# ADSA – Unit 6
# Advanced Topics

Dr. Sreeja S R

# Topics to be covered

- Network Flows
- Randomized Algorithms
- Computational Complexity
  - NP-completeness
  - Polytime reductions

# Network Flows

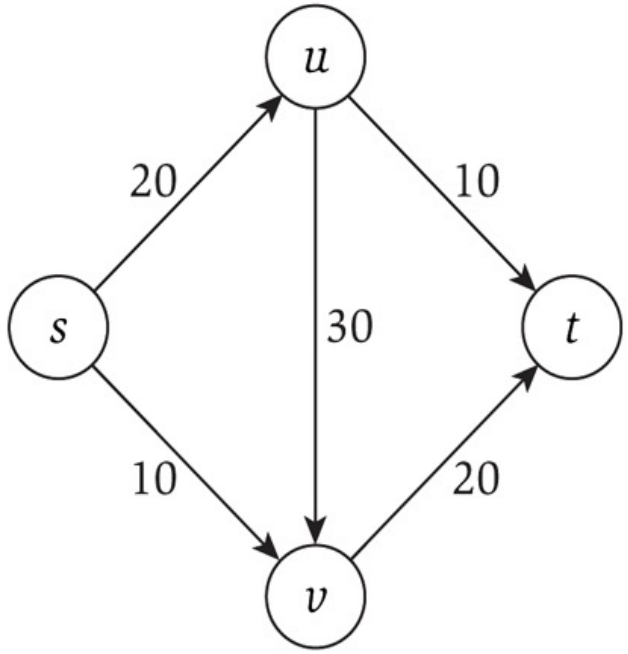
# Network Flows

- A wide variety of engineering and management problems involve optimization of network flows – that is, how objects move through a network.
- Examples include
  - coordination of trucks in a transportation system
  - routing of packets in a communication network
  - sequencing of legs for air travel.

# Flow Networks

A *flow network* is a directed graph $G(V, E)$

- Each edge $e \in E$ has a capacity $c(e) > 0$.
- There is a single *source* node $s \in V$.
- There is a single *sink* node $t \in V$.
- Nodes other than s and t are *internal*.

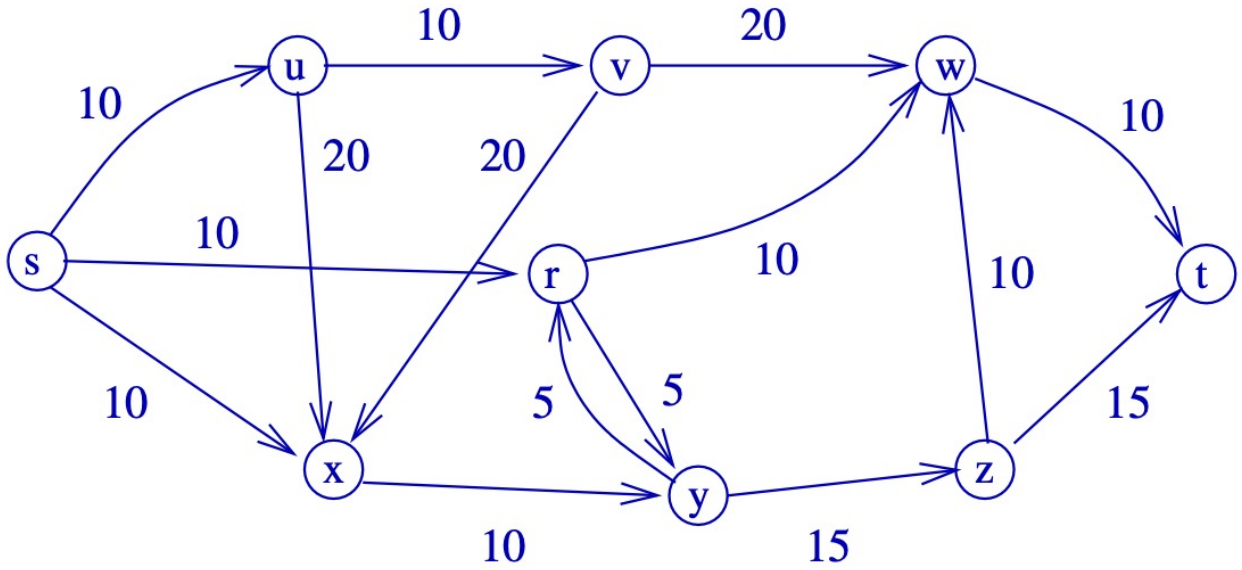



*A flow network with source s and sink t. The numbers next to the edges are the capacities.*

**Note: A capacity function $c: V \times V \to \mathbb{R}$ such that $c(u, v) \geq 0$ if $(u, v) \in E$ and $c(u, v) = 0$ for all $(u, v) \notin E$.**

# Flow Networks

Example:




For this graph, $V = \{s, r, u, v, w, x, y, z, t\}$

The edge set is

$$E = \{(s,u),(s,r),(s,x),(u,v),(u,x),(v,x),(v,w),(r,w),(r,y),(x,y),(y,r),$$
$$(y,z),(z,w),(z,t),(w,t)\}.$$

Some examples of capacities are $c(s,x) = 10$, $c(r,y) = 5$, $c(v,x) = 20$ and $c(v,r) = 0$ (since there is no arc from v to r).

# Network Flows

Let $\mathcal{N} = (G = (V, E), c, s, t)$ be a flow network.

A *flow* in $\mathcal{N}$ is a function $\boldsymbol{f}: \boldsymbol{V} \times \boldsymbol{V} \to \mathbb{R}$ satisfying the following conditions:

**Capacity constraint:** $f(u, v) \leq c(u, v)$ for all $u, v \in V$.

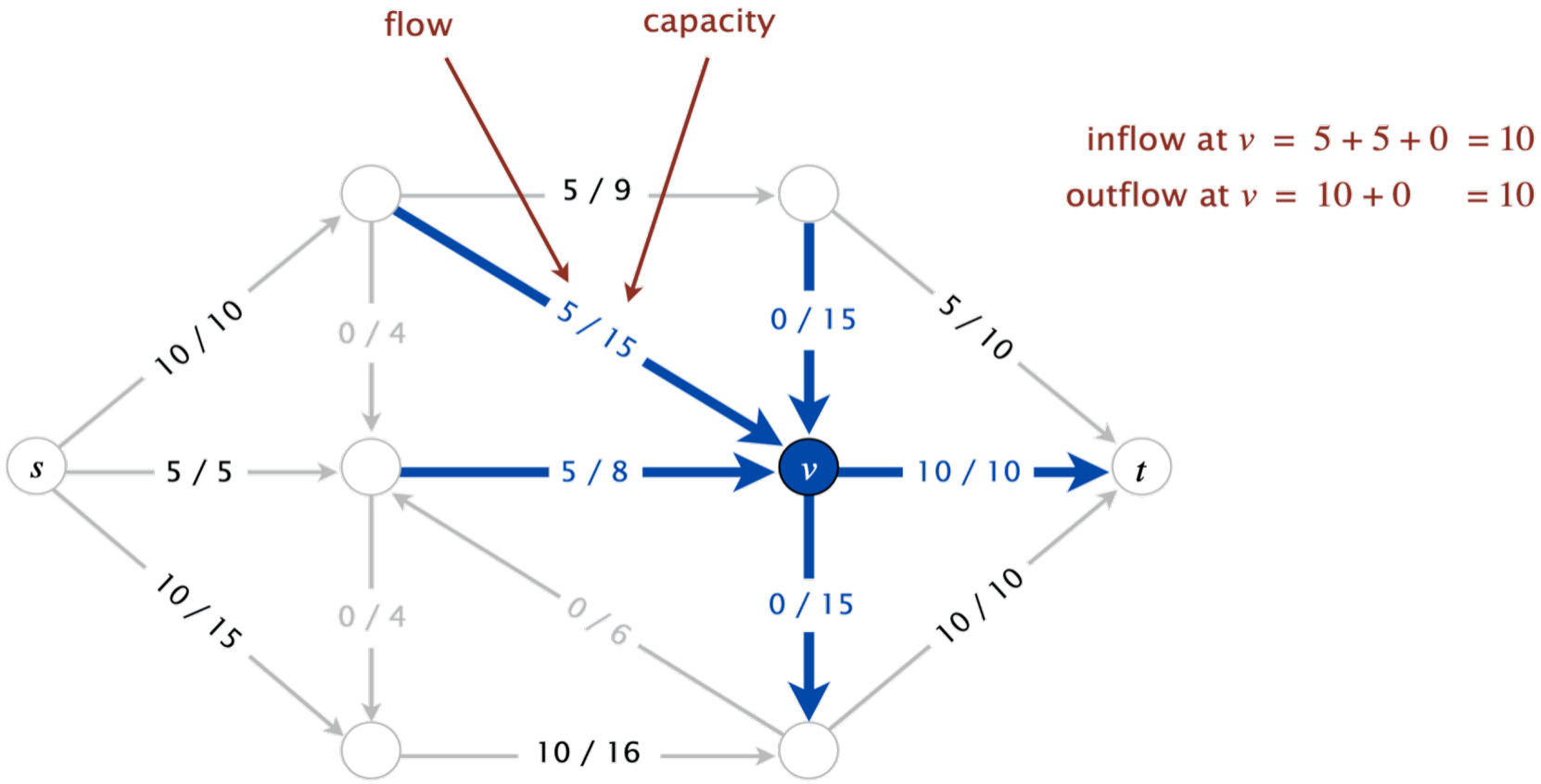
**Flow conservation:** For all $u \in V \setminus \{s, t\}$,

$$\underbrace{\sum_{v \in V} f(v, u)}_{\text{flow into u}} = \underbrace{\sum_{v \in V} f(u, v)}_{\text{flow out of u}}$$

Note: In the case that flow conservation is satisfied, one can prove that the net flow out of *s* equals the net flow into *t*. This quantity is called the **flow value**, or simply the magnitude, of $f$ .

# Network Flows

- **Capacity:** We cannot overload an edge
- **Conservation:** Flow entering any vertex must be equal to flow leaving that vertex
- We want to maximize the **value of a flow**, subject to these constraints
- A **saturated edge** is at maximum capacity



# Maximum-Flow Problem

Input: Network $\mathcal{N}$
Output: Flow of maximum value in $\mathcal{N}$

The problem is to find the flow $f$ such that $|f| = \sum_{v \in V} f(s, v)$ is the largest possible (over all "legal" flows).

**A maximum flow is defined as the maximum amount of flow that the graph or network would allow to flow from the source node to its sink node.**

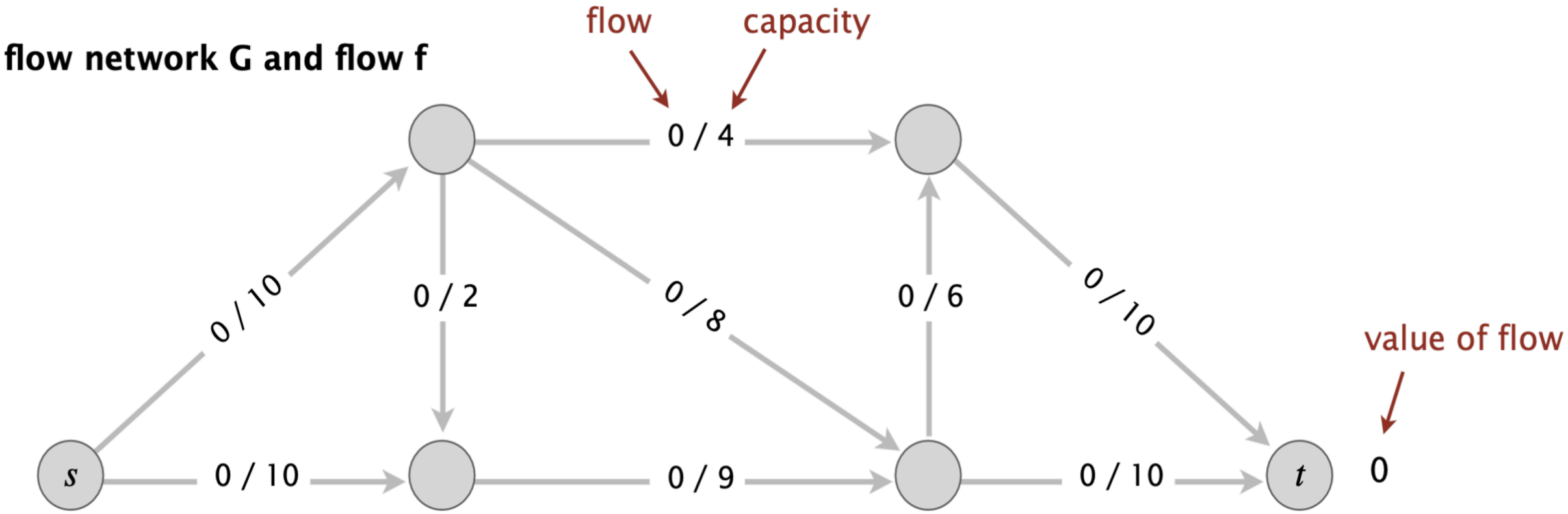
**Assumptions:**
1. No edges enter s, no edges leave t.
2. There is at least one edge incident on each node.
3. All edge capacities are integers.

# Maximum-Flow Problem

**No known dynamic programming** algorithm to solve this problem.

Let us take a greedy approach.

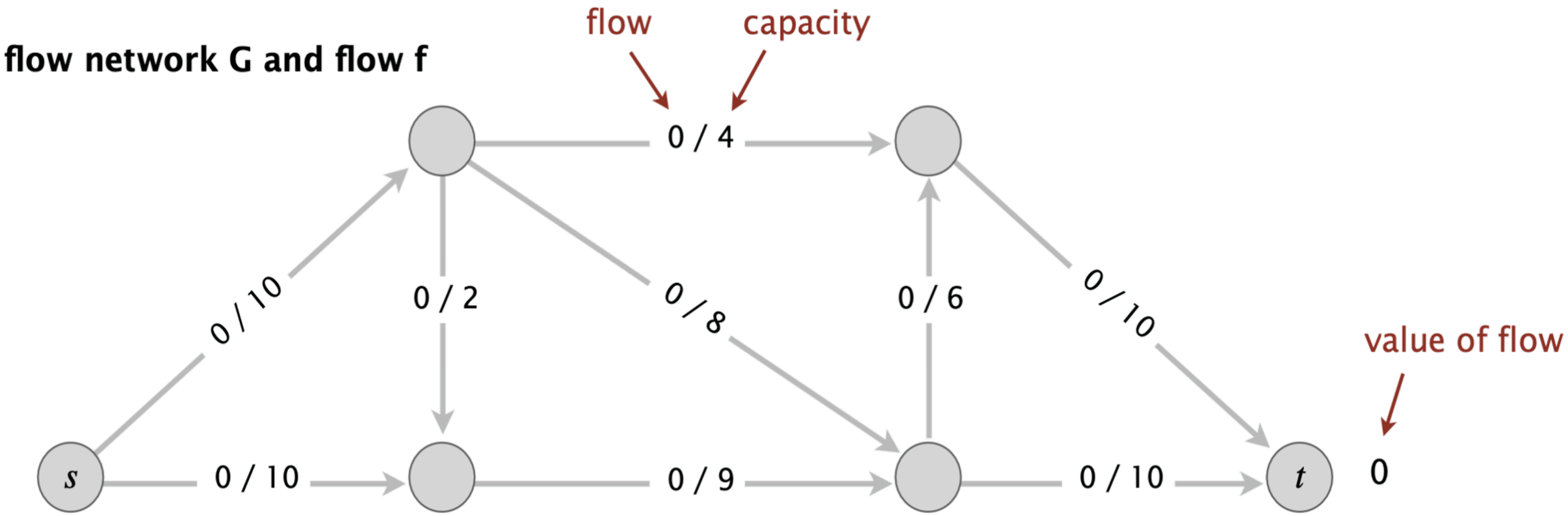
1. Start with zero flow along all edges.



# Maximum-Flow Problem

No known dynamic programming algorithm to solve this problem.
Let us take a greedy approach.

1. Start with zero flow along all edges.

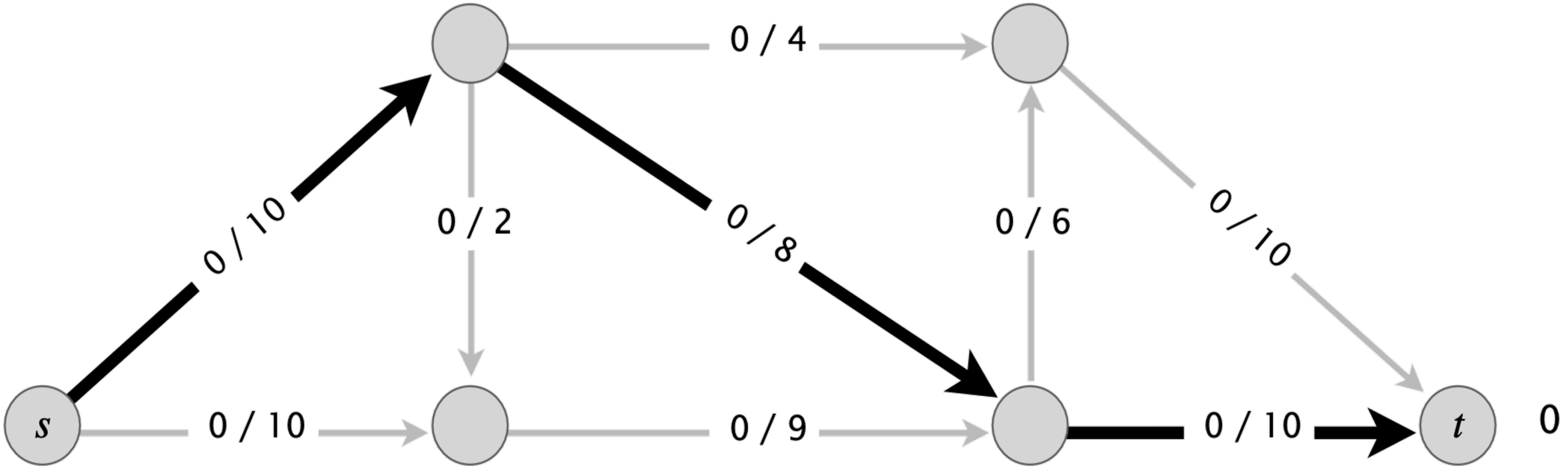


# Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s-t path P where each edge has $f(u, v) < c(u, v)$

**flow network G and flow f**

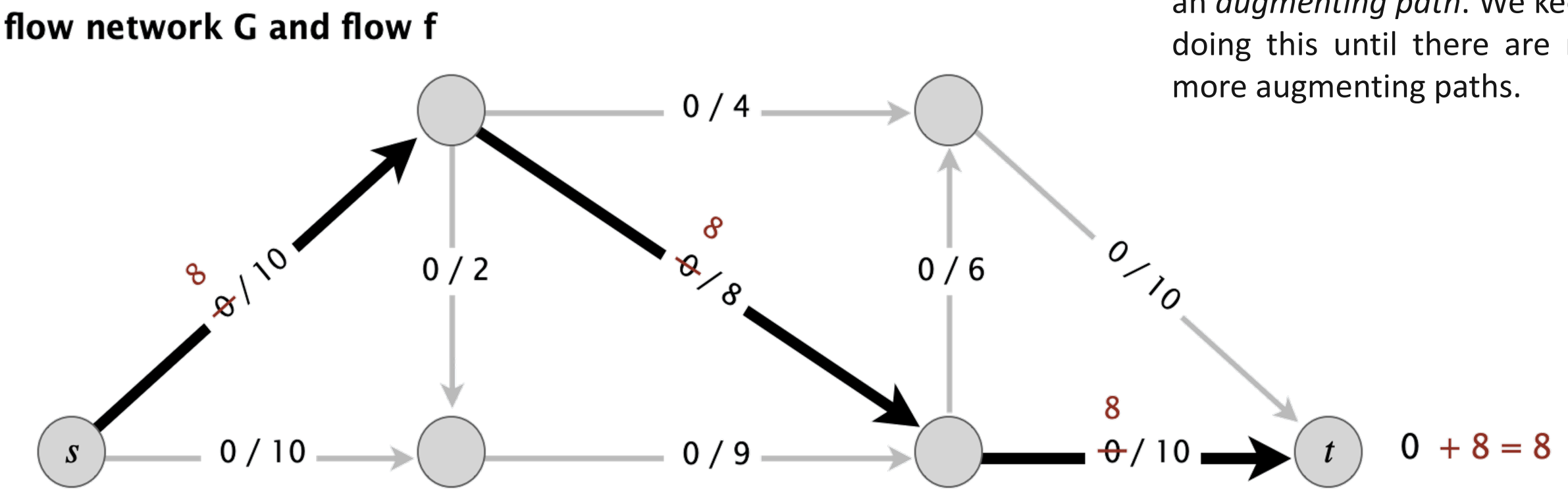


# Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s-t path $P$ where each edge has $f(u,v) < c(u,v)$
3. Augment flow along path $P$.

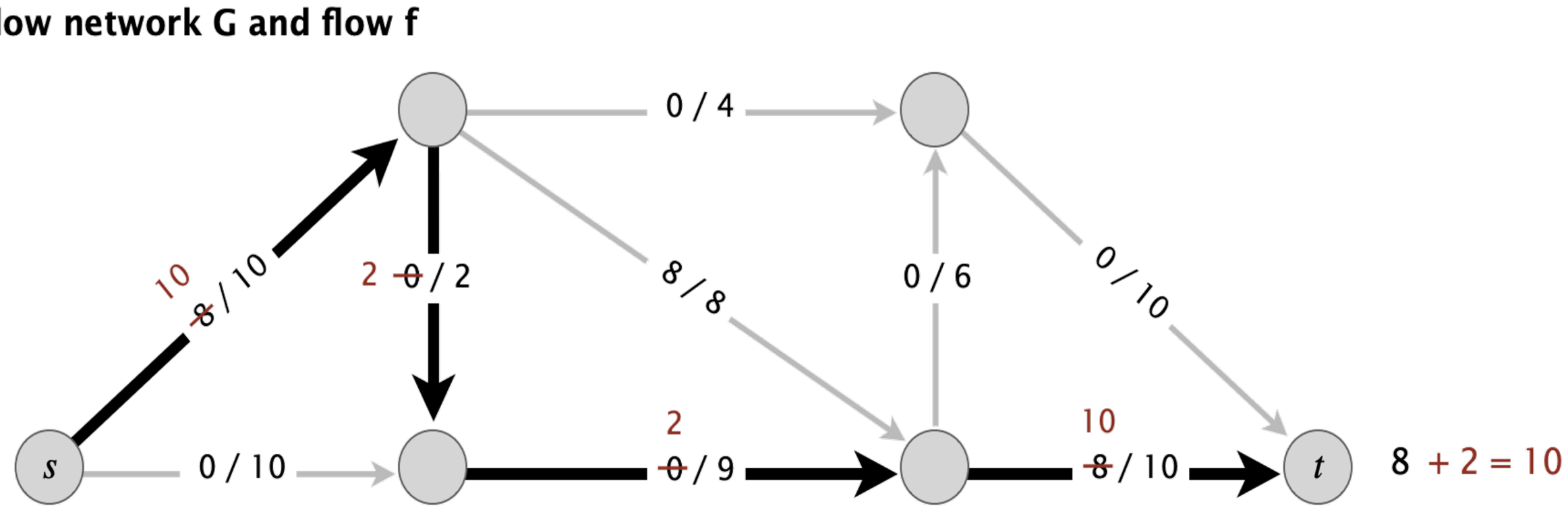
**Augmenting Path:** As long as there is a path from the source to the sink that can take some flow the entire way. This path is called an *augmenting path*. We keep doing this until there are no more augmenting paths.



# Maximum-Flow Problem

Greedy algorithm.

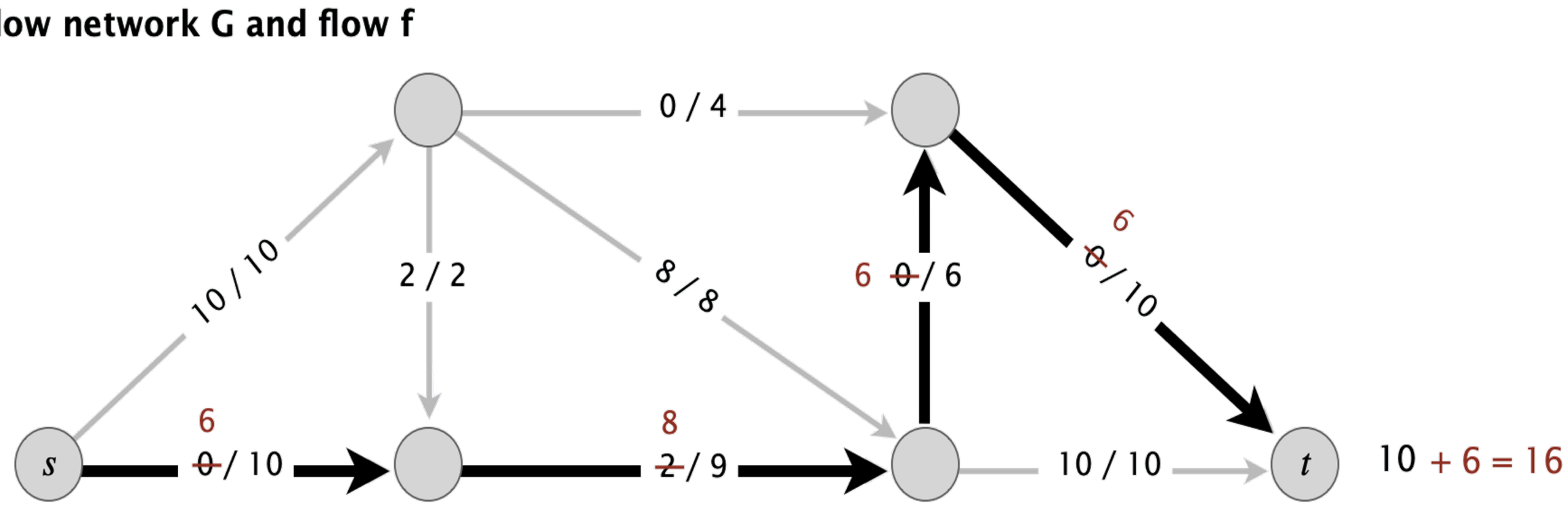
1. Start with zero flow along all edges.
2. Find an s-t path $P$ where each edge has $f(u,v) < c(u,v)$
3. Augment flow along path $P$.
4. Repeat until you get stuck.



# Maximum-Flow Problem

Greedy algorithm.

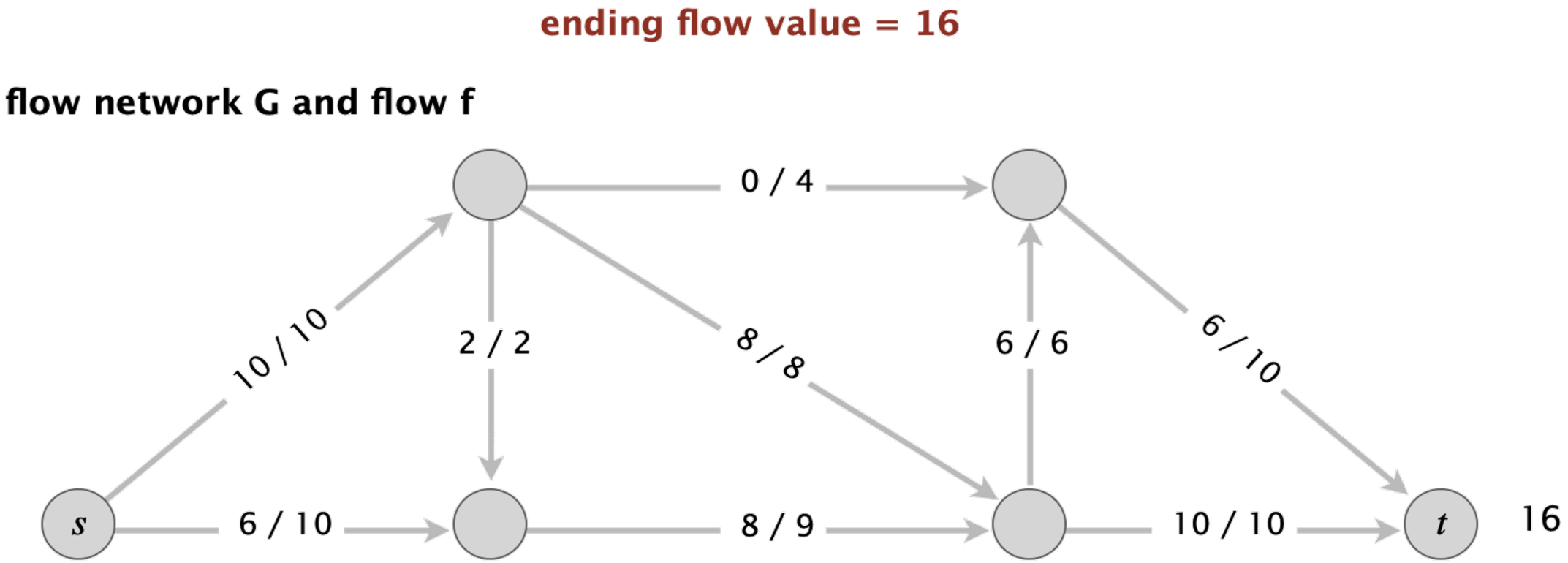
1. Start with zero flow along all edges.
2. Find an s-t path $P$ where each edge has $f(u,v) < c(u,v)$
3. Augment flow along path $P$.
4. Repeat until you get stuck.



# Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s-t path $P$ where each edge has $f(u, v) < c(u, v)$
3. Augment flow along path $P$.
4. Repeat until you get stuck.

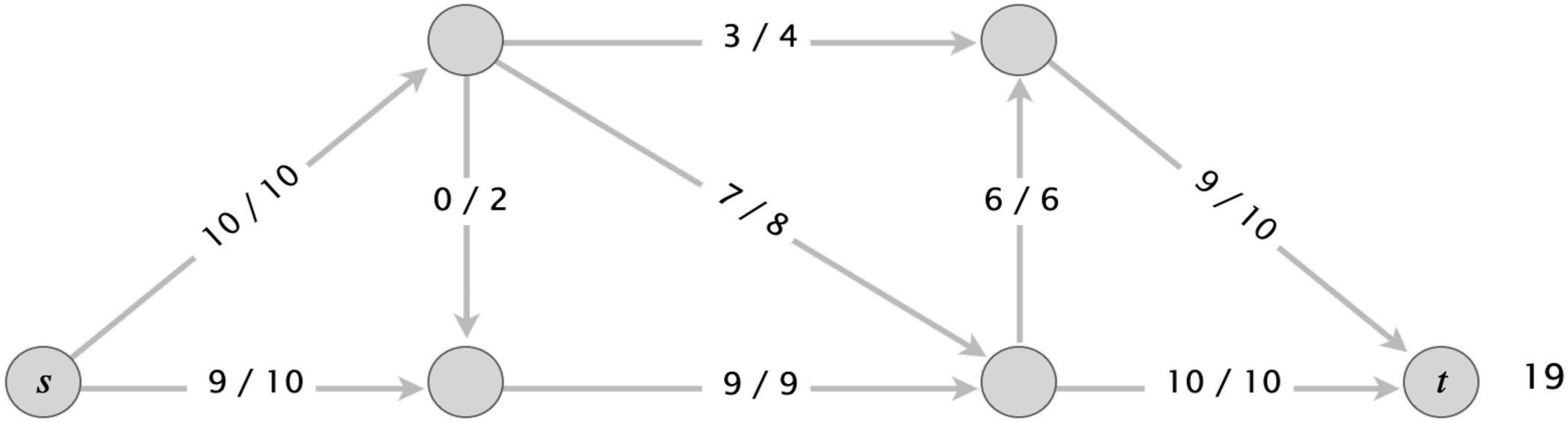


# Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s-t path $P$ where each edge has $f(u, v) < c(u, v)$
3. Augment flow along path $P$.
4. Repeat until you get stuck.

**but max-flow value = 19**



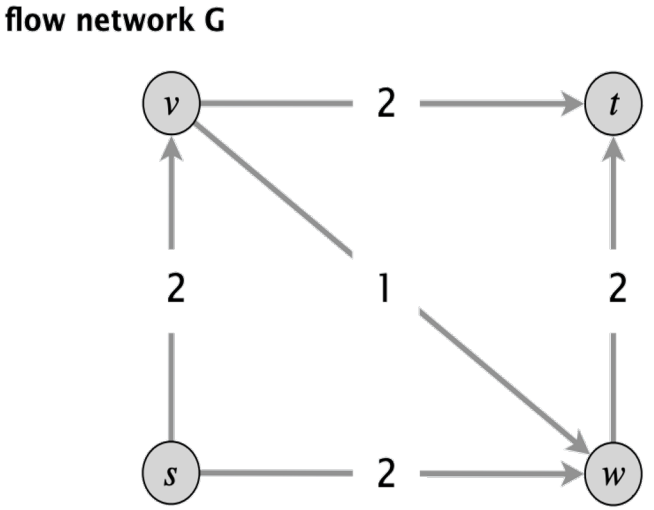
# Why does the greedy algorithm fail?

Q. **Why does the greedy algorithm fail?**

A. Once greedy algorithm increases flow on an edge, it never decreases it.

Ex. Consider flow network *G*

- Greedy algorithm could choose $s \to v \to w \to t$ as first path.



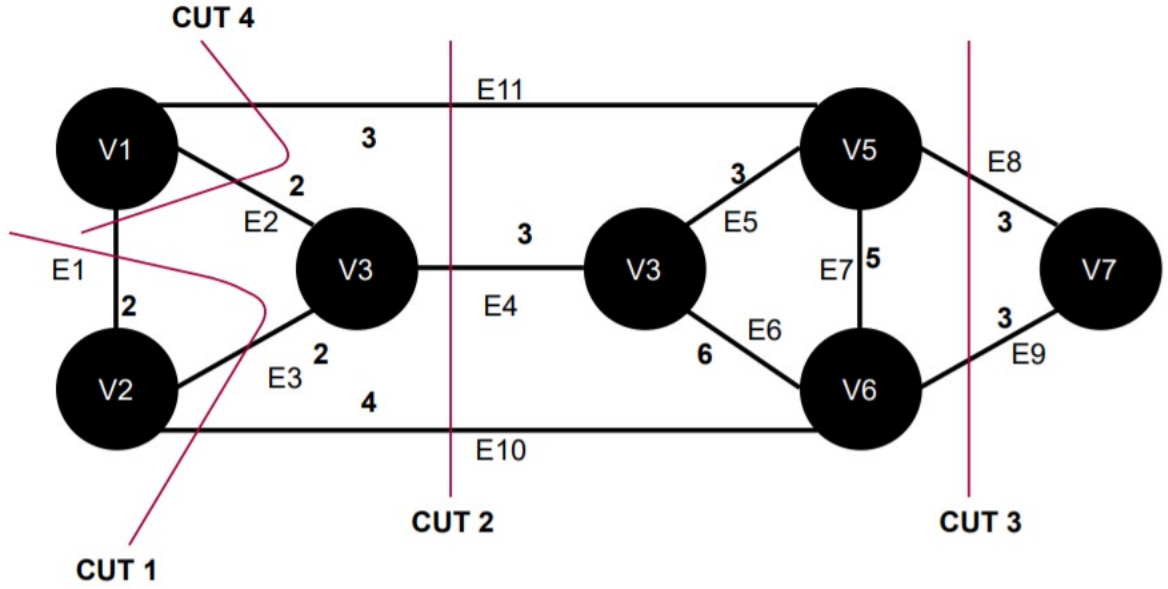

We need some mechanism to ***undo*** a bad decision. Can we do it by just modifying the graph?

- **Yes.**

# Minimum Cut in a Graph

A **cut** is a set of edges whose removal divides a connected graph into two disjoint subsets.

**The minimum cut of a weighted graph is defined as the minimum sum of weights of edges that, when removed from the graph, divide the graph into two sets.**




Here in this graph, CUT 3 is an example of a minimum cut. It removes the edges E8 and E9, and the sum of weights of these two edges are minimum among all other cuts in this graph.

# Max-Flow Min-Cut Theorem

**The max-flow min-cut theorem states that the maximum flow through any network from a given source to a given sink is exactly equal to the minimum sum of a cut.**

The **Ford-Fulkerson algorithm** is an algorithm that tackles the max-flow min-cut problem. This algorithm finds the maximum flow of a network or graph.

**The Ford-Fulkerson algorithm is based on the three important concepts: the residual network, augmented path and cut.**

# Ford–Fulkerson algorithm

The **Ford–Fulkerson algorithm** is an elegant solution to the maximum flow problem.

Ford–Fulkerson augmenting path algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \to t$ path $P$ in the **residual network** $G_f$.
- **Augment flow** along path $P$.
- Repeat until you get stuck.

Note: If there are multiple possible augmenting paths, the decision of which path to use is completely arbitrary. Thus, like any terminating greedy algorithm, the Ford–Fulkerson algorithm will find a locally optimal solution.

# Residual Graph

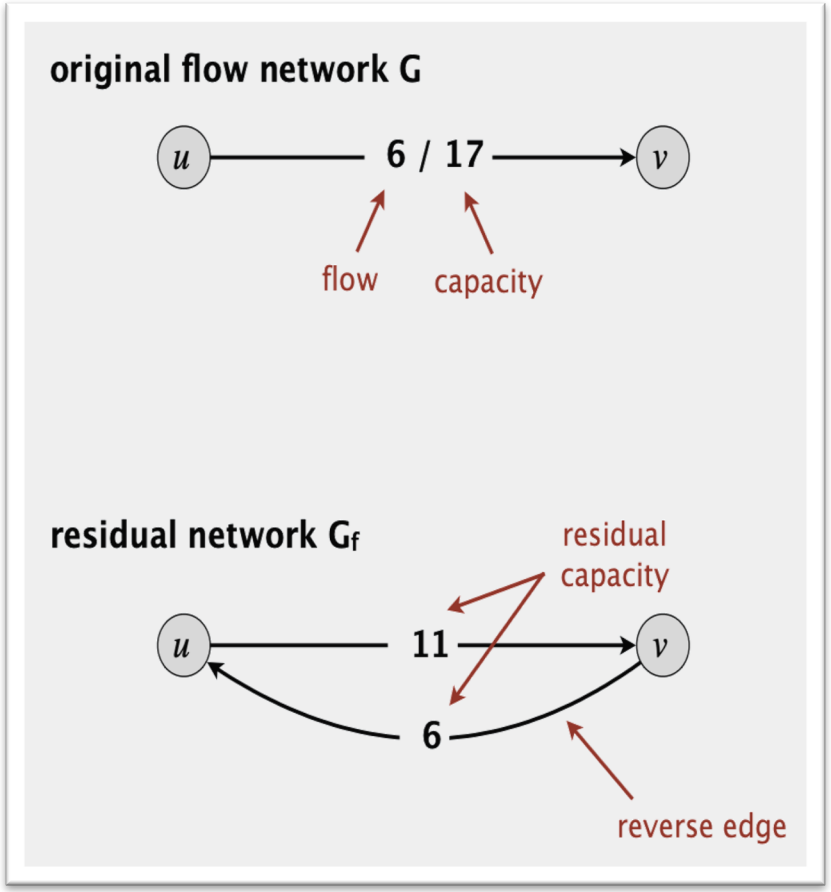
***Residual capacity*** is an important attribute and it plays an important role in residual graph creation. Residual capacity is defined as the new capacity after a given flow has been taken away. In other words, for a given edge $(u, v)$ the residual capacity, $C_f$ is defined as

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

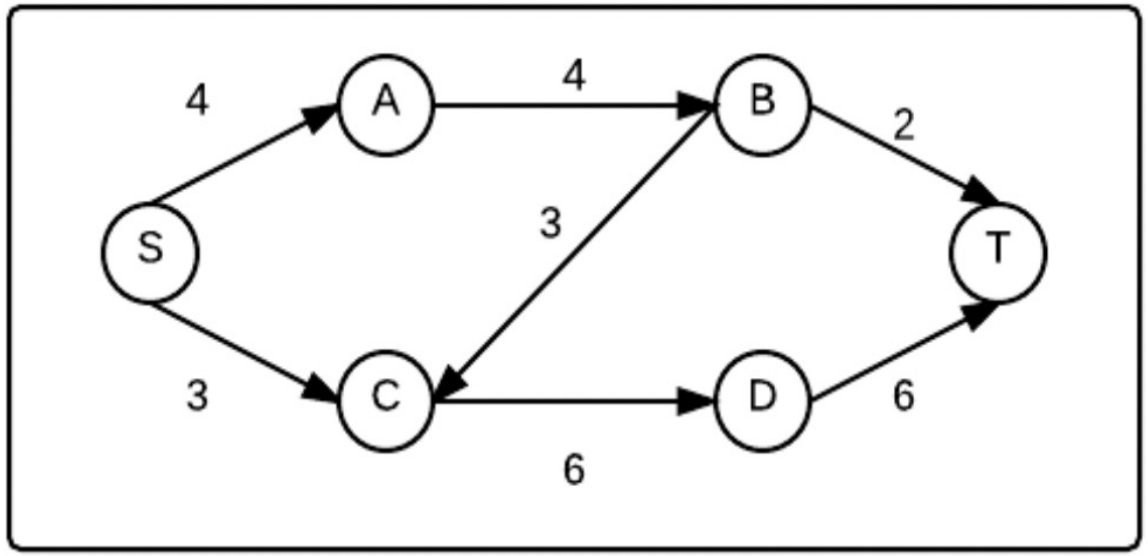
However, there must also be a residual capacity for the ***reverse*** **edge** as well. The max-flow min-cut theorem states that flow must be preserved in a network. So, the following equality always holds:

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

With these tools, it is possible to calculate the residual capacity of any edge, forward or backward, in the flow network. Then, these residual capacities are used to make a residual network, $G_f$



# Example




*Flow network before Ford-Fulkerson*

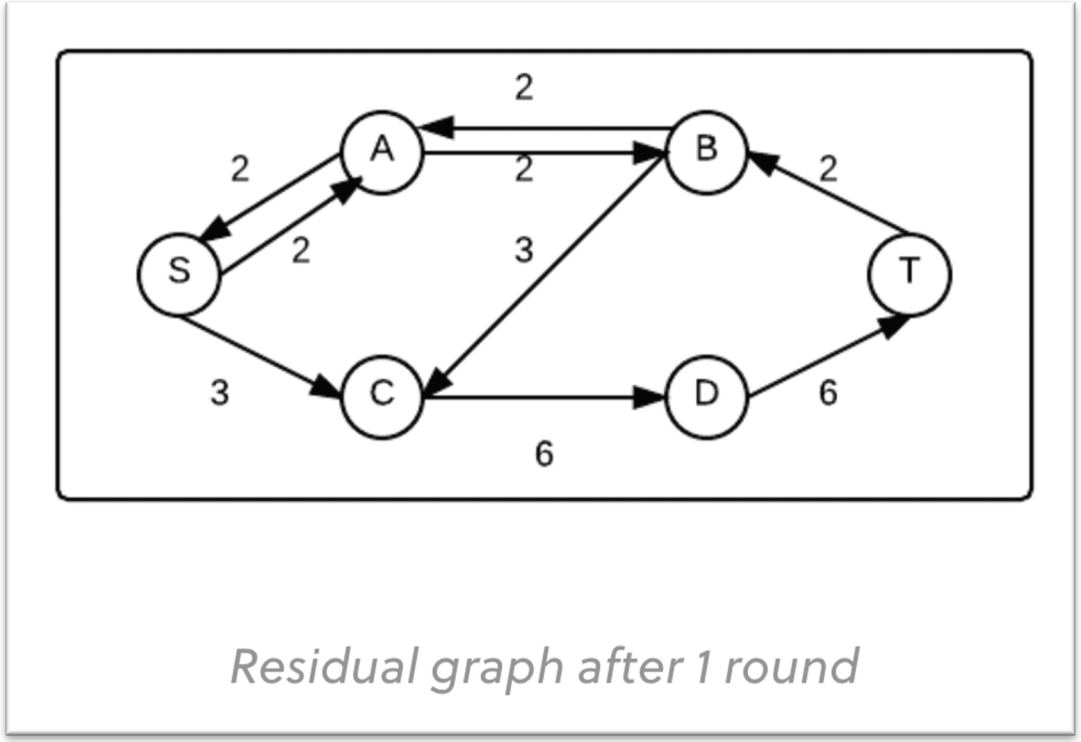
# Example

Initially, 2 units of flow are pushed along the top-most path (S, A), (A, B), and (B, T). Two things happen:

**1. In the forward direction**, the edges now have a residual capacity equal to $c_f(u, v) = c(u, v) - f(u, v)$. The flow is equal to 2, so the residual capacity of (S, A) and (A, B) is reduced to 2, while the edge (B, T) has a residual capacity of 0.

**2. In the backward direction**, the edges now have a residual capacity equal to$c_f(v, u) = c(v, u) - f(v, u)$. Because of flow preservation, this can be written $c_f(v, u) = c(v, u) + f(u, v)$. And since the capacity of those backward edges was initially 0, all of the backward edges (T, B), (B, A), and (A, S) now have a residual capacity of 2.

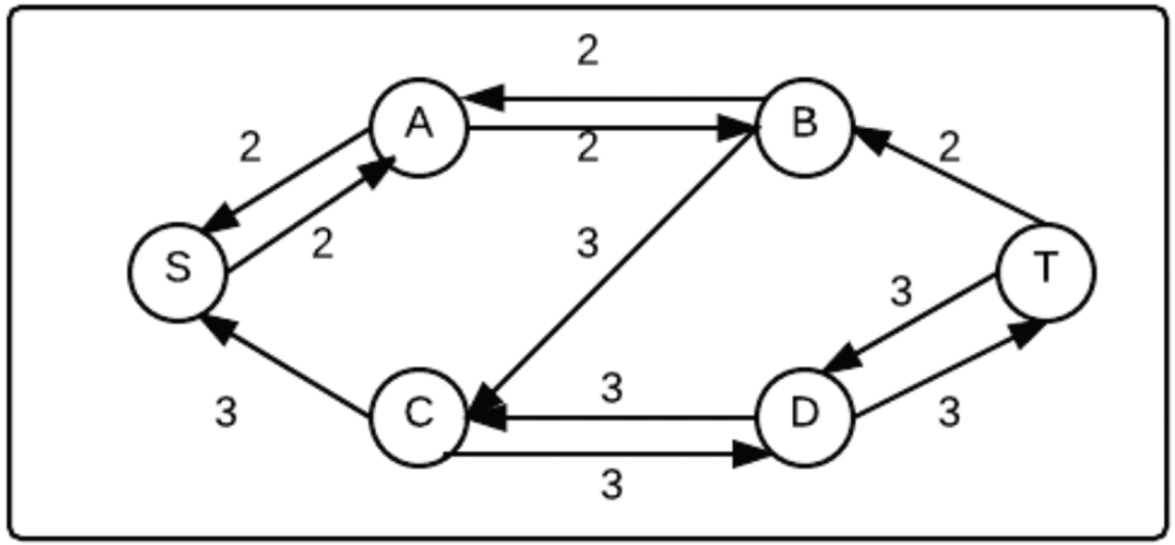
When a new residual graph is constructed with these new edges, any edges with a residual capacity of 0—like (B, T)—are not included.




*Residual graph after 1 round*

# Example

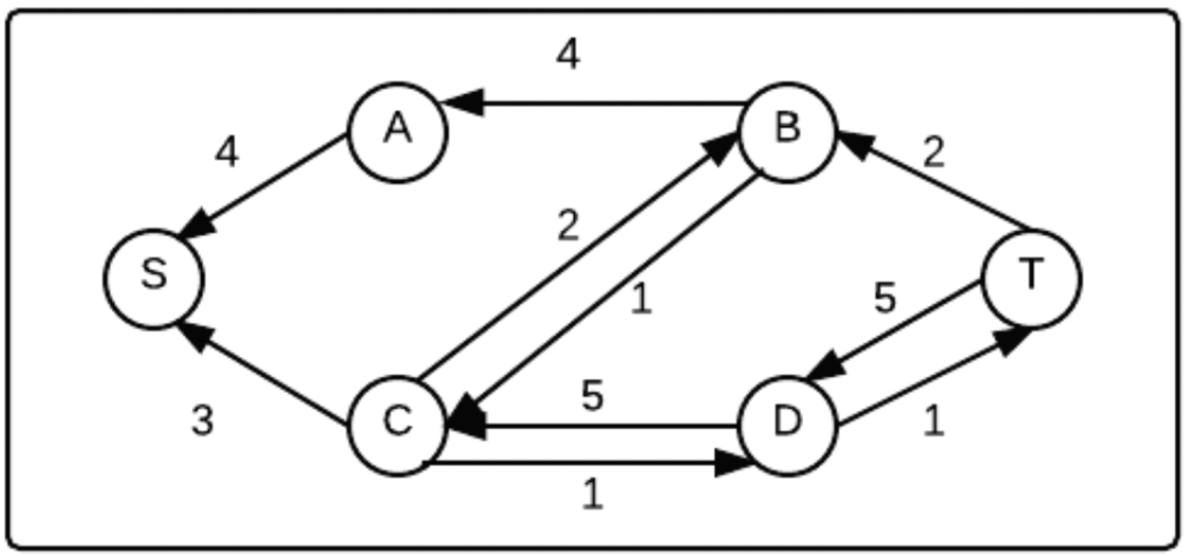
Now, a new augmenting path must be found (the top-most path can never be used again because the edge (B, T) was erased). The bottom path can be chosen and flow of 3 can be sent along it.




*Residual graph after 2 rounds*

# Example

Finally, a flow of 2 can be sent along the path [(S, A), (A, B), (B, C), (C, D), (D, T)] because the minimum residual capacity along that path is 2. The final residual graph is given here.




*Residual graph after 3 rounds*

There are no more paths from the source to the sink, so there can be no more augmenting paths. Therefore, the loop is complete. The flow, 7, is a maximum flow.

# Complexity of Ford–Fulkerson algorithm

The analysis of Ford-Fulkerson depends heavily on how the augmenting paths are found. The typical method is to use breadth-first search to find the path. If this method is used, Ford-Fulkerson runs in polynomial time.

If all flows are integers, then the while loop of Ford-Fulkerson is run at most $|f^*|$ times, where $f^*$ is the maximum flow. This is because the flow is increased, at worst, by 1 in each iteration. Finding the augmenting path inside the while loop takes $O(V + E')$, where $E'$ is the set of edges in the residual graph. This can be simplified to $O(E)$.

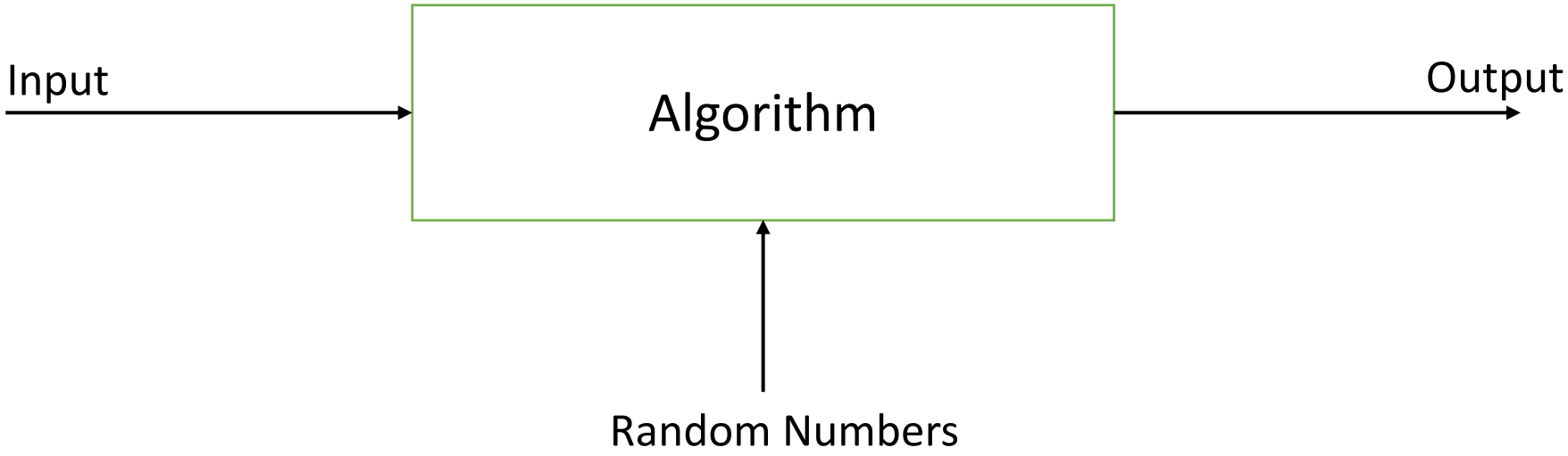
So, the runtime of Ford-Fulkerson is $O(E.|f^*|)$

# Randomized Algorithms

# Deterministic Algorithms


Goal: To prove that the algorithm solves the problem correctly (always) and quickly (typically, the number of steps should be polynomial in the **size of the input**).

# Randomized Algorithms




- In addition to input, algorithm takes a source of random numbers and makes random choices during execution.
- Behavior can vary even on a fixed input.
- Design algorithm and analysis so that this behavior is likely to be good, on every input.

# Randomized Algorithms

- A randomized algorithm is a technique that uses a source of randomness as part of its logic.
- It is typically used to reduce either the running time, or time complexity; or the memory used, or space complexity, in a standard algorithm.
- The algorithm works by generating a random number, r, within a specified range of numbers, and making decisions based on r's value.

# Randomized Algorithms

- Randomized algorithms are usually designed in one of two common forms: as a **Las Vegas** or as a **Monte Carlo algorithm**.

- Practically speaking, computers cannot generate completely random numbers, so randomized algorithms in computer science are approximated using a pseudorandom number generator in place of a true source of random number, such as the drawing of a card, flipping a coin, etc.

# Advantages of randomized algorithms

- Simplicity
- Performance

For many problems, a randomized algorithm is the simplest, the fastest, or both.

# Monte Carlo and Las Vegas

- A **Monte Carlo algorithm** runs for a fixed number of steps, and produces an answer that is correct with probability $\geq 1/3$.
- A **Las Vegas algorithm** always produces the correct answer; its running time is a random variable whose expectation is bounded (say by a polynomial).
- These probabilities/expectations are only over the random choices made by the algorithm (i.e., independent of the input).
- Thus independent repetitions of Monte Carlo algorithms drive down the failure probability exponentially.

# Monte Carlo Algorithms

- The term for this algorithm, Monte Carlo, was coined by mathematicians Nicholas Metropolis, Stanislaw Ulam, and John von Neumann, working on the Manhattan Project, around the 1940s.
- The name was found in a research paper published in 1949, attributed by some sources to the fact that Ulam's uncle made a yearly trip to gamble at Monte Carlo, in Monaco.

# Monte Carlo Algorithms - Example

- The game, Wheel of Fortune, can be played using a Monte Carlo randomized algorithm.
- Instead of mindfully choosing letters, a player (or computer) picks random letters to obtain a solution, as shown in the image.
- The more letters a player reveals, the more confident a player becomes in their solution. However, if a player does not guess quickly, the chance that other players will guess the solution also increases.
- Therefore, a Monte Carlo algorithm is given a deterministic amount of time, in which it must come up with a "guess" based on the information revealed; the best solution it can come up with.

# Example contd..

- This allows for the possibility of being wrong, maybe even a large probability of being wrong if the Monte Carlo algorithm did not have sufficient time to reveal enough useful letters.
- But providing it with a time limit controls the amount of time the algorithm will take, thereby decreasing the risk of another player guessing and getting the prize.

- **Note:** **A useful property of a Monte Carlo algorithm is if the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time.**

# Approximating π

- A classic example of a Monte Carlo algorithm lies in the solution to approximating pi, π, the ratio of a circle's circumference to its diameter.
- Imagine you're blindfolded and someone asks you to find the shape of an object. Intuitively, the solution would be to touch the object in as many places as possible until a familiar object come to mind, at which point you make a guess. The same strategy is applied when approximating π.

- **For better understanding of Monte Carlo simulation for approximating π, you can watch this video.**

# Las Vegas Algorithms

- The term for this algorithm, Las Vegas, is attributed to mathematician Laszlo Babai, who coined it in 1979 simply as a parallel to the much older Monte Carlo algorithm, as both are major world gambling centers.
- However, the gambling styles of the two have nothing to do with the styles of the algorithms, as it cannot be said that gambling in Las Vegas always gives a correct, or even positive turnout.

# Las Vegas Algorithms

- A Las Vegas algorithm runs within a specified amount of time. If it finds a solution within that timeframe, the solution will be exactly correct; however, it is possible that it runs out of time and does not find any solutions.

- A Las Vegas algorithm is by definition a Monte Carlo algorithm with error probability 0.

# Las Vegas Algorithm - Example

- A Las Vegas algorithm could be thought of as the strategy used by a user who searches for something online.
- Since searching every single website online is extremely inefficient, the user will generally use a search engine to get started. The user will then surf the web until a website is found which contains exactly what the user is looking for.
- Since clicking through links is a randomized process, assuming the user does not know exactly what's contained on the website at the other end, the time complexity ranges from getting lucky and reaching the target website on the first link, to being unlucky and spending countless hours to no avail.

**Why this example follows Las Vegas algorithm?**

# Example contd..

Because here the user knows exactly what she is looking for, so once the website is found, there is no probability of error. Similarly, if the user's allotted time to surf the web is exceeded, she will terminate the process knowing that the solution was not found.

# Randomized Quicksort

- A common Las Vegas randomized algorithm is **quicksort**, a sorting algorithm that sorts elements in place, using no extra memory.
- Since this is a comparison based algorithm, the worst case scenario will occur when performing pairwise comparison, taking $O(n^2)$, where the time taken grows as a square of the number of digits to be sorted grows. However, through randomization, the runtime of this algorithm can be reduced up to O(n log(n)).

# Complexity of Randomized Algorithms

- Randomized algorithms have a complexity class of their own.
- The basic probabilistic complexity class is called RP, randomized polynomial time algorithms.
- It encompass problems with an efficient randomized algorithm, taking polynomial time, and recognizes bad solutions with absolute certainty, and correct solutions with probability of at least ½.

# Complexity of Randomized Algorithms

- Randomized algorithms with polynomial time runtime complexity, whose output is always correct (Las Vegas algorithms), are said to be in ZPP, or zero-error probabilistic polynomial time algorithms.

- Lastly, the class of problems for which both YES and NO-instances are allowed to be identified with some error, commonly known as Monte Carlo algorithms, are in the complexity class called BPP, bounded-error probabilistic polynomial time.

# Randomized Algorithms - Examples

## Primality Testing

- Testing whether a number is prime or composite
- Modern cryptographic techniques securing important information depend on it.
- Primality Testing was one of the first randomized algorithms formally developed in the 1970s.
- Randomized algorithms are used to perform primality testing in order to avoid a brute force search, which would consist of a time consuming linear search of every prime number leading up to the number at hand.

# Randomized Algorithms - Examples

## Randomized Minimum Cut

- The Max-Flow Min-cut algorithm is another basic randomized algorithm applied on network flow and general graph problems.
- The goal is to find the smallest total weight of edges which, if removed, would disconnect the source from the sink in a max-flow network problem.
- Due to the fact that every edge must be checked in order to confidently assure that the right answer is found, randomized algorithms are developed.
- The famous randomized algorithm for a minimum cut in a graph is Ford-Fulkerson Algorithm.

# Randomized Algorithms - Examples

## Frievalds' Algorithm for Matrix Product Checking

- Given three $n \times n$ matrices, A, B, and C, a common problem is to verify whether $A \times B = C$. Since matrix multiplication is a rather costly process and checking the stated equality would require term-by-term comparisons of both sides of the equation.
- Randomized algorithms have been developed in order to avoid this brute force method of checking the equality.
- The algorithm is named after Rusins Frievalds, who realized that by using randomization, he could reduce the running time of this problem from brute force matrix multiplication using Strassen's algorithm, taking a runtime of $O(n^{2.37})$ to $O(kn^2)$, where k is the number of times the algorithm is executed.

# THANK YOU