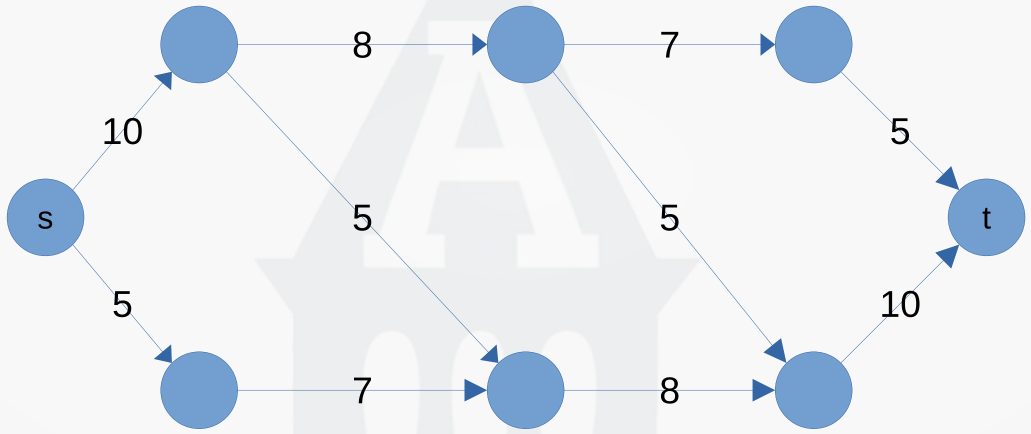
Graphs – Network Flow

Maximum Network Flow Problem

- How can we maximize the flow in a network from a source, or set of sources, to a destination or destinations?
- Supposedly rose to prominence in relation to the rail networks of the Soviet Union, during the 1950's
  - US wanted to know how quickly the Soviet Union could get supplies through its rail network to its satellite states in eastern Europe
  - US also wanted to know which rails it could destroy to most easily cut off the satellite states
- These two problems were closely related; the first called ***max flow***, the second called ***min cut***
  - Solving the max flow problem also solves the min cut problem
- The first efficient algorithm for maximum flow was conceived by two Computer Scientists, last names of Ford and Fulkerson

Maximum Network Flow Problem

Supply chain



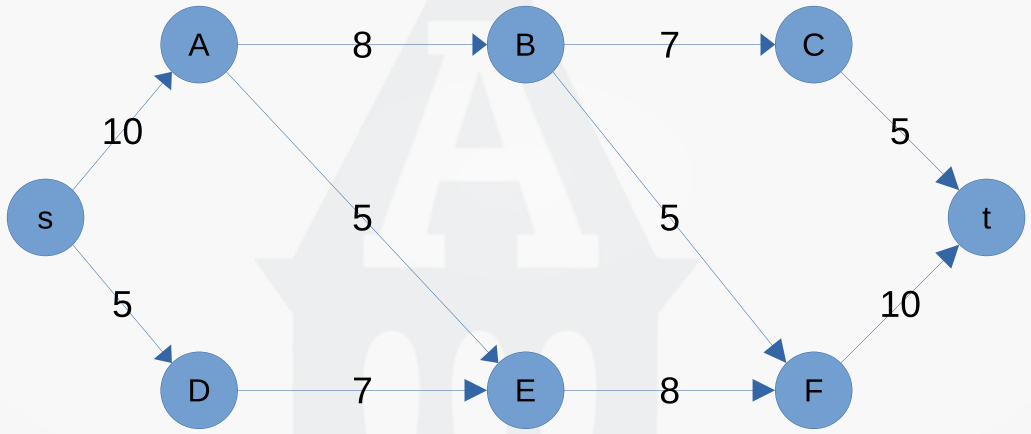
Satellite states

Network Flow

- The network flow problem is as follows:
  - Given a connected directed graph G
    - with non-negative integer weights
    - where each edge weight is the capacity of that edge
  - 2 different vertices, s and t, called the source and the sink
    - such that the source only has out-edges and the sink only has in-edges
  - Find the maximum amount of some commodity that can flow through the network from source to sink

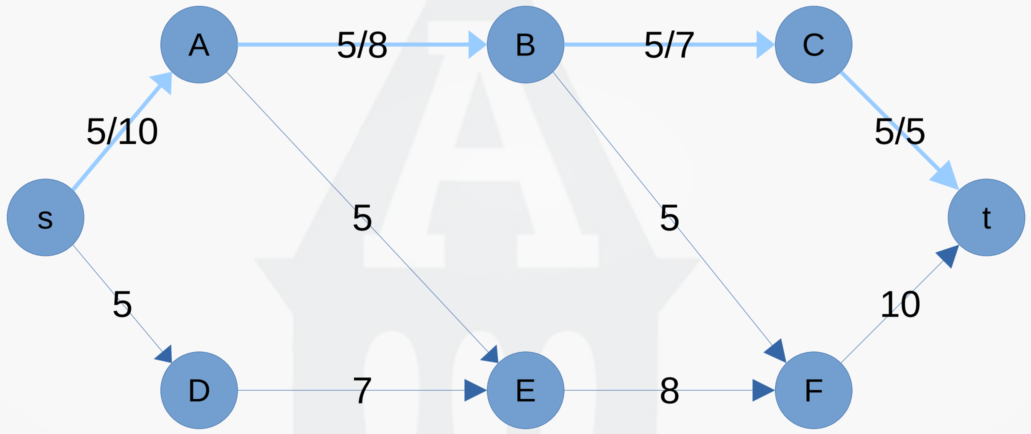
What is the best we can do?

Find a path that carries flow and how much



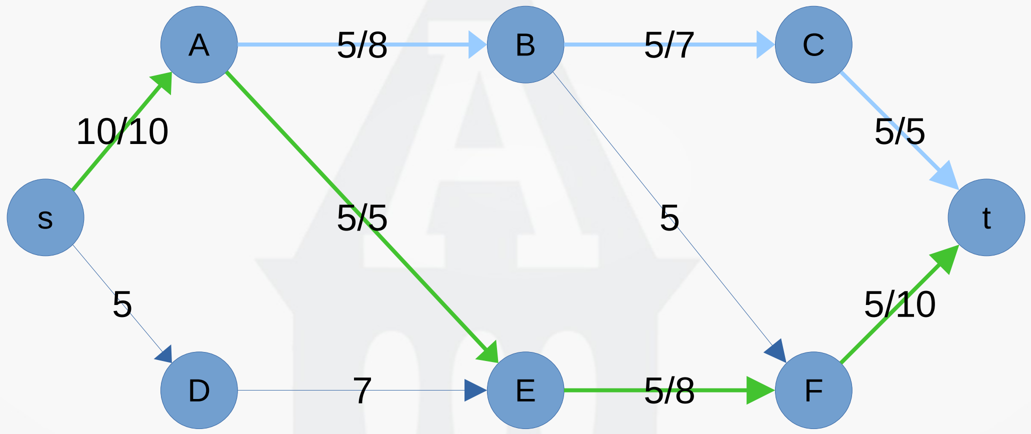
What is the best we can do?

Find a path that carries flow and how much



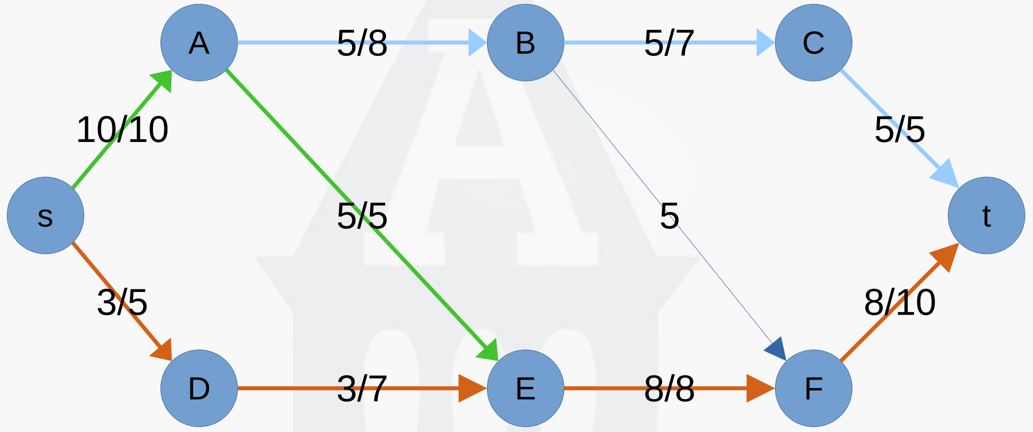
What is the best we can do?

Find a path that carries flow and how much



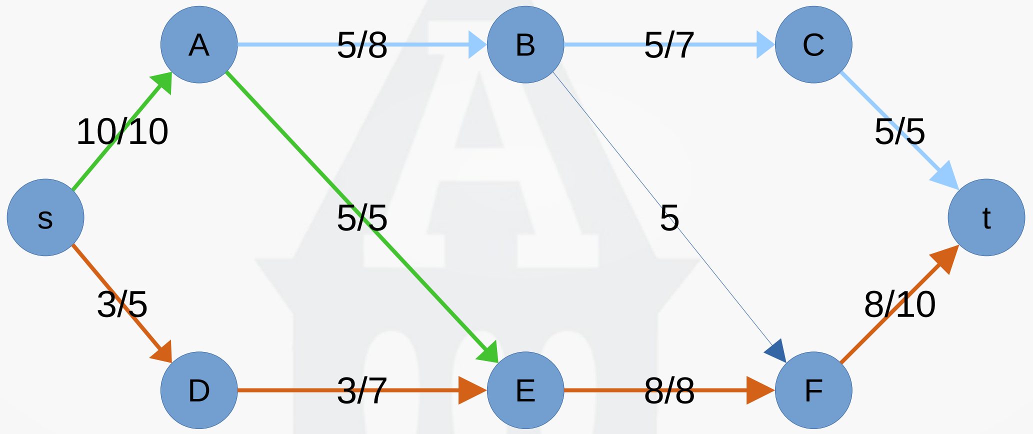
What is the best we can do?

Find a path that carries flow and how much



What is the best we can do?

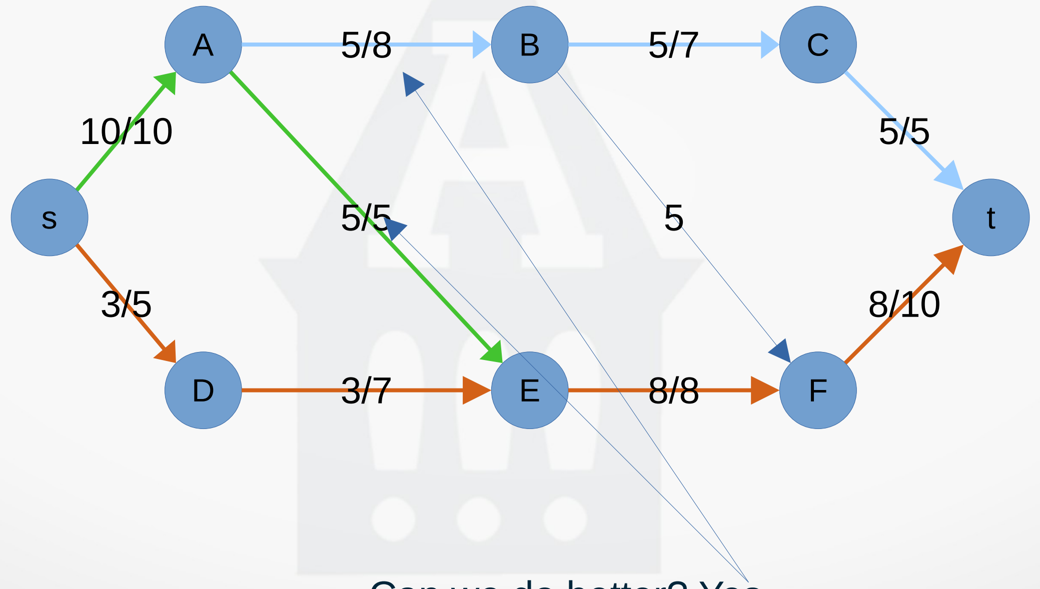
Find a path that carries flow and how much



Can we do better?

What is the best we can do?

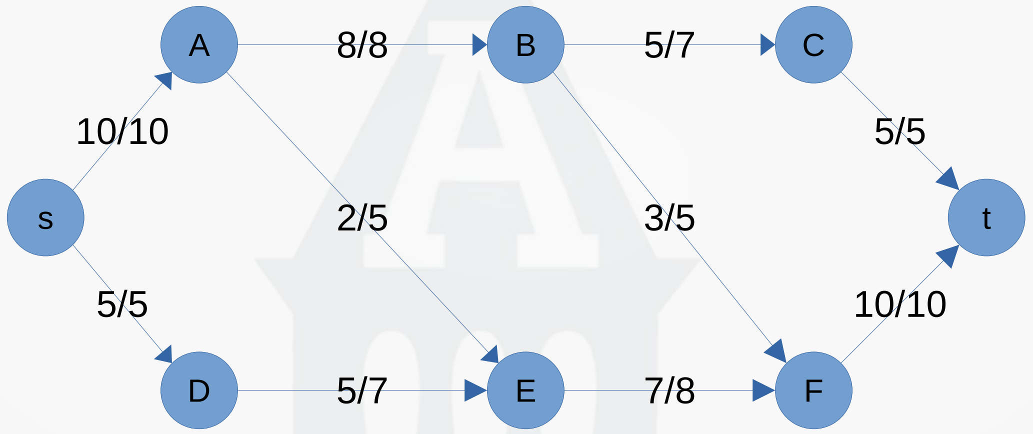
Find a path that carries flow and how much



Can we do better? Yes

What is the best we can do?

Find a path that carries flow and how much

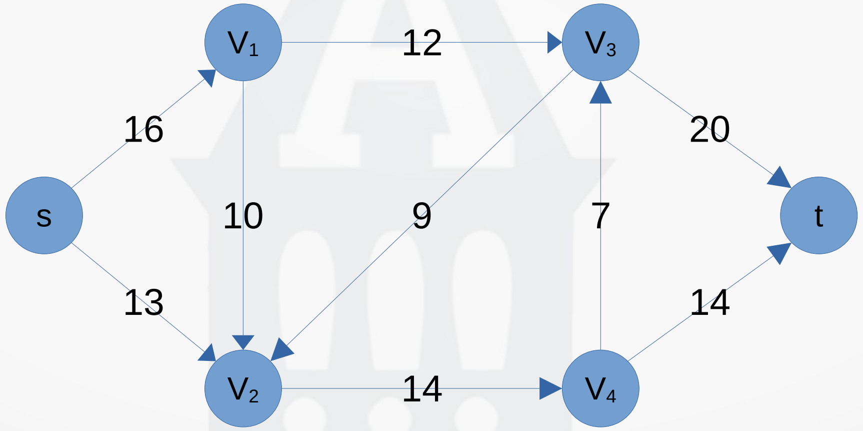

$$\begin{tikzcd}
 & A \arrow[rr, "8/8"] \arrow[drr, "2/5"] & & B \arrow[r, "5/7"] \arrow[dr, "3/5"] & C \arrow[dr, "5/5"] & \\
s \arrow[ur, "10/10"] \arrow[dr, "5/5"'] & & & & & t \\
 & D \arrow[rr, "5/7"] & & E \arrow[r, "7/8"] & F \arrow[ur, "10/10"'] &
\end{tikzcd}$$

But how do we know to do that...efficiently, rather than exhaustively searching?

Network Flow

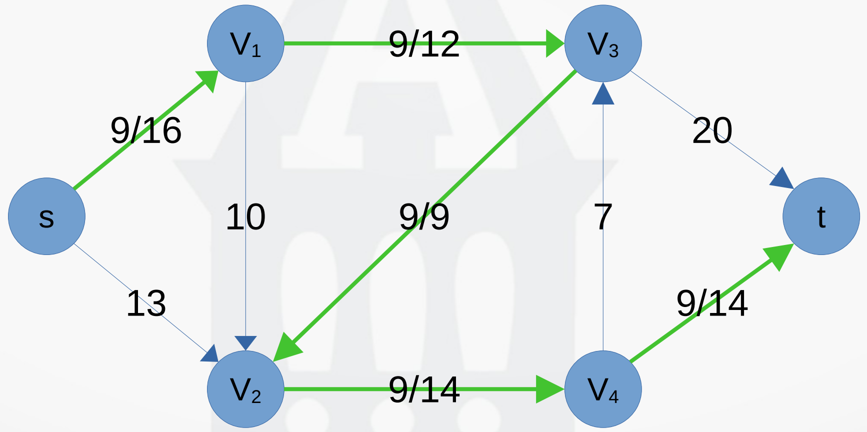
- **Conservation Rule**
  - In order for the assignment of flows to be valid, we must have the sum of flow coming into a vertex equal to the flow coming out of a vertex, for each vertex in the graph except the source and the sink
- **Capacity Rule**
  - Each flow must be less than or equal to the capacity of the edge
- The ***flow of the network*** is defined as the flow from the source to the sink

Network Flow – Residual Graph



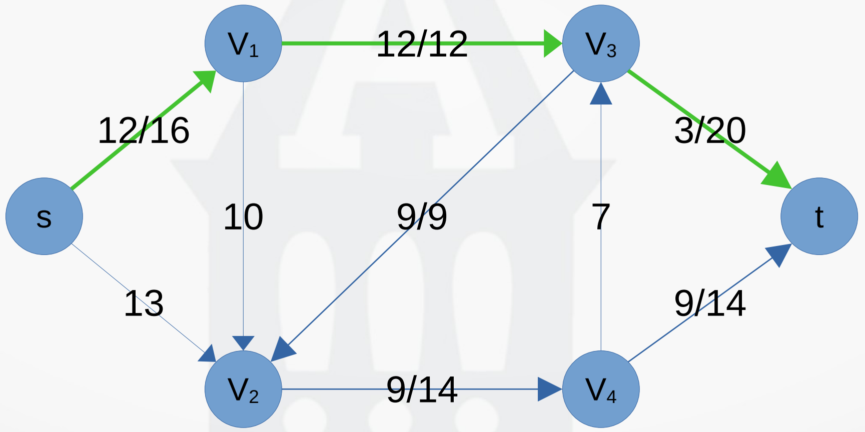
Network Flow – Residual Graph

- Let's start with the green path...yes, I know, it is a bad path



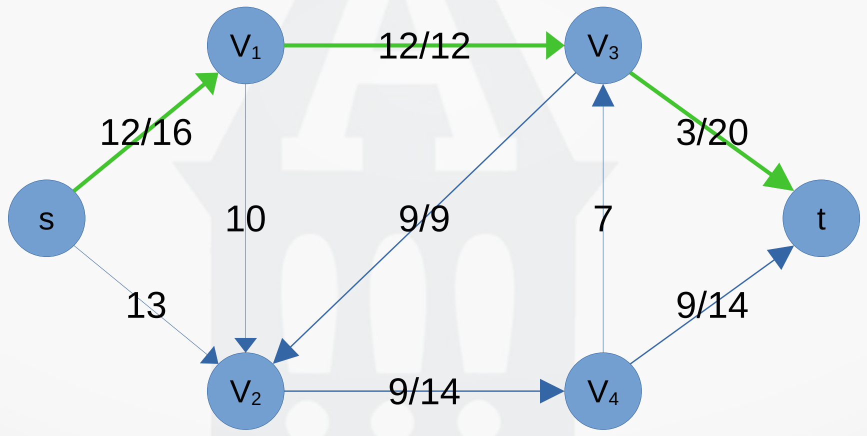
Network Flow – Residual Graph

- Let's look for more flow: 3 more along this path



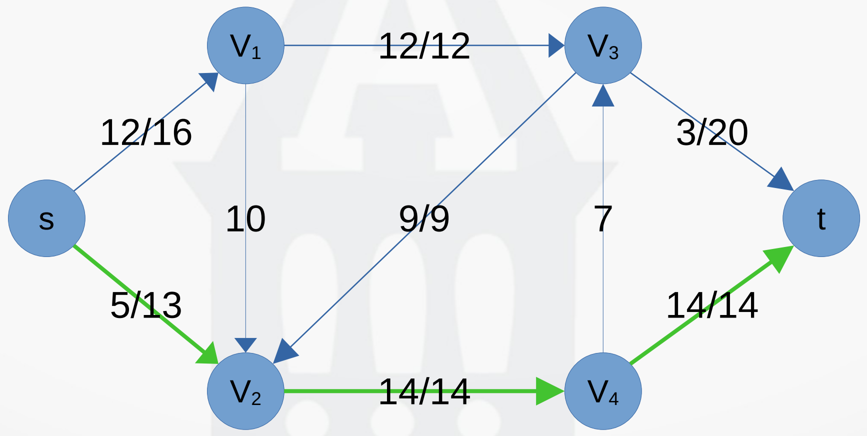
Network Flow – Residual Graph

- Let's look for more flow: 3 more along this path
- Can we do any more?



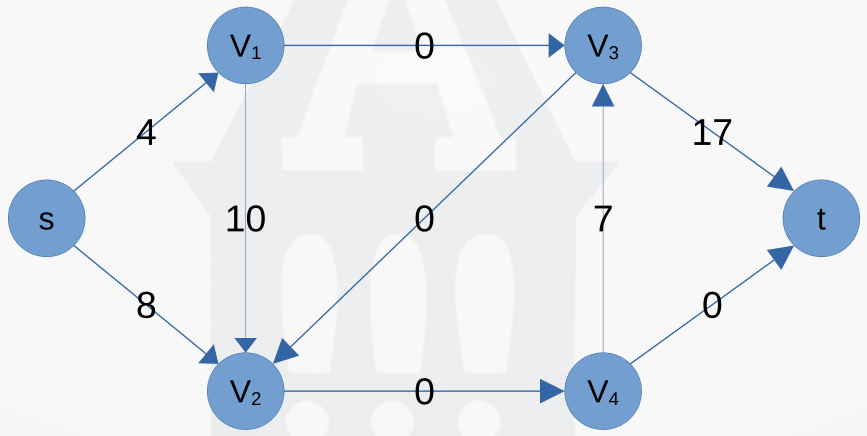
Network Flow – Residual Graph

- Let’s look for more flow: 3 more along this path
- Can we do any more?



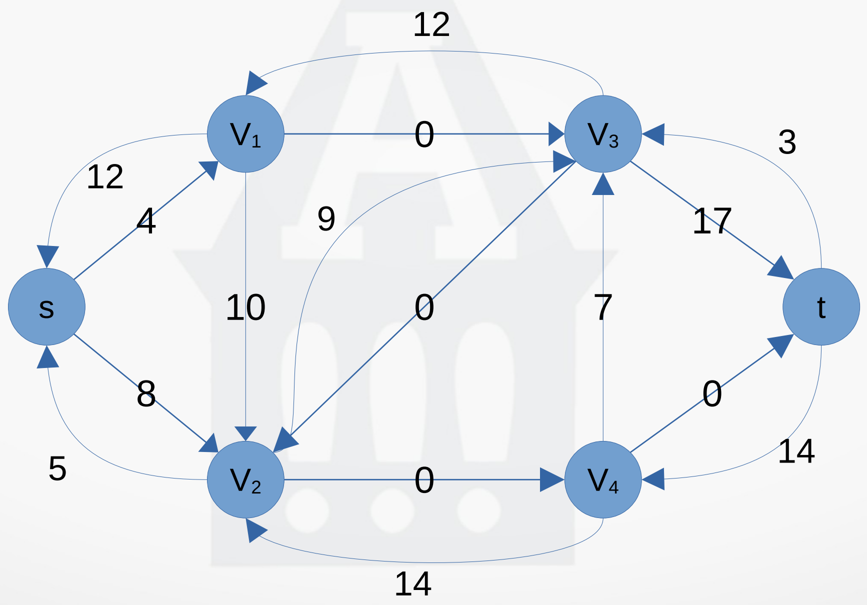
Network Flow – Residual Graph

- Let’s start by creating a ***residual graph*** of ***forward*** flow



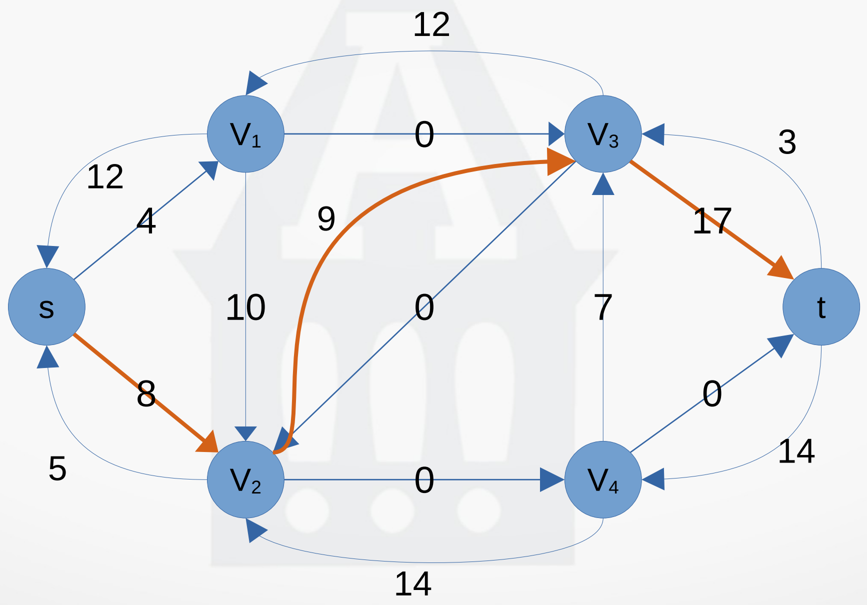
Network Flow – Residual Graph

- Now create a ***residual graph*** of ***forward and backward*** flow
  - backward flow is flow we can “push” backwards (not really, but conceptually)



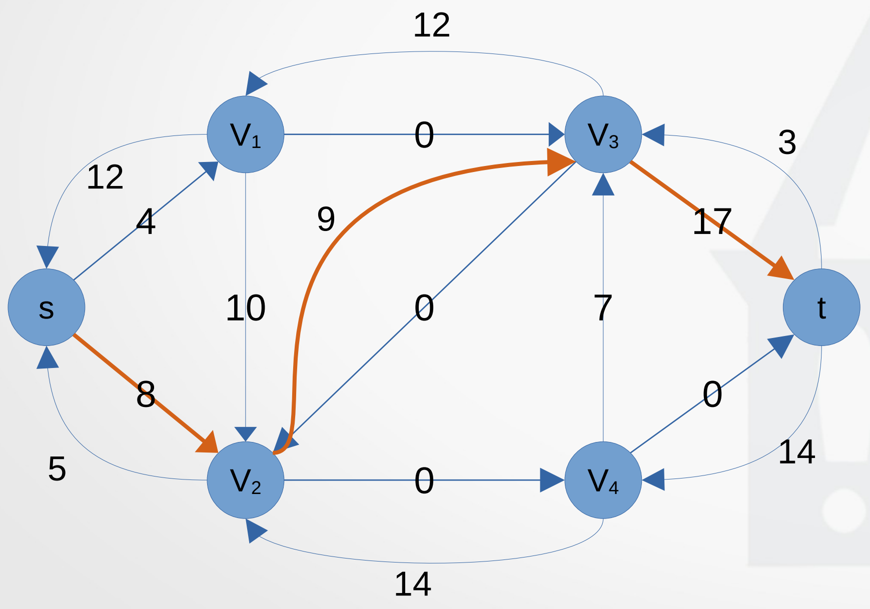
Network Flow – Augmenting Path

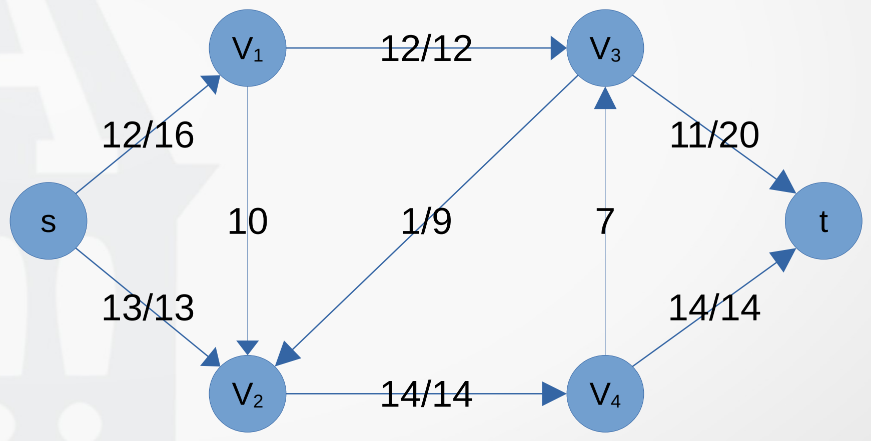
- I could send 8 to V_2, then push 8 to V_3, then send 8 to t



Network Flow – Residual Graph

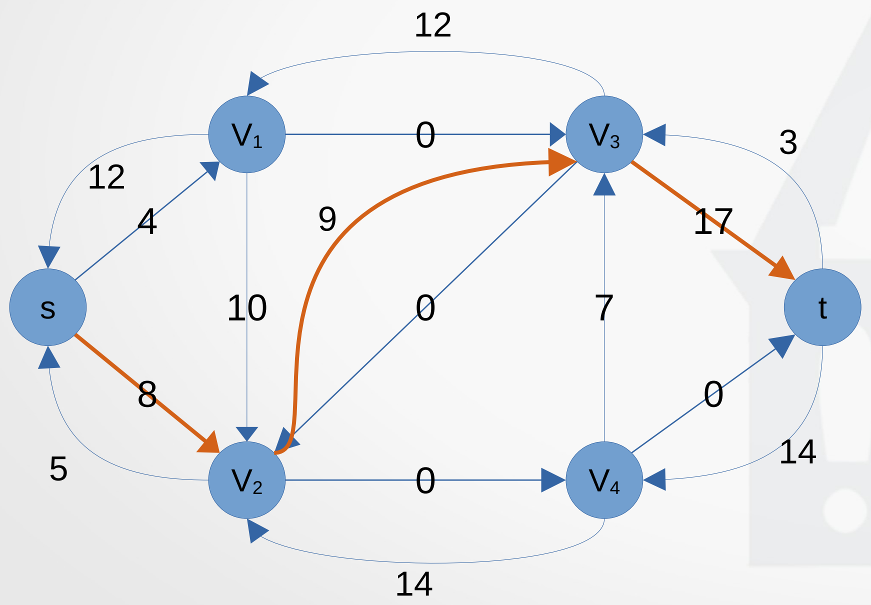
- Then, update the graph, from the newly found ***augmenting path***

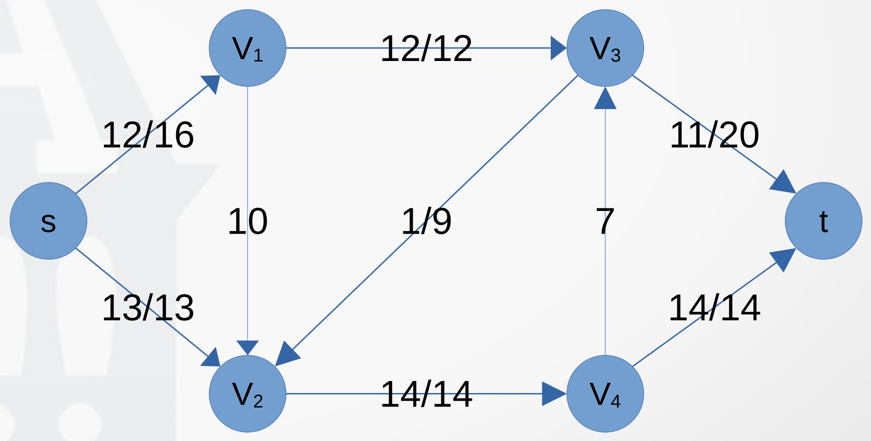




Network Flow – Residual Graph

- This approach allows us to “undo” a bad decision, without starting over and still be guaranteed progress
- This isn’t a way to design a plumbing system, ***flow*** is just a descriptive word





Ford-Fulkerson – Algorithm Notes

- While there exists an augmenting path
  - Add the appropriate flow from that augmenting path

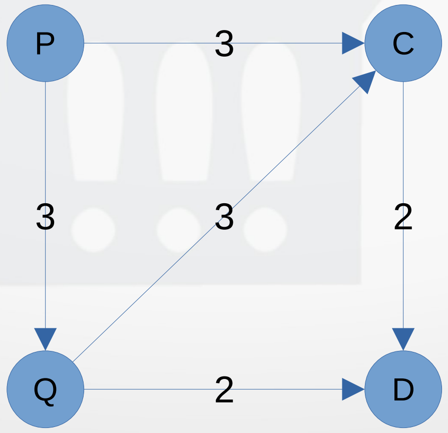
- Can check the existence of an augmenting path by doing a graph traversal on the network; with all fully utilized edges removed
  - This graph, a subgraph, is called a ***residual graph***

- It is difficult to analyze the true running time of this algorithm because it is unclear exactly how many augmenting paths can be found in an arbitrary flow network
  - In the worst case, each augmenting path adds 1 to the network flow, and each search for an augmenting path takes O(E) time; where E is the number of edges in the graph
  - Therefore, the worst-case algorithm takes O(|f|E) time; where |f| is the maximal flow of the network

- Nice tool: https://www-m9.ma.tum.de/graph-algorithms/flow-ford-fulkerson/index_en.html

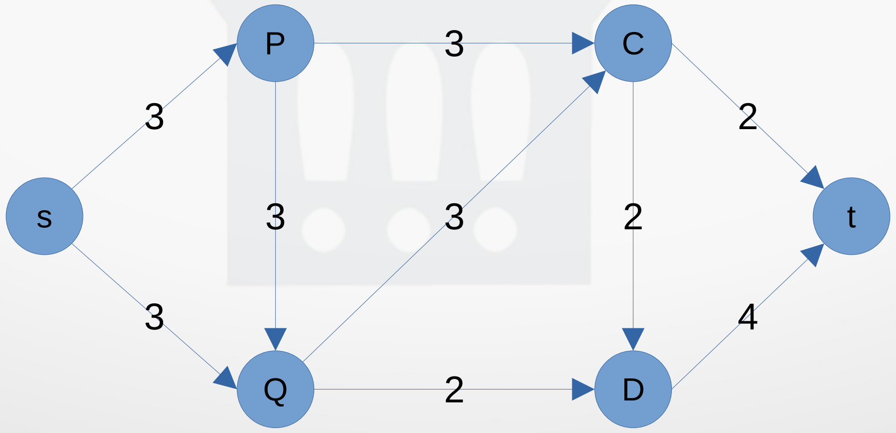
Applications of Network Flow – Producer/Consumer

- There are some factories that produce goods and some villages where the goods have to be delivered
  - They are connected by a network of roads, each road has capacity c
- The problem is to find if there is a way of transferring the goods that satisfies the demand
- In this example
  - factories P and Q can each produce 3
  - Village D wants 4 and Village C wants 2
- The road capacities don't consider production ability of plants or needs of cities
- *Need to turn this into a maximum flow problem*



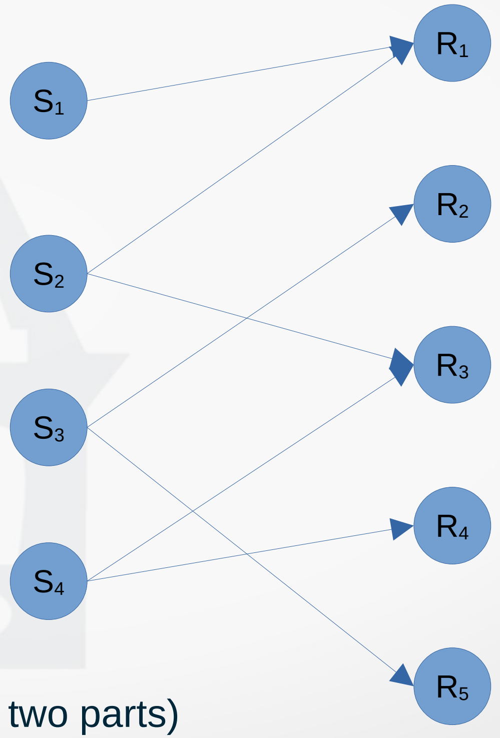
Applications of Network Flow – Producer/Consumer

- There are some factories that produce goods and some villages where the goods have to be delivered
  - They are connected by a network of roads, each road has capacity c
- The problem is to find if there is a way of transferring the goods that satisfies the demand
- In this example
  - factories P and Q can each produce 3
  - Village D wants 4 and Village C wants 2
- The road capacities don't consider production ability of plants or needs of cities



Applications of Network Flow – Matching Pairs

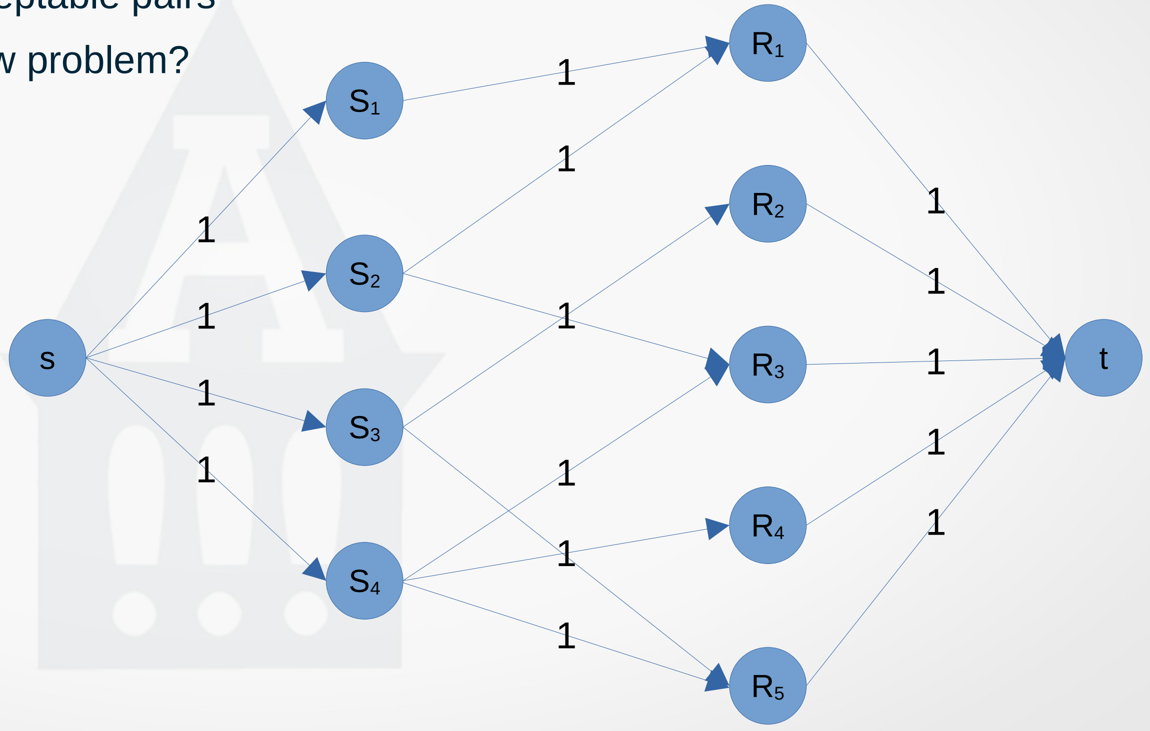
- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?


$$\begin{tikzcd}
 & R_1 \\
S_1 \arrow[ur] & \\
 & R_2 \\
S_2 \arrow[uuur] \arrow[dr] & \\
 & R_3 \\
S_3 \arrow[uuur] \arrow[dddr] & \\
 & R_4 \\
S_4 \arrow[uuur] \arrow[ur] & \\
 & R_5
\end{tikzcd}$$

Side note: This is called a bipartite graph (consists of two parts)

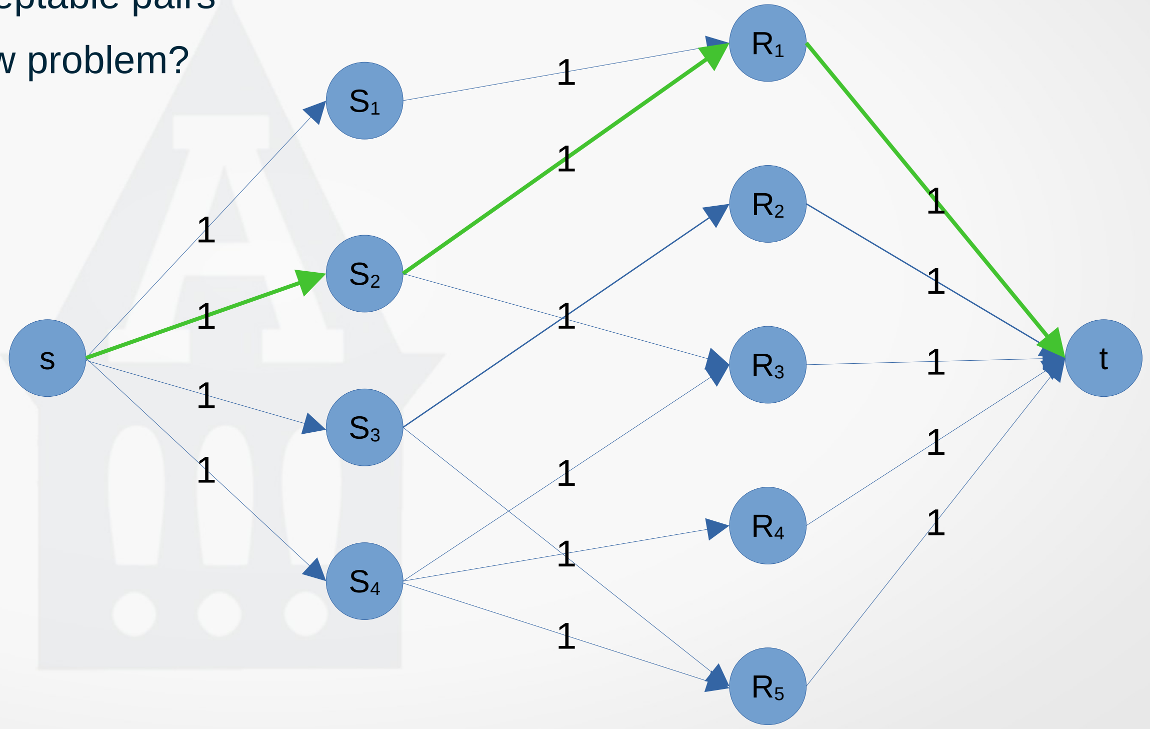
Applications of Network Flow – Matching Pairs

- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?



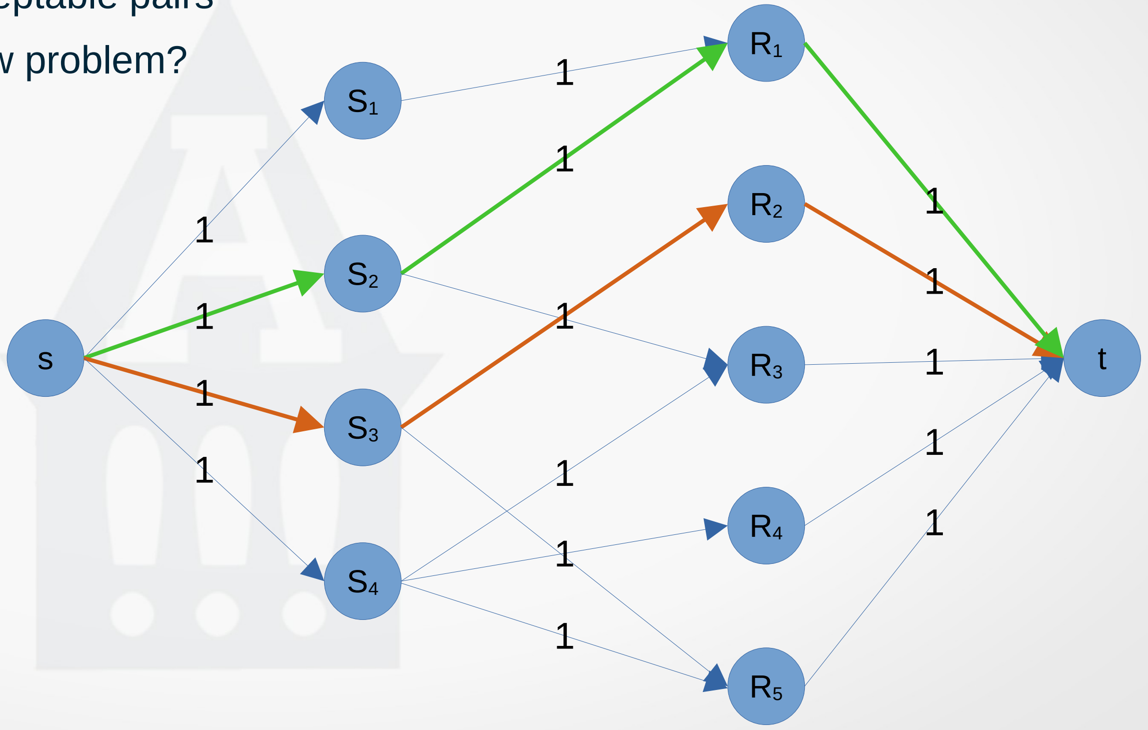
Applications of Network Flow – Matching Pairs

- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?



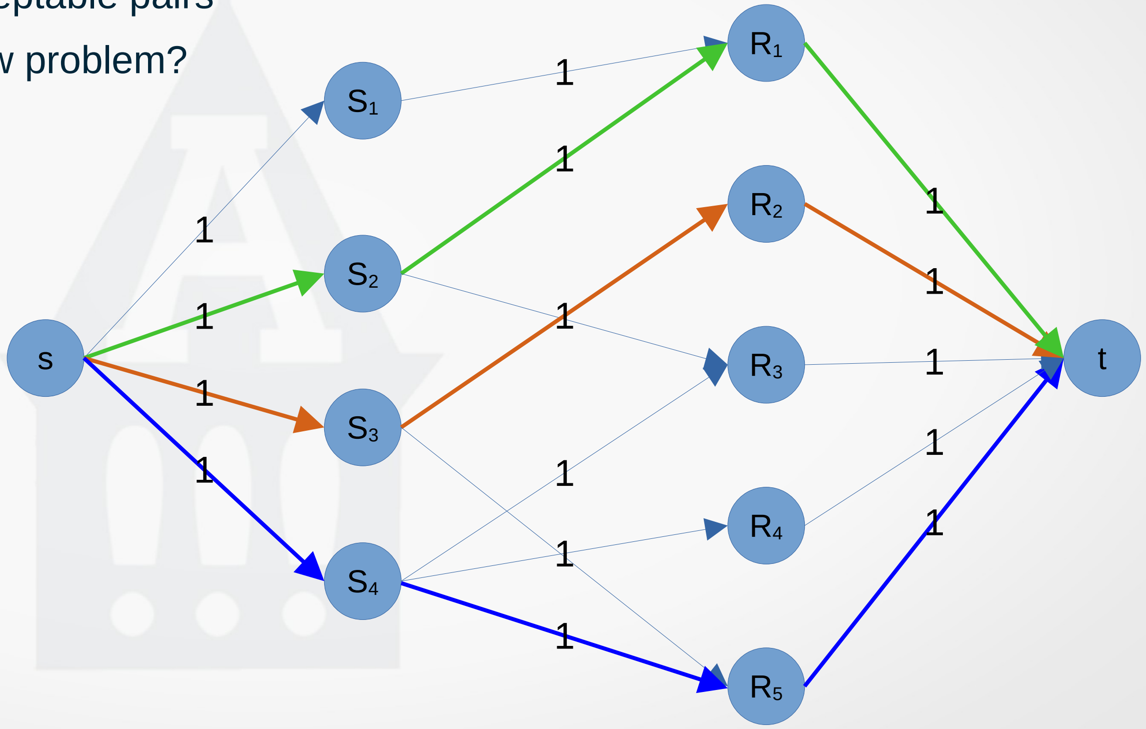
Applications of Network Flow – Matching Pairs

- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?



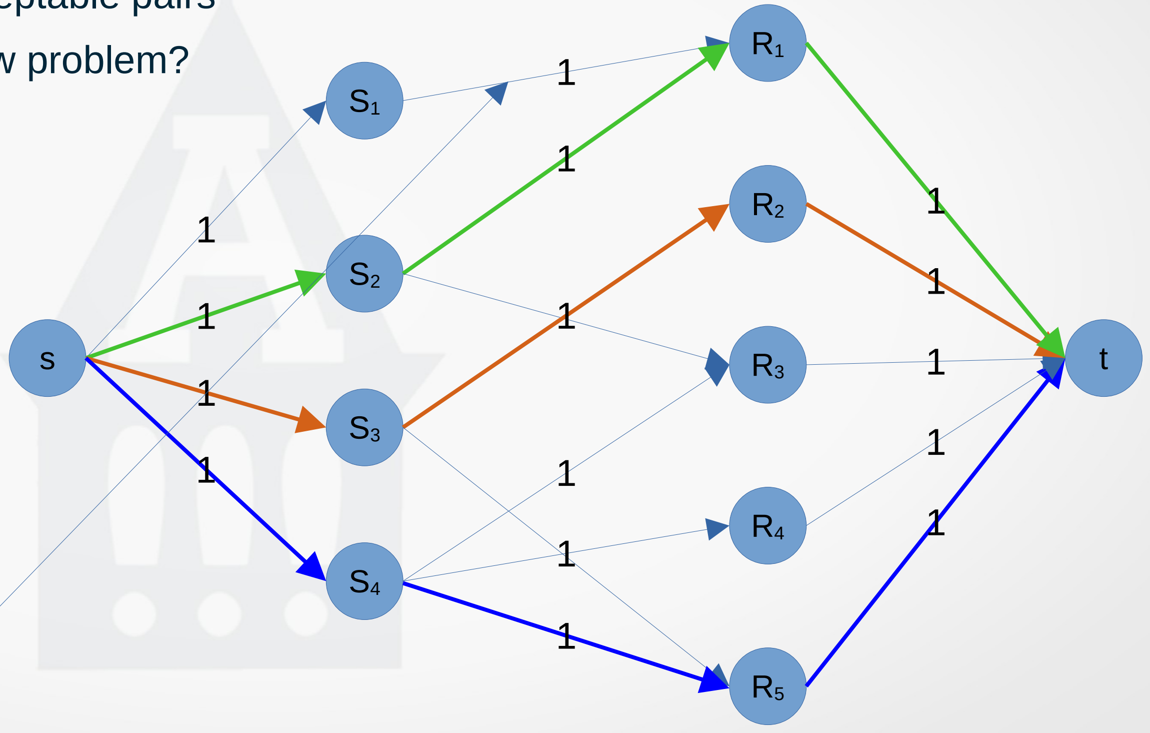
Applications of Network Flow – Matching Pairs

- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?



Applications of Network Flow – Matching Pairs

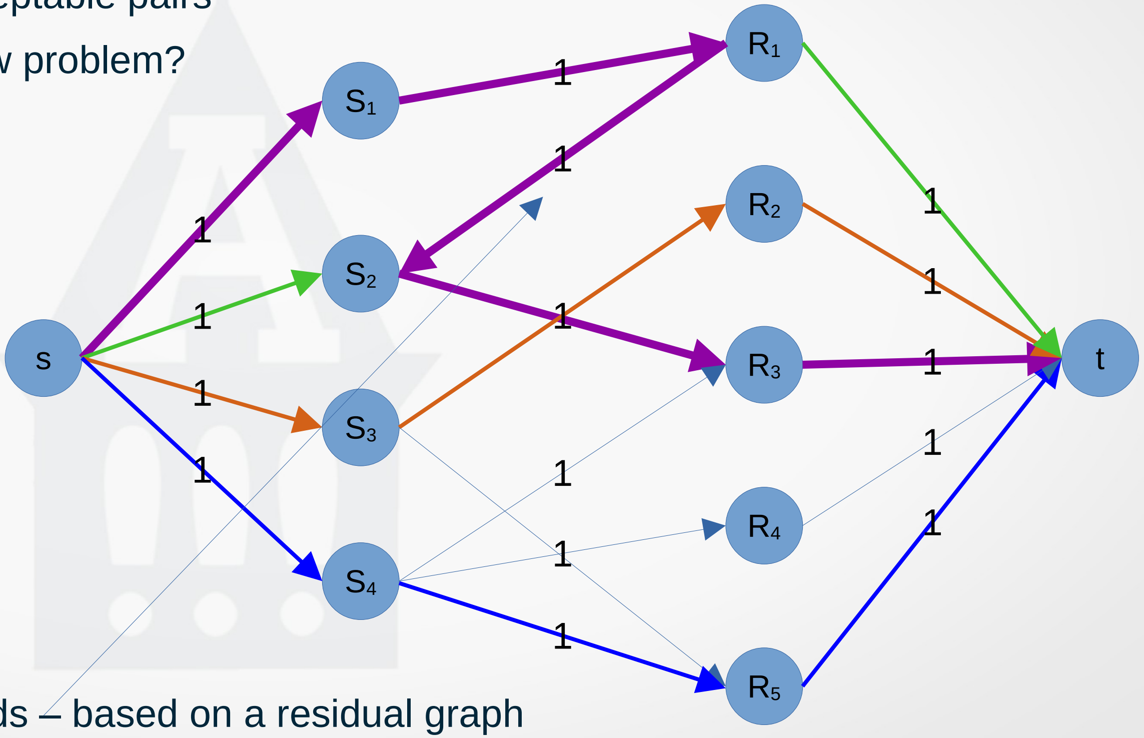
- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?



But now have a problem here

Applications of Network Flow – Matching Pairs

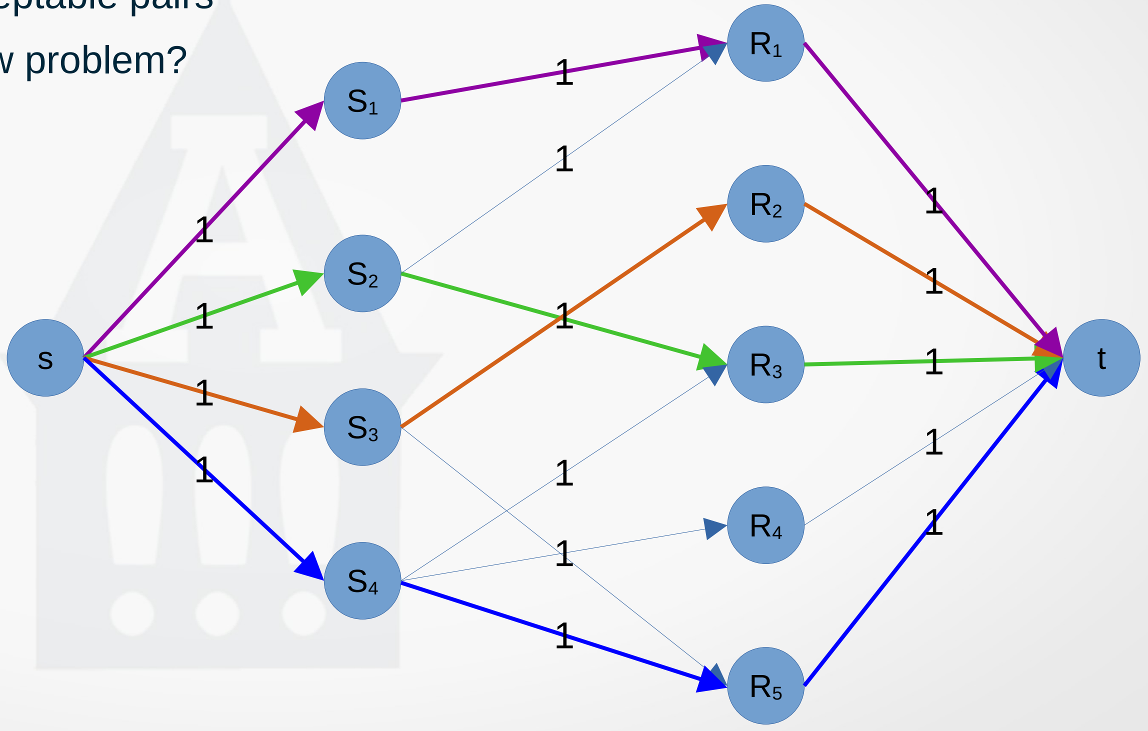
- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?



To solve, let's "push" flow backwards – based on a residual graph

Applications of Network Flow – Matching Pairs

- Students (S) specify room (R) preference: arrows represent those preferences
- Goal: Maximize the number of acceptable pairs
- Can we turn this into a network flow problem?



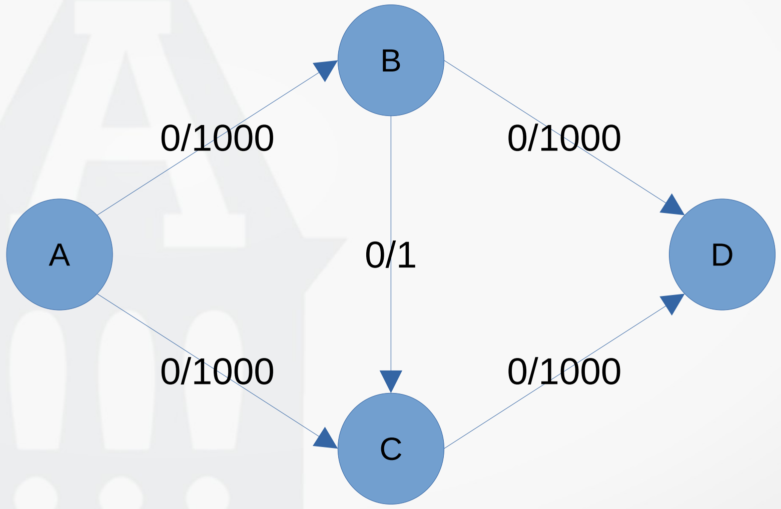
Resolve the graph

Edmonds-Karp Algorithm

- A variation on the Ford-Fulkerson algorithm, which is intended to increase the speed; can also think of it as trying to avoid making “bad” choices.
- Concept is to try to choose good augmenting paths
  - Augmenting path with the minimal number of edges
  - Augmenting path with the maximal flow

Edmonds-Karp Algorithm – Motivation

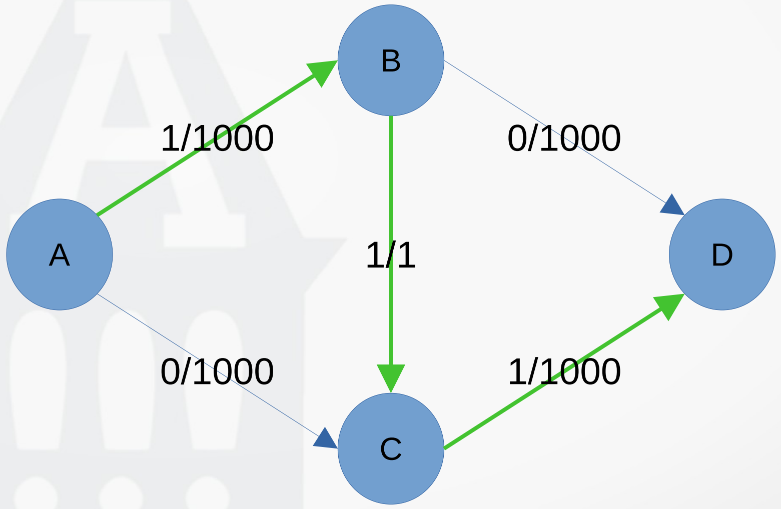
- Consider this network flow graph
- What paths can flow take from A to D?



$$\begin{tikzcd}
 & B \arrow[dr, "0/1000"] \arrow[dd, "0/1"] & \\
A \arrow[ur, "0/1000"] \arrow[dr, "0/1000"'] & & D \\
 & C \arrow[ur, "0/1000"'] &
\end{tikzcd}$$

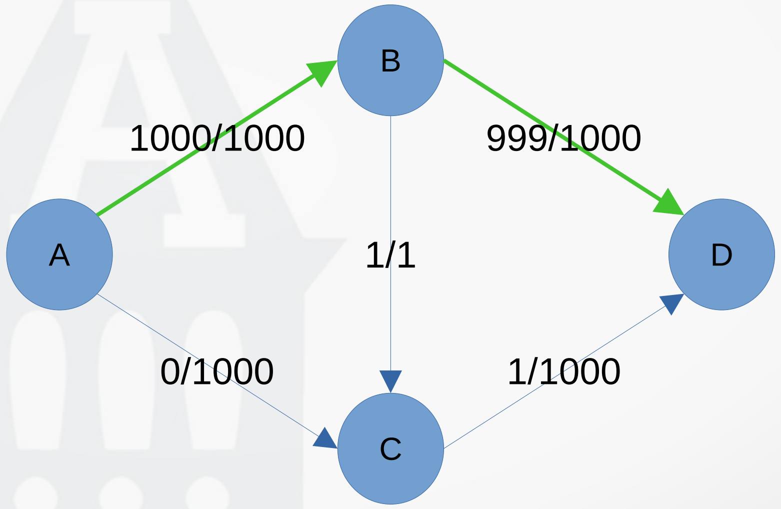
Edmonds-Karp Algorithm – Motivation

- Consider this network flow graph
- What paths can flow take from A to D?
  - A → B → C → D (1)



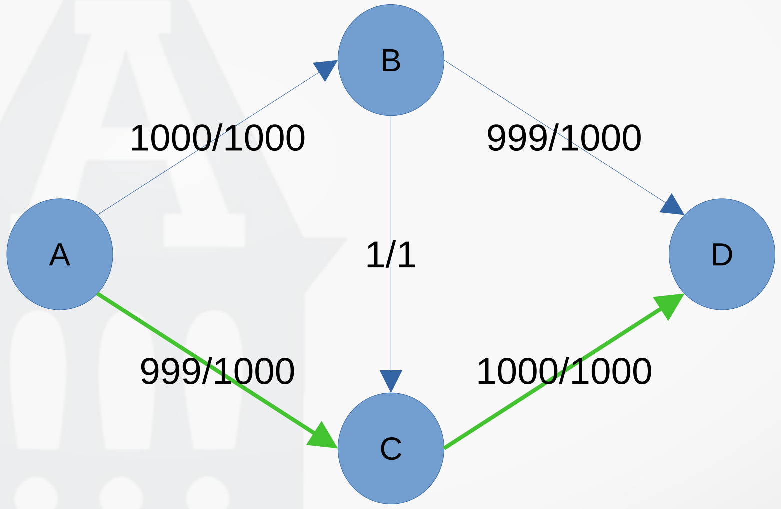
Edmonds-Karp Algorithm – Motivation

- Consider this network flow graph
- What paths can flow take from A to D?
  - A → B → C → D (1)
  - A → B → D (999)



Edmonds-Karp Algorithm – Motivation

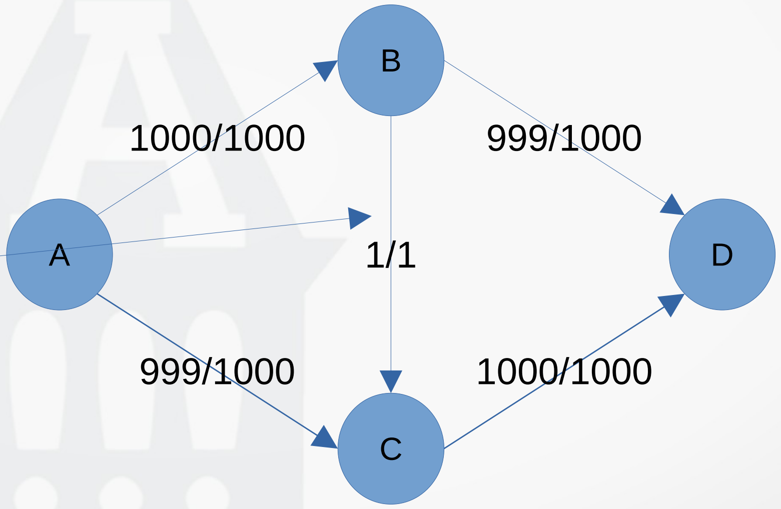
- Consider this network flow graph
- What paths can flow take from A to D?
  - A → B → C → D (1)
  - A → B → D (999)
  - A → C → D (999)



Edmonds-Karp Algorithm – Motivation

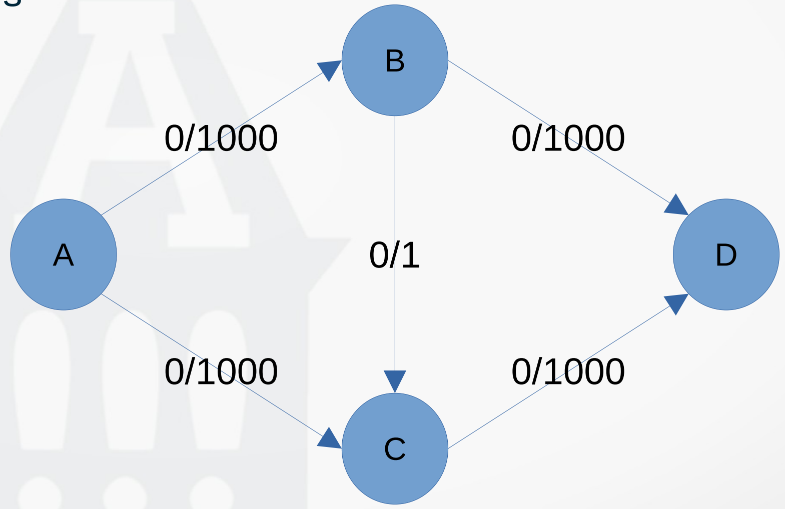
- Consider this network flow graph
- What paths can flow take from A to D?
  - A → B → C → D (1)
  - A → B → D (999)
  - A → C → D (999)

- If we do that, then we have to push and pull to fix the “bad” choice



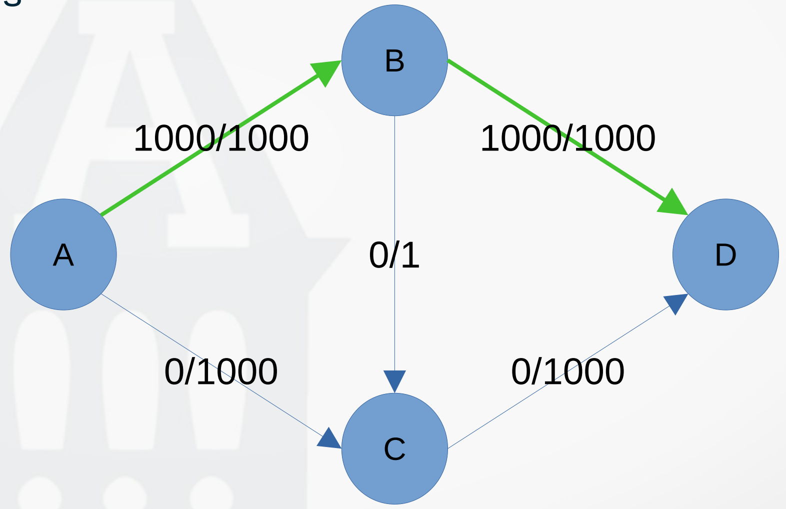
Edmonds-Karp Algorithm

- Consider this network flow graph
- What paths can flow take from A to D?
- Let’s start by choosing the shortest path’s



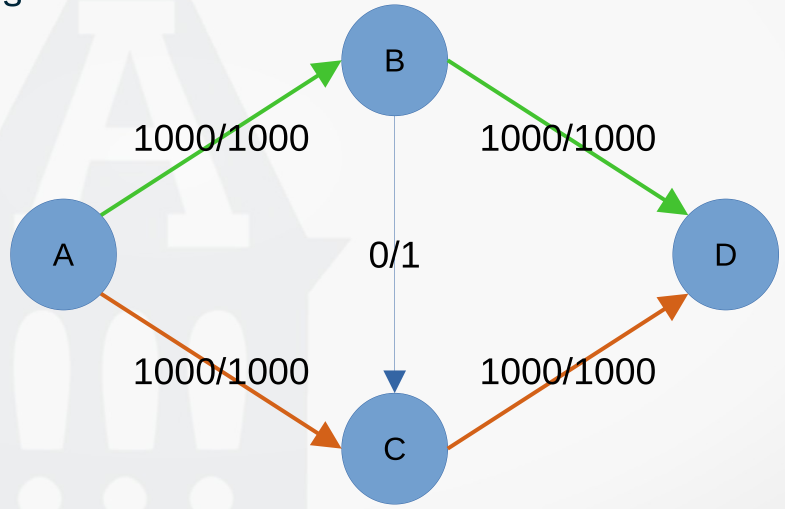
Edmonds-Karp Algorithm

- Consider this network flow graph
- What paths can flow take from A to D?
- Let’s start by choosing the shortest path’s



Edmonds-Karp Algorithm

- Consider this network flow graph
- What paths can flow take from A to D?
- Let’s start by choosing the shortest path’s

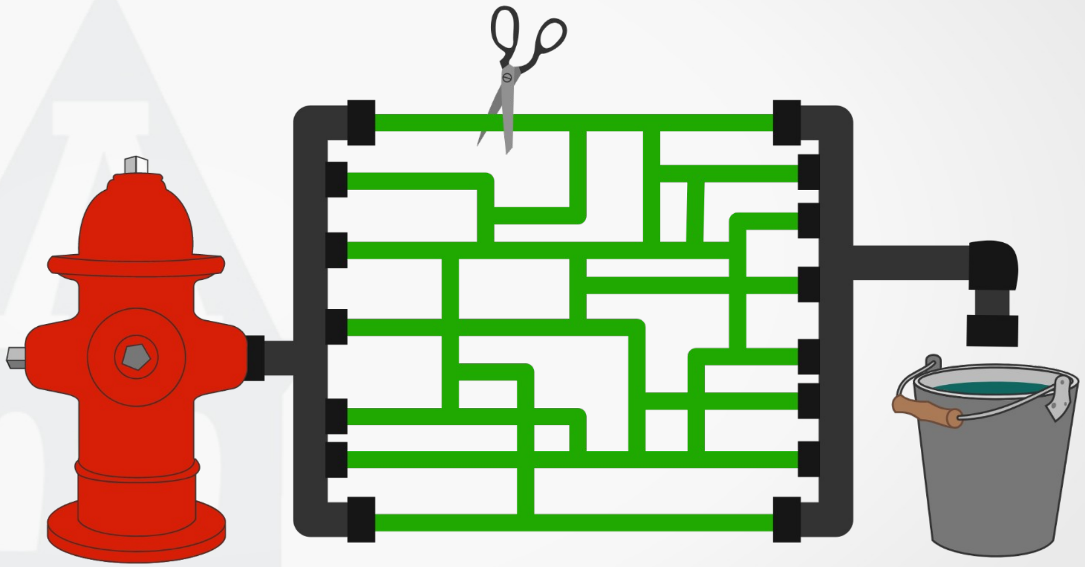


Max-Flow, Min-Cut Theorem

- This theorem states that the maximum flow through any network from a given source to a given sink is exactly the sum of the ***edge weights*** that, ***if removed***, would totally disconnect the source from the sink

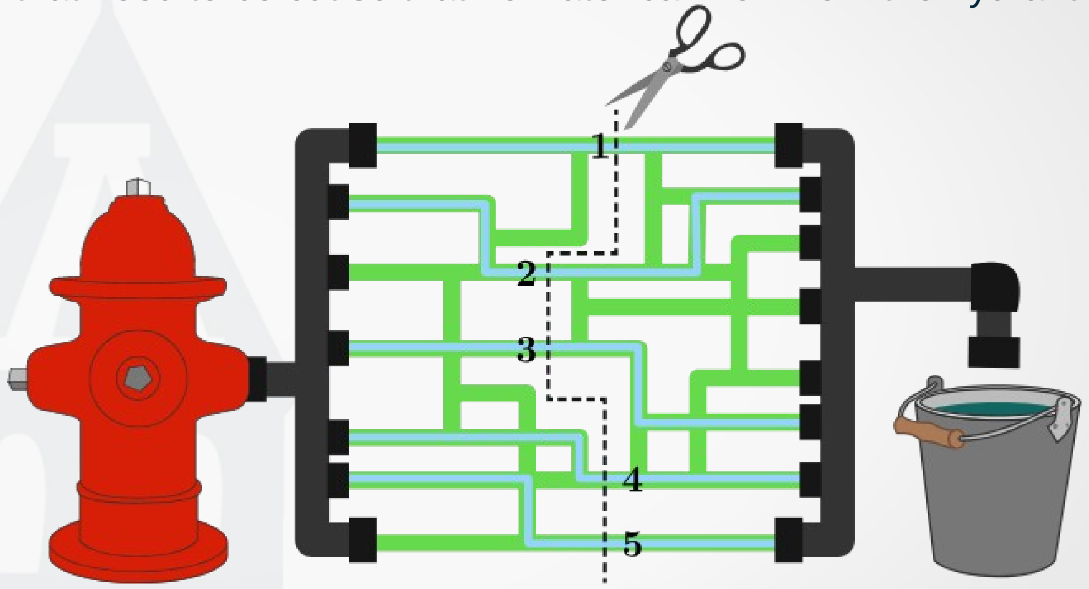
Max-Flow, Min-Cut Motivation

- What is the fewest number of green tubes that need to be cut so that no water can flow from the hydrant to the bucket?
- Find a tube segment that water is flowing at full capacity. Somewhere along the path that each stream of water takes, there will be at least one such tube (otherwise the system it isn't being used at full capacity).
- Once found, test squeezing it shut. If squeezing it shut reduces the capacity of the system, because water can't find another way to get through it, then cut it.



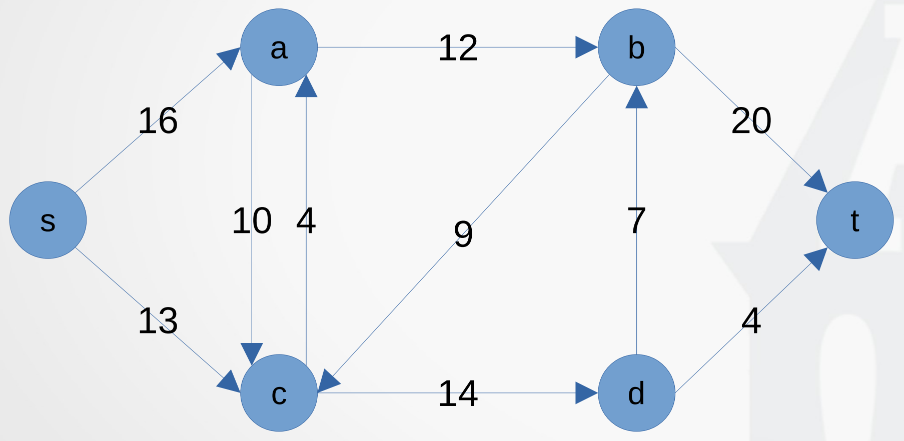
Max-Flow, Min-Cut Motivation

- What is the fewest number of green tubes that need to be cut so that no water can flow from the hydrant to the bucket?
- Find a tube segment that water is flowing at full capacity. Somewhere along the path that each stream of water takes, there will be at least one such tube (otherwise the system it isn't being used at full capacity).
- Once found, test squeezing it shut. If squeezing it shut reduces the capacity of the system, because water can't find another way to get through it, then cut it.

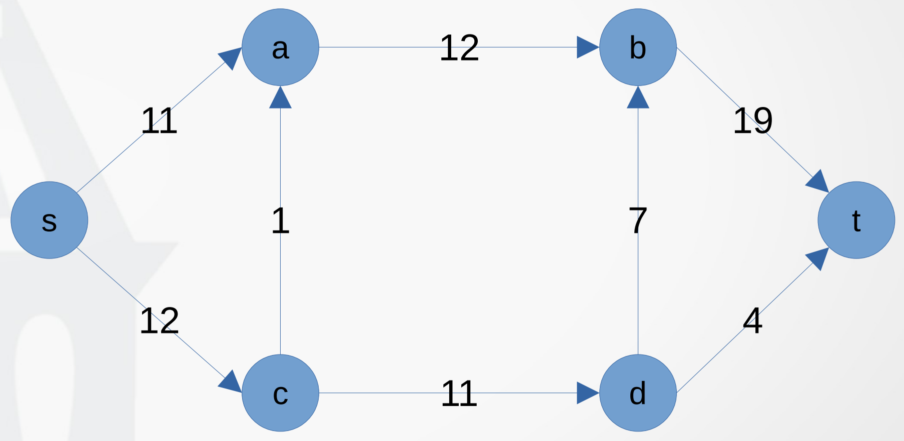


Computing Min-Cut

Max-Flow

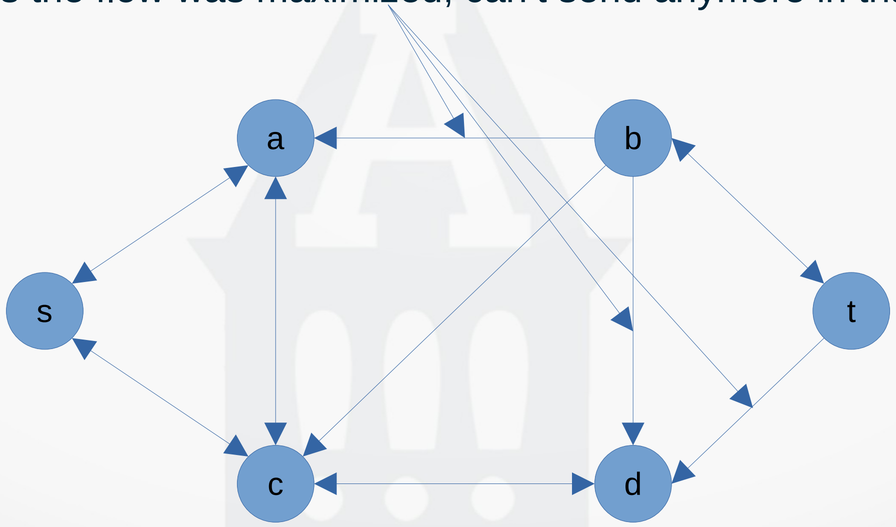


Actual Flow



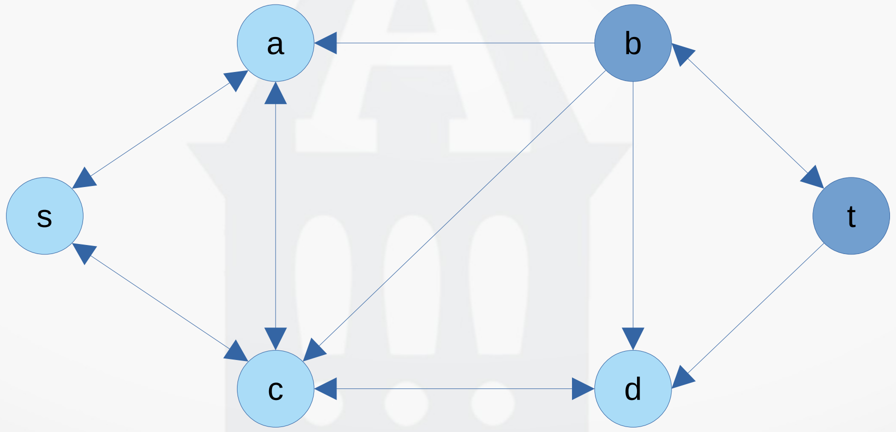
Computing Min-Cut

- Create the ***residual graph*** (forward and backward), but only the topology
  - Notice the direction of the arrows: can “push” flow in some, but not all directions
    - In some cases the flow was maximized, can’t send anymore in that direction



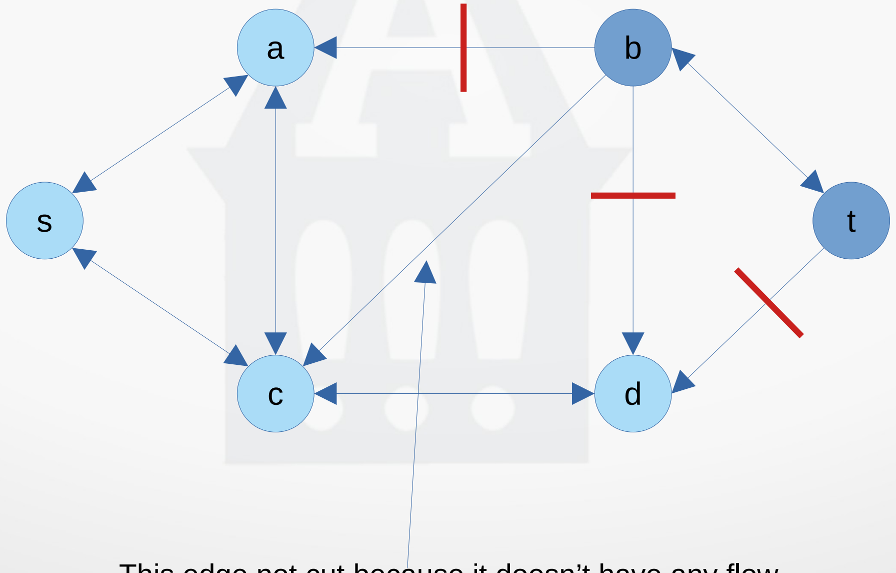
Computing Min-Cut

- Create the ***residual graph*** (forward and backward), but only the topology
- Mark all nodes reachable from s: call this set A



Computing Min-Cut

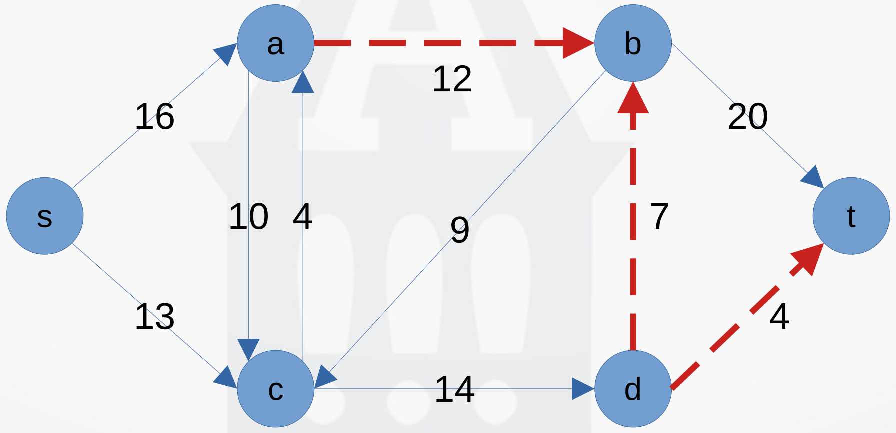
- Create the ***residual graph*** (forward and backward), but only the topology
- Mark all nodes reachable from s: call this set A
- Identify the edges between the reachable and not reachable; those are the ones to cut



This edge not cut because it doesn't have any flow

Computing Min-Cut

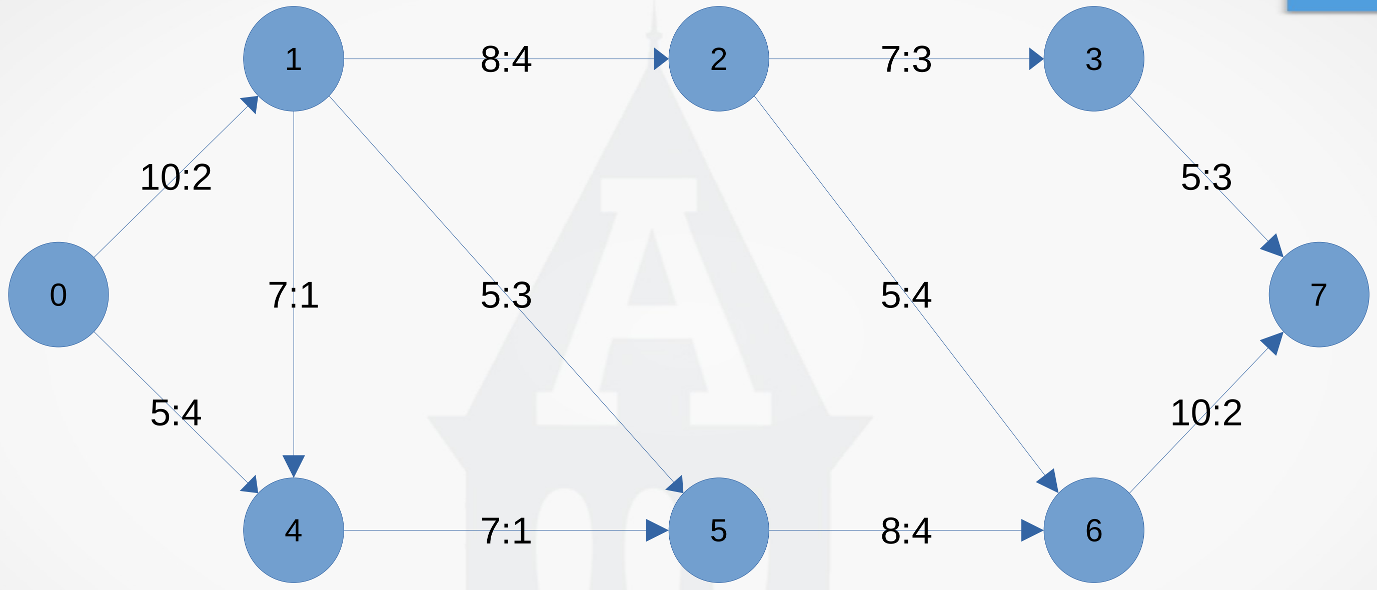
- Create the ***residual graph*** (forward and backward), but only the topology
- Mark all nodes reachable from s: call this set A
- Identify the edges between the reachable and not reachable; those are the ones to cut
- Algorithm identifies the ***min edge weight*** to cut, not actually the min number of edges



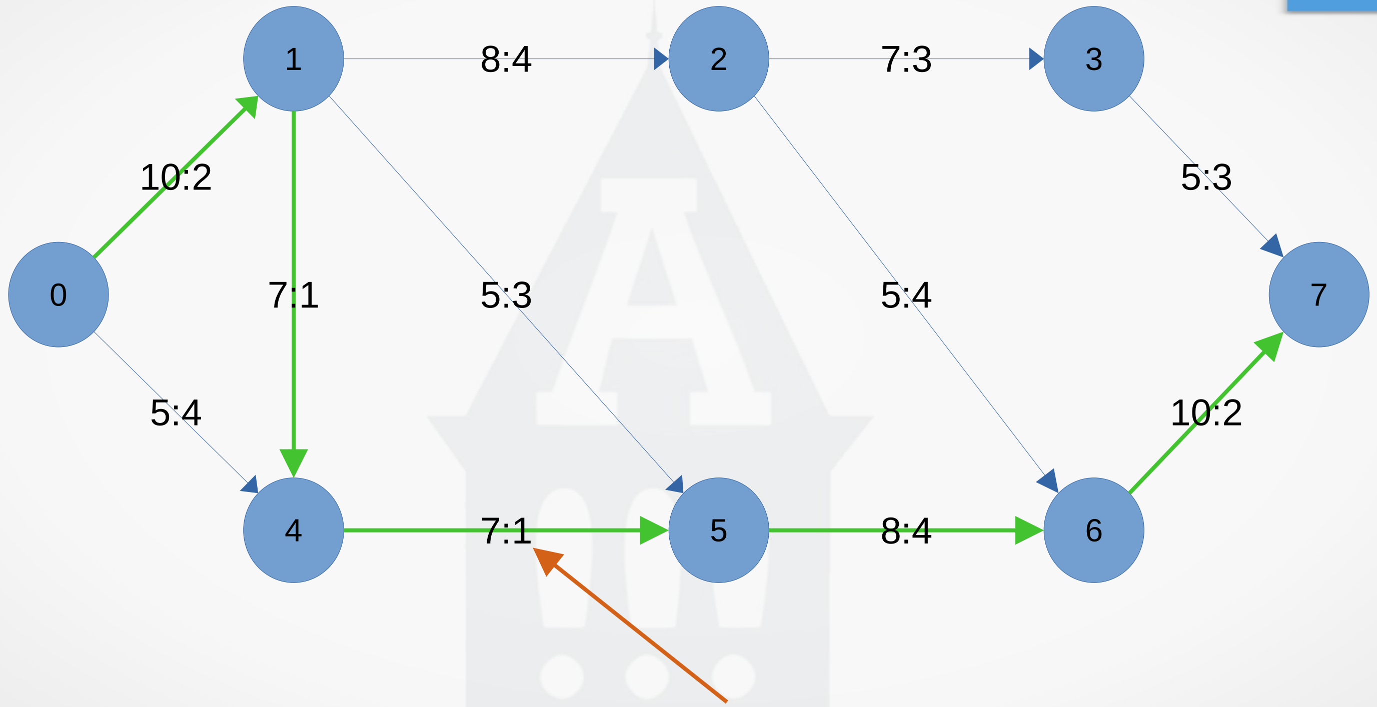
Min-Cost, Max-Flow

- We modify the network as follows:
  - For each edge (i, j), add the reverse edge (j, i) to the network with capacity (U) of 0 and the cost C(i, j) = -C(i, j)
    - This indicates that going backwards along an edge removes the cost of going forward
  - We keep the same condition that flow can be pushed backwards along an edge
- We define the ***residual network*** in the same manner as Ford-Fulkerson
  - The residual network contains only unsaturated edges; flow < capacity
- The algorithm
  - At each iteration, we find the shortest path in the residual graph from source to sink
    - Shortest path is determined by cost, not number of edges
    - Can’t use Dijkstra because cost is a negative edge; use Bellman-Ford & Modified Dijkstra
  - If a path is found, flow is increased as much as possible
  - Repeat until no more flow is possible or a path no longer exists

Min-Cost, Max-Flow

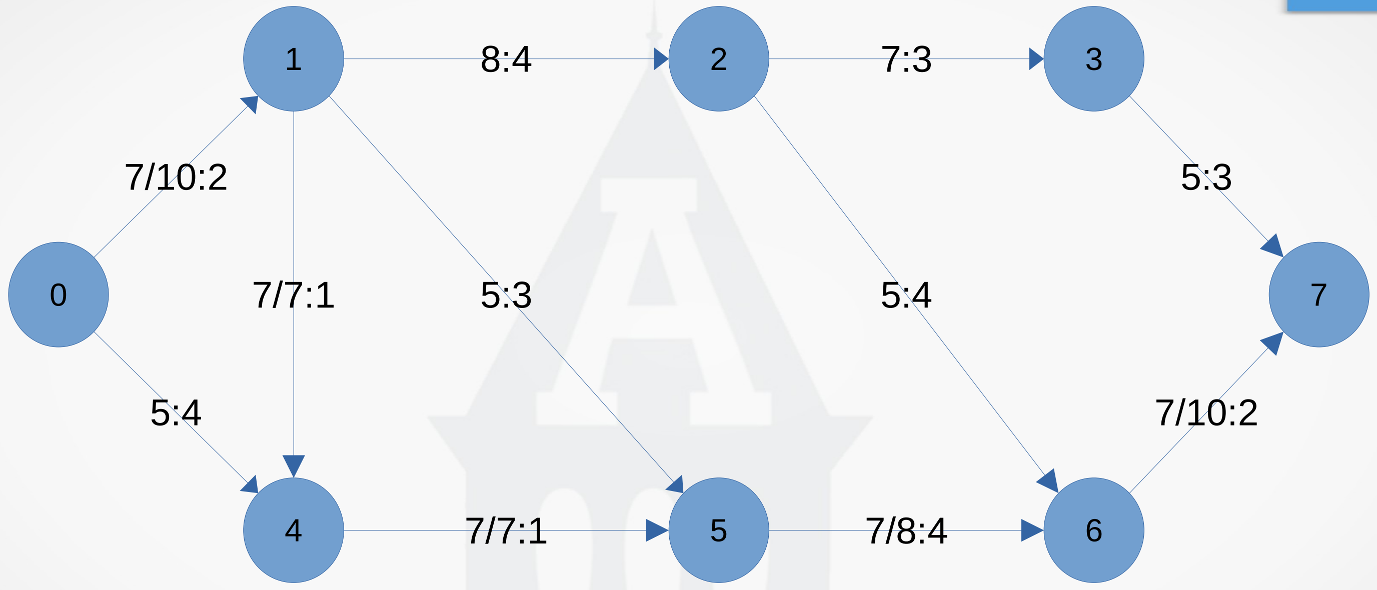


Min-Cost, Max-Flow



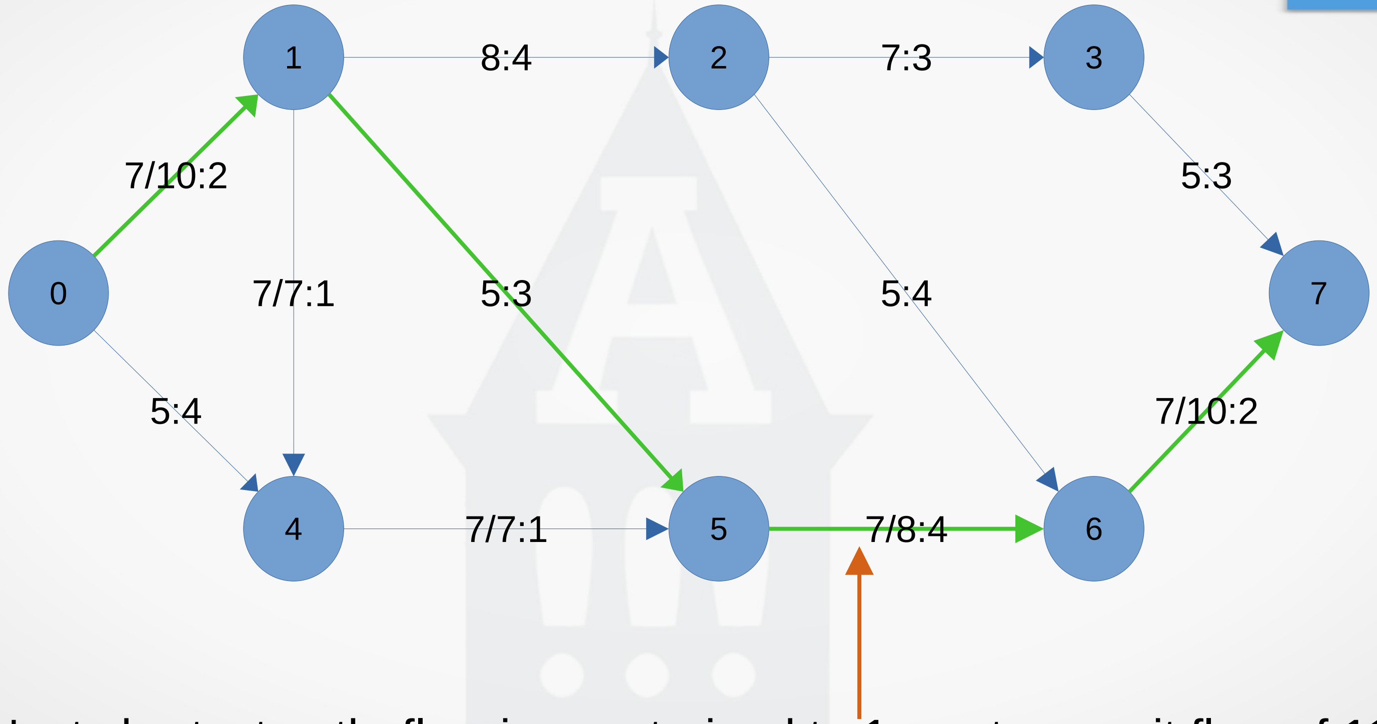
Using shortest path: flow is constrained to 7, cost per unit flow of 10

Min-Cost, Max-Flow



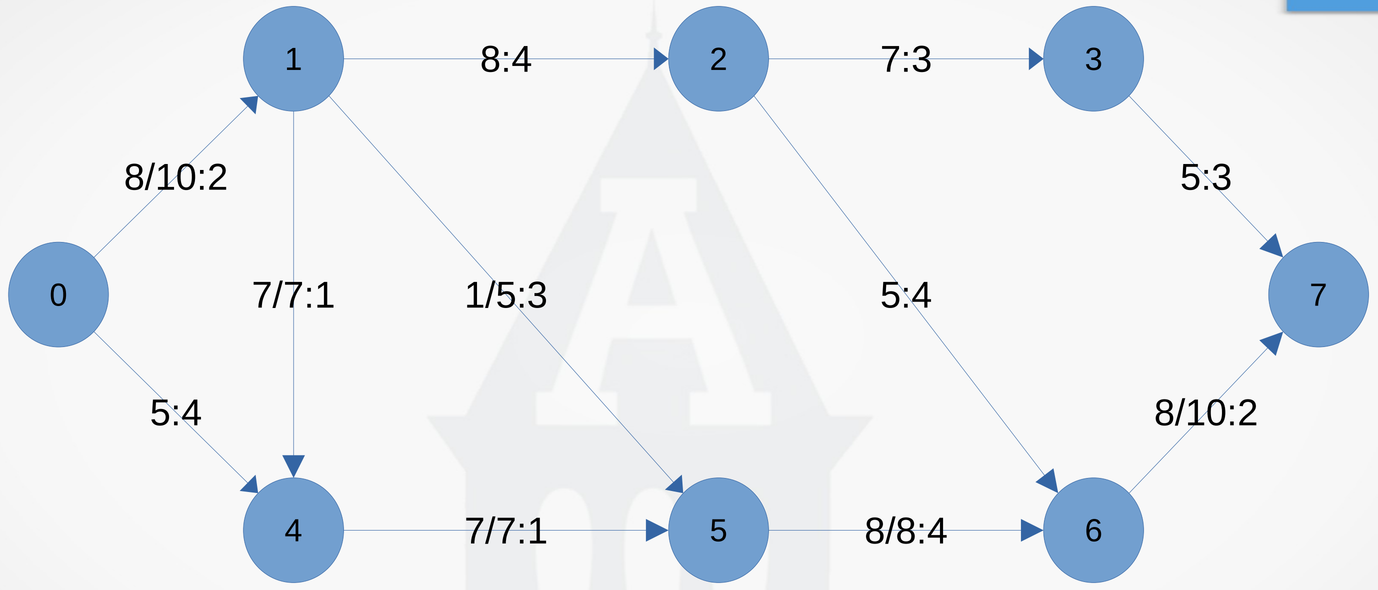
Update with this flow

Min-Cost, Max-Flow



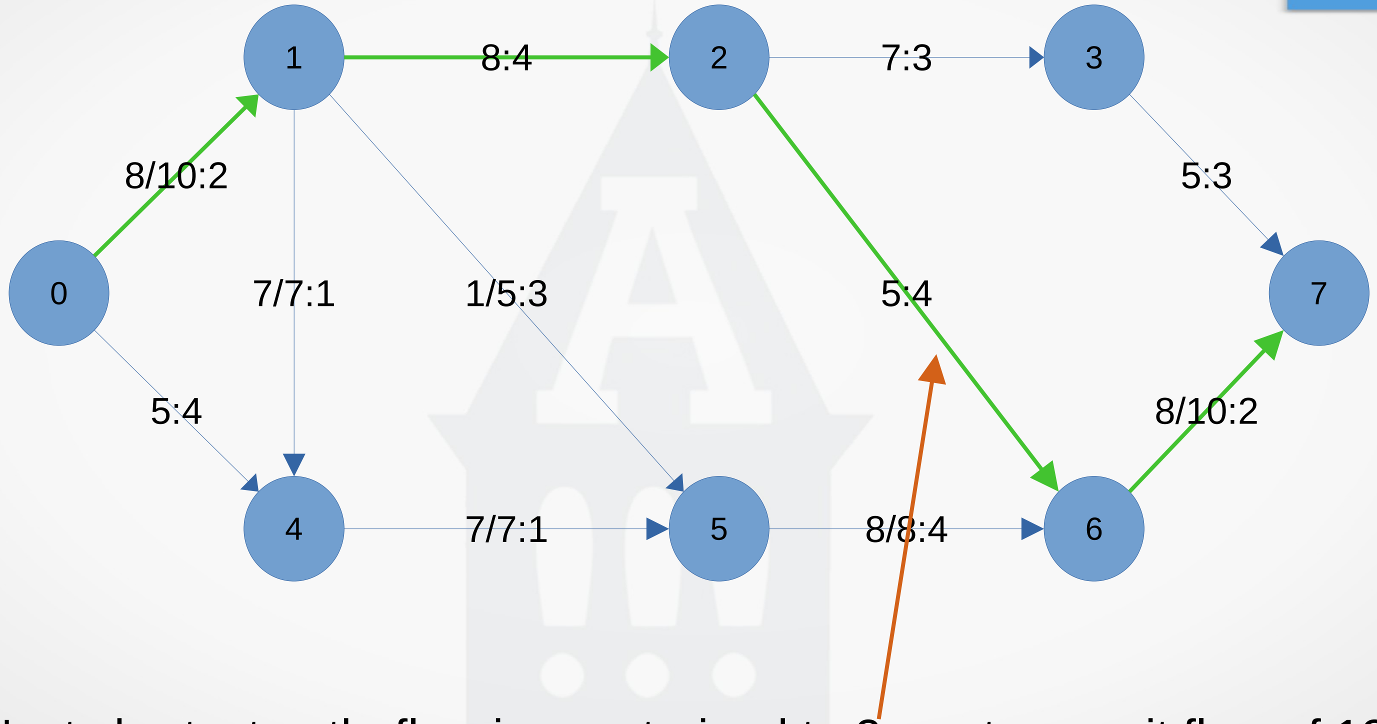
Next shortest path; flow is constrained to 1, cost per unit flow of 11

Min-Cost, Max-Flow



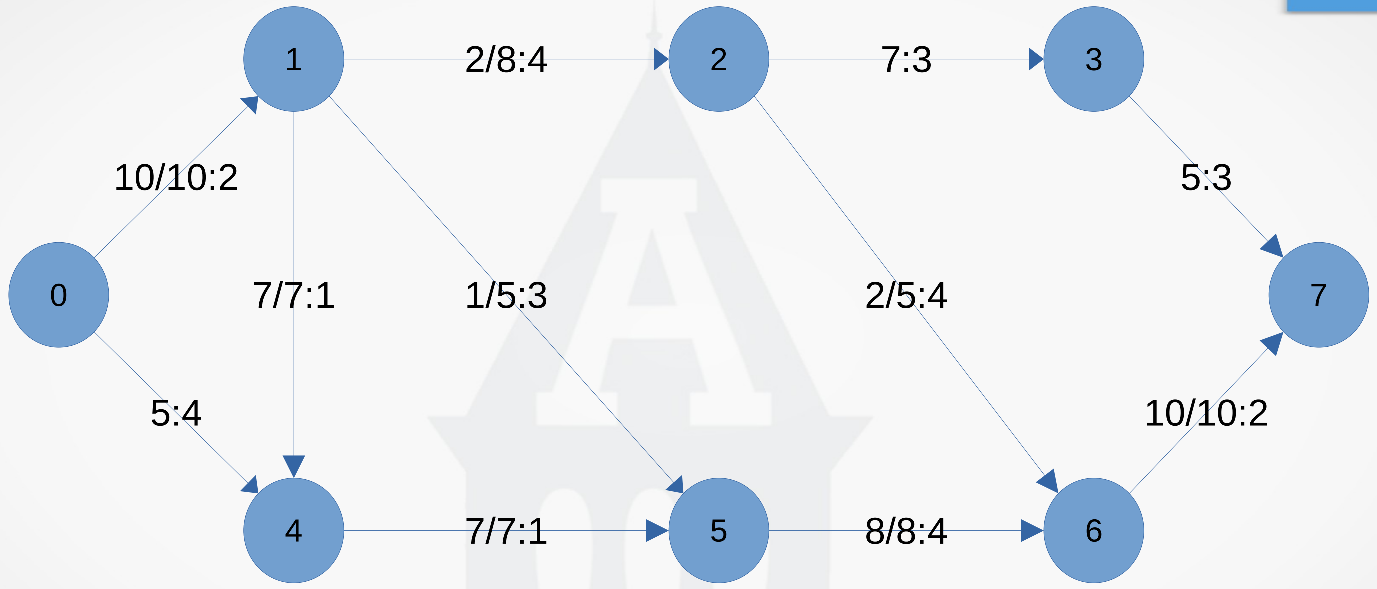
Update with this flow

Min-Cost, Max-Flow



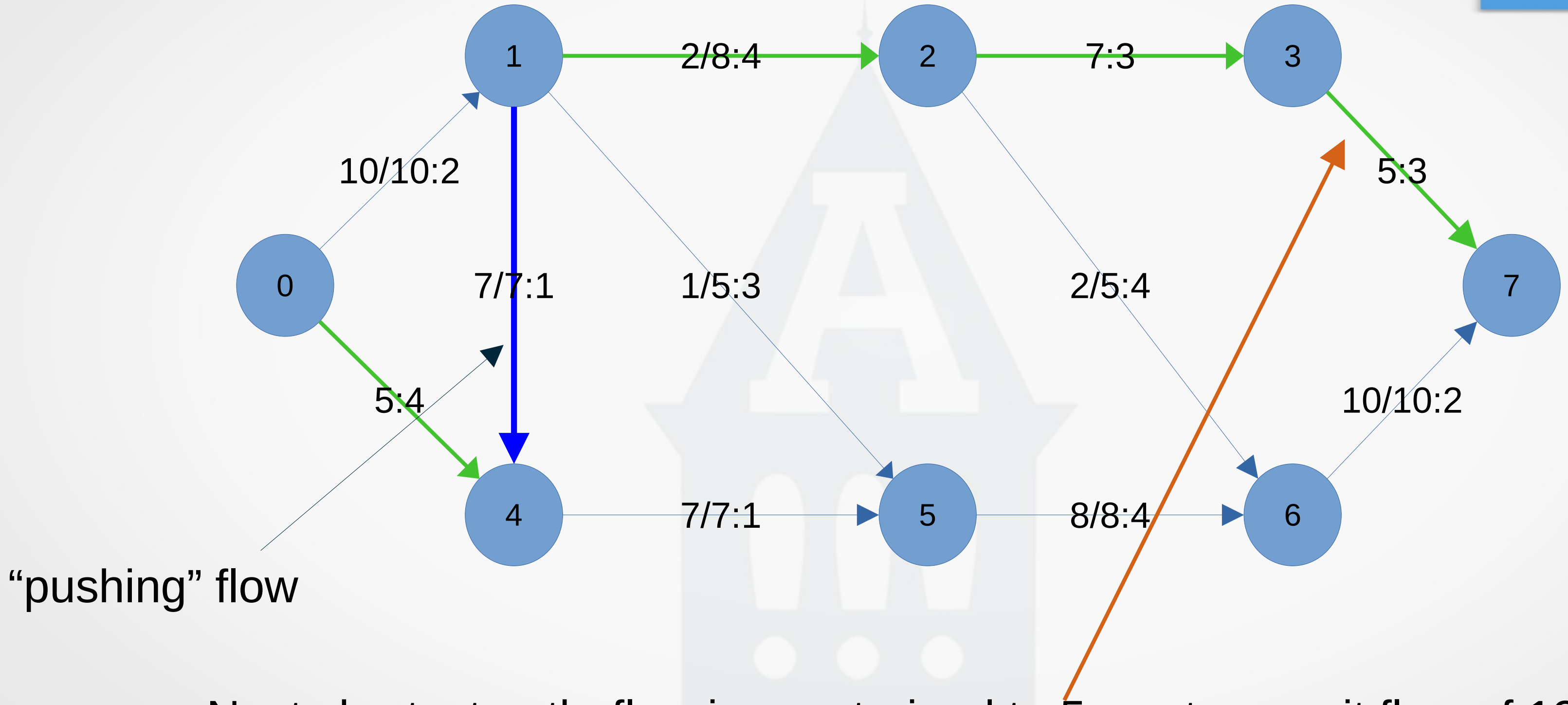
Next shortest path; flow is constrained to 2, cost per unit flow of 12

Min-Cost, Max-Flow



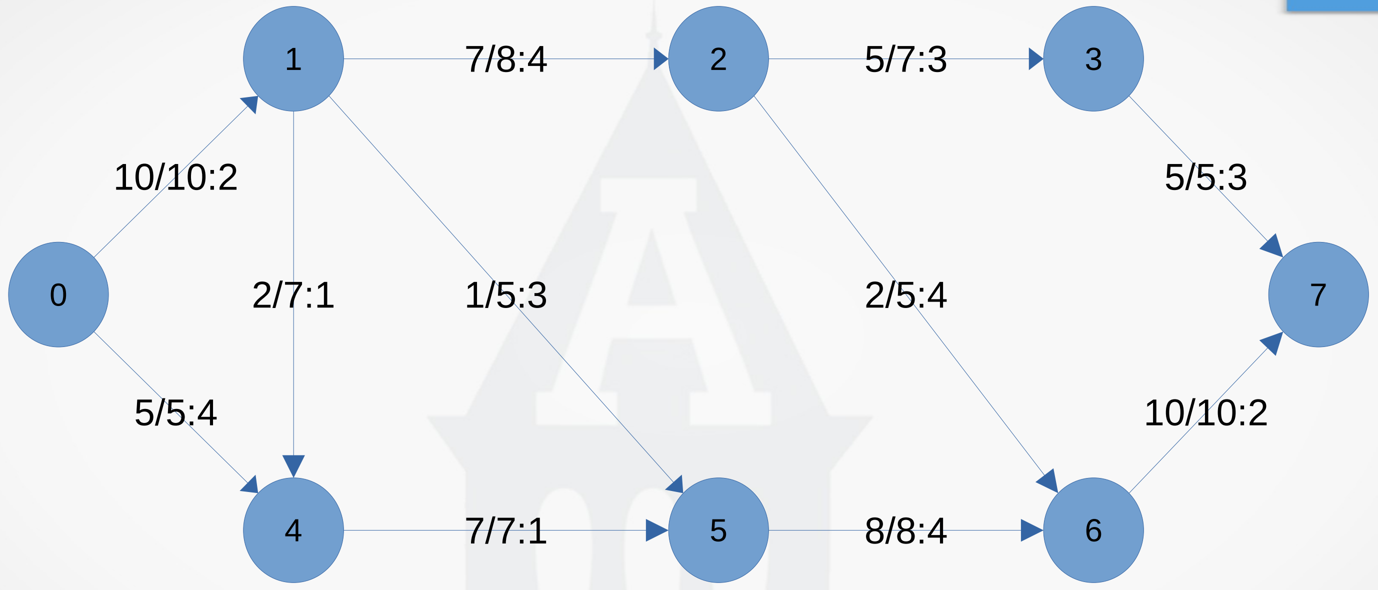
Update with this flow

Min-Cost, Max-Flow



Next shortest path; flow is constrained to 5, cost per unit flow of 13

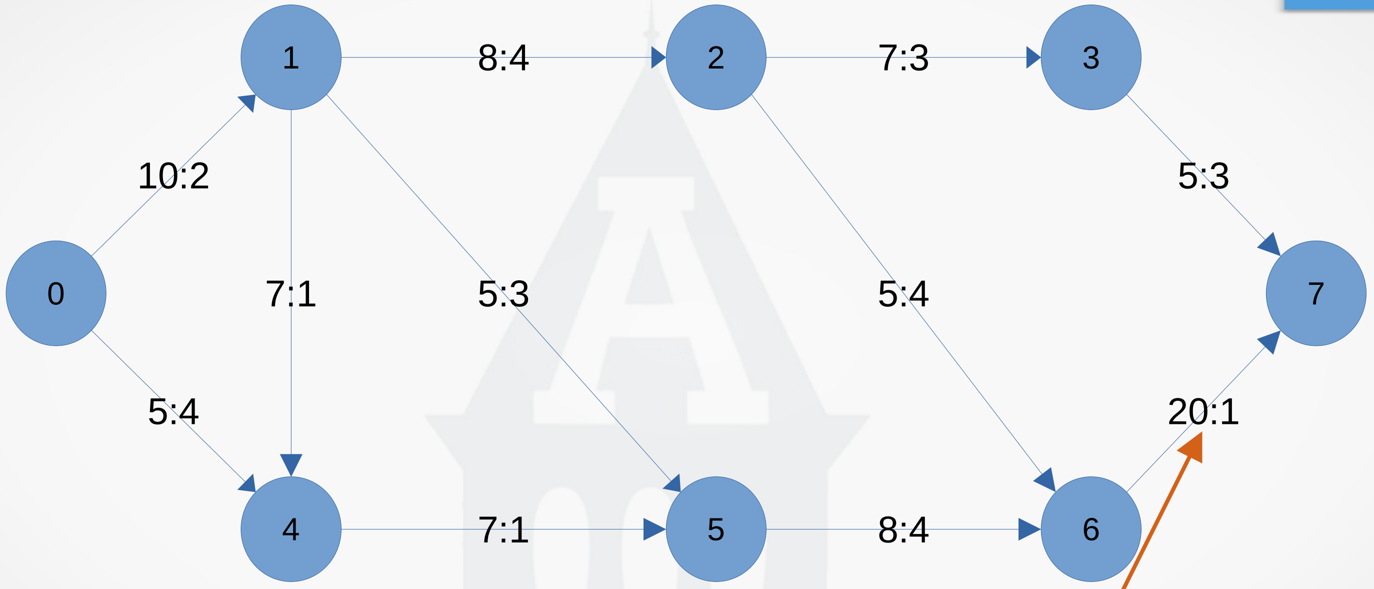
Min-Cost, Max-Flow



Update with this flow

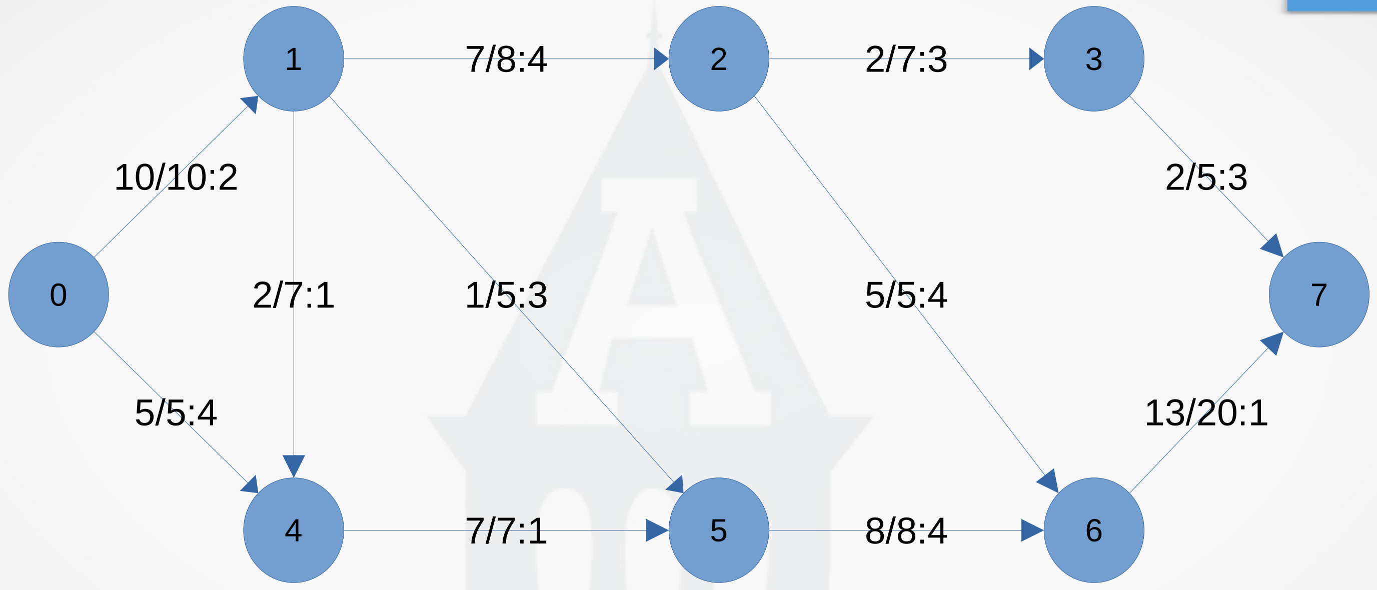
Total cost is 170

Min-Cost, Max-Flow



Let’s change this from 10:2 to 20:1

Min-Cost, Max-Flow



Same amount of flow, but cost of 157