CS161 Final Review

Hi, I'm Will! :)

Disclaimer: there's a TON to get through, so I'll do my best + focus on some topics from the latter half of the course, but please please please study up on everything!

Agenda

1. Shortest Path Algorithms 🎯
2. Dynamic Programming 😎
3. Greedy Algorithms 🤑
4. Minimum Spanning Trees 🌲
5. Max-Flow, Min-Cut, Ford-Fulkerson 🚰

Shortest-Path Algorithms

For a given graph…

Shortest path problem: what's the way to get from Stanford to everywhere else with the **least cost**?

Big idea for Dijkstra!

Shortest paths of shortest paths are shortest paths.

Dijkstra Pseudocode

Dijkstra(G,s):

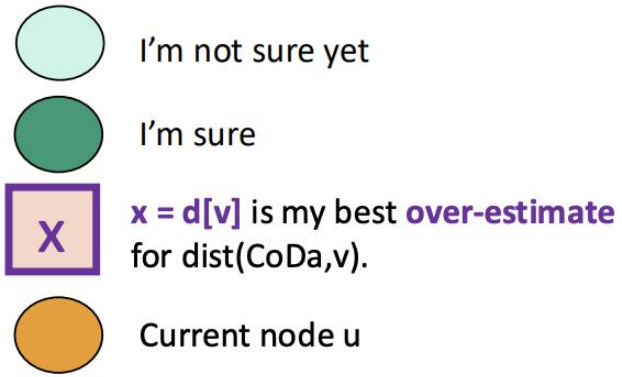
- Set all vertices to **not-sure**
- d[v] = ∞ for all v in V (except s)
- d[s] = 0
- **While** there are **not-sure** nodes:
  - Pick the **not-sure** node u with the smallest estimate **d[u]**.
  - Mark u as **sure**.
  - **For** v in u.neighbors:
    - d[v] ⇐ min(d[v] , d[u] + edgeWeight(u,v))
- Now d(s, v) = d[v]

Shouldn't be used on graphs with negative edges!

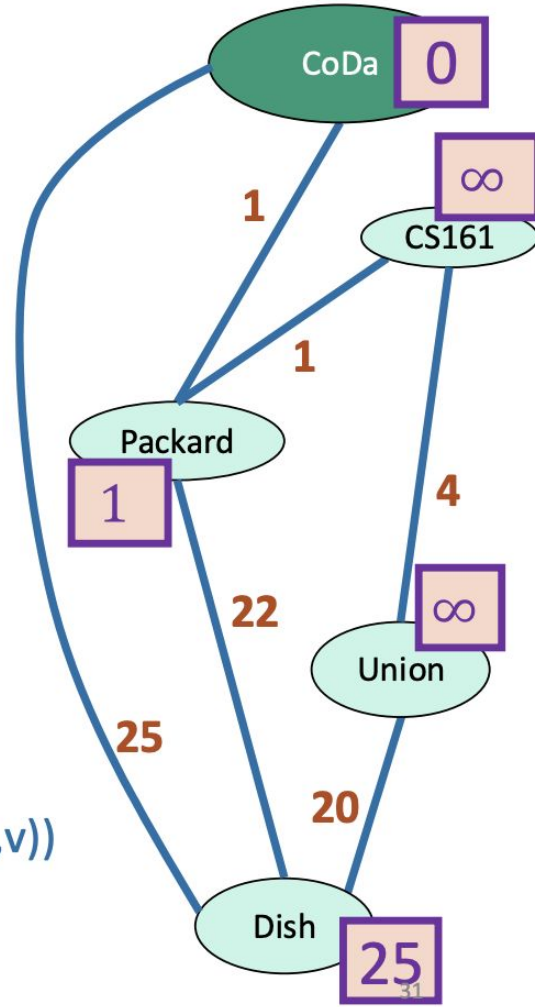
Runtime? Will need to use Fibonacci heaps
to make operations highlighted operations run faster!

Dijkstra by example

How far is a node from CoDa?



- Pick the **not-sure** node u with the smallest estimate **d[u].**
- Update all u's neighbors v:
  - d[v] = min(d[v] , d[u] + edgeWeight(u,v))
- Mark u as **sure**.
- Repeat



Dijkstra with **Fibonacci Heap**

- T(extractMin) = O(log(n)) (amortized time*)
- T(decreaseKey) = O(1) (amortized time*)
- See CS166 for more! (or CLRS)

- Running time of Dijkstra
  - = O(n(T(`extractMin`)) + m T(`decreaseKey`))
  - = O(nlog(n) + m)

*This means that any sequence of d `extractMin` calls takes time at most O(dlog(n)). But a few of the d may take longer than O(log(n)) and some may take less time..

Check-In!

1. What is Dijkstra's runtime? With what data structure?

2. T/F: Dijkstra can be used for finding a single-source shortest path solution with negative edges but not negative cycles

3. T/F: Dijkstra is the most efficient way to solve for a shortest path in an unweighted graph

4. T/F: Adding a new positive edge to an undirected weighted graph w/ positive edges cannot increase the output values of Dijkstra

Bellman-Ford

Dijkstra…but now we update every vertex each time!

Bellman-Ford (algorithm)

Bellman-Ford(G,s):

- $d^{(0)}[v] = \infty$ for all v in V
- $d^{(0)}[s] = 0$

// initialize:
$d^{(i)}[v]$ is distance from s to v with $\leq$i edges

- **For** i=0,…,n-2:
  - $d^{(i+1)}[v] = d^{(i)}[v]$ for all v in V

// baseline distance:
v doesn't need $(i+1)^{th}$ edge

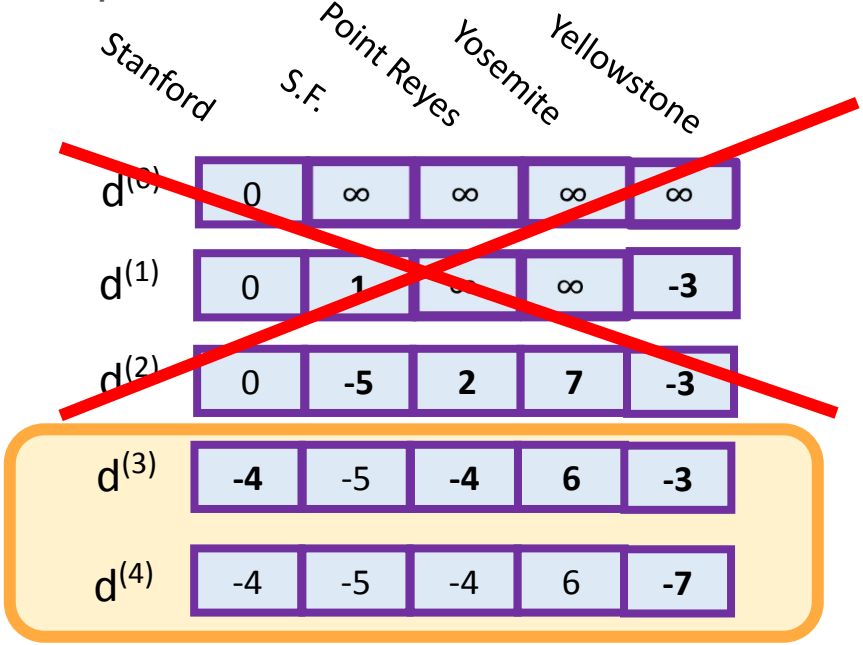
  - **For** v in V:
    - **For** u in v.neighbors:
      - $d^{(i+1)}[v] \leftarrow \min(d^{(i+1)}[v], d^{(i)}[u] + \text{edgeWeight}(u,v))$

// found a better path through u

- **Return** $d^{(n-1)}$

Bellman-Ford (implementation)

- Don't actually keep all n arrays around.
- Just keep two at a time: "last round" and "this round"



Only need these two in order to compute $d^{(4)}$

n = # of vertices
m = # of edges

Bellman-Ford (takeaways)

- Running time is O(mn)
  - For each of n rounds, update m edges.

- **For** i=0,…,n-1:
  - **For** u in V:
    - **For** v in u.neighbors:

m = # of edges
$= \frac{1}{2}\sum_{v \in V} \text{degree}(v)$

- Works fine with negative edges.
- Does not work with negative cycles.
  - But it can detect negative cycles!

Pop Quiz! 😱

What are the differences between Dijkstra and Bellman-Ford?

Floyd-Warshall algorithm (motivation)

- This is an algorithm for **All-Pairs Shortest Paths (**APSP)
  - That is, I want to know the shortest path from u to v for **ALL pairs** u,v of vertices in the graph.
  - Not just from a special single source s.
- Naïve solution:
  - For all s in G:
    - Run Bellman-Ford on G starting at s.
  - Time $O(n \cdot nm) = O(n^2 m)$,
    - may be as bad as n^4 if $m=n^2$

Can we do better?

Floyd-Warshall algorithm

- Initialize n-by-n arrays $D^{(k)}$ for k = 0,…,n
  - $D^{(k)}[u,u] = 0$ for all u, for all k
  - $D^{(k)}[u,v] = \infty$ for all $u \neq v$, for all k
  - $D^{(0)}[u,v]$ = weight(u,v) for all (u,v) in E.
- **For** k = 1, …, n:
  - **For** pairs u,v in V^2:
    - $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$
- **Return** $D^{(n)}$

The base case checks out: the only path through zero other vertices are edges directly from u to v.

$D^{(k)}[u,v]$:= length of shortest path between u and v that only pass through vertices 1,2,...,k in G

n = # of vertices
m = # of edges

Floyd-Warshall algorithm (analysis)

- Theorem:
  If there are no negative cycles in a weighted directed graph G, then the Floyd-Warshall algorithm, running on G, returns a matrix $D^{(n)}$ so that:
  $D^{(n)}[u,v]$ = distance between u and v in G.

- Running time: $O(n^3)$
  - Better than running Bellman-Ford n times!

Work out the details of a proof!

- Storage:
  - Need to store **two** n-by-n arrays, and the original graph.

As with Bellman-Ford, we don't really need to store all n of the $D^{(k)}$.

B-F and F-W are examples of DYNAMIC PROGRAMMING!!

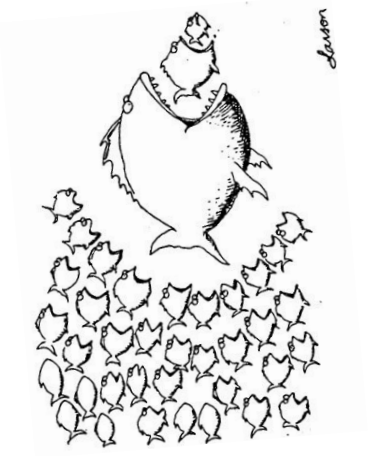
This is a great transition to…

What is ***dynamic programming***?

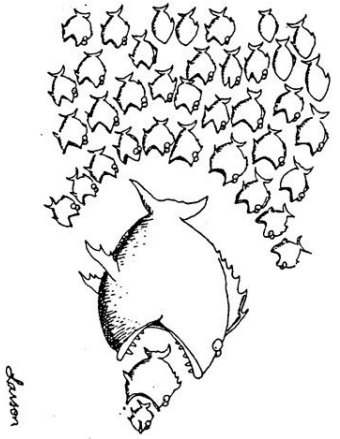
- It is an algorithm design paradigm
  - (like divide-and-conquer is an algorithm design paradigm)
- Usually it is for solving **optimization problems**
  - **Most x given constraint y**
- Examples:
  - (Fibonacci numbers aren't an optimization problem, but they are a good example…)
  - Knapsack
  - Longest-Common Subsequence
  - Graph Algorithms! (which ones?)

Bottom up approach

- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]



Top down approach



- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..

- The difference from divide and conquer:
  - **Memoization**
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.

Quick check-in!

Which is iterative? Which is recursive? What do we use to improve the recursive one?

Ok cool…how do I get these cool DP solutions?

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.

What are some subproblems and recursive formulations we've seen?

Practice: # of paths

- You have an nxn grid where each cell is 0 (free) or 1 (obstacle)
- A robot starts at the top-left (0,0) and wants to make it to (n-1, n-1)
- It can move down one space OR right one space BUT it can't go through an obstacle
- Compute the number of unique paths from start to finish that avoid obstacles
- Specifically, what are our subproblems? What is the recurrence? Base cases?

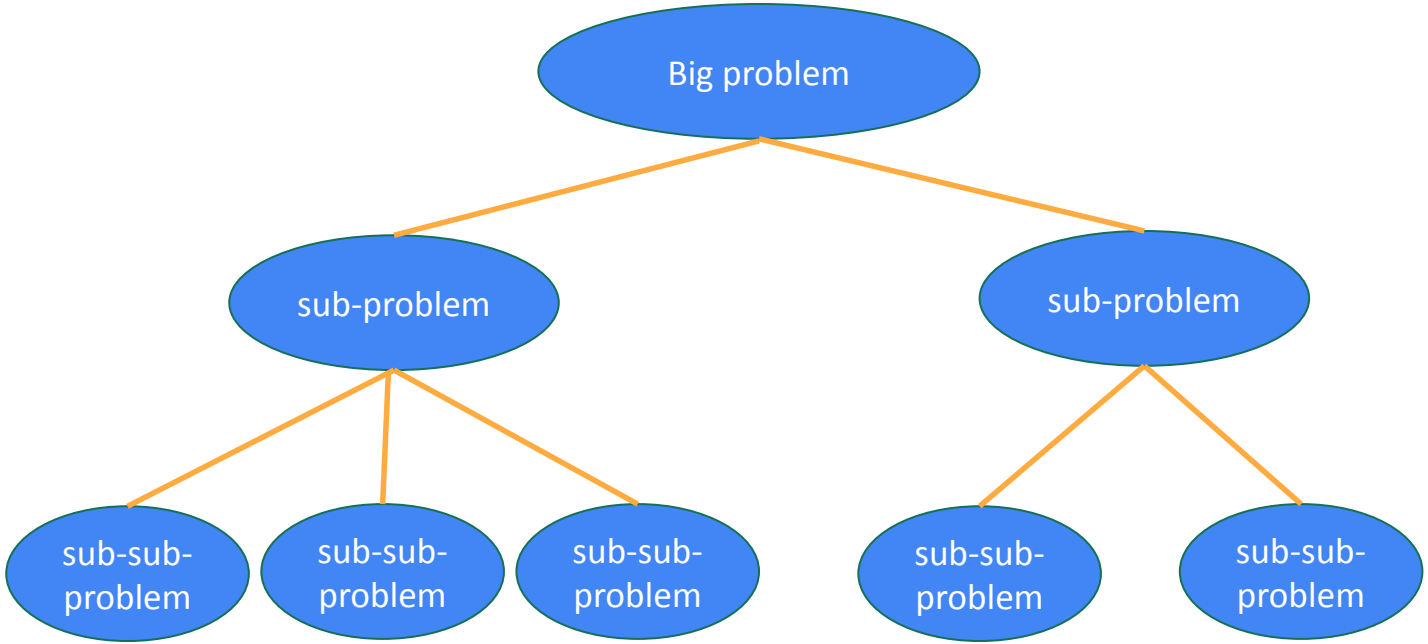
Greedy Algorithms

Intuition

Let me just pick what’s best in this current moment (very short-sighted)!

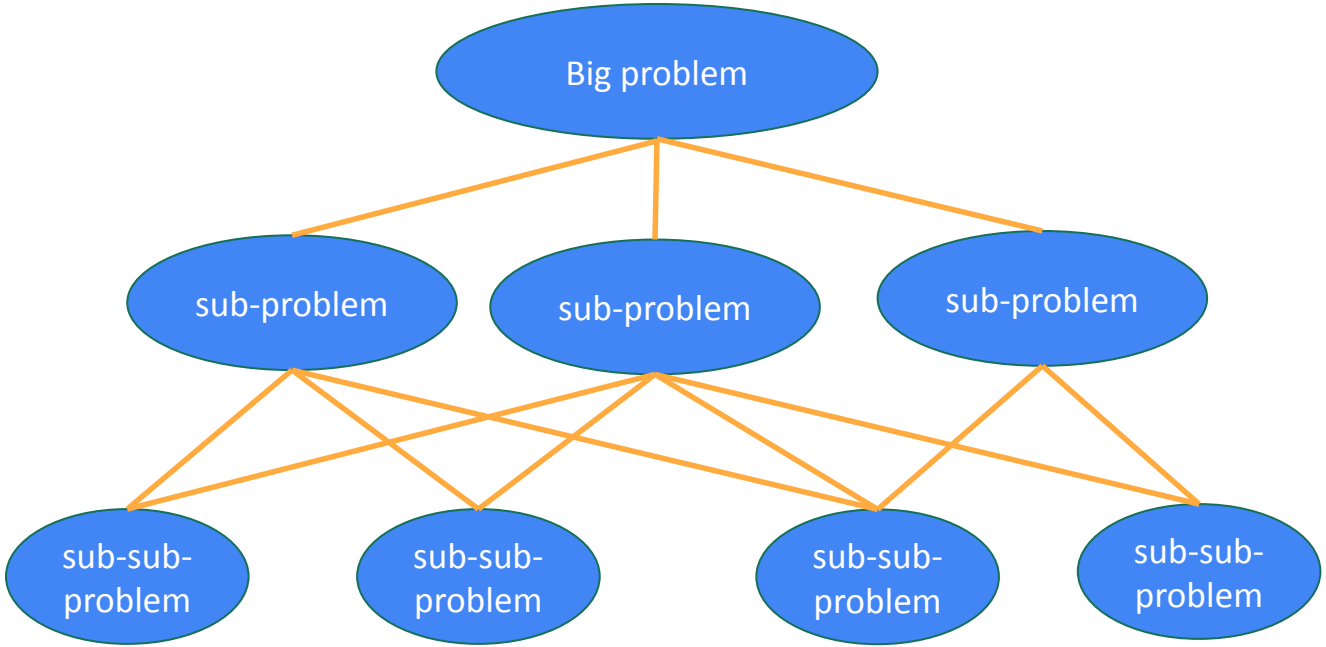
Sub-problem graph view

- Divide-and-conquer:



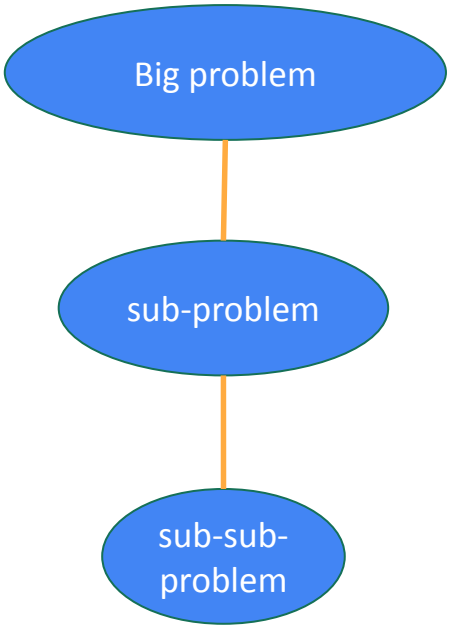
Sub-problem graph view

- Dynamic Programming:



Sub-problem graph view

- Greedy algorithms:



- Not only is there **optimal sub-structure:**
  - optimal solutions to a problem are made up from optimal solutions of sub-problems
- but each problem **depends on only one sub-problem**.

When does this not work?

Capacity: 10

Item:	🐢	💡	🍉	🌮	🚒
Weight:	6	2	4	3	11
Value:	20	8	14	13	35

- Unbounded Knapsack:
  - Suppose I have infinite copies of all of the items.
  - What's the most valuable way to fill the knapsack?

🌮 🌮 💡 💡

Total weight: 10
Total value: 42

- "Greedy" algorithm for unbounded knapsack:
  - Tacos have the best Value/Weight ratio!
  - Keep grabbing tacos!

🌮 🌮 🌮

Total weight: 9
Total value: 39

Not optimal!! 😮

When DOES this work?
How do we show that???

Common strategy
for greedy algorithms

- Make a **series of choices**.
- Show that, at each step, our choice **won't rule out all optimal solutions** at the end of the day.
- After we've made all our choices, there is an optimal solution we haven't ruled out, **so we must have found one.**

Common strategy (formally)
for greedy algorithms

"Success" here means "finding an optimal solution."

- Inductive Hypothesis:
  - After greedy choice t, you haven't **ruled out** success.
- Base case:
  - Success is possible before you make any choices.
- Inductive step:
  - If you haven't ruled out success after choice t, then you won't rule out success after choice t+1.
- Conclusion:
  - If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

Common strategy
for showing we don't rule out success

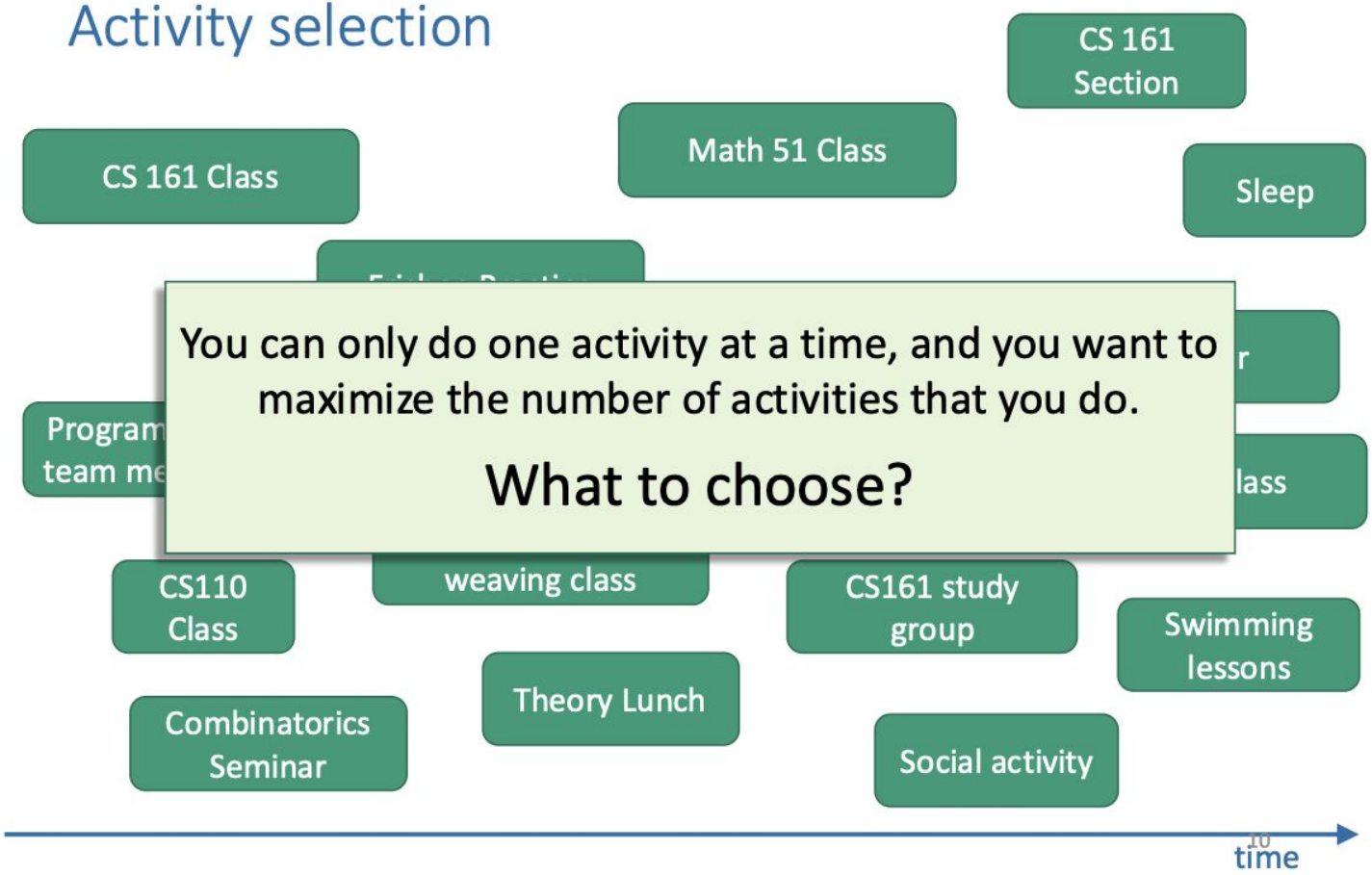
- Suppose that you're on track to make an optimal solution T*.
  - Eg, after you've picked activity i, you're still on track.
- Suppose that T* *disagrees* with your next greedy choice.
  - Eg, it *doesn't* involve activity k.
- Manipulate T* in order to make a solution T that's not worse but that *agrees* with your greedy choice.
  - Eg, swap whatever activity T* did pick next with activity k.

Uhhh… what does that mean?

Let’s see an example!

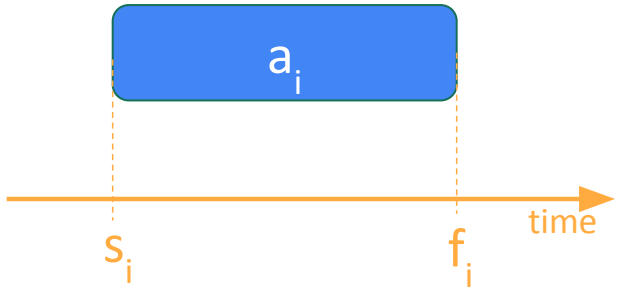
Example where greedy works

Activity selection



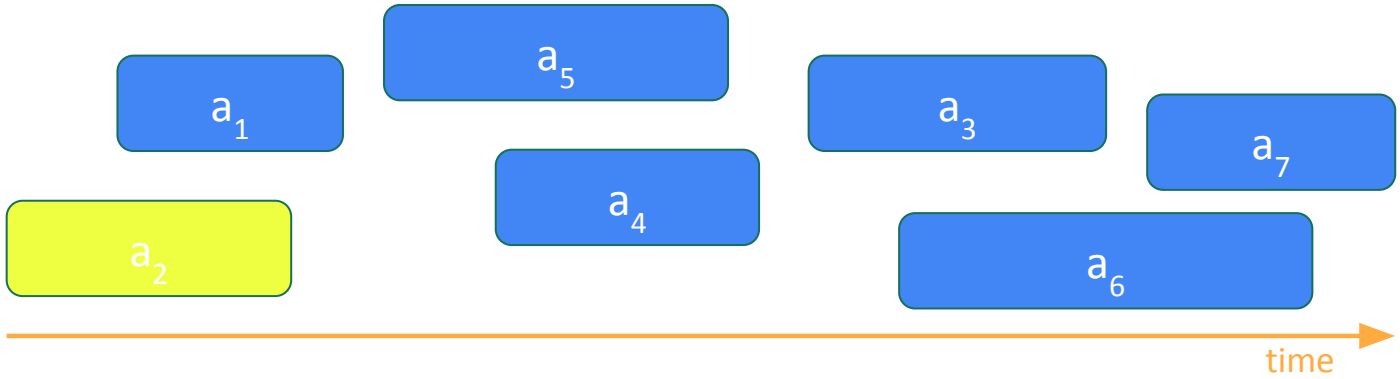
Activity selection

- Input:
  - Activities $a_1, a_2, \ldots, a_n$
  - Start times $s_1, s_2, \ldots, s_n$
  - Finish times $f_1, f_2, \ldots, f_n$



- Output:
  - A way to maximize the **number of activities** you can do today.

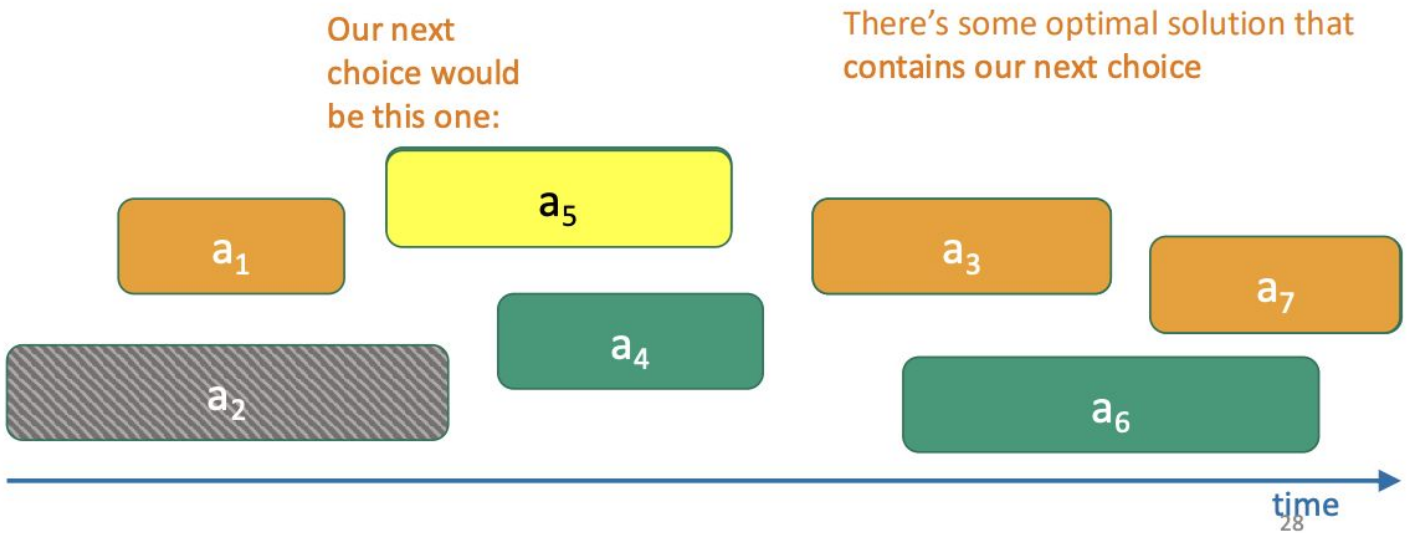
Greedy algorithm



- Pick activity you can add with the smallest finish time.
- Repeat.

Why does it work?

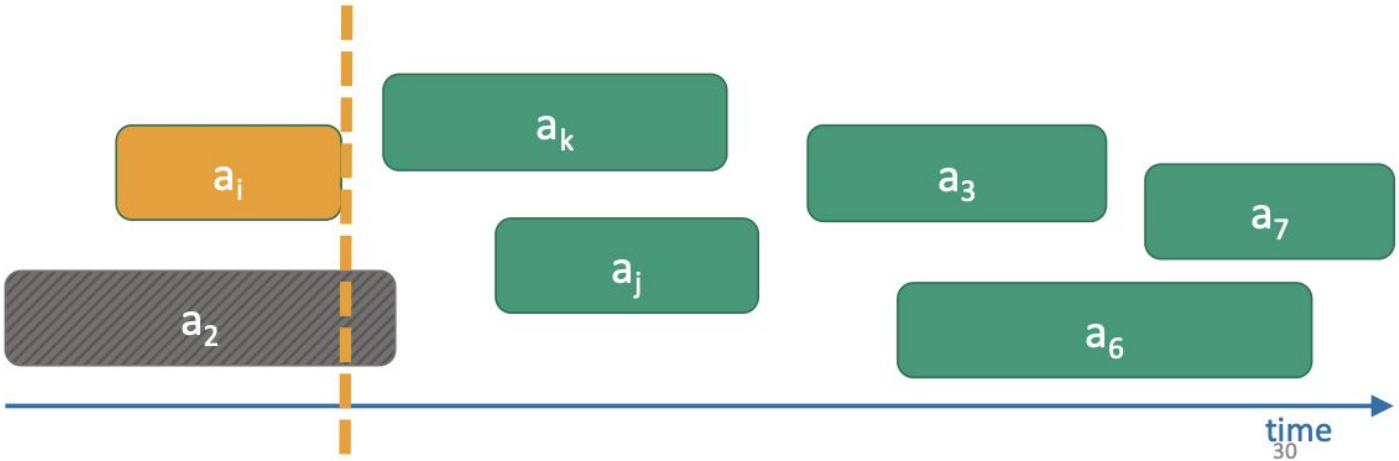
- Whenever we make a choice, **we don't rule out an optimal solution.**



Proof of:

We never rule out an optimal solution

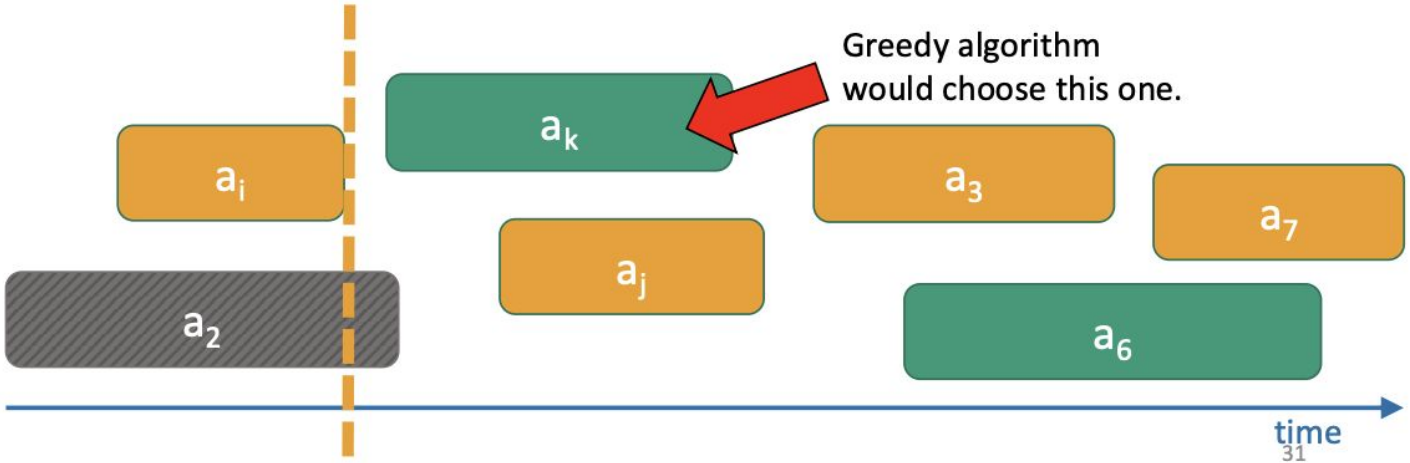
- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.



Proof of:

We never rule out an optimal solution

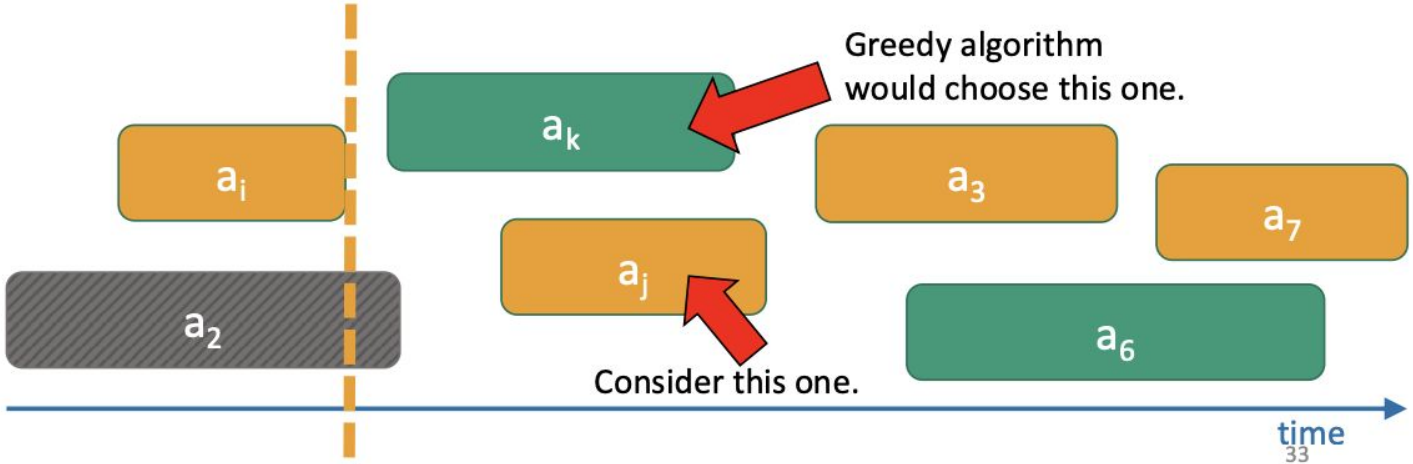
- Suppose we've already chosen a_i, and there is still an optimal solution T* that extends our choices.
- Now consider the next choice we make, say it's a_k.
- If a_k is in T*, we're still on track.



Proof of:

We never rule out an optimal solution ctd.

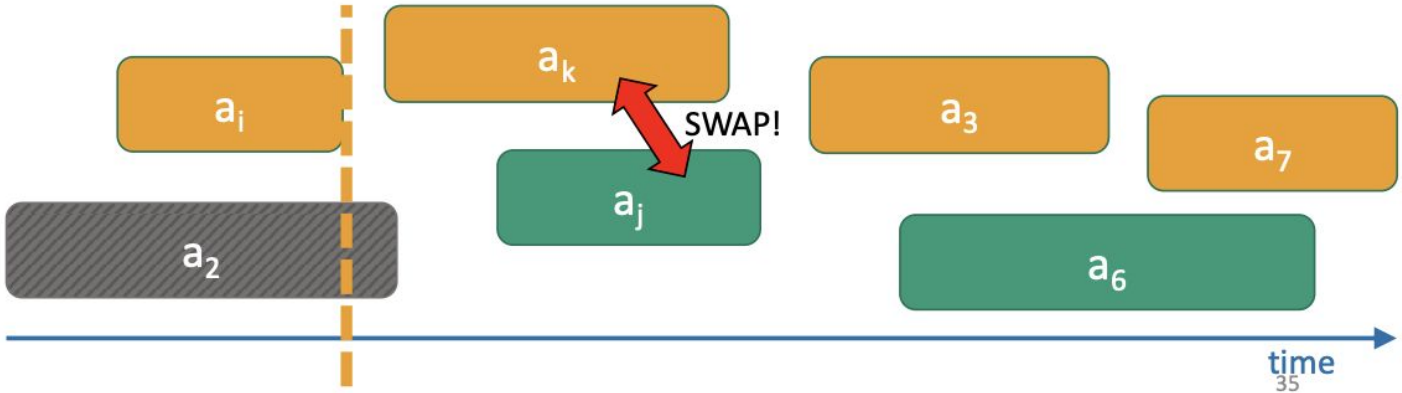
- If a_k is **not** in T*…
- Let a_j be the activity in T* with the smallest end time.
- Now consider schedule T you get by swapping a_j for a_k



Proof of:

We never rule out an optimal solution ctd.

- This schedule T is still allowed.
  - Since a_k has the smallest ending time, it ends before a_j.
  - Thus, a_k doesn't conflict with anything chosen after a_j.
- And, T is still optimal.
  - It has the same number of activities as T*.



Common strategy (formally)
for greedy algorithms

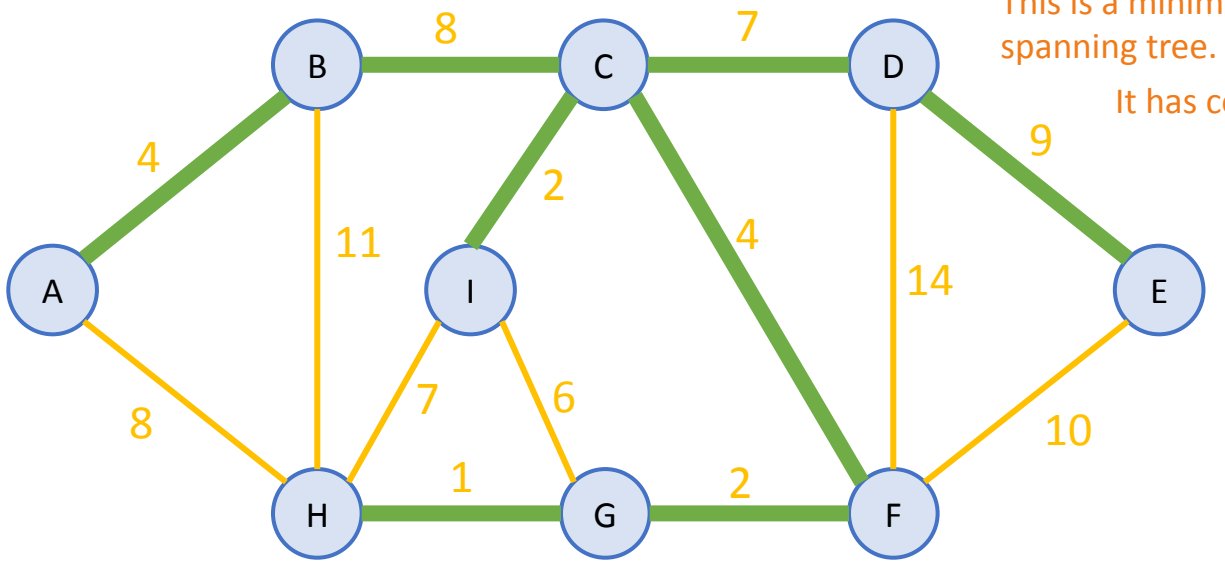
"Success" here means "finding an optimal solution."

- Inductive Hypothesis:
  - After greedy choice t, you haven't **ruled out** success.
- Base case:
  - Success is possible before you make any choices.
- **Inductive step:**
  - ✅✅✅✅✅✅✅✅✅ **If you haven't ruled out success after choice t, then you won't rule out success after choice t+1.**
- Conclusion:
  - If you reach the end of the algorithm and haven't ruled out success then you must have succeeded.

Minimum Spanning Trees

Minimum Spanning Tree

Say we have an undirected weighted graph



This is a minimum spanning tree.

It has cost 37

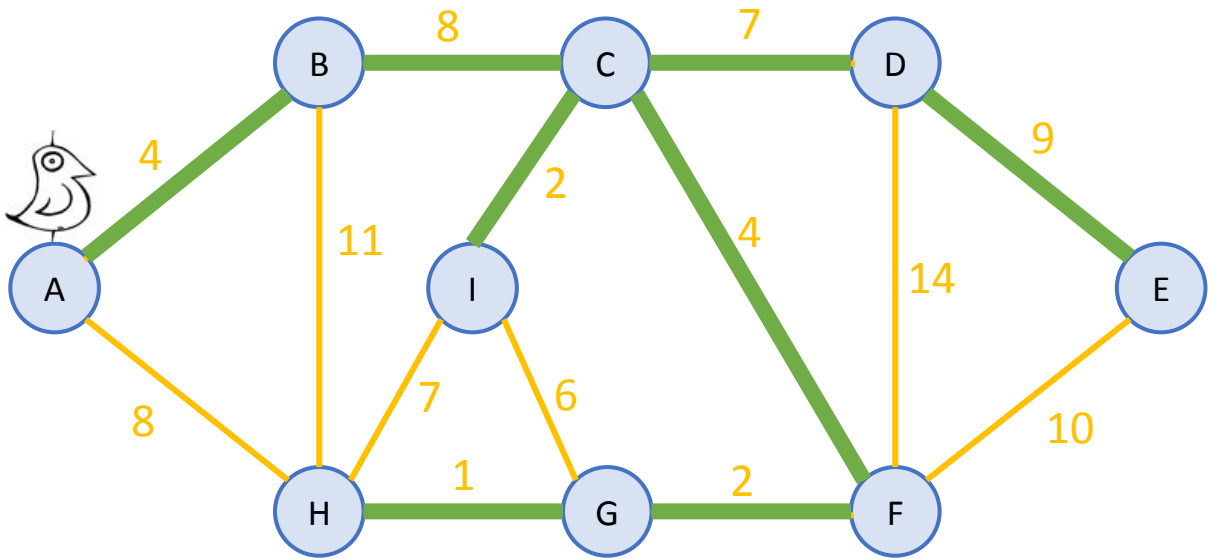
A **spanning tree** is a **tree** that connects all of the vertices.

minimum

of minimal cost

Algorithms for Minimum Spanning Tree

Plan 1: Start growing a tree, greedily add the shortest edge we can to grow the tree.



Prim's Algorithm

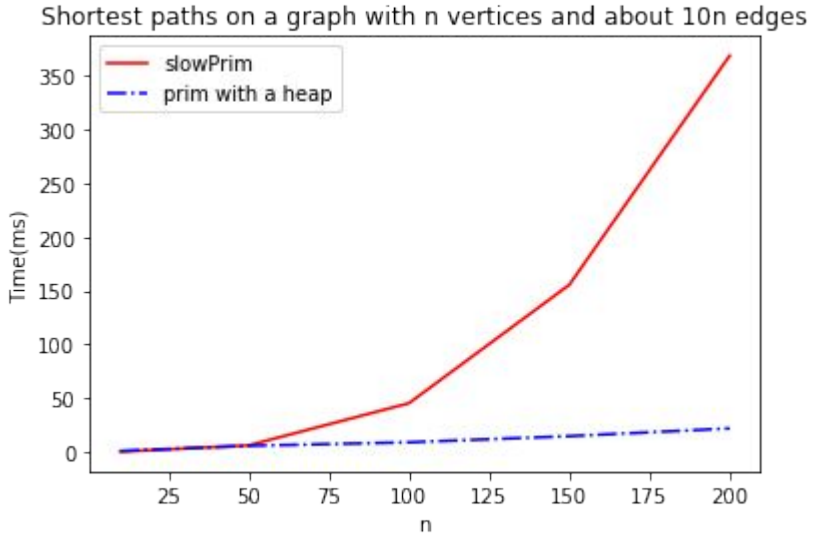
Prim(G = (V,E), starting vertex s):

- MST = {}
- **for each** vertex v:
  - k[v] = ∞
  - mark v as **disconnected**
- k[s] = 0; mark s as **connected**

- **Until** all the vertices are **connected:**
  - Activate the **disconnected** vertex u with the smallest key.
  - Mark u as **connected,** and **add (p[u],u) to MST.**
  - **for each** of u's neighbors v:
    - k[v] = min(k[v], weight(u,v))
    - if k[v] updated, p[v] = u

Prim's Runtime

- Exactly the same as Dijkstra: O(nlogn + m) with a Fibonacci heap!

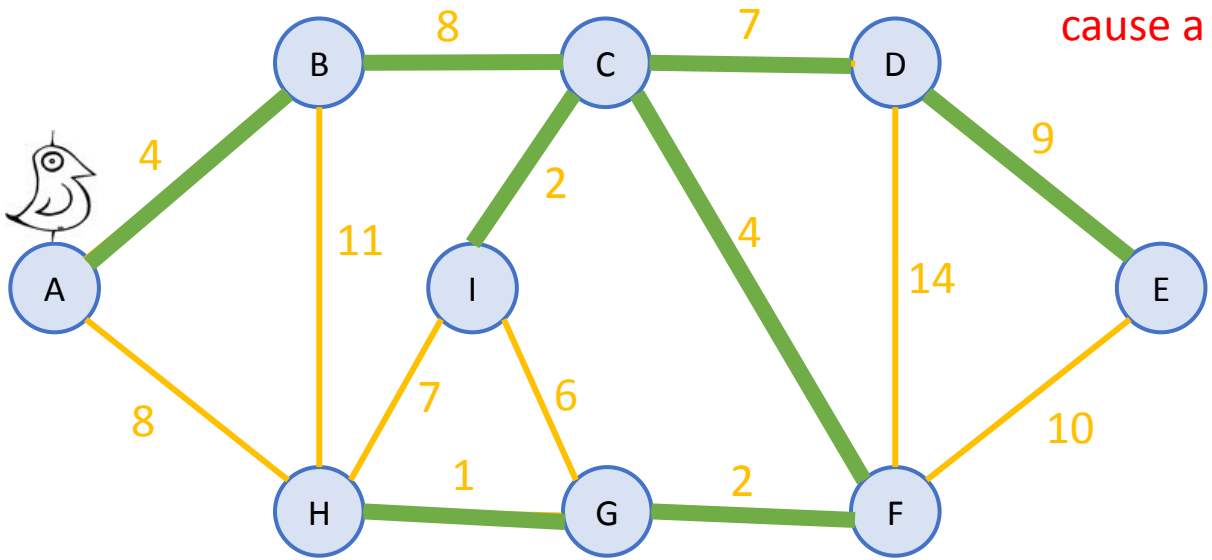


Idea 2

what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?

That won't cause a cycle



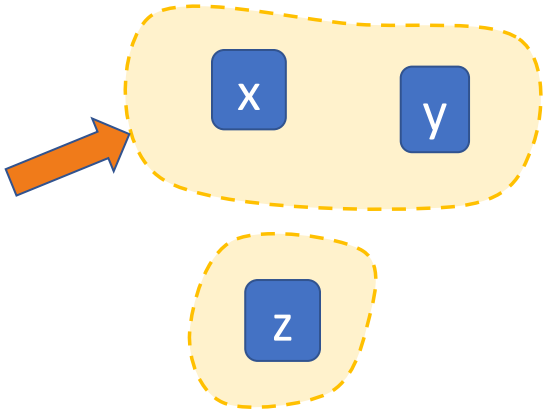
Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

find(x)
```



Kruskal's algorithm

- **kruskal**(G = (V,E)):
  - Sort E by weight in non-decreasing order
  - MST = {} // initialize an empty tree
  - **for** v in V:
    - **makeSet**(v) // put each vertex in its own tree in the forest
  - **for** (u,v) in E: // go through the edges in sorted order
    - **if find**(u) != **find**(v): // if u and v are not in the same tree
      - add (u,v) to MST
      - **union**(u,v) // merge u's tree with v's tree
  - **return** MST

Compare and contrast

- Prim:
  - Grows a tree.
  - Time O(mlog(n)) with a red-black tree
  - Time O(m + nlog(n)) with a Fibonacci heap
- Kruskal:
  - Grows a forest.
  - Time O(mlog(n)) with a union-find data structure
  - If you can do radixSort on the edge weights, close to O(m)

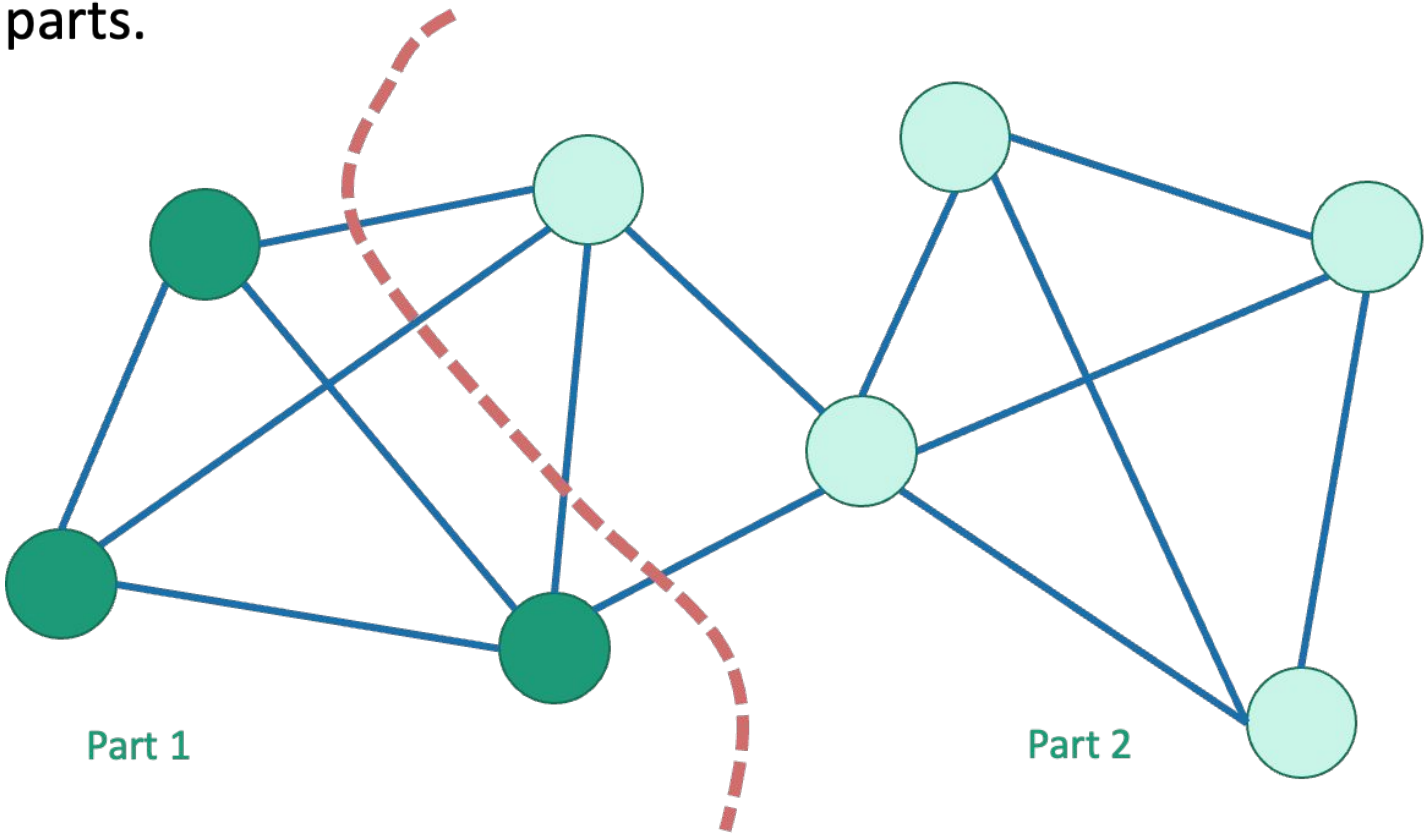
Prim might be a better idea on dense graphs if you can't radixSort edge weights

Kruskal might be a better idea on sparse graphs if you can radixSort edge weights

min-cut max-flow

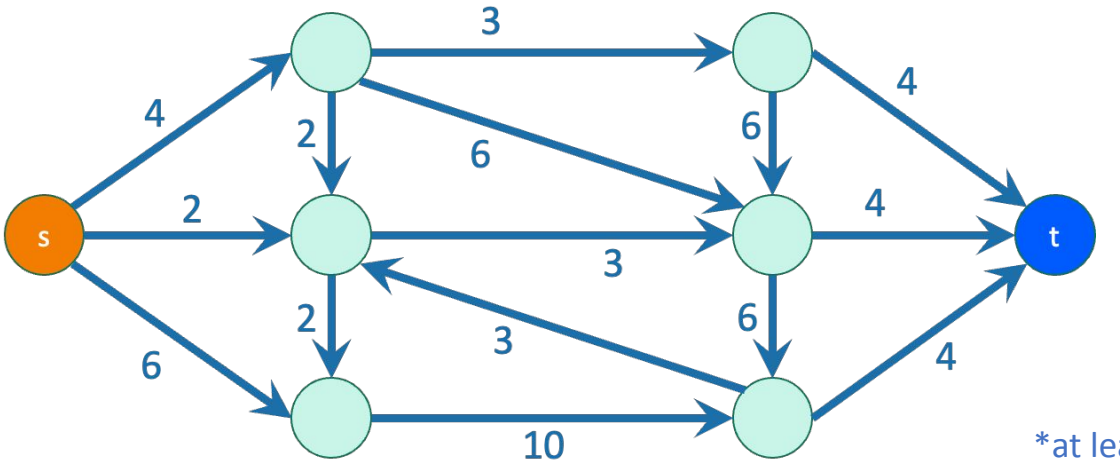
Cuts in graphs

- A cut is a partition of the vertices into two nonempty parts.



s-t Cuts

- Graphs are directed and edges have “capacities” (weights)
- We have a special “source” vertex s and “sink” vertex t.
  - s has only outgoing edges*
  - t has only incoming edges*
- s and t are on different sides of the cut

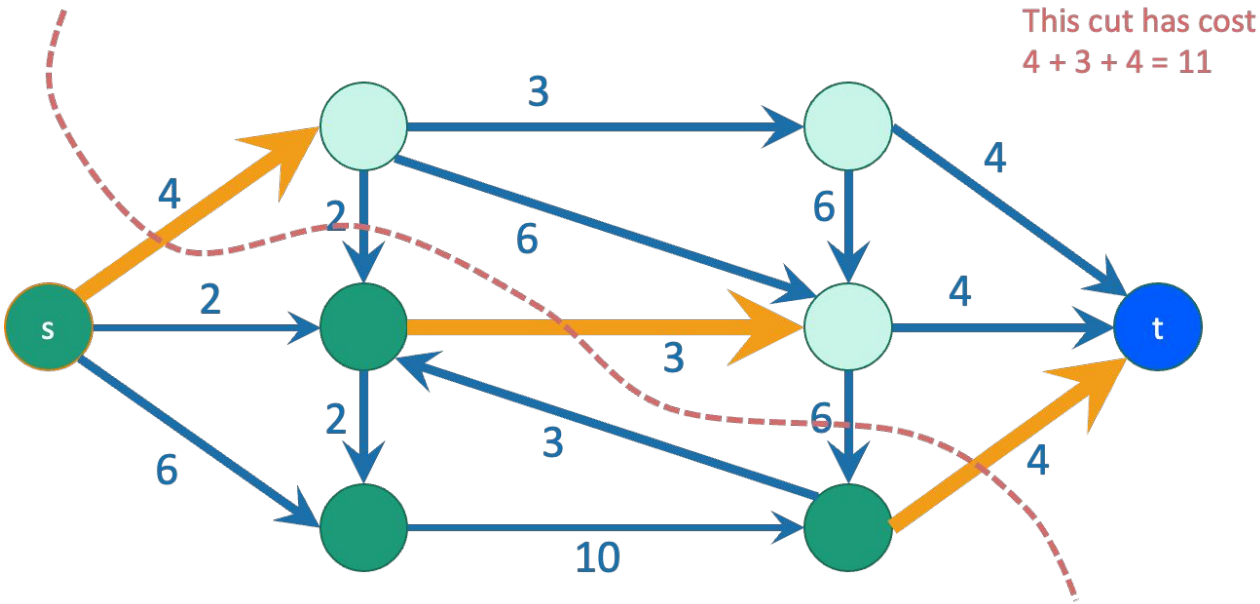


*at least for this class

A minimum **s-t cut**

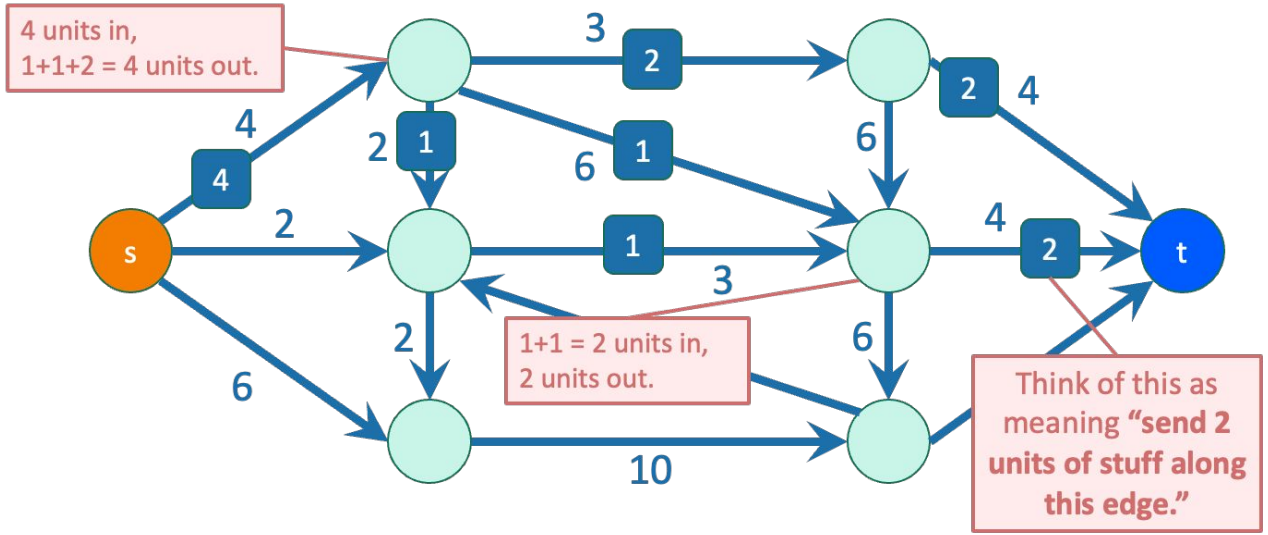
is a cut which separates s from t
with minimum capacity.

- Question: how do we find a minimum s-t cut?



Flows

- In addition to a capacity, each edge has a **flow**
  - (unmarked edges in the picture have flow 0)
- The flow on an edge must be less than or equal to its capacity.
- At each vertex other than s and t, the incoming flows must equal the outgoing flows.



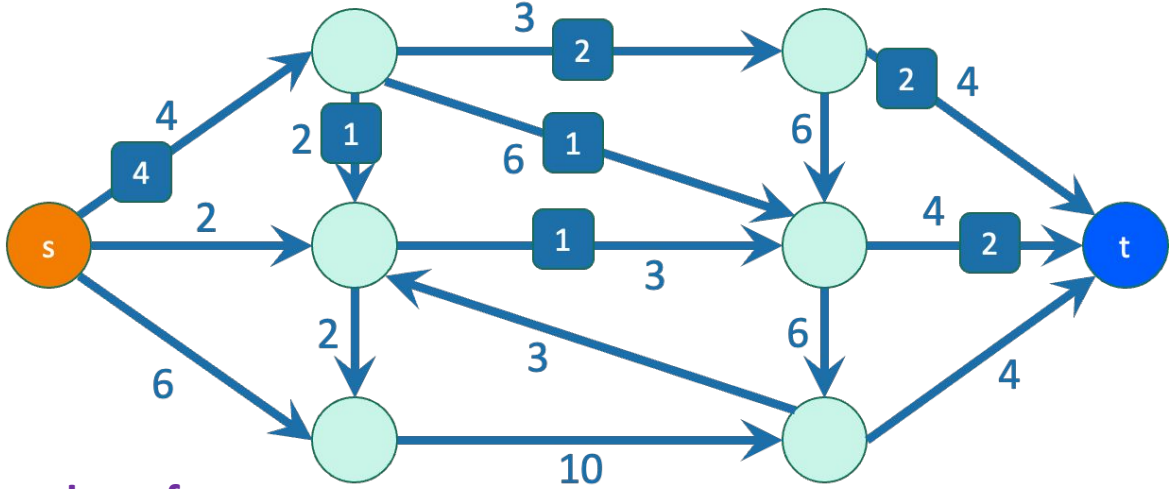
But what is a “flow,” really?

- Think of s as a “water source” and t as a “water sink”
- Each edge is a pipe, and the capacity is the amount of water that can flow through the edge
- Clearly, for each pipe, as much water comes in must go out!
- Some pipes will be the “bottleneck” pipes… could these be related to the min-cut somehow?

Flows

- The value of a flow is:
  - The amount of stuff coming out of s
  - The amount of stuff flowing into t
  - These are the same!

Because of conservation of flows at vertices,
stuff you put in
=
stuff you take out.



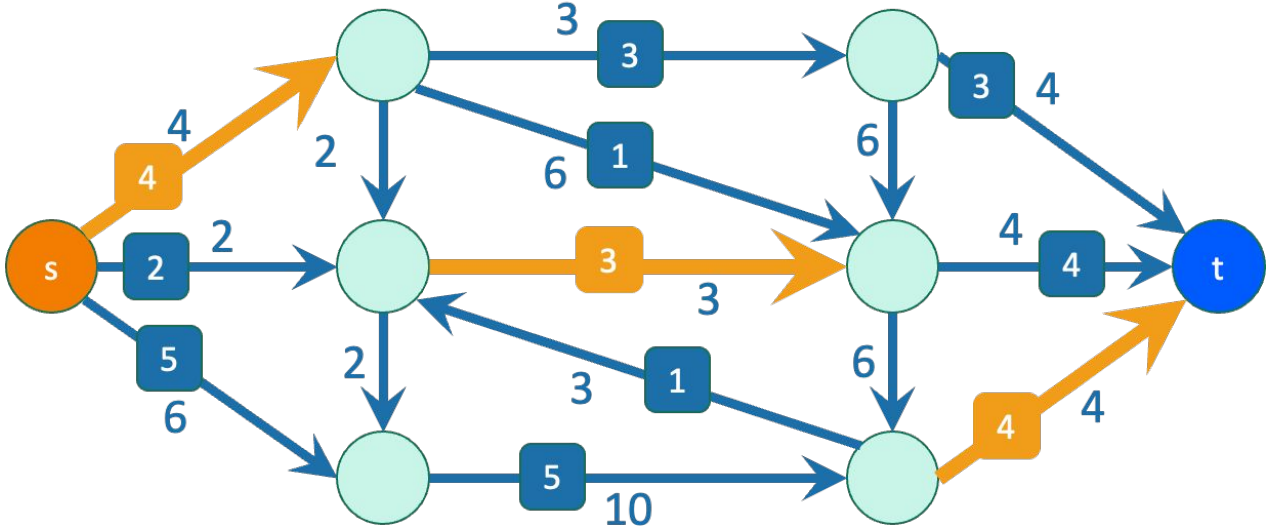
The value of this flow is 4.

Theorem

Max-flow min-cut theorem

The value of a max flow from s to t
is equal to
the capacity of a min s-t cut.

Intuition: in a max flow, the min cut better fill up, and this is the bottleneck.



Max-flow min-cut in water pipes

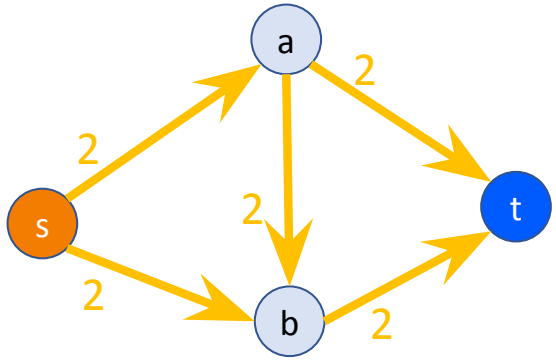
- What does the max-flow min-cut theorem mean in the context of our water pipes example before?
- Max-flow = maximum amount of water we can send
- Min-cut = the minimum capacity of pipes, which when removed, completely disconnects the water source and sink (i.e., no water flows)
- Intuitively makes sense!

Ford-Fulkerson algorithm

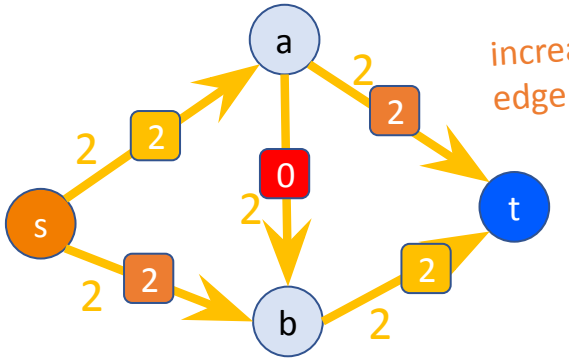
Outline of algorithm:

- Start with zero flow
- While we haven't found the max flow:
  - Find a way to send more from s to t

Original graph:



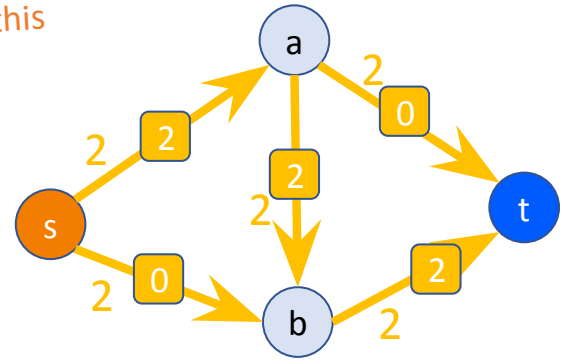
Iteration 3: optimal flow!



increased flow on this edge

decreased flow on this edge

Iteration 2: increase the flow again:



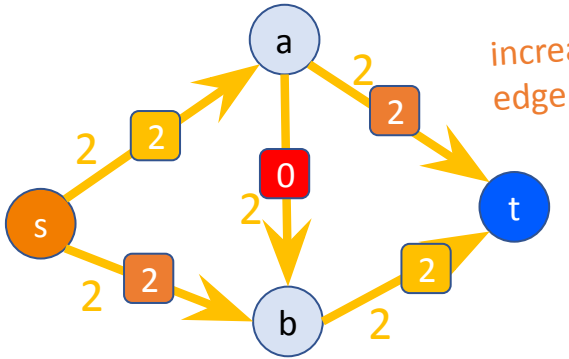
2 key ideas

1. In order to increase total flow, we *decreased* flow on some edges

2. Decreasing flow in one direction = increasing flow in the opposite direction

= "sending back" this flow

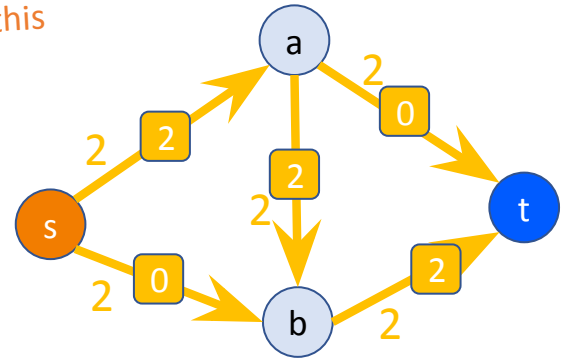
Iteration 3: optimal flow!



increased flow on this edge

decreased flow on this edge

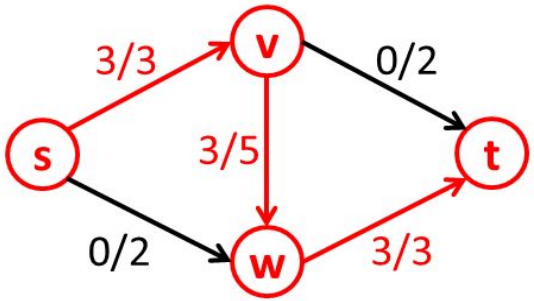
Iteration 2: increase the flow again:



Why residual graphs?

Let's think of a greedy idea for finding flows:

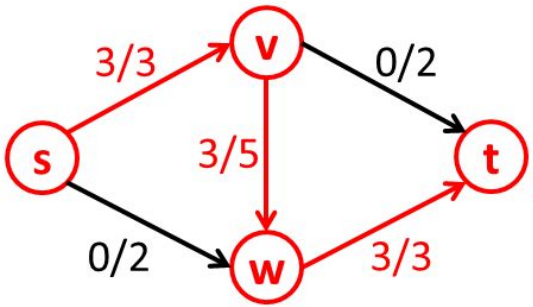
- Pick an arbitrary augmenting path, and push the maximum flow you can through this path
- Keep repeating this until no paths exist!
- We reach a "blocking" flow: no more paths exist, even though the flow is not optimal… what to do?

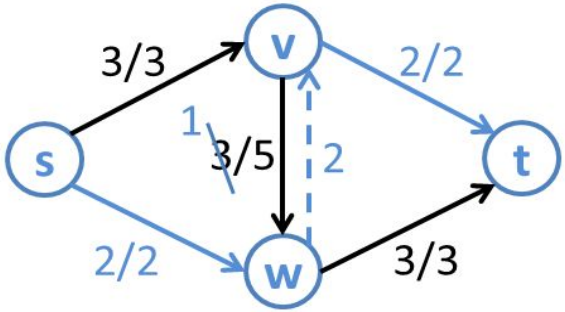


Reference: https://cs.stackexchange.com/a/71505

Why residual graphs?

- Fix this issue by allowing "**undo**" operations!
- Akin to reversing our previous decision in a greedy algorithm
- Encoding these operations is the main goal of a residual graph

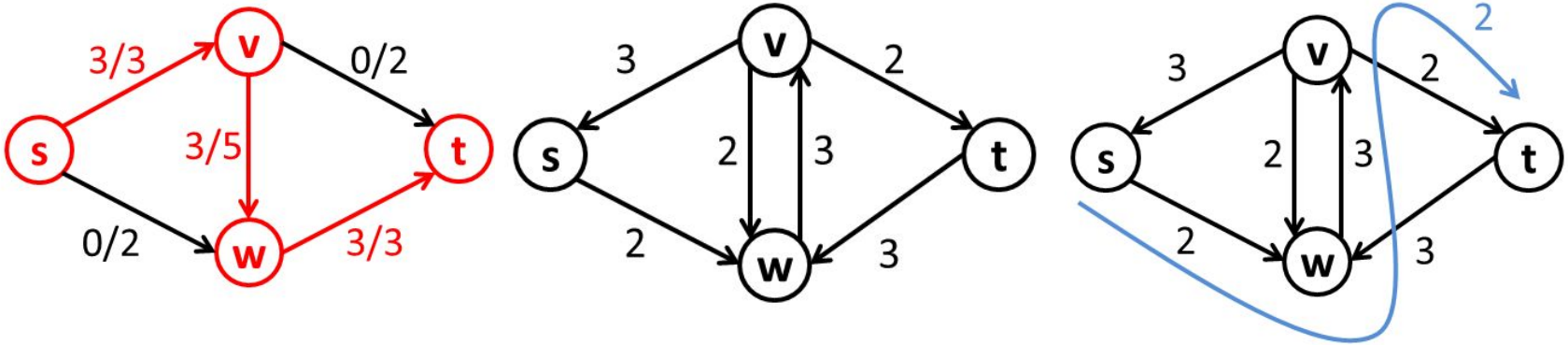




Reference: https://cs.stackexchange.com/a/71505

What is a residual graph?

- Stores how much flow can be pushed through an edge
- Having the reverse edge capacities = current flow represents that the current flow can be “undone!”



Reference: https://cs.stackexchange.com/a/71505

Time complexity of max flow?

Suppose all capacities are integers,
and let $|f|$ be the maximum flow in the graph.

- **Theorem**: If you find augmenting path using BFS or DFS, the Ford-Fulkerson algorithm runs in time $\boldsymbol{O(m|f|)}$.

- Proof:

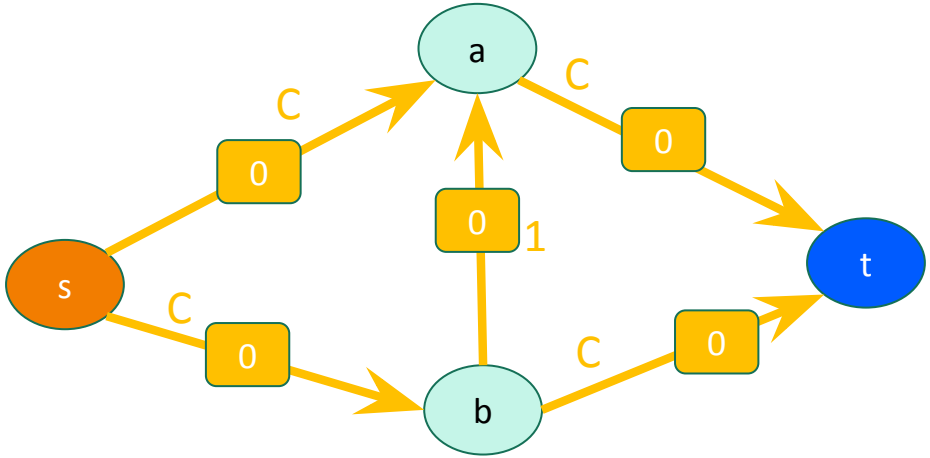
  In each iteration, the flow increases by at least +1.

  $\Rightarrow$ after $\leq |f|$ iterations, the max-flow is found.

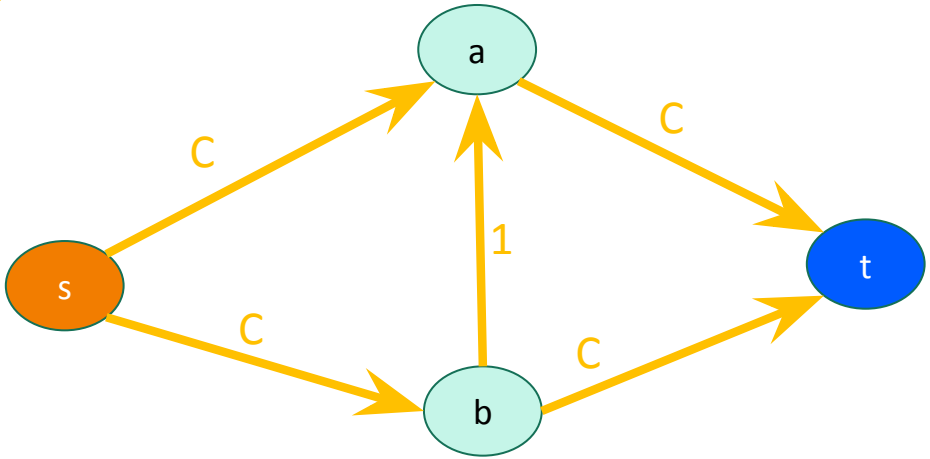
  The theorem follows because each iteration (DFS) takes $O(m)$!

- Caveat: for large $|f|$, this can be very slow.

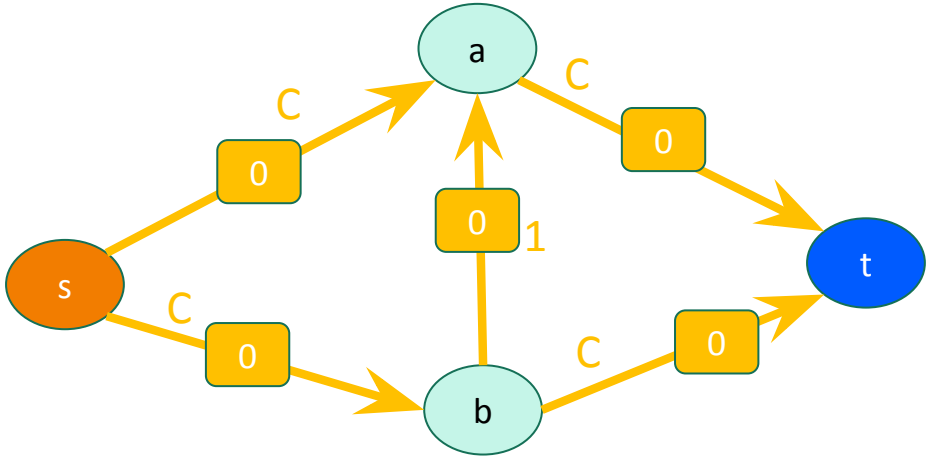
How *not* to pick the augmenting path



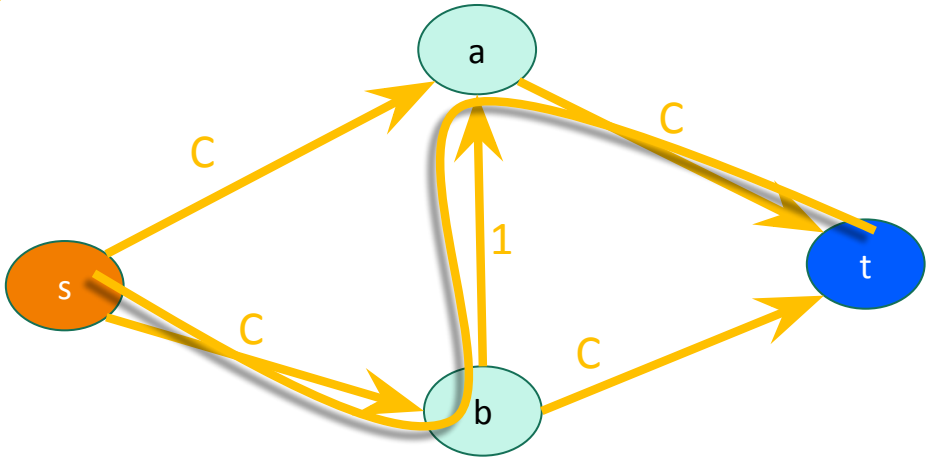
Choose a really big number C.



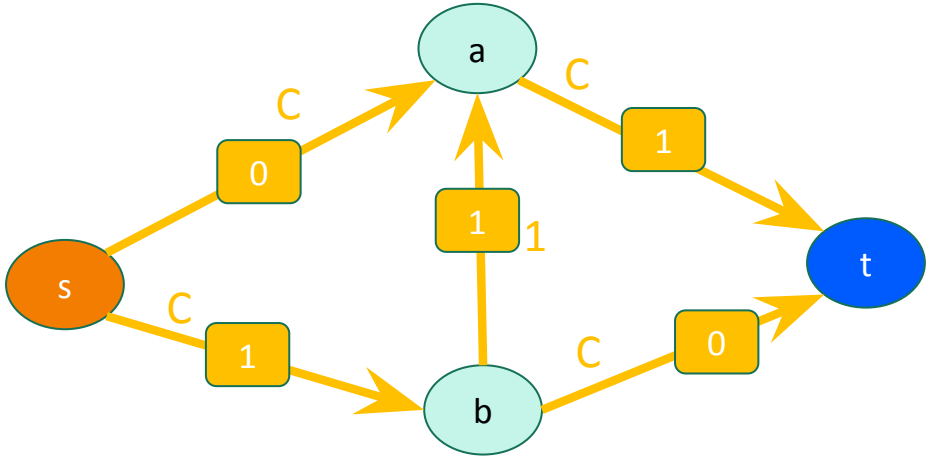
How *not* to pick the augmenting path



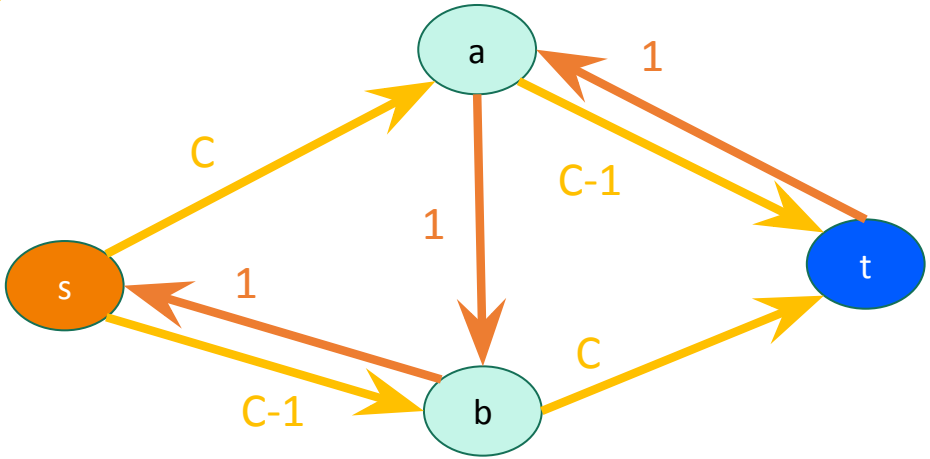
Choose a really big number C.



How *not* to pick the augmenting path

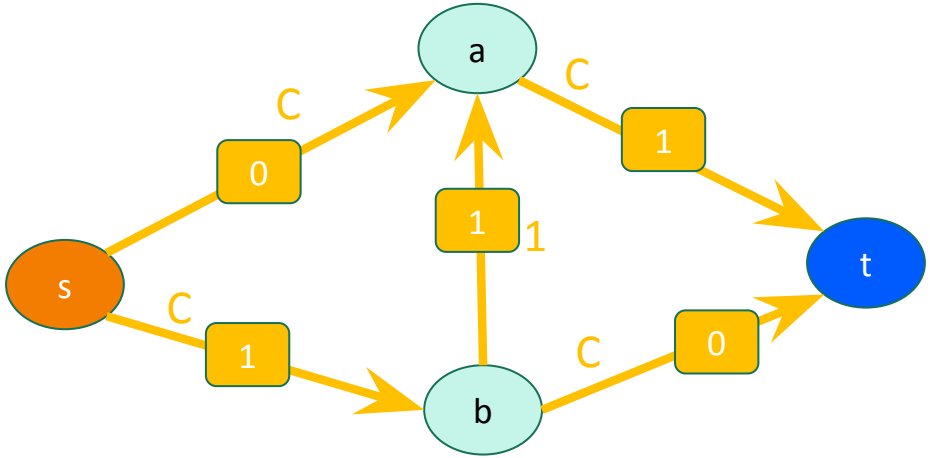


Choose a really big number C.

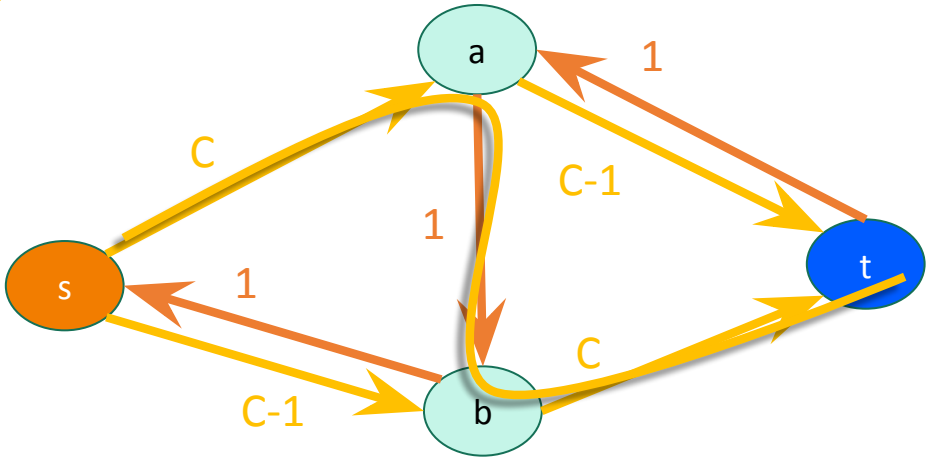


The edge (b,a) disappeared from the residual graph!

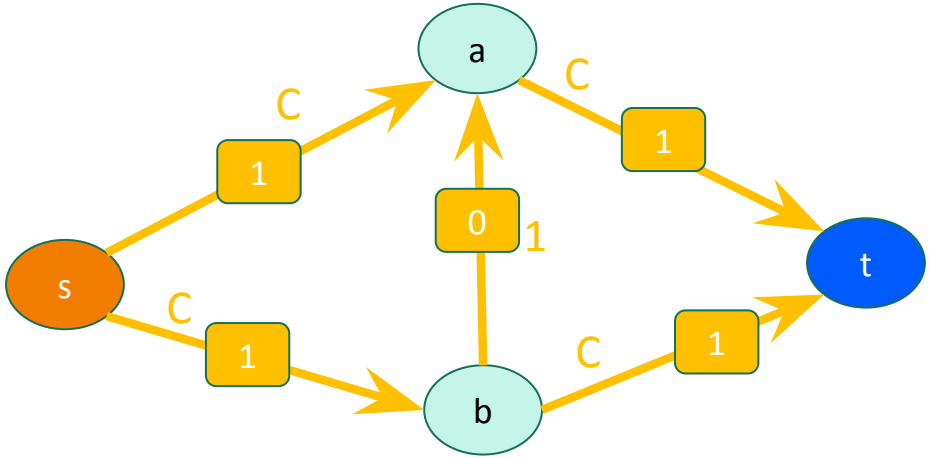
How *not* to pick the augmenting path



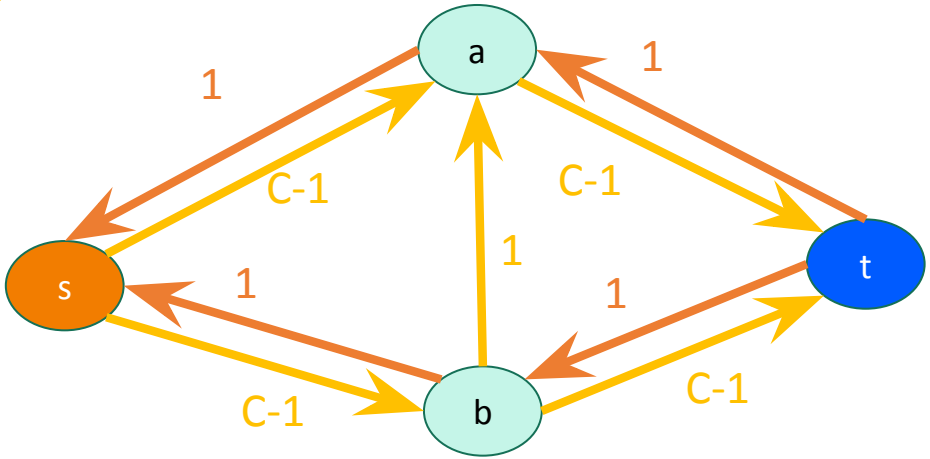
Choose a really big number C.



How *not* to pick the augmenting path

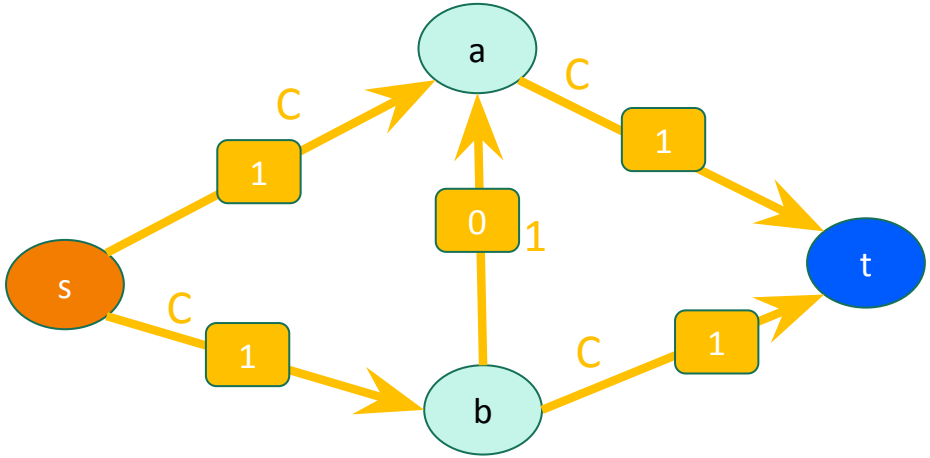


Choose a really big number C.

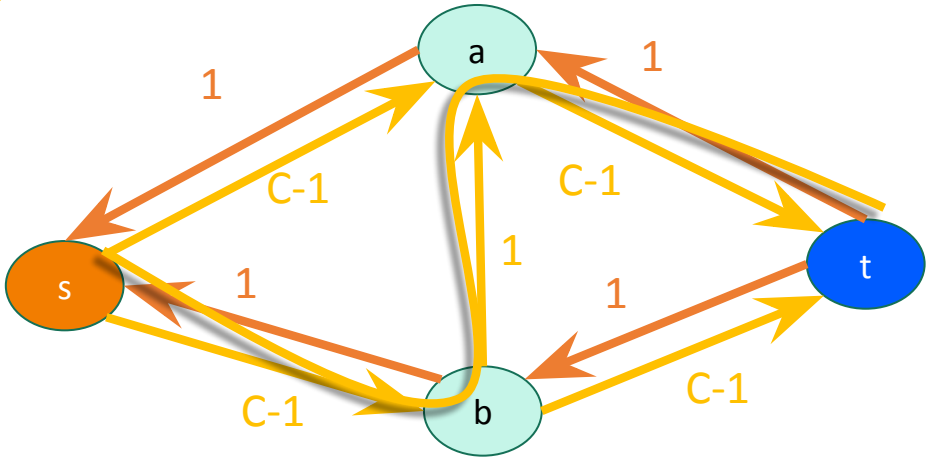


The edge (b,a) re-appeared in the residual graph!

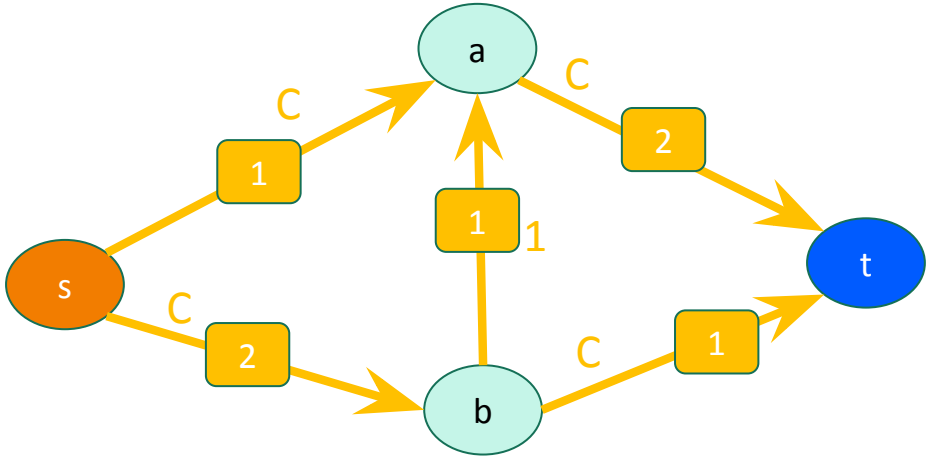
How *not* to pick the augmenting path



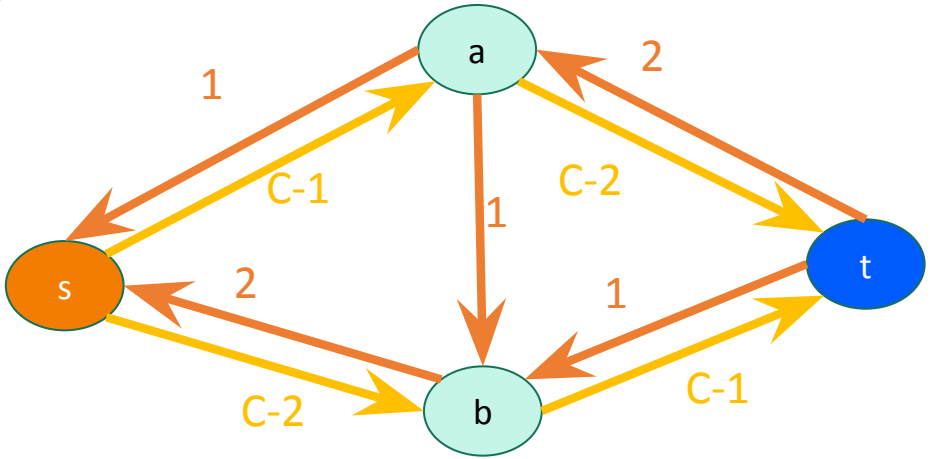
Choose a really big number C.



How *not* to pick the augmenting path

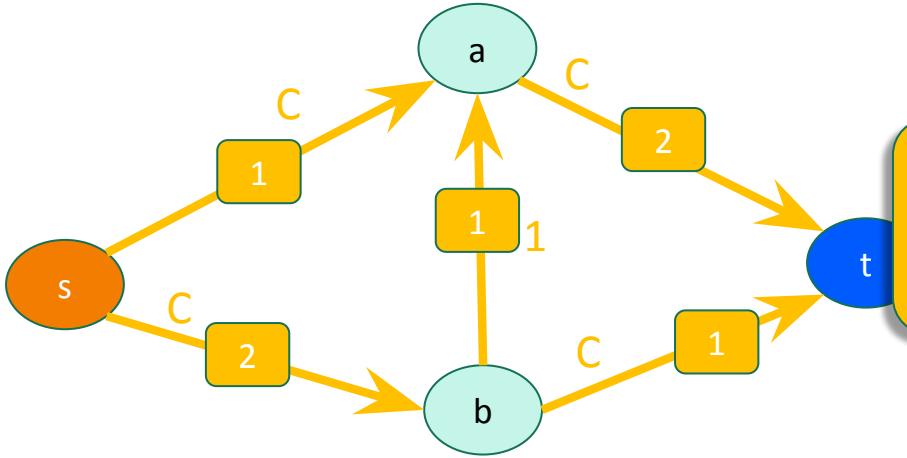


Choose a really big number C.



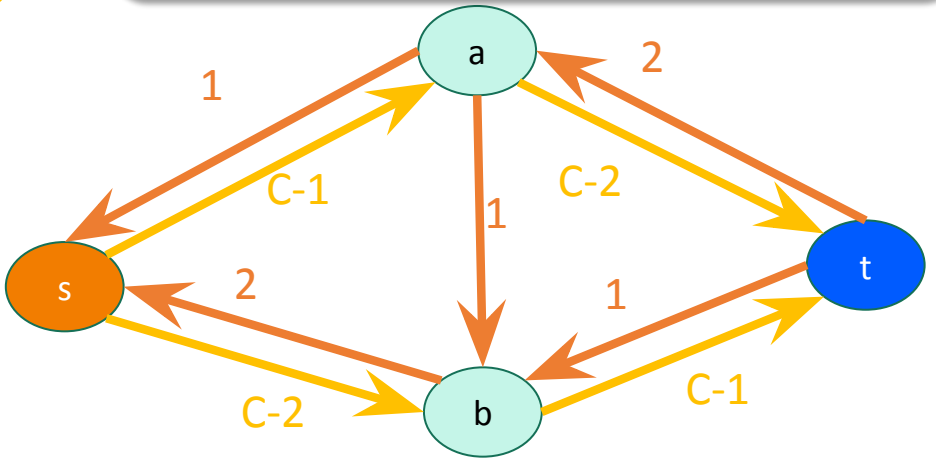
The edge (b,a) disappeared from the residual graph!

How *not* to pick the augmenting path



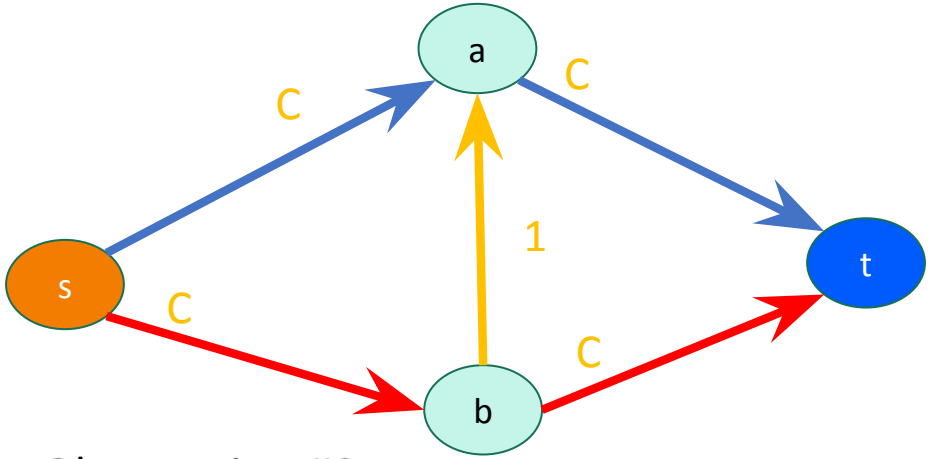
Choose a really big number C.

This will go on for C steps, adding flow along (b,a) and then subtracting it again.



The edge (b,a) disappeared from the residual graph!

So how should we pick the path?



In this example, red/blue paths would have been much better!

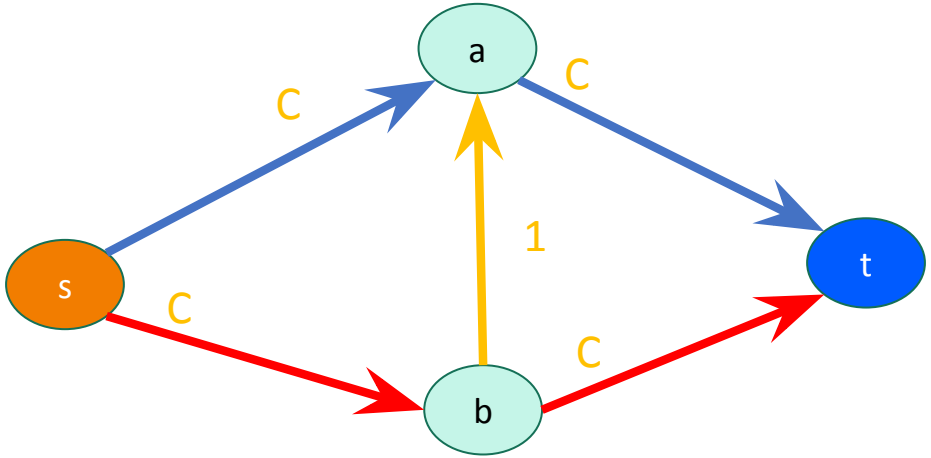
What about in a general graph?

Observation #2:
The old path used 3 edges, now only 2 edges.
Why is this good?

Theorem (see lecture notes for proof):

Shortest path variant of Ford-Fulkerson in time $O(m^2 n)$

So how should we pick the path?



In this example, red/blue paths would have been much better!

What about in a general graph?

Observation #1:

On the old path (s-b-a-t), we could only route 1 unit of flow.

New paths increase flow by C>>1. This is good!

Theorem (see lecture notes for proof):

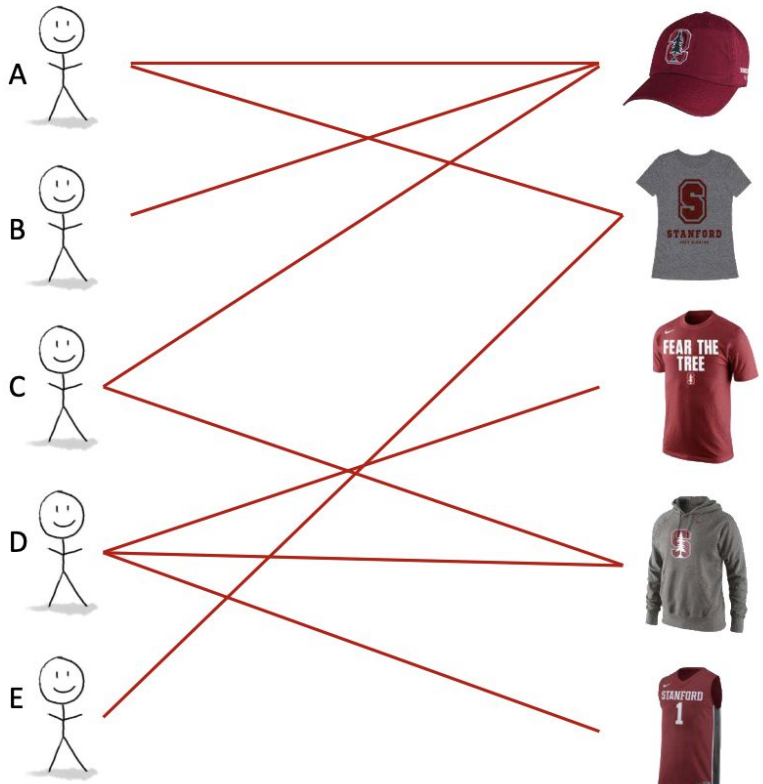
Fattest path variant of Ford-Fulkerson in time $O(m^2 \log n \log|f|)$

Ford Fulkerson additional notes!

- *Shortest path* variant of Ford-Fulkerson runs in time $O(m^2 n)$
- *Fattest path* variant of Ford-Fulkerson runs in time $O(m^2 \log n \log|f|)$
- If all the capacities are integers, then the flow on any edge in any max flow is also integral.
  - When we update flows in Ford-Fulkerson, we're only ever adding or subtracting integers.
  - Since we started with 0 (an integer), everything stays integral.

Maximum matching in bipartite graphs

- Different students only want certain items of Stanford swag (depending on fit, style, etc).
- **How can we make as many students as possible happy?**

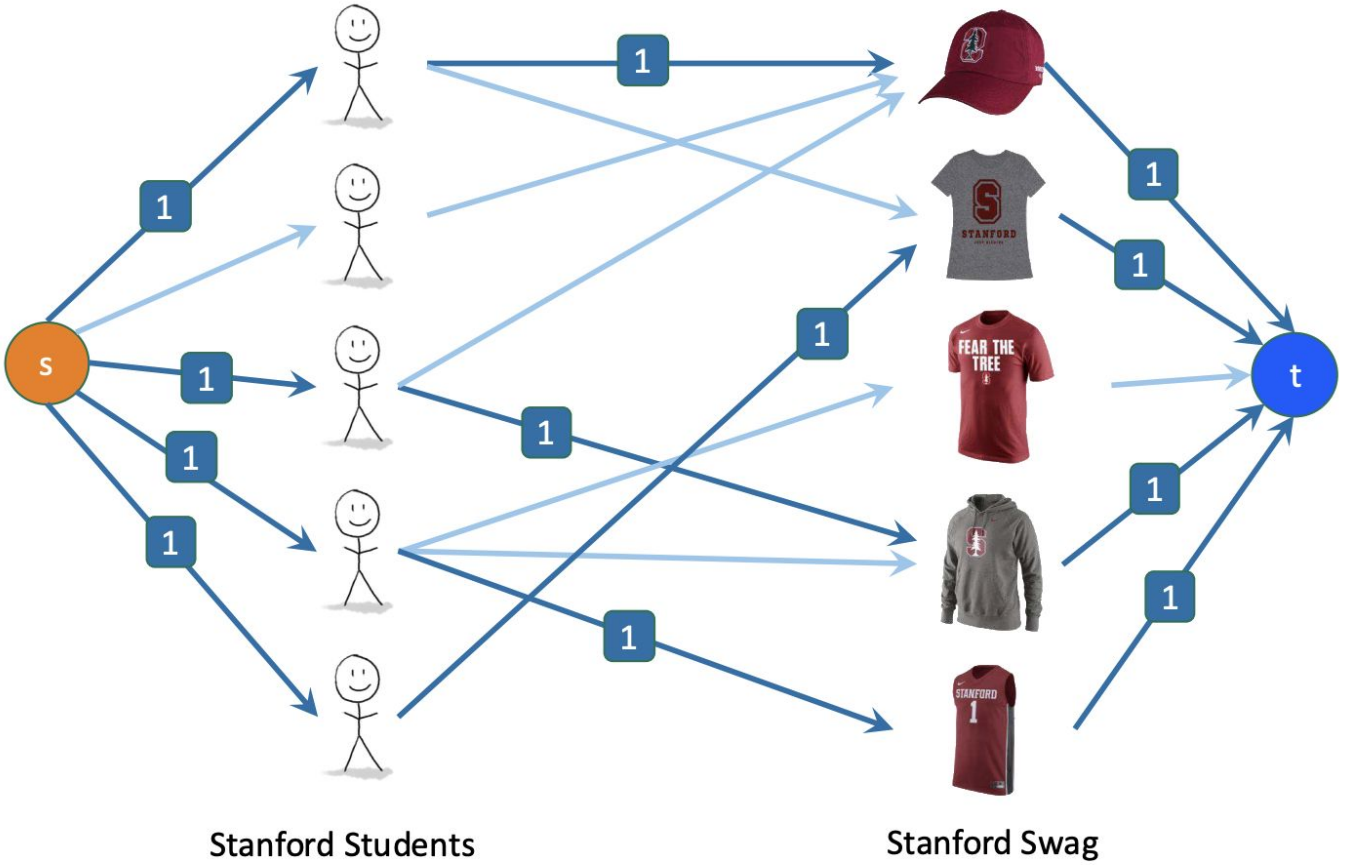


Stanford Students

Stanford Swag

Solution via max flow

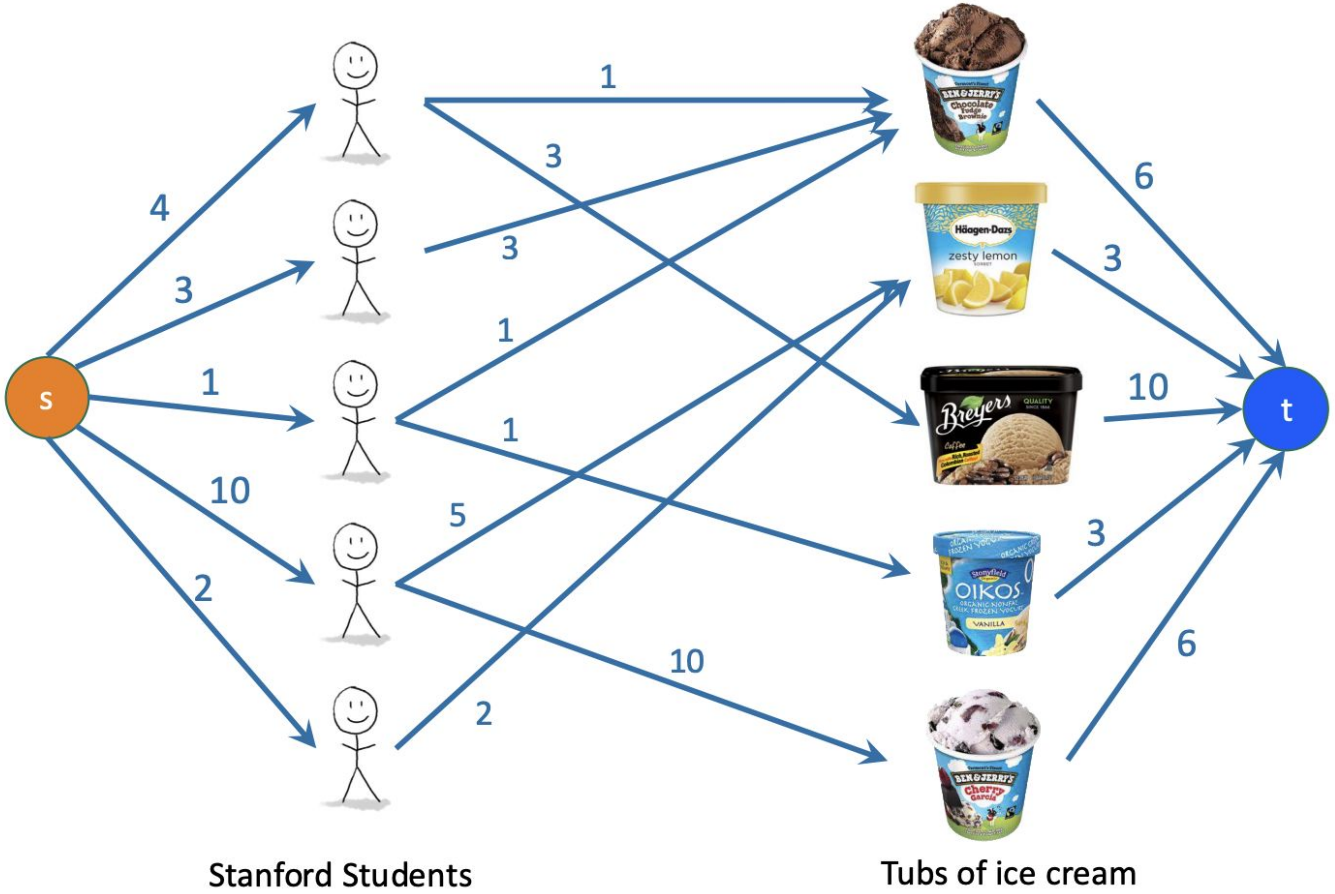
All edges have capacity 1.



A slightly more complicated example:
assignment problems

- One set X
  - Example: Stanford students
- Another set Y
  - Example: tubs of ice cream
- Each x in X can participate in c(x) matches.
  - Student x can only eat 4 scoops of ice cream.
- Each y in Y can only participate in c(y) matches.
  - Tub of ice cream y only has 10 scoops in it.
- Each pair (x,y) can only be matched c(x,y) times.
  - Student x only wants 3 scoops of flavor y
  - Student x' doesn't want any scoops of flavor y'
- **Goal: assign as many matches as possible.**

Solution via max flow



Practice: Project Team Matching

- There are X students and Y project topics
- Each student has the capacity for f(X) projects and g(X, Y) indicates if student X is interested in project Y (1 or 0)
- The projects can take on a max number of students, let’s call this h(Y)
- How do we design a graph that maximizes assignments?