UPE Tutoring:

CS 180 Final Review

Sign-in: https://bit.ly/2F4cKlb

Slides link available upon sign-in

Topics

- **Dynamic Programming Problems Guidelines**
  - Weighted Interval Scheduling
  - Knapsack Problem
- **Network Flow**
  - Ford-Fulkerson
- **NP-Completeness**

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).

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.

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.

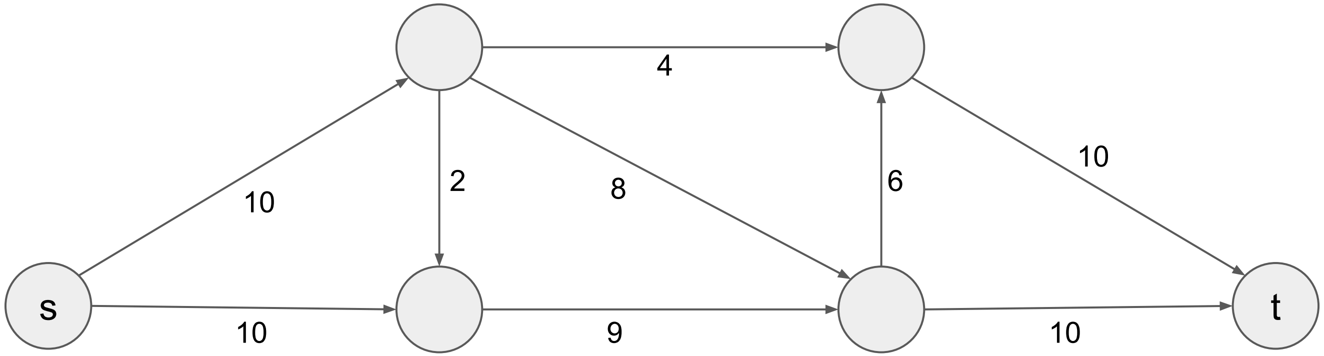
Ford Fulkerson (cont'd)

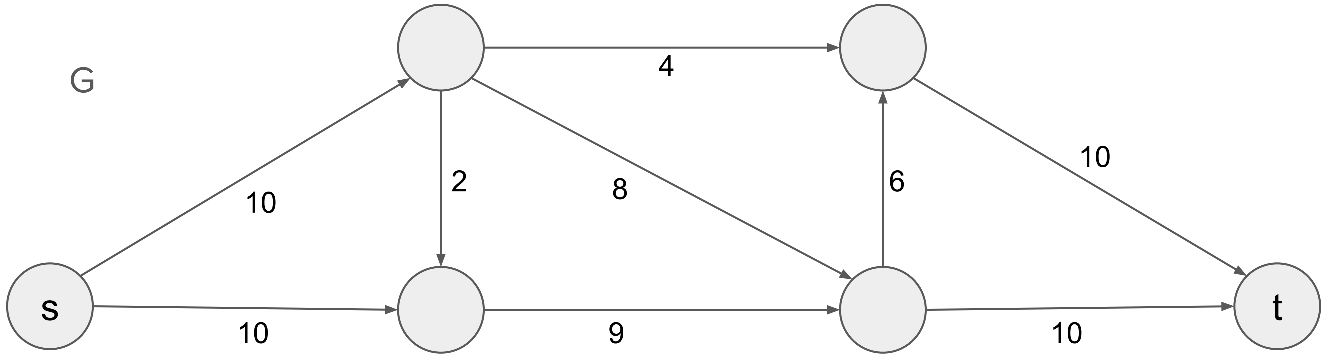
- An (S,T) cut is a cut such that we can partition V in G = (V,E) into S and T (disjoint).
  - Because ford fulkerson starts from some node s, and ends at some node t, we need $s \in S$, and $t \in T$.
- There is a very important theorem: Max-flow, min-cut
  - 1. $|f| = c(S,T)$ for some cut S, T. $c(S,T)$ 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:

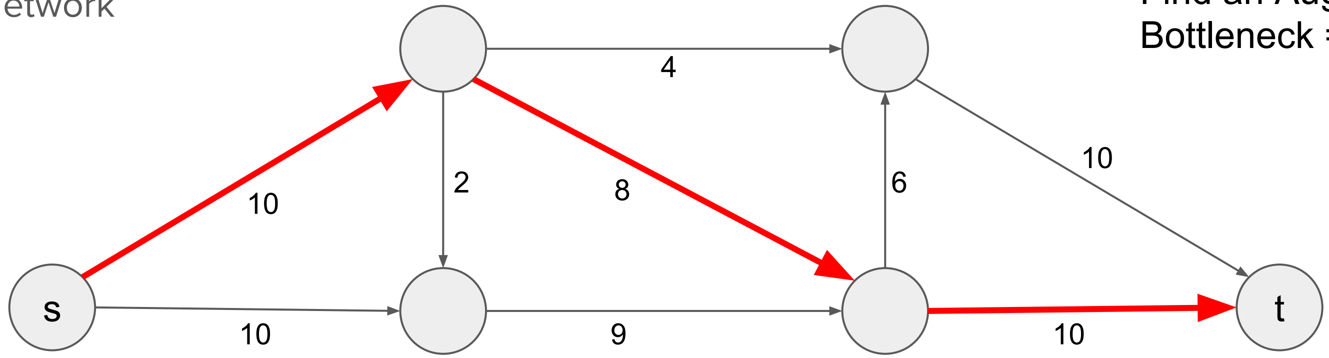
```
While there is an augmenting path f':
    Add f' to f
Return f
```

Ford Fulkerson Walkthrough

Given a graph G:

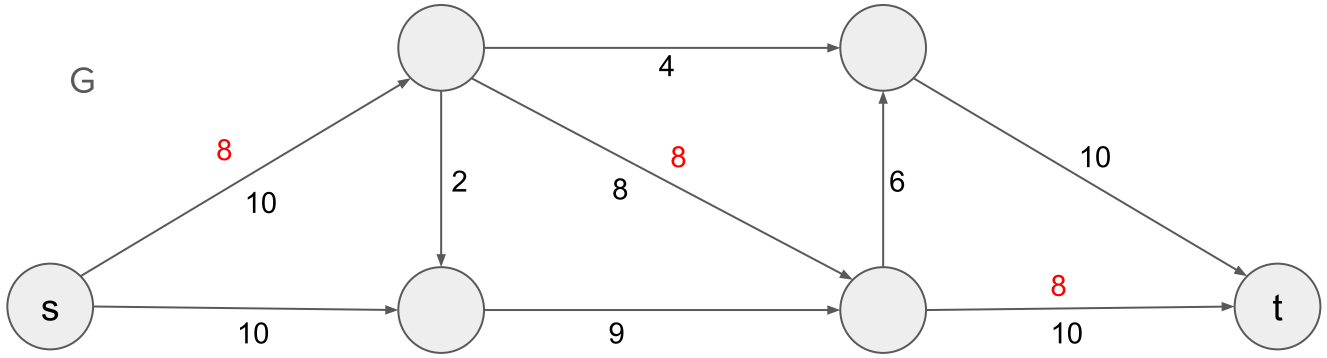


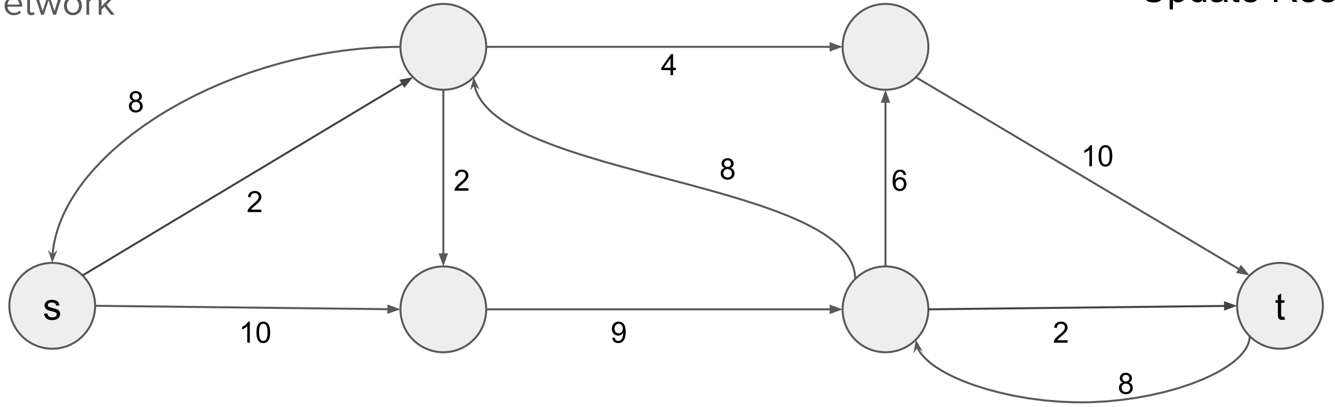


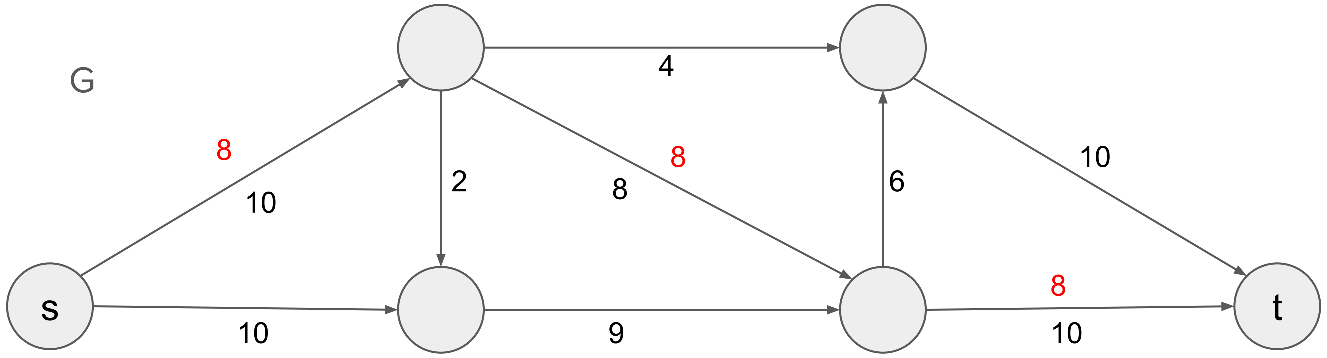


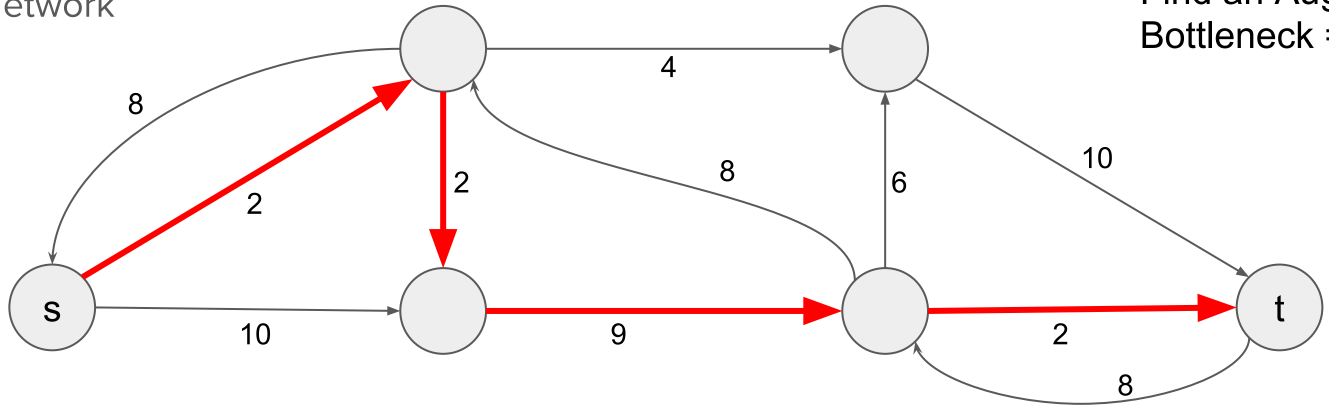
Find an Augmenting Path
Bottleneck = 8

Update Residual Network

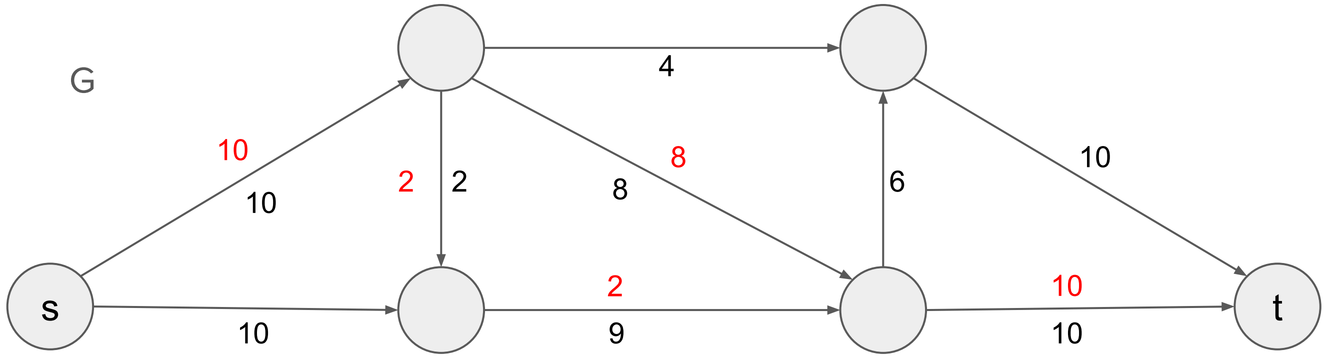


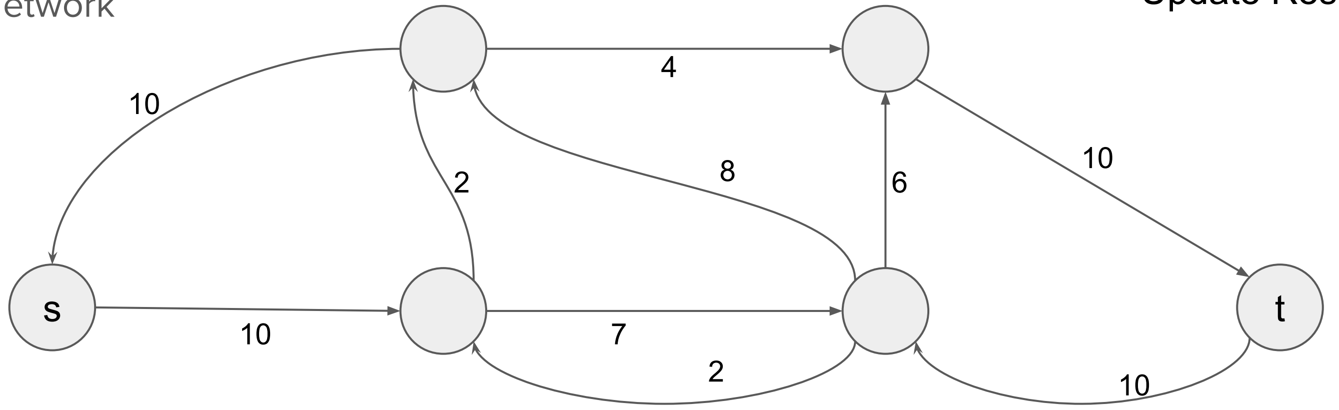




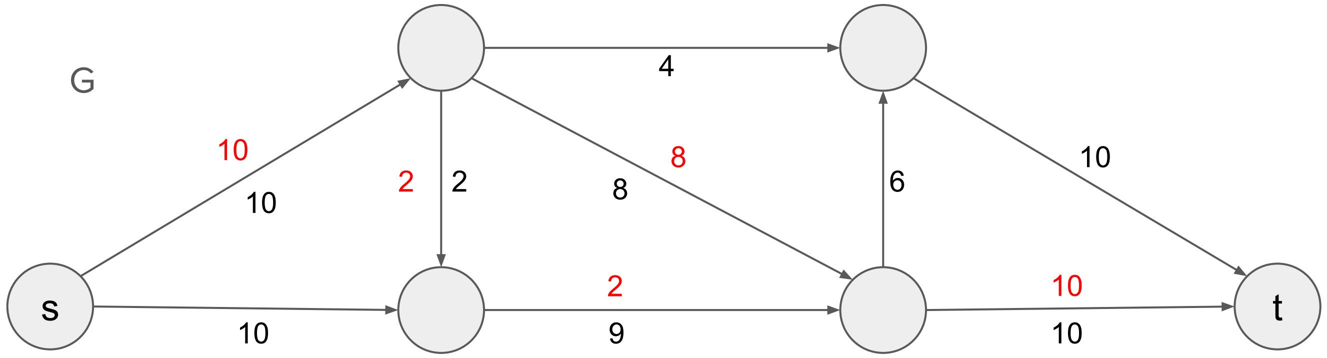


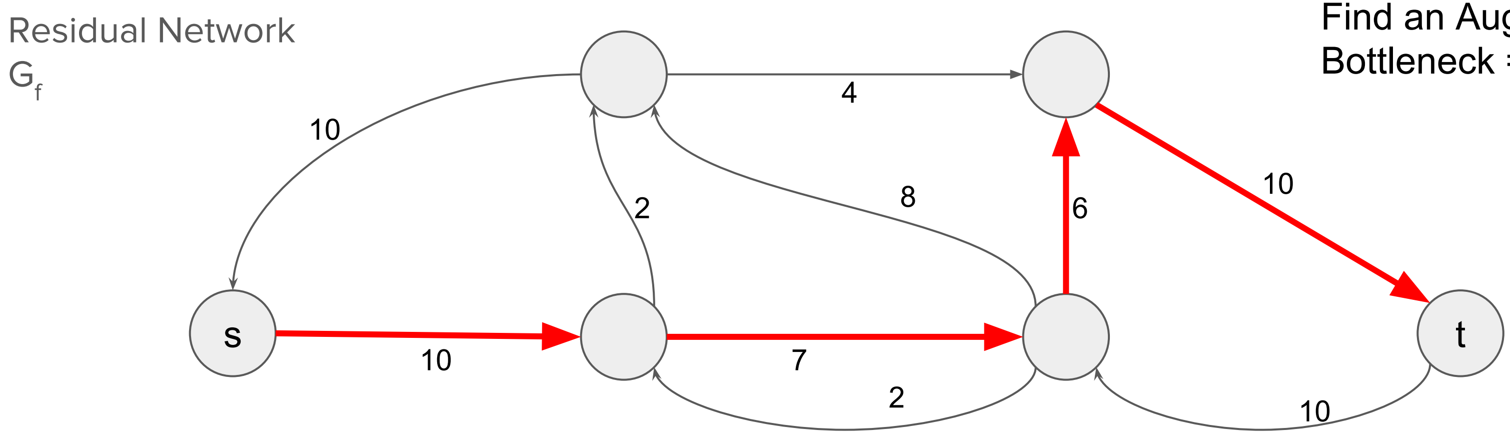
Find an Augmenting Path
Bottleneck = 2





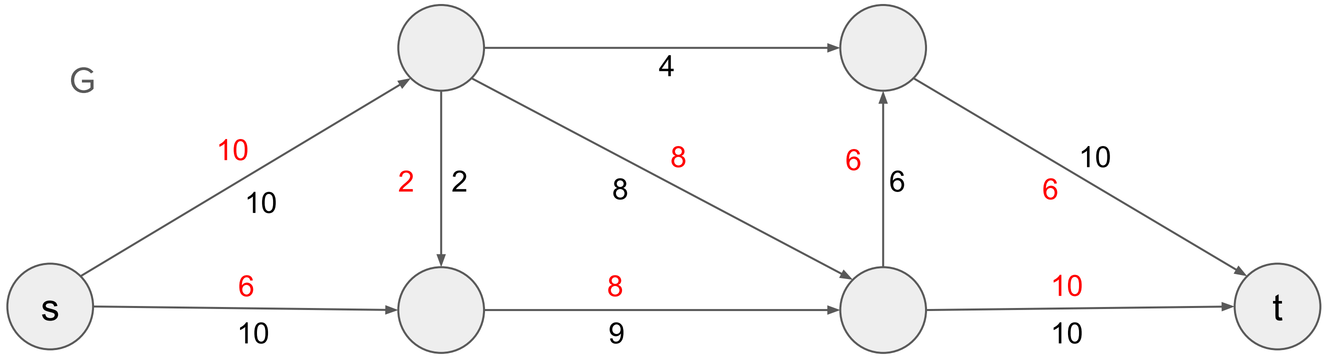
Update Residual Network

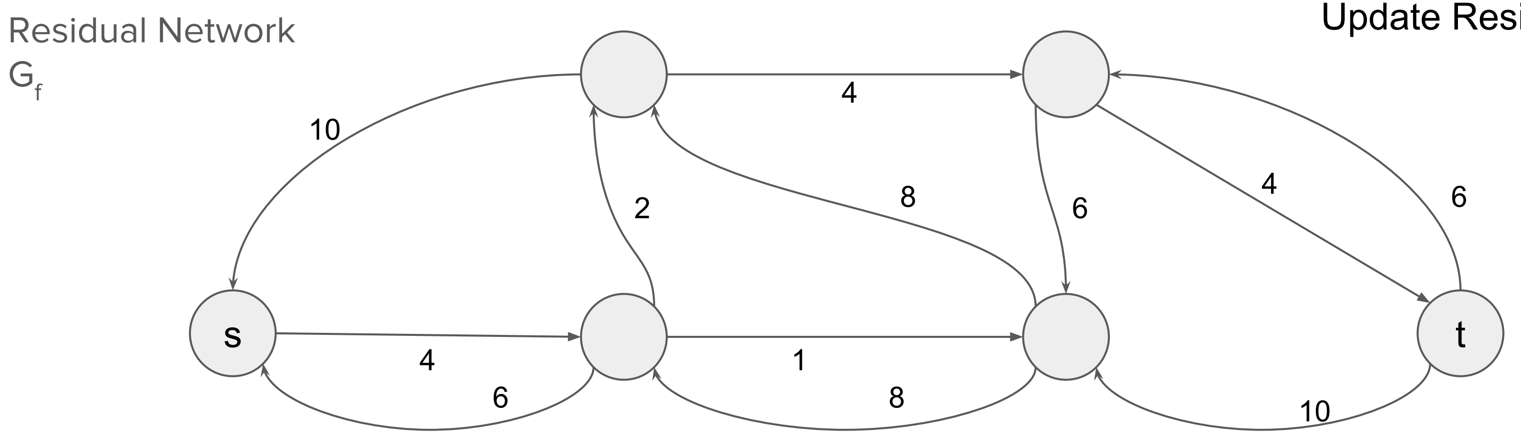




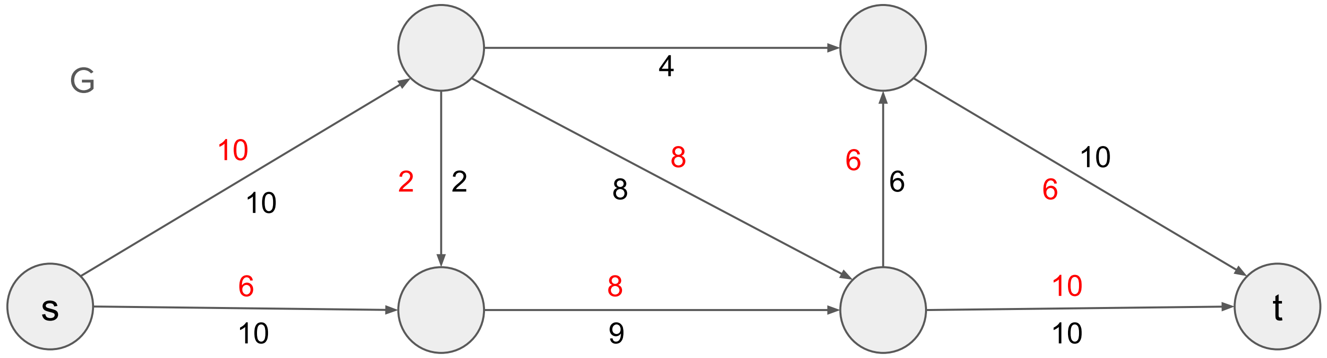
Find an Augmenting Path
Bottleneck = 6

Update Residual Network

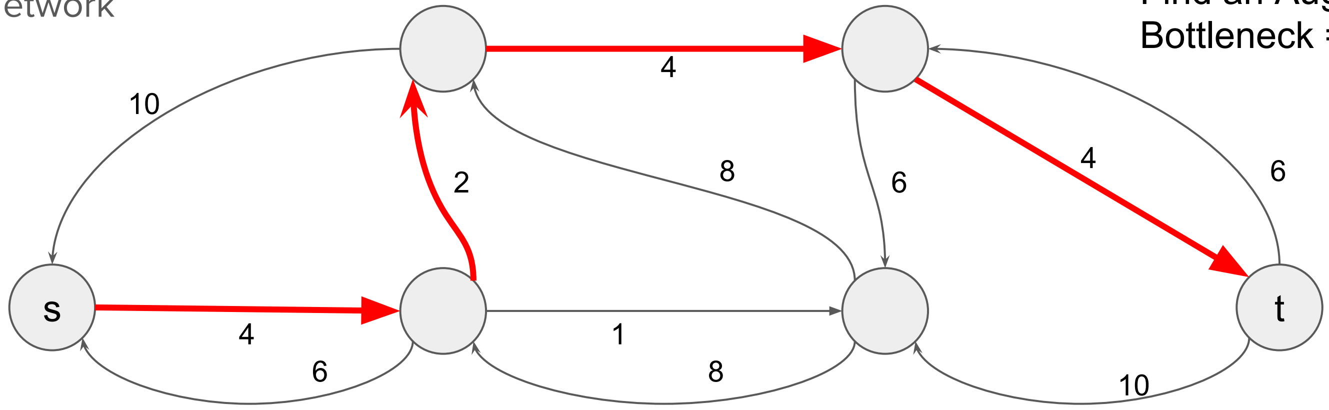




G

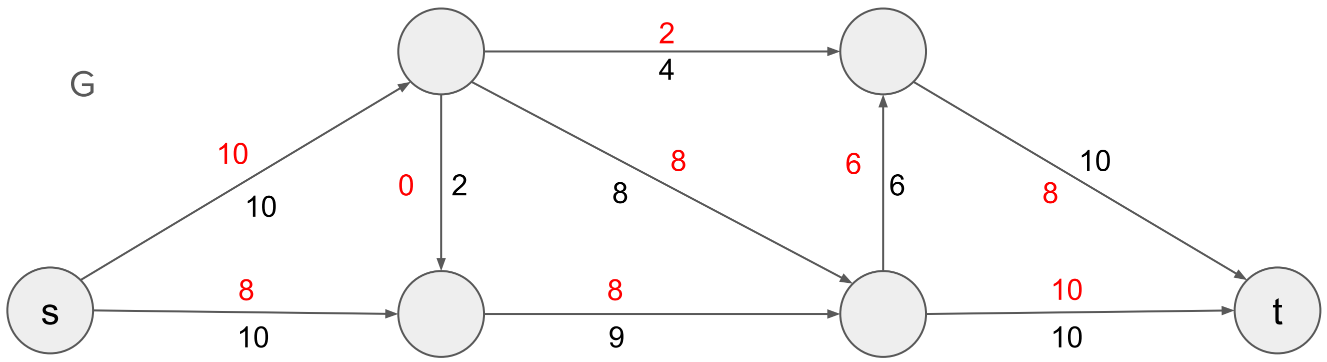


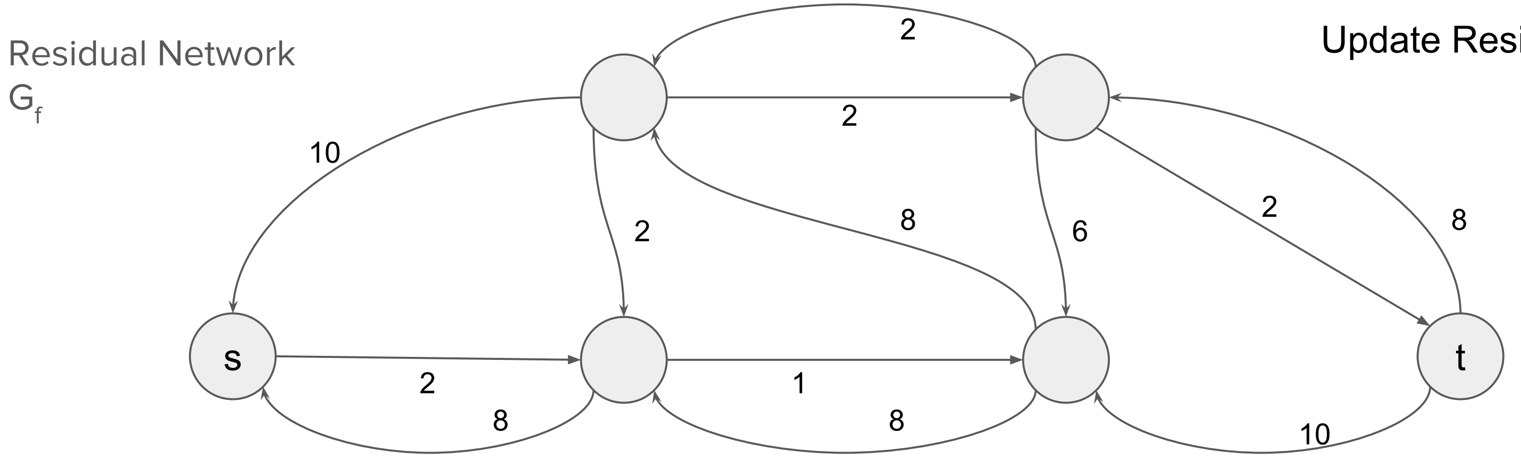
Residual Network
G_f

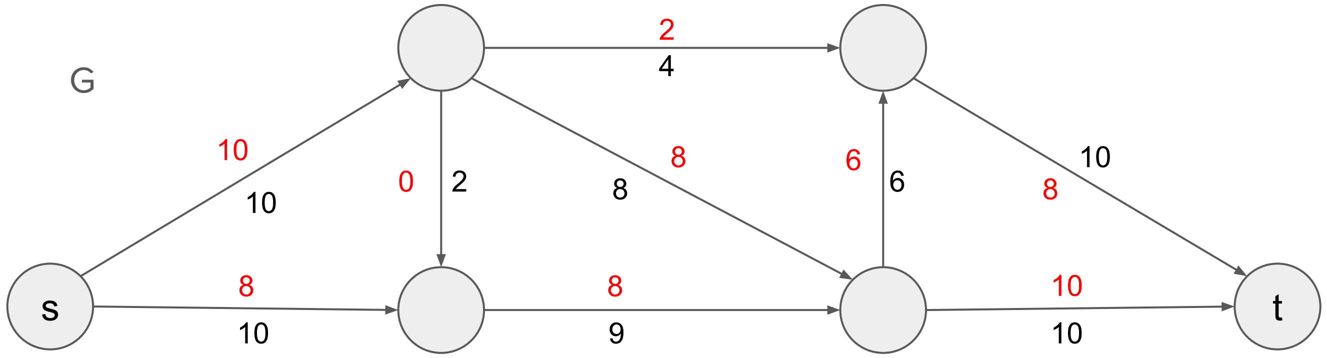


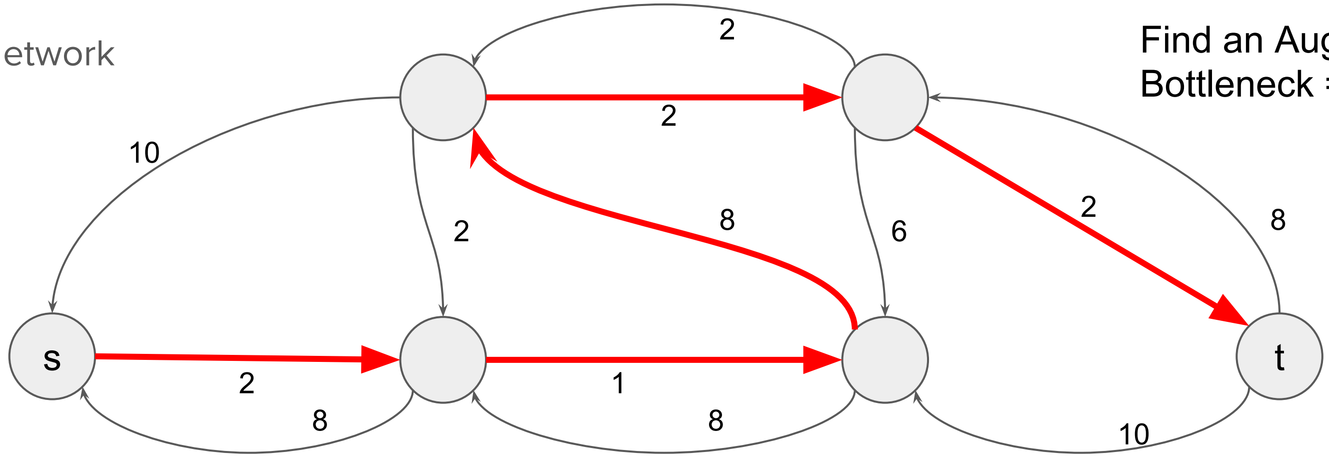
Find an Augmenting Path
Bottleneck = 2

Update Residual Network



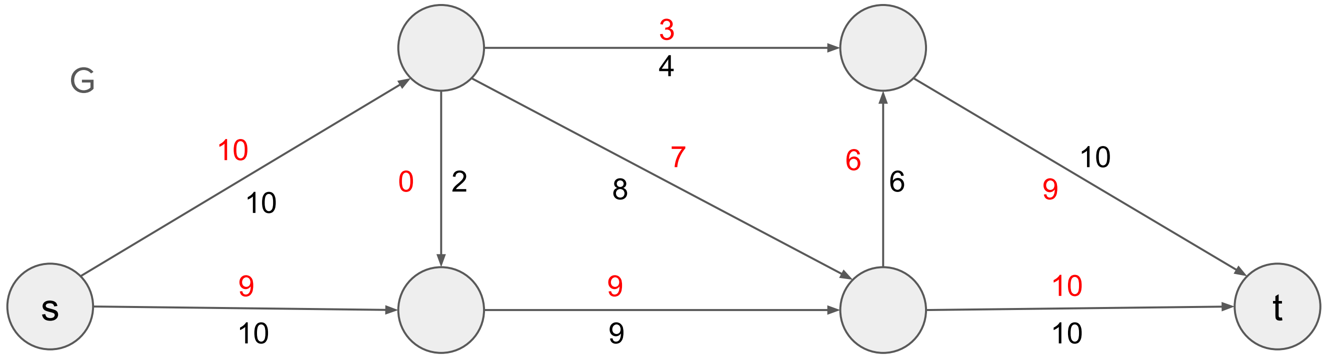






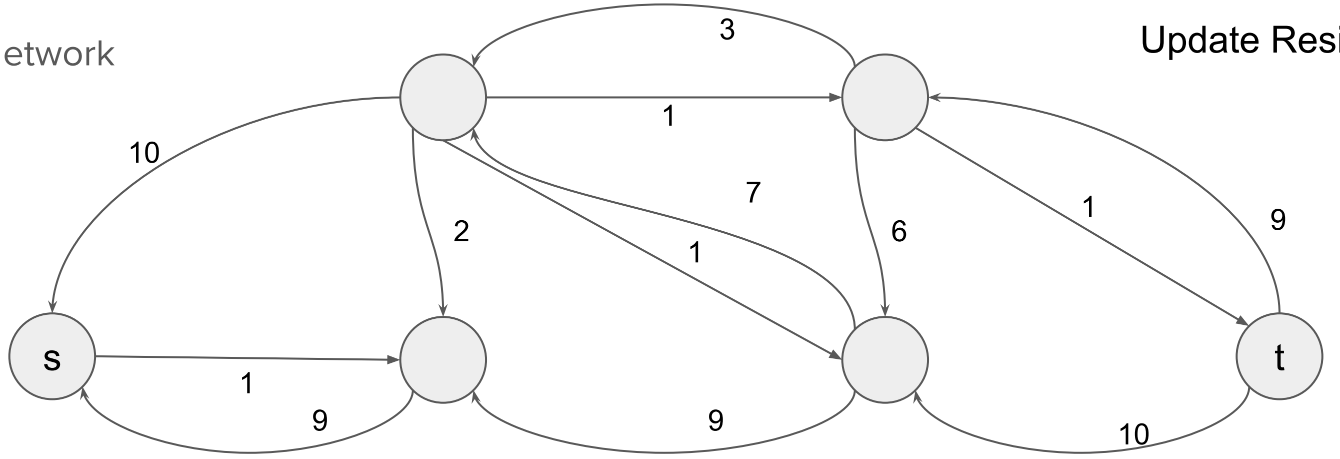
Find an Augmenting Path
Bottleneck = 1

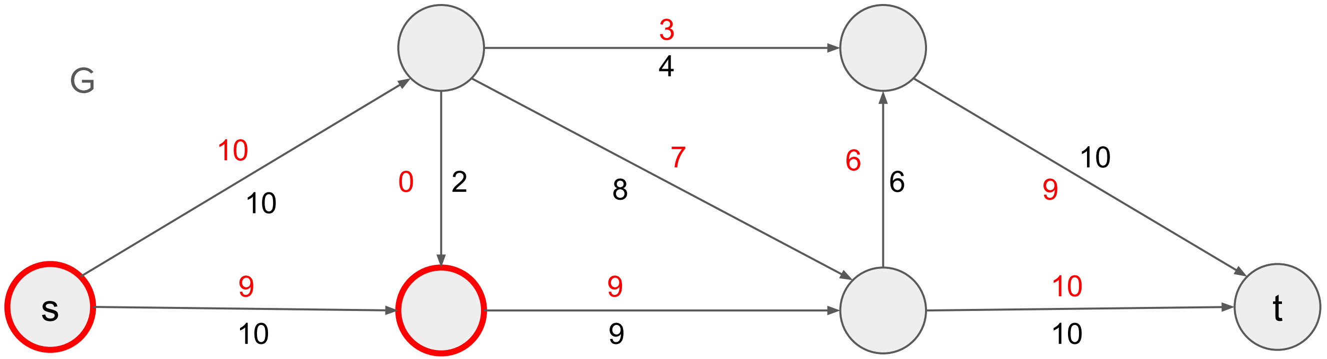
G



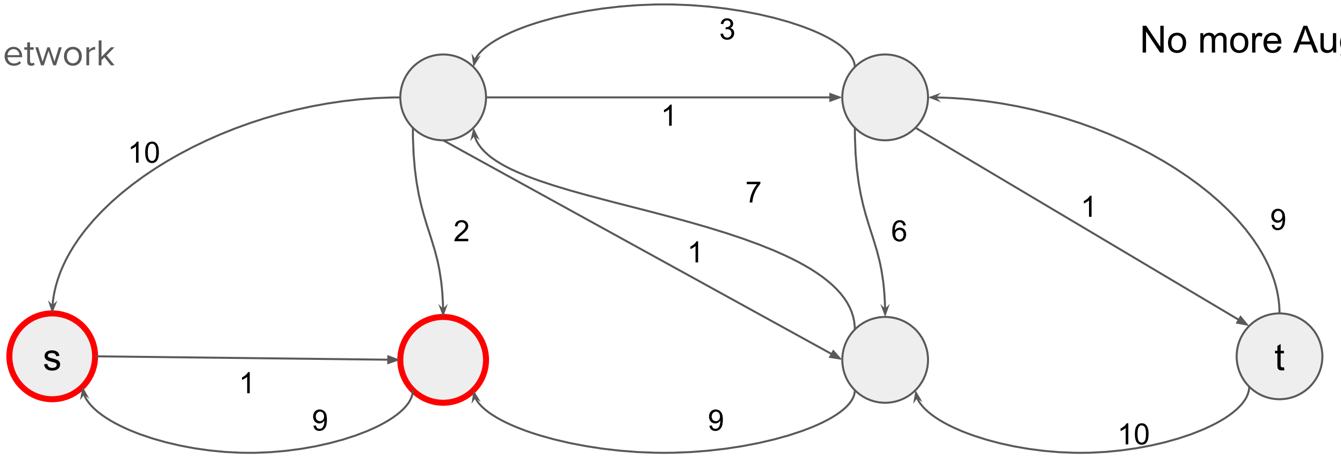
Residual Network G_f

Update Residual Network



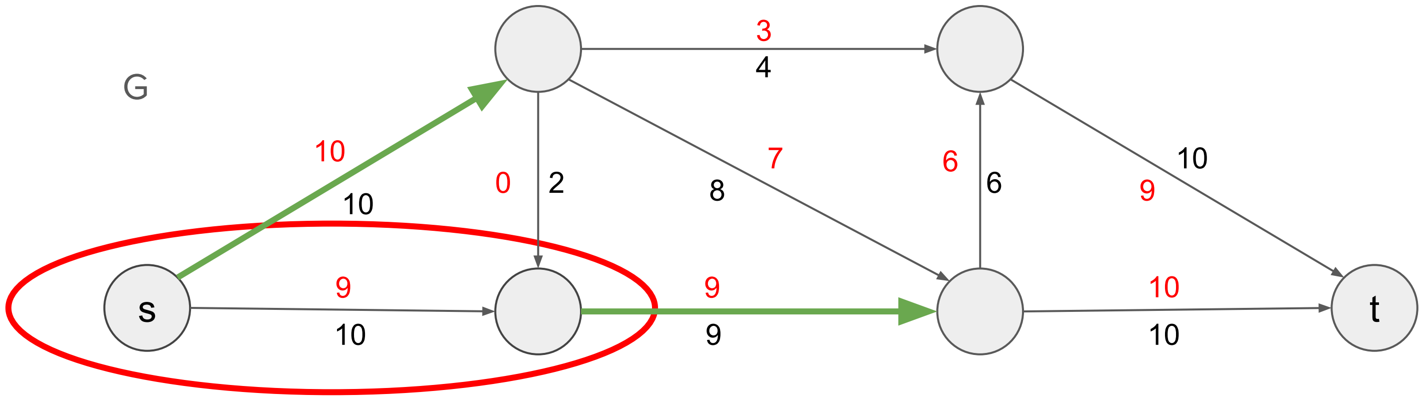


Residual Network G_f



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.

We can use the algo to compute heterosexual Tinder!

- All guys on 1 side, all girls on the other. Connect all guys from s, all girls to t. An edge exists between guys and girls who match each other. All edge capacity = 1. Run max flow.

NP-Completeness

NP-Complete problems are a set of problems such that, if a polynomial-time algorithm for any one of them is found, it would imply the existence of a polynomial time algorithm for all of them.

Problems that are “computationally hard for all practical purposes, though we can’t prove it”.

Polynomial Time Reductions
Assuming that X can be solved in polynomial time, can arbitrary instances of Y be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black box that solves problem X?

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?

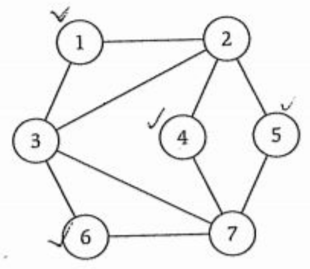


Figure 8.1 A graph whose largest independent set has size 4, and whose smallest vertex cover has size 3.

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 as of now have no known efficient algorithms.

Partition Sums + Load Balancing

Partition: Consider a set of N non-negative integers. It is said that the set can partition if the set can be divided into disjoint sets {x_1,...,x_k} and {x_k+1,...,x_N} and the sum of each set is equal.

Load Balancing: Given M jobs, each with runtime T_i, ask whether all the jobs can be finished in under some T time given you have two machines that can run jobs in parallel.

We know partition is NP-hard. How do we show Load Balancing is NP-hard?

Furthermore, can you give me a 2-approximation for Min-Load-Balancing?

Medium Exam Problem: Hamiltonian Path vs. Cycle

How can you reduce Hamiltonian Cycle into Hamiltonian Path? HamCycle $<=_p$ HamPath

Formally, HamiltonianCycle(G) returns whether there exists a hamiltonian cycle in G. HamiltonianPath(v1,v2) returns whether there exists a hamiltonian path starting from v1 and ends at v2.

To reduce to Hamiltonian Path, it means you can solve Hamiltonian Cycle problem on any G = (V,E), (assume undirected) by calling Hamiltonian Path a polynomial amount of times with respect to G.

You can also do other things like make insane-looking gadgets to make the reduction work, as long as it does not expand the input size exponentially.

Solution (one possible sol.):

Take any edge = {v1, v2} in E. Run HamiltonianPath(v2, v1) and see if it is true. If it is true, then we have some path v2 -> v1. If this edge is in the path, then the graph only contains v1, v2(exercise). If this edge is not in the path, then add this edge to the path to get a cycle, since v1->v2->v1 is a cycle.

Do this for every edge in the graph (linear with respect to the input), and we have if any of them can satisfy the above condition we have HamiltonianCycle(G) = true. Else, it must be false.

<u>However, can you come up with a solution that calls HamiltonianPath ONLY ONCE?</u>

Medium Exam Problem Follow-Up

Given a HamiltonianCycle(G) function that outputs “yes” or “no”, come up with an efficient algorithm to extract the actual cycle given oracle access to HamiltonianCycle.

Hint: Fixed points.

Hard Exam Problem (Discuss amongst your peers)

Setup:
As we all know, traveling salesman problem is NP hard. However, there exists ways to approximate solutions to an NP hard problem. For an algorithm to give the optimal solution(denote optimal distance as d*), we define it as a 1-approximation. Such algorithms run in exponential time or larger for NP hard problems. For a algorithm for TSP to be K-approximation(for K >= 1), we require the solution from the algorithm, d, to satisfy:

$$d^* < Kd$$

Given a complete undirected graph G = (V,E=VxV) where f({v1,v2}) = distance between v1 and v2, and f({v1,v2}) <= f({v1,v3}) + f({v2,v3}) for all v1,v2,v3. Give a 2-approximation for traveling salesman on G.

Hint: Use some existing graph algorithm

You’ve seen algorithms like BFS, DFS, Dijkstra’s, Prims/Kruskals, Ford Fulkerson, etc etc. Try to see if you can construct a very **greedy** solution to this problem.

BTW, we call this graph a graph induced on the Euclidean metric space. The inequality:

$$f(\{v1,v2\}) <= f(\{v1,v3\}) + f(\{v2,v3\})$$

Is the **triangle inequality**.

Solution (Using minimum spanning trees):

Some notation: sum is a function that sums all the edge weights. MST is a function that takes a graph and outputs a minimum spanning tree. ETSP is a function that takes a graph and outputs the graph with only edges included in the optimal ETSP tour.
Recall that the sum of all the weights from MST(G) is minimized such that G is still connected, and that MST(G) is a tree. Then we have that:

sum(MST(G)) <= sum(ETSP(G))

implies 2 * sum(MST(G)) <= 2 * sum(ETSP(G))

Then we already have something that's a 2-approximation, but we can't use the same edge twice. Root the MST, perform any pre/in/post-order traversal and enumerate the nodes. We can get from any node to another because it is a complete graph, and any traversal that is not a part of the MST must be shorter than taking any path from the tree to the next node due to triangle inequality on edge(s). We have a hamiltonian cycle with weights less than 2 * ETSP(G).

Harder Exam Problem (Discuss even more)

Given that we know a 2-approximation exists like the construction above, can we construct a **1.5-approximation** E-TSP?

Hint: What was the major issue with previous E-TSP approximation that made it 2-approximation?

Harder Exam Problem Hints

Hint: There exists a polynomial algorithm to solve perfect matching of a set of nodes, i.e. maps from V to {(v1,v2),...,(vn-1,vn)} where the sum of the edge weights between the nodes in pairs are minimized.

Hint Hint: There exists a polynomial algorithm to solve Eulerian tour

Good luck!

Sign-in	https://bit.ly/2F4cKIb
Slides	https://bit.ly/2SVF1h6

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
    - Location: ACM/UPE Clubhouse (Boelter 2763)
    - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.