CS 161: Design and Analysis of Algorithms

Linear Programming I: Maximum Flow

- Definition
- Algorithm
- Max Flow/Min Cut
- Linear Programming

Flows in Graphs

- Given a weighted graph G=(V,E), two nodes s and t
  - Weights represent capacities
  - s represents the source, t represents the target
- A **flow** is a setting of variables f_e for all edges e in E such that
  - $0 \leq f_e \leq w(e)$
  - For any node n other than s or t,

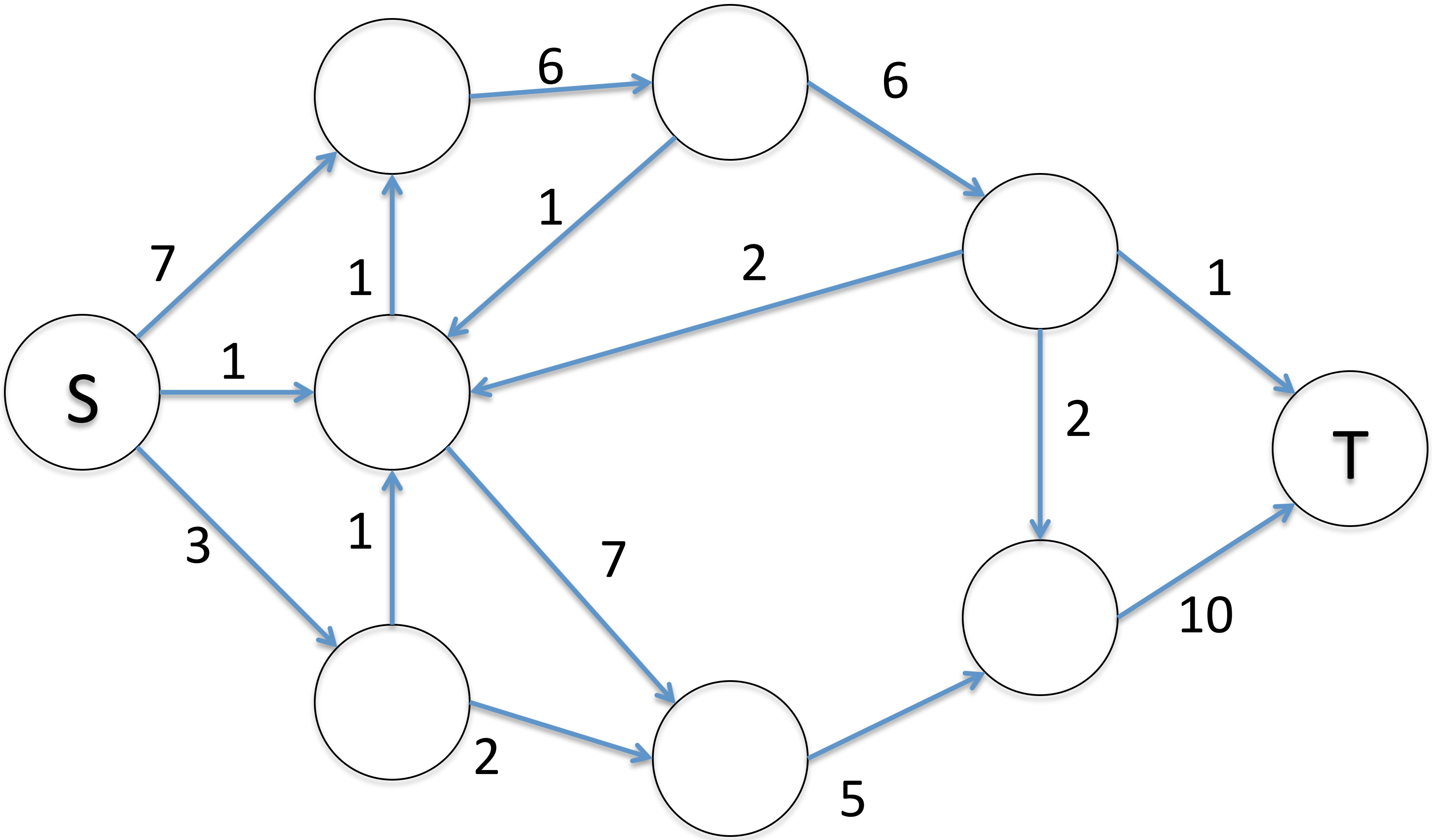
$$\sum_{(u,v)\in E} f_{(u,v)} = \sum_{(v,w)\in E} f_{(v,w)}$$

Maximum Flow

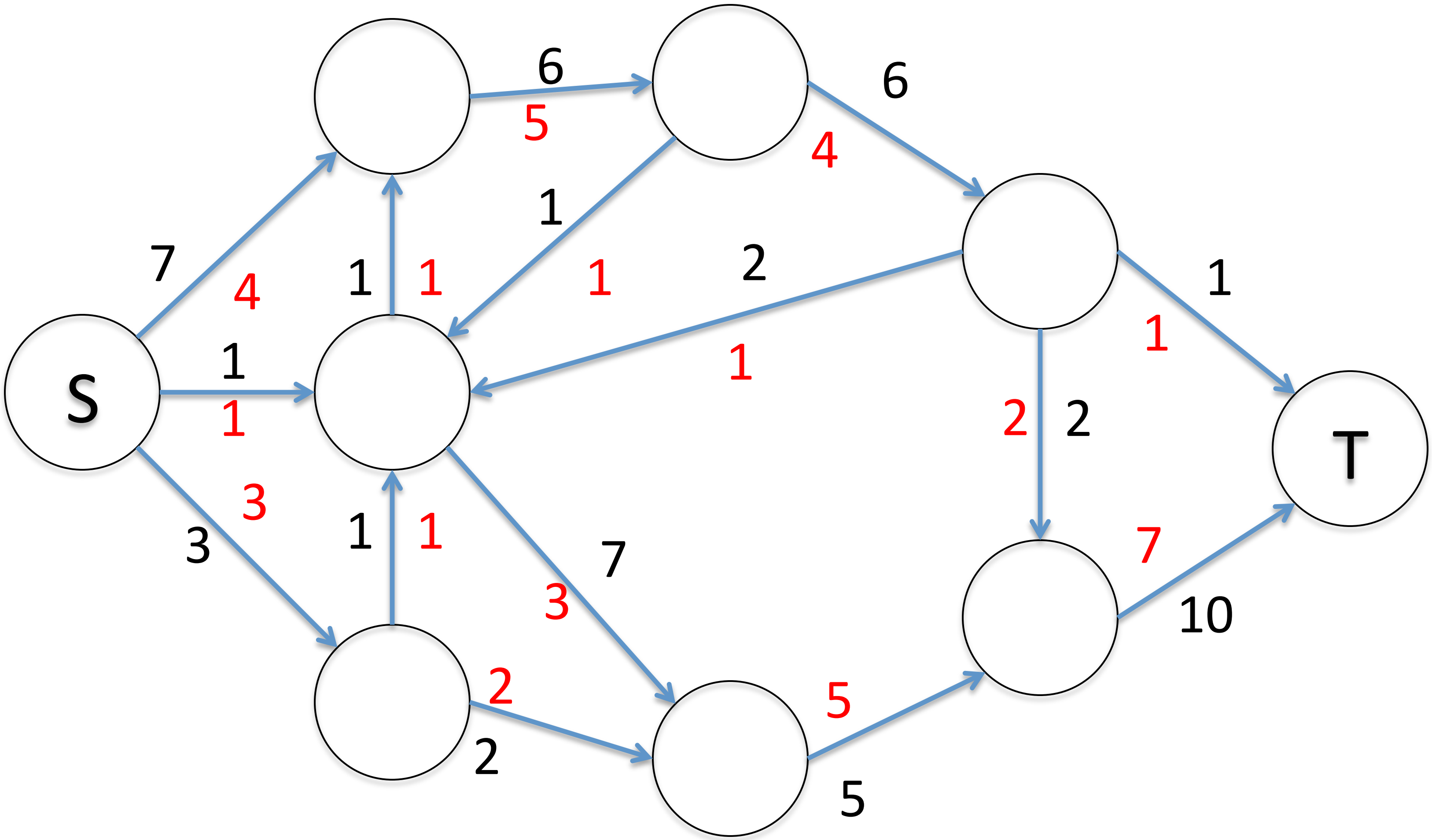
- A **maximum flow** is a flow that maximizes the amount leaving s (or entering t). That is,

$$\sum_{(s,v)} f_{(s,v)} - \sum_{(v,s)} f_{(v,s)}$$

Maximum Flow



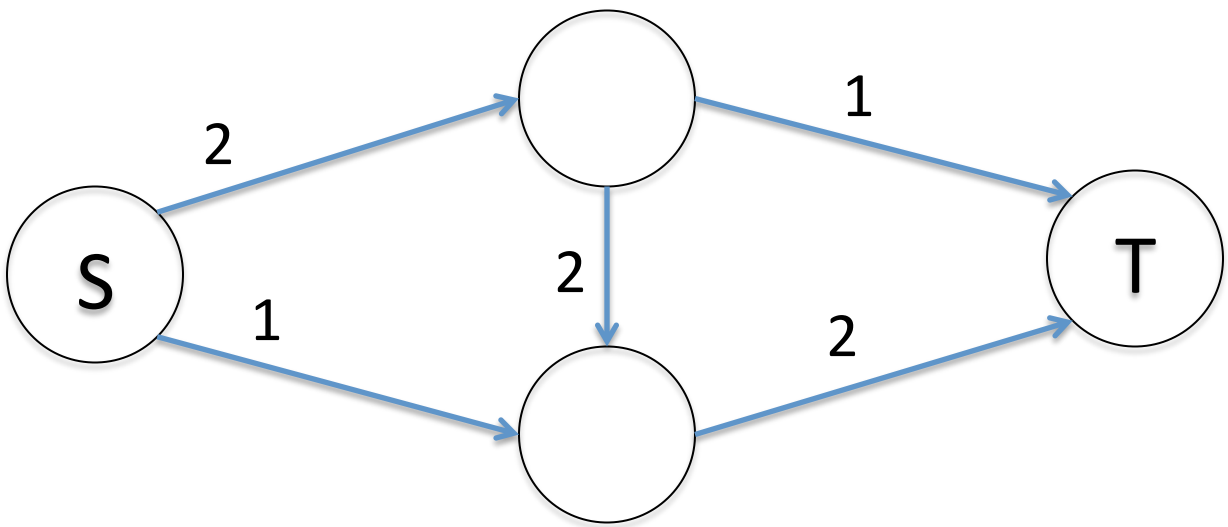
Maximum Flow



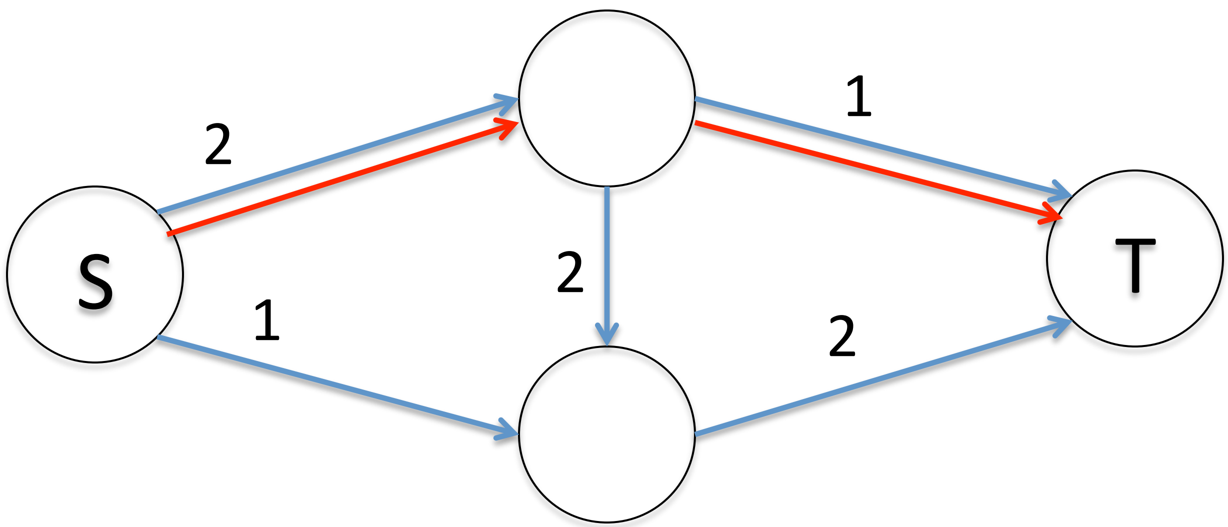
How to Compute Maximum Flow

- How can we compute **any** flow?
  - Find path in graph from s to t
  - Put 1 unit of flow along each edge in graph (or better yet, maximum possible)
- Given a flow, how can we compute a better flow?
  - Compute **residual capacities**, the remaining capacity of each edge
  - Compute flow in using residual capacities

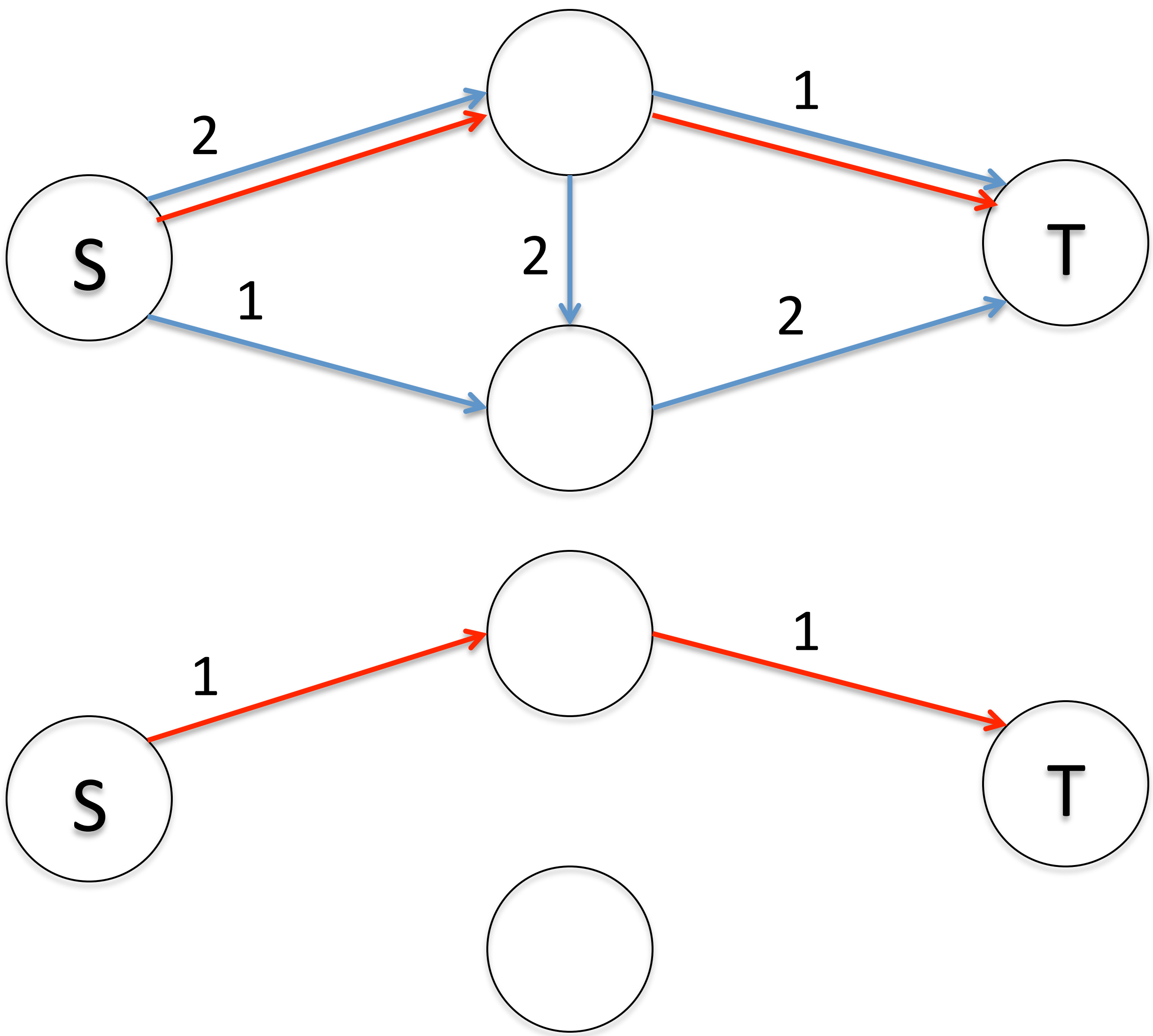
Computing Maximum Flow



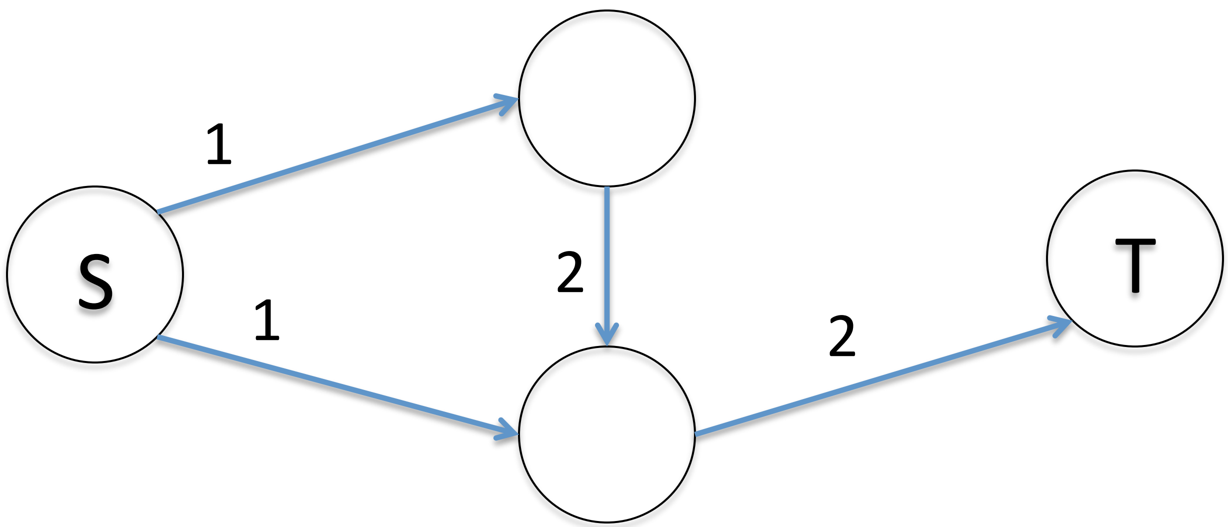
Computing Maximum Flow

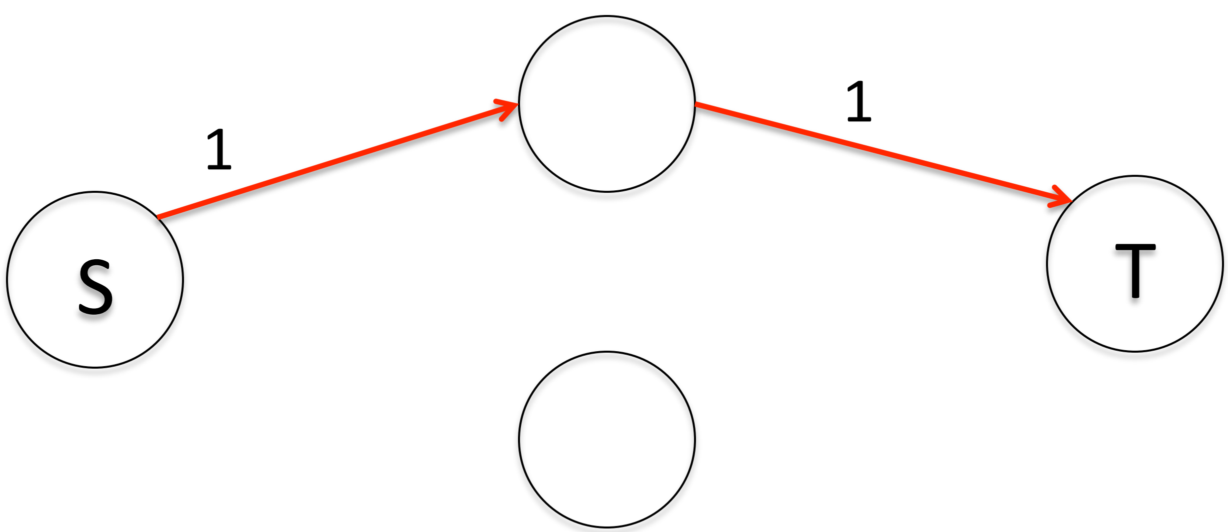


Computing Maximum Flow

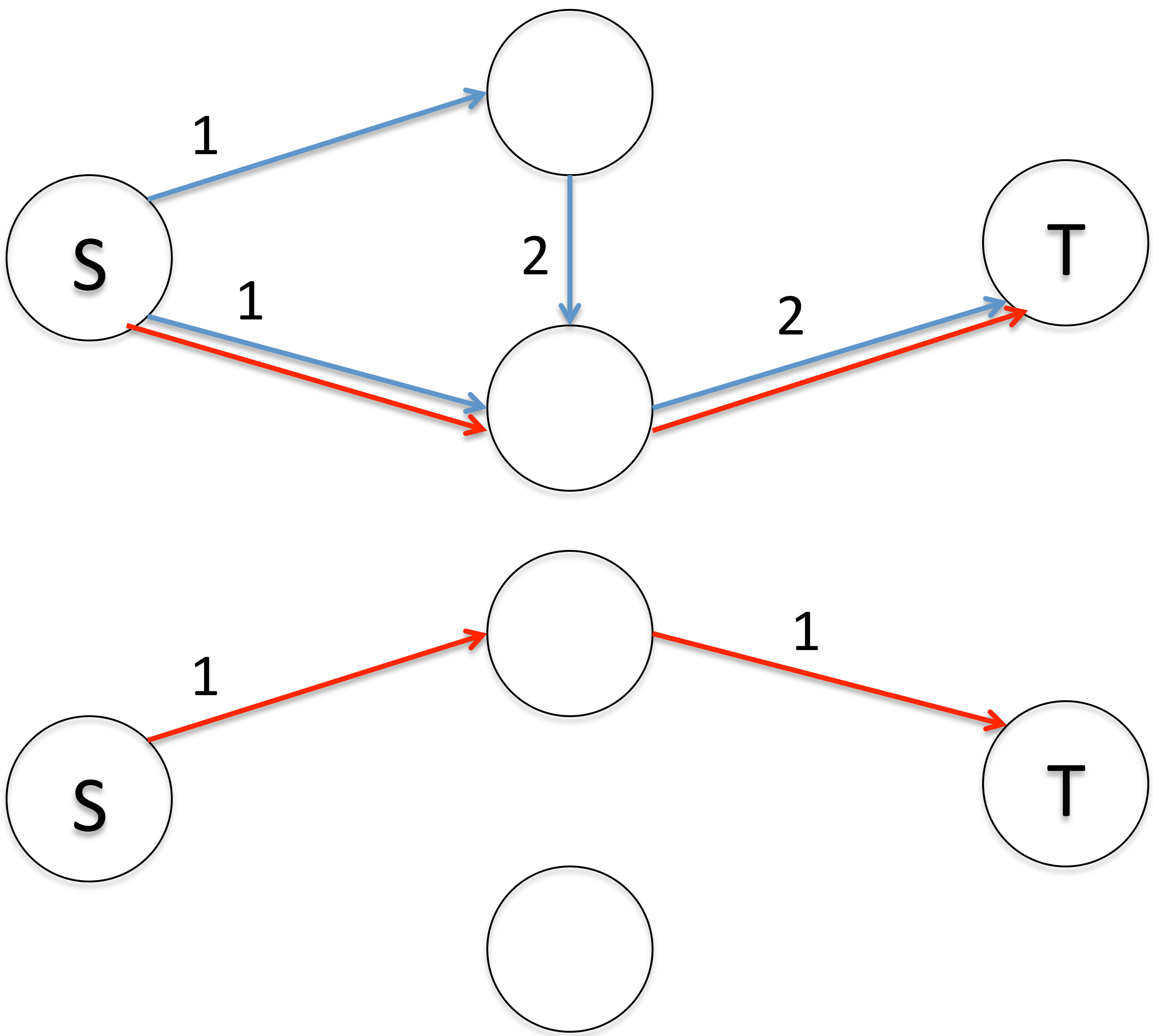


Computing Maximum Flow

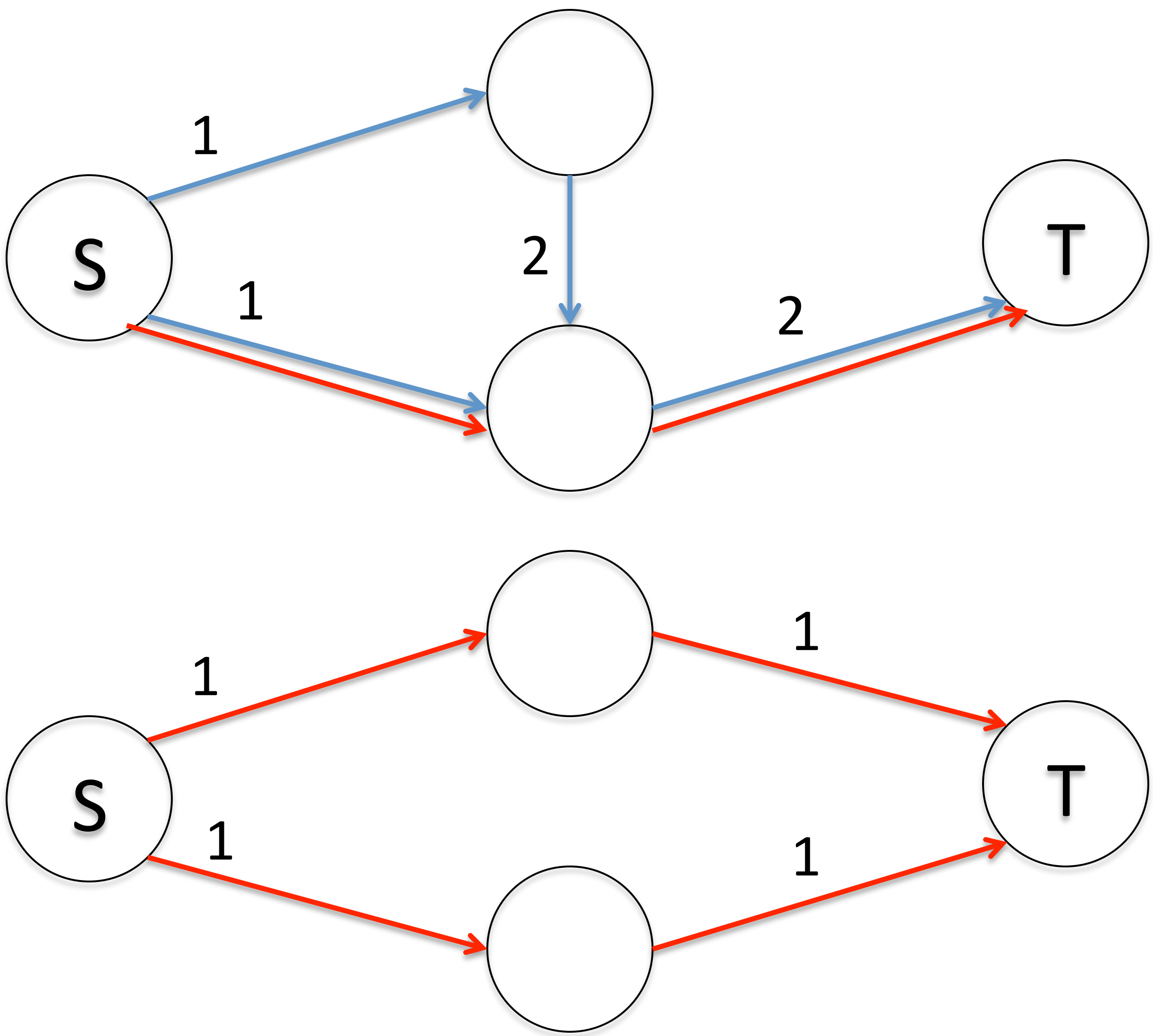




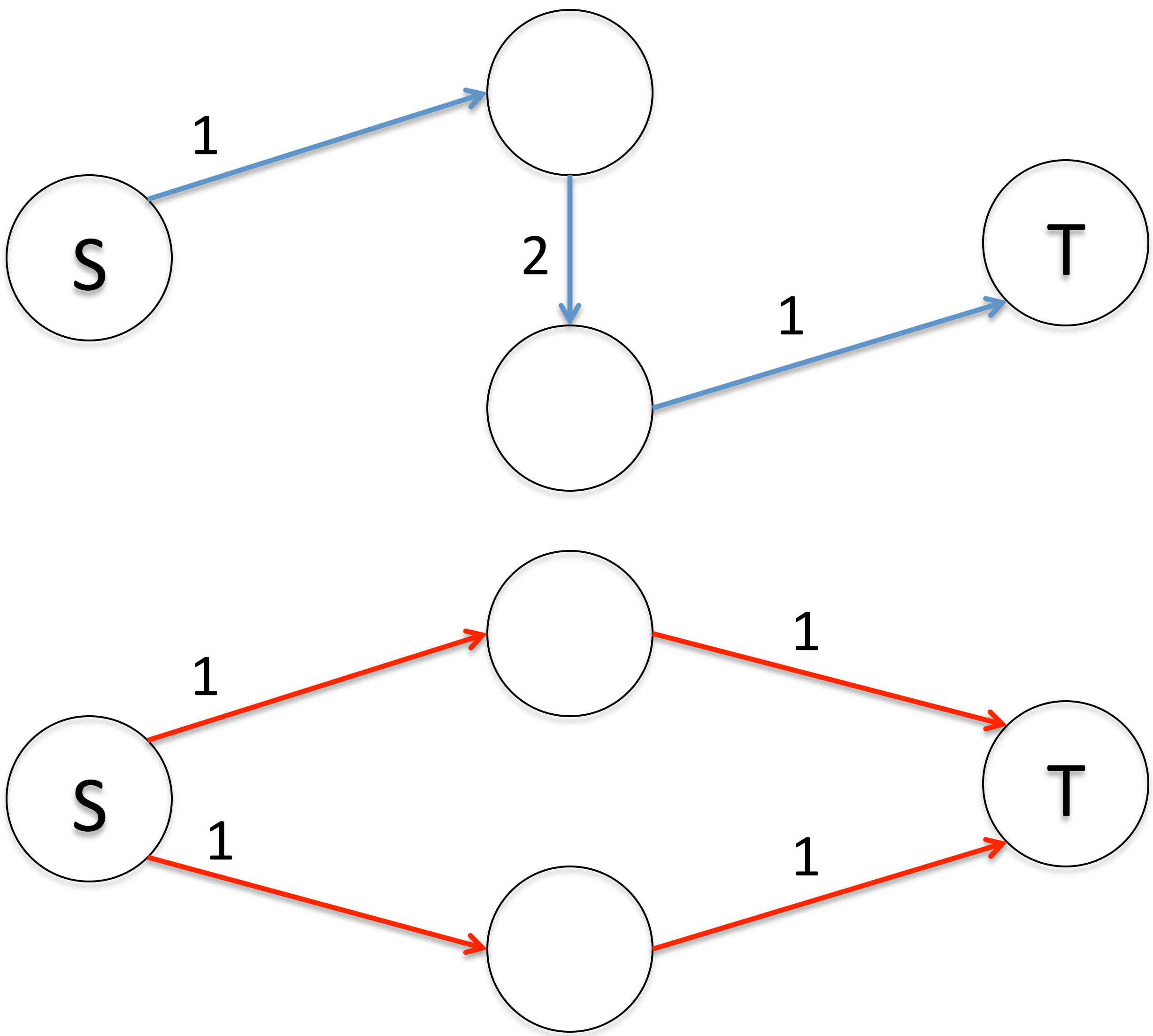
Computing Maximum Flow



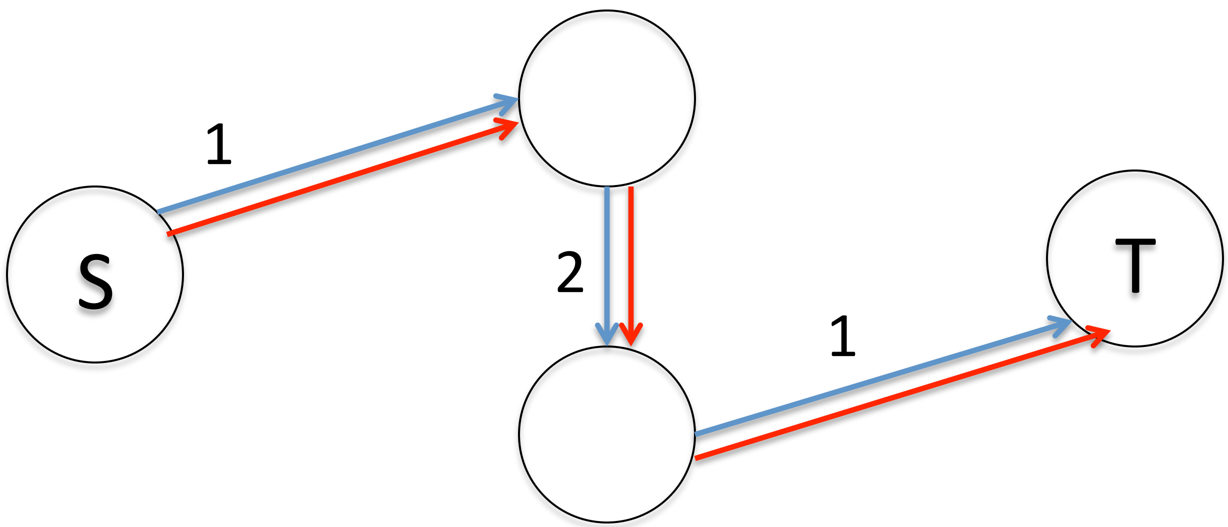
Computing Maximum Flow

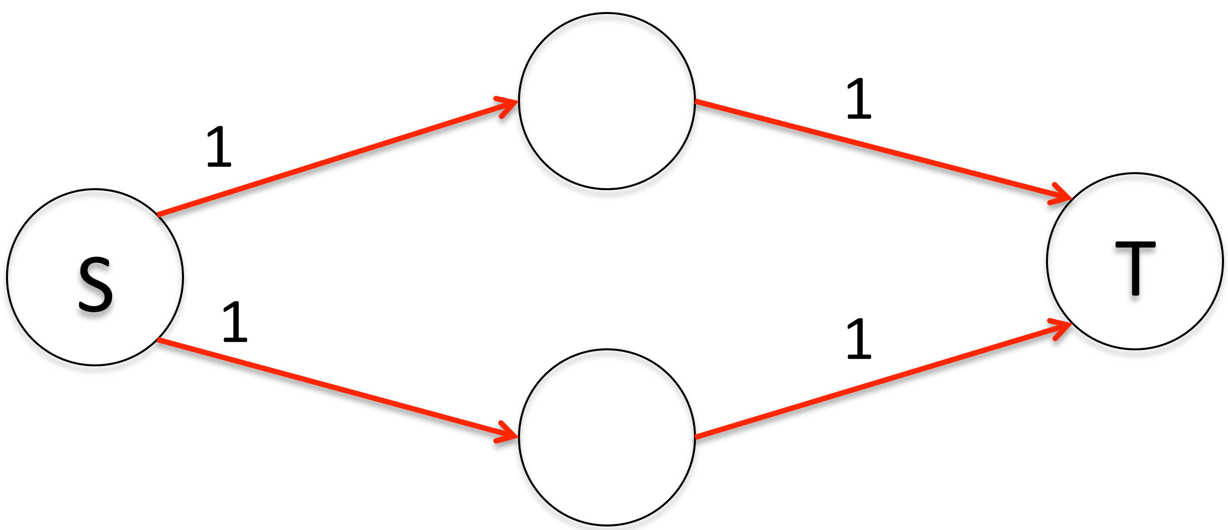


Computing Maximum Flow

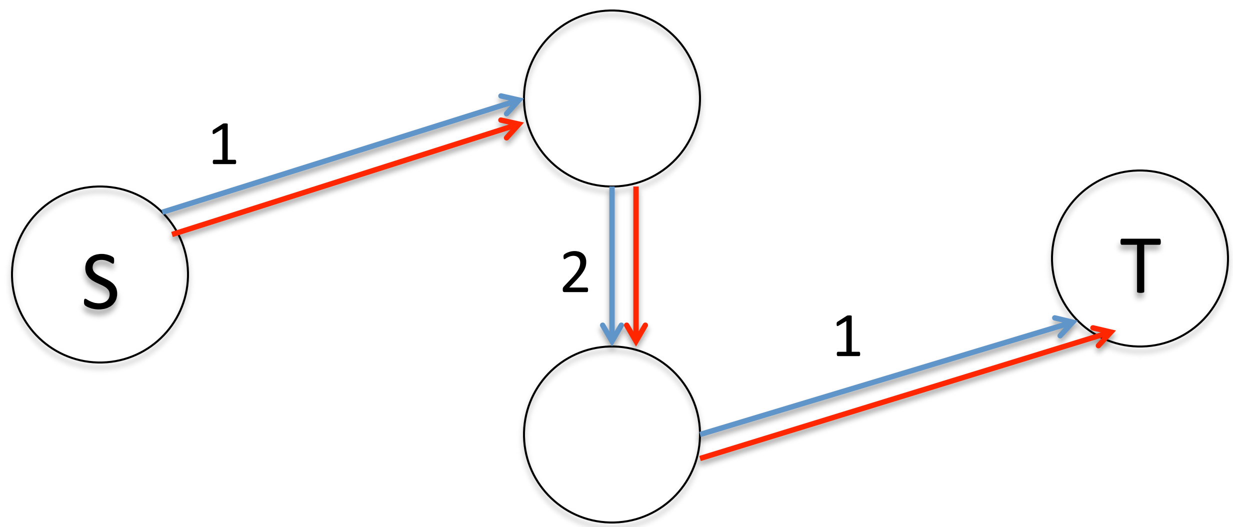


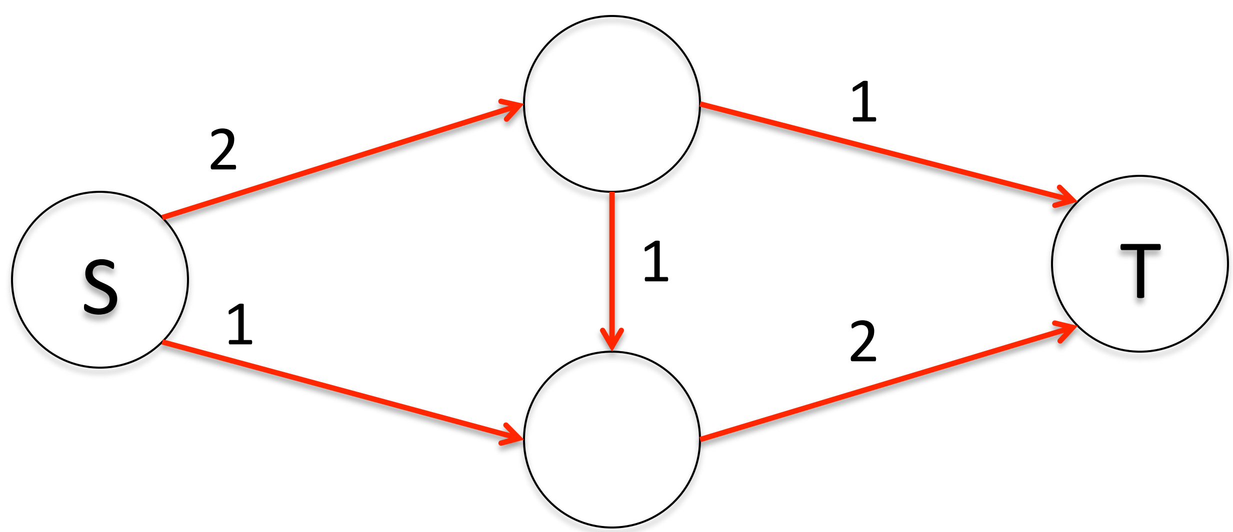
Computing Maximum Flow



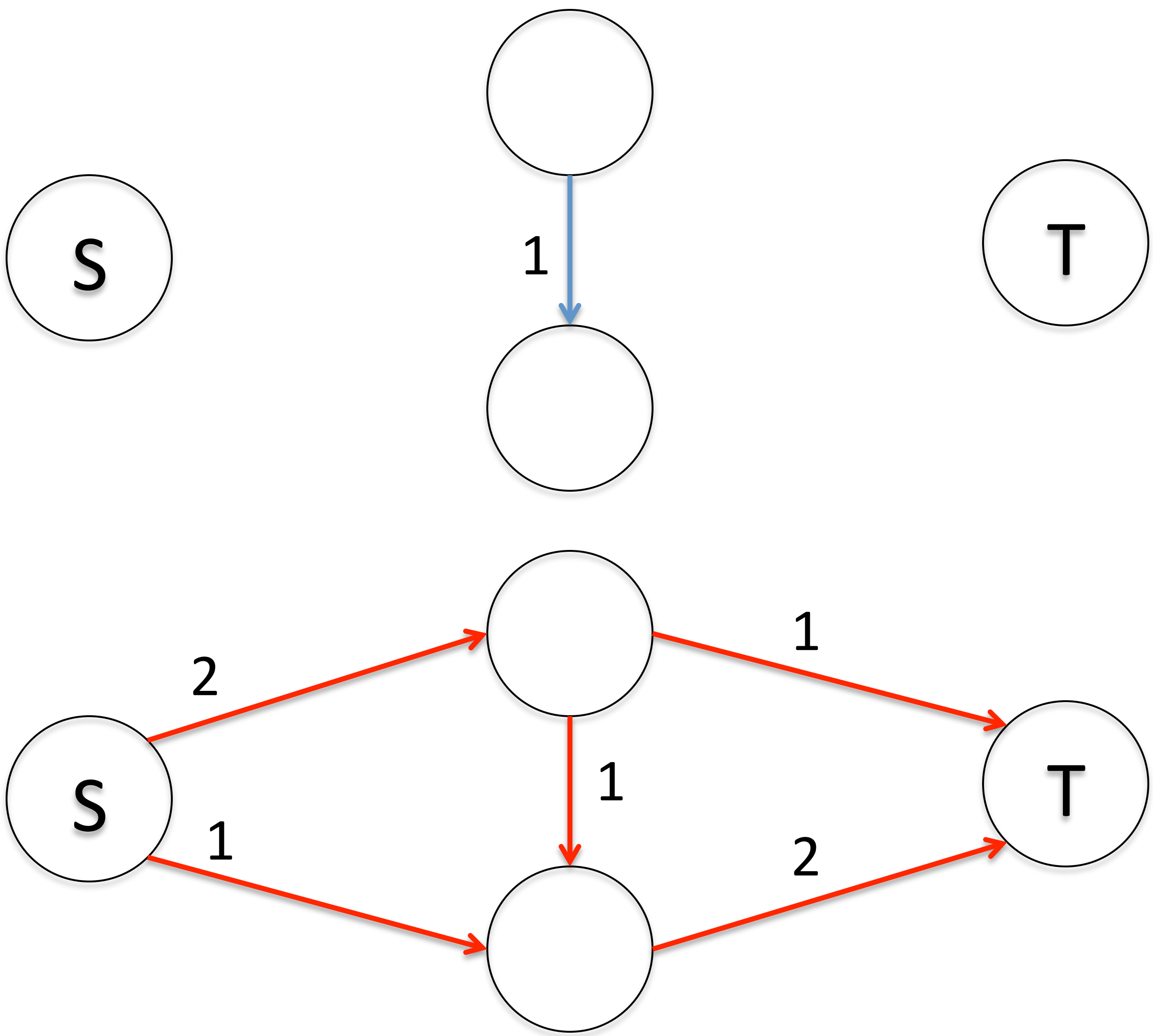


Computing Maximum Flow

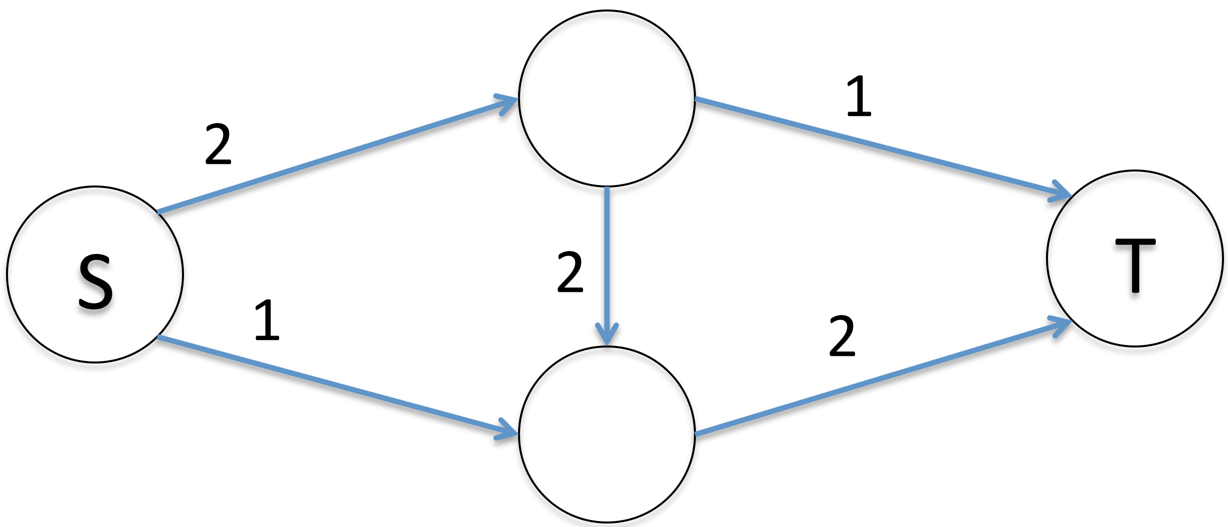




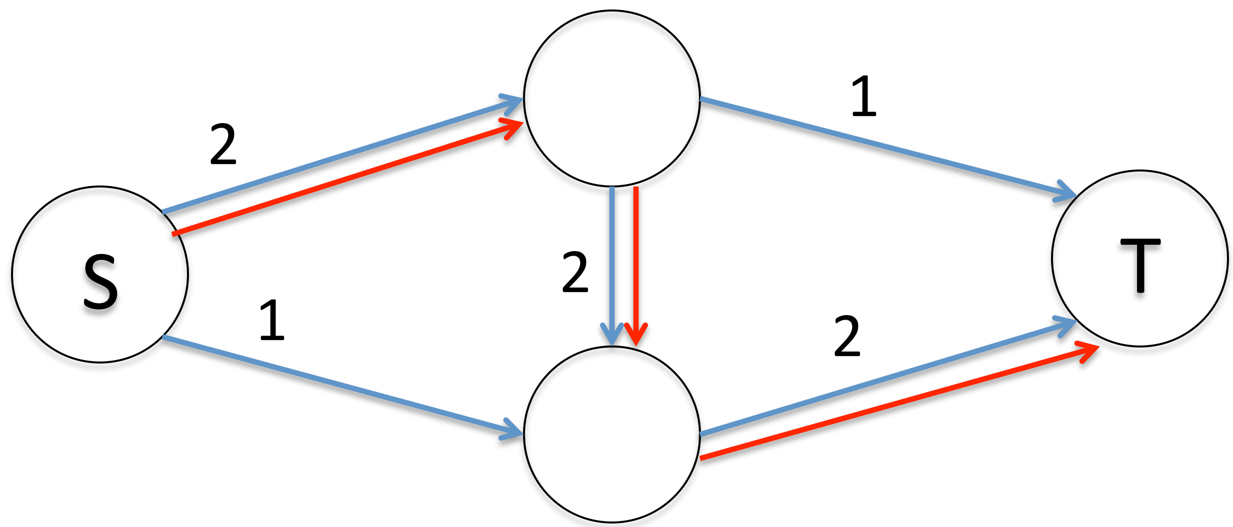
Computing Maximum Flow

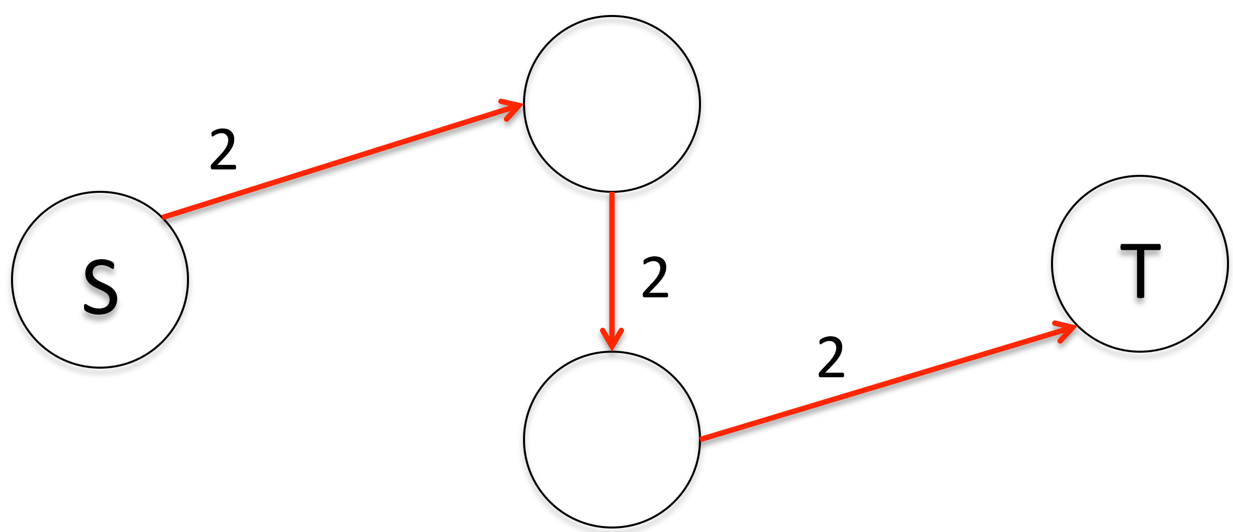


Problem!

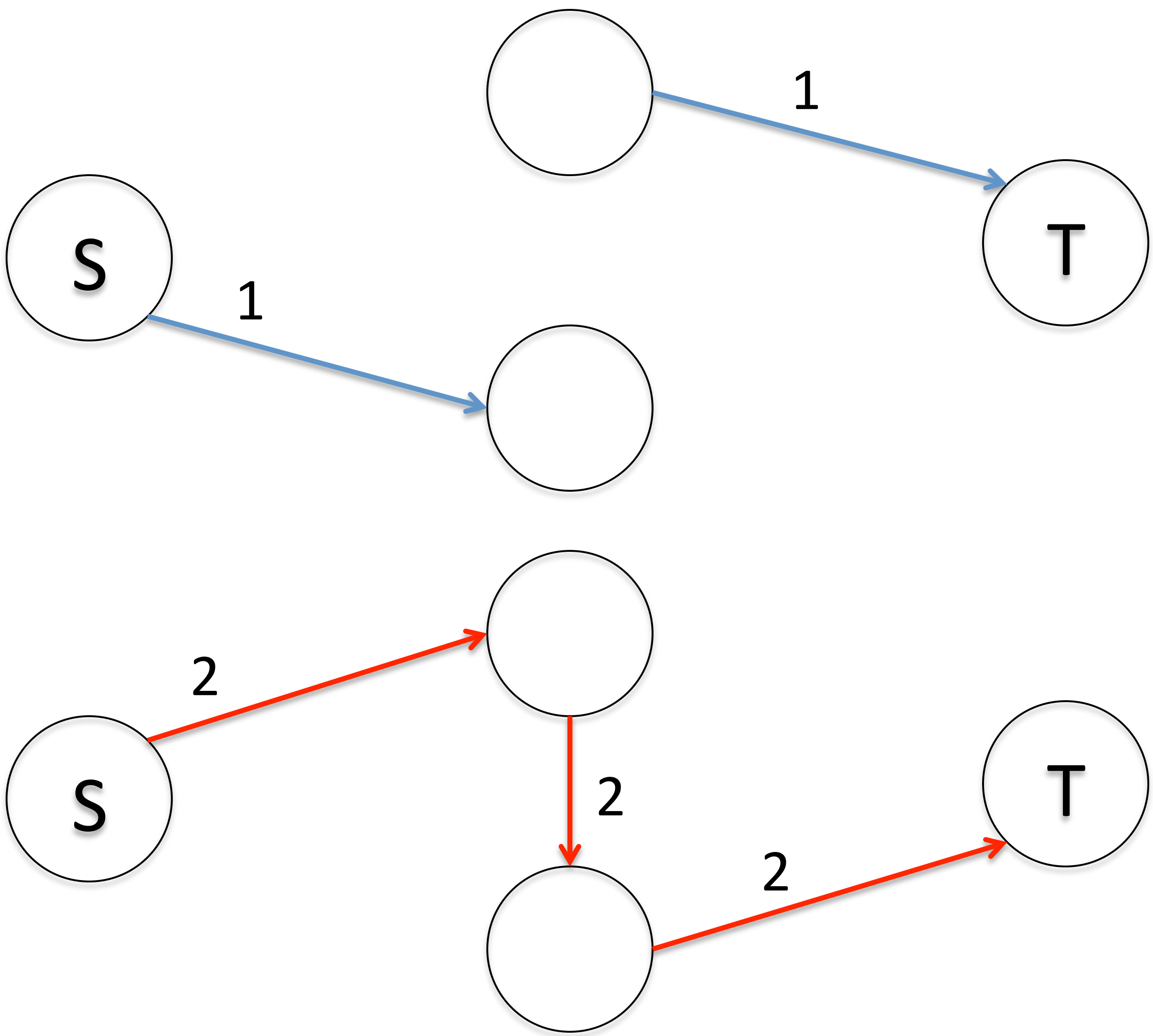


Problem!





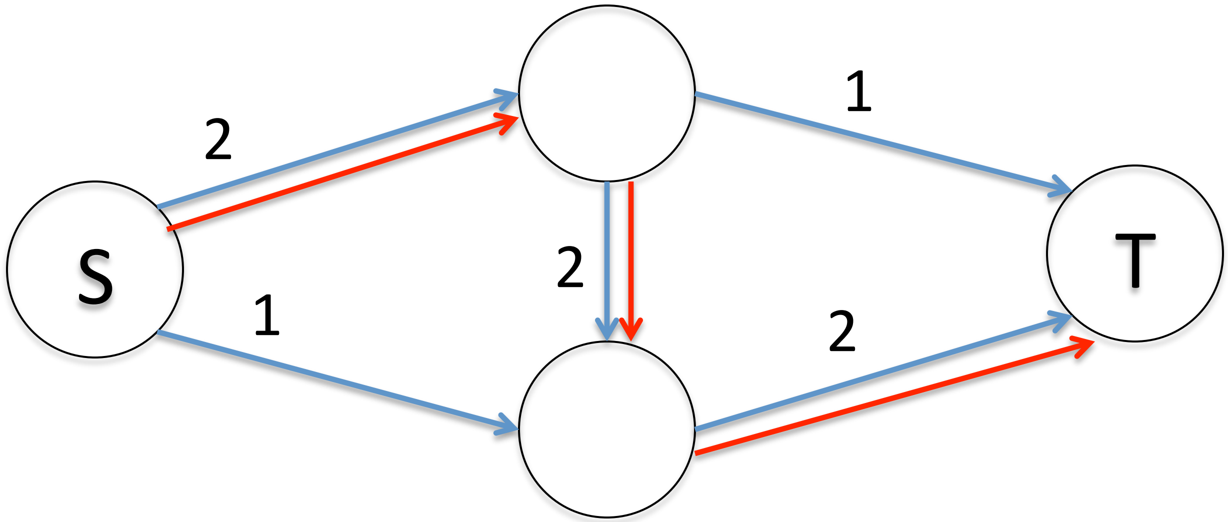
Problem!

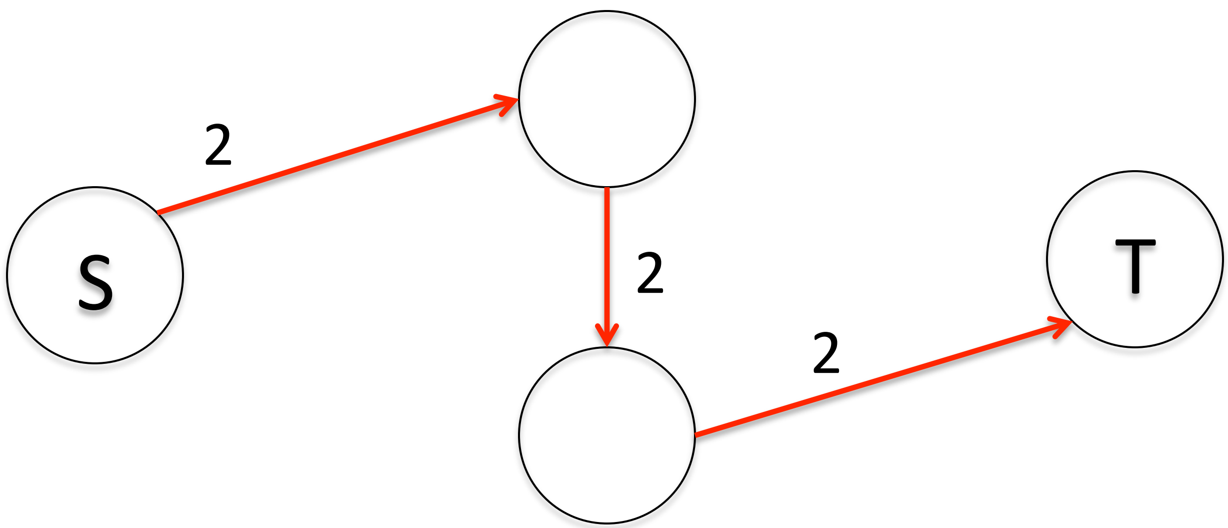


Problem!

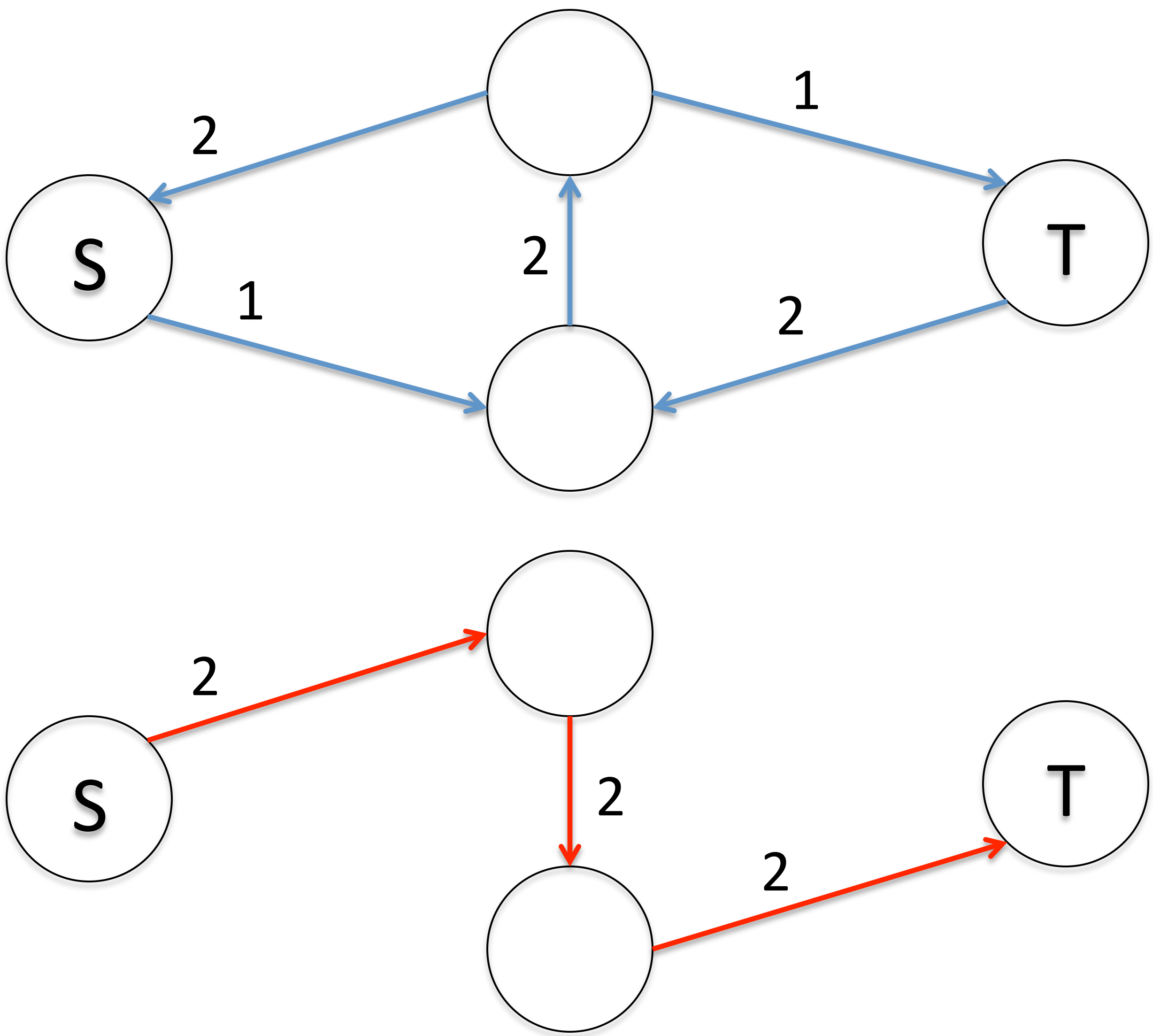
- Choosing a bad path can result in the wrong answer
- Solution: allow flows to cancel

Cancelling Flows

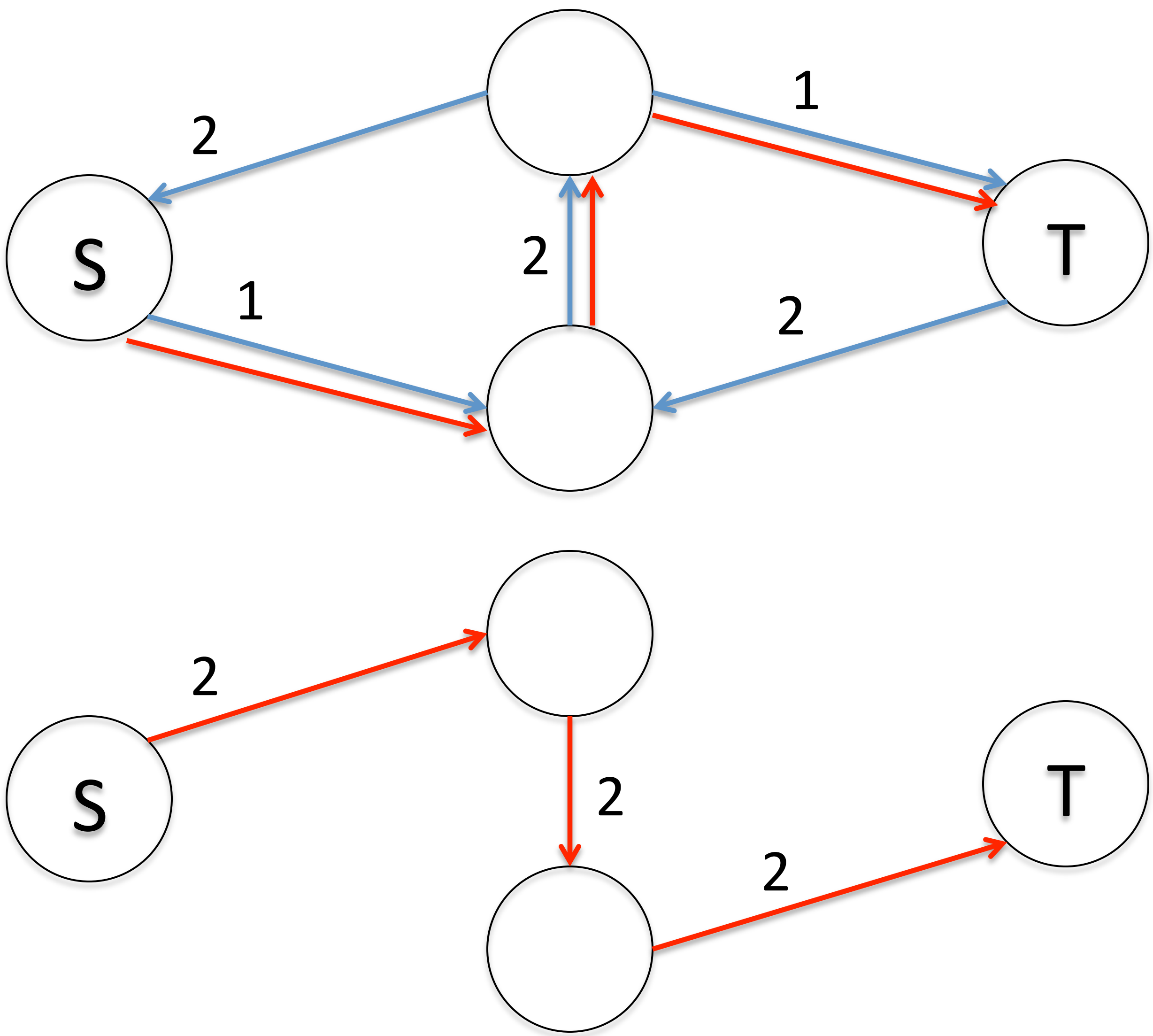




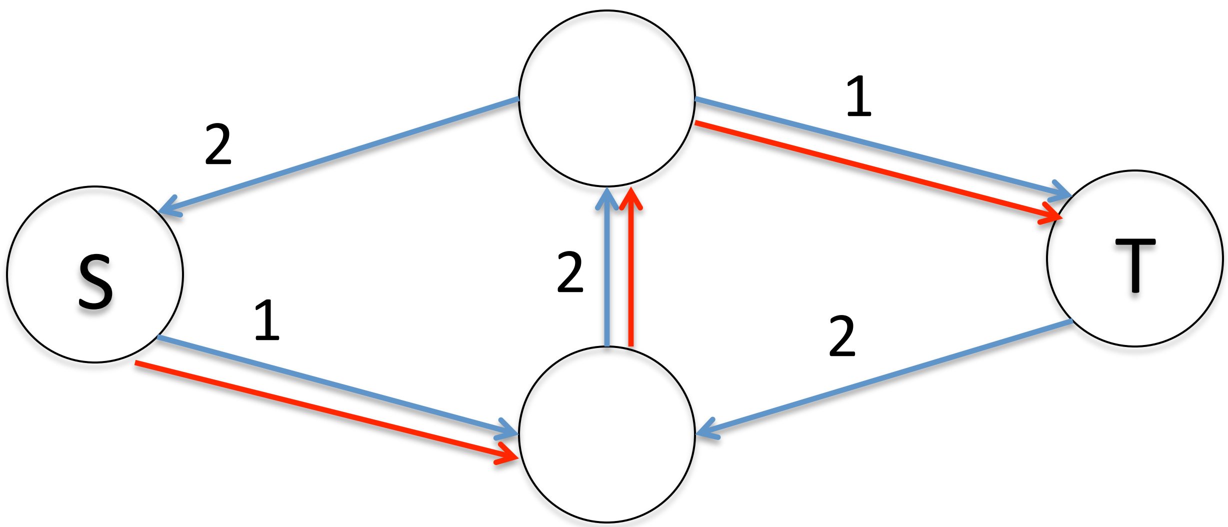
Cancelling Flows

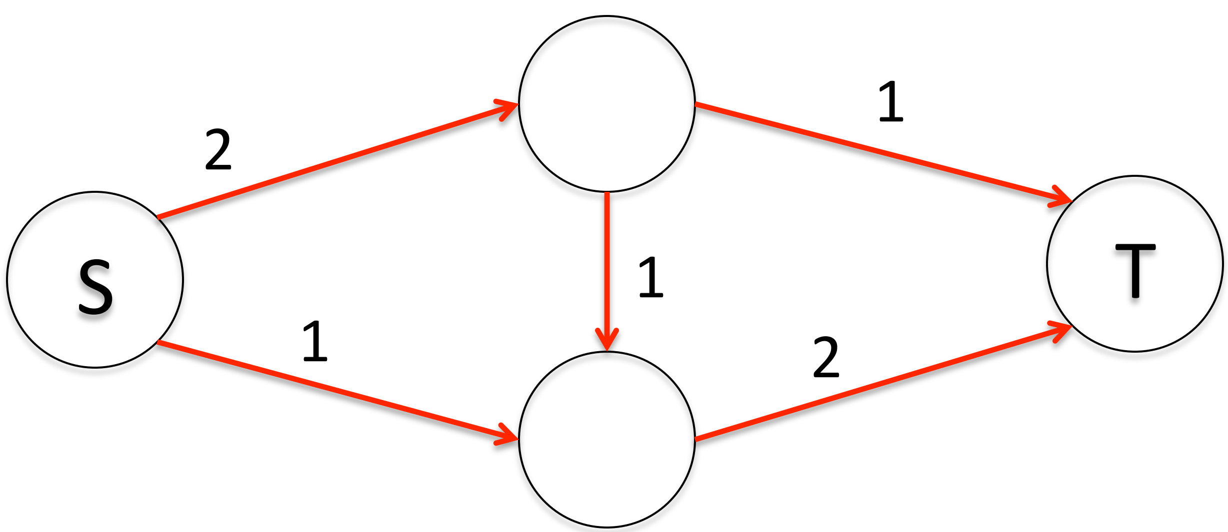


Cancelling Flows

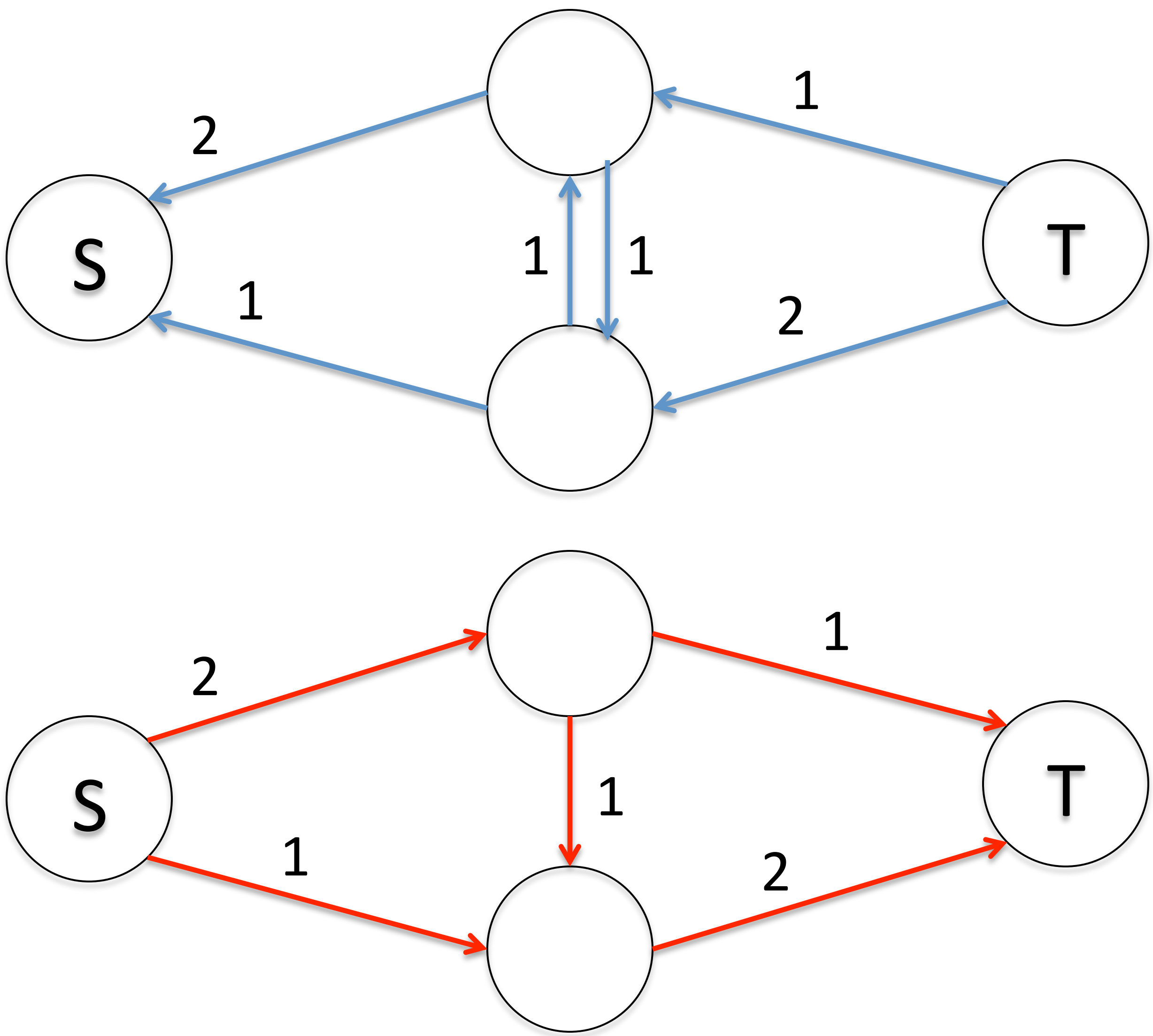


Cancelling Flows





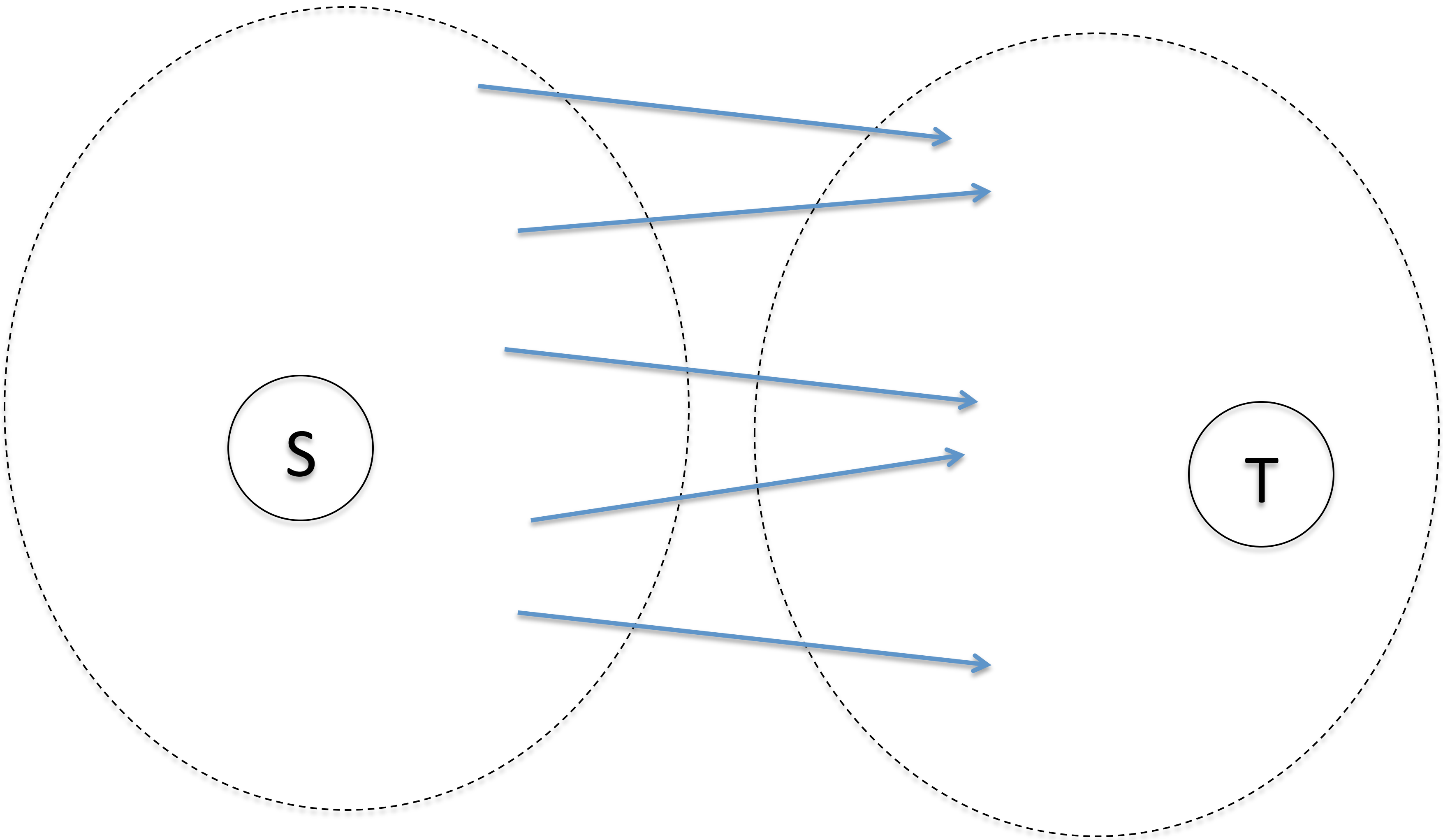
Cancelling Flows



Min Cut

- For any cut (C,V-C) where C contains s and V-C contains t, let the weight of the cut be the sum of the weights of all edges from C into V-C
- **Observation**: No flow can be greater than the weight of any cut

Max Flow/Min Cut



Max Flow/Min Cut

- **Theorem**: The weight of the maximum flow is equal to the weight of the minimum cut
- Proof: Suffices to show a flow and a cut with the same weight

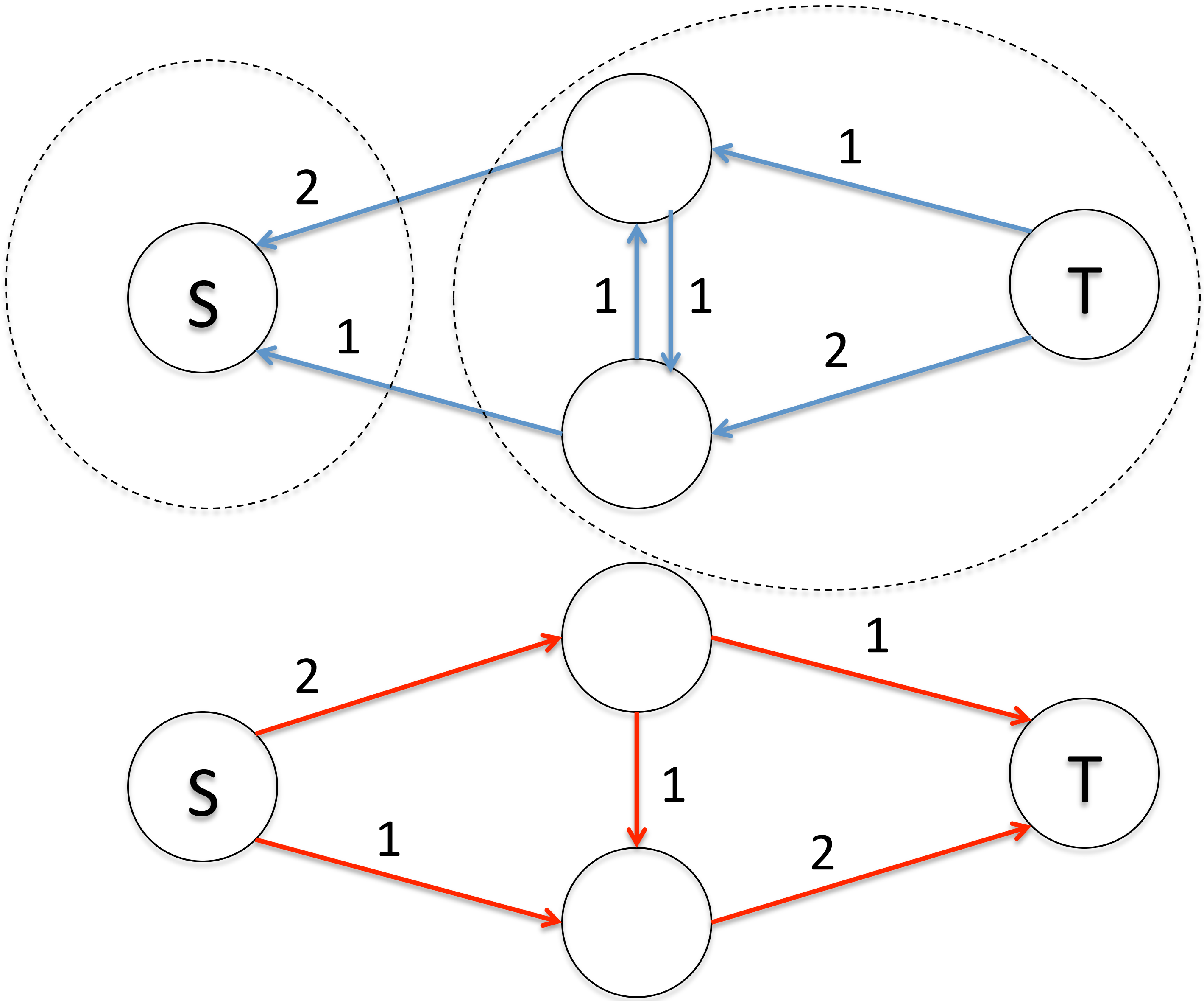
Max Flow/Min Cut

- Our algorithm for max flow halts exactly when the residual flow graph has no paths from s to t
- Run explore from s on the residual graph
- Let C be set of visited nodes, V-C set of unvisited nodes
- Claim: the cut (C,V-C) has the same weight as the flow

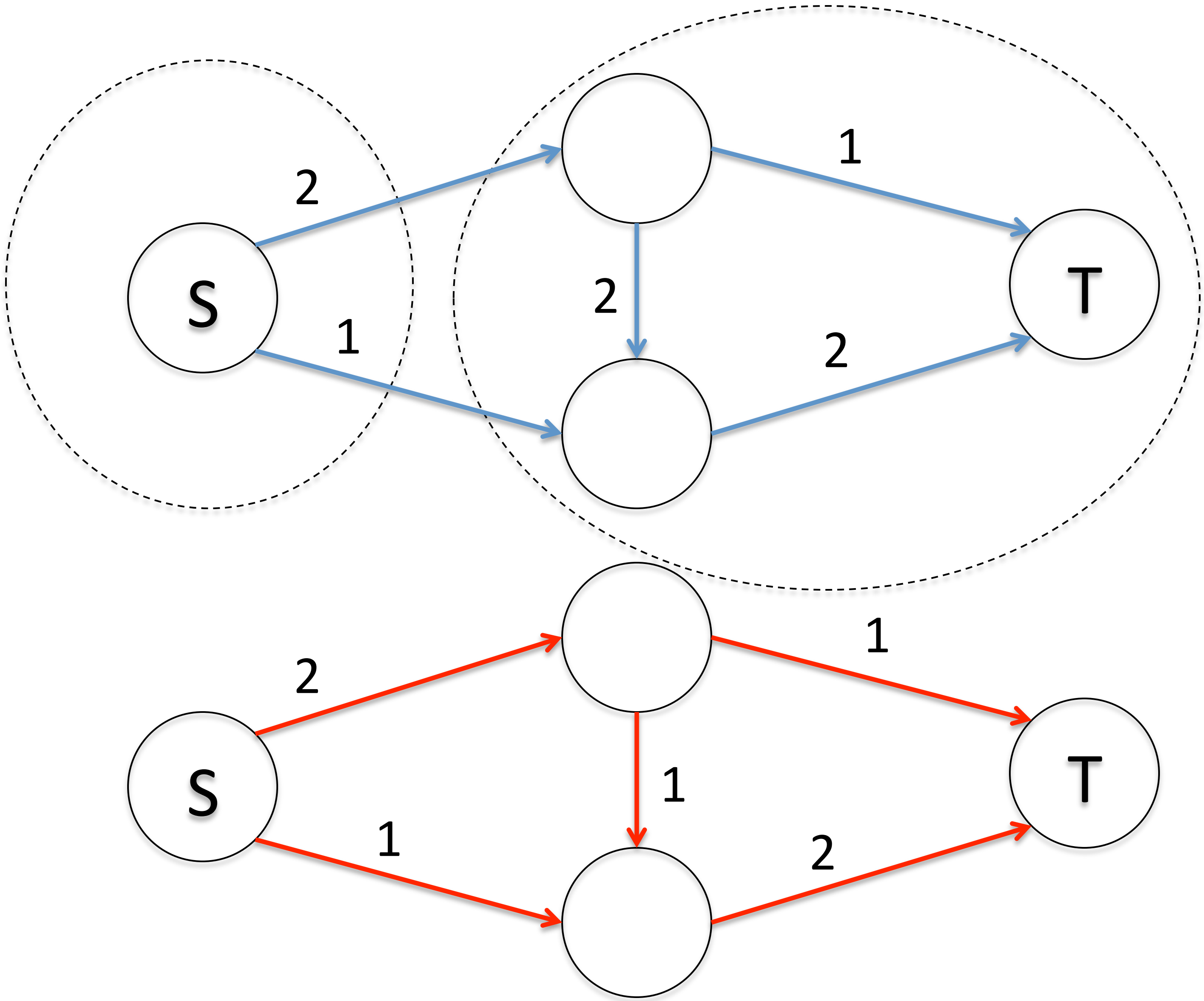
Max Flow/Min Cut

- In residual graph G^F, no edges from C to V-C
- Therefore, in G, every edge from C to V-C has its capacity used up
- Weight of cut = sum of weights of edges from C to V-C = amount of flow from s to t

Max Flow/Min Cut



Max Flow/Min Cut



Max Flow Algorithm

- We showed that our flow algorithm yields a flow that is equal to the weight of a cut
- Therefore, our flow is optimal, and the min cut is equal to the max flow
- We can also modify our algorithm to obtain the max cut
  - We can prove to someone else that our flow is optimal

Max Flow Algorithm

- Running Time?
  - Each updates requires O(|E|) time
  - How many updates?
  - Naïve answer: each update increases flow by at least 1, so if max flow has weight W, running time is O(|E| W)
  - What if W is huge?

Max Flow Algorithm

- What if we always find the path with the largest bottleneck?
  - "Fattest" path
  - Can show O(|E| log W) iterations,
  - Time: $O(|E|^2 \log W)$
  - Since log W is the number of bits needed to represent W, this is polynomial time
- What if we use BFS?
  - Can show O(|E||V|) iterations

Strong vs Weak Polynomial Time

- An algorithm is said to run in polynomial time if it runs in $O(n^c)$ where n is the size of the input
  - Graph G= (V,E) has size $O(|V|+|E|)$
  - Integer W has size $O(\log W)$

Strong vs Weak Polynomial Time

- Two models of computation:
  - Model 1: Treat all integers as consuming a constant amount of space and requiring a constant amount of time for all arithmetic operations
  - Model 2: All integers require O(log n) space and arithmetic operations take the correct amount of time.

Strong vs Weak Polynomial Time

- **Strongly Polynomial Time**:
  - The running time is polynomial in Model 1. That is, the number of arithmetic operations is $O(n^c)$ where n is the number of integers in the input.
  - The space used is polynomial in the Model 2 (correct) size of the input

Strong vs Weak Polynomial Time

- **Strongly Polynomial Time**:
  - Any strong polynomial time algorithm can be converted into a polynomial time algorithm by replacing O(1)-time operations with correct operations
  - $O(|V|^2\ |E|)$ does not depend on the size of the weights, so it is strong polynomial time

String vs Weak Polynomial Time

- **Weak Polynomial Time**:
  - Polynomial time, but not strong polynomial
  - $O(|V|^2 \log W)$ is polynomial, but number of operations in not just function of of number of integers ($|E|$), but also of their size

Max Flow as Linear Programming

- Recall what we are computing:
  - We have variables f_e for all edges e
  - We require that $0 \leq f_e \leq w(e)$ for all e
  - We also require that, for all nodes v,

$$\sum_{(u,v)\in E} f_{(u,v)} = \sum_{(v,w)\in E} f_{(v,w)}$$

  - We want to maximize

$$\sum_{(s,v)} f_{(s,v)} - \sum_{(v,s)} f_{(v,s)}$$

Max Flow as Linear Programming

- We can write the max flow problem as follows:
  - Maximize $$\sum_e c_e f_e$$
  - Subject to the constraints:

$$f_e \geq 0 \qquad f_e \leq w(e)$$

$$\sum_e a_{i,e} f_e = 0 \forall i$$

Linear Programming

- Set of variables x_i
- Goal: maximize $$\sum_i c_i x_i$$
- Subject to the constraints

$$\sum_i A_{j,i} x_i \leq b_j \forall j$$

$$x_i \geq 0 \forall i$$

Linear Programming

- Variants
  - Can be max or min problem
  - Constrains can be equations or inequalities
  - Variables can be only non-negative, or unrestricted in sign
- Turns out all equivalent!

Linear Programming

- Convert max problem to min?

$$\max \sum_i c_i x_i \longrightarrow \min \sum_i (-c_i) x_i$$

- Min to max?

$$\min \sum_i c_i x_i \longrightarrow \max \sum_i (-c_i) x_i$$

Linear Programming

- Equations to inequalities?

$$\sum_i a_i x_i = b \longrightarrow \begin{array}{c} \sum_i a_i x_i \leq b \\ \sum_i a_i x_i \geq b \end{array}$$

- Inequalities to equations?

$$\sum_i a_i x_i \leq b \longrightarrow \begin{array}{c} \sum_i a_i x_i + z = b \\ z \geq 0 \end{array}$$

Linear Programming

- Unrestricted to non-negative?
  - For each variable x, introduce new variables x^+, x^-
  - Add constraints $x^+ \geq 0$, $x^- \geq 0$
  - Replace each occurrence of x with $x^+ - x^-$

Solving Linear Programming

$$\max \sum_i c_i x_i$$

$$\sum_i A_{j,i} x_i \leq b_i \forall j$$

$$x_i \geq 0 \forall i$$

Solving Linear Programming

- Each inequality defines a plane, feasible solutions all to one side of plane (half-space)
- Intersection of all half-spaces is feasible region. Result is a **polytope**
- **Theorem**: maximum solution must lie on a vertex of the polytope

The Simplex Algorithm

- Start at any vertex of the polytope, and repeatedly:
  - Follow an edge from the current vertex to a more optimal vertex
  - Stop when the current vertex is better than all its neighbors

Simplex and Max Flow

- Starting with a solution, and repeatedly improving is exactly what we did in our max flow algorithm
- Simplex algorithm on max flow problem gives exactly the algorithm we had

The Simplex Algorithm

- Issues:
  - Finding a starting point
  - If we pick a bad edge to follow, can run poorly
- Though not polynomial time on all instances, simplex tends to work well on many real-world inputs

Linear Programming

- Invented during WWII
- 1947 – Simplex method
- 1979 - Provably weak polynomial time
- Unlike the max flow algorithm, no algorithm known that solves linear programming in strongly polynomial time