CSC373

Week 4:
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

This Lecture

- Some more DP
  - Edit distance (aka sequence alignment)
  - Traveling salesman problem (TSP)

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

Edit Distance

- Edit distance (aka sequence alignment) problem
  - How similar are strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$?

- Suppose we can delete or replace symbols
  - We can do these operations on any symbol in either string
  - How many deletions & replacements does it take to match the two strings?

Edit Distance

- Example: ocurrance vs occurrence

o	c	u	r	r	a	n	c	e	–
o	c	c	u	r	r	e	n	c	e

6 replacements, 1 deletion

o	c	–	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 replacement, 1 deletion

Edit Distance

- Edit distance problem
  - Input
    - Strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$
    - Cost $d(a)$ of deleting symbol a
    - Cost $r(a, b)$ of replacing symbol a with b
      - Assume r is symmetric, so $r(a, b) = r(b, a)$
  - Goal
    - Compute the minimum total cost for matching the two strings

- Optimal substructure?
  - Want to delete/replace at one end and recurse

Edit Distance

- Optimal substructure
  - Goal: match $x_1, \dots, x_m$ and $y_1, \dots, y_n$
  - Consider the last symbols x_m and y_n
  - Three options:
    - Delete x_m, and optimally match $x_1, \dots, x_{m-1}$ and $y_1, \dots, y_n$
    - Delete y_n, and optimally match $x_1, \dots, x_m$ and $y_1, \dots, y_{n-1}$
    - Match x_m and y_n, and optimally match $x_1, \dots, x_{m-1}$ and $y_1, \dots, y_{n-1}$

  - Hence in the DP, we need to compute the optimal solutions for matching $x_1, \dots, x_i$ with $y_1, \dots, y_j$ for all (i, j)

Edit Distance

- $E[i,j]$ = edit distance between $x_1, \dots, x_i$ and $y_1, \dots, y_j$
- Bellman equation

$$E[i,j] = \begin{cases} 0 & \text{if } i = j = 0 \\ d(y_j) + E[i, j-1] & \text{if } i = 0 \wedge j > 0 \\ d(x_i) + E[i-1, j] & \text{if } i > 0 \wedge j = 0 \\ \min\{A, B, C\} & \text{otherwise} \end{cases}$$

where

$A = d(x_i) + E[i-1, j], B = d(y_j) + E[i, j-1]$

$C = r(x_i, y_j) + E[i-1, j-1]$

- $O(n \cdot m)$ time, $O(n \cdot m)$ space

Edit Distance

$$E[i,j] = \begin{cases} 0 & \text{if } i = j = 0 \\ d(y_j) + E[i, j-1] & \text{if } i = 0 \wedge j > 0 \\ d(x_i) + E[i-1, j] & \text{if } i > 0 \wedge j = 0 \\ \min\{A, B, C\} & \text{otherwise} \end{cases}$$

where

$A = d(x_i) + E[i-1, j], B = d(y_j) + E[i, j-1]$

$C = r(x_i, y_j) + E[i-1, j-1]$

- Space complexity can be improved to $O(n+m)$
  - To compute $E[\cdot, j]$, we only need $E[\cdot, j-1]$ stored
  - So we can forget $E[\cdot, j]$ as soon as we reach $j+2$
  - But this is not enough if we want to compute the actual solution (sequence of operations)

Hirschberg's Algorithm

This slide is not in the scope of the course

- The optimal solution can be computed in $O(n \cdot m)$ time and $O(n + m)$ space too!

Programming Techniques — G. Manacher, Editor

A Linear Space Algorithm for Computing Maximal Common Subsequences

D.S. Hirschberg
Princeton University

The problem of finding a longest common subsequence of two strings has been solved in quadratic time and space. An algorithm is presented which will solve this problem in quadratic time and in linear space.

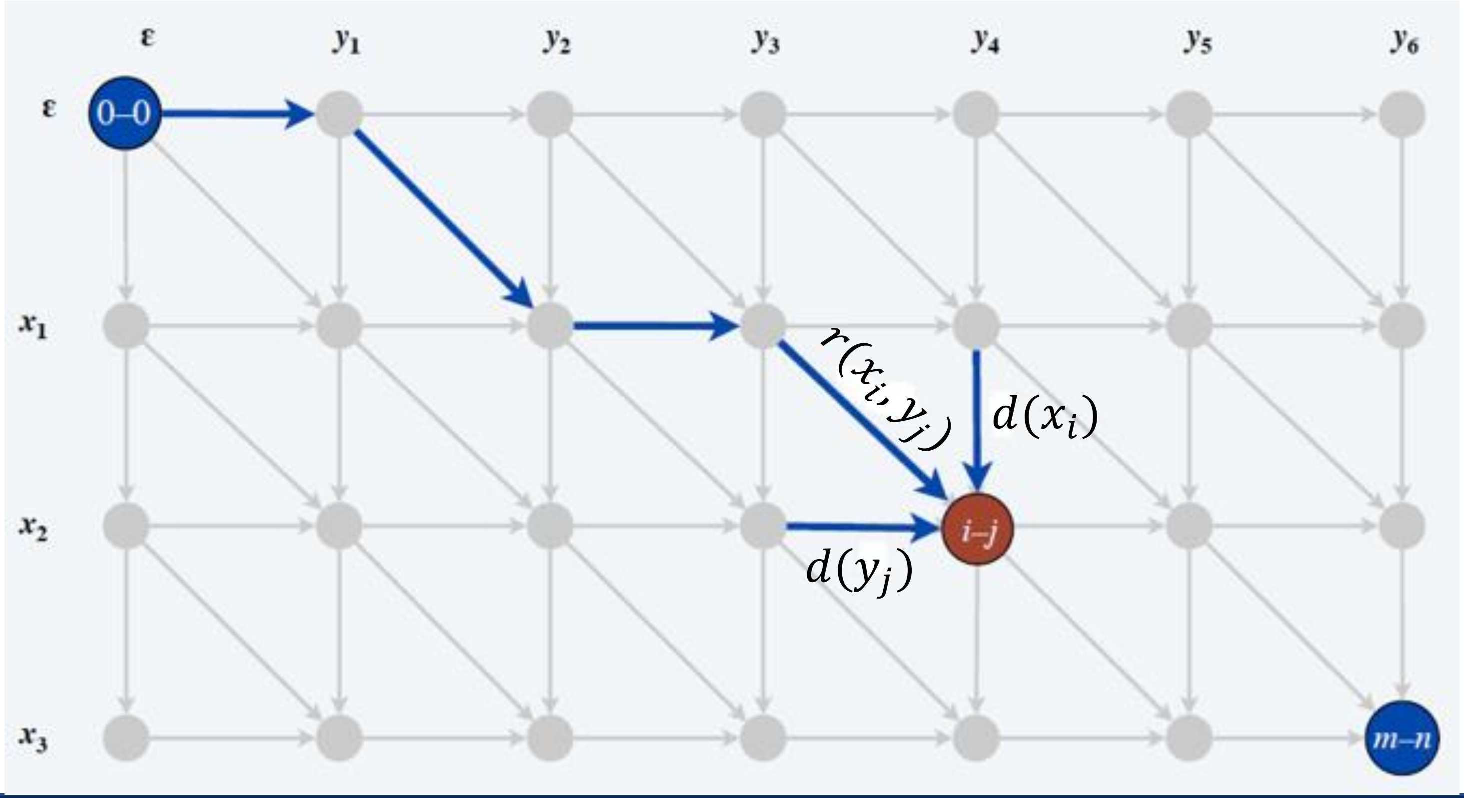
Key Words and Phrases: subsequence, longest common subsequence, string correction, editing

CR Categories: 3.63, 3.73, 3.79, 4.22, 5.25

Hirschberg’s Algorithm

This slide is not in the scope of the course

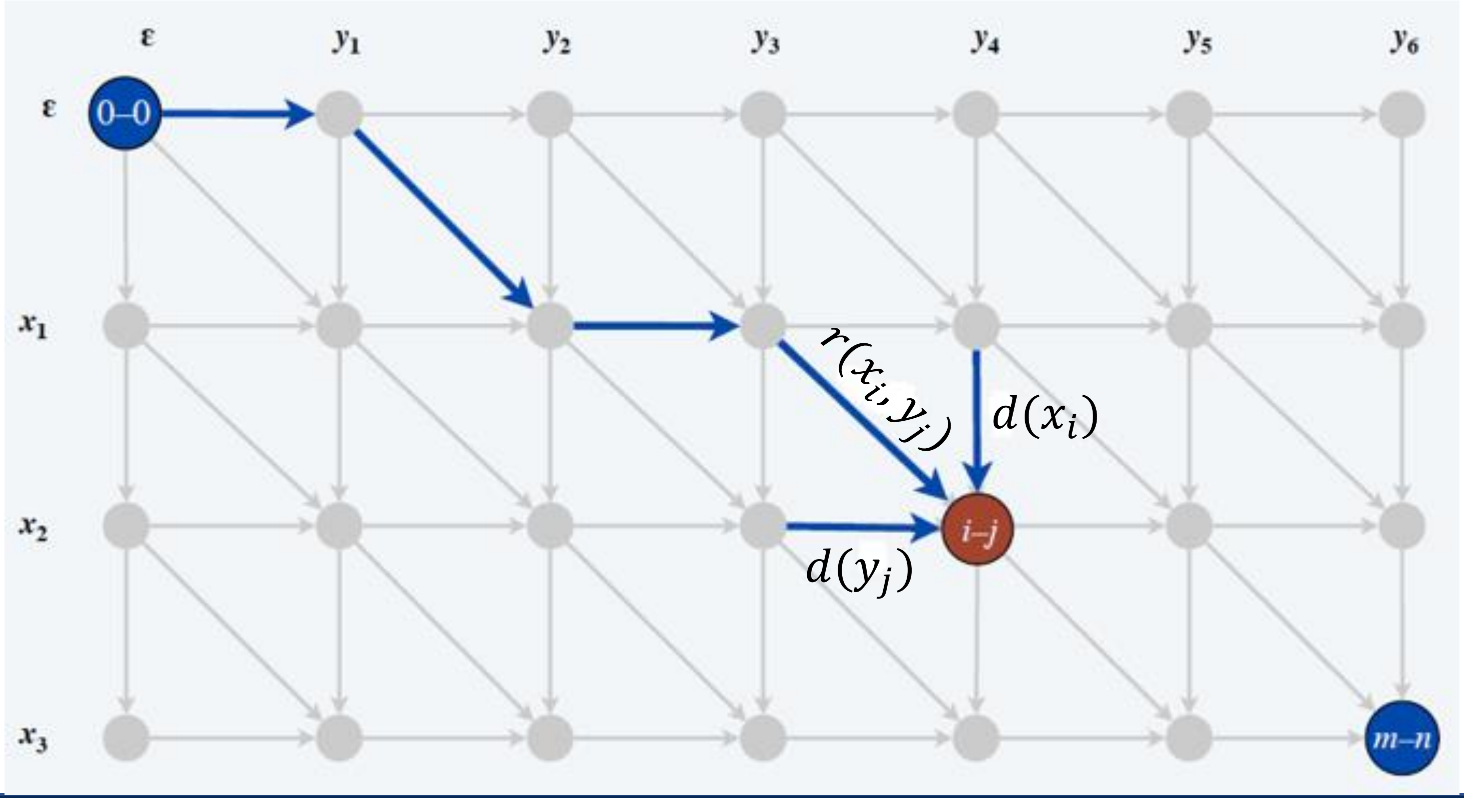
- Key idea nicely combines divide & conquer with DP
- Edit distance graph



Hirschberg's Algorithm

This slide is not in the scope of the course

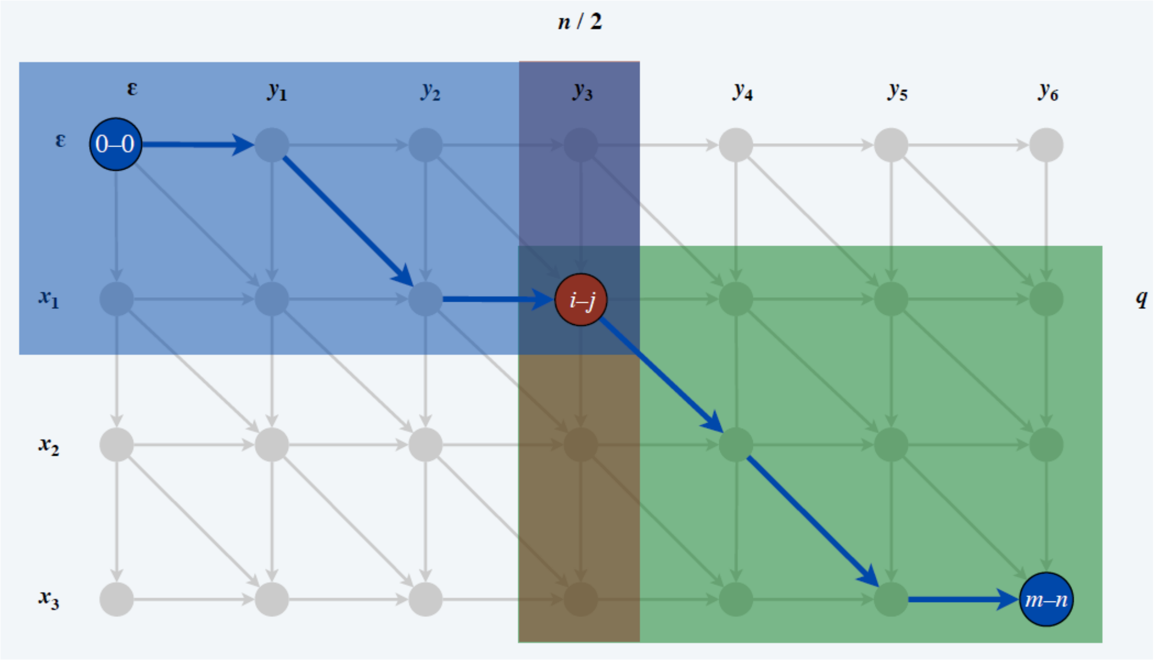
- Observation (can be proved by induction)
  - $E[i,j]$ = length of shortest path from $(0,0)$ to (i,j)



Hirschberg's Algorithm

This slide is not in the scope of the course

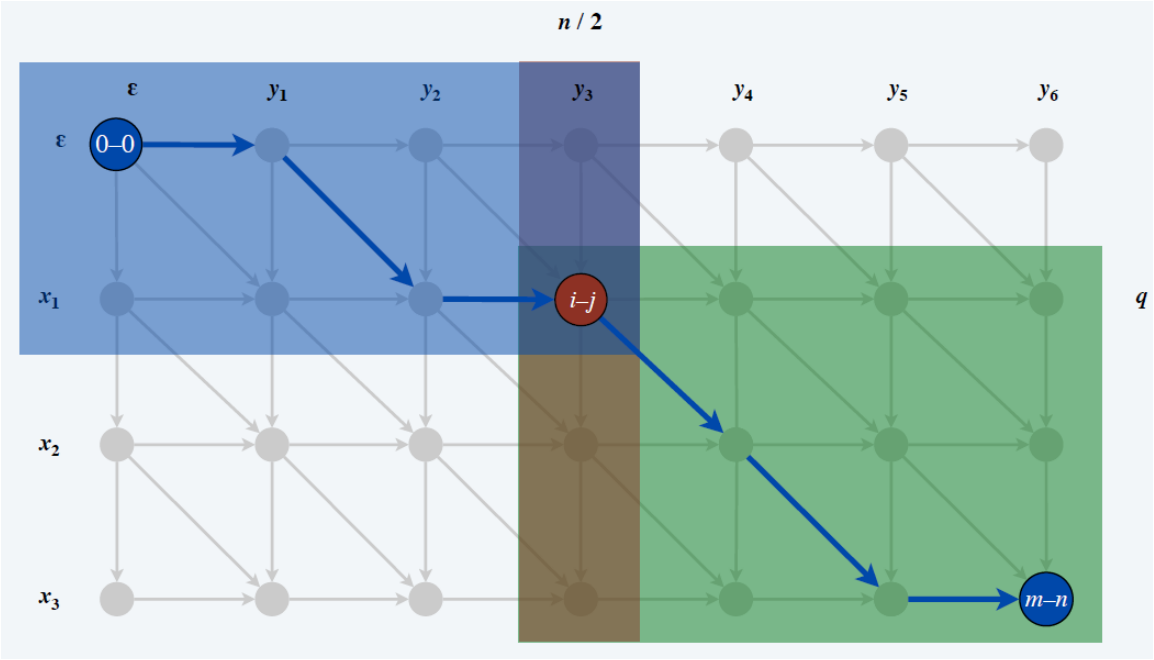
- Lemma
  - Shortest path from $(0,0)$ to (m,n) passes through $(q, {}^{n}/_{2})$ where q minimizes length of shortest path from $(0,0)$ to $(q, {}^{n}/_{2})$ + length of shortest path from $(q, {}^{n}/_{2})$ to (m,n)



Hirschberg's Algorithm

This slide is not in the scope of the course

- Idea
  - Find q using divide-and-conquer
  - Find shortest paths from $(0,0)$ to $(q, {}^{n}/_{2})$ and $(q, {}^{n}/_{2})$ to (m, n) using DP



Application: Protein Matching

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	7	-3	-3	-3	-1	-2	-2	0	-3	-3	-3	-1	-2	-4	-1	2	0	-5	-4	-1
R	-3	9	-1	-3	-6	1	-1	-4	0	-5	-4	3	-3	-5	-3	-2	-2	-5	-4	-4
N	-3	-1	9	2	-5	0	-1	-1	1	-6	-6	0	-4	-6	-4	1	0	-7	-4	-5
D	-3	-3	2	10	-7	-1	2	-3	-2	-7	-7	-2	-6	-6	-3	-1	-2	-8	-6	-6
C	-1	-6	-5	-7	13	-5	-7	-6	-7	-2	-3	-6	-3	-4	-6	-2	-2	-5	-5	-2
Q	-2	1	0	-1	-5	9	3	-4	1	-5	-4	2	-1	-5	-3	-1	-1	-4	-3	-4
E	-2	-1	-1	2	-7	3	8	-4	0	-6	-6	1	-4	-6	-2	-1	-2	-6	-5	-4
G	0	-4	-1	-3	-6	-4	-4	9	-4	-7	-7	-3	-5	-6	-5	-1	-3	-6	-6	-6
H	-3	0	1	-2	-7	1	0	-4	12	-6	-5	-1	-4	-2	-4	-2	-3	-4	3	-5
I	-3	-5	-6	-7	-2	-5	-6	-7	-6	7	2	-5	2	-1	-5	-4	-2	-5	-3	4
L	-3	-4	-6	-7	-3	-4	-6	-7	-5	2	6	-4	3	0	-5	-4	-3	-4	-2	1
K	-1	3	0	-2	-6	2	1	-3	-1	-5	-4	8	-3	-5	-2	-1	-1	-6	-4	-4
M	-2	-3	-4	-6	-3	-1	-4	-5	-4	2	3	-3	9	0	-4	-3	-1	-3	-3	1
F	-4	-5	-6	-6	-4	-5	-6	-6	-2	-1	0	-5	0	10	-6	-4	-4	0	4	-2
P	-1	-3	-4	-3	-6	-3	-2	-5	-4	-5	-5	-2	-4	-6	12	-2	-3	-7	-6	-4
S	2	-2	1	-1	-2	-1	-1	-1	-2	-4	-4	-1	-3	-4	-2	7	2	-6	-3	-3
T	0	-2	0	-2	-2	-1	-2	-3	-3	-2	-3	-1	-1	-4	-3	2	8	-5	-3	0
W	-5	-5	-7	-8	-5	-4	-6	-6	-4	-5	-4	-6	-3	0	-7	-6	-5	16	3	-5
Y	-4	-4	-4	-6	-5	-3	-5	-6	3	-3	-2	-4	-3	4	-6	-3	-3	3	11	-3
V	-1	-4	-5	-6	-2	-4	-4	-6	-5	4	1	-4	1	-2	-4	-3	0	-5	-3	7

Traveling Salesman

- Input
  - Directed graph $G = (V, E)$
  - Distance $d_{i,j}$ is the 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 $v_n = c$
      - We now want to find the optimal order of visiting nodes in $\{2, \dots, n\} \setminus \{c\}$
      - So we will need to keep track of which subset of nodes we need to visit and where we need to end
  - $OPT[S, c] =$ minimum total distance of starting at 1, visiting each node in S exactly once, and ending at $c \in S$ (without counting the distance for returning from c to 1)
    - Then the answer to our original problem can easily be computed as $\min_{c \in S} OPT[S, c] + d_{c,1}$, where $S = \{2, \dots, n\}$

Traveling Salesman

- DP Approach
  - To compute $OPT[S, c]$, we condition over the vertex which is visited right before c
- Bellman equation

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

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

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

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?

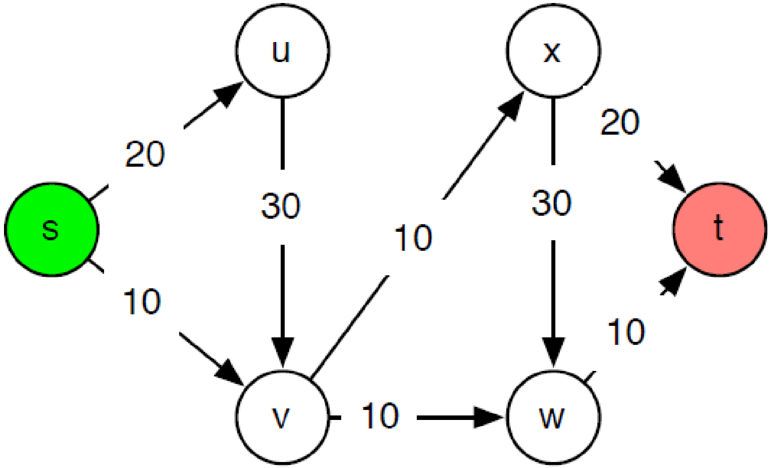
DP Concluding Remarks

- Key steps in designing a DP algorithm
  - "Generalize" the problem first
    - E.g. instead of computing edit distance between strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$, we compute $E[i, j]$ = edit distance between i-prefix of X and j-prefix of Y for all (i, j)
    - The right generalization is often obtained by looking at the structure of the "subproblem" which must be solved optimally to get an optimal solution to the overall problem
  - Remember the difference between DP and divide-and-conquer
  - Sometimes you can save quite a bit of space by only storing solutions to those subproblems that you need in the future

Network Flow

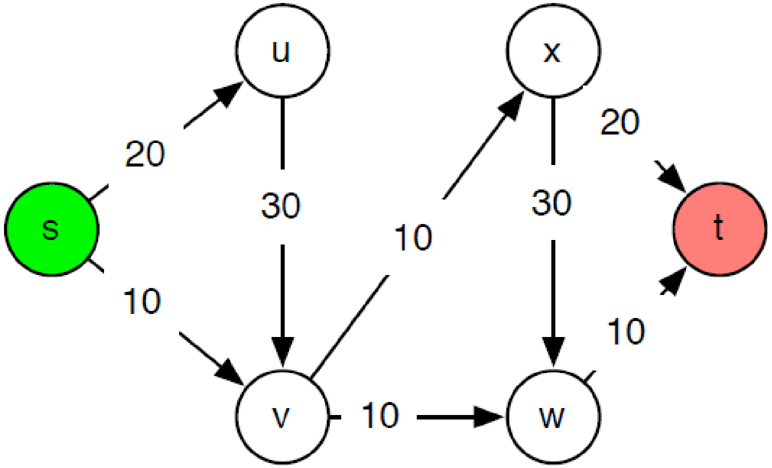
Network Flow

- Input
  - A directed graph $G = (V, E)$
  - Edge capacities $c : E \to \mathbb{R}_{\geq 0}$
  - Source node s, target node t
- Output
  - Maximum "flow" from s to t



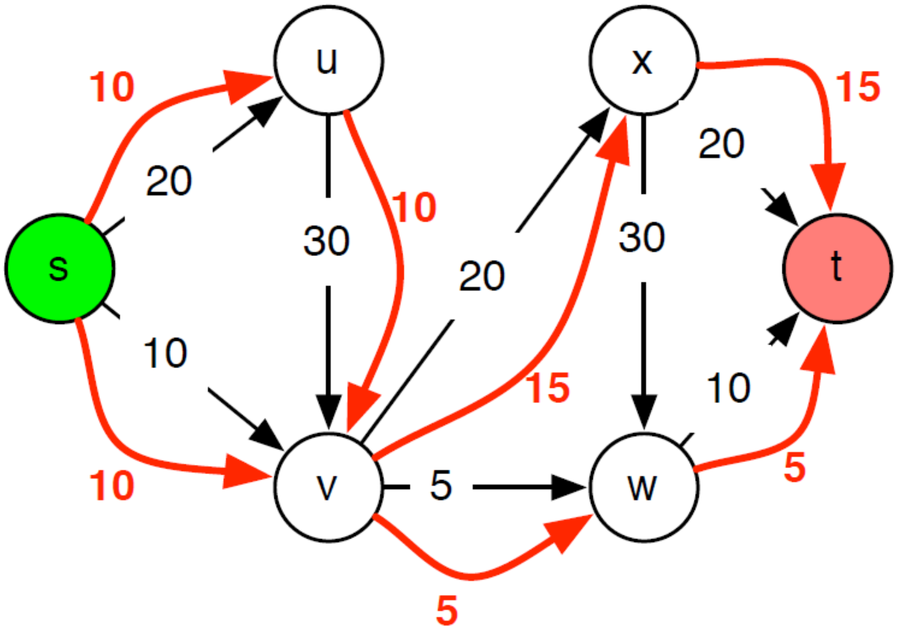
Network Flow

- Assumptions
  - For simplicity, assume that…
  - No edges enters s
  - No edges comes out of t
  - Edge capacity $c(e)$ is a non-negative integer
    - Later, we'll see what happens when $c(e)$ can be a rational number



Network Flow

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



Network Flow

- Constraints on flow f

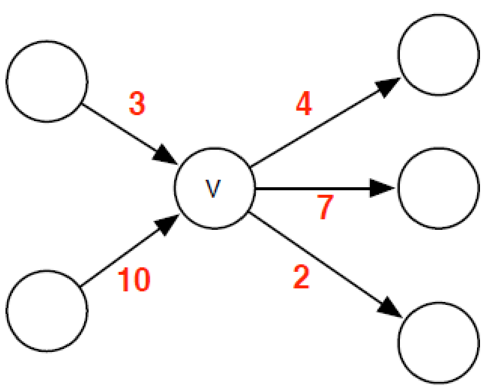
1. Respecting capacities

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

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

2. Flow conservation

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



Flow out at s = flow in at t

Network Flow

- $f^{in}(v) = \sum_{e \text{ into } v} f(e)$
- $f^{out}(v) = \sum_{e \text{ leaving } v} f(e)$
- Value of flow f is $v(f) = f^{out}(s) = f^{in}(t)$

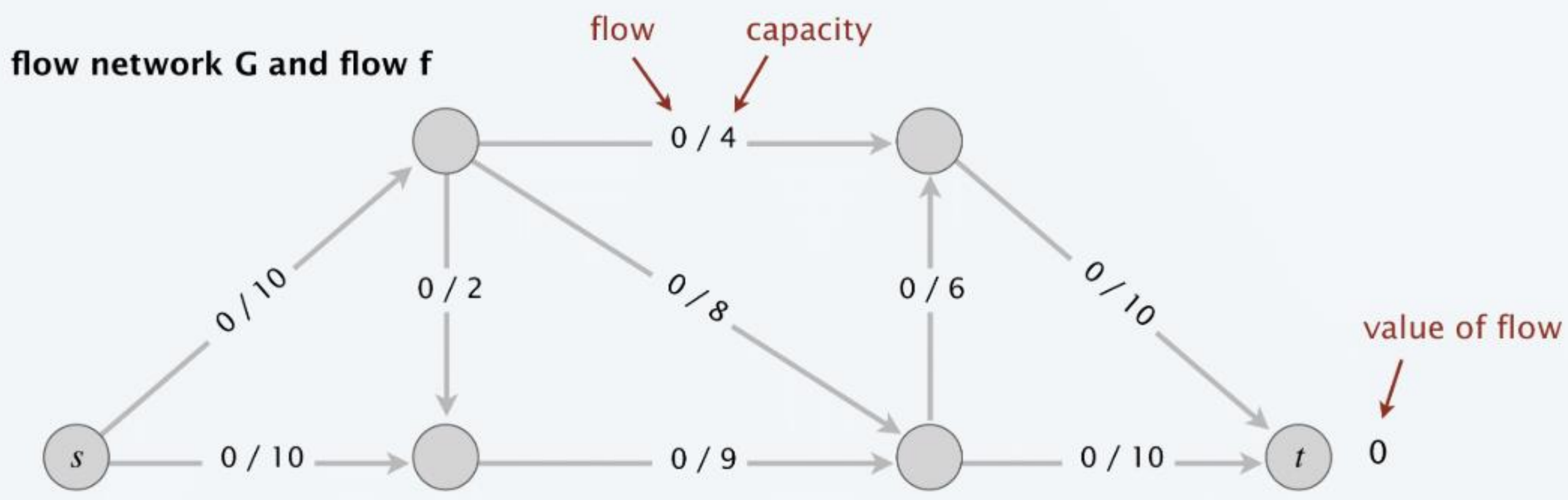
- Restating the problem:
  - Given a directed graph $G = (V, E)$ with edge capacities $c: E \to \mathbb{R}_{\geq 0}$, find a flow f^* with the maximum value.

First Attempt

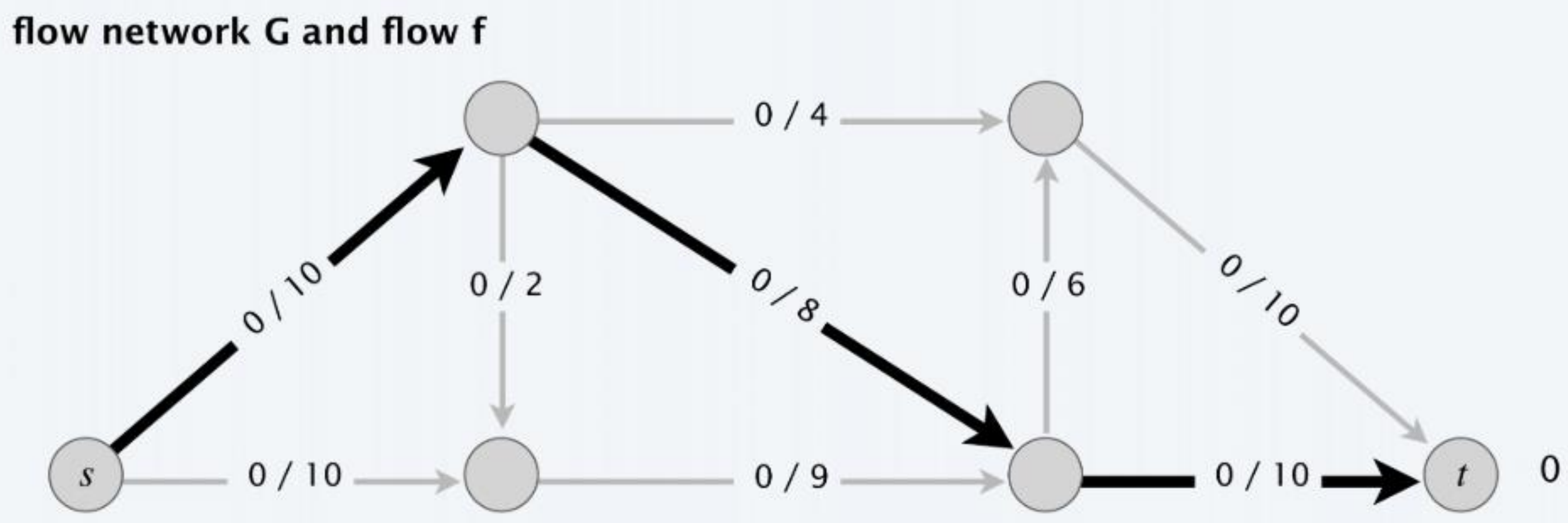
- A natural greedy approach
  1. Start from zero flow ($f(e) = 0$ for each e).
  2. While there exists an s-t path P in G such that $f(e) < c(e)$ for each $e \in P$
     a. Find one such path P
     b. Increase the flow on each edge $e \in P$ by $\min_{e \in P} \left(c(e) - f(e) \right)$

- Let's run it on an example!

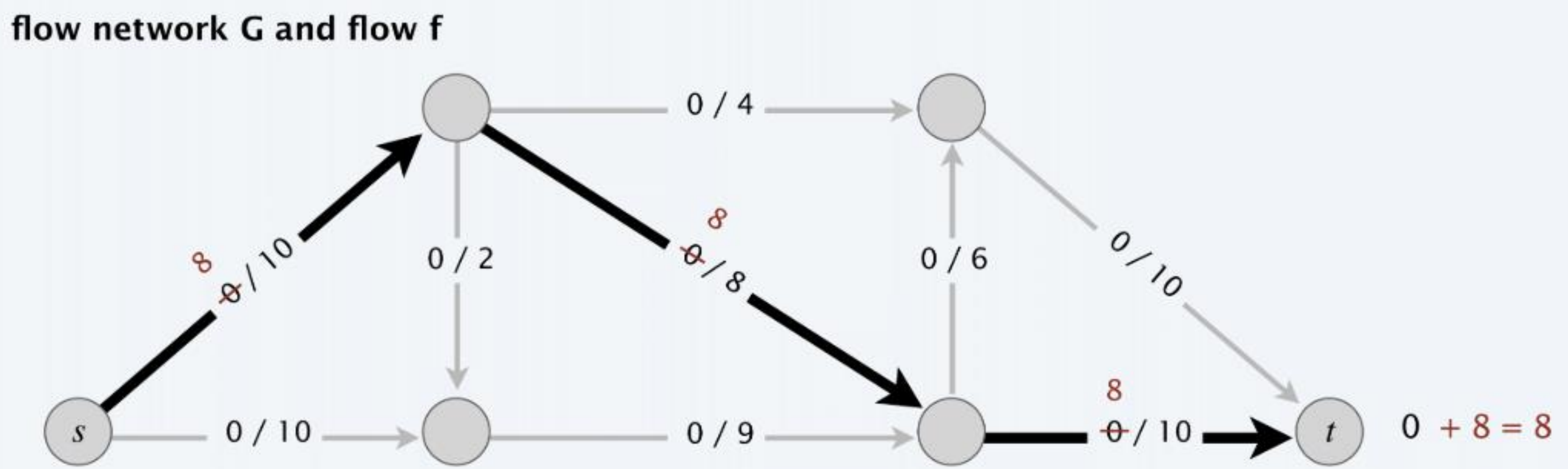
First Attempt



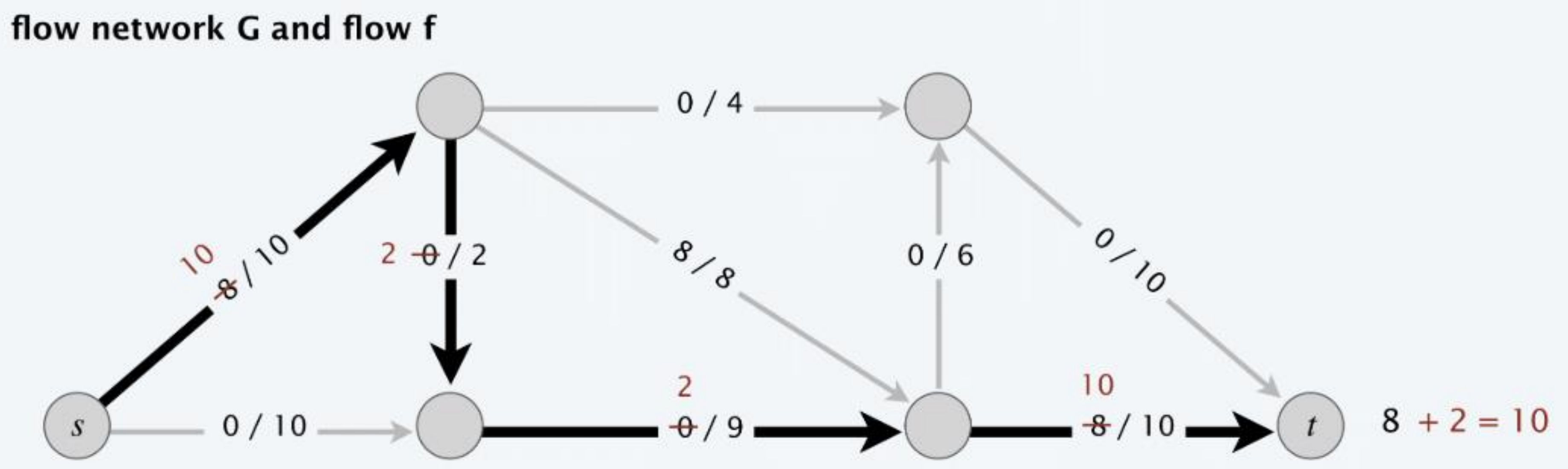
First Attempt



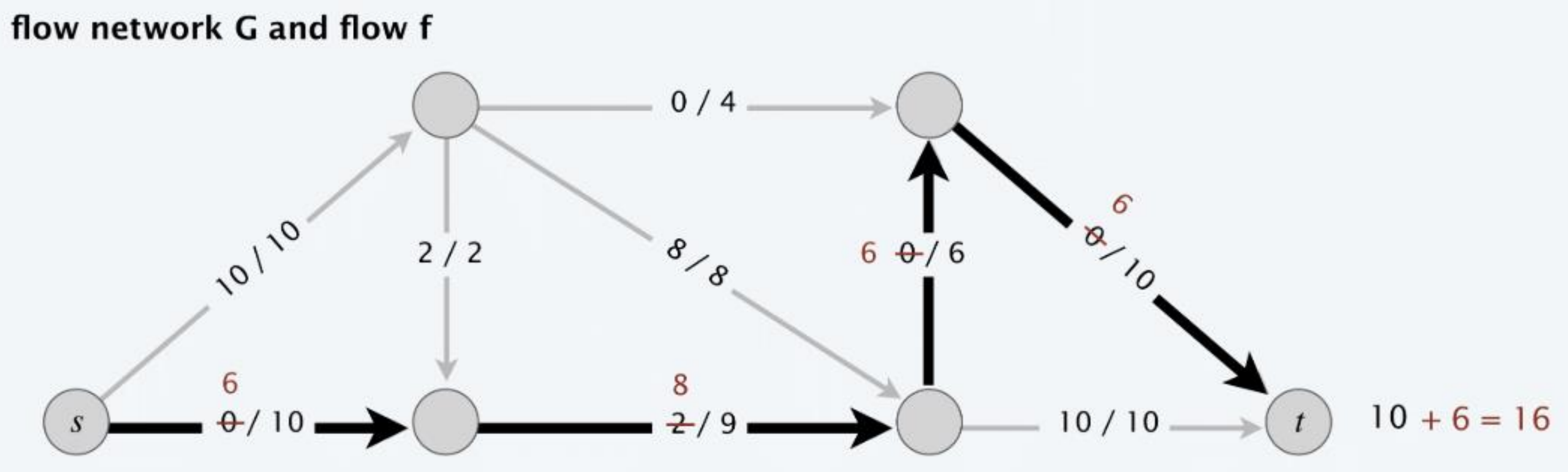
First Attempt



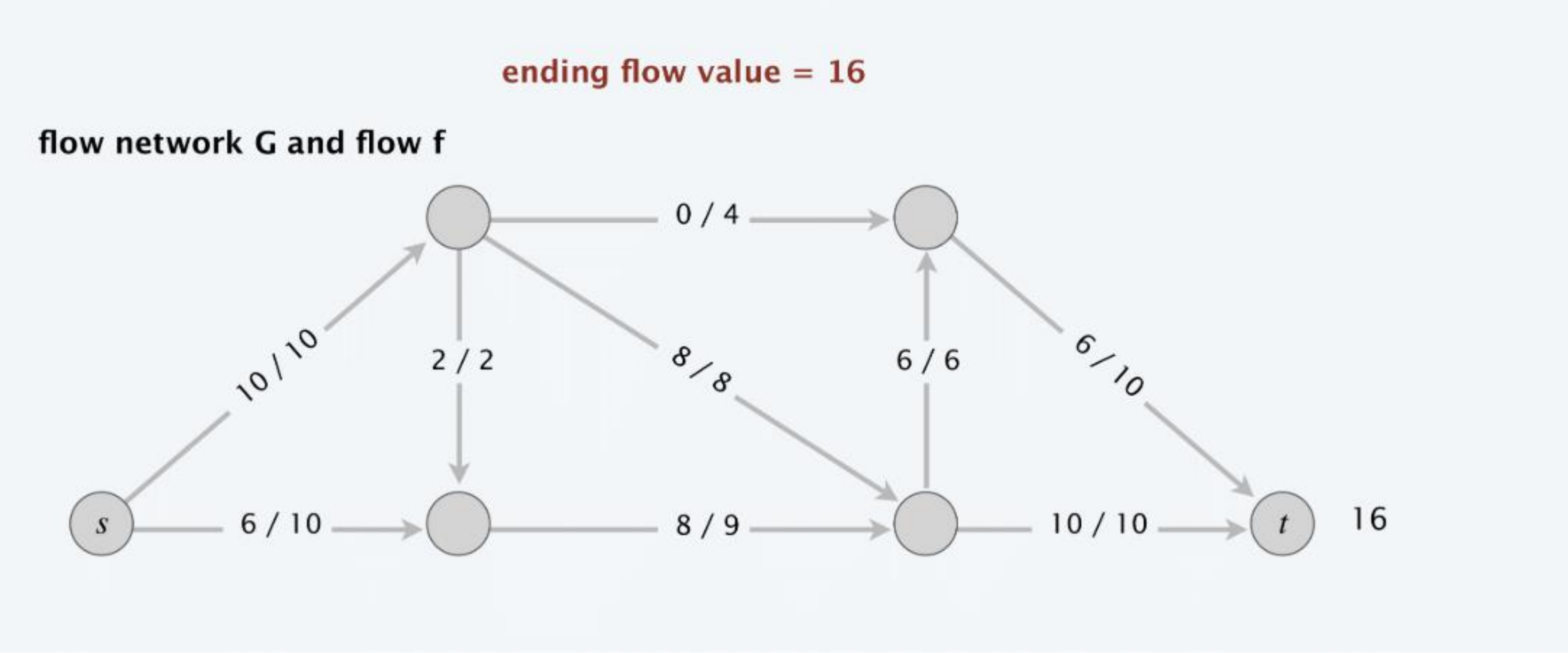
First Attempt



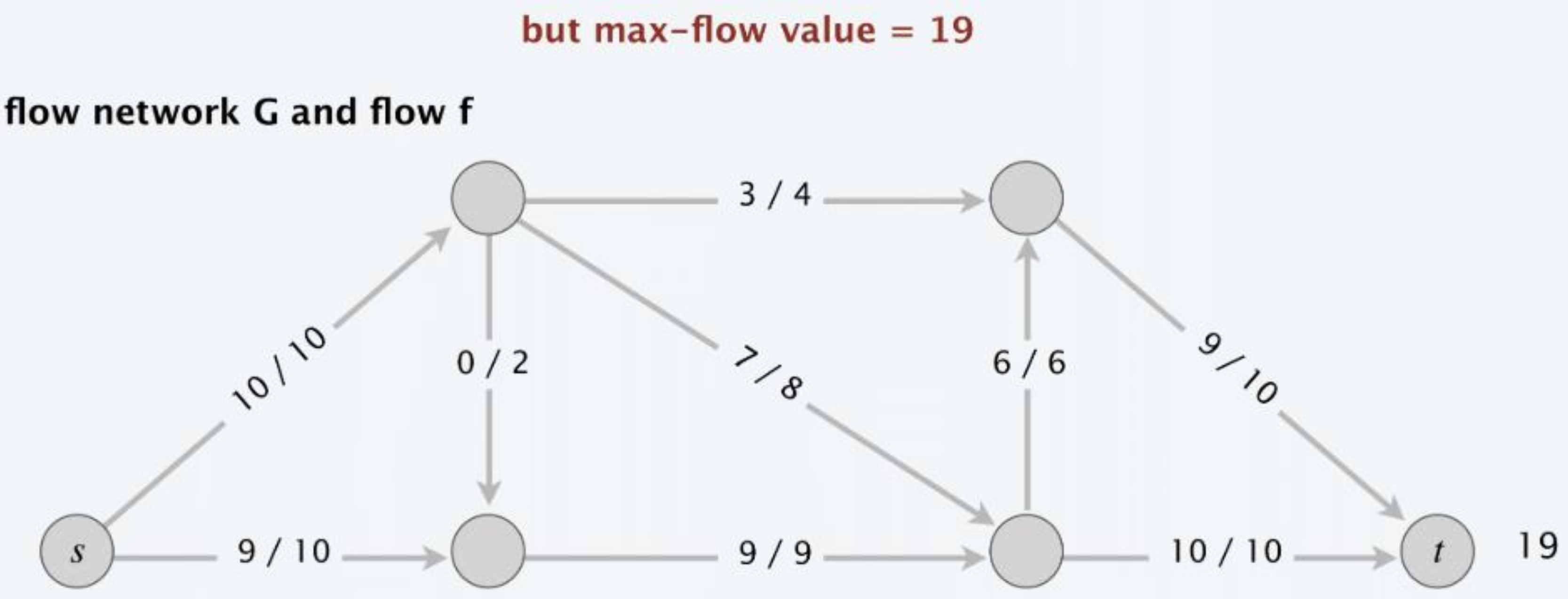
First Attempt



First Attempt



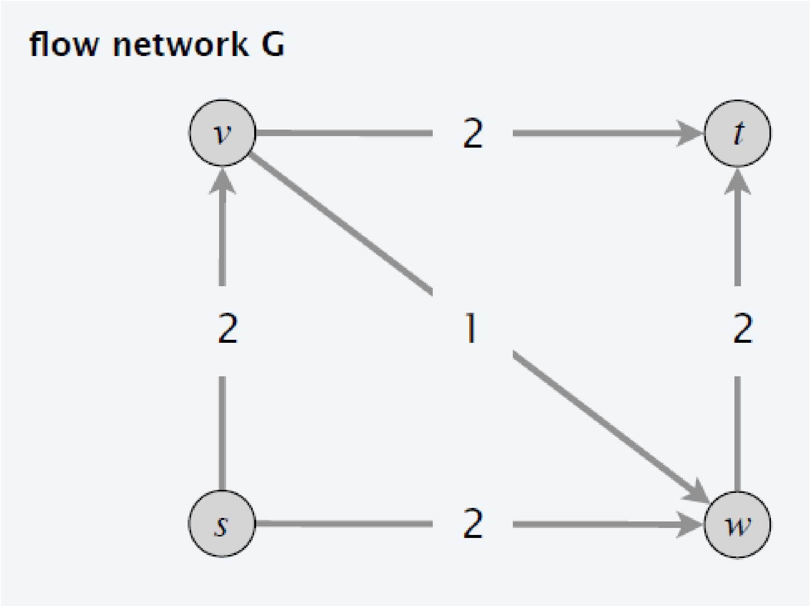
First Attempt



First Attempt

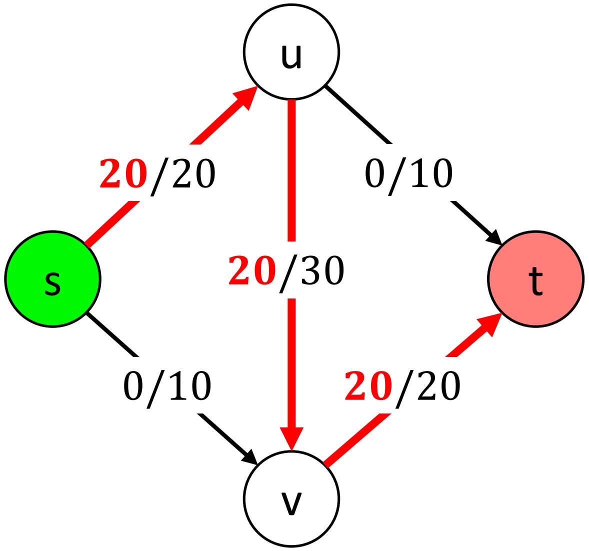
- Q: Why does the simple greedy approach fail?
- A: Because once it increases the flow on an edge, it is not allowed to decrease it.

- Need a way to "reverse" bad decisions

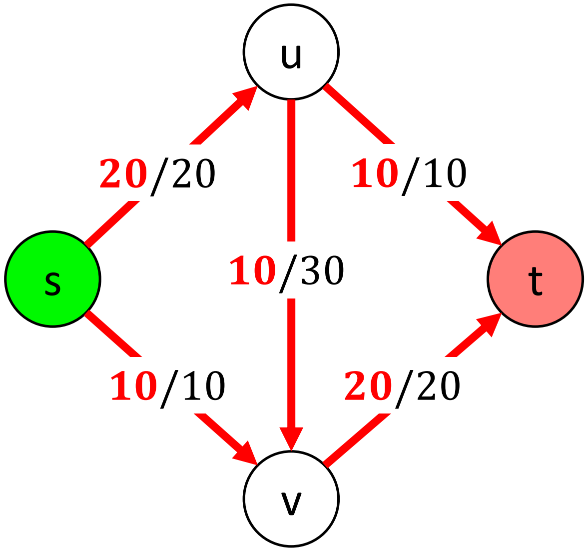


Reversing Bad Decisions

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

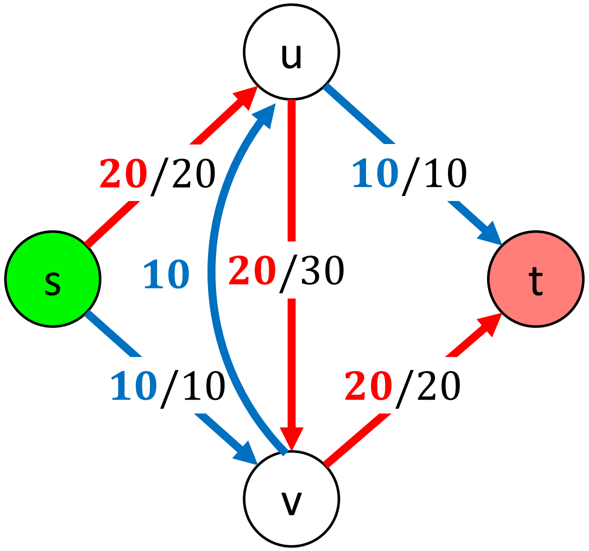


But the optimal configuration requires 10 fewer units of flow on $u \to v$

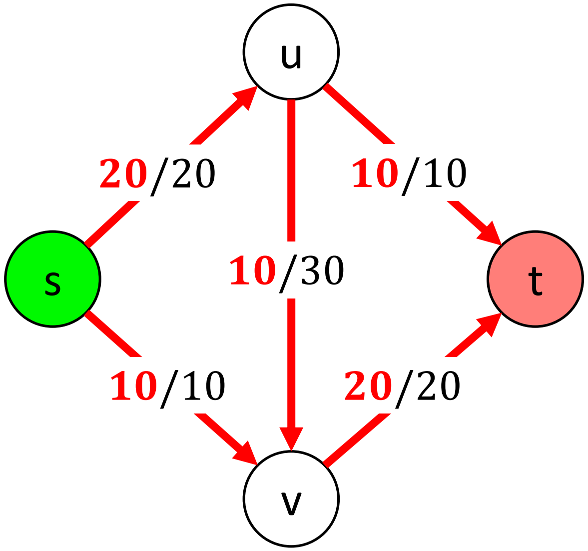


Reversing Bad Decisions

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



So now we get this optimal flow



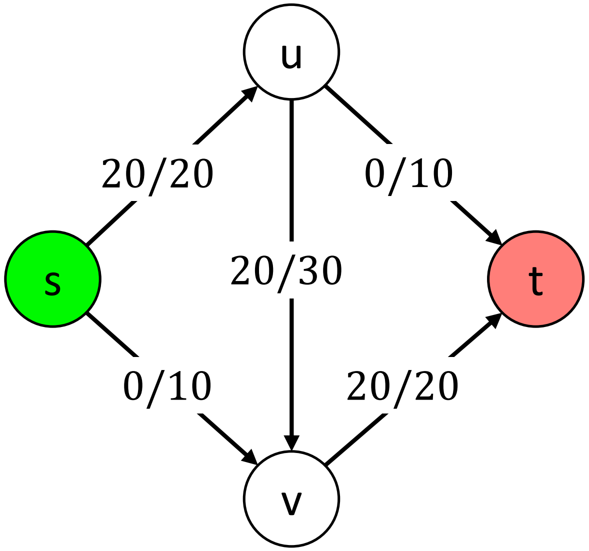
Residual Graph

- 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 add each edge if its capacity > 0

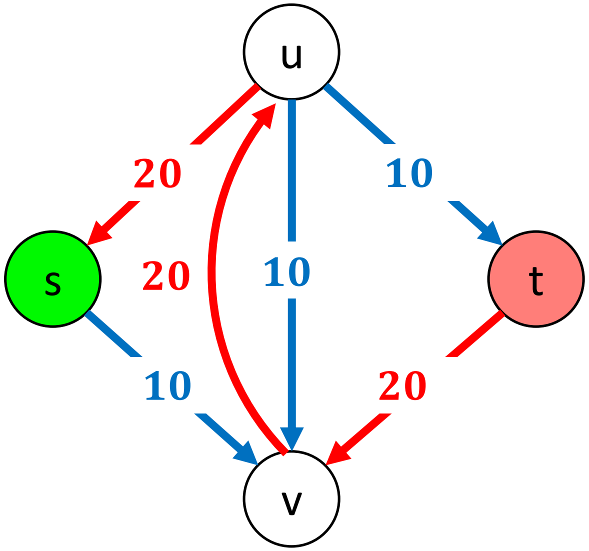
Residual Graph

- 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

Augmenting Paths

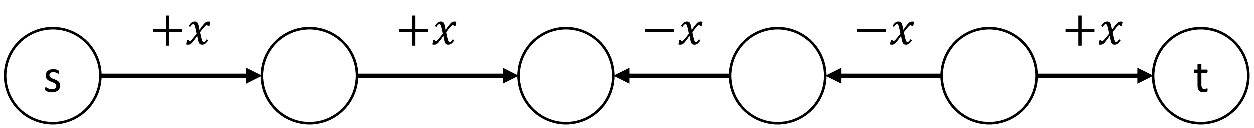
- Let's argue that the new flow is a valid flow

- Capacity constraints:
  - 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 $f(e) + \big(c(e) - f(e)\big) = c(e)$
  - 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:
  - Each node on the path (except s and t) has exactly two incident edges
    - Both forward / both reverse ⇒ one is incoming, one is outgoing
    - One forward, one reverse ⇒ both incoming / both outgoing
    - Net flow remains 0



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, $\text{bottleneck}(P, f) > 0$ implies $\text{bottleneck}(P, f) \geq 1$
    - Each augmentation increases flow by at least 1
    - At most $C = \sum_{e \text{ leaving } s} c(e)$ augmentations
  - Time for an augmentation:
    - G_f has n vertices and at most $2m$ edges
    - Finding an s-t path in G_f takes $O(m + n)$ time
  - Total time: $O((m + n) \cdot C)$

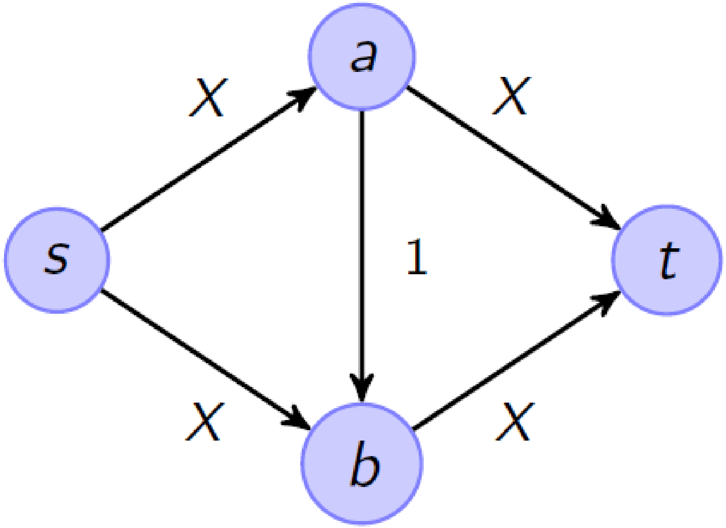
Ford-Fulkerson Algorithm

- Total time: $O((m+n) \cdot C)$
  - This is pseudo-polynomial time
  - C can be exponentially large in the input length (the number of bits required to write down the edge capacities)
  - Note: We assumed integer capacities, but this also gives a pseudo-polynomial time algorithm for rational capacities
    - Why?

- 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



Ford-Fulkerson Algorithm

- Ways to achieve polynomial time
  - Find the shortest augmenting path using BFS
    - Edmonds-Karp algorithm
    - Runs in $O(nm^2)$ time
    - Can be found in CLRS
  - Find the maximum bottleneck capacity augmenting path
    - Runs in $O(m^2 \cdot \log C)$ time
    - "Weakly polynomial time" (number of arithmetic operations depends on the number of bits used to write integers)
  - ...

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: $O(n\,m + n^{2+\epsilon})$ King-Rao-Tarjan
  - 1996: $O\left(n\,m\log_{m/_{n\log n}} n\right)$ King-Rao-Tarjan
    - Note: These are $O(n\,m)$ when $m = \omega(n)$
  - 2013: $O(n\,m)$ Orlin
    - Breakthrough!

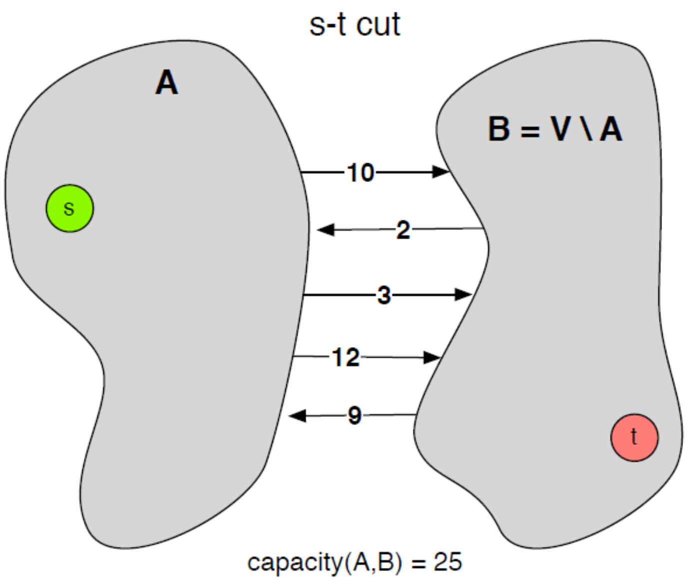
Back to Ford-Fulkerson

- We argued that the algorithm must terminate, and must do so 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

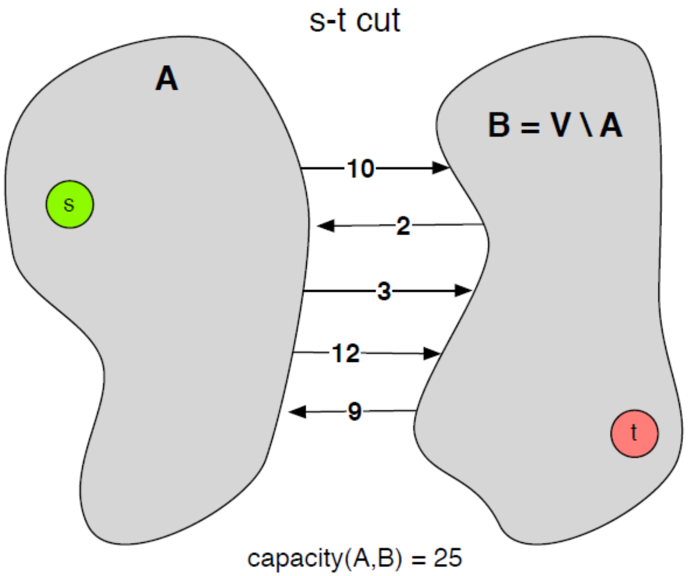
Cuts and Cut Capacities

- (A, B) is an s-t cut if it is a partition of vertex set (i.e. $A \cup B = V, A \cap B = \emptyset$), $s \in A$, and $t \in B$
- Capacity of this cut, denoted $cap(A, B)$, is the sum of capacities of edges *leaving* A



Cuts and Flows

- Theorem: For any flow f and any s-t cut (A, B), $v(f) = f^{out}(A) - f^{in}(A)$
- Proof: Just need to apply flow conservation (exercise!)



Cuts and Flows

- Theorem: For any flow f and any s-t cut (A, B), $v(f) \leq cap(A, B)$
- Proof:

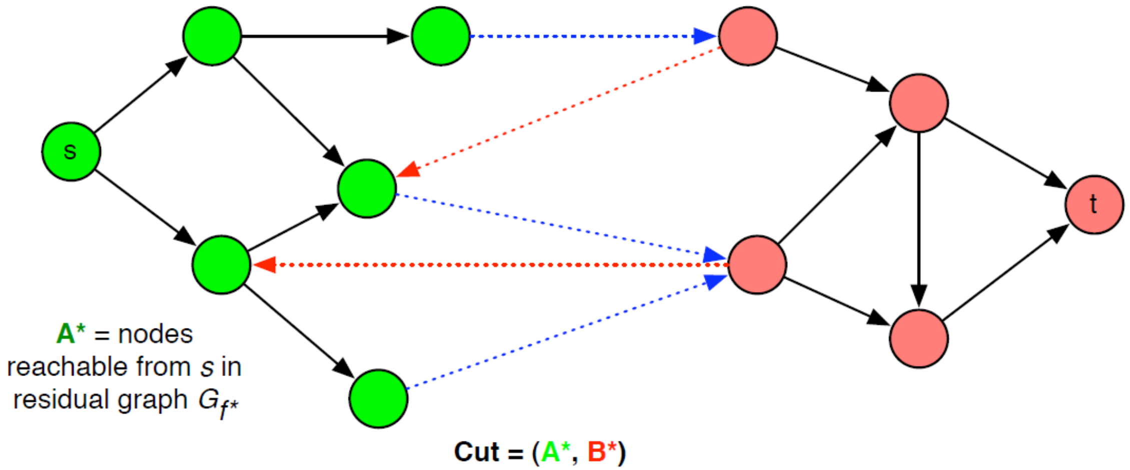
$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \\ &\leq f^{out}(A) \\ &= \sum_{e \text{ leaving } A} f(e) \\ &\leq \sum_{e \text{ leaving } A} c(e) \\ &= cap(A, B) \end{aligned}$$

Cuts and Flows

- Theorem: For any flow f and any s-t cut (A, B), $v(f) \leq cap(A, B)$
- So, the maximum flow is at most the minimum capacity of any cut.
- In fact, we will show that the maximum flow is *equal to* the minimum capacity of any cut.
  - To demonstrate the correctness (i.e. optimality) of Ford-Fulkerson algorithm, all we need to show is that the flow it generates is equal to the capacity of *some* cut.

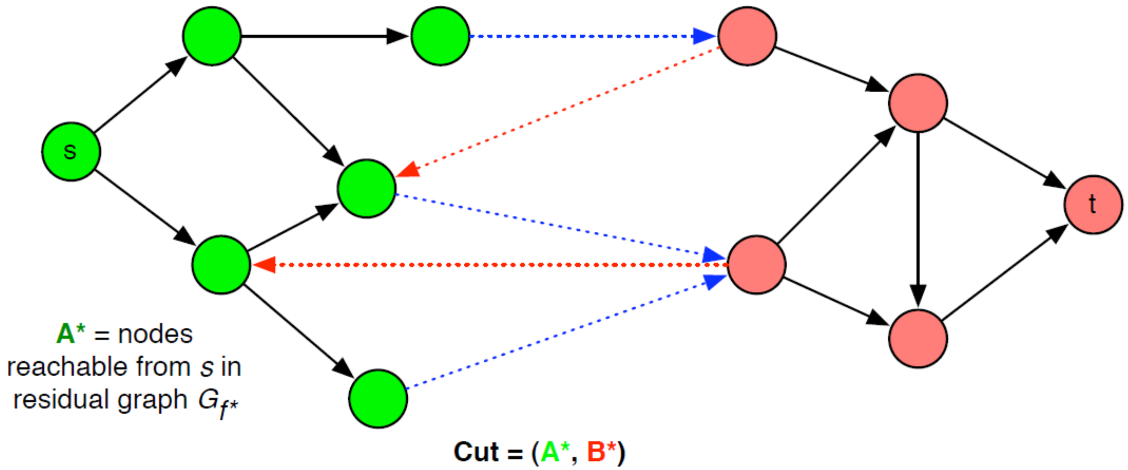
Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - Let f^* denote the flow returned by Ford-Fulkerson.
  - Look at G_{f^*} but define a cut in G



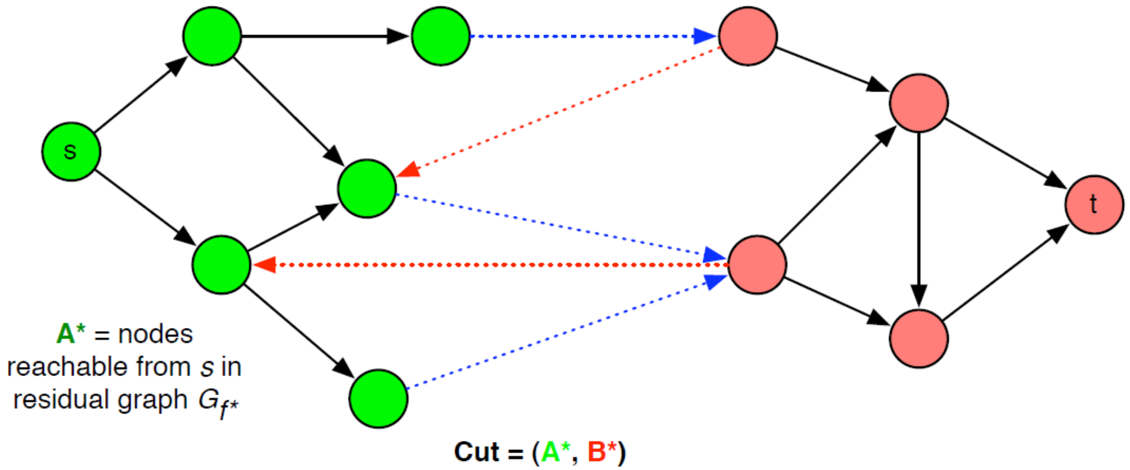
Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - (A^*, B^*) is a valid cut because there is no s-t path in G_{f^*} when Ford-Fulkerson terminates, so $t \notin A^*$



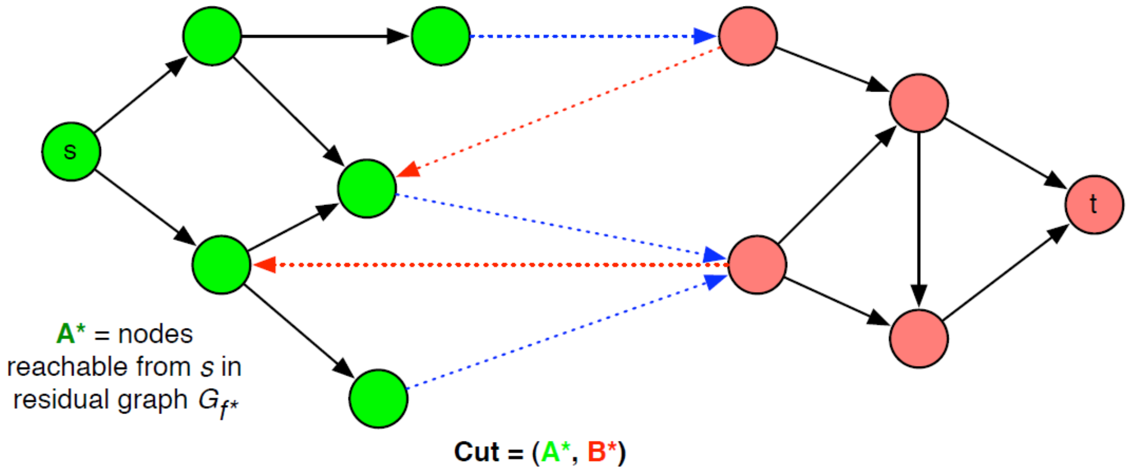
Cuts and Flows

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



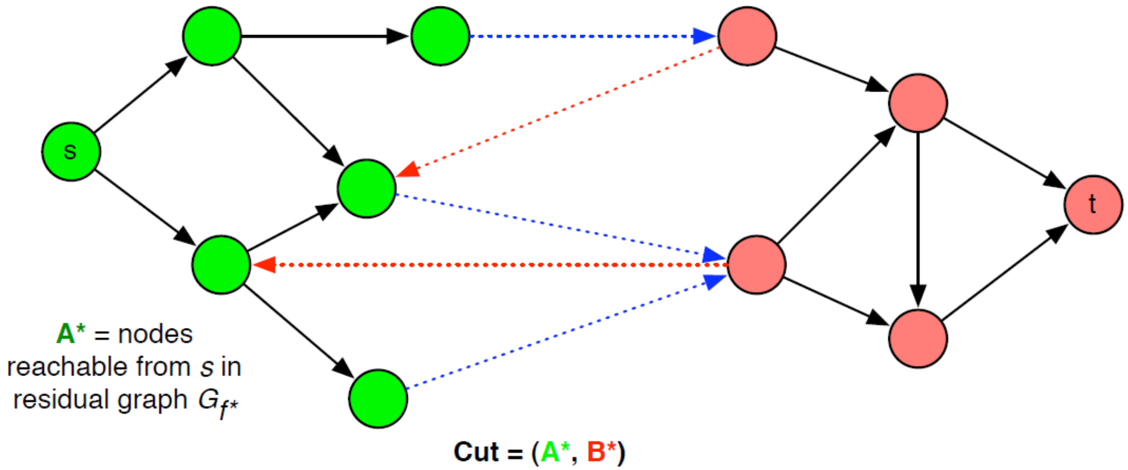
Cuts and Flows

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



Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - Each blue edge (u, v) must be saturated
  - Each red edge (v, u) must have zero flow
  - So $v(f^*) = cap(A^*, B^*)$ ■



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 showed that Ford-Fulkerson can be used to find the min cut
  - Find the max flow f^*
  - Let $A^* =$ set of all nodes reachable from s in G_{f^*}
    - Easy to compute using BFS
  - Then $(A^*, V \setminus A^*)$ is min cut

Why Study Flow Networks?

- Unlike divide-and-conquer, greedy, or dynamic programming, this doesn't seem like a framework
  - It is more like a single problem

- It turns out that many problems can be reduced to this single problem
  - Hence, it is a very versatile technique

- Next lecture!