CS310 - Advanced Data Structures and Algorithms

Flow Networks

December 8, 2021

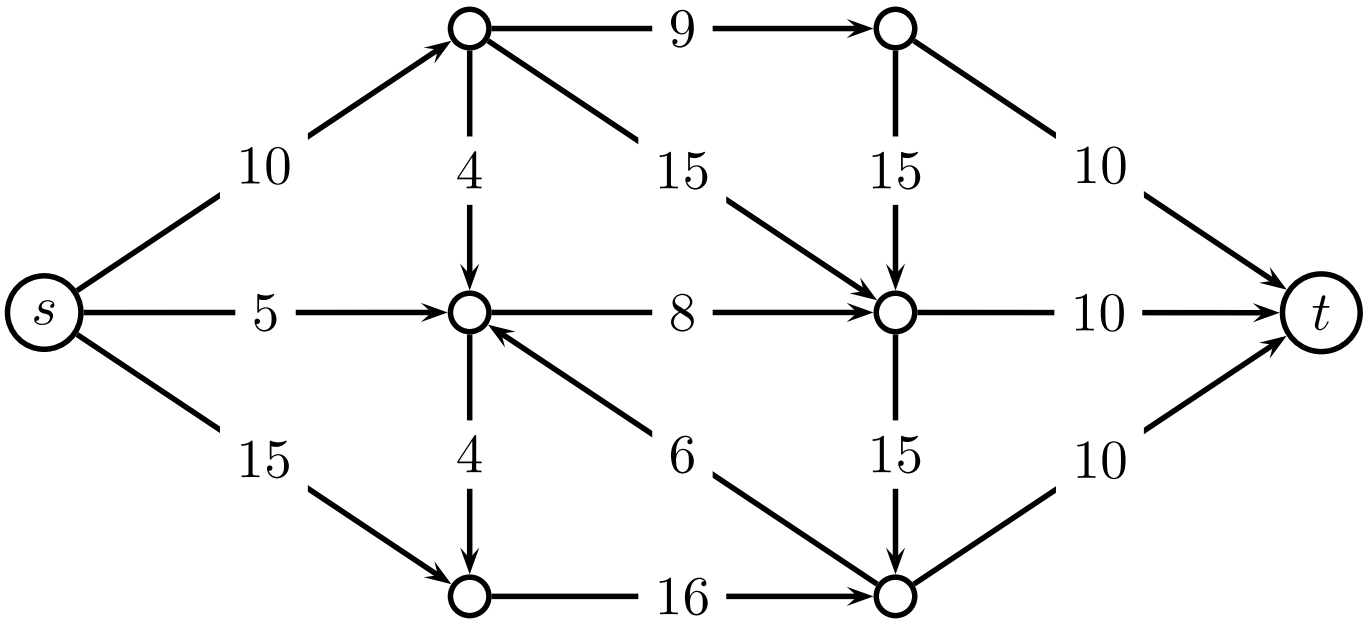
Definition – Flow Networks

- A *flow graph* is a directed graph with two distinguished vertices, s (the "source") and t (the "sink" or "target").
- We also assume that There is at least one path from s to t.
- In fact, even more is true: for each node u in the graph, there is at least one path from s through u.
- Every edge has a positive *capacity*
- You can think of the edge as a pipe and the capacity as the maximum amount of material that can flow through the pipe at any moment.

Definition – Flow Networks

- A *flow graph* is a directed graph with two distinguished vertices, s (the "source") and t (the "sink" or "target").
- We also assume that There is at least one path from s to t.
- In fact, even more is true: for each node u in the graph, there is at least one path from s through u.
- Every edge has a positive *capacity*
- You can think of the edge as a pipe and the capacity as the maximum amount of material that can flow through the pipe at any moment.

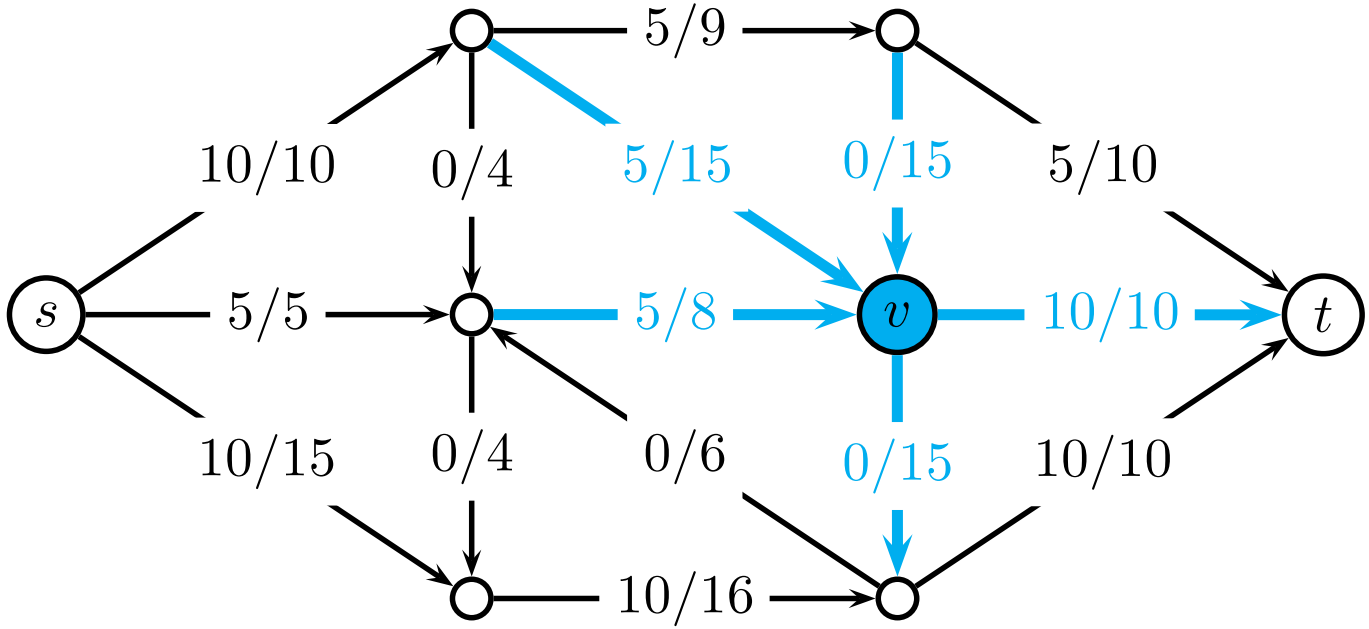
Example – Flow Networks



Definition – s-t Flow

- An st-flow (flow) is an assignment of values to the edges such that:
- Capacity constraint: For every edge, $0 \leq \textit{flow} \leq \textit{capacicy}$
- Local equilibrium: inflow = outflow at every vertex (except s and t)
- In the following example: inflow at v $= 5 + 5 + 0 = 10$. Outflow $= 10 + 0 = 0$.
- Number on the left – flow. Number on the right – capacity.
- The *value* of the flow is the flow going into t (or out of s).
- In the example below, the value of the flow is 5+10+10=25.

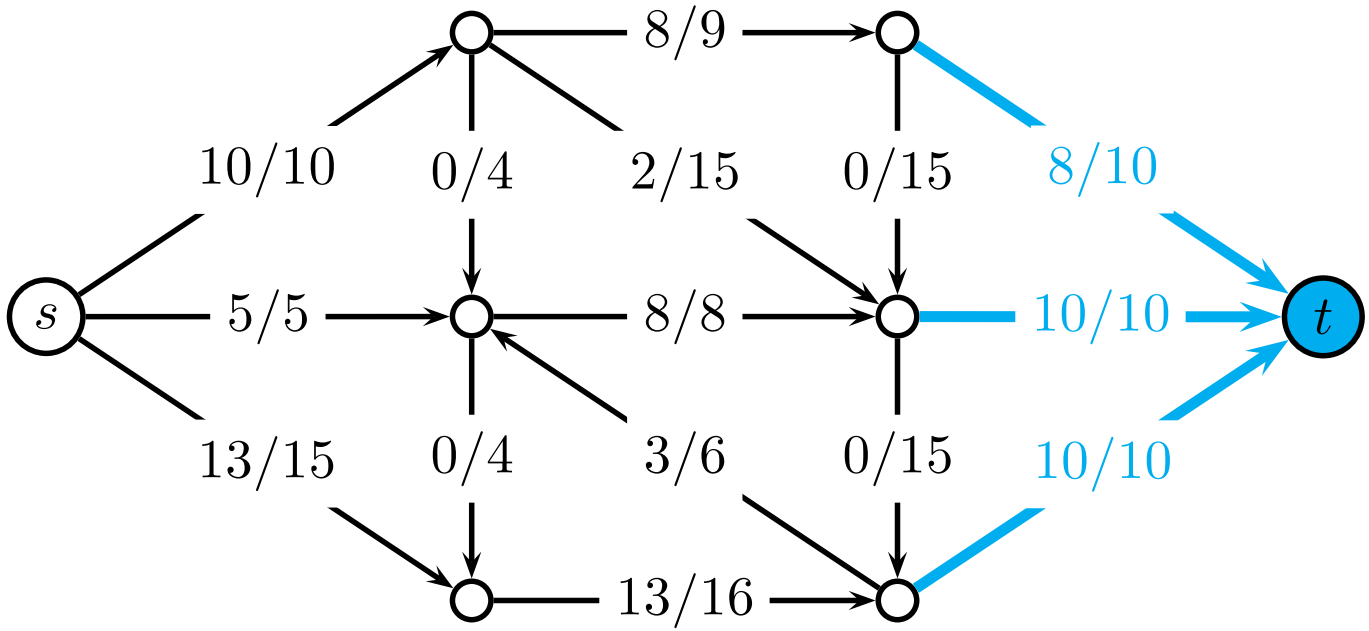
Example – s-t flow



Definition – max-flow

- The *max-flow* problem is to find the flow with maximum capacity.
- In real life – find the maximum amount of supply/data/materials you can deliver at one time.
- In the example below the max-flow is 28.

Example – max-flow

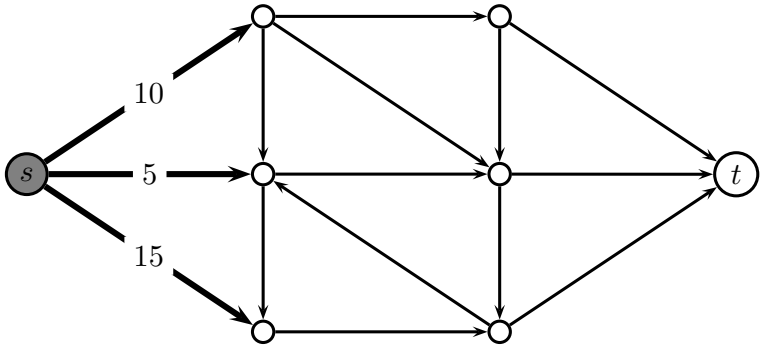


Definition – A Cut

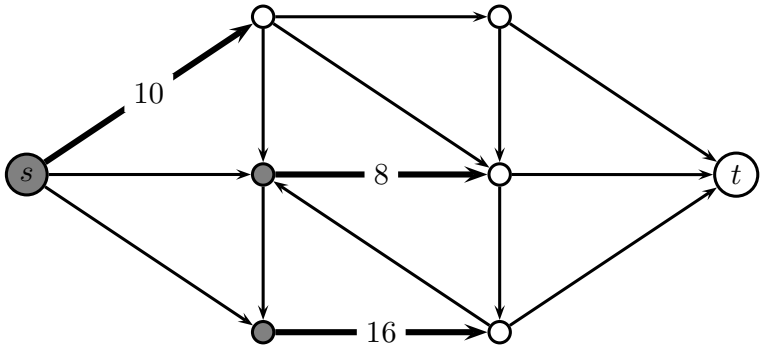
- an *st-cut* (or cut) is a a partition of the vertices into two disjoint sets A and B, with s in A and t in B.
- The other vertices can go either way.
- The *capacity* of the cut is the sum of the capacities of the edges from A to B.
- The Minimum st-cut (mincut) problem: Find a cut of minimum capacity.

Example – Cuts

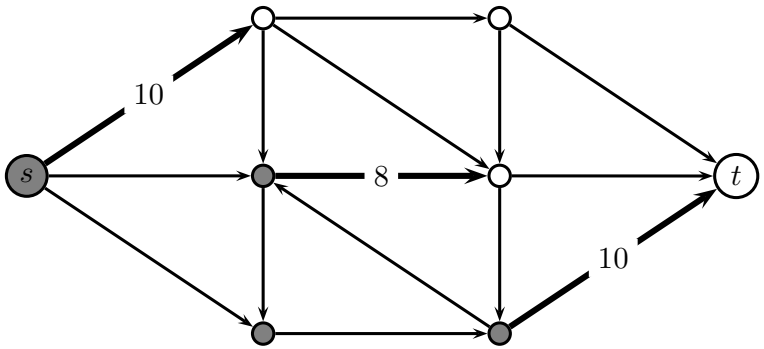
Capacity: 30



Capacity: 34

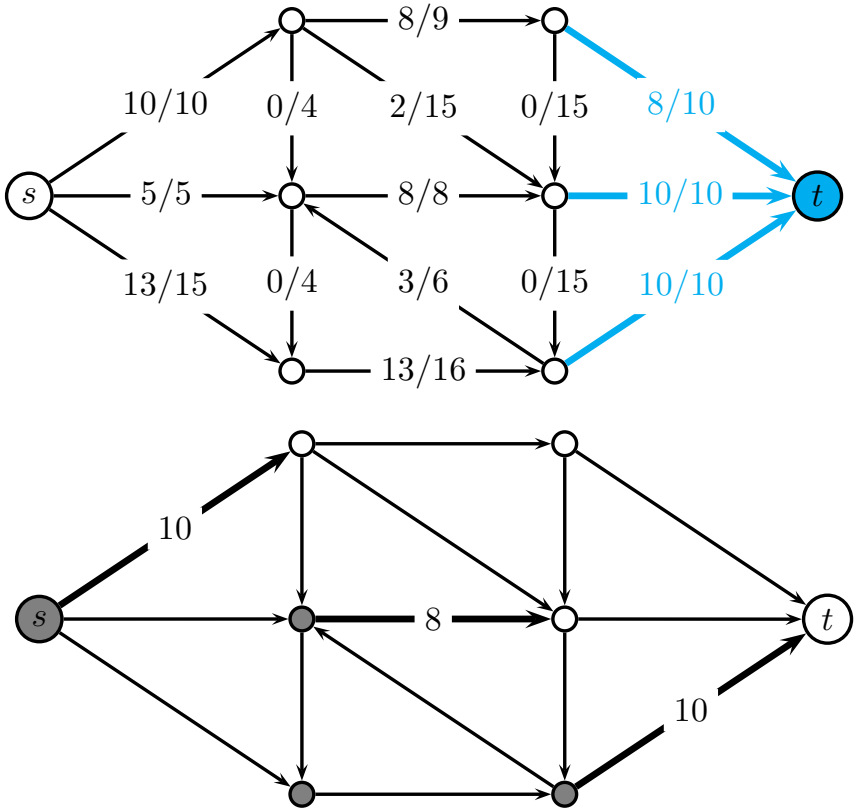


Capacity: 28



Max-flow Min-cut Duality

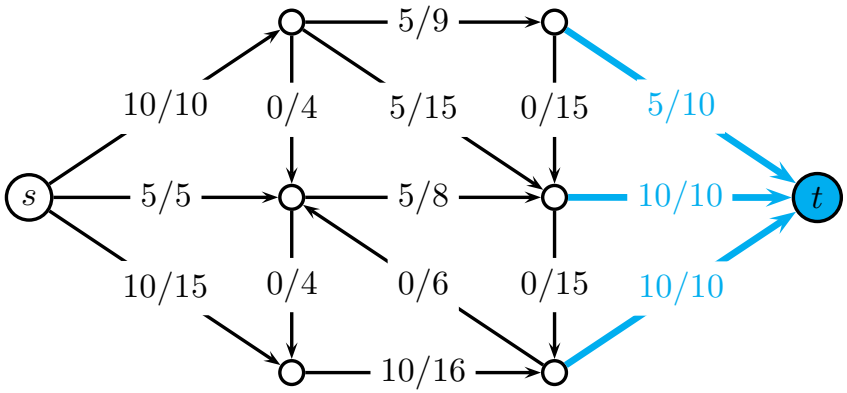
- The two problems are dual!
- The value of the maximum flow = capacity of minimum cut.



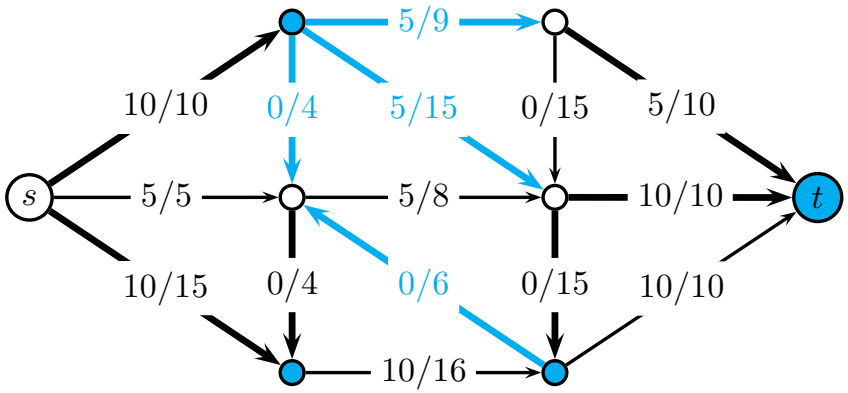
Relationships Between Flows and Cuts

- The *net flow* across a cut (A,B) is the sum of the flow on the edges from A to B, minus the sum of the flow on the edges from B to A.
- See two examples below:

Net flow = 25



Net flow = 35 - 10 = 25



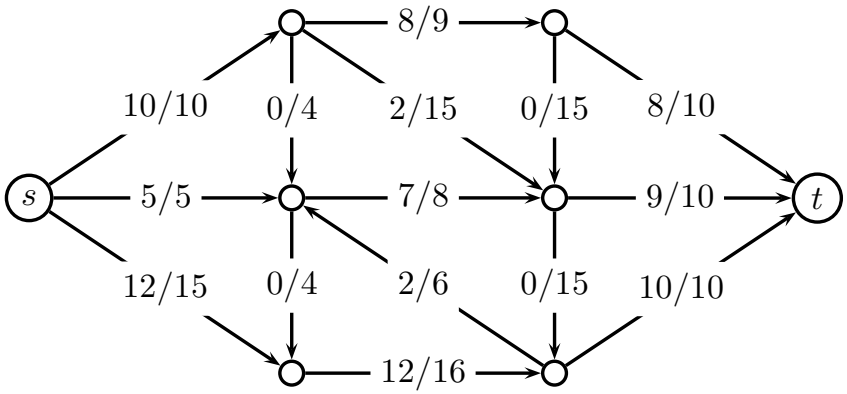
Relationships Between Flows and Cuts

- **Flow-value lemma:** Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f.
- **Intuition:** Conservation of flow (and the fact that s is in A and t is in B).
- **Proof:** By induction on the size of B.
  - Base case: True for $B = \{t\}$.
  - Induction step: remains true by local equilibrium when moving any vertex from A to B.
- **Corollary:** Outflow from s = inflow to t = value of flow.

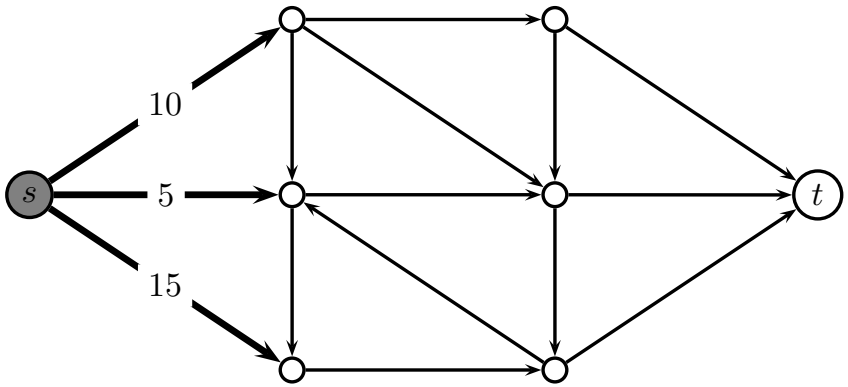
Relationships Between Flows and Cuts

- **Weak Duality:** Let f be any flow and let (A,B) be any cut. Then, the value of the flow is $\leq$ the capacity of the cut.
- **Proof:** Value of flow f = net flow across cut (A, B) $\leq$ capacity of cut (A, B) (since the flow is bound by the capacity).

Value of flow = 27



Capacity of cut = 30



Augmenting Paths

- An *Augmenting path* is a path from s to t that can be used to increase the s-t flow.
- The *bottleneck capacity* is the minimum residual (remaining) capacity of an edge on a path.

Ford-Fulkerson's algorithm for max-flow:

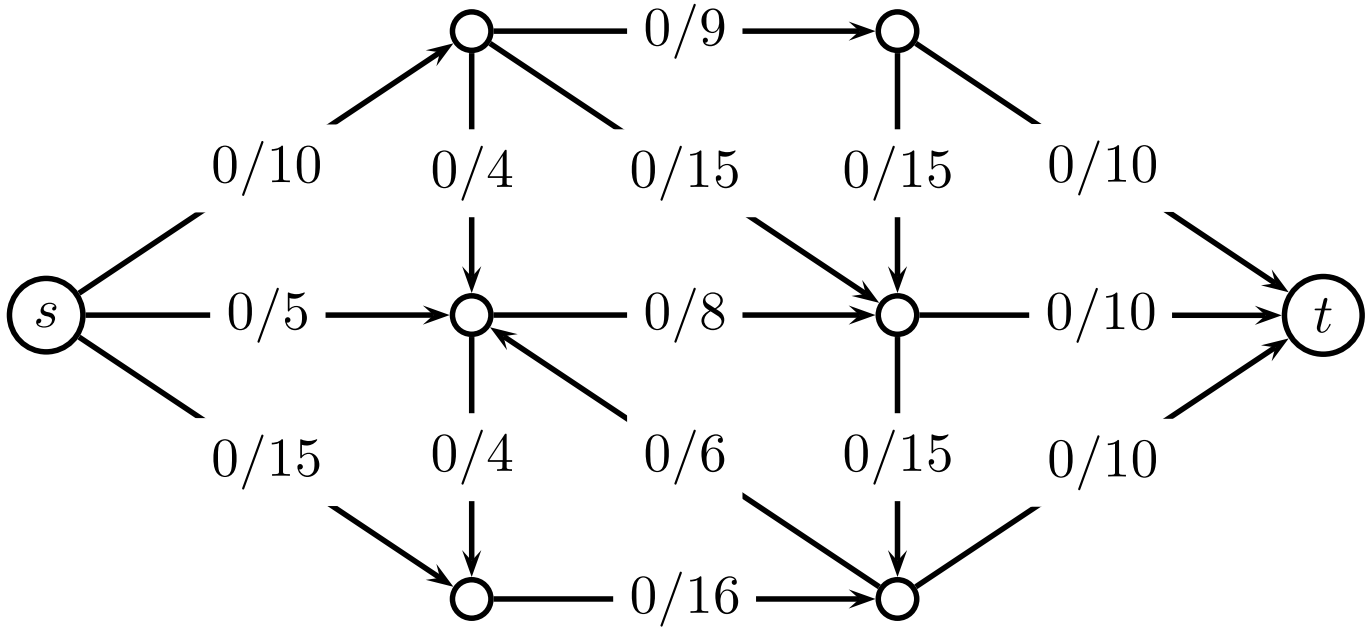
- Start with 0 flow.
- While there are augmenting paths: Find an augmenting path.
- Find the bottleneck capacity across the path.
- Increase the flow across the path by the bottleneck capacity.
- Update residual graph.

Residual Graphs

- Given a flow network, the *residual graph* shows us how much could the flow on each edge in **either direction**.
- Notice that every edge is replaced by two anti-parallel edges.
- The **forward edge** goes in the direction of the original edge in the flow graph and tells us how much we can still flow.
- This number is the capacity minus current flow.
- The **backwards edge** goes in the opposite direction and tells us how much we can "return".
- This number is the current flow, but pointing in the opposite direction.
- For convenience, if any number is 0, we will omit it.

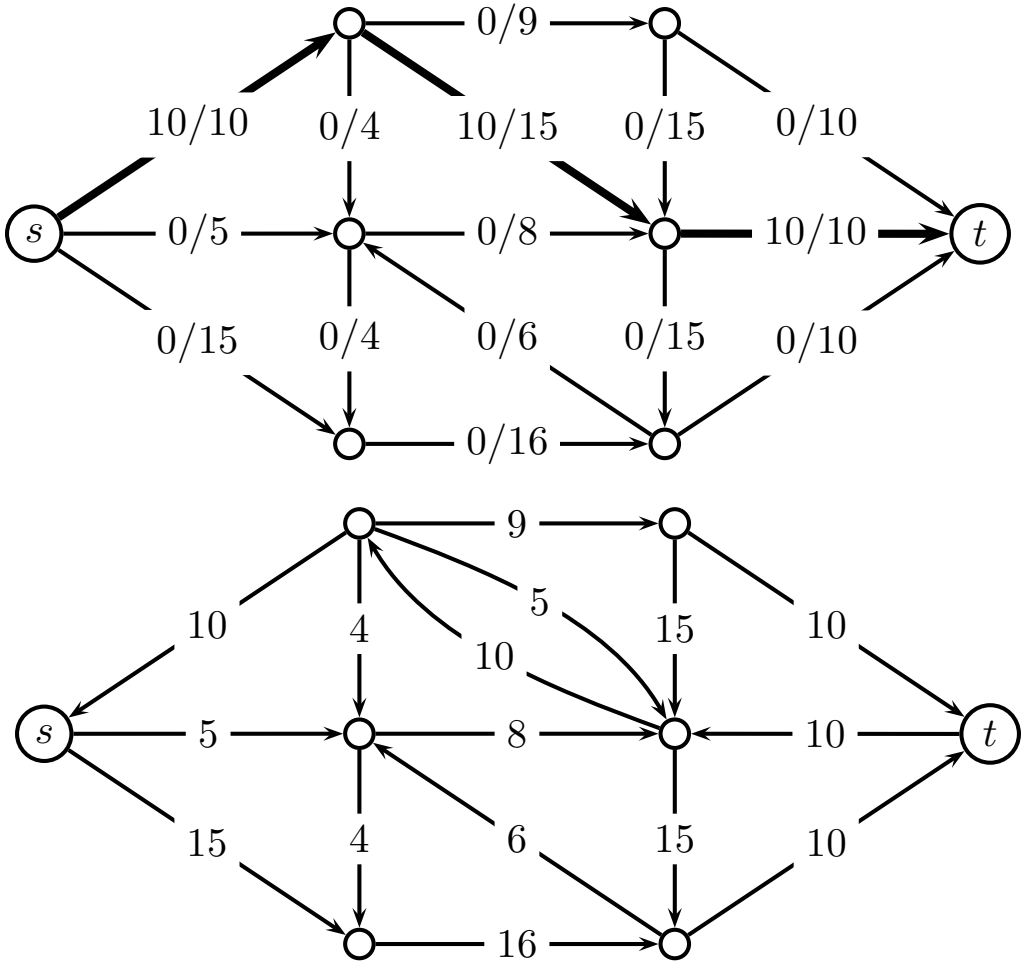
Example

Start with 0 flow. The graph is the original flow graph (backward edges are empty since we did not flow anything):



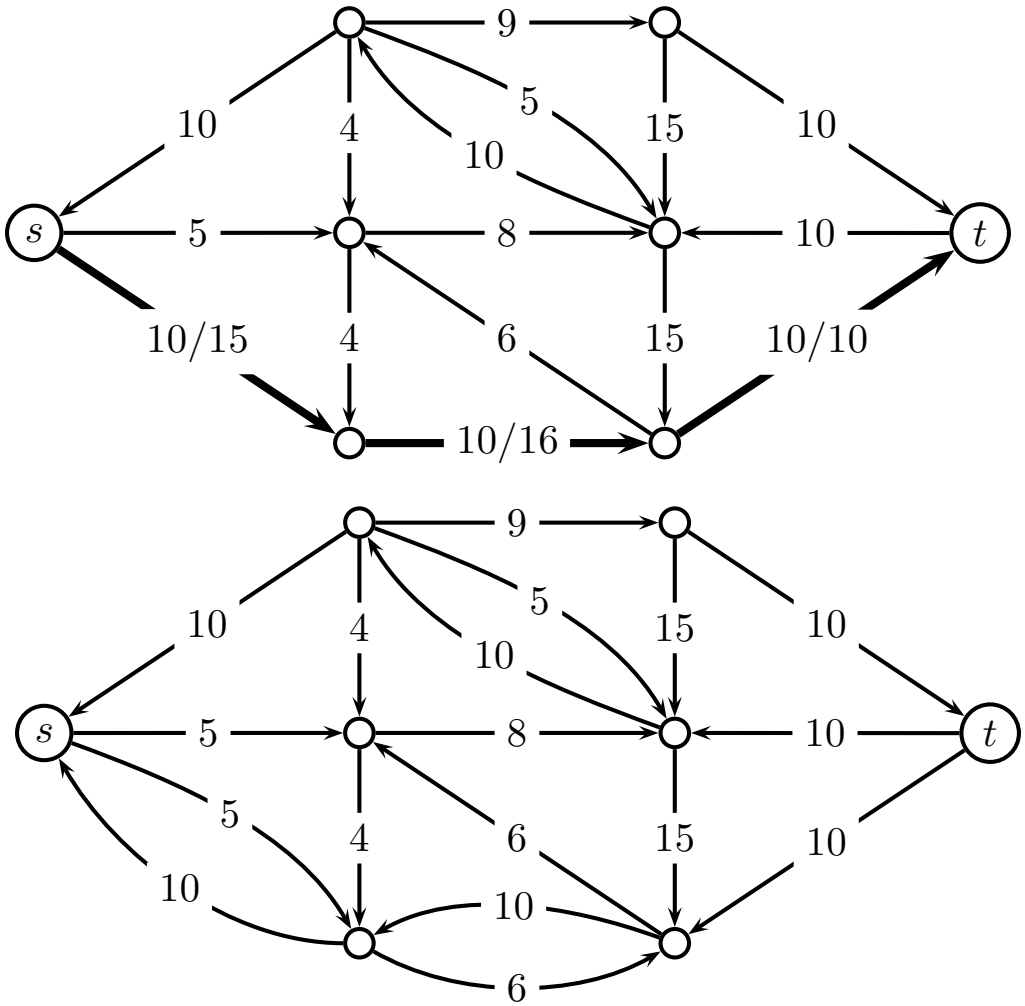
Example

The first augmenting path is calculated The bottleneck capacity is 10. The residual graph is shown below.



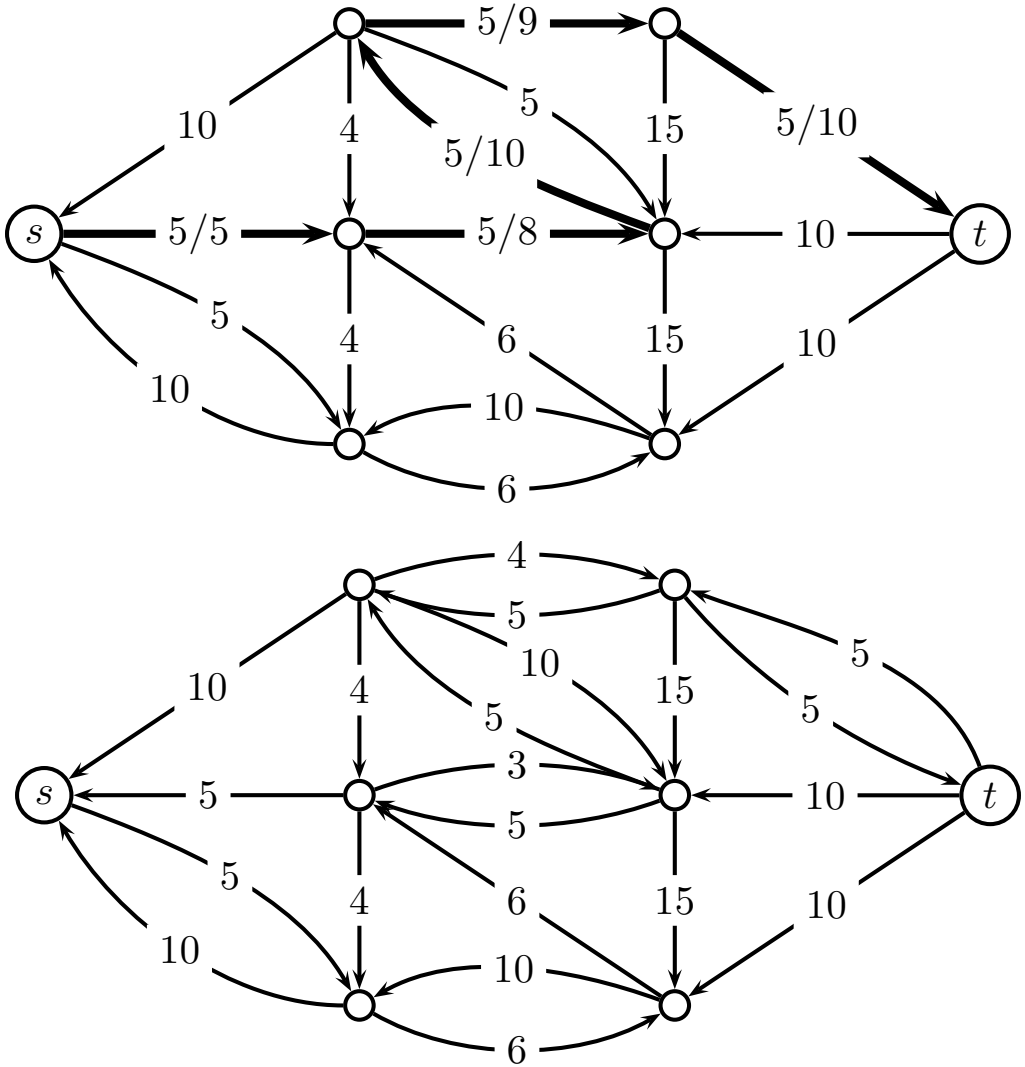
Example

The second augmenting path is calculated The bottleneck capacity is 10. The residual graph is shown below.



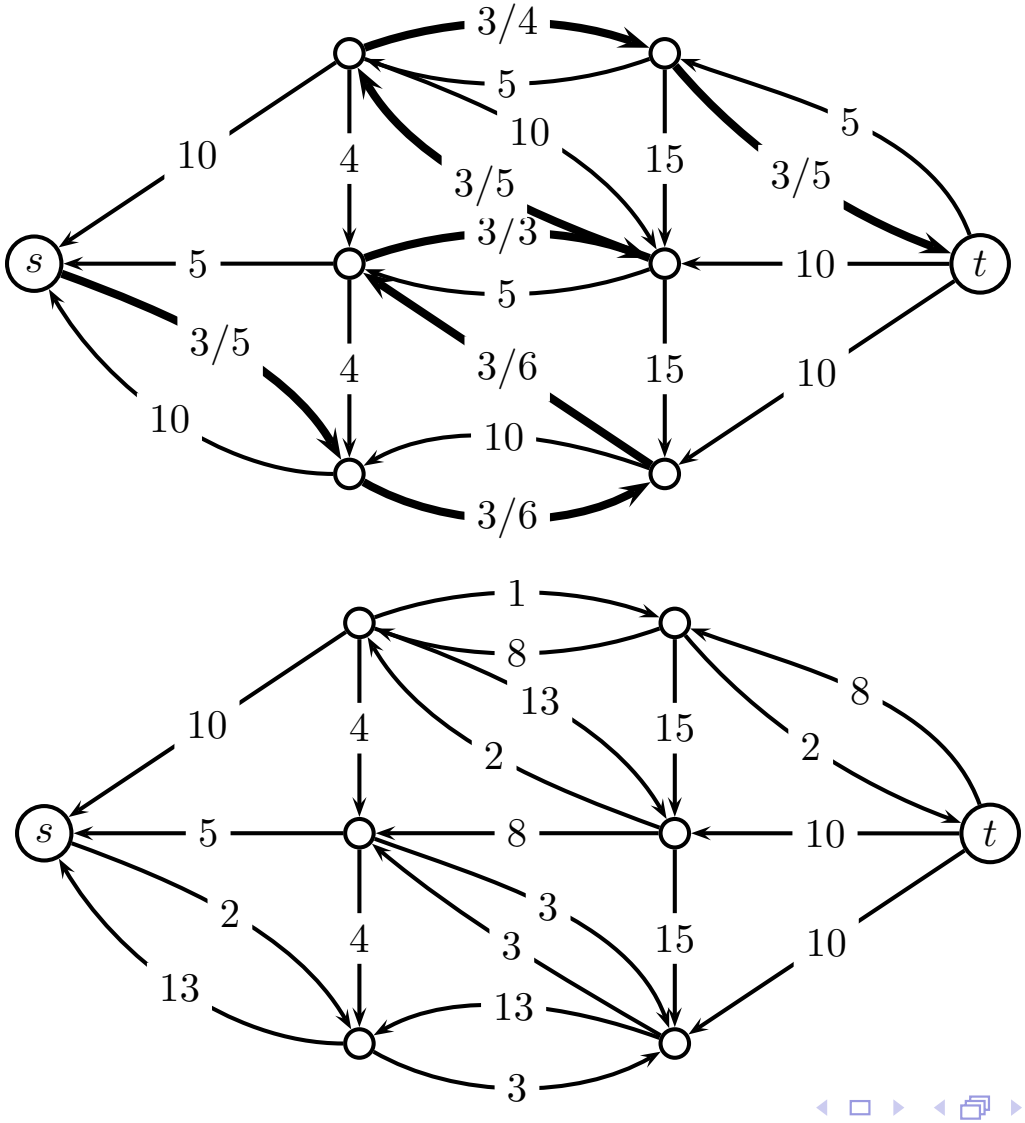
Example

The third augmenting path is calculated The bottleneck capacity is 5. The residual graph is shown below.



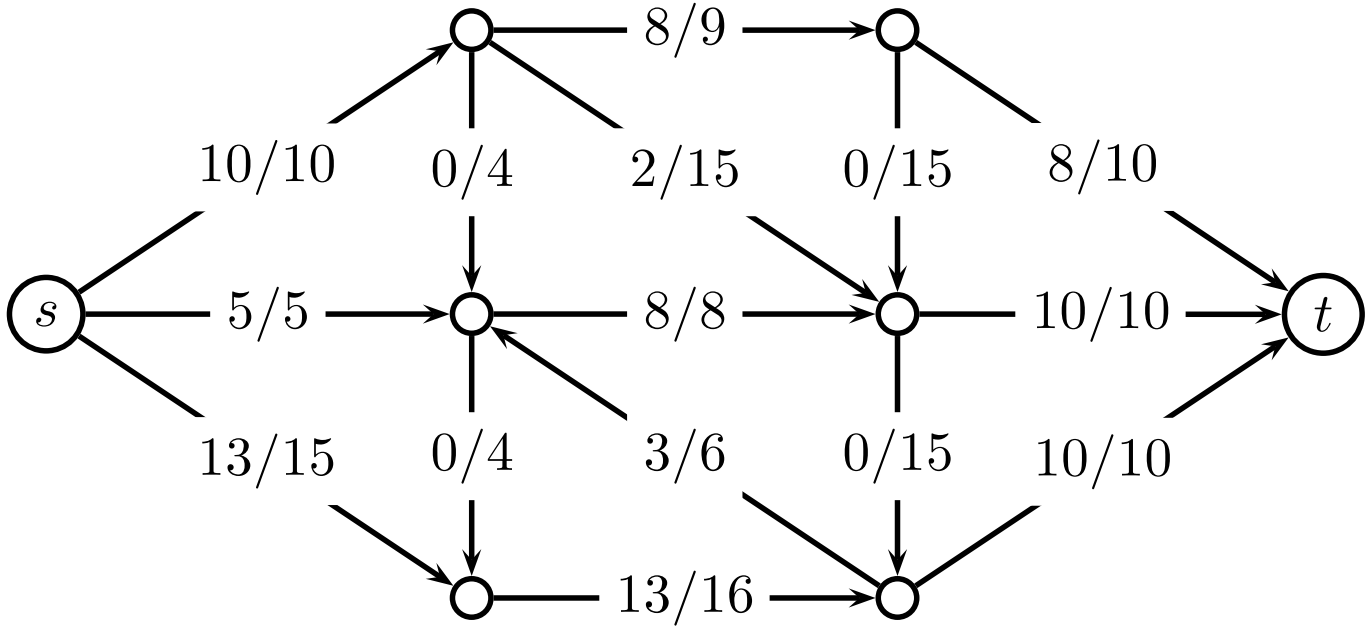
Example

The fourth augmenting path is calculated The bottleneck capacity is 3. There are no more augmenting paths. Done.



Example

The final flow graph can be obtained by "collapsing" the pairs of edges in the final residual graph.



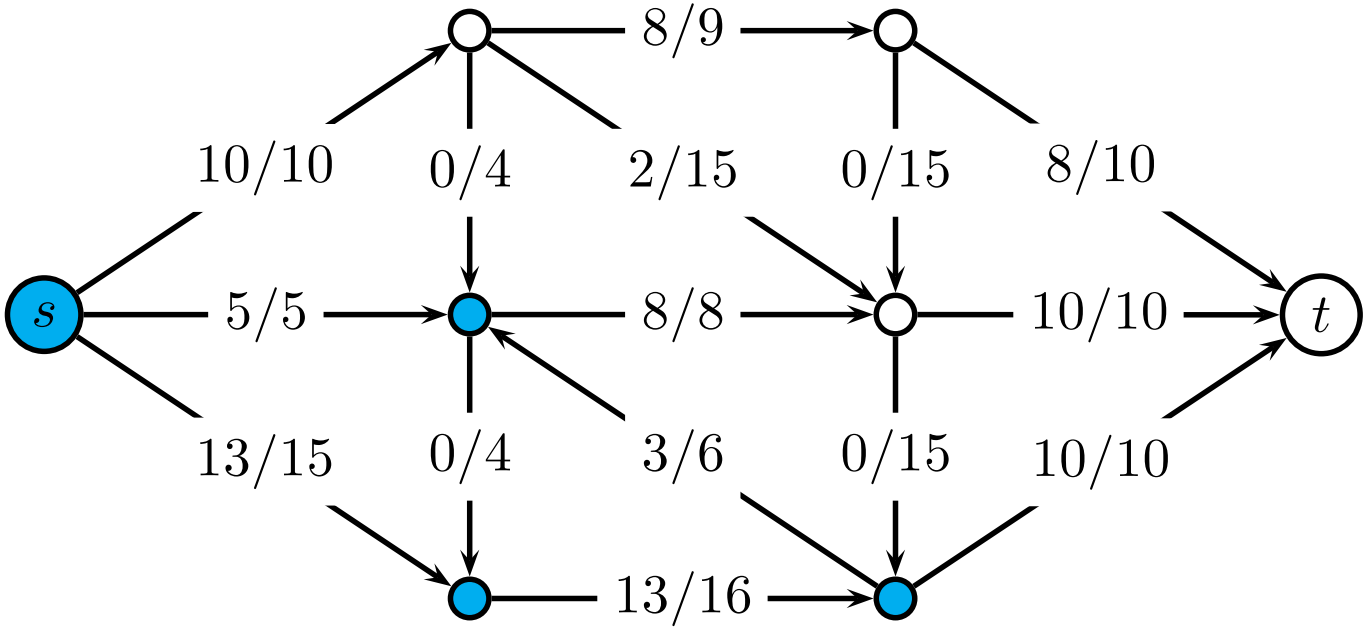
Calculating Augmenting Paths

- The augmenting paths are calculated on the residual graph!
- To update the residual graph, subtract the flow from the forward edge, add the flow to the backwards edge.
- We are done when there are no paths from s to t on the residual graph.
- The flow can be calculated as the flow out of s or into t.
- But how do we get the minimum cut?

Calculating The Min-Cut

- Once the max-flow is calculated, the min-cut can be extracted as follows:
- Subset A contains s and all the vertices accessible from it when the maximum flow is applied.
- That is, no full forward edges or empty backwards edges.
- Subset B contains the rest of the vertices (those not accessible by s).
- This is a legitimate cut, since t must be on side B (why?).

Calculating The Min-Cut



Max-flow Min-cut theorem

- **The augmenting paths theorem:** A flow f is a max flow iff there are no augmenting paths.
- **The Max-flow Min-cut theorem:** The value of the max flow = capacity of the min cut.
- Proof: We will show the the following three statements are equivalent for a flow f:
  1. There exists a cut whose capacity equals the value of the flow
  2. f is a max flow.
  3. There is no augmenting path with respect to f

Proof of Max-flow Min-cut theorem

$1 \rightarrow 2$:
- Suppose (A,B) is a cut with capacity equal to the value of f
- Then, the value of any flow $f' \leq capacity(A, B) = f$ (by weak duality above)
- Thus, f is a maxflow.

$2 \rightarrow 3$: (we will show contra-positive)
- Suppose that there is an augmenting path with respect to f
- We can improve f by sending flow along this path.
- Thus, f is not a maxflow.

$3 \rightarrow 1$:
- Let (A,B) be a cut where A is the set of vertices accessible by s.
- By definition of cut, s is in A
- Since no augmenting path, t is in B
- Capacity of cut = net flow across cut = value of flow f (because all forward edges full and all backward edges empty, hence cut at full capacity by flow.)

Analysis of Ford-Fulkerson's Algorithm

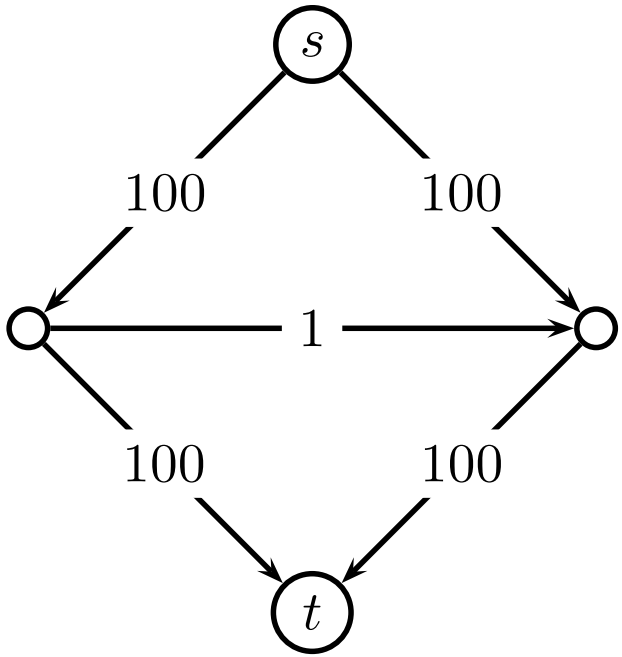
- Notice that the algorithm is not very specific about how to find the paths.
- Also, it doesn't say in what order to apply the paths.
- Does it even always terminate? And how fast?
- In practice, BFS or DFS is a good way to find paths.
- When capacities are integer, the algorithm is guaranteed to terminate.

Ford-Fulkerson's Algorithm with Integer Capacities

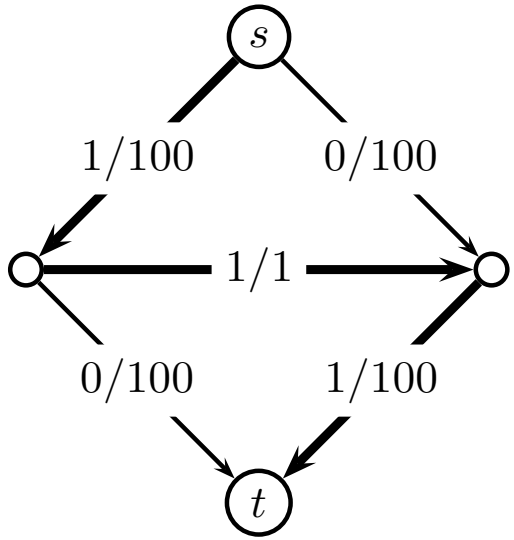
- Edge capacities are all integers.
- Hence, bottleneck capacities are integers too.
- Flow changes by an integer.
- Every augmentation increases the flow by at least 1.
- Hence, the number of augmentations is bound by the value of max-flow and the algorithm will terminate.
- From this we conclude that there is an integer-valued max flow, and it will be found by FF's algorithm.

A Bad Example

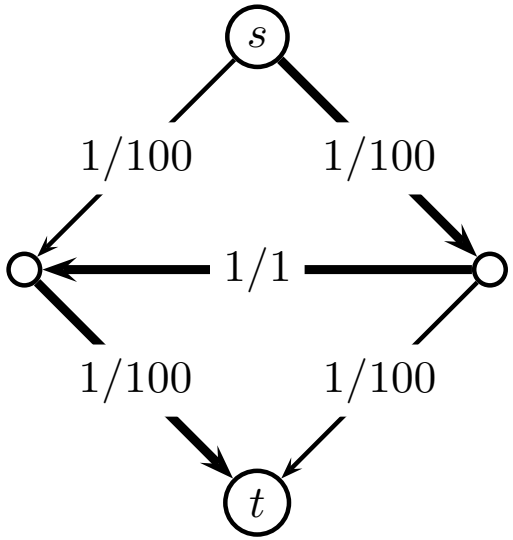
- The algorithm indeed terminates, but how long does it take?
- Some unfortunate choices can make the number of augmentations = max flow.
- See example below:



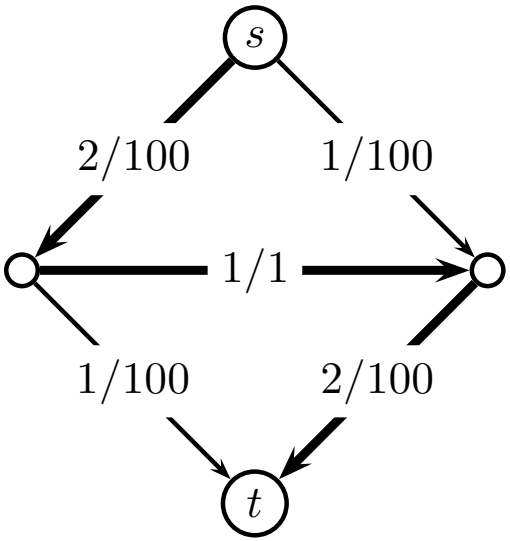
A Bad Example



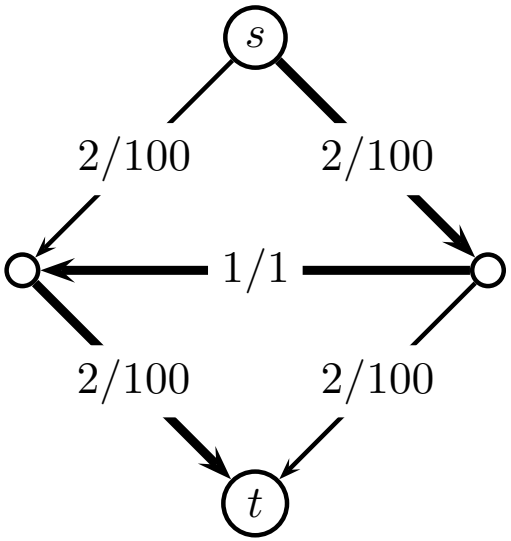
1



2



3



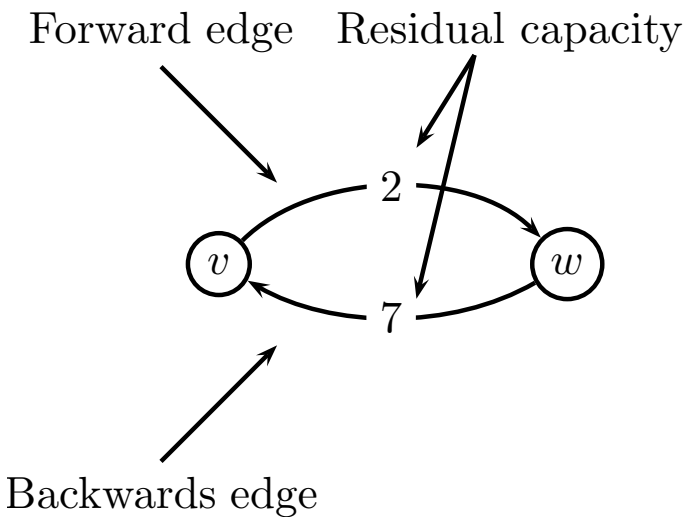
4

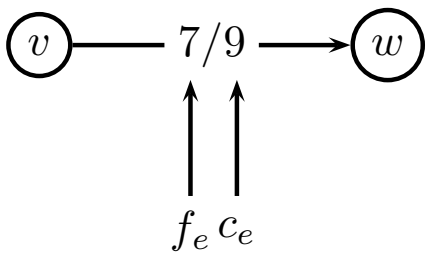
A Bad Example

- The max flow is 200.
- A not-very-smart choice of paths will make the algorithm run 200 times...
- A smart choice – only 2 iterations.
- How do you choose augmenting paths?
- There is no clear-cut way that always works.
- Some heuristics: Shortest paths (BFS, for minimum number of edges), fattest paths (Priority queue, max. bottleneck capacity), DFS, etc.

Implementation – Flow Edge

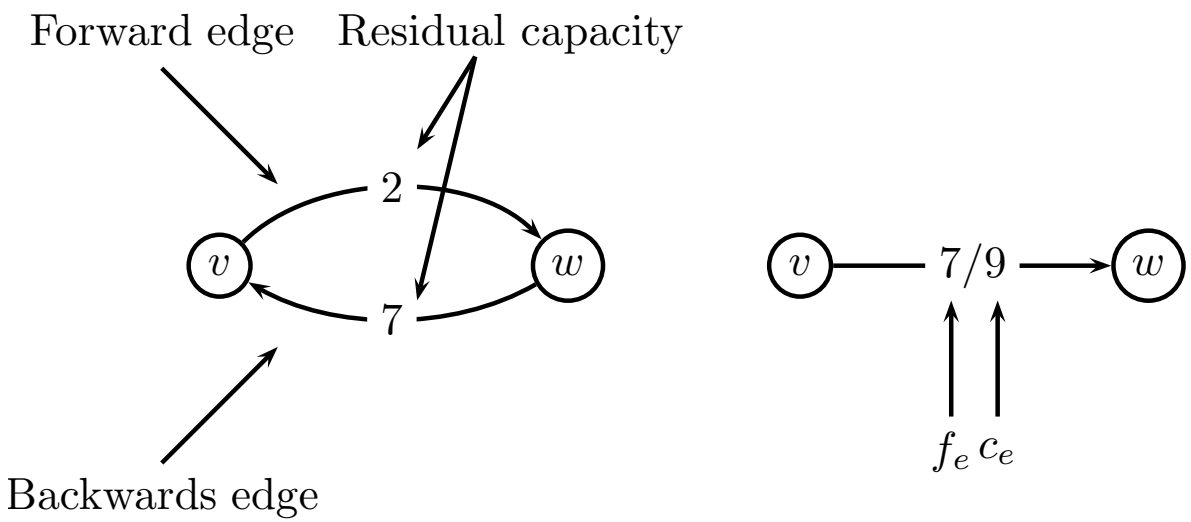
- FlowEdge associates a flow f_e and capacity c_e with edge $e = v \rightarrow w$
- Must be able to process edge e in either direction: include e in adjacency lists of both v and w.
- Forward edge: residual capacity $= c_e - f_e$
- Backward edge: residual capacity $= f_e$
- Augment Flow: Add Δ to Forward edge.
- Subtract Δ from Backward edge.





Implementation – Flow Edge

```
public class FlowEdge
----------------------
FlowEdge(int v, int w, double capacity) //create a flow edge v-w
int from() // vertex this edge points from
int to() // vertex this edge points to
int other(int v) // other endpoint
double capacity() // capacity of this edge
double flow() // flow in this edge
double residualCapacityTo(int v) // residual capacity toward v
void addResidualFlowTo(int v, double delta) // add delta flow
    toward v
```



Implementation – Flow Edge

```
public class FlowEdge
{
   private final int v, w; // from and to
   private final double capacity;
   private double flow;

   public FlowEdge(int v, int w, double capacity) {
      this.v = v;
      this.w = w;
      this.capacity = capacity;
   }
   public int from()       { return v;}
   public int to()         { return w;}
   public double capacity() { return capacity;}
   public double flow()    { return flow;}
   public int other(int vertex) {
      if (vertex == v) return w;
      else if (vertex == w) return v;
      else throw new IllegalArgumentException();
   } // TBC Next slide
```

Implementation – Change Flow

```
public double residualCapacityTo(int vertex)
{
   if (vertex == v) return flow; // Forward edge
   else if (vertex == w)
      return capacity - flow; // Backward edge
   else throw new IllegalArgumentException();
}

public void addResidualFlowTo(int vertex, double delta)
{
   if (vertex == v) flow -= delta; // Forward edge
   else if (vertex == w)
      flow += delta; // Backward edge
   else throw new IllegalArgumentException();
}
```

Implementation – Flow Network

```
public class FlowNetwork
------------------------
// create an empty flow network with V vertices
FlowNetwork(int V)
FlowNetwork(In in) // construct flow network input stream
void addEdge(FlowEdge e) // add flow edge e to this flow network
// forward and backward edges incident to v
Iterable<FlowEdge> adj(int v)
Iterable<FlowEdge> edges() // all edges in this flow network
int V() //number of vertices
int E() // number of edges
String toString() // String representation
```

Implementation – Flow Network

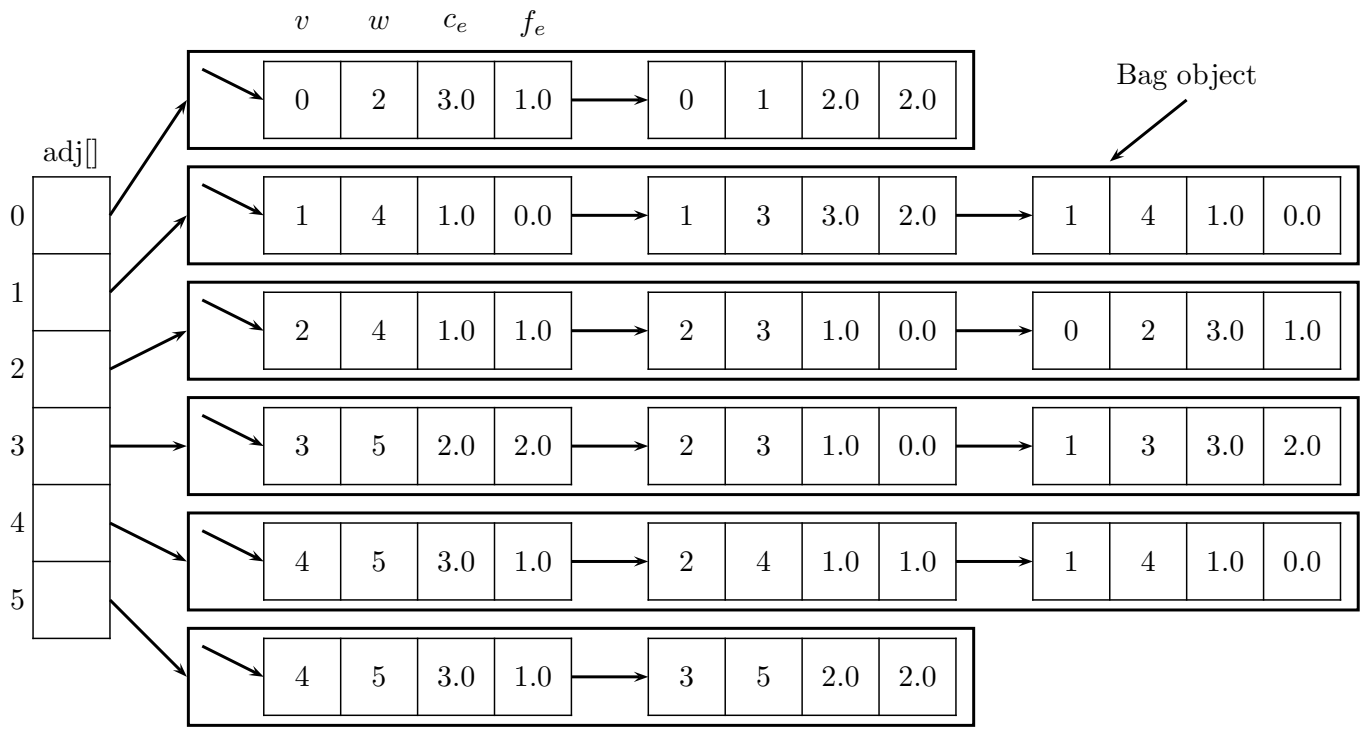
```
public class FlowNetwork
{
   private final int V;
   private Bag<FlowEdge>[] adj;
   public FlowNetwork(int V) {
      this.V = V;
      adj = (Bag<FlowEdge>[]) new Bag[V];
      for (int v = 0; v < V; v++)
      adj[v] = new Bag<FlowEdge>();
   }
   public void addEdge(FlowEdge e) {
      int v = e.from();
      int w = e.to();
      adj[v].add(e);
      adj[w].add(e);
   }
   public Iterable<FlowEdge> adj(int v)
   {  return adj[v]; }
}
```

Adjacency List Representation

- Note: Adjacency list includes edges with 0 residual capacity. (residual network is represented implicitly)
- Every edge is referenced twice.

tinyFN.txt

```
6
8
0   1   2.0
0   2   3.0
1   3   3.0
1   4   1.0
2   3   1.0
2   4   1.0
3   5   2.0
4   5   3.0
```



Ford-Fulkerson

```
public class FordFulkerson
{
   private boolean[] marked; // true if s->v path in res. network
   private FlowEdge[] edgeTo; // last edge on s->v path
   private double value;  // value of flow

   public FordFulkerson(FlowNetwork G, int s, int t) {
      value = 0.0;
      while (hasAugmentingPath(G, s, t)) {
         double bottle = Double.POSITIVE_INFINITY;
         for (int v = t; v != s; v = edgeTo[v].other(v))
            bottle = Math.min(bottle,
                edgeTo[v].residualCapacityTo(v));
         for (int v = t; v != s; v = edgeTo[v].other(v))
            edgeTo[v].addResidualFlowTo(v, bottle);
         value += bottle;
      }
   }
   public boolean inCut(int v) { return marked[v];}
   // TBC next slide
}
```

Find Augmenting Path via BFS

```
private boolean hasAugmentingPath(FlowNetwork G, int s, int t) {
   edgeTo = new FlowEdge[G.V()];
   marked = new boolean[G.V()];
   Queue<Integer> queue = new Queue<Integer>();
   queue.enqueue(s);
   marked[s] = true;
   while (!queue.isEmpty()) {
      int v = queue.dequeue();
      for (FlowEdge e : G.adj(v)) {
         int w = e.other(v);
         if (!marked[w] && (e.residualCapacityTo(w) > 0)) {
            edgeTo[w] = e;
            marked[w] = true;
            queue.enqueue(w);
         }
      }
   }
   return marked[t]; // is t reachable from s in residual
       network?
}
```

Applications – Bipartite Matching

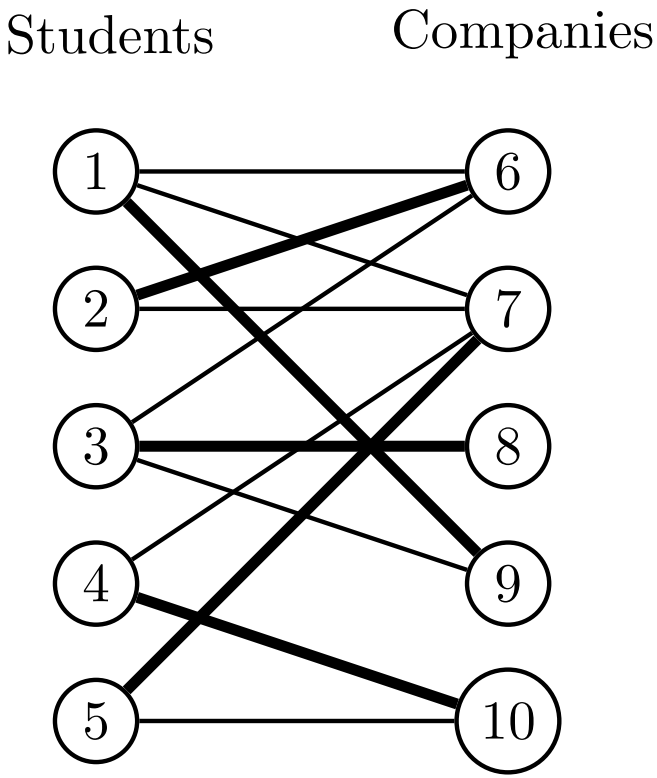
- Problem: N people apply to N jobs, each person may get several offers.
- Question: Is there a way to perfectly match people and jobs?

1 Alice		6 Adobe	
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2 Bob		7 Amazon	
	Adobe		Alice
	Amazon		Bob
3 Carol			Dave
	Adobe		Eliza
	Facebook	8 Facebook	
	Google		Carol
4 Dave		9 Google	
	Amazon		Alice
	Yahoo		Carol
5 Eliza		10 Yahoo	
	Amazon		Dave
	Yahoo		Eliza

Applications – Bipartite Matching

Perfect matching

Alice	–	Google
Bob	–	Adobe
Carol	–	Facebook
Dave	–	Yahoo
Eliza	–	Amazon



1 Alice
- Adobe
- Amazon
- Google

2 Bob
- Adobe
- Amazon

3 Carol
- Adobe
- Facebook
- Google

4 Dave
- Amazon
- Yahoo

5 Eliza
- Amazon
- Yahoo

6 Adobe
- Alice
- Bob
- Carol

7 Amazon
- Alice
- Bob
- Dave
- Eliza

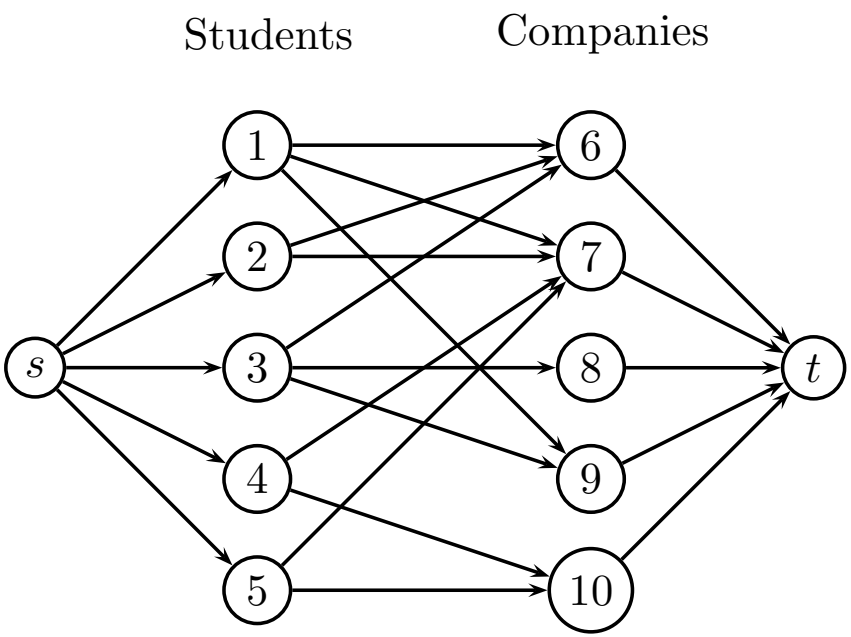
8 Facebook
- Carol

9 Google
- Alice
- Carol

10 Yahoo
- Dave
- Eliza

Network Flow Formulation of Bipartite Matching

- Create s, t, one vertex for each student, and one vertex for each job.
- Add edge from s to each student (capacity 1).
- Add edge from each job to t (capacity 1).
- Add edge from student to each job offered (infinite capacity).

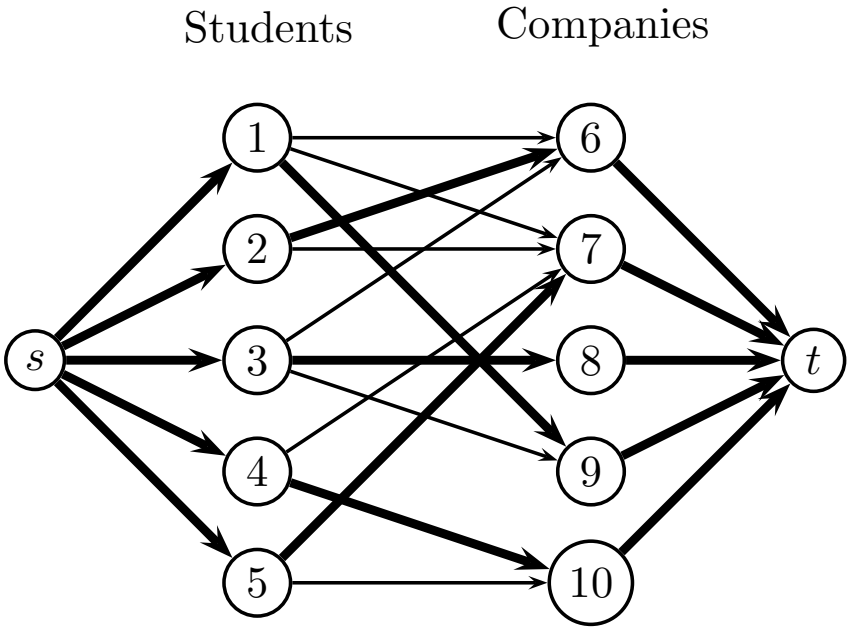


1 Alice		6 Adobe	
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2 Bob		7 Amazon	
	Adobe		Alice
	Amazon		Bob
3 Carol			Dave
	Adobe		Eliza
	Facebook	8 Facebook	
	Google		Carol
4 Dave		9 Google	
	Amazon		Alice
	Yahoo		Carol
5 Eliza		10 Yahoo	
	Amazon		Dave
	Yahoo		Eliza

Network Flow Formulation of Bipartite Matching

1-1 correspondence between perfect matchings in bipartite graph and integer-valued maxflows of value N

Flow graph

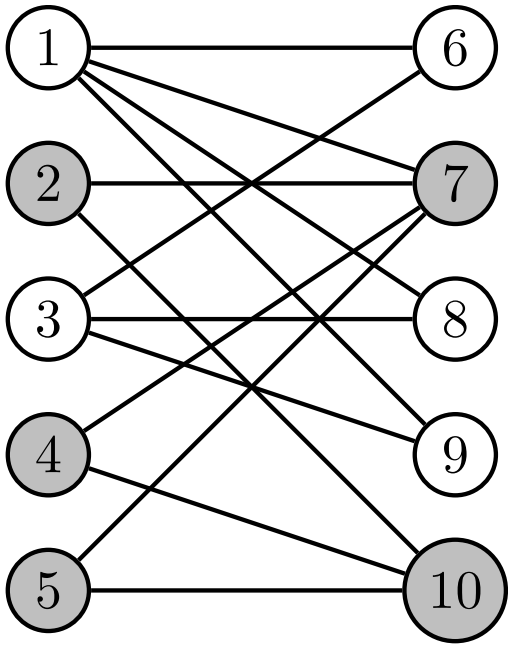


Bipartite matching problem

1 Alice		6 Adobe	
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2 Bob		7 Amazon	
	Adobe		Alice
	Amazon		Bob
3 Carol			Dave
	Adobe		Eliza
	Facebook	8 Facebook	
	Google		Carol
4 Dave		9 Google	
	Amazon		Alice
	Yahoo		Carol
5 Eliza		10 Yahoo	
	Amazon		Dave
	Yahoo		Eliza

What the mincut Tells Us

- **Goal:** When no perfect matching, explain why.
- $S = \{2, 4, 5\}$ and $T = \{7, 10\}$.
- Student in S can be matched only to companies in T and $|S| > |T|$
- No perfect matching exists.



What the mincut Tells Us

- Consider mincut (A, B).
- Let S = students on s side of cut.
- Let T = companies on s side of cut.
- **Fact:** $|S| > |T|$; students in S can be matched only to companies in T.
- **Bottom line:** When no perfect matching, mincut explains why