlow = 0` `path = s13t` `flow = 12`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
▶      for (int v = t; v != s; v = parent[v]) {
▶        int u = parent[v];
▶        res[u][v] -= flow;  // forward
▶        res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



Subtract 12 from the forward edges on the path and add 12 to the reverse edges (dashed arrows).

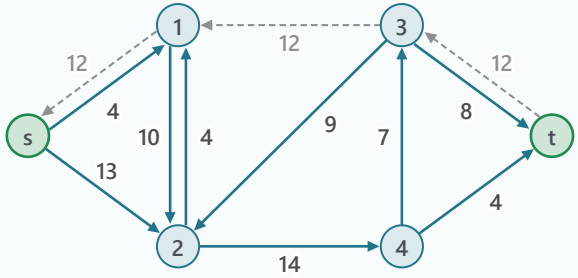
Step 4 / 14

VARIABLES

`maxFlow = 0` `path = s13t` `flow = 12`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
▶      maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



`maxFlow` = 0 + 12 = **12**.

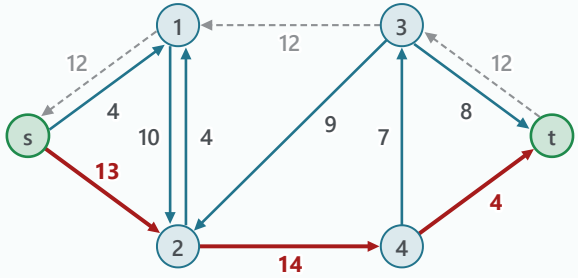
Step 5 / 14

VARIABLES

`maxFlow` = **12** `path` = **s13t** `flow` = **12**

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
▶    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



`bfs` found a shortest path: s → 2 → 4 → t (3 edges).

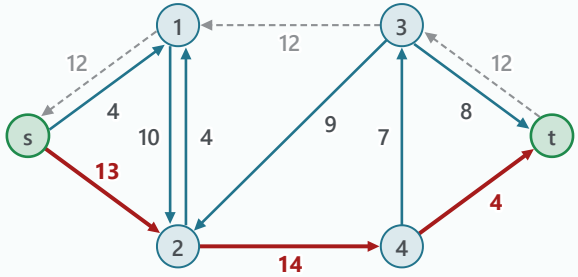
Step 6 / 14

VARIABLES

`maxFlow = 12` `path = s24t` `flow = –`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
▶      for (int v = t; v != s; v = parent[v])
▶        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



Walking back from the target: flow = min(13, 14, 4) = **4**.

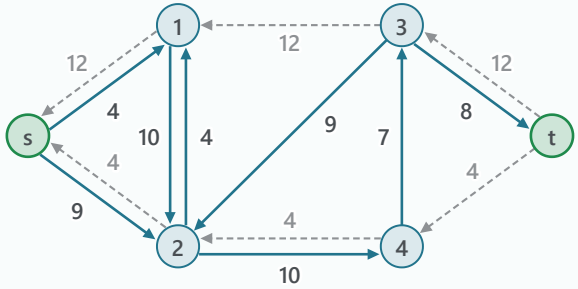
Step 7 / 14

VARIABLES

`maxFlow = 12` `path = s24t` `flow = 4`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
▶      for (int v = t; v != s; v = parent[v]) {
▶        int u = parent[v];
▶        res[u][v] -= flow;  // forward
▶        res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



Subtract 4 from the forward edges on the path and add 4 to the reverse edges (dashed arrows).

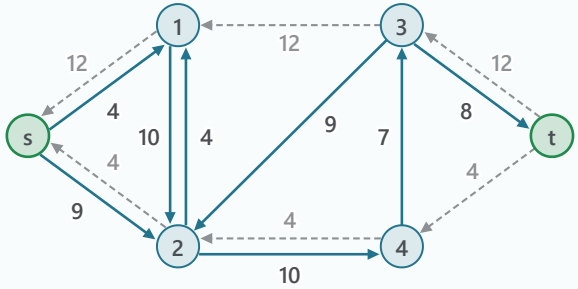
Step 8 / 14

VARIABLES

`maxFlow = 12` `path = s24t` `flow = 4`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
▶      maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



`maxFlow` = 12 + 4 = **16**.

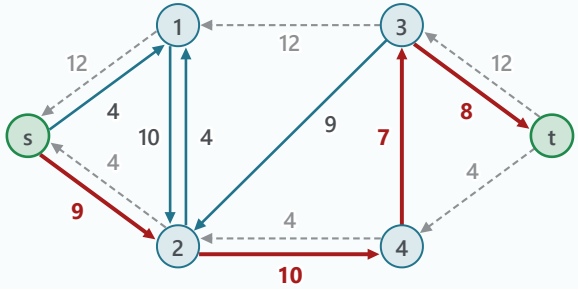
Step 9 / 14

VARIABLES

`maxFlow = 16` `path = s24t` `flow = 4`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
▶    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



`bfs` found a shortest path: s → 2 → 4 → 3 → t (4 edges).

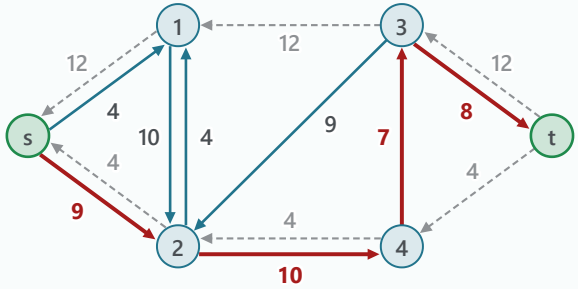
Step 10 / 14

VARIABLES

`maxFlow = 16` `path = s243t` `flow = –`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
▶      for (int v = t; v != s; v = parent[v])
▶        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



Walking back from the target: flow = min(9, 10, 7, 8) = **7**.

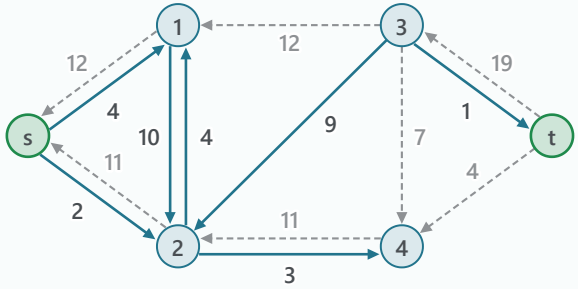
Step 11 / 14

VARIABLES

`maxFlow = 16` `path = s243t` `flow = 7`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
▶      for (int v = t; v != s; v = parent[v]) {
▶        int u = parent[v];
▶        res[u][v] -= flow;  // forward
▶        res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



Subtract 7 from the forward edges on the path and add 7 to the reverse edges (dashed arrows).

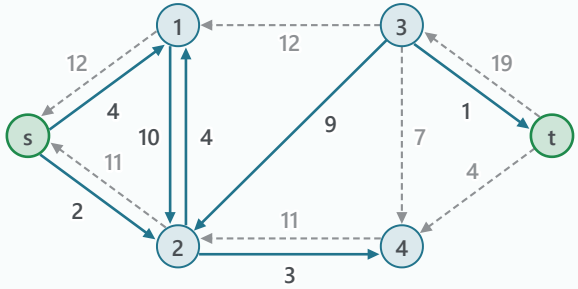
Step 12 / 14

VARIABLES

`maxFlow = 16` `path = s243t` `flow = 7`

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
4    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
▶      maxFlow += flow;
14   }
15   return maxFlow;
16 }
```



`maxFlow` = 16 + 7 = **23**.

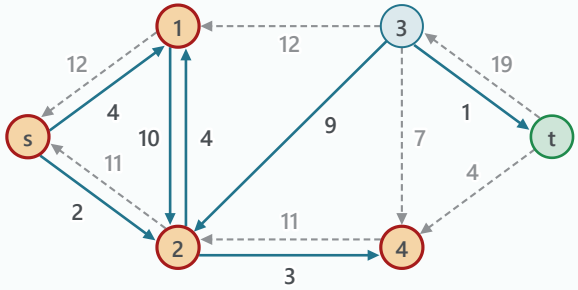
Step 13 / 14

VARIABLES

`maxFlow` = **23** `path` = **s243t** `flow` = **7**

edmondsKarp(s, t): Step Through the Code

```
1  int edmondsKarp(int s, int t) {
2    int maxFlow = 0;
3    int[] parent = new int[V];
▶    while (bfs(s, t, parent)) {
5      int flow = Integer.MAX_VALUE;
6      for (int v = t; v != s; v = parent[v])
7        flow = Math.min(flow, res[parent[v]][v]);
8      for (int v = t; v != s; v = parent[v]) {
9        int u = parent[v];
10       res[u][v] -= flow;  // forward
11       res[v][u] += flow;  // reverse
12     }
13     maxFlow += flow;
14   }
▶    return maxFlow;
16 }
```



BFS cannot reach t (from s only s, 1, 2, 4 are reachable). `return 23`.

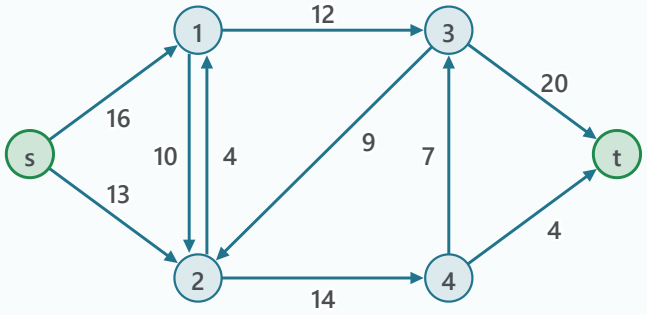
Step 14 / 14

VARIABLES

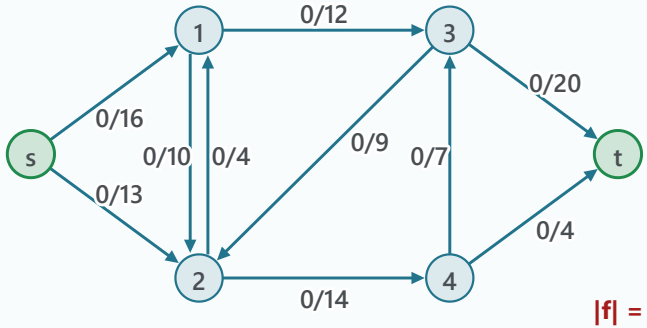
`maxFlow = 23` `path = none` `flow = 7`

Edmonds-Karp Example

Residual network



Flow f

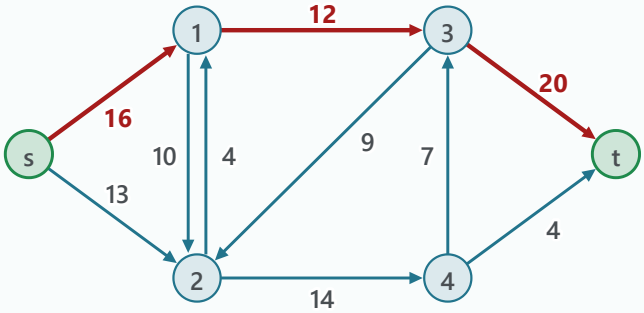


|f| = 0

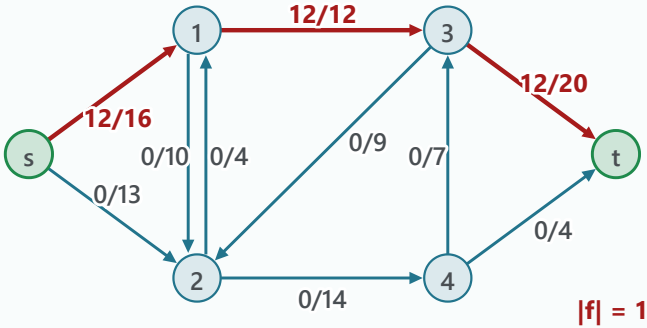
Take this network. Upper bound: |f*| ≤ **24** (capacity out of s is 16 + 13 = 29, into t 20 + 4 = 24; the smaller one).

Edmonds-Karp Example

Residual network: p_1



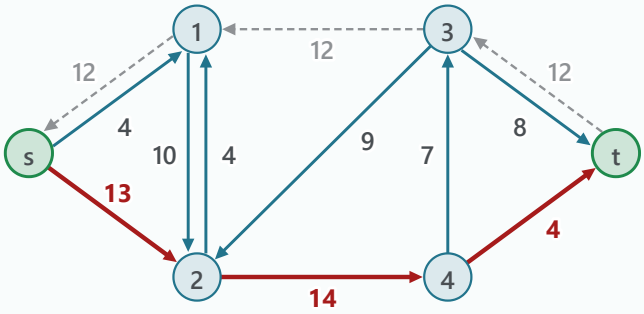
Flow f



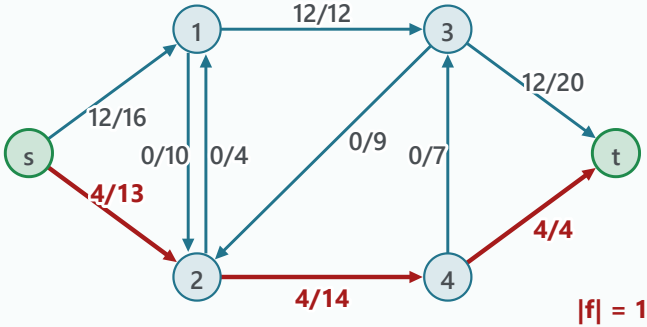
Step 1: the augmenting path $p_1 = \langle s, 1, 3, t \rangle$ is chosen, with $c_f(p_1) = \mathbf{12}$ because of c(1, 3). |f| = 12.

Edmonds-Karp Example

Residual network: p_2



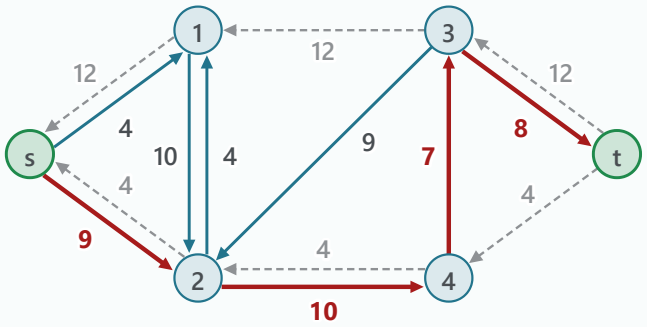
Flow f



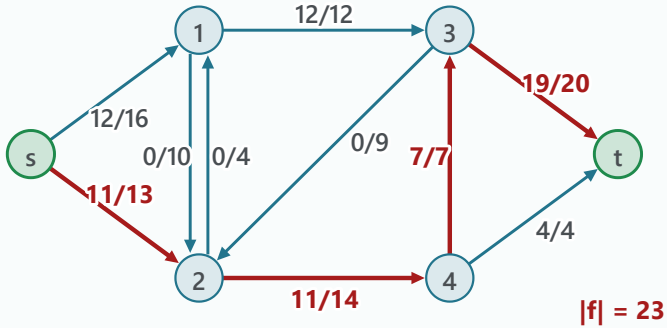
Step 2: the augmenting path $p_2 = \langle s, 2, 4, t \rangle$ is chosen, with $c_f(p_2) = \mathbf{4}$ because of c(4, t). |f| = 16.

Edmonds-Karp Example

Residual network: p_3



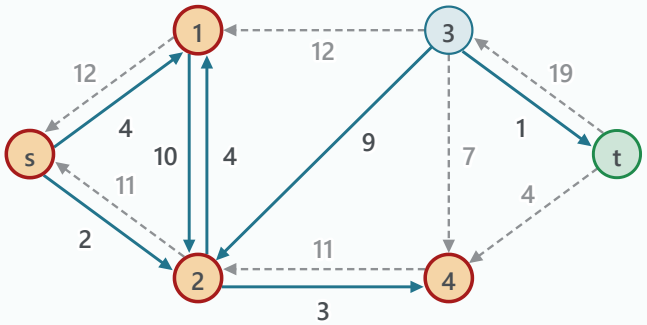
Flow f



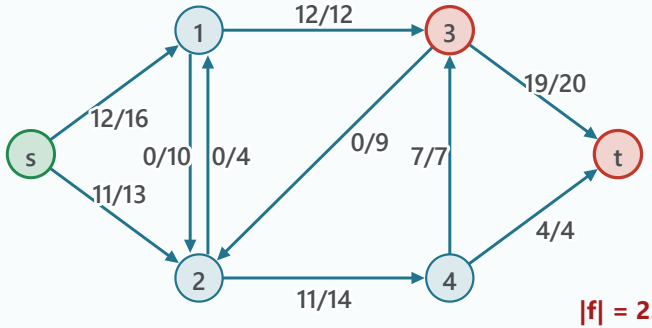
Step 3: the augmenting path $p_3 = \langle s, 2, 4, 3, t \rangle$ is chosen, with $c_f(p_3) = \mathbf{7}$ because of c(4, 3). |f| = 23.

Edmonds-Karp Example

Final residual network



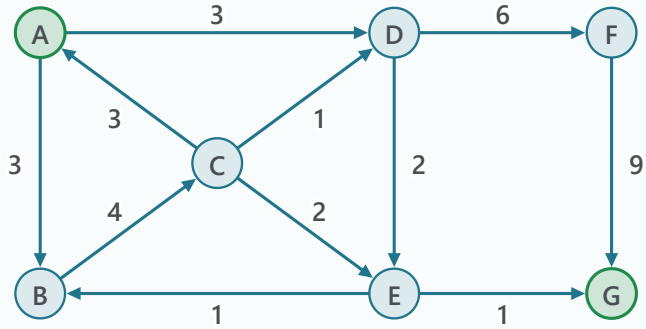
Final flow and min cut



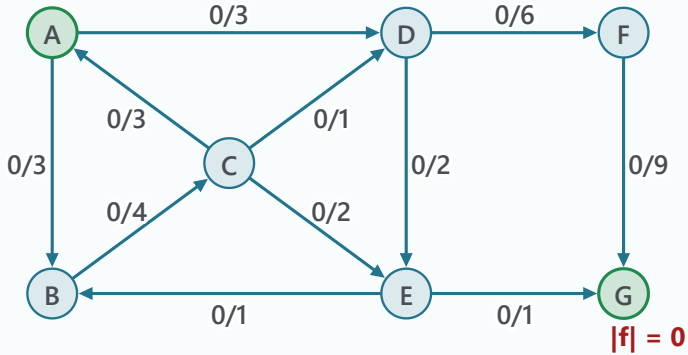
Final state: in the residual network s reaches only 1, 2, 4. The cut ({s, 1, 2, 4}, {3, t}) (red: T side): 12 + 7 + 4 = **23** = |f*| (flow into t: 19 + 4).

Edmonds-Karp Example 2

Residual network



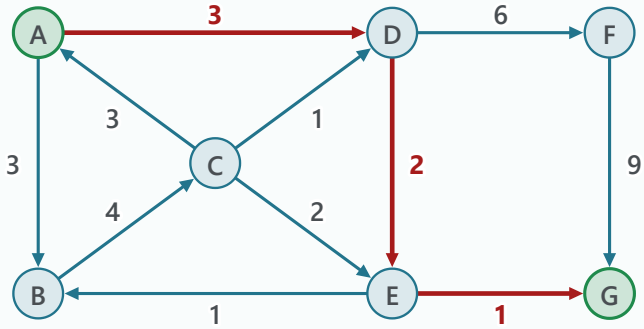
Flow f



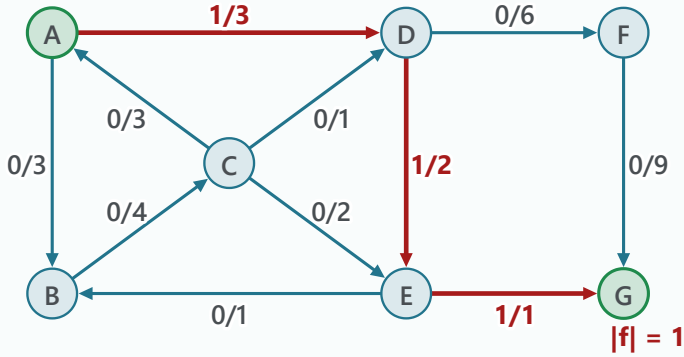
Source A, sink G. All flows start at 0.

Edmonds-Karp Example 2

Residual network: path (+1)



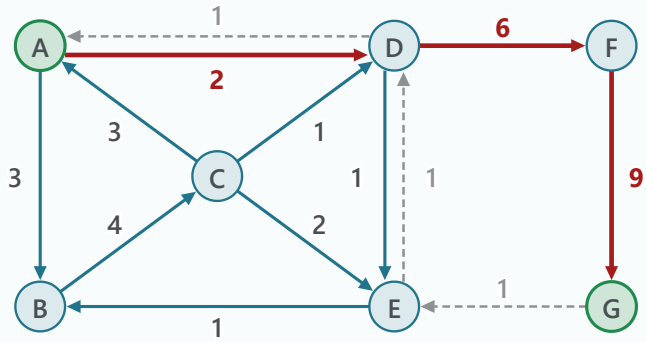
Flow f



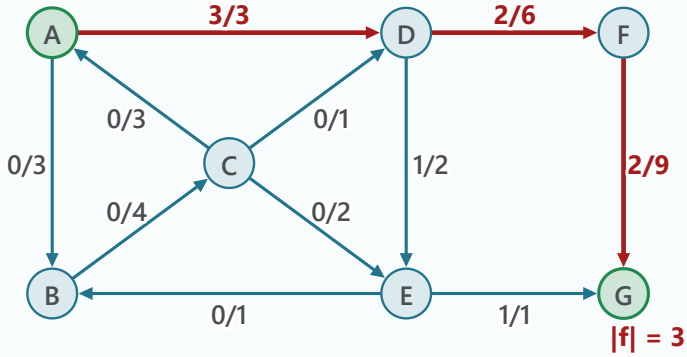
p = {A, D, E, G}: bottleneck E → G (1). Flow = 1.

Edmonds-Karp Example 2

Residual network: path (+2)



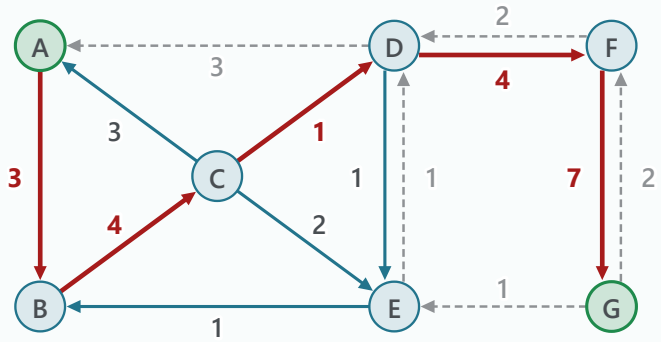
Flow f



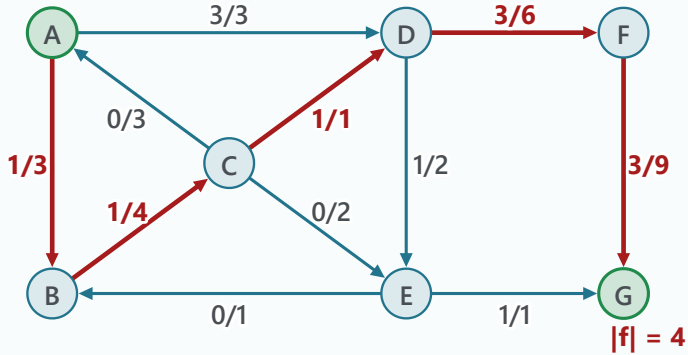
p = {A, D, F, G}: 2 was left on A → D. Flow = 1 + 2 = 3.

Edmonds-Karp Example 2

Residual network: path (+1)



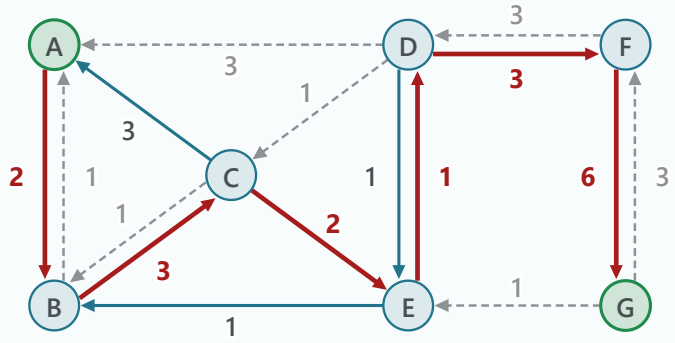
Flow f



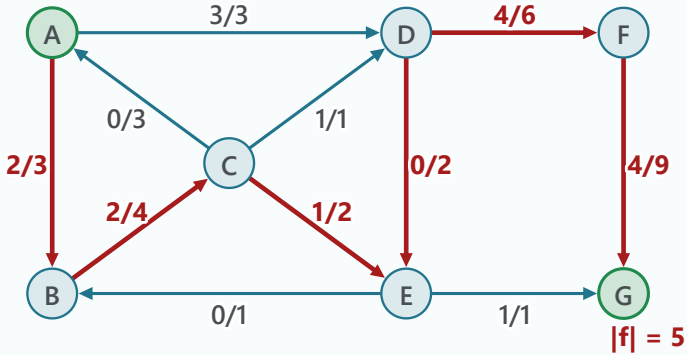
p = {A, B, C, D, F, G}: A → D is saturated, so BFS finds a 5-edge path; bottleneck C → D (1). Flow = 4.

Edmonds-Karp Example 2

Residual network: path (+1)



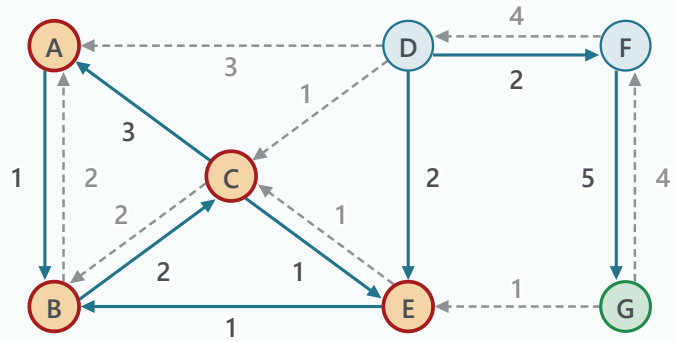
Flow f



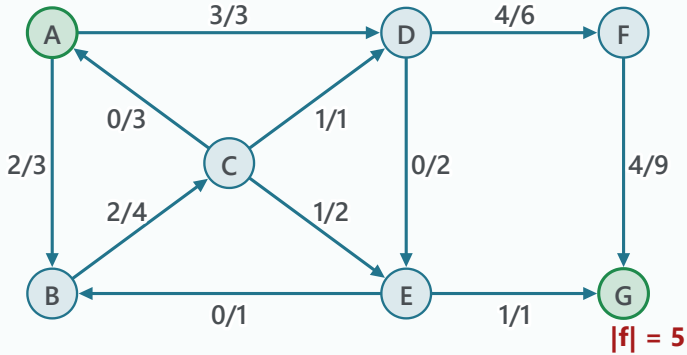
p = {A, B, C, E, D, F, G}: the **reverse edge E → D** lowers the flow on D → E from 1 to 0. Flow = **5**.

Edmonds-Karp Example 2

Final residual network



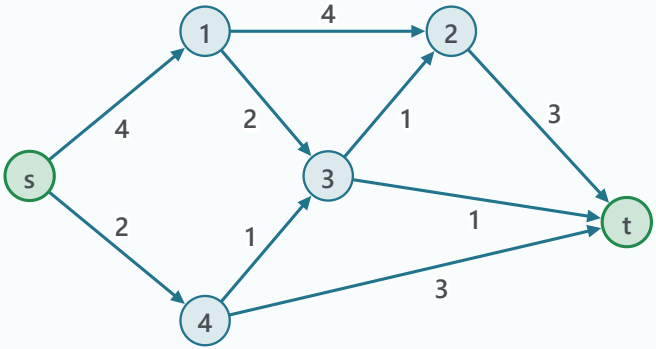
Maximum flow



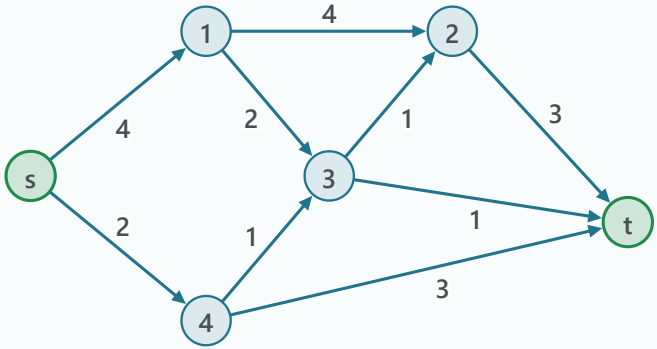
BFS cannot reach G; from A, the vertices A, B, C, E are reachable. Cut capacity A→D 3 + C→D 1 + E→G 1 = **5** = |f*|.

Edmonds-Karp Example 3: Working on the Residual Network

Network (capacities)



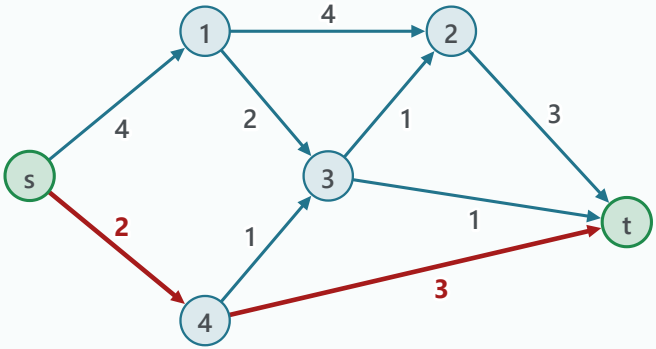
Residual network



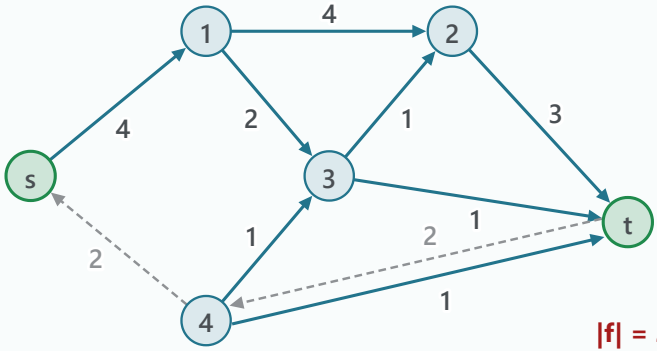
From s to t; initially the residual network = the capacities. BFS first finds the 2-edge path s→4→t, then the 3-edge paths s→1→2→t and s→1→3→t.

Edmonds-Karp Example 3: Working on the Residual Network

Before: path



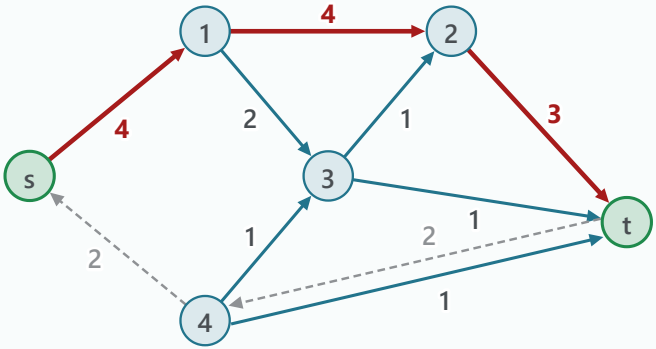
Then: +2



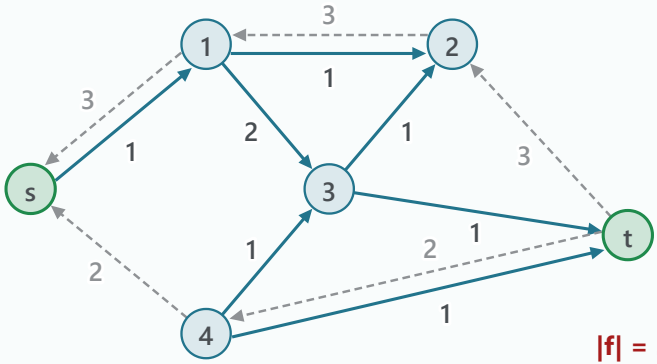
Path s → 4 → t: bottleneck s → 4 (2). **Push 2 units**; s → 4 has 0 left, and the reverse edge 4 → s has 2.

Edmonds-Karp Example 3: Working on the Residual Network

Before: path



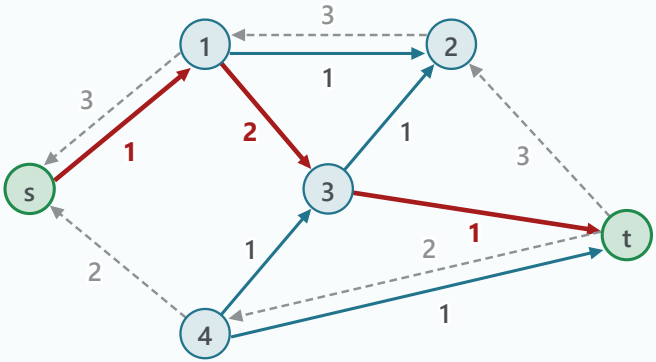
Then: +3



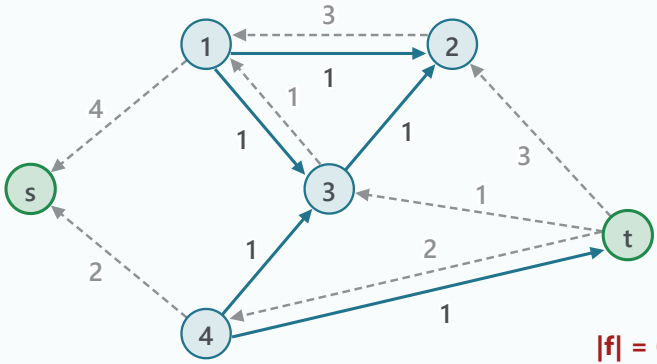
Path s → 1 → 2 → t: bottleneck 2 → t (3). **Push 3 units**.

Edmonds-Karp Example 3: Working on the Residual Network

Before: path



Then: +1

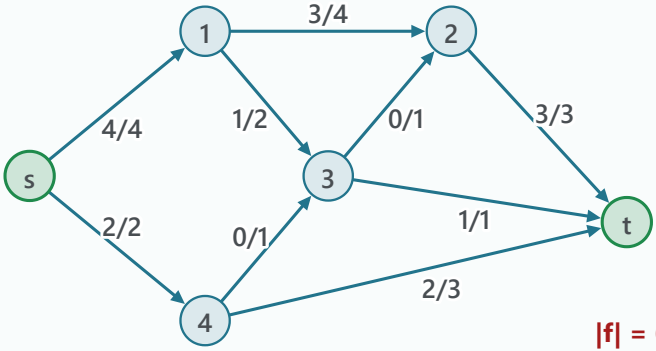


|f| = 6

Path s → 1 → 3 → t: 1 was left on s → 1. **Push 1 unit**.

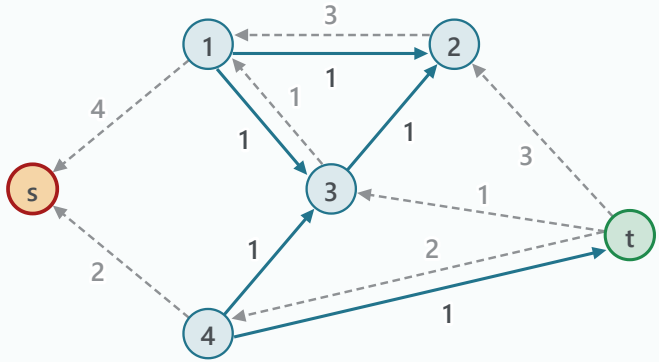
Edmonds-Karp Example 3: Working on the Residual Network

Final flow (f/c)



|f| = 6

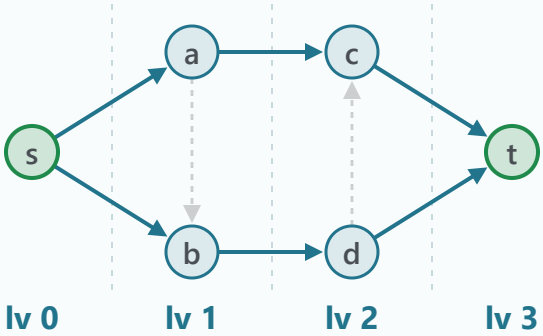
Final residual network



The edges out of s are saturated → no path. |f*| = 2 + 3 + 1 = **6**: 3 units along s→1→2→t, 1 unit along s→1→3→t, 2 units along s→4→t.

Dinic's Algorithm

- Finds a maximum flow in a graph (Y. Dinitz, 1970); works on a capacitated (weighted) graph.
- **Level graph**: BFS on the residual network gives each vertex a level (its distance from the source); only edges with level(v) = level(u) + 1 are kept.
- **Blocking flow**: a flow in the level graph that leaves at least one saturated edge on every s–t path; it is found with DFS.
- The flow is updated with reverse edges; phases repeat until the sink is unreachable in the level graph.



Edmonds-Karp uses **one** path after each BFS; Dinic saturates **all** shortest paths of the same level graph in one phase.

Dinic: The Steps

1. Set the flow on every edge to 0.

The residual network is identical to the original.

Dinic: The Steps

1. Set the flow on every edge to 0.

The residual network is identical to the original.

2. Run BFS from the source to find each vertex's **level** (distance from the source). If the sink has no level, stop.

BFS uses edges with positive residual capacity; if the sink is unreachable, the flow is maximum.

Dinic: The Steps

1. Set the flow on every edge to 0.

The residual network is identical to the original.

2. Run BFS from the source to find each vertex's **level** (distance from the source). If the sink has no level, stop.

BFS uses edges with positive residual capacity; if the sink is unreachable, the flow is maximum.

3. Use DFS to find augmenting paths that go **one level up** at each step.

Only edges with level(v) = level(u) + 1 are followed; dead-end edges are never tried again.

Dinic: The Steps

1. Set the flow on every edge to 0.

The residual network is identical to the original.

2. Run BFS from the source to find each vertex's **level** (distance from the source). If the sink has no level, stop.

BFS uses edges with positive residual capacity; if the sink is unreachable, the flow is maximum.

3. Use DFS to find augmenting paths that go **one level up** at each step.

Only edges with level(v) = level(u) + 1 are followed; dead-end edges are never tried again.

4. Add each path's minimum capacity to the total flow; update the capacities of the edges and reverse edges.

When no path is left, this phase's blocking flow is found.

Dinic: The Steps

1. Set the flow on every edge to 0.

The residual network is identical to the original.

2. Run BFS from the source to find each vertex's **level** (distance from the source). If the sink has no level, stop.

BFS uses edges with positive residual capacity; if the sink is unreachable, the flow is maximum.

3. Use DFS to find augmenting paths that go **one level up** at each step.

Only edges with level(v) = level(u) + 1 are followed; dead-end edges are never tried again.

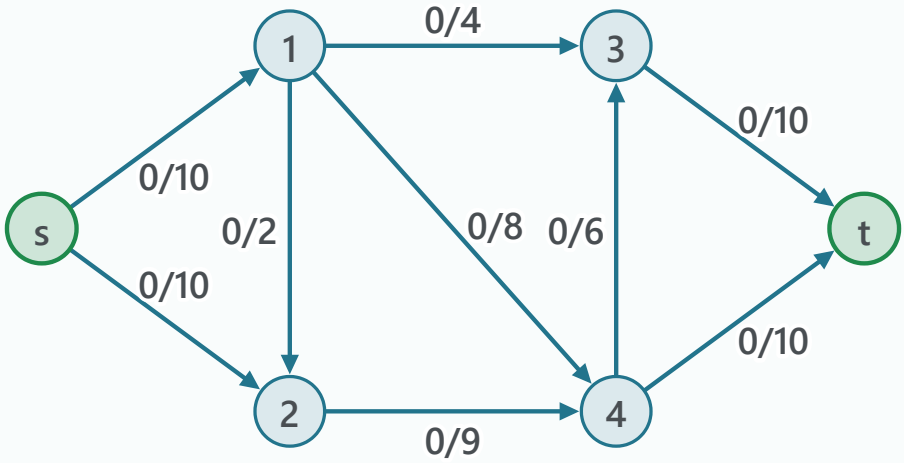
4. Add each path's minimum capacity to the total flow; update the capacities of the edges and reverse edges.

When no path is left, this phase's blocking flow is found.

5. Go back to step 2 and repeat.

The s–t distance grows by at least 1 in each phase, so there are at most V − 1 phases.

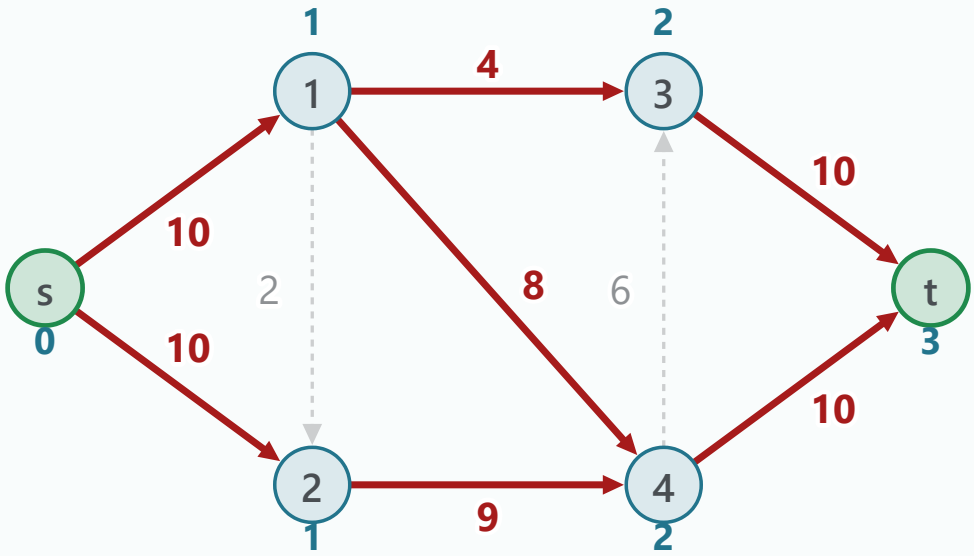
Dinic Example: Phases



|f| = 0

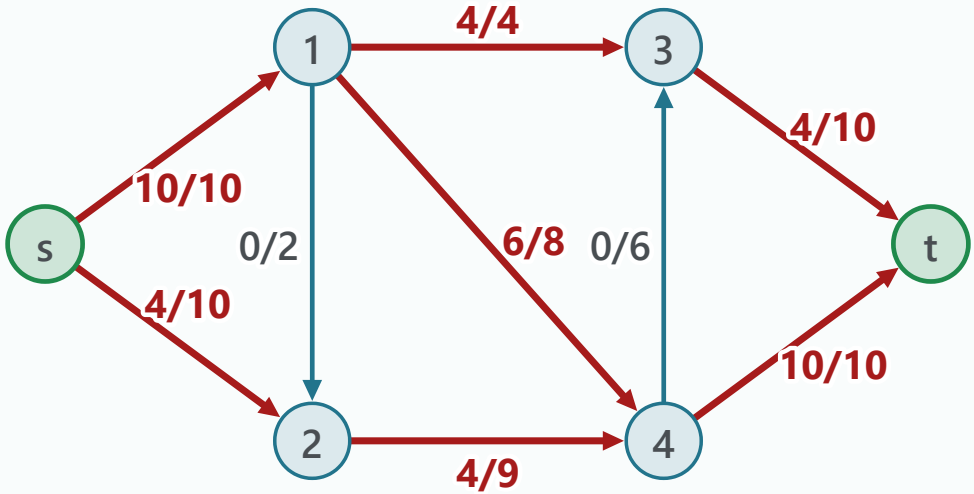
The network (flow/capacity). All flows start at 0; residual network = capacities.

Dinic Example: Phases



Phase 1 – BFS: levels s = 0; 1, 2 = 1; 3, 4 = 2; t = 3. The faded edges (1 → 2, 4 → 3) do not go up a level and are not used.

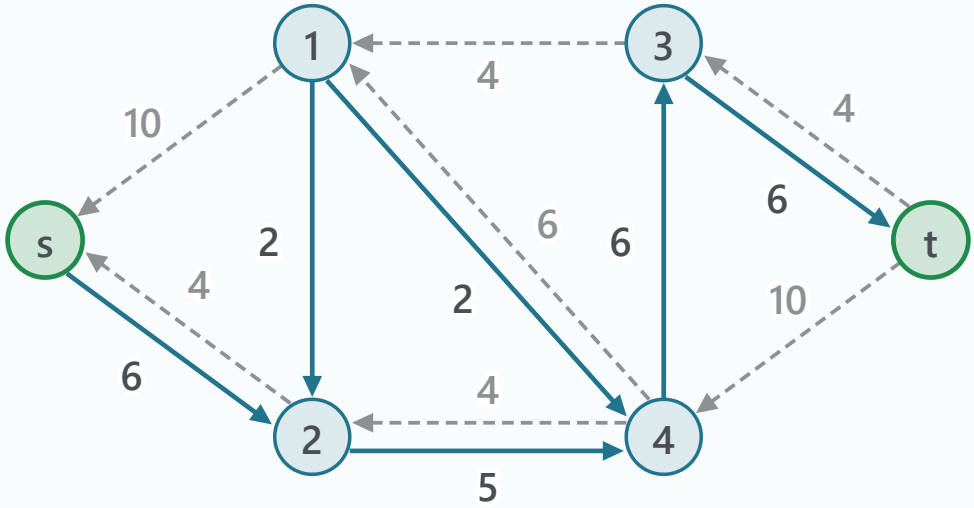
Dinic Example: Phases



|f| = 14

Phase 1 – blocking flow (DFS): {s, 1, 3, t} 4 units, {s, 1, 4, t} 6 units, {s, 2, 4, t} 4 units → |f| = **14**.

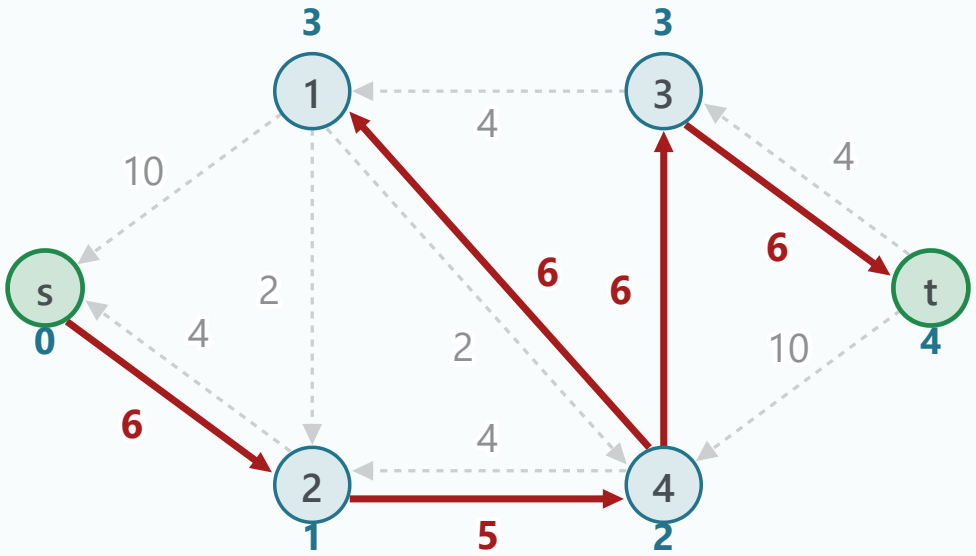
Dinic Example: Phases



|f| = 14

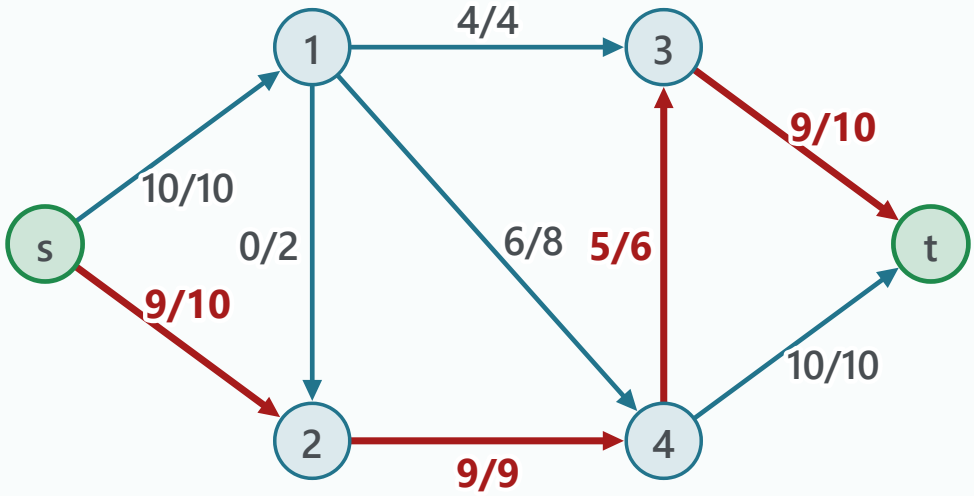
Residual network after phase 1 (G_f): saturated edges remain only in the reverse direction (dashed).

Dinic Example: Phases



Phase 2 – BFS: s = 0; 2 = 1; 4 = 2; 1, 3 = 3; t = 4. The s–t distance went from 3 to 4. The edge 4 → 1 is a dead end.

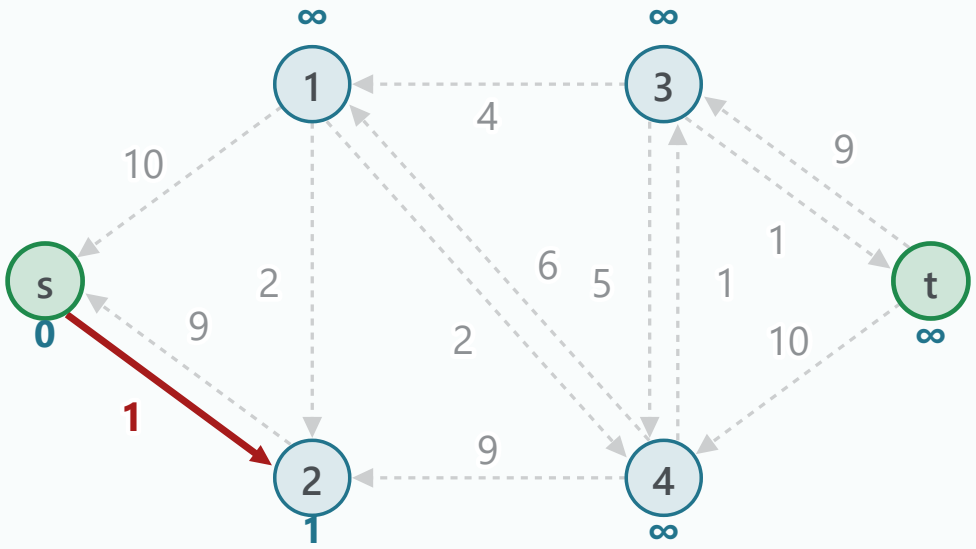
Dinic Example: Phases



|f| = 19

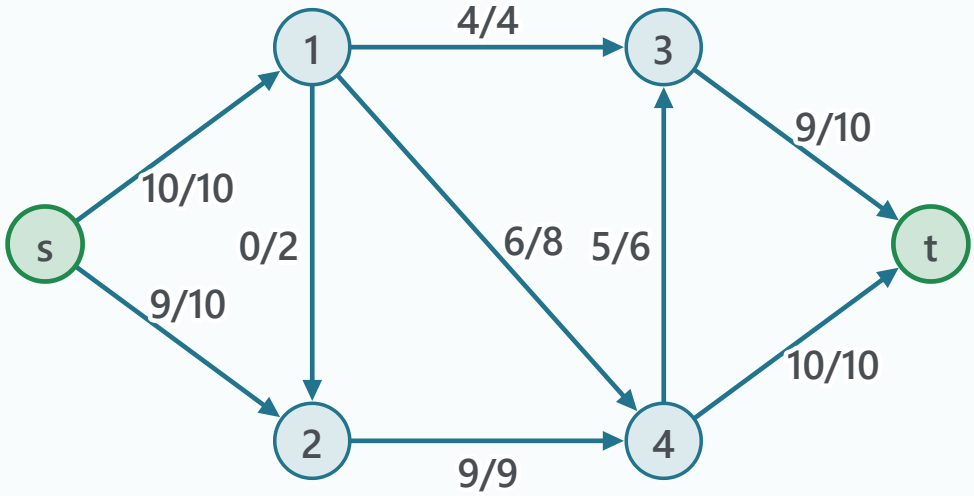
Phase 2 – blocking flow: {s, 2, 4, 3, t} 5 units (bottleneck 2 → 4) → |f| = 14 + 5 = **19**.

Dinic Example: Phases



Phase 3 – BFS: only s (0) and 2 (1) are reachable; there is no path to t → the algorithm ends. Min cut ({s, 2}, the rest) = 10 + 9 = 19.

Dinic Example: Phases



|f| = 19

Result: maximum flow **19**; just **2 phases** (2 + 1 BFS runs) were enough. On the same network Ford-Fulkerson used 5 augmenting paths.

Dinic: Pseudocode

```
DINIC(G, source, sink):
  flow = [ ]       // 0 for every edge
  maxFlow = 0
  while BFS(G, source, sink, flow) do:
    // level graph is built
    while a blocking path exists do:
      delta = DFS(G, source, sink,
                  flow, level, ∞)
      maxFlow += delta
  return maxFlow

BFS(G, source, sink, flow):
  level[v] = -1, for every v
  level[source] = 0
  queue.add(source)
  while queue is not empty do:
    u = queue.remove()
    for each (u, v) in G[u]:
      if level[v] < 0 and
           capacity[u,v] - flow[u,v] > 0:
        level[v] = level[u] + 1
        queue.add(v)
  return level[sink] ≥ 0
```

```
DFS(G, u, sink, flow, level, lim):
  if u == sink: return lim
  for each (u, v) in G[u]:
    left = capacity[u,v] - flow[u,v]
    if level[v] == level[u] + 1
         and left > 0:
      delta = DFS(G, v, sink, flow,
                  level, min(lim, left))
      if delta > 0:
        flow[u, v] += delta
        flow[v, u] -= delta   // reverse
        return delta
  return 0
```

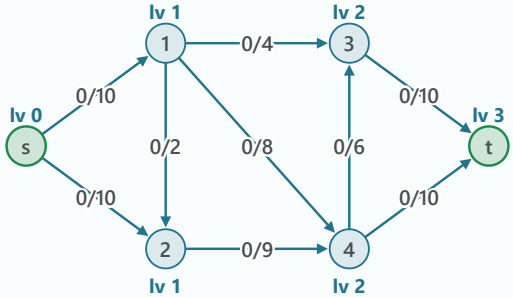
Java: DFS on Levels with a Current-Arc Array

```java
int dfs(int u, int t, int flow, int[] next) {
  if (u == t) return flow;
  for (; next[u] < adj[u].size(); next[u]++) {
    Edge e = adj[u].get(next[u]);
    if (level[e.to] == level[u] + 1 && e.flow < e.cap) {
      int d = dfs(e.to, t, Math.min(flow, e.cap - e.flow), next);
      if (d > 0) {
        e.flow += d;                                  // forward edge
        adj[e.to].get(e.rev).flow -= d;               // reverse edge
        return d;
      }
    }
  }
  return 0;   // no level path from u to t left
}
```

- Edges are kept in adjacency lists; each edge knows the index of its reverse edge (`rev`).
- `next[u]` : the next edge to try at u (the *current arc*). A dead-end edge is never retried; all DFS calls in one phase take O(V · E) in total.
- At the start of each phase `next` is reset and the levels are recomputed with BFS.

Dinic: The Blocking Flow in Phase 1

```
▶ int dfs(int u, int t, int flow, int[] next) {
2   if (u == t) return flow;
3   for (; next[u] < adj[u].size(); next[u]++) {
4     Edge e = adj[u].get(next[u]);
5     if (level[e.to] == level[u] + 1
6         && e.flow < e.cap) {
7       int d = dfs(e.to, t, Math.min(flow,
8           e.cap - e.flow), next);
9       if (d > 0) {
10        e.flow += d;
11        adj[e.to].get(e.rev).flow -= d;
12        return d;
13      }
14    }
15  }
16  return 0;   // no level path left
17 }
```



calls: —

Phase 1, with the BFS levels next to the vertices (lv). Only edges that go up one level are used; next[u] is the next edge to try at u.

Step 1 / 10

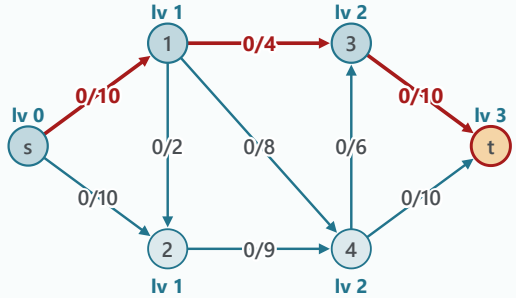
VARIABLES

`u = –` `flow = –`

`next = [0, 0, 0, 0, 0, 0]` `|f| = 0`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
▶    if (u == t) return flow;
3    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
▶      if (level[e.to] == level[u] + 1
▶          && e.flow < e.cap) {
▶        int d = dfs(e.to, t, Math.min(flow,
▶            e.cap - e.flow), next);
9        if (d > 0) {
10         e.flow += d;
11         adj[e.to].get(e.rev).flow -= d;
12         return d;
13       }
14     }
15   }
16   return 0;   // no level path left
17 }
```



calls: dfs(s, ∞) → dfs(1, 10) → dfs(3, 4) → dfs(t, 4)

Descend one level at a time: s → 1 (residual 10), 1 → 3 (residual 4), 3 → t (residual 10). Each call caps the flow at the residual (min); 4 reaches t.

Step 2 / 10

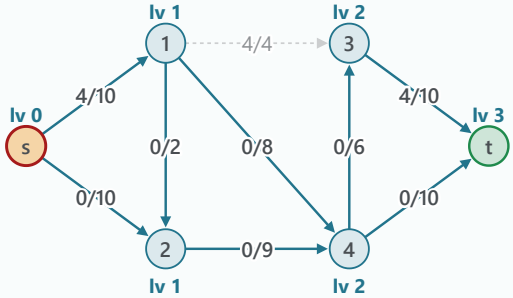
VARIABLES

`u = t` `flow = 4`

`next = [0, 1, 0, 2, 0, 0]` `|f| = 0`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
2    if (u == t) return flow;
3    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
5      if (level[e.to] == level[u] + 1
6          && e.flow < e.cap) {
7        int d = dfs(e.to, t, Math.min(flow,
8            e.cap - e.flow), next);
▶        if (d > 0) {
▶          e.flow += d;
▶          adj[e.to].get(e.rev).flow -= d;
▶          return d;
13       }
14     }
15   }
16   return 0;   // no level path left
17 }
```



calls: dfs(s, ∞)

As the flow returns along the path, 4 is added to each edge (and subtracted from the reverse edges). Back at s: |f| = 4.

Step 3 / 10

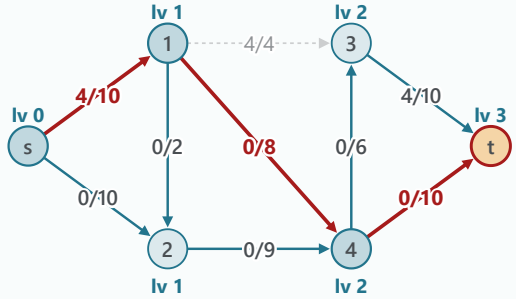
VARIABLES

`u = s` `flow = ∞`

`next = [0, 1, 0, 2, 0, 0]` `|f| = 4`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
   if (u == t) return flow;
3    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
       if (level[e.to] == level[u] + 1
           && e.flow < e.cap) {
         int d = dfs(e.to, t, Math.min(flow,
             e.cap - e.flow), next);
9        if (d > 0) {
10         e.flow += d;
11         adj[e.to].get(e.rev).flow -= d;
12         return d;
13       }
14     }
15   }
16   return 0;   // no level path left
17 }
```



calls: dfs(s, ∞) → dfs(1, 6) → dfs(4, 6) → dfs(t, 6)

s → 1 (residual 6), 1 → 4 (residual 8), 4 → t (residual 10). Each call caps the flow at the residual (min); 6 reaches t.

Step 4 / 10

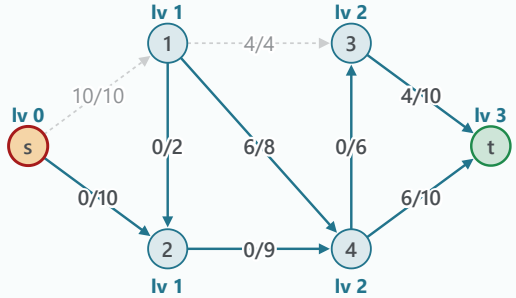
VARIABLES

`u = t` `flow = 6`

`next = [0, 3, 0, 2, 3, 0]` `|f| = 4`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
2    if (u == t) return flow;
3    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
5      if (level[e.to] == level[u] + 1
6          && e.flow < e.cap) {
7        int d = dfs(e.to, t, Math.min(flow,
8            e.cap - e.flow), next);
▶        if (d > 0) {
▶          e.flow += d;
▶          adj[e.to].get(e.rev).flow -= d;
▶          return d;
13       }
14     }
15   }
16   return 0;   // no level path left
17 }
```



calls: dfs(s, ∞)

As the flow returns along the path, 6 is added to each edge (and subtracted from the reverse edges). Back at s: |f| = 10. s → 1 is saturated.

Step 5 / 10

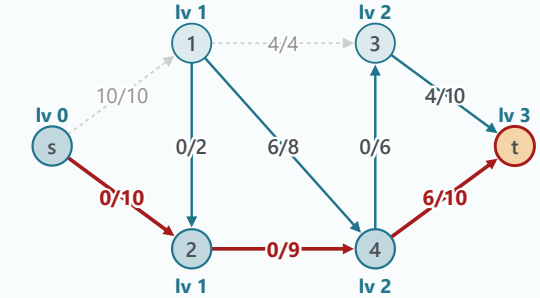
VARIABLES

`u = s` `flow = ∞`

`next = [0, 3, 0, 2, 3, 0]` `|f| = 10`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
   if (u == t) return flow;
3    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
       if (level[e.to] == level[u] + 1
           && e.flow < e.cap) {
         int d = dfs(e.to, t, Math.min(flow,
             e.cap - e.flow), next);
9        if (d > 0) {
10         e.flow += d;
11         adj[e.to].get(e.rev).flow -= d;
12         return d;
13       }
14     }
15   }
16   return 0;   // no level path left
17 }
```



calls: dfs(s, ∞) → dfs(2, 10) → dfs(4, 9) → dfs(t, 4)

s → 2 (residual 10), 2 → 4 (residual 9), 4 → t (residual 4). Each call caps the flow at the residual (min); 4 reaches t.

Step 6 / 10

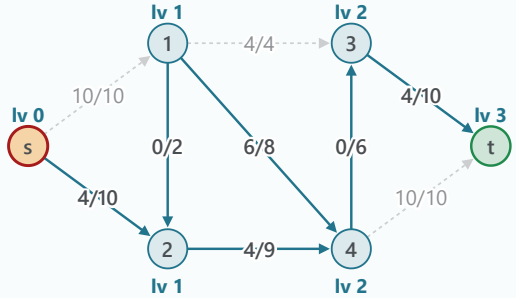
VARIABLES

`u = t` `flow = 4`

`next = [1, 3, 2, 2, 3, 0]` `|f| = 10`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
2    if (u == t) return flow;
3    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
5      if (level[e.to] == level[u] + 1
6          && e.flow < e.cap) {
7        int d = dfs(e.to, t, Math.min(flow,
8            e.cap - e.flow), next);
▶        if (d > 0) {
▶          e.flow += d;
▶          adj[e.to].get(e.rev).flow -= d;
▶          return d;
13       }
14     }
15   }
16   return 0;   // no level path left
17 }
```



calls: dfs(s, ∞)

As the flow returns along the path, 4 is added to each edge (and subtracted from the reverse edges). Back at s: |f| = 14.

Step 7 / 10

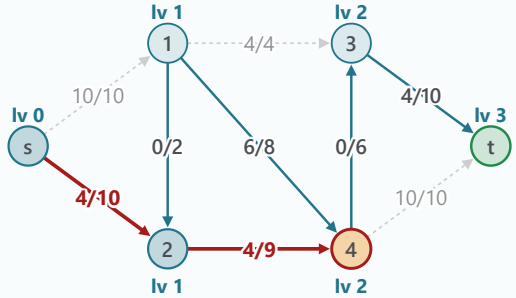
VARIABLES

`u = s` `flow = ∞`

`next = [1, 3, 2, 2, 3, 0]` `|f| = 14`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
2    if (u == t) return flow;
3    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
5      if (level[e.to] == level[u] + 1
6          && e.flow < e.cap) {
7        int d = dfs(e.to, t, Math.min(flow,
8            e.cap - e.flow), next);
9        if (d > 0) {
10         e.flow += d;
11         adj[e.to].get(e.rev).flow -= d;
12         return d;
13       }
14     }
15   }
▶    return 0;   // no level path left
17 }
```



Descended via s → 2 (residual 6), 2 → 4 (residual 5). No level path is left from 4 (4 → 3 stays on the same level, 4 → t is saturated): it returns 0.

Step 8 / 10

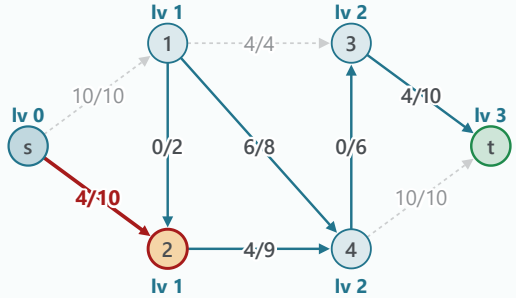
VARIABLES

`u = 4` `flow = 5`

`next = [1, 3, 2, 2, 4, 0]` `|f| = 14`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
2    if (u == t) return flow;
▶    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
5      if (level[e.to] == level[u] + 1
6          && e.flow < e.cap) {
7        int d = dfs(e.to, t, Math.min(flow,
8            e.cap - e.flow), next);
9        if (d > 0) {
10         e.flow += d;
11         adj[e.to].get(e.rev).flow -= d;
12         return d;
13       }
14     }
15   }
▶    return 0;   // no level path left
17 }
```



calls: dfs(s, ∞) → dfs(2, 6)

No flow came back over 2 → 4: `next[2]++`.
No level path is left from 2: it returns 0.

Step 9 / 10

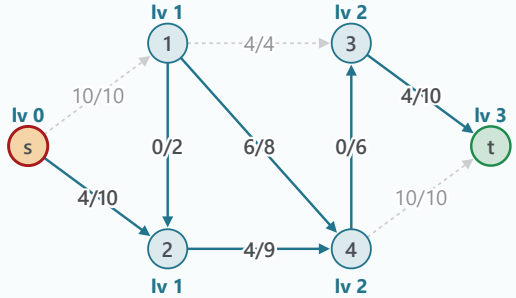
VARIABLES

`u = 2` `flow = 6`

`next = [1, 3, 3, 2, 4, 0]` `|f| = 14`

Dinic: The Blocking Flow in Phase 1

```
1  int dfs(int u, int t, int flow, int[] next) {
2    if (u == t) return flow;
▶    for (; next[u] < adj[u].size(); next[u]++) {
4      Edge e = adj[u].get(next[u]);
5      if (level[e.to] == level[u] + 1
6          && e.flow < e.cap) {
7        int d = dfs(e.to, t, Math.min(flow,
8            e.cap - e.flow), next);
9        if (d > 0) {
10         e.flow += d;
11         adj[e.to].get(e.rev).flow -= d;
12         return d;
13       }
14     }
15   }
▶    return 0;   // no level path left
17 }
```



calls: dfs(s, ∞)

No flow came back over s → 2: `next[s]++`. No level path is left from s: it returns 0. The blocking flow is complete; the phase ends with |f| = 14.

Step 10 / 10

VARIABLES

`u = s` `flow = ∞`

`next = [2, 3, 3, 2, 4, 0]` `|f| = 14`

Dinic: Complexity $O(V^2 \cdot E)$

- Each phase needs one BFS: **$O(V + E)$**.
- After each phase the s–t distance **strictly increases**, and since it is $\leq V - 1$ there are at most **$V - 1$ phases**.
- Blocking flow in one phase: each augmenting path has length $O(V)$ and at most E paths are found; with the current-arc array that is **$O(V \cdot E)$** per phase.
- Total: **$O(V) \cdot O(V \cdot E) = O(V^2 \cdot E)$**.

Unit capacities

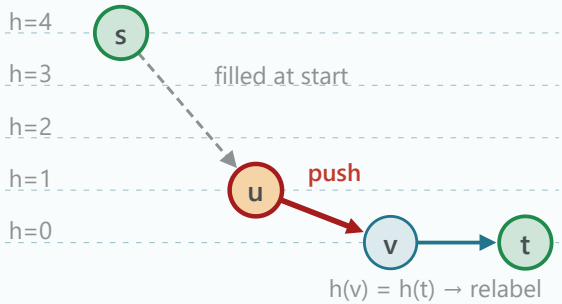
If every capacity is 1, $O(E \cdot \sqrt{E})$; on unit networks such as bipartite matching, $O(E \cdot \sqrt{V})$ (Hopcroft-Karp).

In practice

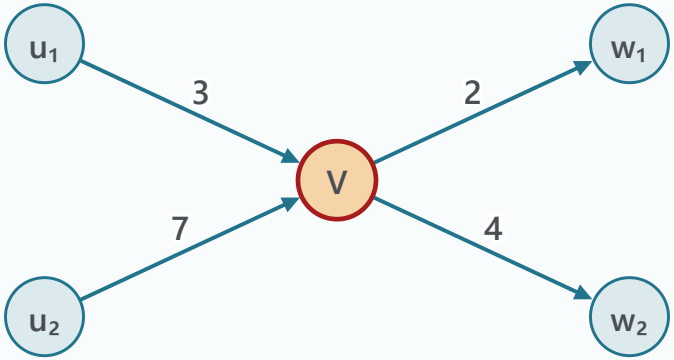
The most popular maximum-flow algorithm in competitive programming: simple, and it usually runs far below its bound.

Push–Relabel

- A. Goldberg and R. Tarjan (1988). It does **not search** for augmenting paths; it works through local operations on vertices.
- It starts by filling every edge out of the source, creating **excess flow**; the flow is a **preflow** that temporarily violates conservation.
- Each vertex carries a **height** label; flow is pushed only **downhill**, from higher to lower, like water.
- If a vertex cannot push its excess, its height is raised (**relabel**).
- When no vertex has excess left, the preflow is a true flow; the **excess at the sink** is the maximum flow.



Preflow and Excess Flow



$e(V) = (3 + 7) - (2 + 4) = \mathbf{4}$

- **Excess:** e(v) = total flow in − total flow out.
- **Preflow:** obeys the capacity constraint, but instead of conservation requires e(v) ≥ 0 (except at s).
- A vertex (other than s and t) with e(v) > 0 is **active**; the algorithm ends when no active vertex is left.

The Height Function and Initialization

Initialization:

- Every edge has flow 0.
- $h(s) = |V|$ (the number of vertices); every other vertex has $h = 0$.
- Every edge out of s is **filled**: $f(s, v) = c(s, v)$, $e(v) = c(s, v)$.

Invariants:

- $h(s) = |V|$ and $h(t) = 0$ never change.
- For every edge (u, v) of the residual network, $\mathbf{h(u) \le h(v) + 1}$.
- So $h(u)$ is a **lower bound** on the residual distance from u to t.

We choose $h(s) = |V|$ because a residual s–t path would have length $\le |V| - 1$, which would force $h(s) \le |V| - 1$. So the residual network never contains an s–t path, and in the end the flow is maximum.

Push and Relabel

Push: push(u, v)

Condition: $e(u) > 0$, $c_f(u, v) > 0$ (the edge's flow is below its capacity) and **$h(u) = h(v) + 1$**.

Push $\delta = \min(e(u), c_f(u, v))$: $e(u) -= \delta$, $e(v) += \delta$. If the edge becomes saturated it is a *saturating* push, otherwise a *nonsaturating* one.

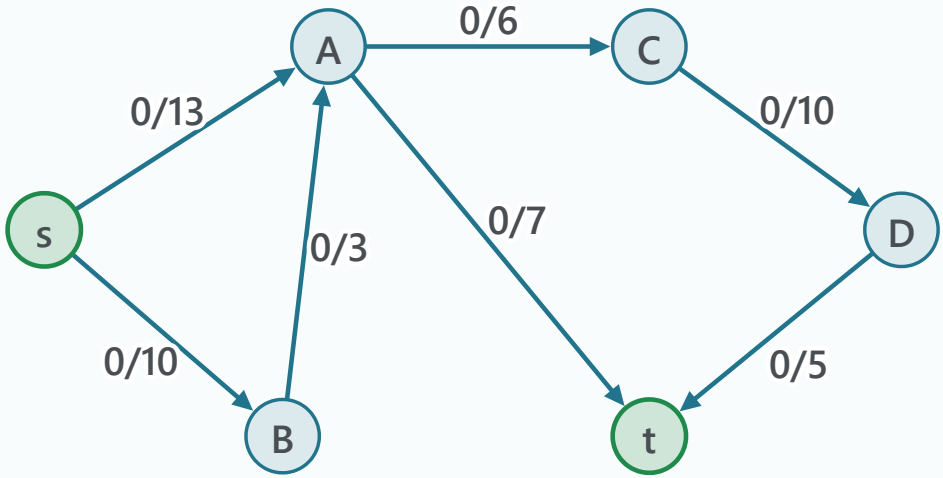
Relabel: relabel(u)

Condition: $e(u) > 0$ but $h(u) \leq h(v)$ for every residual edge (u, v); that is, it cannot push to any neighbor.

$h(u) = 1 + \min h(v)$ over residual edges (u, v). This makes at least one push possible.

Excess that cannot reach the sink is pushed **back to the source** once its height exceeds $|V|$. Flow goes downhill; heights only increase.

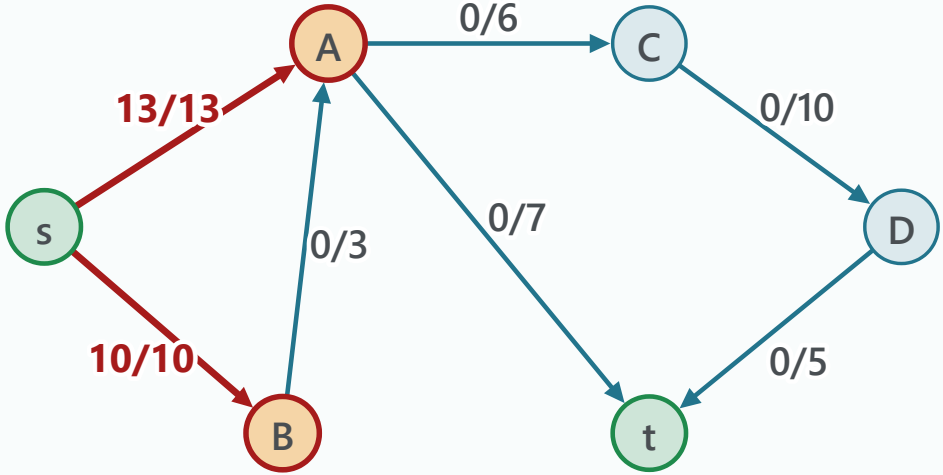
Push–Relabel Example



Vertex	h	e
s	0	–
A	0	0
B	0	0
C	0	0
D	0	0
t	0	0

Take this network (flow/capacity). There are 6 vertices: $|V| = 6$.

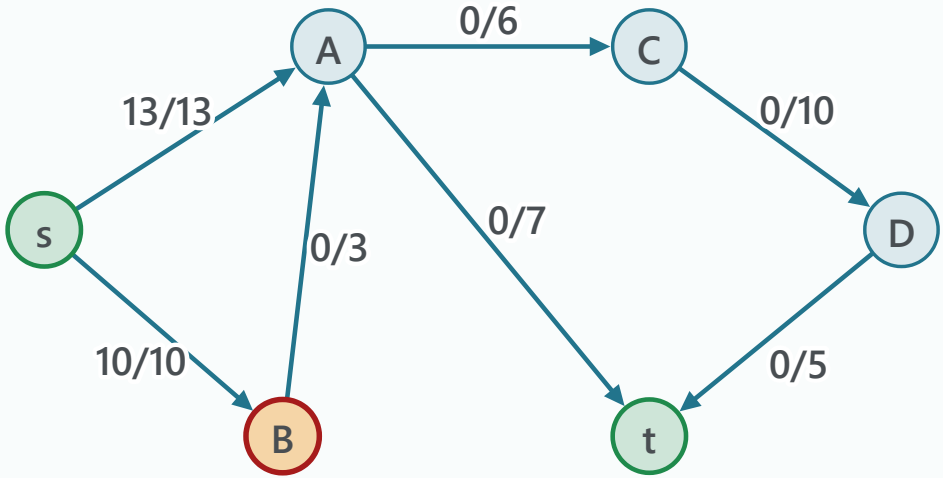
Push–Relabel Example



Vertex	h	e
s	**6**	–
A	0	**13**
B	0	**10**
C	0	0
D	0	0
t	0	0

Start: h(s) = 6; s → A and s → B are filled. e(A) = 13, e(B) = 10.

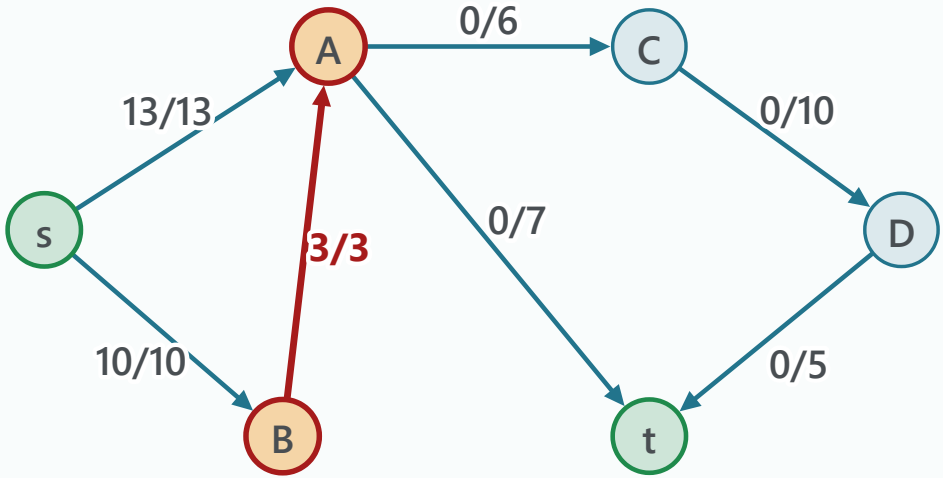
Push–Relabel Example



Vertex	h	e
s	6	–
A	0	13
B	**1**	10
C	0	0
D	0	0
t	0	0

Take B. It is at the same height as A (0), so it cannot send its excess to A → **relabel**: h(B) = 1 + min(h(A) = 0, h(s) = 6) = 1.

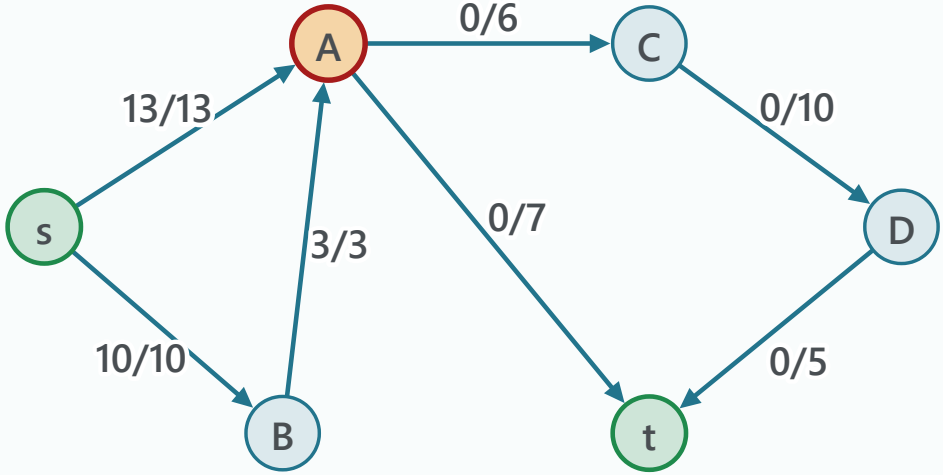
Push–Relabel Example



Vertex	h	e
s	6	–
A	0	**16**
B	1	**7**
C	0	0
D	0	0
t	0	0

B can now push its excess to A: $\delta = \min(10, 3) = 3$. $e(A) = 16$, $e(B) = 7$.

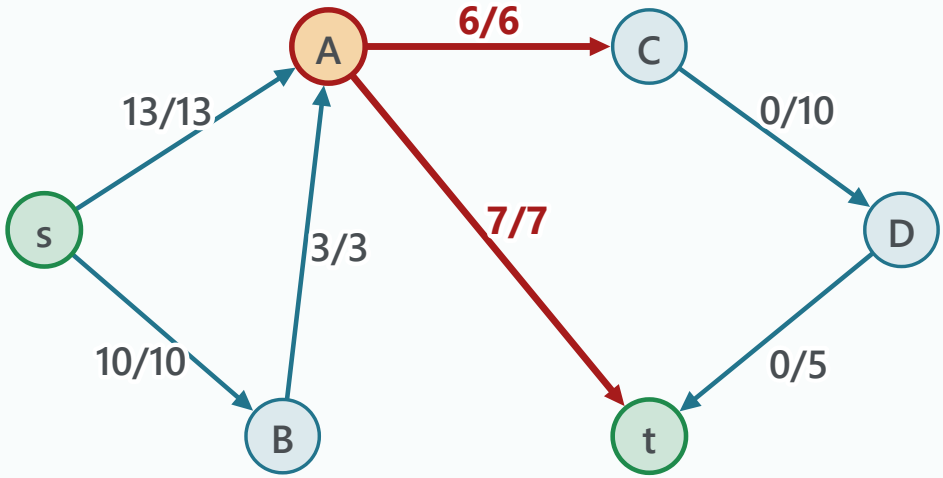
Push–Relabel Example



Vertex	h	e
s	6	–
A	**1**	16
B	1	7
C	0	0
D	0	0
t	0	0

Take A. It has no lower neighbor to push to → relabel: h(A) = 1 + min(h(C) = 0, h(t) = 0, h(B) = 1, h(s) = 6) = 1.

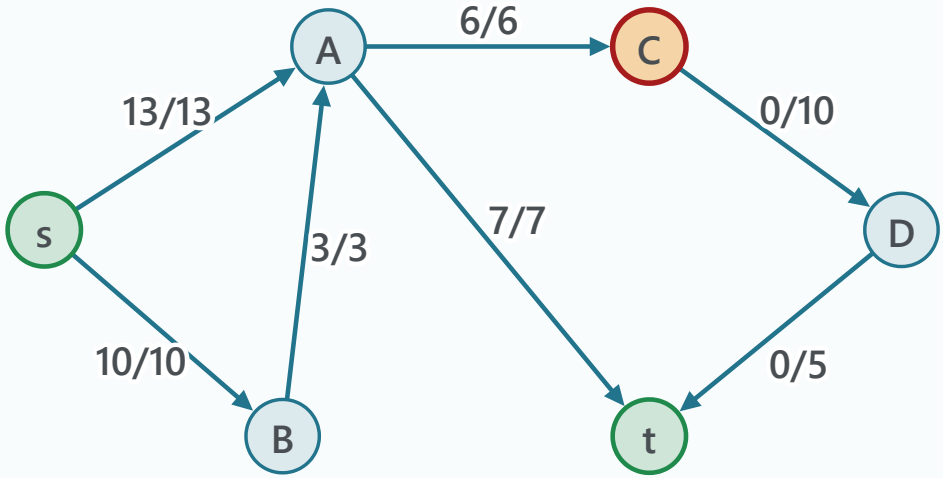
Push–Relabel Example



Vertex	h	e
s	6	–
A	1	**3**
B	1	7
C	0	**6**
D	0	0
t	0	**7**

A now pushes 6 units to C and 7 to the sink (both edges saturate). e(A) = 3, e(C) = 6, 7 at the sink.

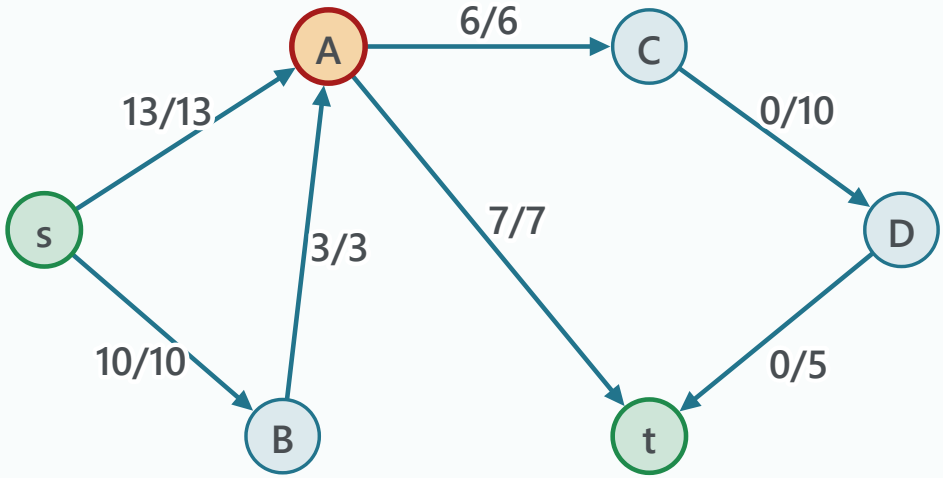
Push–Relabel Example



Vertex	h	e
s	6	–
A	1	3
B	1	7
C	**1**	6
D	0	0
t	0	7

C is relabeled: h(C) = 1 + h(D) = 1. (C's 6 units of excess wait for now; active vertices may be chosen in any order.)

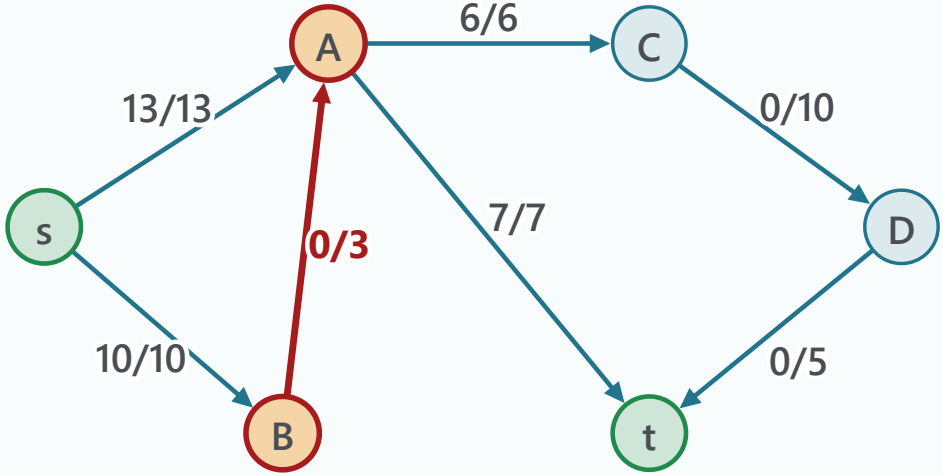
Push–Relabel Example



Vertex	h	e
s	6	–
A	**2**	3
B	1	7
C	1	6
D	0	0
t	0	7

A is relabeled: only B (h = 1) and s (h = 6) remain in the residual network → h(A) = 2.

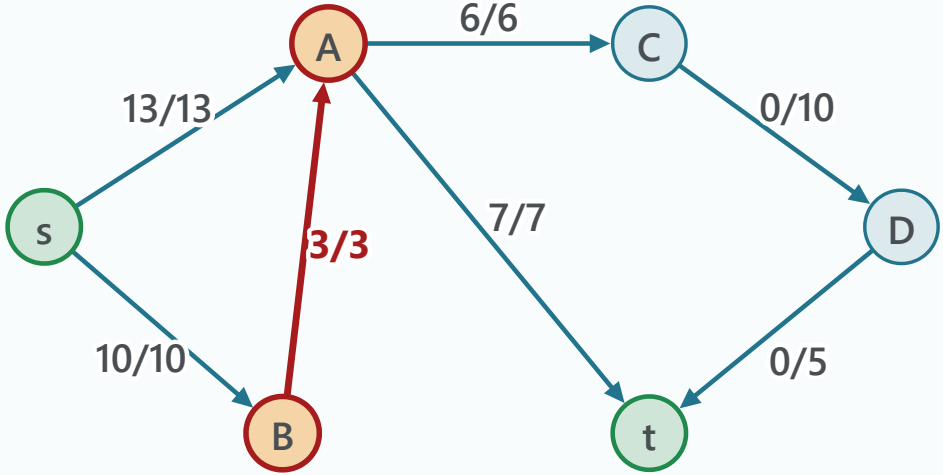
Push–Relabel Example



Vertex	h	e
s	6	–
A	2	**0**
B	1	**10**
C	1	6
D	0	0
t	0	7

A is higher than B: it pushes 3 units to B (cancelling flow on B → A). e(A) = 0, e(B) = 10.

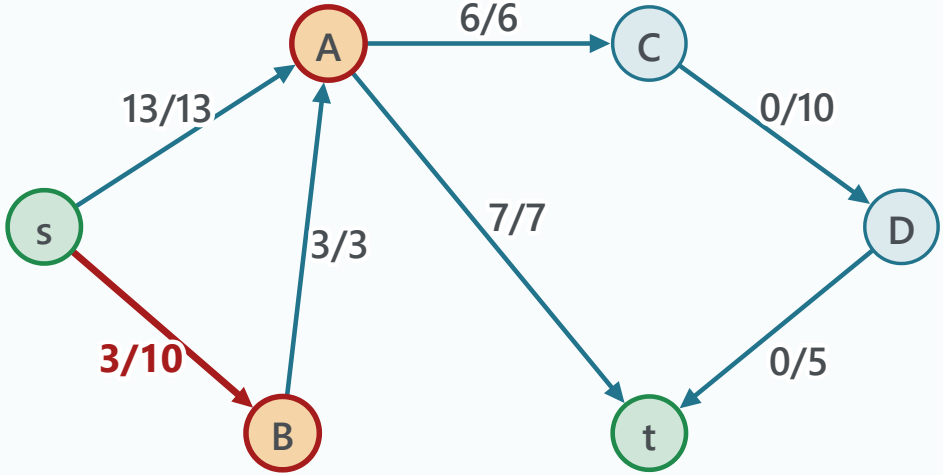
Push–Relabel Example



Vertex	h	e
s	6	–
A	2	**3**
B	**3**	**7**
C	1	6
D	0	0
t	0	7

B has no other edge to use → relabel: h(B) = 3; it pushes 3 units back to A. e(A) = 3, e(B) = 7.

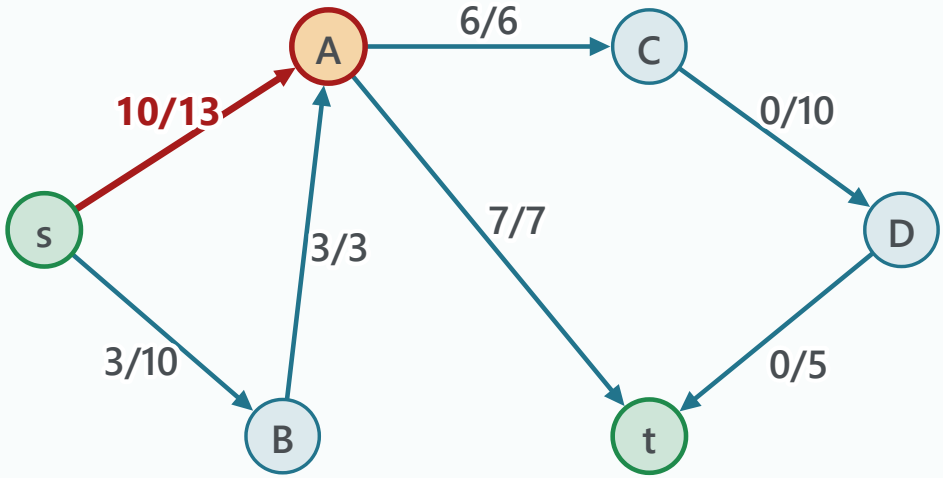
Push–Relabel Example



Vertex	h	e
s	6	–
A	**6**	3
B	**7**	**0**
C	1	6
D	0	0
t	0	7

Relabels and pushes alternate between A and B (h(A) = 4, h(B) = 5, h(A) = 6, h(B) = 7). Once B rises above the source, it pushes 3 units to A and **7 units to the source**.

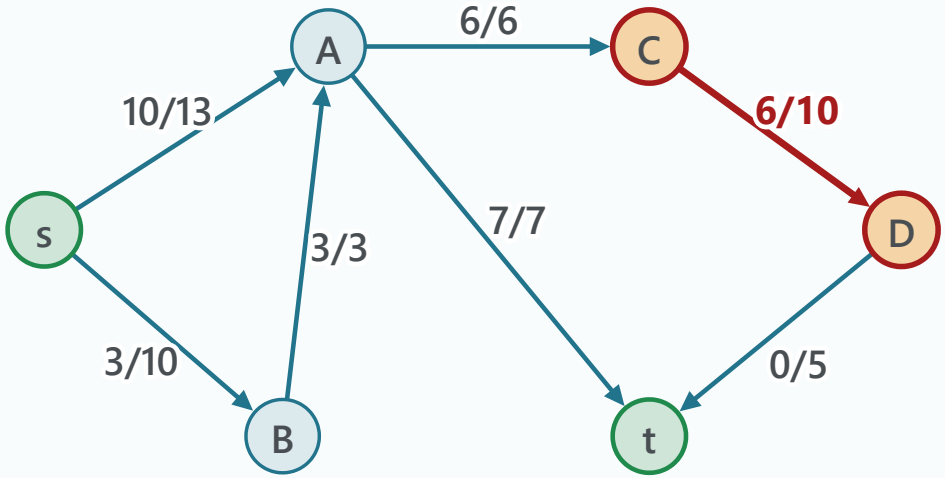
Push–Relabel Example



Vertex	h	e
s	6	–
A	**7**	**0**
B	7	0
C	1	6
D	0	0
t	0	7

A is relabeled: h(A) = 1 + min(h(s) = 6, h(B) = 7) = 7; it pushes 3 units **to the source**. A and B have no excess left.

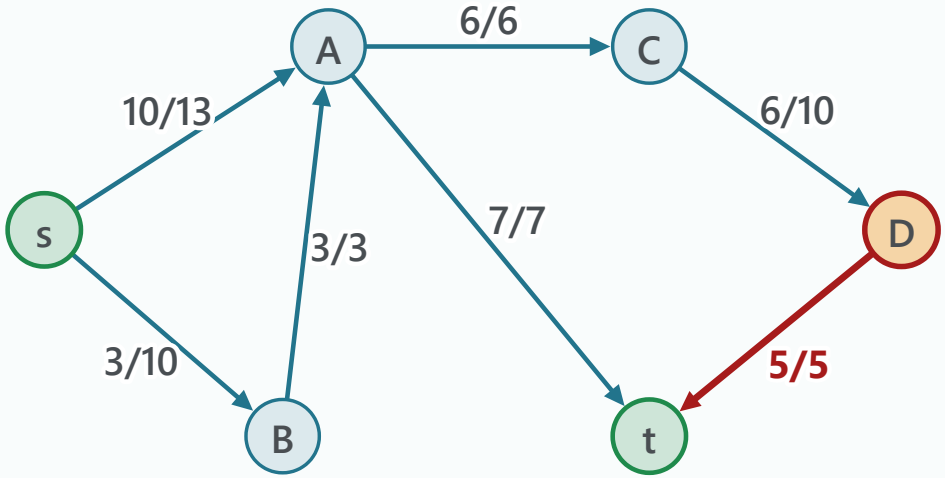
Push–Relabel Example



Vertex	h	e
s	6	–
A	7	0
B	7	0
C	1	**0**
D	0	**6**
t	0	7

Take C: its 6 units of excess are pushed to D (h(C) = 1 > h(D) = 0).

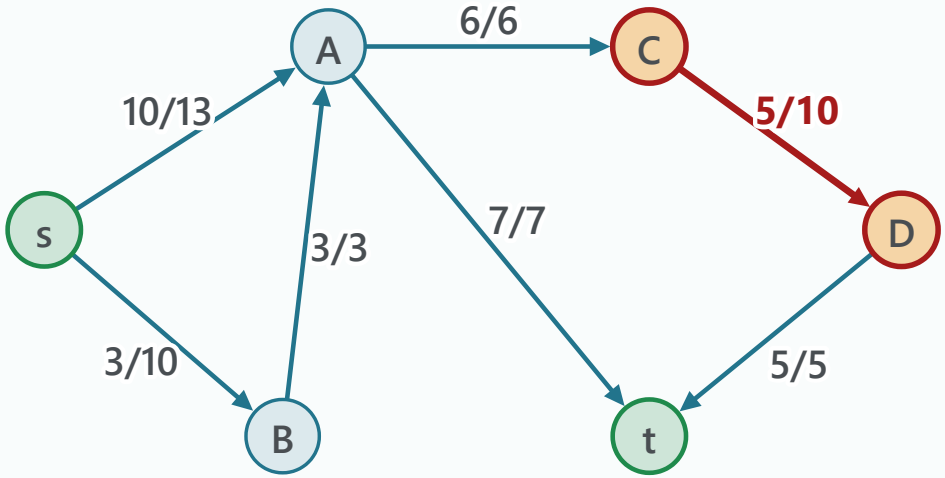
Push–Relabel Example



Vertex	h	e
s	6	–
A	7	0
B	7	0
C	1	0
D	**1**	**1**
t	0	**12**

D is relabeled (h = 1) and pushes 5 units to the sink. D still has **1** unit of excess.

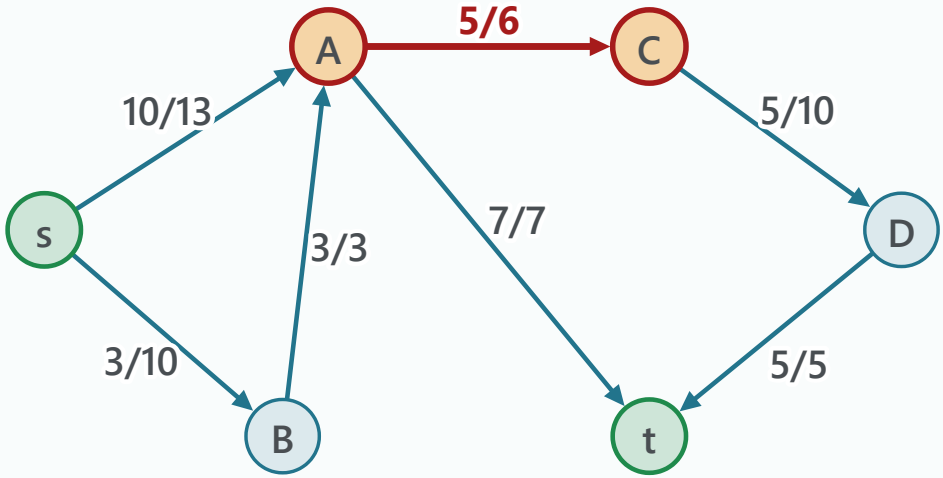
Push–Relabel Example



Vertex	h	e
s	6	–
A	7	0
B	7	0
C	1	**1**
D	**2**	**0**
t	0	12

D is relabeled: the edge to the sink is full and its only neighbor is C (h = 1) → h(D) = 2. It pushes 1 unit to C.

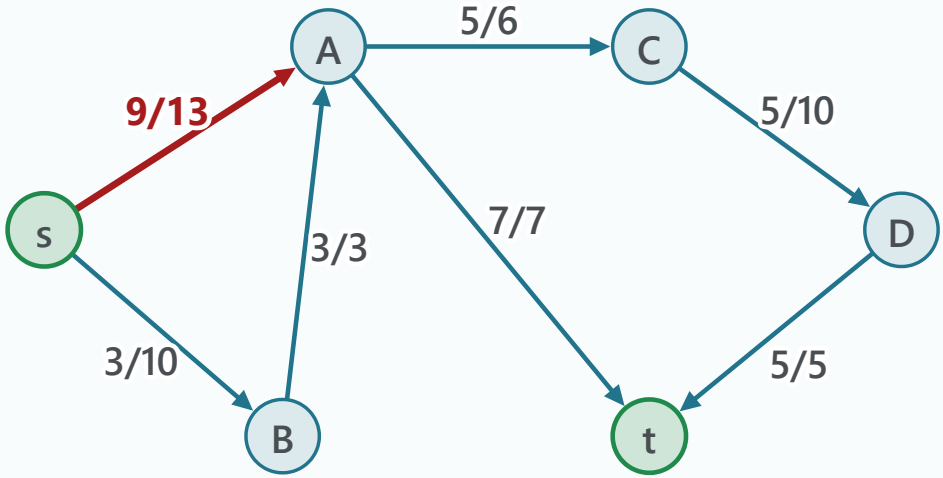
Push–Relabel Example



Vertex	h	e
s	6	–
A	7	**1**
B	7	0
C	**8**	**0**
D	**8**	0
t	0	12

Relabels and pushes alternate between C and D until C rises above A (7) (C: 3, 5, 7, 8; D: 4, 6, 8). C pushes its 1 unit of excess to A.

Push–Relabel Example



Vertex	h	e
s	6	–
A	7	**0**
B	7	0
C	8	0
D	8	0
t	0	12

A pushes its 1 unit of excess to the source. No active vertex is left: **maximum flow = flow into the sink = 7 + 5 = 12** = 9 + 3 out of the source.

Push–Relabel: Pseudocode

```
PUSH_RELABEL(G, source, sink):
  height[source] = |V|
  height[v] = 0, v ≠ source
  excess[v] = 0, for every v
  flow[e] = 0, for every edge e
  // preflow: fill the source edges
  for each (source, v) ∈ E:
    c = capacity[source, v]
    flow[source, v] = c
    flow[v, source] = -c
    excess[v] = c
    excess[source] -= c
```

```
while some u has excess[u] > 0:
  // u ≠ source, u ≠ sink
  if residual(u,v) > 0 and
       height[u] == height[v]+1:
    // Push
    delta = min(excess[u], residual(u,v))
    flow[u,v] += delta
    flow[v,u] -= delta
    excess[u] -= delta
    excess[v] += delta
  else:
    // Relabel
    m = ∞
    for each (u, v) in G[u]:
      if flow[u,v] < capacity[u,v]:
        m = min(m, height[v])
    height[u] = m + 1
return excess[sink]
// residual = capacity - flow
```

Java: Push and Relabel

```java
boolean push(Node n) {
  for (Edge e : n.edges) {
    if (n.height > e.to.height
        && e.flow != e.cap) {
      int flow = Math.min(
        e.cap - e.flow, n.excess);
      n.excess -= flow;
      e.to.excess += flow;
      e.flow += flow;
      updateReverse(e, flow);
      return true;   // pushed
    }
  }
  return false;
}
```

```java
void relabel(Node n) {
  int minH = Integer.MAX_VALUE;
  for (Edge e : n.edges) {
    // residual edges only
    if (e.flow != e.cap
        && e.to.height < minH) {
      minH = e.to.height;
      n.height = minH + 1;
    }
  }
}
```

Main loop: while an active vertex exists, `if (!push(n)) relabel(n);`. The answer is the sink's excess. Reverse edges hold negative flow (capacity 0).

Push–Relabel: Complexity

Operation	Total count	Reason
Relabel	$O(V^2)$	each height is $\leq 2\|V\| - 1$ and only increases
Saturating push	$O(V \cdot E)$	to saturate an edge again, the heights at both ends must rise by ≥ 2
Nonsaturating push	$O(V^2 \cdot E)$	via the potential function $\Sigma\, h(v)$ (active v)

- The generic algorithm is **$O(V^2 \cdot E)$** in the worst case; a relabel scans at most degree-many edges ($O(V \cdot E)$ in total).
- $O(V^3)$ with **FIFO** selection, $O(V^2 \cdot \sqrt{E})$ with **highest-label** selection.
- With the *gap* and *global relabeling* heuristics it is among the fastest general-purpose maximum-flow methods in practice.

Comparing the Algorithms

Algorithm	Basic operation	Time	On the example network ($\lvert f^* \rvert = 19$)
Ford-Fulkerson	any augmenting path (DFS)	$O(E \cdot \lvert f^* \rvert)$	5 augmenting paths (order of Example 2)
Edmonds-Karp	shortest augmenting path (BFS)	$O(V \cdot E^2)$	4 augmenting paths, 5 BFS runs
Dinic	level graph + blocking flow	$O(V^2 \cdot E)$	2 phases, 3 BFS runs
Push–relabel	local push / relabel	$O(V^2 \cdot E)$, FIFO $O(V^3)$	no path search

Choosing

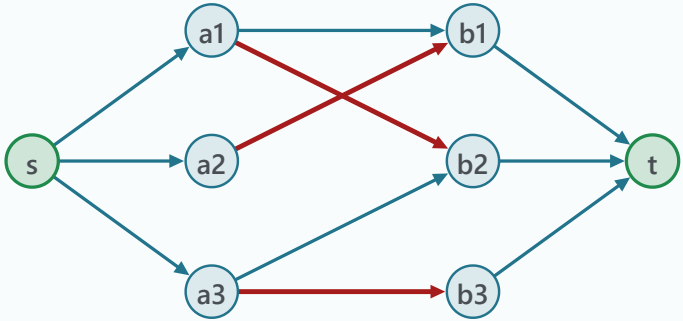
Small integer capacities: Ford-Fulkerson is enough. General purpose: Dinic. Very large, dense networks: push–relabel.

Common ground

All of them find the same $\lvert f^* \rvert$; how the flow is spread over the edges may differ. The min cut is read off the final residual network.

Applications

- **Bipartite matching:** worker–job and student–project assignment (unit capacities).
- **Edge-disjoint paths** (Menger): maximum flow with unit capacities.
- **Image segmentation:** separating foreground from background with a minimum cut.
- **Network reliability:** the smallest total capacity that must be cut to split the network.
- **Transportation and logistics:** pipelines, traffic, airline scheduling, project selection.



Maximum flow 3 = maximum matching: a_1–b_2, a_2–b_1, a_3–b_3.

Summary

Topic	Key points
Flow	$0 \le f \le c$; flow in = flow out at internal vertices; $\lvert f \rvert$ = net flow out of s
Residual network	forward $c - f$, reverse f; an augmenting path is an $s \to t$ path in G_f, bottleneck = smallest c_f
Cut	$c(S, T)$ = capacities from S to T; maximum flow = min cut; S = what s reaches in the final G_f
Ford-Fulkerson	any augmenting path; $O(E \cdot \lvert f^* \rvert)$, terminates with integer capacities
Edmonds-Karp	shortest paths via BFS; $O(V \cdot E^2)$
Dinic	BFS level graph + DFS blocking flow; $\le V - 1$ phases; $O(V^2 \cdot E)$
Push–relabel	$h(s) = \lvert V \rvert$, preflow; flow is pushed downhill, and a vertex that cannot push is relabeled; $O(V^2 \cdot E)$

SUMMARY

Common Mistakes

Forgetting reverse edges

Searching with forward edges only is a greedy algorithm and can miss the maximum (18 < 19 in the example).

Computing the bottleneck from capacities

The bottleneck is the minimum of the **residual capacities**, not the capacities: c − f (or f on a reverse edge).

Thinking a min cut counts edges

A min cut cuts the smallest **total capacity**, not the fewest **edges**.

Counting T → S edges

Only edges from S to T contribute to the cut capacity; edges going back are not added.

Thinking Ford-Fulkerson is polynomial

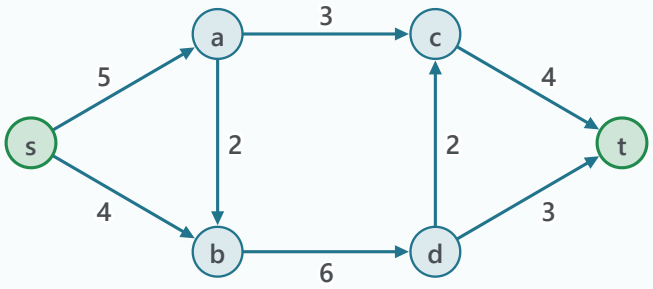
$O(E \cdot |f^*|)$ depends on the capacity values; BFS (Edmonds-Karp) removes that dependence.

Pushing the wrong way

Flow goes downhill, only when $h(u) = h(v) + 1$; initially $h(s)$ must be $|V|$.

SUMMARY

Worked Problems



1. What upper bounds on |f*| do the source side and the sink side give?
2. Which augmenting paths does Edmonds-Karp find, in order? (BFS tries neighbors alphabetically.)
3. What is the maximum flow, and a minimum cut?
4. What are $c_f(s, a)$ and $c_f(c, d)$ at the end?
5. What are the h and e values at the start of push–relabel?

▶ **Show answers**

SUMMARY

Exercises

1. If one edge's capacity grows by 1, by how much can the maximum flow grow at most?
2. If every capacity is multiplied by k, how do the maximum flow and the minimum cut change?
3. How do you model the problem when vertices have capacities c(v)?
4. Write a Java method that finds a minimum cut from the final residual network.
5. On the "bad case" network, at most how many augmentations do DFS-based Ford-Fulkerson and Edmonds-Karp make?
6. How is the number of edge-disjoint s–t paths found with maximum flow?

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapter 24: Maximum Flow).
- Kleinberg and Tardos, *Algorithm Design*, Pearson, 2006 (Chapter 7: Network Flow).
- Sedgewick and Wayne, *Algorithms*, 4th ed., 2011 (Section 6.4: Maximum Flow).
- L. R. Ford and D. R. Fulkerson, "Maximal Flow Through a Network", *Canadian Journal of Mathematics* 8, 1956.
- J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of the ACM* 19(2), 1972.
- E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation", *Soviet Math. Doklady* 11, 1970.
- A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem", *Journal of the ACM* 35(4), 1988.

The End

Chapter 8: Network Flow

Previous: Chapter 7: Minimum Spanning Trees · Next: Chapter 9: String Matching and Compression