UPE Tutoring:

CS 180 Final Review

Sign-in https://tinyurl.com/upe180final

Slides link available upon sign-in

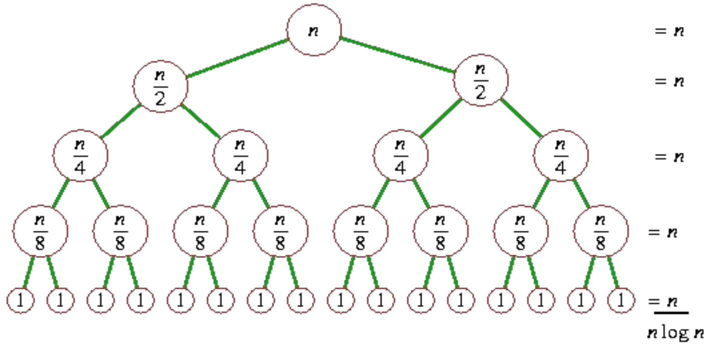
Topics

- **Divide and Conquer**
    - Inversions
    - Majority (prev MT question)
- **Dynamic Programming**
    - 1D Example
    - 2D Example - Knapsack
    - Coins Practice Problem
    - House Robber Practice Problem
- **Ford Fulkerson**
- **NP-Completeness**

Divide and Conquer

Divide and Conquer - Principles

- Solve a recurrence relation that bounds an algorithm's runtime recursively in terms of runtime on smaller inputs
- Most common and intuitive algorithm: MergeSort
- Typically look similar to: T(n) <= 2T(n/2) + cn for n > 2 and T(2) <= c
- **When to use**: when the brute force version is polynomial time, and we want a better polynomial time algorithm



Merge Sort

- The classic divide and conquer problem:
  - Divide step: Just split input in half
  - Conquer step: Merge two sorted inputs
- Find the pattern:
  - At level j, there are 2^j subproblems
  - Each is size $n/2^j$, so they take $cn/2^j$ times
  - Each level takes $2^j(cn/2^j) = cn$
- There are $\log_2(n)$ levels, so when we get:
  - $O(cn\log_2 n) \rightarrow O(n\log_2 n)$ runtime

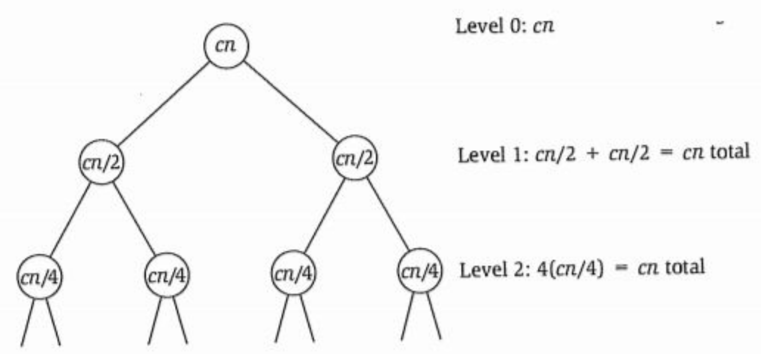

cn
$cn/2$
$cn/2$
$cn/4$
$cn/4$
$cn/4$
$cn/4$
Level 0: cn
Level 1: $cn/2 + cn/2 = cn$ total
Level 2: $4(cn/4) = cn$ total

Figure 5.1 Unrolling the recurrence $T(n) \leq 2T(n/2) + O(n)$.

Counting Inversions - Background

Say you and a friend ranked the same set of n movies. We want to see how similar your tastes are.

We do this by labelling the movies by your ranking, then sorting by your friend's ranking, and seeing how many pairs are "out of order".

Counting Inversions - Problem Statement

Given a sequence of n distinct numbers $\{a_1, \ldots, a_n\}$, find number of **inversions** in the sequence.

An inversion is a case where two indices i and j satisfy $a_i > a_j$ in the sequence.

Ex: {2, 4, 1, 3, 5} has the inversions (4, 1), (4, 3), and (2, 1)

Counting Inversions - Approach

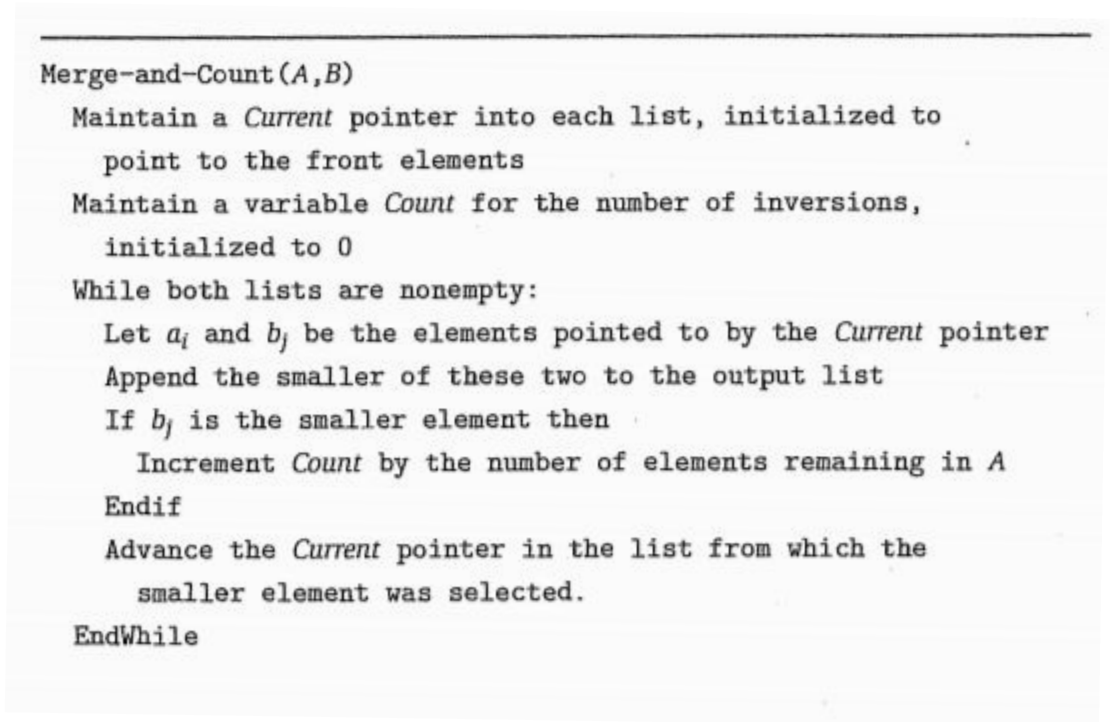
General Approach

- Set m = ceiling(n / 2) and divide the sequence into
    - $\{a_1, \ldots, a_m\}$ and $\{a_{m+1}, \ldots, a_n\}$
- Count the inversions in each half, then sort (Divide)
- Count the inversions between elements in each half (Conquer)

Specifics

- Merge-and-Count:
    - Given sorted lists A and B, merge them into a sorted list C and count inversions
    - Regular merge: have pointers into A and B, a_i and b_j, and add the smaller element to C and advance its pointer
    - **Key Property:** Every time we add b_j, it is smaller than all remaining items in A, so all of those are inversions

Counting Inversions - Formal Algorithm



```
Merge-and-Count(A,B)
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  Maintain a variable Count for the number of inversions,
    initialized to 0
  While both lists are nonempty:
    Let $a_i$ and $b_j$ be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    If $b_j$ is the smaller element then
      Increment Count by the number of elements remaining in A
    Endif
    Advance the Current pointer in the list from which the
      smaller element was selected.
  EndWhile
```

Counting Inversions - Explanation

Same as Merge Sort!

The algorithm is the same; we just do an extra calculation while we merge.

Time Complexity: O(NlogN)

Divide and Conquer - Majority (On previous midterm)

Given a sequence of keys x_1,...,x_n, and the only query we can ask is whether x_i = x_j and get the answer yes or no. We would like to find whether there exists a key that constitutes a majority(occurs more than > N/2 times).

a) What will the time complexity of a brute force algorithm be?
b) Design an algorithm with O(n log n) time complexity using divide and conquer.

Majority

a) Is pretty obvious. What's the answer?

b) Hint: start with one element or two elements in your sequence. What is the majority? How can you generalize this?

Majority b)

Divide: For any sequence of length n, divide it into x_1,...,x_{n/2}, and x_{n/2+1},...,x_n. Perform this iteratively until we arrive at every sequence is of length 1.

Conquer:
Case 1: Single element: return that single element as majority
Case 2: Get result from two halves - both return same majority: return same majority
Case 3: Get result from two halves - different majorities: Perform O(N) scan to check if there is a majority.
Case 4: No majority from two halves: return no majority

The time recurrence relation is T(N) = T(N/2) * 2 + **O(N)**, so it's O(nlogn).

Dynamic Programming

Dynamic Programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems

Three important steps:

1. Define the subproblems
2. Write down the recurrence relation
3. Recognize and solve base cases

1D Dynamic Programming - Example

For a given n, find the number of ways to write n as a sum of the numbers 1, 3, and 4 (order matters / each of the different orders are counted).

Example:
For n = 5, the answer is 6:

$$5 = 1 + 1 + 1 + 1 + 1$$
$$5 = 1 + 1 + 3$$
$$5 = 1 + 3 + 1$$
$$5 = 3 + 1 + 1$$
$$5 = 1 + 4$$
$$5 = 4 + 1$$

1D Dynamic Programming - Example

Let's go through the three steps...

1. Define the subproblems
   Let dp[i] be the number of ways to write i as a sum of 1, 3, and 4

2. Find the recurrence relation
   dp[i] = ?
   Consider one possible solution: $i = x_1 + x_2 + \ldots + x_n$
   If $x_n = 1$, how many ways are there to sum up to i?
   This is equivalent to dp[i - 1]
   Now consider the other two cases (if $x_n = 3$ or $x_n = 4$)

1D Dynamic Programming - Example

2. Find the recurrence relation

   dp[i] = dp[i - 1] + dp[i - 3] + dp[i - 4]

3. Recognize and solve the base cases

   Our recurrence relation depends on the past 4 elements.

   dp[0] = dp[1] = dp[2] = 1, dp[3] = 2

```
dp[0] = dp[1] = dp[2] = 1; dp[3] = 2;
for(i = 4; i <= n; i++)
    dp[i] = dp[i - 1] + dp[i - 3] + dp[i - 4];
```

2D Dynamic Programming - Knapsack Problem

We are given a set of n items, where each item i is specified with a size s_i and value v_i . We are also given the capacity C of our backpack.

Problem statement for the 0-1 Knapsack Problem:

Find the maximum value of items we can store in our backpack such that each item can only be put in our backpack 0 or 1 times, and the sum of all of the sizes of the items in our backpack cannot exceed the capacity C.

2D Dynamic Programming - Knapsack Problem

1. Define the subproblems:
   Let dp[i][j] be the maximum value we can store in our backpack if we only consider the first i items and have a maximum capacity of j.

2. Find the recurrence relation:
   We either take the current item or we don't.
   dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - s[i]] + v[i])

3. Recognize and solve the base cases:
   dp[i][0] = 0 for all 0 <= i <= n
   dp[0][j] = 0 for all 0 <= j <= C

2D Dynamic Programming - Knapsack Problem

Example where n = 4 and C = 5:

Items (s_i, v_i) are: {(2, 3), (3, 4), (4, 5), (5, 6)}

i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Dynamic Programming - Practice Problem

Given unlimited coins of the values 1, 3, and 4, find the minimum number of coins needed to make change for a value v.

Example:

For 6, the minimum number of coins needed is 2 (3 + 3).

Note that the greedy approach, while temping, does not work here!
6 = 4 + 1 + 1 is not the best solution

Dynamic Programming - Practice Problem Solution

Given unlimited coins of the values 1, 3, and 4, find the minimum number of coins needed to make change for a value v.

1. Define subproblems
   Let dp[i] be the minimum number of coins needed to make change for value i
2. Find the recurrence relation
   dp[i] = min(dp[i - 1], dp[i - 3], dp[i - 4]) + 1
3. Recognize and solve the base cases
   dp[0] = 0; dp[1] = 1; dp[2] = 2; dp[3] = 1

Loop from 4 to v. dp[v] is your final answer.

Dynamic Programming - House Robber

Leetcode #198
You are a professional **robber** planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if **two adjacent houses** were broken into on the same night.

Given a list of **non-negative integers** representing the amount of money of each house, determine the **maximum amount of money** you can rob tonight without alerting the police.

Example:
input: nums = **[200, 20, 30, 400]**
output: **600** (nums[0] + nums[3])

Dynamic Programming - House Robber Solution

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

1. Define subproblems
   Let dp[i] be the maximum amount of money you can steal up to house at index i
2. Find the recurrence relation
   dp[i] = max(nums[i] + dp[i - 2], dp[i - 1])
3. Recognize and solve the base cases
   dp[0] = nums[0]; dp[1] = max(nums[0], nums[1])

Loop from 2 to number of houses = l. dp[l-1] is your final answer.

Ford Fulkerson

Ford Fulkerson

- 3 Important concepts: Residual network, augmenting paths, cuts.
- **Residual network:** Given a graph G, we have a residual network G_f which tells us how many units of flow we can add:
  - $c_f(u,v) = c(u,v) - f(u,v)$, where c_f is the residual capacity, c is the capacity, f is the flow.
  - Formally speaking: $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$. There is no edge between u,v if the residual flow is 0.
- **Augmenting path:** A flow f, augmented by a flow f', is denoted by f + f'.
  - We can increase the flow from (u,v) if we can add augmenting flow in the residual network, but at the same time, we lose flow from (v,u).
  - An augmenting path is a path such that we can add an augmenting flow to f:
    - $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$, is the amount of flow we can augment.

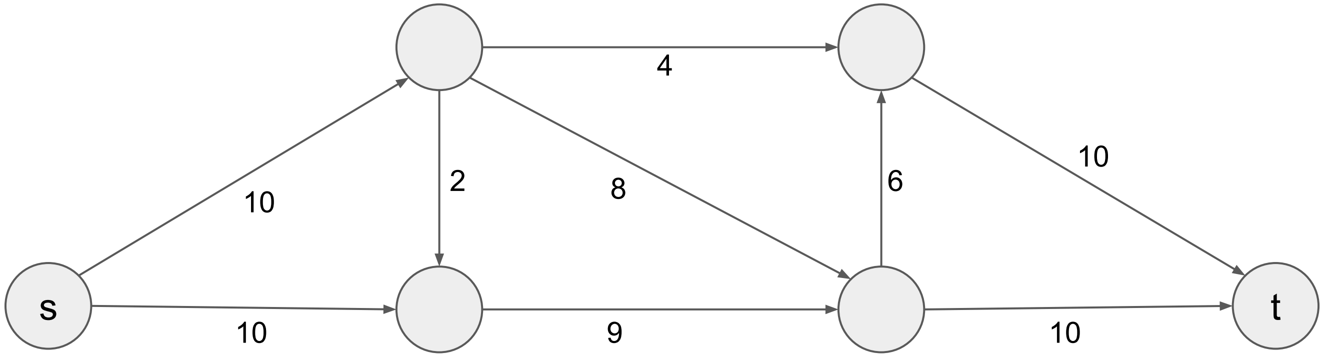
Ford Fulkerson (cont'd)

- An (S,T) cut is a cut such that we can partition V in G = (V,E) into S and T (disjoint).
    - Because ford fulkerson starts from some node s, and ends at some node t, we need $s \in S$, and $t \in T$.
- There is a very important theorem: Max-flow, min-cut
    - 1. |f| = c(S,T) for some cut S, T. c(S,T) is the total capacity from S to T.
    - 2. f is a maximum flow (there is some cut that is **saturated**)
    - 3. f admits no augmenting paths.
- From this, we can admit the following simple pseudocode:

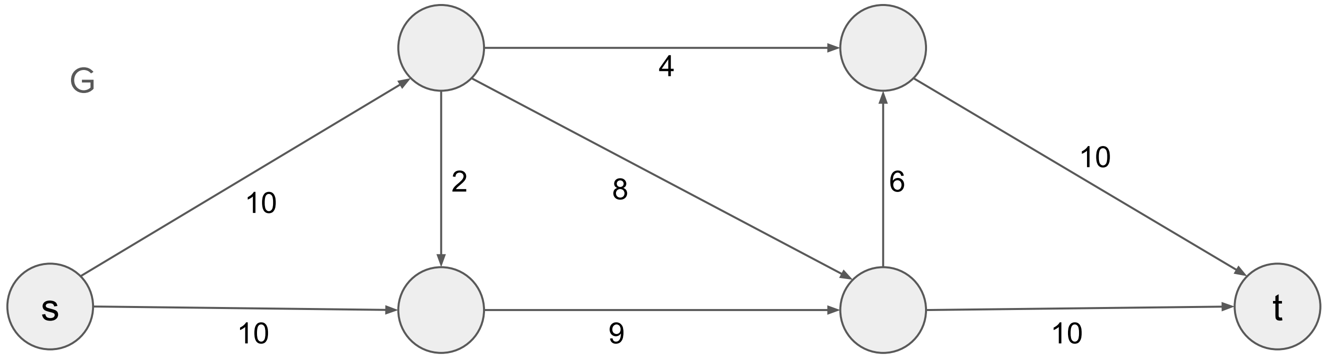
```
While there is an augmenting path f':
    Add f' to f
Return f
```

Ford Fulkerson Walkthrough

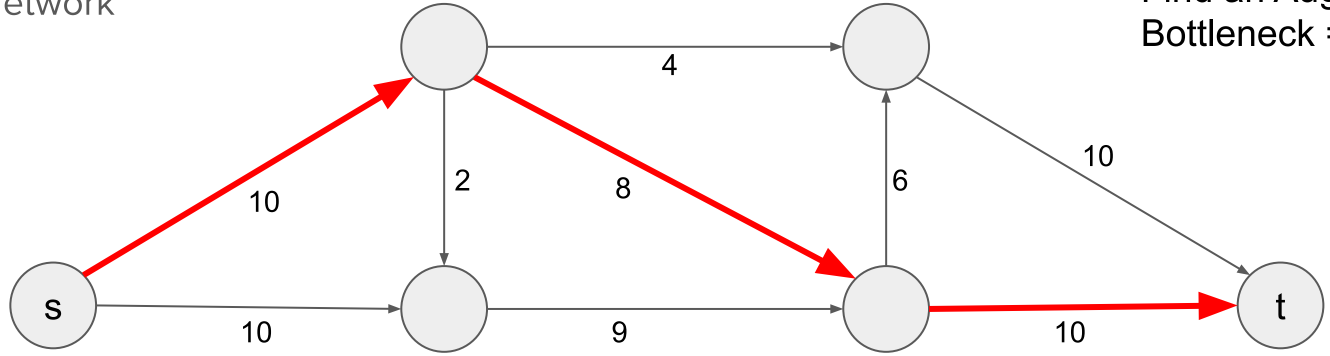
Given a graph G:



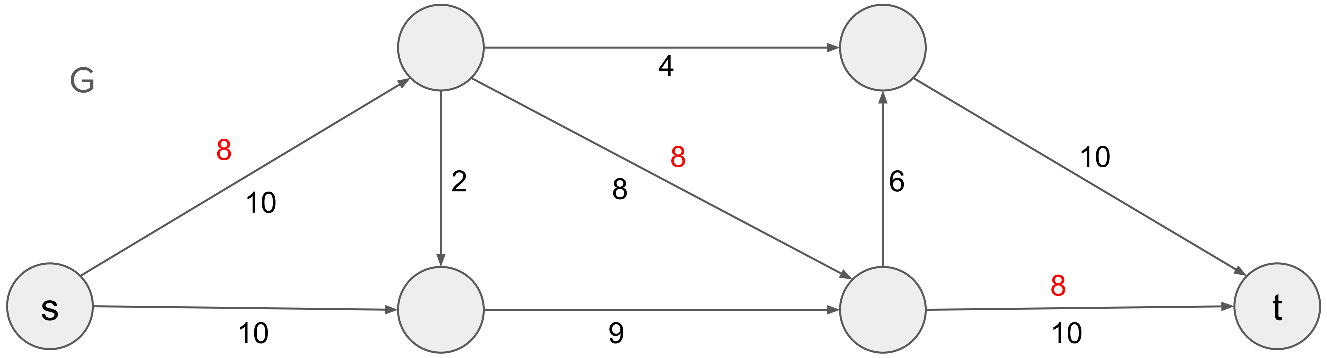
G



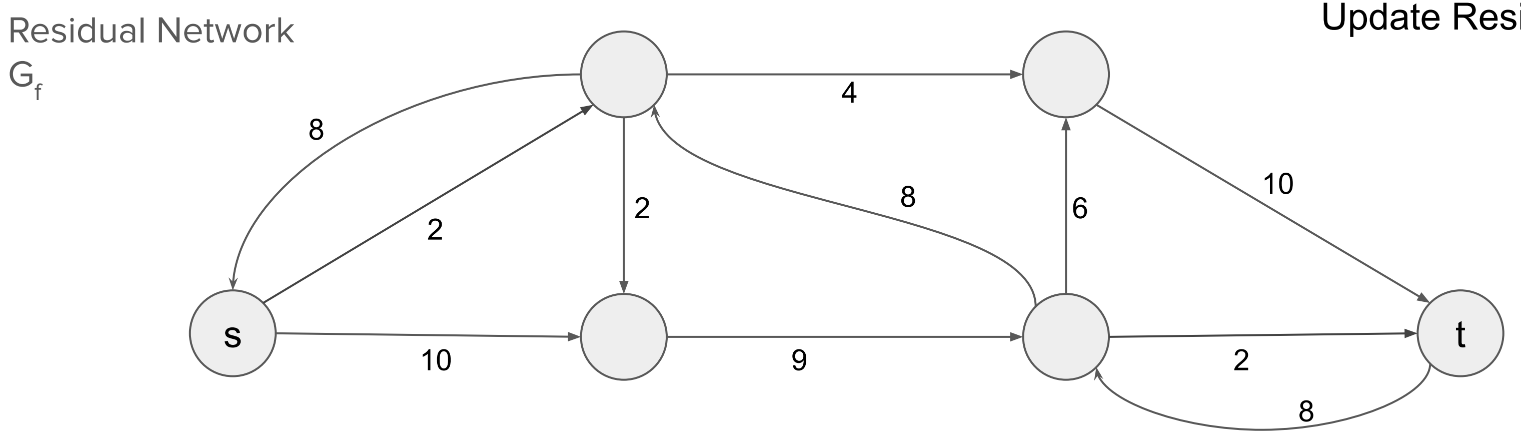
Residual Network G_f

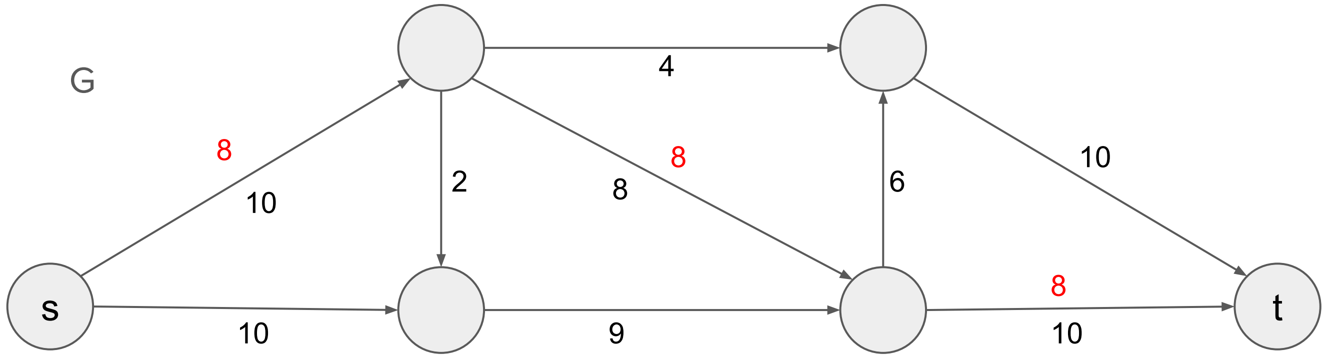


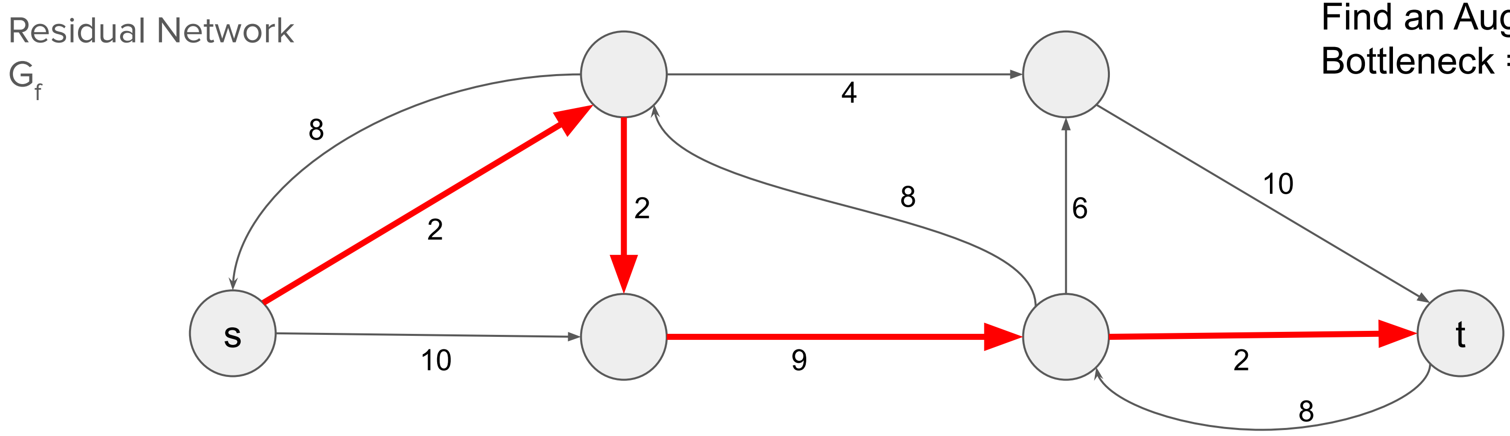
Find an Augmenting Path
Bottleneck = 8



Update Residual Network

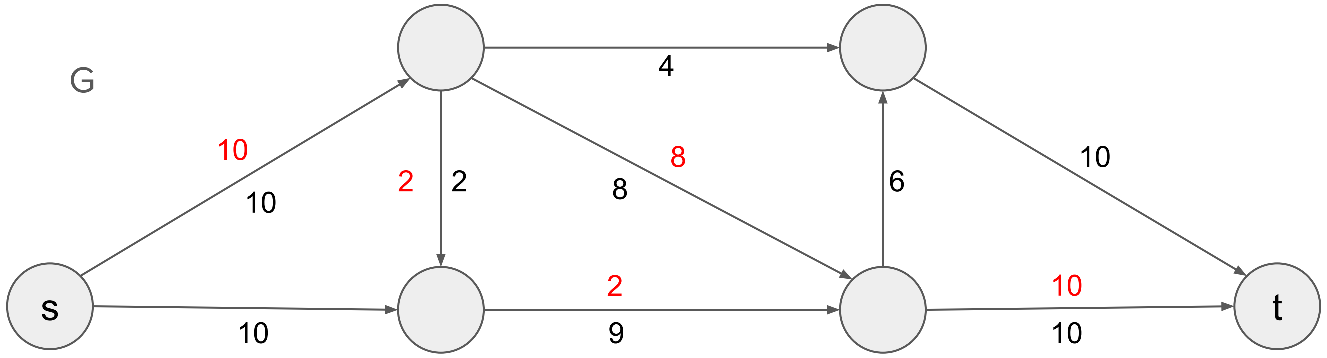


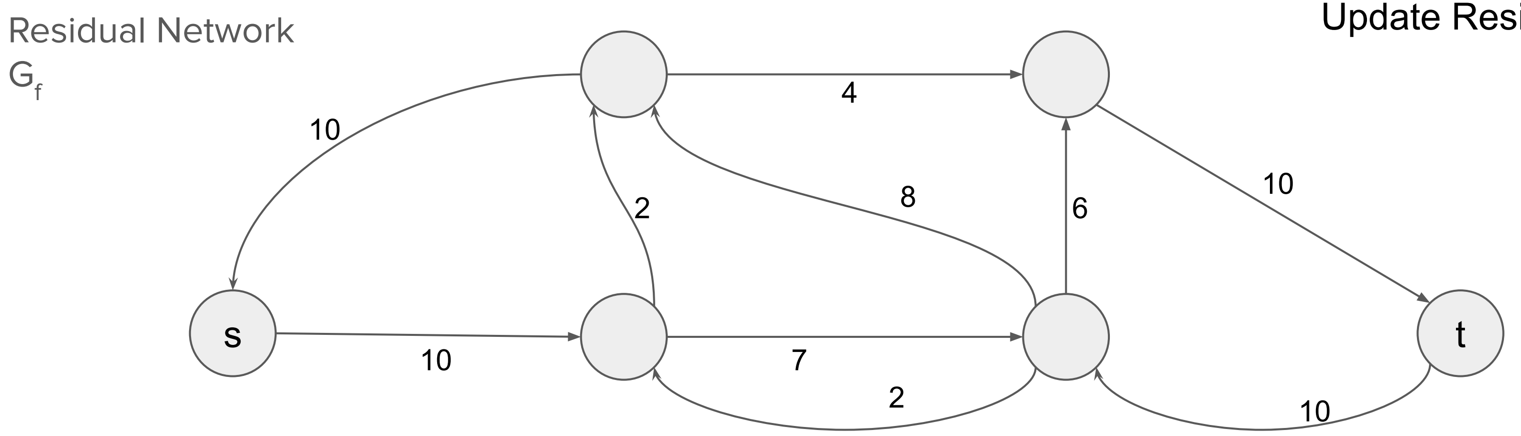


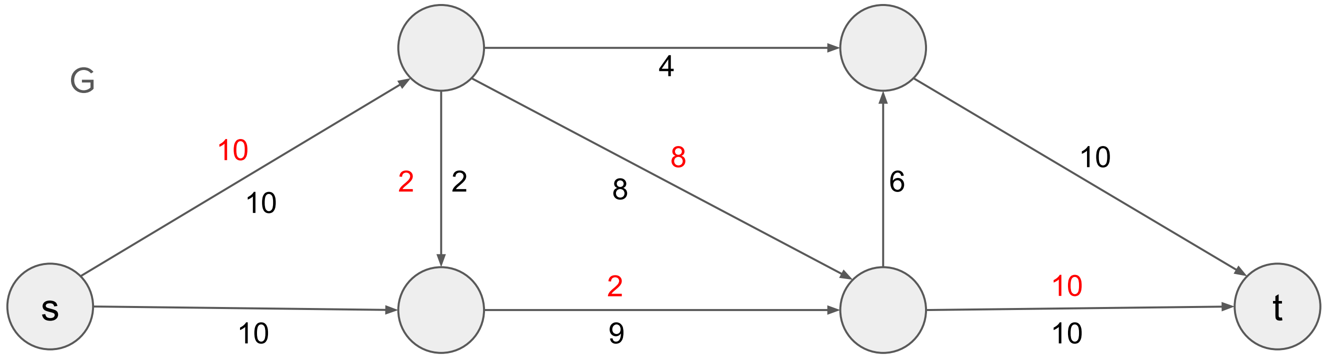


Find an Augmenting Path
Bottleneck = 2

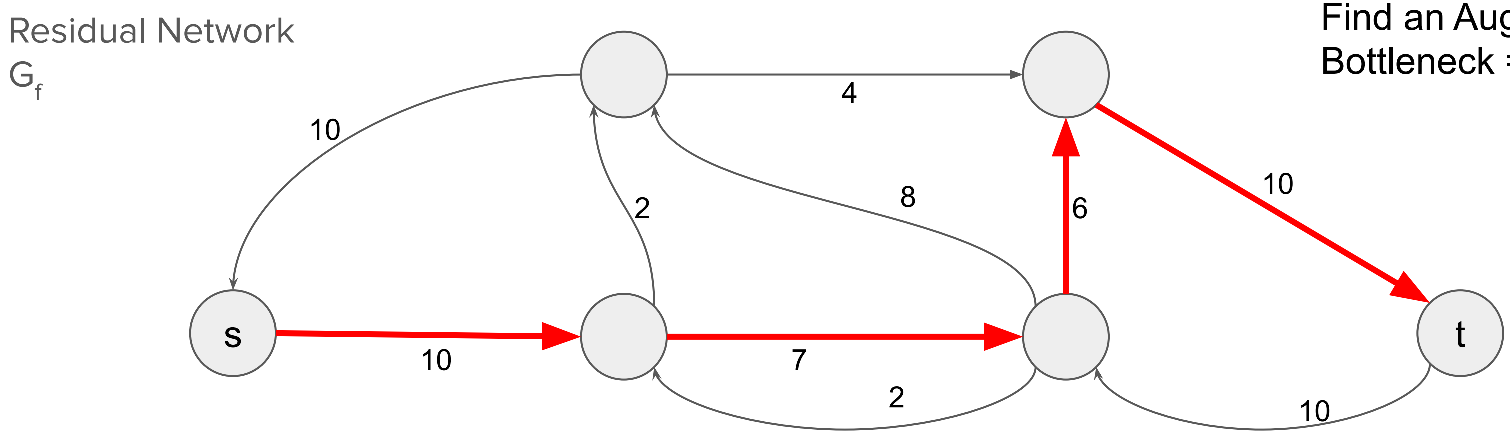
Update Residual Network

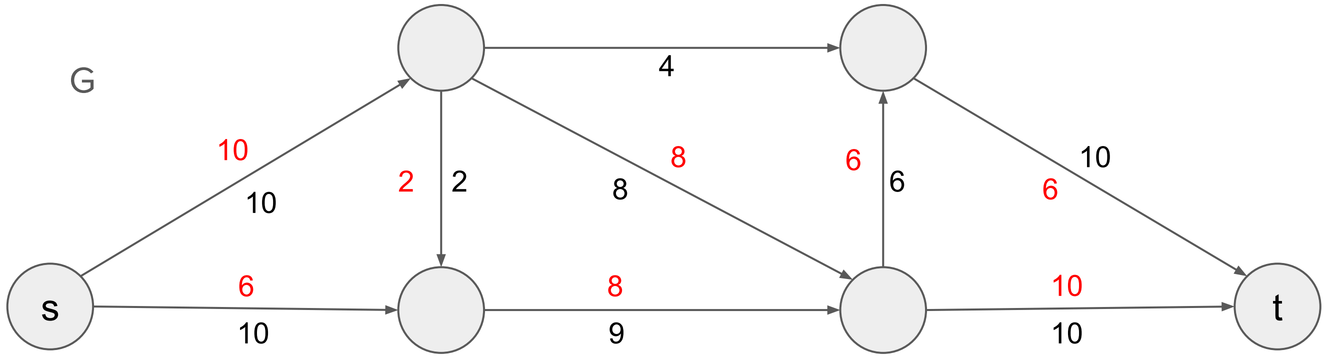




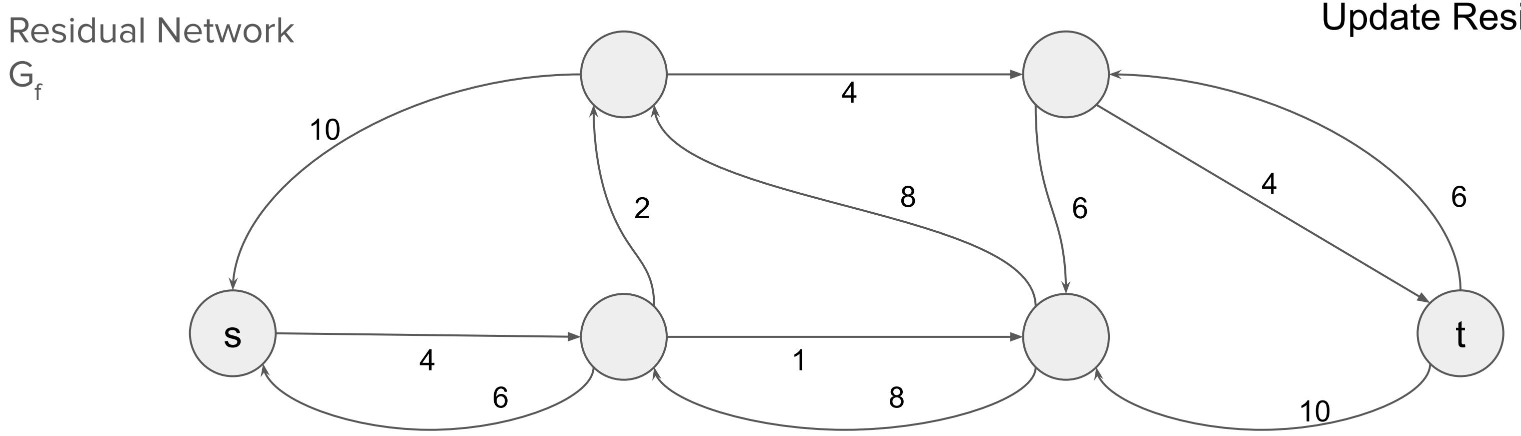


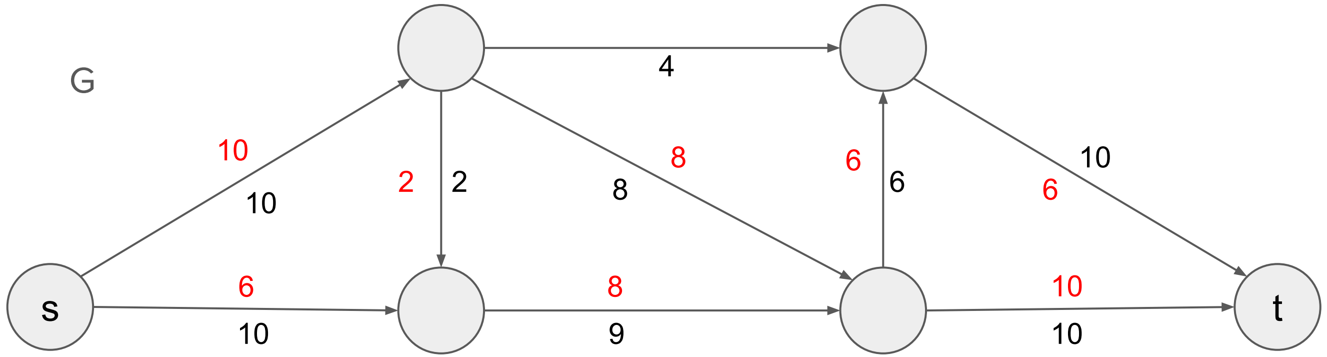
Find an Augmenting Path
Bottleneck = 6

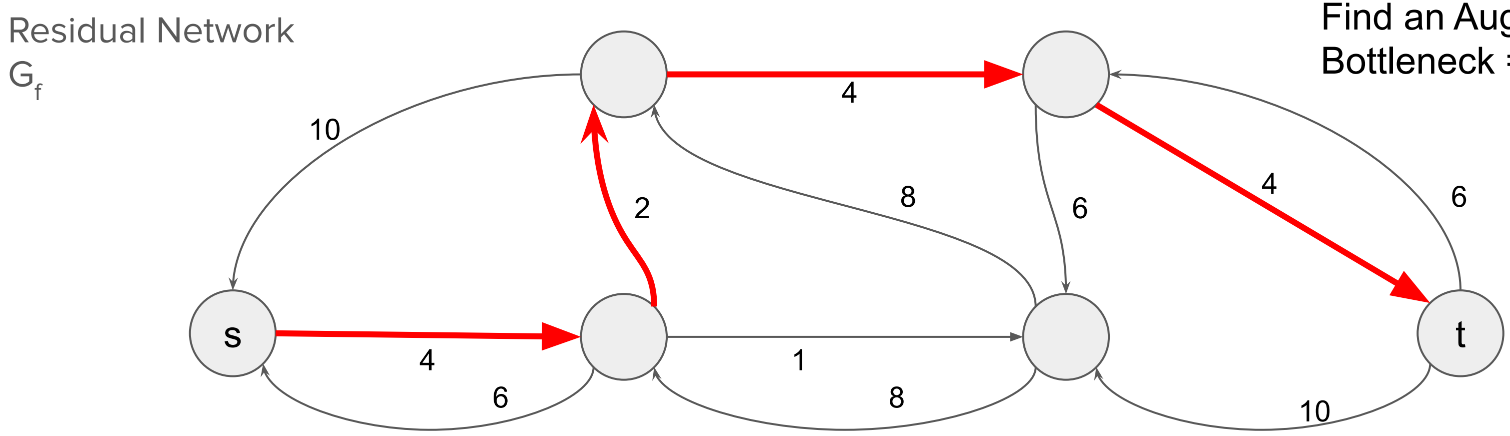




Update Residual Network



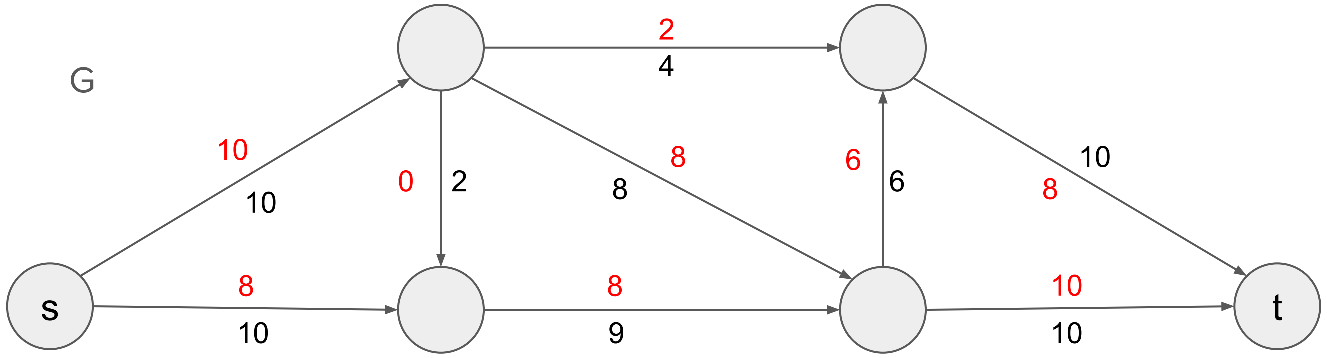




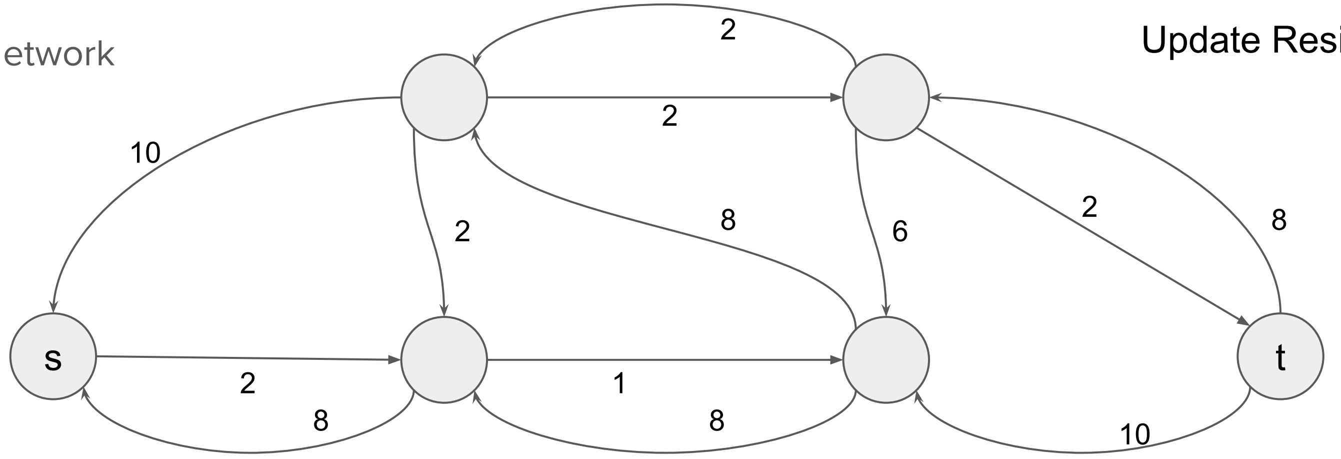
Find an Augmenting Path
Bottleneck = 2

Update Residual Network

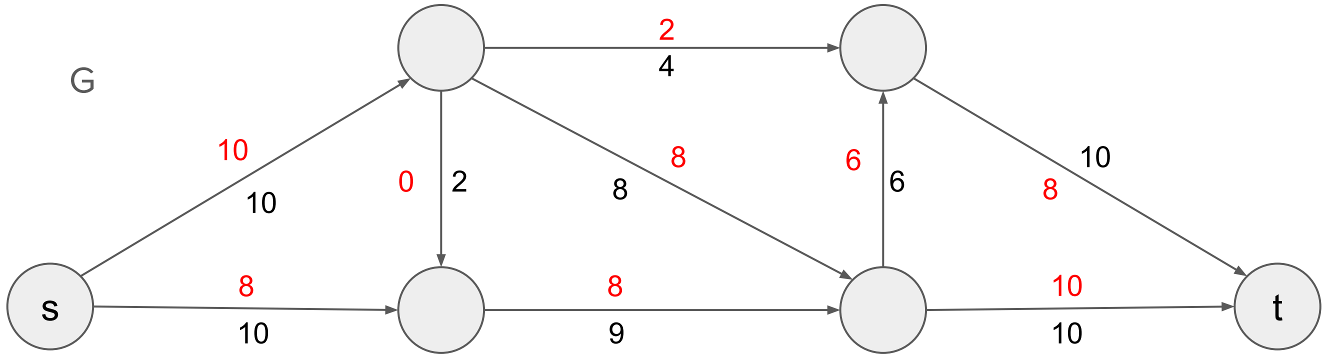
G



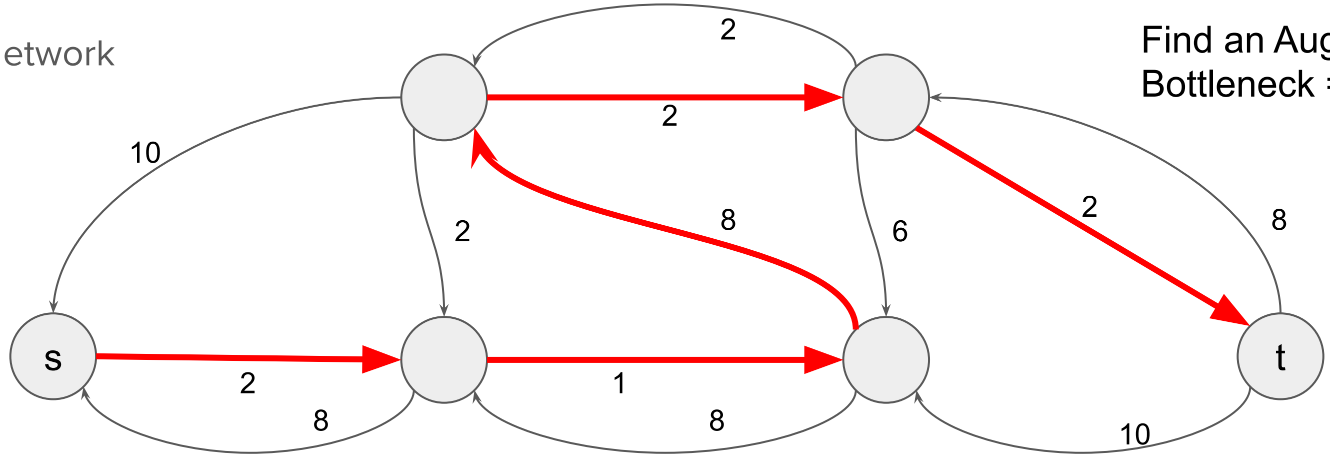
Residual Network G_f



G

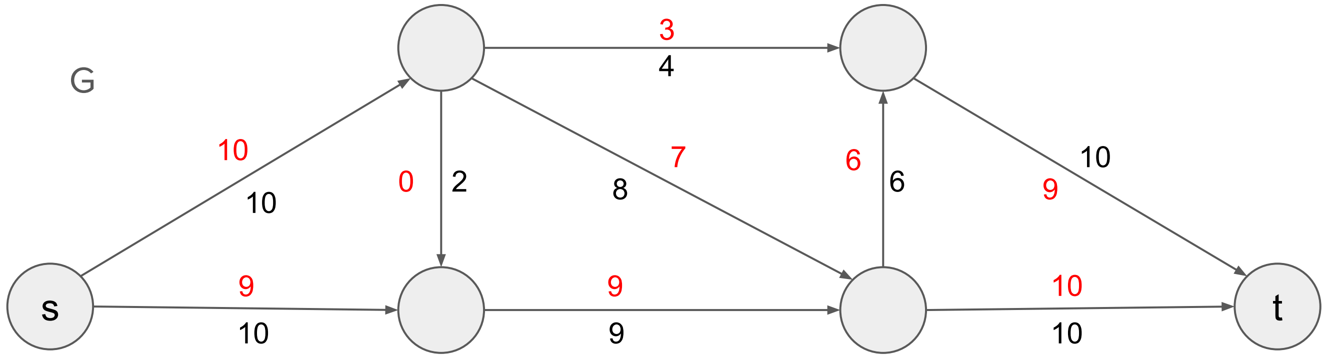


Residual Network
G_f

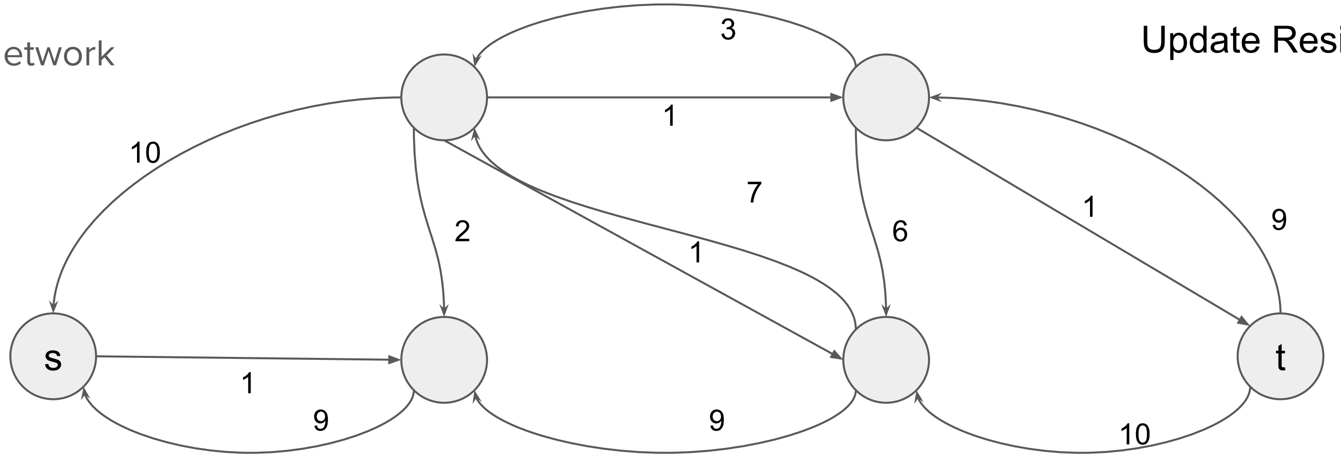


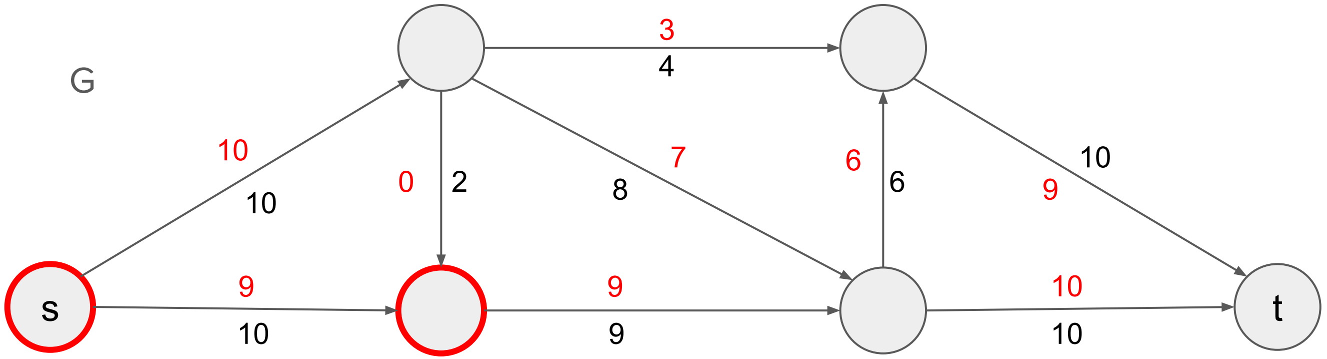
Find an Augmenting Path
Bottleneck = 1

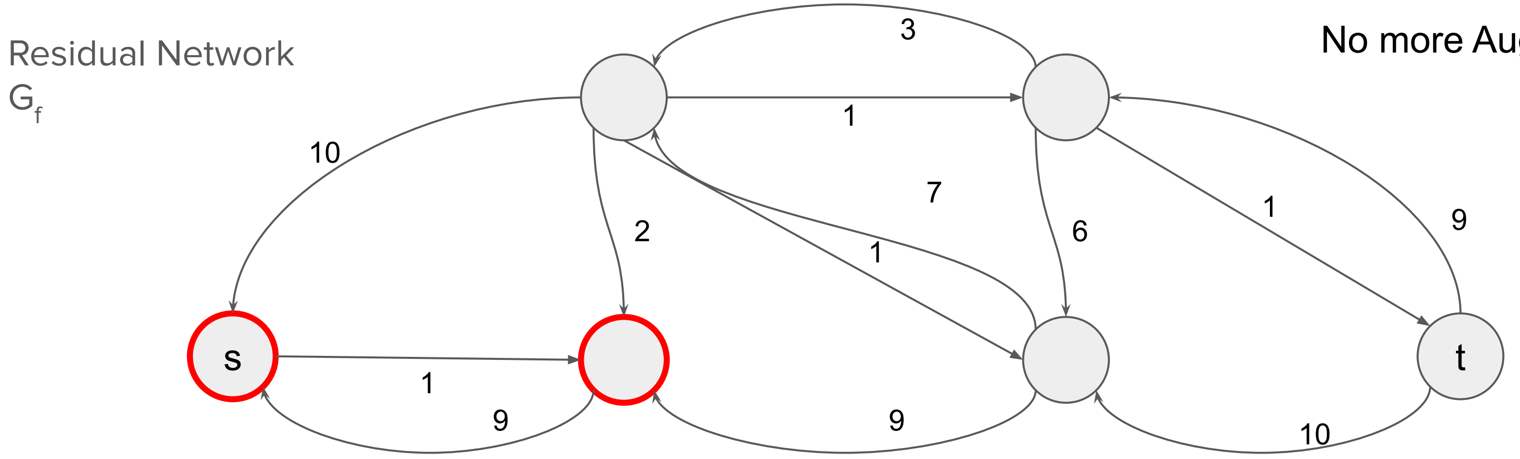
Update Residual Network



Residual Network G_f

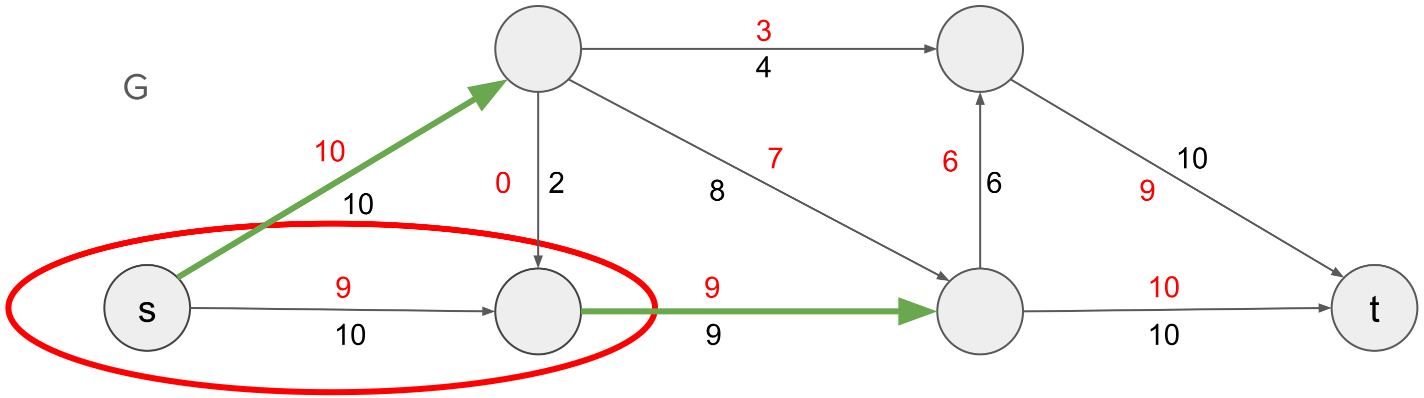






No more Augmenting Paths

Ford Fulkerson Walkthrough



From the Residual Network, we have found the min cut.
The flow going out of the min cut is 9 + 10 = 19.
Thus, max flow is 19.

Ford Fulkerson Applications

We can use the algo to compute things like “Maximum edge-disjoint paths” of a graph from s to t. How do we do it?

- Set each edge to capacity 1, and then run max flow.

We can use the algo to compute “Maximum vertex-disjoint paths” of a graph from s to t as well!

- Turn each node into 2 nodes with a single edge of capacity 1, and all original edges with infinite capacity. Run max flow.

Let’s do an example of internship matching for CS students

- Given a list of CS students, a list of possible companies and the number of internship positions for each, and the list of which students have been accepted to each internship, determine the maximum number of students which can have internships this summer.

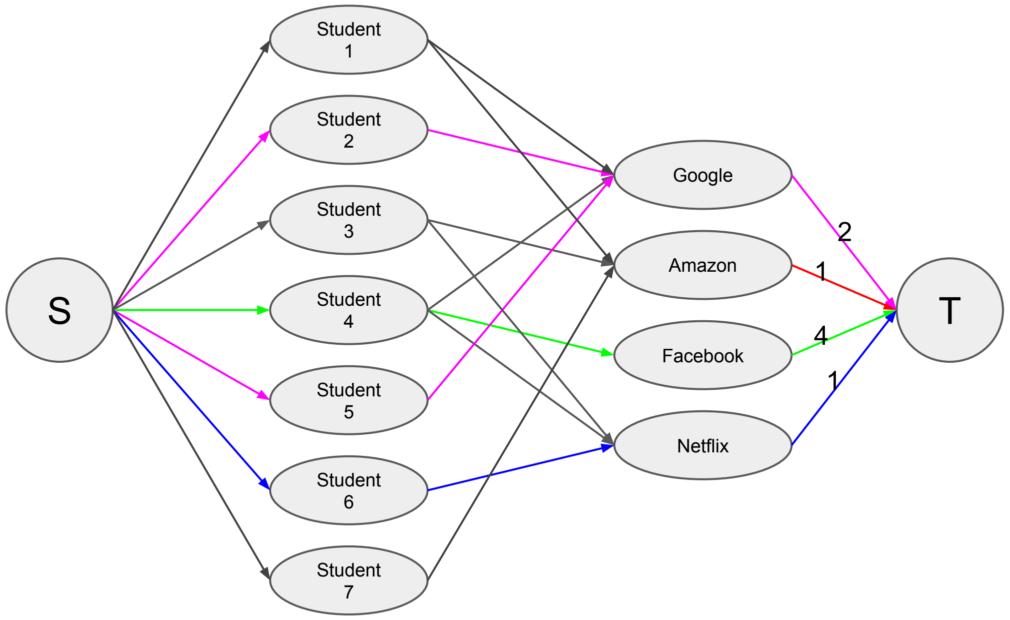
Ford Fulkerson CS Internship Matching Example

Input

Student	Offers
Student 1	Google, Amazon
Student 2	Google
Student 3	Amazon, Netflix
Student 4	Google, Facebook, Netflix
Student 5	Google
Student 6	Netflix
Student 7	Amazon

Company	# Positions
Google	2
Amazon	1
Facebook	4
Netflix	1

Max Flow Graph



*Unlabeled edges have weight 1

Ford Fulkerson Applications

- Observe that the question can be seen as a bipartite matching between two sets, then solved through Ford-Fulkerson
- Create two disjoint sets, one of students, one of internship positions
- Make an edge from each student to each internship they're accepted to of capacity 1
- Create a source node and edges to all students of capacity 1, and a target node with edges to it from all internship positions of capacity number of positions available
- Run Ford-Fulkerson and return the max-flow
- The edges which are filled up represent the matches

NP-Completeness

NP-Completeness

P is the set of problems which can be solved in polynomial time.

NP is the set of problems for which a solution can be verified in polynomial time. (P $\subseteq$ NP)
The question of whether P = NP is an important unsolved question in Computer Science.

NP-Complete problems are like the "hardest" problems in NP. Specifically, NP-Complete problems are those whose membership in P would imply P=NP.

Polynomial-Time Reductions

For two problems X and Y, we say that "X is **polynomial-time reducible** to Y" if X can be solved using Y as a subroutine, and that the number of times X calls Y is polynomial in the size of the input to X. We denote that X is polynomial-time reducible to Y as $X \leq_p Y$. This relationship is transitive.

TL;DR: $X \leq Y$ means "X can be solved using Y." And the p stands for polynomial-time.

NP-Completeness Proofs

To prove that a problem is NP-Complete,

1. We show that the problem is in NP (Don't forget this!)
2. We use our problem to solve a known NP-Complete problem (with a polynomial number of calls). i.e. Known NP-Complete problem $\leq_p$ Our problem

Example: Frequency Allocation

- We have a few cellphone towers, T_1, T_2, … , T_N, each of which can send a set of frequencies F_1, F_2, … , F_N. For example: $F_1 = \{75, 80, 85\}$, $F_2 = \{100, 85\}$, … , $F_N = \{100, 106, 89, 80\}$.
- When two cell phone towers are too close to each other, they can't send the same frequency or else there will be interference. (Assume we are given the distances between each pair of towers, and there is some interference distance threshold x)
- Question: Can we assign a frequency to each cell phone tower from its set so that there is no interference?

Frequency Allocation - Is it NP-Complete?

- Step 1: Check it’s in NP.
  - How can we verify a solution (an assignment of a frequency to each tower) in polynomial time?

Frequency Allocation - Is it NP-Complete?

- Step 1: Check it's in NP.
  - For each tower, we check that its assigned frequency is in the tower's set of allowable frequencies (linear time in the number of towers).
  - For each pair of towers, we check that if their distance is less than x, then their frequencies are different (quadratic time in the number of towers).
- Step 2: What NP-Complete problem can we reduce to Frequency Allocation?
  - Hint: We can reduce a graph problem to Frequency Allocation - Connected nodes in a graph can be represented by towers which are close together.

K-Coloring $\leq_p$ Frequency Allocation

- K-Coloring: Given some graph G = (V,E), and some K colors 0,1,...,K-1, we want to allocate a color to each node such that there are no two adjacent nodes that share the same color.
- K-Coloring $\leq_p$ Frequency Allocation
  - We represent each vertex v in G by a cell tower.
  - For each edge e, we define the distance between the towers it connects to be less than x. For all pairs of towers not joined by an edge, we define their distance to be greater than x. We give each tower the SAME set of frequencies: {0,1,...,K-1}
  - If a frequency can be assigned to each tower, this provides a solution to the K-Coloring problem for graph G (with only a single subroutine call)

Independent Set + Vertex Cover

An **independent set** is a set of nodes in a graph such that no two nodes are joined by an edge.

Given a graph G and a number k, does G contain an independent set of at least k?

A **vertex cover** is a set of nodes in a graph such that every edge of the graph has at least one end in S.

Given a graph G and a number k, does G contain a vertex cover of size at most k?

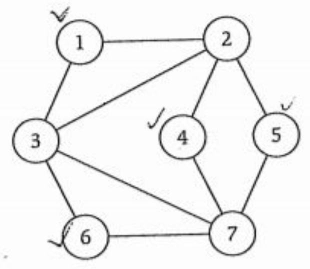


Figure 8.1 A graph whose largest independent set has size 4, and whose smallest vertex cover has size 3.

Independent Set + Vertex Cover

Prove:
Let G = (V, E) be a graph. Then S is an independent set iff V - S is a vertex cover.

Any edge e = (u, v) can only have one node in S. The other is V - S, and this holds true for all edges.

Prove:
Independent Set $<=_p$ Vertex Cover

If we have a black box to solve Vertex Cover, then we can decide if G has an independent set of at least size k by asking the black box if G has a vertex cover of size at most n - k.

Independent Set + Vertex Cover

Prove:

Vertex Cover $<=_p$ Independent Set

If we have a black box to solve Independent Set, then we can decide whether G has a vertex cover of size at most k by asking the black box whether G has an independent set of size at least n - k.

What does this mean?

Though we don't know how to solve each one efficiently, if we solve either one efficiently given an efficient solution to the other, so they are both the same level of difficulty.

These problems are NP-Complete. As of now, they have no known efficient algorithms.

Good luck!

Sign-in https://tinyurl.com/upe180final
Slides https://tinyurl.com/upe180finalslides

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: Zoom
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.