CSC373

Week 5:
Dynamic Programming (contd)
Network Flow (start)

Nisarg Shah

Recap

- **Dynamic Programming Basics**
  - Optimal substructure property
  - Bellman equation
  - Top-down (memoization) vs bottom-up implementations

- **Dynamic Programming Examples**
  - Weighted interval scheduling
  - Knapsack problem
  - Single-source shortest paths
  - Chain matrix product
  - Edit distance (aka sequence alignment)

This Lecture

- Some more DP
  - Traveling salesman problem (TSP)

- Start of network flow
  - Problem statement
  - Ford-Fulkerson algorithm
  - Running time
  - Correctness

Traveling Salesman

- **Input**
  - Complete directed graph $G = (V, E)$
  - $d_{i,j} = $ distance from node i to node j

- **Output**
  - Minimum distance which needs to be traveled to start from some node v, visit every other node exactly once, and come back to v
    - That is, the minimum cost of a Hamiltonian cycle

Traveling Salesman

- **Approach**
  - Let's start at node $v_1 = 1$
    - It's a cycle, so the starting point does not matter
  - Want to visit the other nodes in some order, say $v_2, \ldots, v_n$
  - Total distance is $d_{1,v_2} + d_{v_2,v_3} + \cdots + d_{v_{n-1},v_n} + d_{v_n,1}$
    - Want to minimize this distance

- **Naïve solution**
  - Check all possible orderings
  - $(n-1)! = \Theta\left(\sqrt{n} \cdot \left(\frac{n}{e}\right)^n\right)$ (Stirling's approximation)

Traveling Salesman

- **DP Approach**
  - Consider v_n (the last node before returning to $v_1 = 1$)
    - If $v_n = c$
      - Find the optimal order of visiting nodes $\{2, \dots, n\}$ that ends at c
      - Need to keep track of the subset of nodes to be visited and the end node

  - $OPT[S, c]$ = minimum total travel distance when starting at 1, visiting each node in S exactly once, and ending at $c \in S$

  - Answer to the original problem:
    - $\min\limits_{c \in S} OPT[S, c] + d_{c,1}$, where $S = \{2, \dots, n\}$

Traveling Salesman

- **DP Approach**
  - To compute $OPT[S, c]$, we can condition over the vertex visited right before c in the optimal trip

- **Bellman equation**

$$OPT[S, c] = \min_{m \in S \setminus \{c\}} \left(OPT[S \setminus \{c\}, m] + d_{m,c} \right)$$

$$\text{Final solution} = \min_{c \in \{2, \dots, n\}} \left(OPT[\{2, \dots, n\}, c] + d_{c,1} \right)$$

- **Time:** $O(n \cdot 2^n)$ calls, $O(n)$ time per call $\Rightarrow O\left(n^2 \cdot 2^n\right)$
  - Much better than the naïve solution which has $(n/e)^n$

Traveling Salesman

- **Bellman equation**

$$OPT[S, c] = \min_{m \in S \setminus \{c\}} \left(OPT[S \setminus \{c\}, m] + d_{m,c} \right)$$

$$\text{Final solution} = \min_{c \in \{2, \ldots, n\}} OPT[\{2, \ldots, n\}, c] + d_{c,1}$$

- **Space complexity:** $O(n \cdot 2^n)$
  - ➢ But computing the optimal solution with $|S| = k$ only requires storing the optimal solutions with $|S| = k - 1$

- **Question:**
  - ➢ Using this observation, how much can we reduce the space complexity?

DP Concluding Remarks

- High-level steps in designing a DP algorithm
  - Focus on a single decision in optimal solution
    - Typically, the first/last decision
  - For each possible way of making that decision…
    - [Optimal substructure] Write the optimal solution of the problem in terms of the optimal solutions to subproblems
  - Generalize the problem…
    - …by looking at the type of subproblems needed
    - E.g., in the edit distance problem, we realize that we need to solve the problem for prefixes $(x_1, \dots, x_i)$ and $(y_1, \dots, y_j)$ for all (i, j)
  - Write the Bellman equation, cover your base cases
  - Think about optimizing the running time/space using tricks
    - Often easier in the bottom-up implementation

Network Flow

Network Flow

- **Input**
  - A directed graph $G = (V, E)$
  - Edge capacities $c : E \rightarrow \mathbb{R}_{\geq 0}$
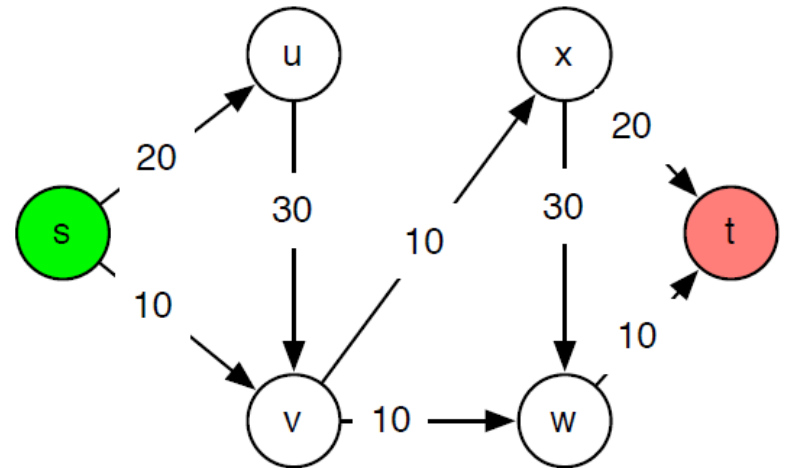 - Source node s, target node t

- **Output**
  - Maximum "flow" from s to t
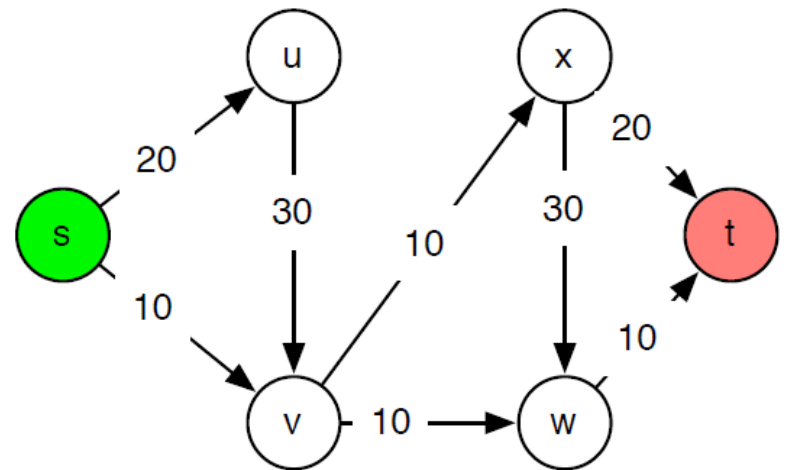
Network Flow
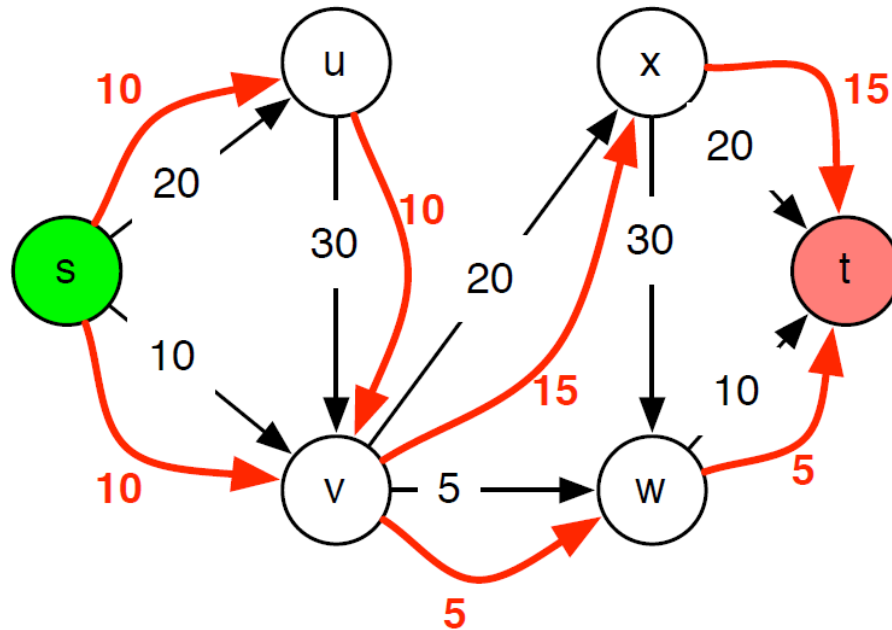
- ## Assumptions
  - ➤ No edges enter s
  - ➤ No edges leave t
  - ➤ Edge capacity $c(e)$ is a non-negative integer
    - ○ Later, we'll see what happens when $c(e)$ can be a rational or irrational number

Network Flow

- Flow
  - An s-t flow is a function $f: E \rightarrow \mathbb{R}_{\geq 0}$
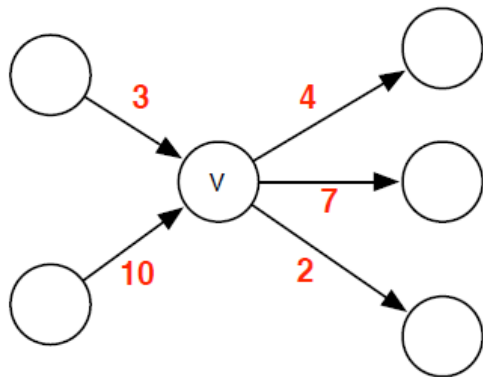 - Intuitively, $f(e)$ is the "amount of material" carried on edge e

Network Flow

- Constraints on flow f

    1. Respecting capacities

        $$\forall e \in E : 0 \leq f(e) \leq c(e)$$

    2. Flow conservation

        $$\forall v \in V \setminus \{s, t\} : \sum_{e \text{ entering } v} f(e) = \sum_{e \text{ leaving } v} f(e)$$
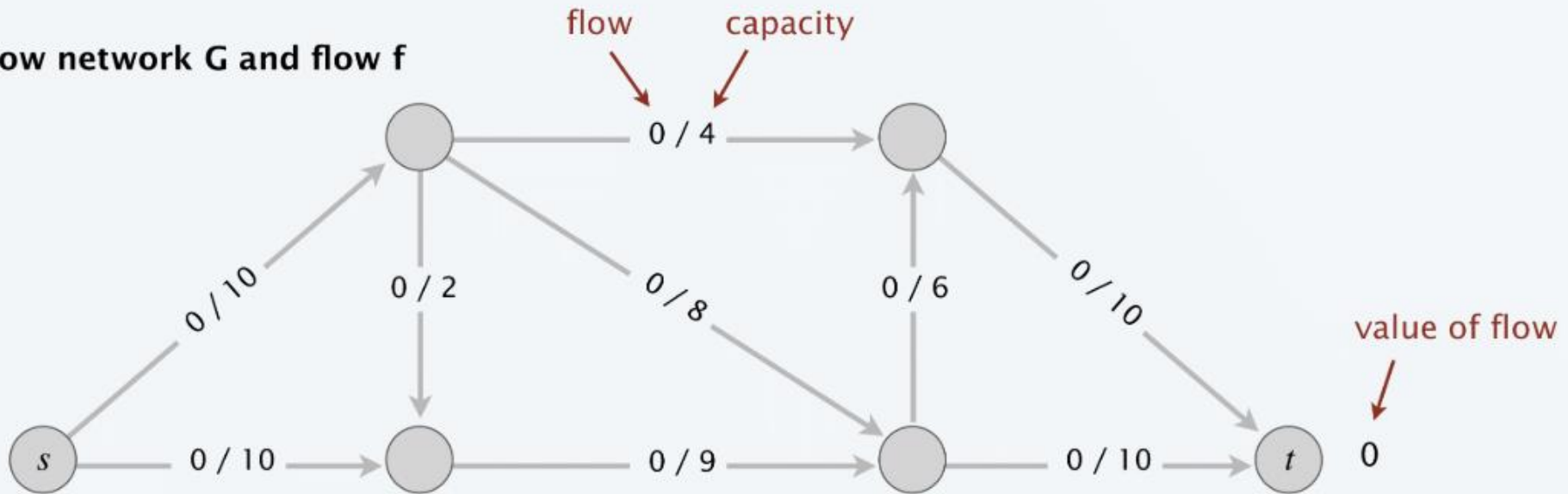

Flow in = flow out at every node other than s and t

Network Flow

- $f^{in}(v) = \sum_{e \text{ entering } v} f(e)$
- $f^{out}(v) = \sum_{e \text{ leaving } v} f(e)$

- Value of flow f is $v(f) = f^{out}(s) = f^{in}(t)$
  - Q: Why is $f^{out}(s) = f^{in}(t)$?

- Restating the problem:
  - Given a directed graph $G = (V, E)$ with edge capacities $c: E \to \mathbb{R}_{\geq 0}$, find a flow f^* with the maximum value.

First Attempt

- **A natural greedy approach**

  1. Start from zero flow ($f(e) = 0$ for each e).

  2. While there exists an s-t path P in G such that $f(e) < c(e)$ for each $e \in P$

     a. Find any such path P

     b. Compute $\Delta = \min_{e \in P}\big(c(e) - f(e)\big)$

     c. Increase the flow on each edge $e \in P$ by Δ

- Note
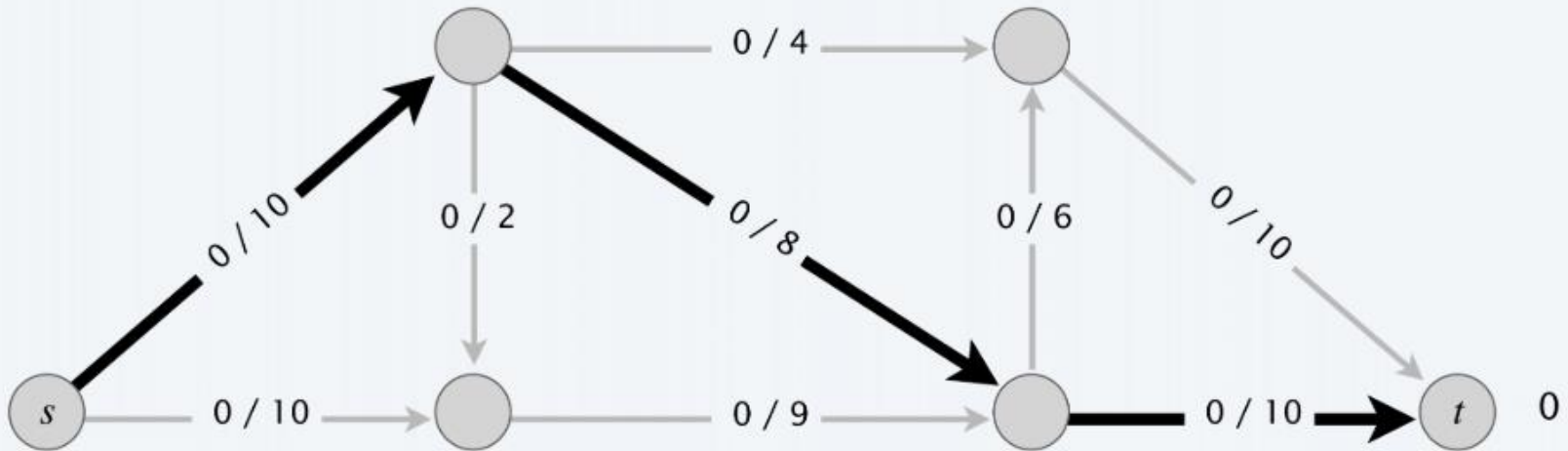 - Capacity and flow conservation constraints remain satisfied
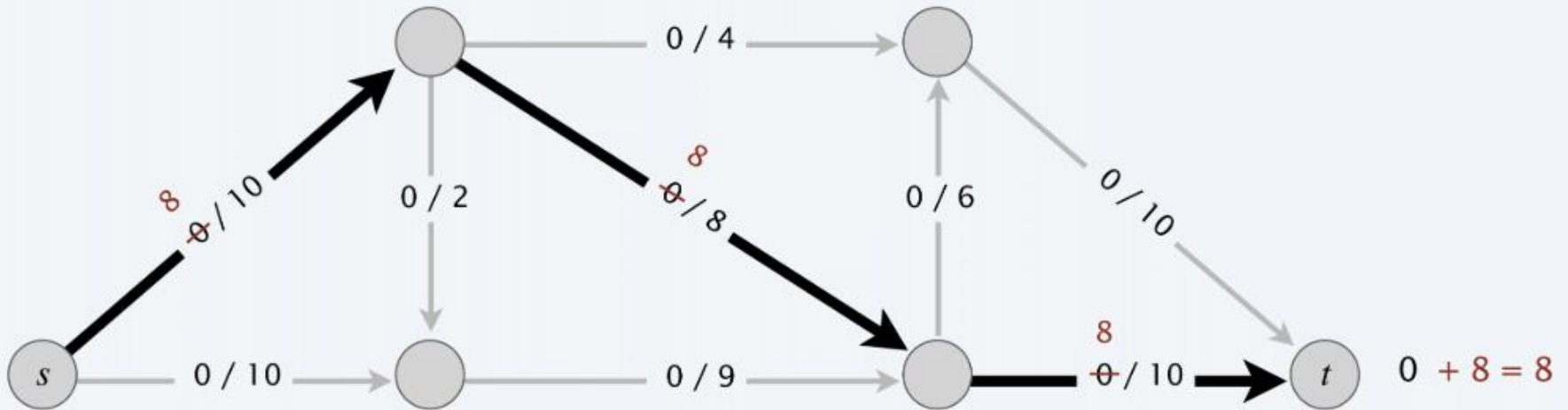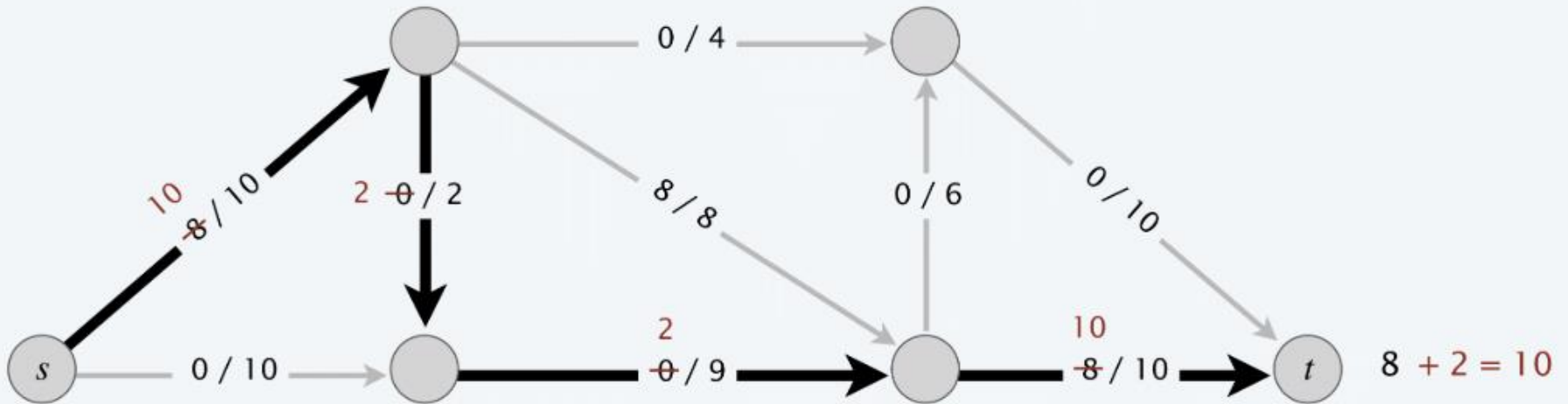
First Attempt

flow network G and flow f

flow   capacity

0 / 4

0 / 10   0 / 2   0 / 8   0 / 6   0 / 10

value of flow

s   0 / 10   0 / 9   0 / 10   t   0

First Attempt

flow network G and flow f

First Attempt

flow network G and flow f

0 + 8 = 8

First Attempt

flow network G and flow f

$8 + 2 = 10$

First Attempt

flow network G and flow f
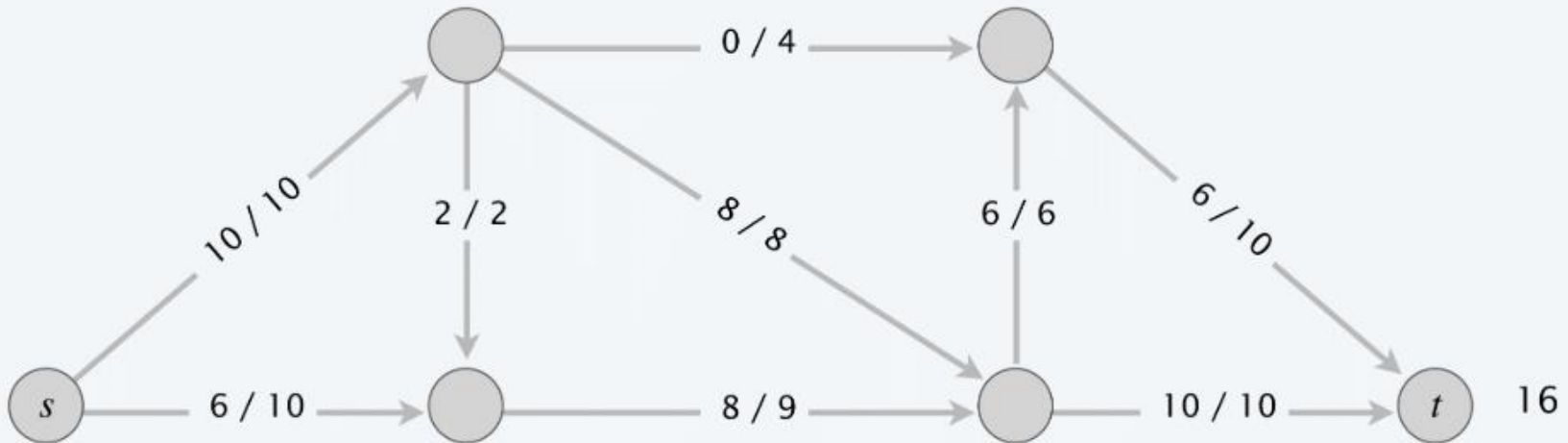
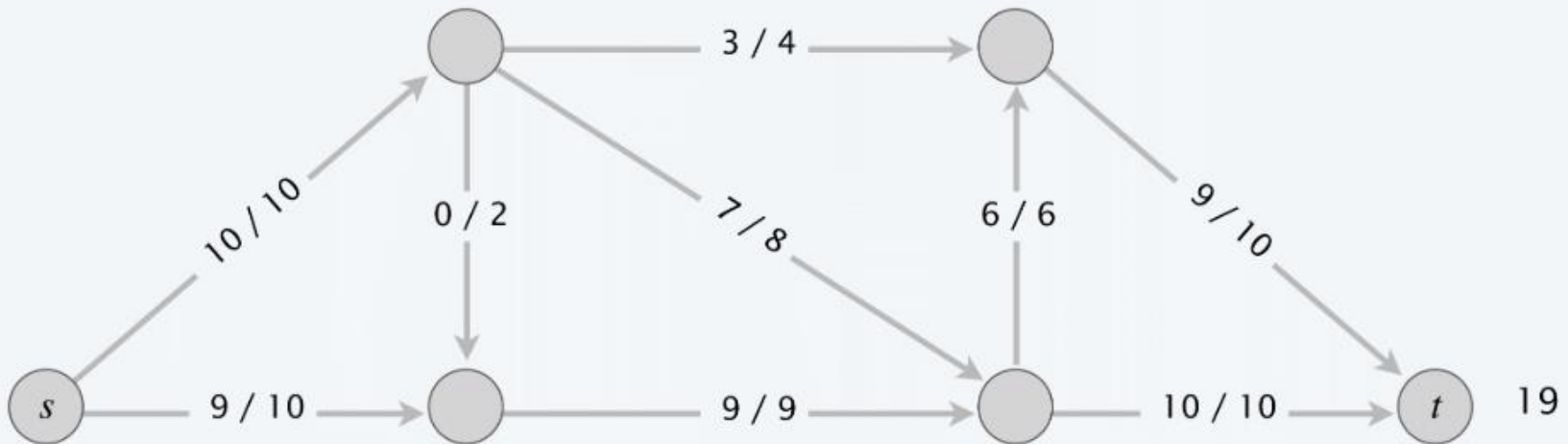10 + 6 = 16
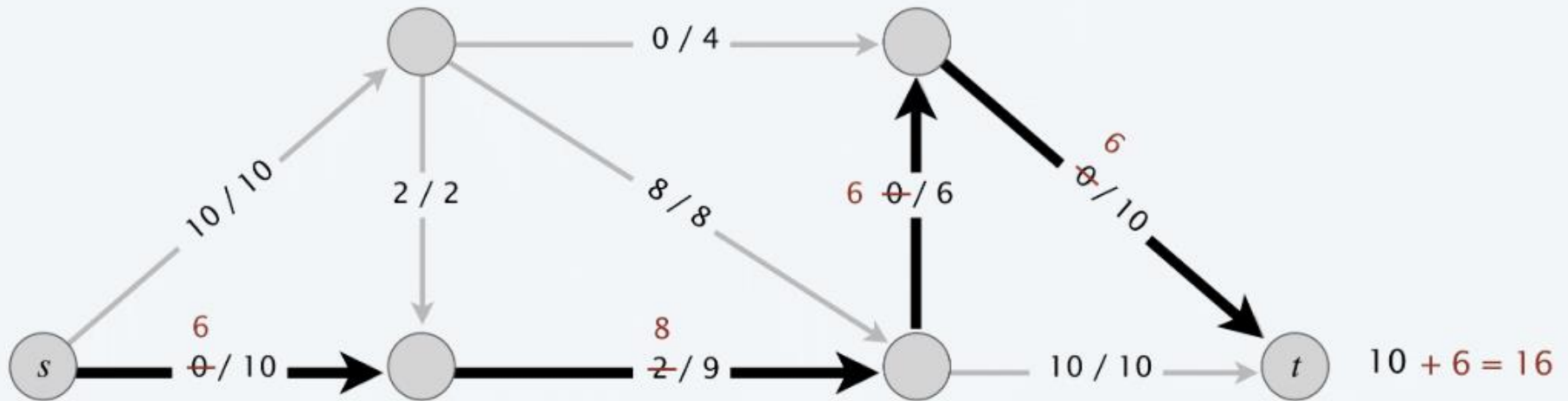
First Attempt

ending flow value = 16

flow network G and flow f

First Attempt
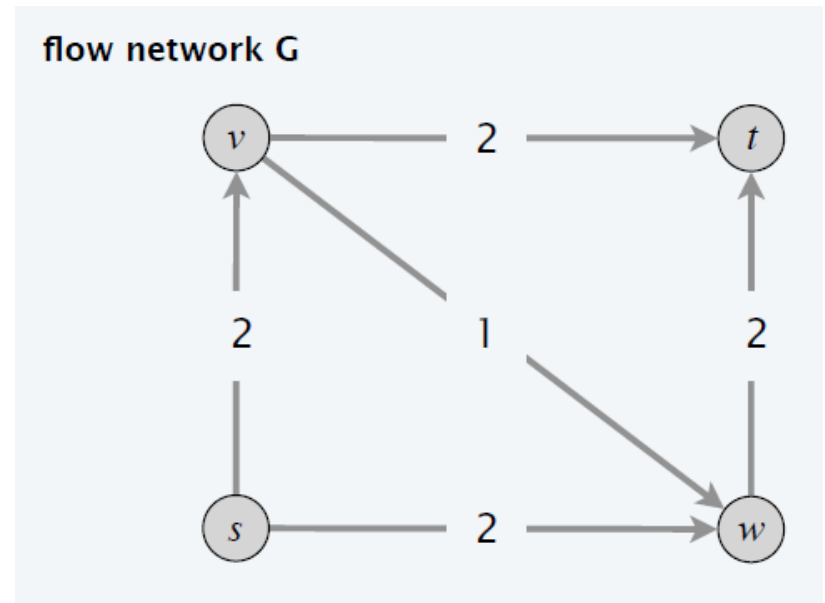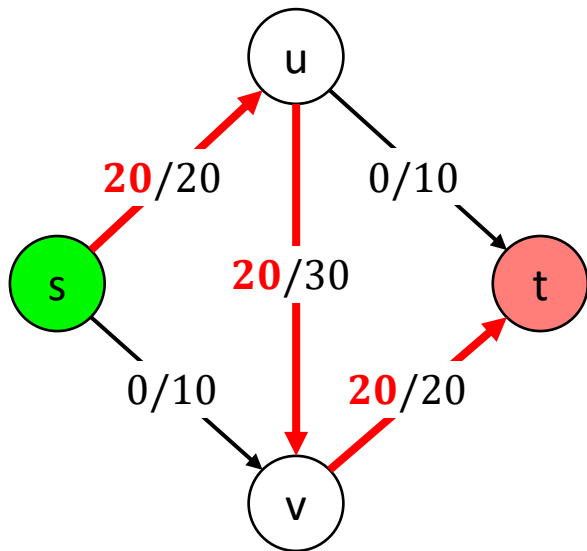

but max–flow value = 19

flow network G and flow f

First Attempt

- Q: Why does the simple greedy approach fail?

- A: Because once it increases the flow on an edge, it is not allowed to decrease it ever in the future.

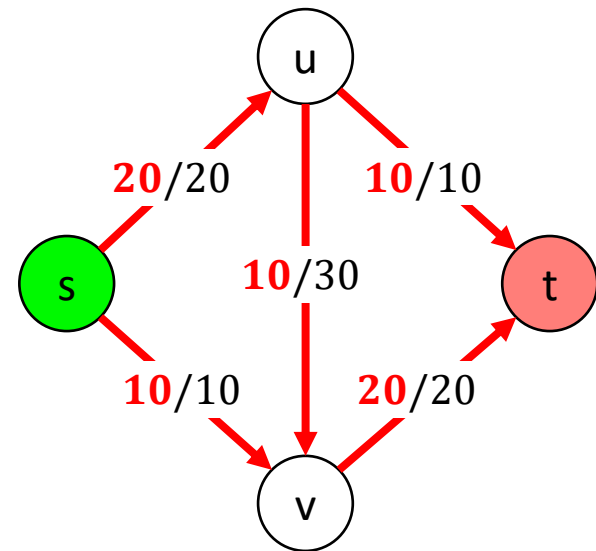- Need a way to "reverse" bad decisions

flow network G

Reversing Bad Decisions

Suppose we start by sending
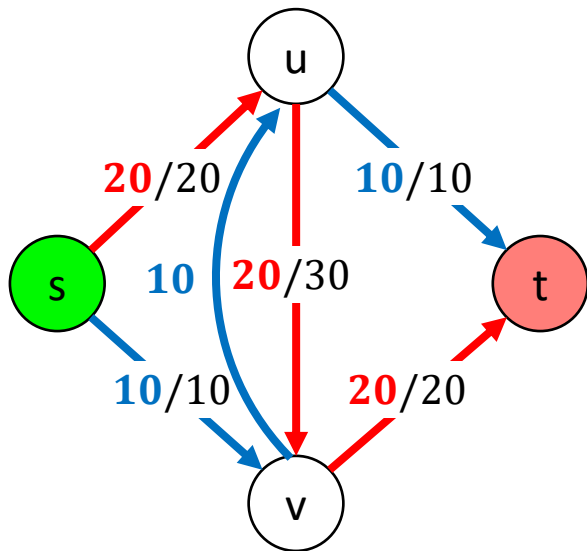20 units of flow along this path

But the optimal configuration requires
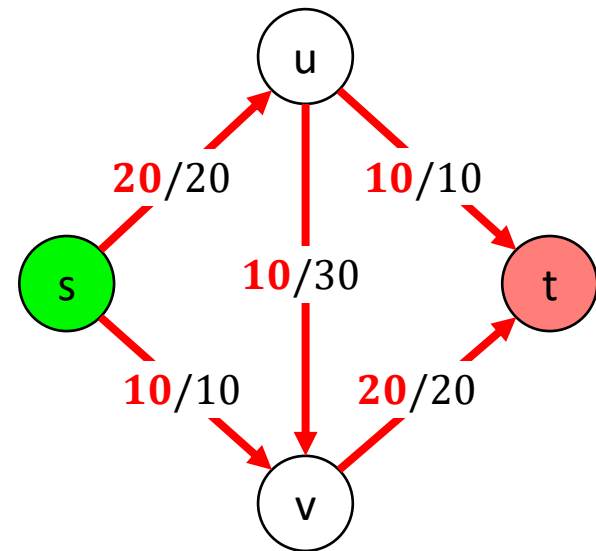10 fewer units of flow on $u \rightarrow v$

Reversing Bad Decisions

We can essentially send a "reverse" flow of 10 units along $v \to u$

So now we get this optimal flow

Residual Graph

- Suppose the current flow is f

- Define the residual graph G_f of flow f
  - G_f has the same vertices as G
  - For each edge e $= (u, v)$ in G, G_f has at most two edges
    - Forward edge $e = (u, v)$ with capacity $c(e) - f(e)$
      - We can send this much additional flow on e
    - Reverse edge $e^{rev} = (v, u)$ with capacity $f(e)$
      - The maximum "reverse" flow we can send is the maximum amount by which we can reduce flow on e, which is $f(e)$
    - We only really add edges of capacity > 0

Residual Graph

- Example!
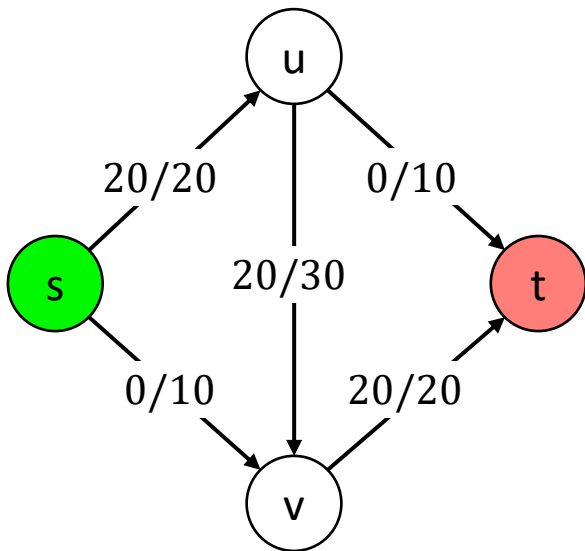
Flow f
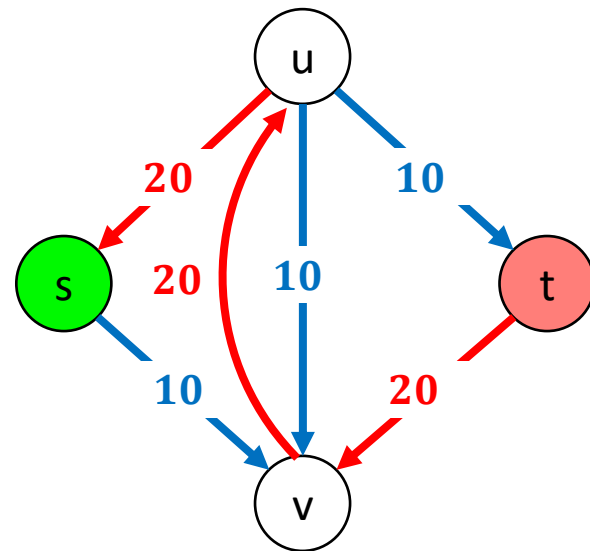
Residual graph G_f

Augmenting Paths

- Let P be an s-t path in the residual graph G_f

- Let bottleneck(P, f) be the smallest capacity across all edges in P

- "Augment" flow f by "sending" bottleneck(P, f) units of flow along P
  - What does it mean to send x units of flow along P?
  - For each forward edge $e \in P$, increase the flow on e by x
  - For each reverse edge $e^{rev} \in P$, decrease the flow on e by x

Residual Graph

- Example!

Flow f
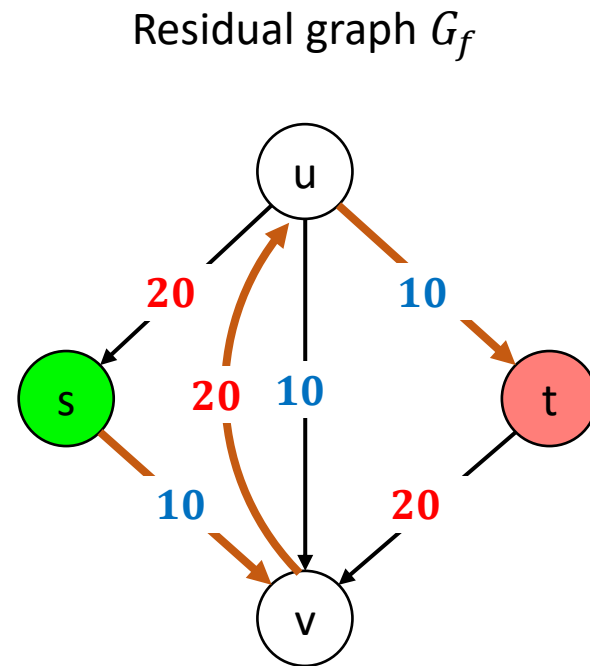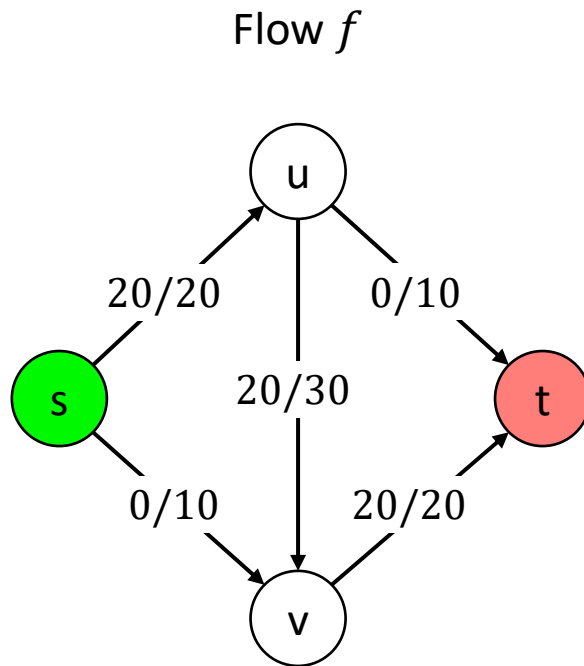
Residual graph G_f

Path P → send flow = bottleneck = 10

Residual Graph

- Example!

New flow f
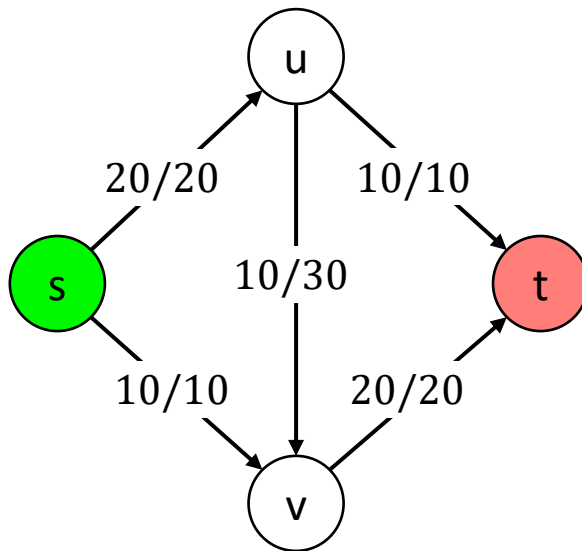
New residual graph G_f
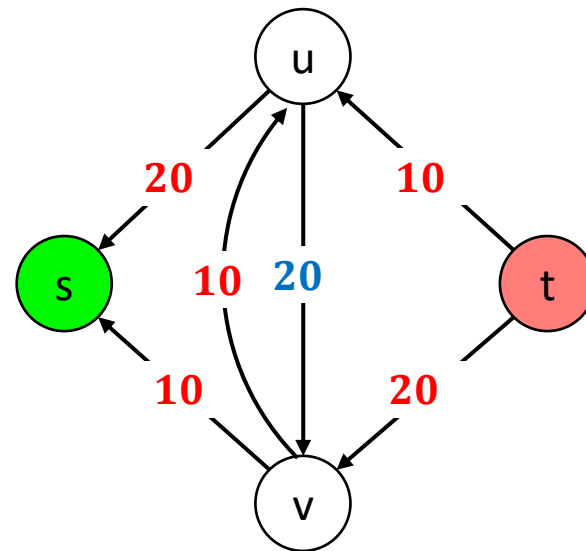
No s-t path because no outgoing edge from s

Augmenting Paths

- Let's argue that the new flow is a valid flow

- Capacity constraints (easy):

  ➤ If we increase flow on e, we can do so by at most the capacity of forward edge e in G_f, which is $c(e) - f(e)$
    - So, the new flow can be at most $f(e) + \left(c(e) - f(e)\right) = c(e)$

  ➤ If we decrease flow on e, we can do so by at most the capacity of reverse edge e^{rev} in G_f, which is $f(e)$
    - So, the new flow is at least $f(e) - f(e) = 0$

Augmenting Paths

- Let's argue that the new flow is a valid flow

- Flow conservation (a bit trickier):
  - Each node on the path (except s and t) has exactly two incident edges
    - Both forward / both reverse ⇒ one is incoming, one is outgoing
      - Flow increased on both or decreased on both
    - One forward, one reverse ⇒ both incoming / both outgoing
      - Flow increased on one but decreased on the other
    - In each case, net flow remains 0

Edge directions as in G

s $\xrightarrow{+x}$ ○ $\xrightarrow{+x}$ ○ $\xleftarrow{-x}$ ○ $\xleftarrow{-x}$ ○ $\xrightarrow{+x}$ t
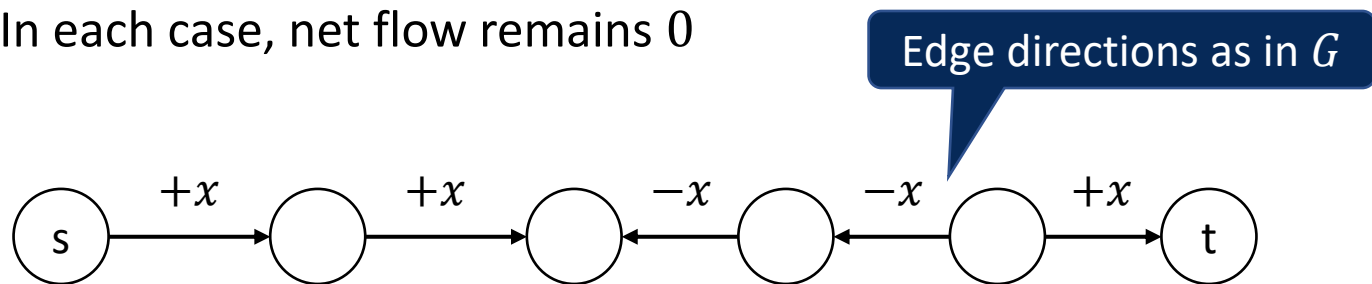
Ford-Fulkerson Algorithm

MaxFlow(G):

    // initialize:
    Set $f(e) = 0$ for all e in G

    // while there is an s-t path in G_f:
    While $P =$ FindPath(s,t,Residual(G,f))!=None:
      $f =$ Augment(f,P)
      UpdateResidual(G,f)
    EndWhile

    Return f

Ford-Fulkerson Algorithm

- Running time:
  - #Augmentations:
    - At every step, flow and capacities remain integers
    - For path P in G_f, bottleneck$(P, f) > 0$ implies bottleneck$(P, f) \geq 1$
    - Each augmentation increases flow by at least 1
    - Max flow (hence max #augmentations) is at most $C = \sum_{e \text{ leaving } s} c(e)$

  - Time to perform an augmentation:
    - G_f has n vertices and at most $2m$ edges
    - Finding P, computing bottleneck(P, f), updating G_f
      - $O(m + n)$ time

  - Total time: $O((m + n) \cdot C)$

Ford-Fulkerson Algorithm

- Total time: $O((m+n) \cdot C)$

  - This is pseudo-polynomial time, but NOT polynomial time

  - The value of C can be exponentially large in the input length (the number of bits required to write down the edge capacities)

  - Note: While we assumed integer capacities, we know that the algorithm must always terminate even with rational capacities.

    - Why?

    - With irrational capacities, there is an example in which the algorithm never terminates.

- Q: Can we convert this to polynomial time?

Ford-Fulkerson Algorithm

- **Q:** Can we convert this to polynomial time?
  - ➤ Not if we choose an *arbitrary* path in G_f at each step
  - ➤ In the graph below, we might end up repeatedly sending 1 unit of flow across $a \rightarrow b$ and then reversing it
    - ○ Takes X steps, which can be exponential in the input length
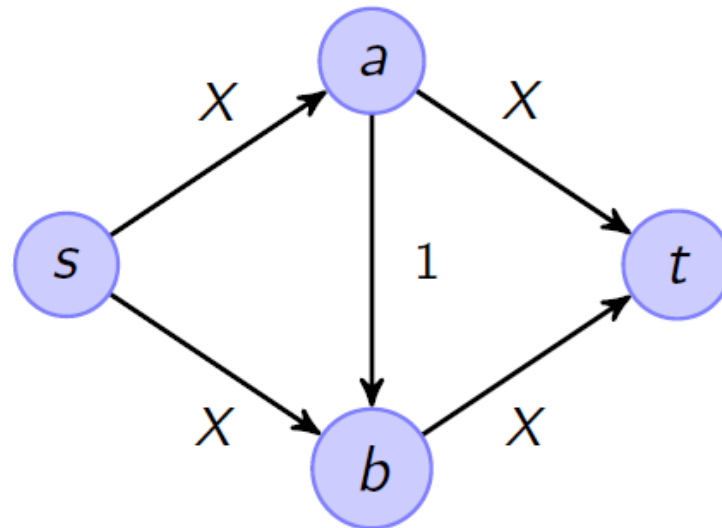
A Brief Note

- Two quantities
  - *Number of integers* provided as input
  - *Total number of bits* provided as input

- Running time
  - Strongly polynomial time
    - #ops polynomial in #integers but not dependent on #bits
    - Running time still polynomial in #bits
  - Weakly polynomial time
    - #ops polynomial in #bits
  - Pseudo-polynomial time
    - #ops polynomial in #bits *if input was written in unary* (i.e., polynomial in the values of the input integers)

Ford-Fulkerson Algorithm

- **Ways to achieve polynomial time**

  - ➢ Find the **maximum bottleneck capacity** augmenting path
    - ○ Runs in $O(m^2 \cdot \log C)$ operations
      - "Weakly polynomial time"

  - ➢ Find the **shortest** augmenting path using BFS
    - ○ Edmonds-Karp algorithm
    - ○ Runs in $O(nm^2)$ operations
      - "Strongly polynomial time"
    - ○ Can be found in CLRS

  - ➢ …

Max Flow Problem

- Race to reduce the running time
  - 1972: $O(n\,m^2)$ Edmonds-Karp
  - 1980: $O(n\,m \log^2 n)$ Galil-Namaad
  - 1983: $O(n\,m \log n)$ Sleator-Tarjan
  - 1986: $O\left(n\,m \log\left(n^2/m\right)\right)$ Goldberg-Tarjan
  - 1992: $O\left(n\,m + n^{2+\epsilon}\right)$ King-Rao-Tarjan
  - 1996: $O\left(n\,m \dfrac{\log n}{\log\ {}^{m}/_{n \log n}}\right)$ King-Rao-Tarjan
    - Note: These are $O(n\,m)$ when $m = \omega(n)$
  - 2013: $O(n\,m)$ Orlin
    - Breakthrough!
  - 2021: $O\left((m + n^{1.5}) \cdot \log X\right)$, where $X = $ max edge capacity
    - Breakthrough based on very heavy techniques!

Back to Ford-Fulkerson

- We argued that the algorithm must terminate, and must terminate in $O\big((m+n)\cdot C\big)$ time

- But we didn't argue correctness yet, i.e., the algorithm must terminate with the optimal flow

Recall: Ford-Fulkerson

```
MaxFlow(G):

    // initialize:
    Set f(e) = 0 for all e in G

    // while there is an s-t path in G_f:
    While P = FindPath(s,t,Residual(G,f))!=None:
        f = Augment(f,P)
        UpdateResidual(G,f)
    EndWhile

    Return f
```

Recall: Notation

- f = flow, s = source, t = target

- f^{out}, f^{in}
  - For a node u: $f^{out}(u), f^{in}(u)$ = total flow out of and into u
  - For a set of nodes X: $f^{out}(X), f^{in}(X)$ defined similarly

- Constraints
  - Capacity: $0 \leq f(e) \leq c(e)$
  - Flow conservation: $f^{out}(u) = f^{in}(u)$ for all $u \neq s, t$

- $v(f) = f^{out}(s) = f^{in}(t)$ = value of the flow

Cuts and Cut Capacities

- (A, B) is an *s-t* cut if it is a partition of vertex set V (i.e., $A \cup B = V$, $A \cap B = \emptyset$) with $s \in A$ and $t \in B$

- Its capacity, denoted $cap(A, B)$, is the sum of capacities of edges *leaving A*

s-t cut

A

B = V \ A

s

—10—→

←—2—

—3—→

—12—→
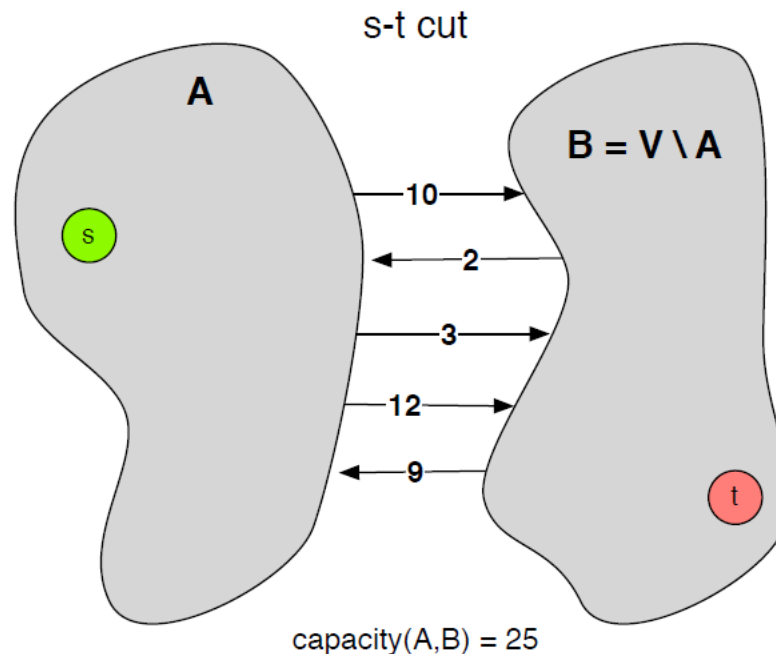
←—9—

t

capacity(A,B) = 25
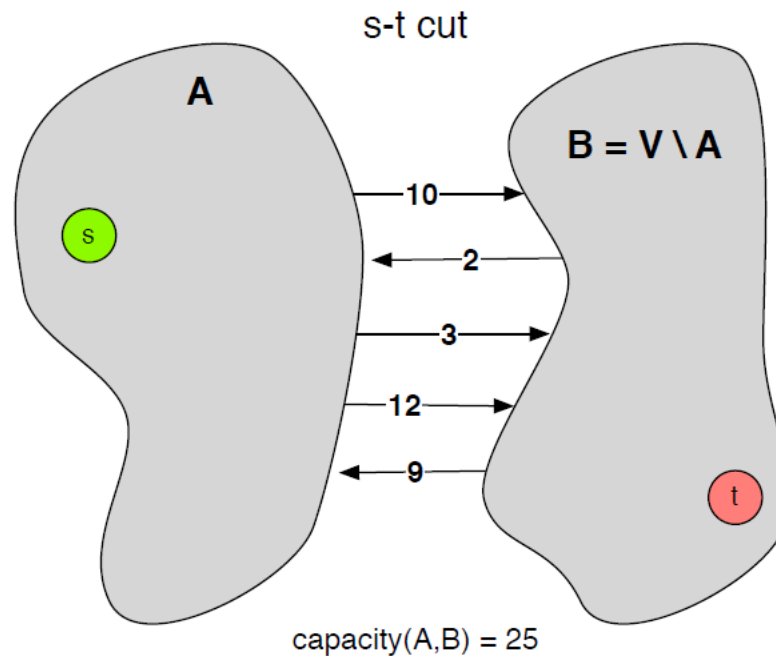
Cuts and Flows

- **Theorem:** For any flow f and any $s\text{-}t$ cut (A, B),
$$v(f) = f^{out}(A) - f^{in}(A)$$

- **Proof (on the board):** Just take a sum of the flow conservation constraint over all nodes in A

s-t cut

A

B = V \ A

s

t

—10—→

←—2—

—3—→

—12—→

←—9—

capacity(A,B) = 25

Cuts and Flows

- **Theorem:** For any flow f and any s-t cut (A, B),

$$v(f) \leq cap(A, B)$$

- **Proof:**

$$v(f) = f^{out}(A) - f^{in}(A)$$

$$\leq f^{out}(A)$$

$$= \sum_{e \text{ leaving } A} f(e)$$

$$\leq \sum_{e \text{ leaving } A} c(e)$$

$$= cap(A, B)$$

Cuts and Flows

- Theorem: For any flow f and any s-t cut (A, B),

$$v(f) \leq cap(A, B)$$

- Hence, $\max\limits_{f} v(f) \leq \min_{(A,B)} cap(A, B)$

  ➢ Max value of any flow $\leq$ min capacity of any s-t cut

- We will now prove:
 ➢ Value of flow generated by Ford-Fulkerson = capacity of _some_ cut

- Implications
  ➢ 1) Max flow = min cut
  ➢ 2) Ford-Fulkerson generates max flow.
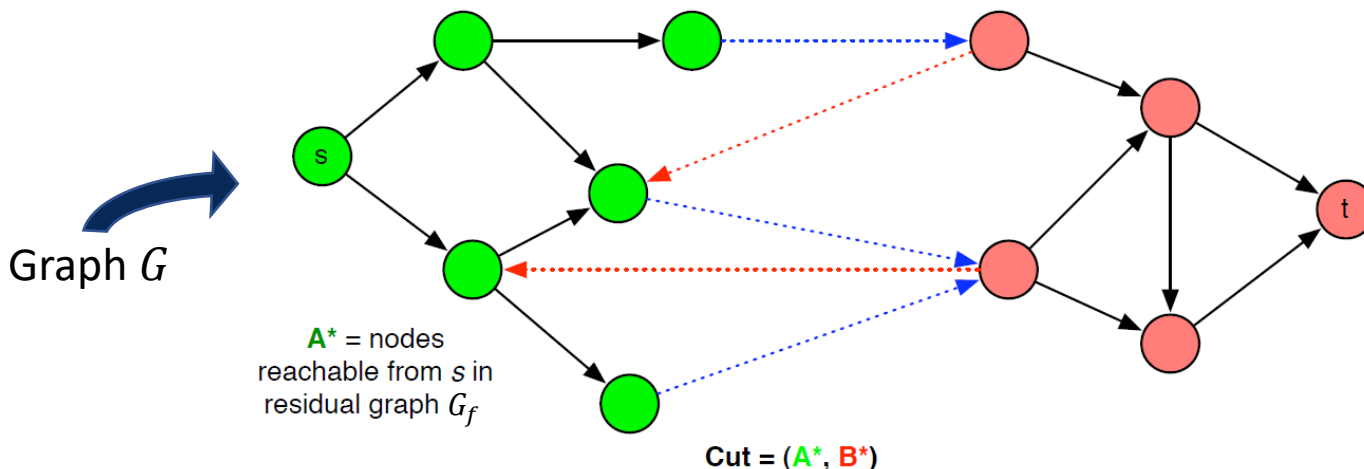
Cuts and Flows

- **Theorem:** Ford-Fulkerson finds maximum flow.
- **Proof:**
  - ➢ f = flow returned by Ford-Fulkerson
  - ➢ A^* = nodes reachable from s in G_f
  - ➢ B^* = remaining nodes $V \setminus A^*$
  - ➢ Note: We look at the residual graph G_f, but define the cut in G

Graph G

A* = nodes reachable from s in residual graph G_f

Cut = (A*, B*)
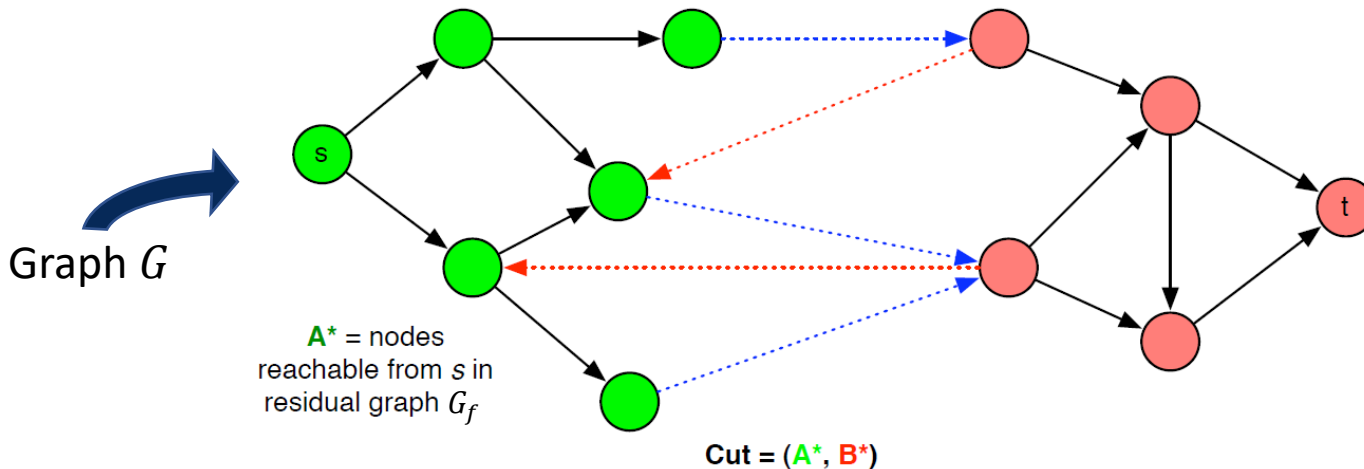
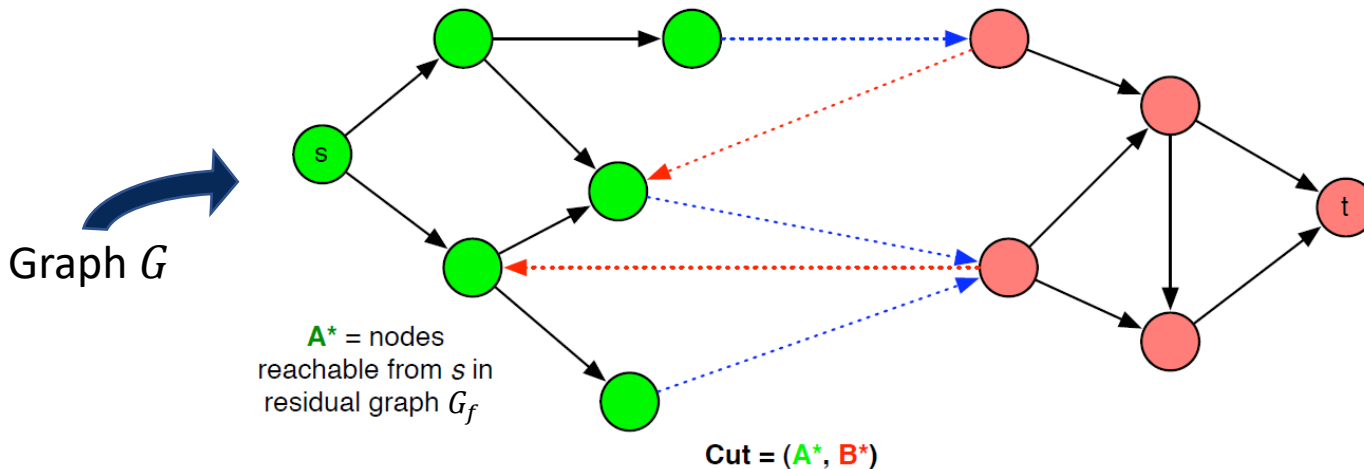# Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.

- Proof:
  - Claim: (A^*, B^*) is a valid cut
    - $s \in A^*$ by definition
    - $t \in B^*$ because when Ford-Fulkerson terminates, there are no s-t paths in G_f, so $t \notin A^*$

Graph G

A* = nodes reachable from s in residual graph G_f

Cut = (A*, B*)

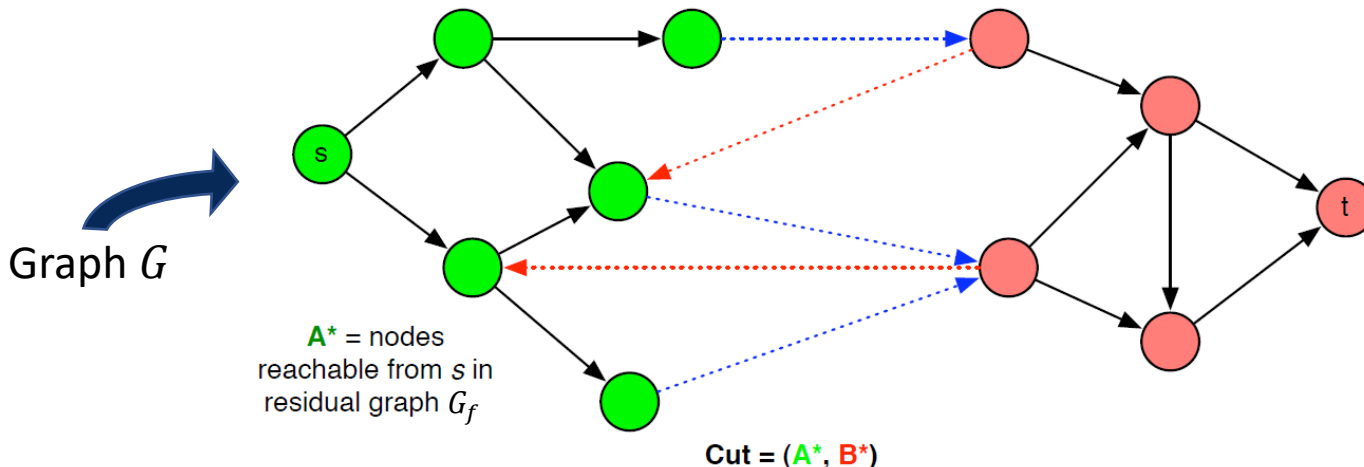# Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - Blue edges = edges going out of A^* in G
  - Red edges = edges coming into A^* in G

Graph G

A* = nodes
reachable from s in
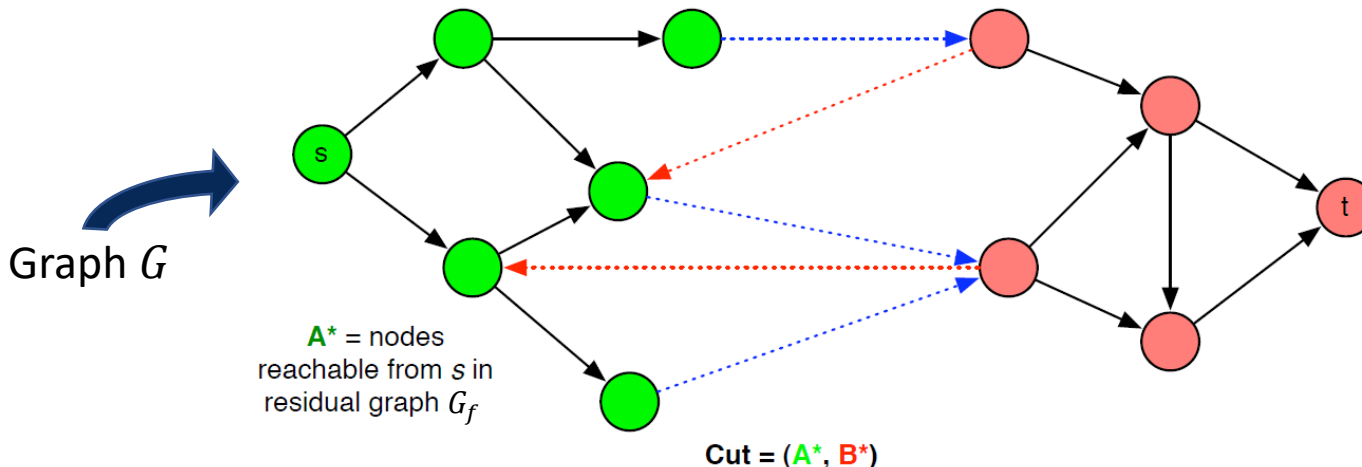residual graph G_f

Cut = (A*, B*)

Cuts and Flows

- **Theorem:** Ford-Fulkerson finds maximum flow.
- **Proof:**
  - Each blue edge (u, v) must be saturated
    - Otherwise G_f would have its forward edge (u, v) and then $v \in A^*$
  - Each red edge (v, u) must have zero flow
    - Otherwise G_f would have its reverse edge (u, v) and then $v \in A^*$

Graph G

A* = nodes reachable from s in residual graph G_f

Cut = (A*, B*)

Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - Each blue edge (u, v) must be saturated $\Rightarrow f^{out}(A^*) = cap(A^*, B^*)$
  - Each red edge (v, u) must have zero flow $\Rightarrow f^{in}(A^*) = 0$
  - So $v(f) = f^{out}(A^*) - f^{in}(A^*) = cap(A^*, B^*)$ ∎

Graph G

A* = nodes
reachable from s in
residual graph G_f

Cut = (A*, B*)

Max Flow - Min Cut

- **Max Flow-Min Cut Theorem:**
  In any graph, the value of the maximum flow is equal to the capacity of the minimum cut.

- Our proof already gives an algorithm to find a min cut
  - Run Ford-Fulkerson to find a max flow f
  - Construct its residual graph G_f
  - Let $A^* =$ set of all nodes reachable from s in G_f
    - Easy to compute using BFS
  - Then $(A^*, V \setminus A^*)$ is a min cut

Poll

<u>Question</u>

- There is a network G with positive integer edge capacities.

- You run Ford-Fulkerson.

- It finds an augmenting path with bottleneck capacity 1, and after that iteration, it terminates with a final flow value of 1.

- Which of the following statement(s) must be correct about G?

(a) G has a single s-t path.

(b) G has an edge e such that all s-t paths go through e.

(c) The minimum cut capacity in G is greater than 1.

(d) The minimum cut capacity in G is less than 1.

Why Study Flow Networks?

- Unlike divide-and-conquer, greedy, or DP, this doesn't seem like an algorithmic framework
  - It seems more like a single problem

- Turns out that many problems can be reduced to this versatile single problem

- Next lecture!