Minimum Spanning Trees and Greedy Algorithms

CSE 417 Winter 24
Lecture 25-26

Greedy Algorithms

What's a greedy algorithm?

An algorithm that builds a solution by:

Considering objects one at a time, in some **order**.

Using a **simple rule** to decide on each object.

Never goes back and changes its mind.

Greedy Algorithms

PROS

Simple

CONS

Rarely correct

Often multiple equally intuitive options

Hard to prove correct

Usually need a fancy "structural result"

Or complicated proof by contradiction

Need to focus on proofs!

Your Takeaways

Greedy algorithms are great *when they work*.

But it's hard to tell when they work – the proofs are subtle.

And you can often invent 2-3 different greedy algorithms; it's rare that 1 works, extremely rare that all would work.

So you have to be EXTREMELY careful.

This will be a crash course in greedy algorithms.

If you have a lot of experience with proofs, I'll be highlighting the general patterns in the proofs.

If you don't, just appreciate the proofs are hard, and promise not to write a greedy algorithm unless someone has proven it correct.

Three Proof Techniques

"Structural result" – the best solution **must** look like this, and the algorithm produces something that looks like this.

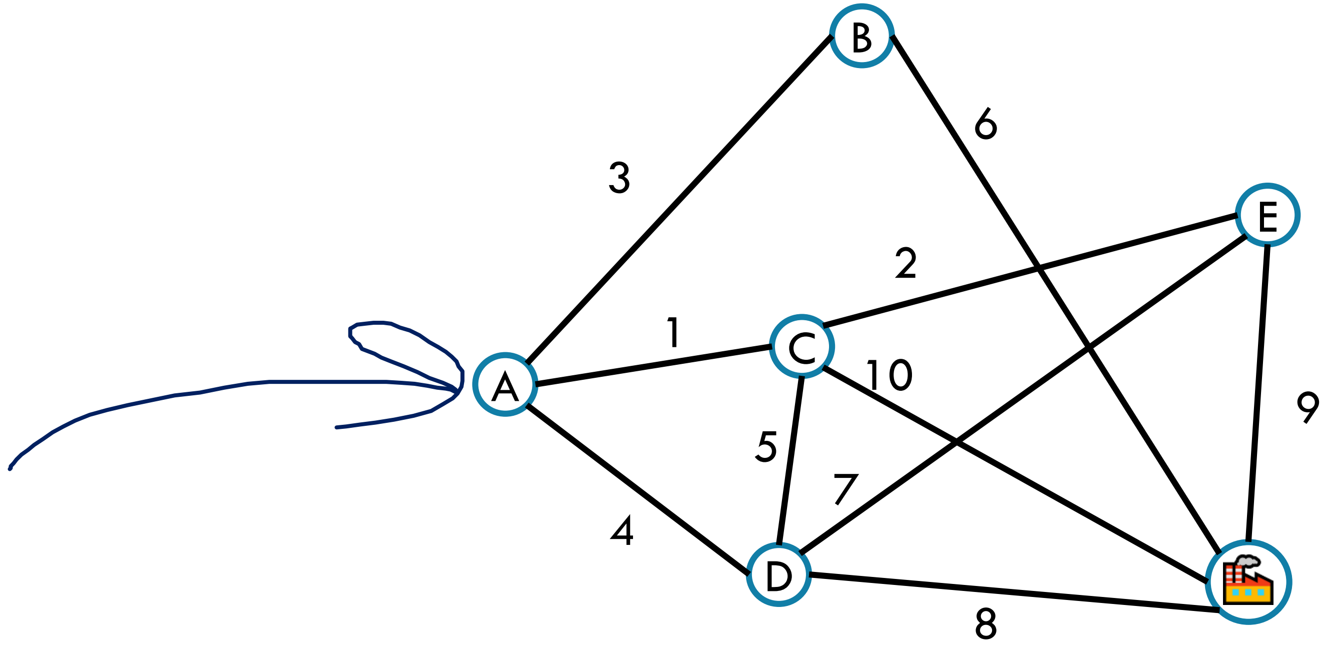
Greedy stays ahead – at every step of the algorithm, the greedy algorithm is at least as good as anything else could be.

Exchange – Contradiction proof, suppose we swapped in an element from the (hypothetical) "better" solution.

Where to start? With some greedy algorithms you've already seen.
Minimum Spanning Trees!

Minimum Spanning Trees

It’s the 1920’s. Your friend at the electric company needs to choose where to build wires to connect all these cities to the plant.



She knows how much it would cost to lay electric wires between any pair of cities, and wants the cheapest way to make sure electricity from the plant to every city.

Minimum Spanning Trees

What do we need? A set of edges such that:

Every vertex touches at least one of the edges. (the edges **span** the graph)

The graph on just those edges is **connected**.

The minimum weight set of edges that meet those conditions.

Minimum Spanning Tree Problem

Given: an undirected, weighted graph G

Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

Greedy MST algorithms

You've seen two algorithms for MSTs

Kruskal's Algorithm:

Order: Sort the edges in increasing weight order

Rule: If connect new vertices (doesn't form a cycle), add the edge.

Prim's Algorithm:

Order: lightest weight edge that adds a new vertex to our current component

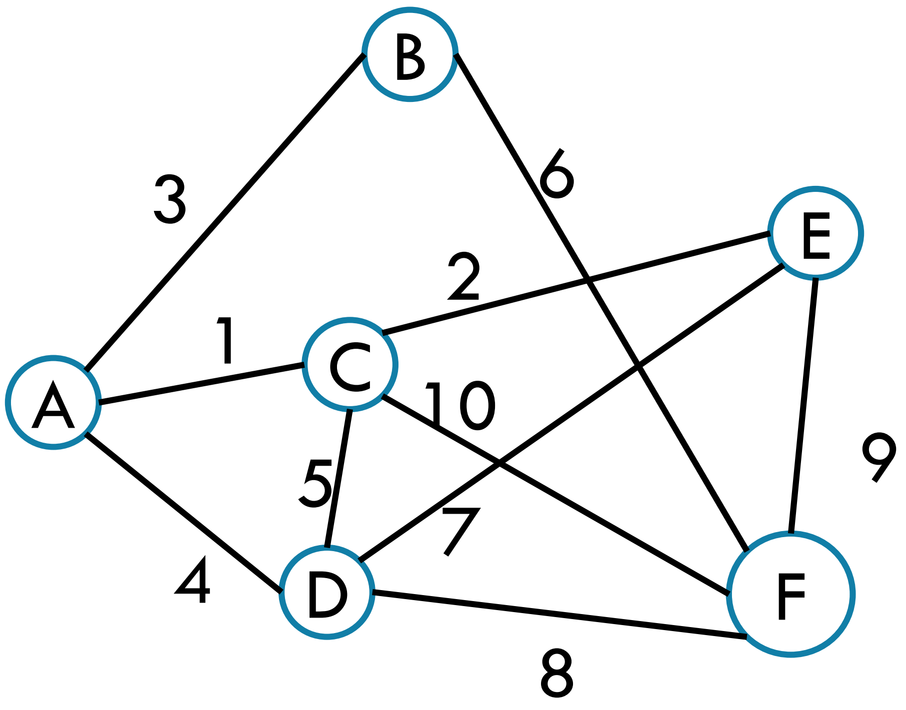
Rule: Just add it!

Kruskal's Algorithm

```
KruskalMST(Graph G)
    initialize each vertex to be its own component
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(u and v are in different components){
            add (u,v) to the MST
            Update u and v to be in the same component
        }
    }
```

Try It Out

```
KruskalMST(Graph G)
   initialize each vertex to be its own component
       sort the edges by weight
       foreach(edge (u, v) in sorted order){
              if(u and v are in different components){
                     add (u,v) to the MST
                     Update u and v to be in the same
component
              }
       }
```

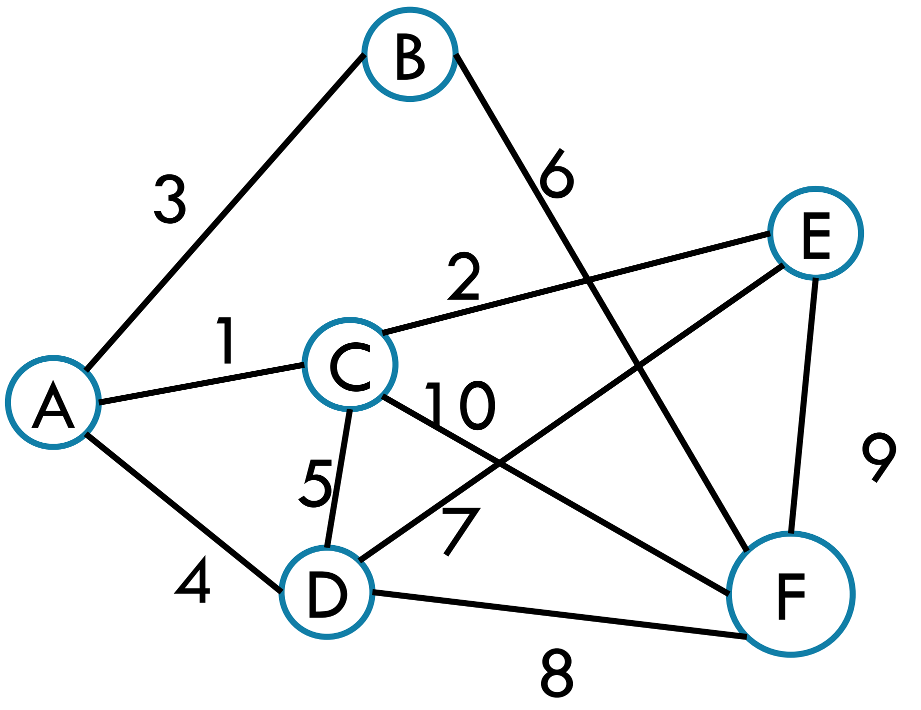


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

Edge (cont.)	Inc?	Reason
(B,F)		
(D,E)		
(D,F)		
(E,F)		
(C,F)		

Try It Out

```
KruskalMST(Graph G)
   initialize each vertex to be its own component
      sort the edges by weight
      foreach(edge (u, v) in sorted order){
           if(u and v are in different components){
                add (u,v) to the MST
                Update u and v to be in the same
component
           }
      }
```



Edge	Include?	Reason
(A,C)	Yes	
(C,E)	Yes	
(A,B)	Yes	
(A,D)	Yes	
(C,D)	No	Cycle A,C,D,A

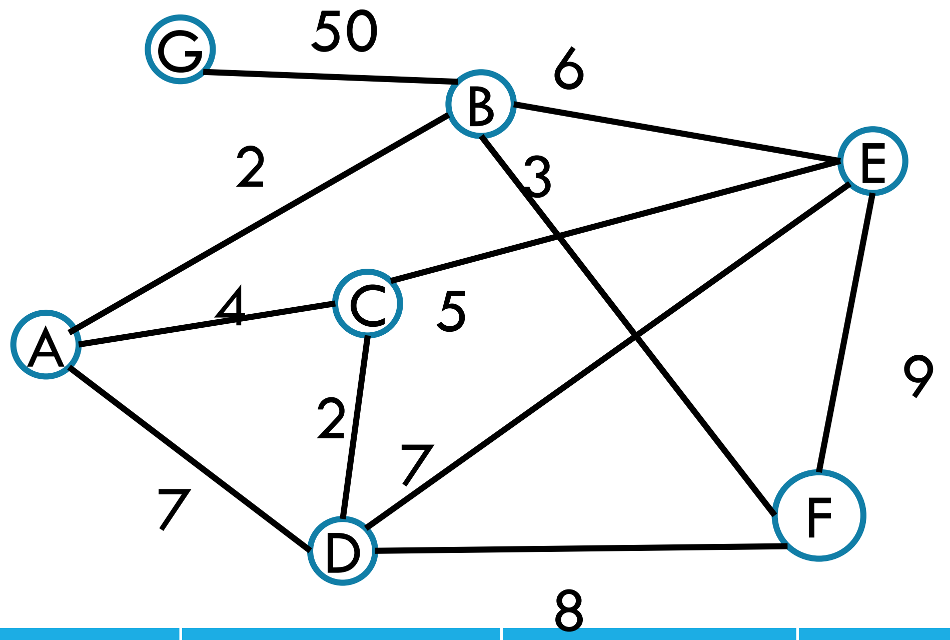
Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Code

```
PrimMST(Graph G)
      initialize costToAdd to ∞
      mark source as costToAdd 0
      mark all vertices unprocessed, mark source as processed
      foreach(edge (source, v) ) {
            v.costToAdd = weight(source,v)
            v.bestEdge = (source,v)
      }
      while(there are unprocessed vertices){
            let u be the cheapest to add unprocessed vertex
            add u.bestEdge to spanning tree
            foreach(edge (u,v) leaving u){
                  if(weight(u,v) < v.costToAdd AND v not processed){
                        v.costToAdd = weight(u,v)
                        v.bestEdge = (u,v)
                  }
            }
      mark u as processed
      }
```

Try it Out

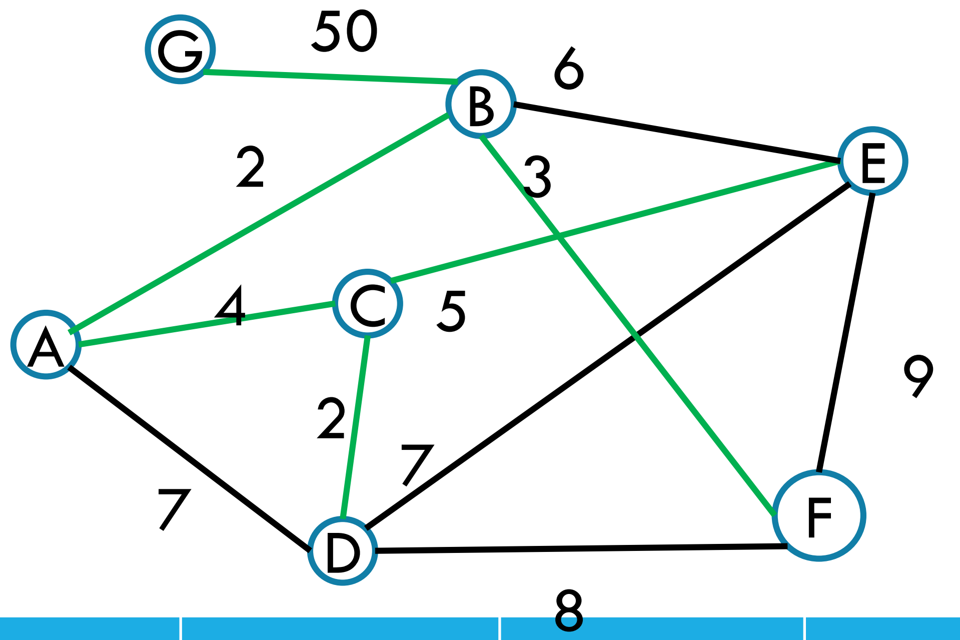
```
PrimMST(Graph G)
   initialize costToAdd to ∞
   mark source as costToAdd 0
   mark all vertices unprocessed
   mark source as processed
   foreach(edge (source, v) ) {
         v.costToAdd = weight(source,v)
         v.bestEdge = (source,v)
   }
   while(there are unprocessed vertices) {
            let u be the cheapest unprocessed vertex
            add u.bestEdge to spanning tree
            foreach(edge (u,v) leaving u){
               if(weight(u,v) < v.costToAdd
                  AND v not processed){
                     v.costToAdd = weight(u,v)
                     v.bestEdge = (u,v)
                 }
            }
            mark u as processed
         }
```



Vertex	costToAdd	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

```
PrimMST(Graph G)
   initialize costToAdd to ∞
   mark source as costToAdd 0
   mark all vertices unprocessed
   mark source as processed
   foreach(edge (source, v) ) {
         v.costToAdd = weight(source,v)
         v.bestEdge = (source,v)
   }
   while(there are unprocessed vertices) {
            let u be the cheapest unprocessed vertex
            add u.bestEdge to spanning tree
            foreach(edge (u,v) leaving u){
               if(weight(u,v) < v.costToAdd
                  AND v not processed){
                     v.costToAdd = weight(u,v)
                     v.bestEdge = (u,v)
                 }
            }
            mark u as processed
         }
```



Vertex	costToAdd	Best Edge	Processed
A	--	--	Yes
B	2	(A,B)	Yes
C	4	(A,C)	Yes
D	~~7~~ 2	~~(A,D)~~(C,D)	Yes
E	~~6~~ 5	~~(B,E)~~(C,E)	Yes
F	3	(B,F)	Yes
G	50	(B,G)	Yes

Correctness

You're already familiar with the algorithms.

We'll use this problem to practice the proof techniques.

We'll do both **structural and exchange**

Structural Proof

For simplicity – assume all edge weights are distinct and that there is only one minimum spanning tree.

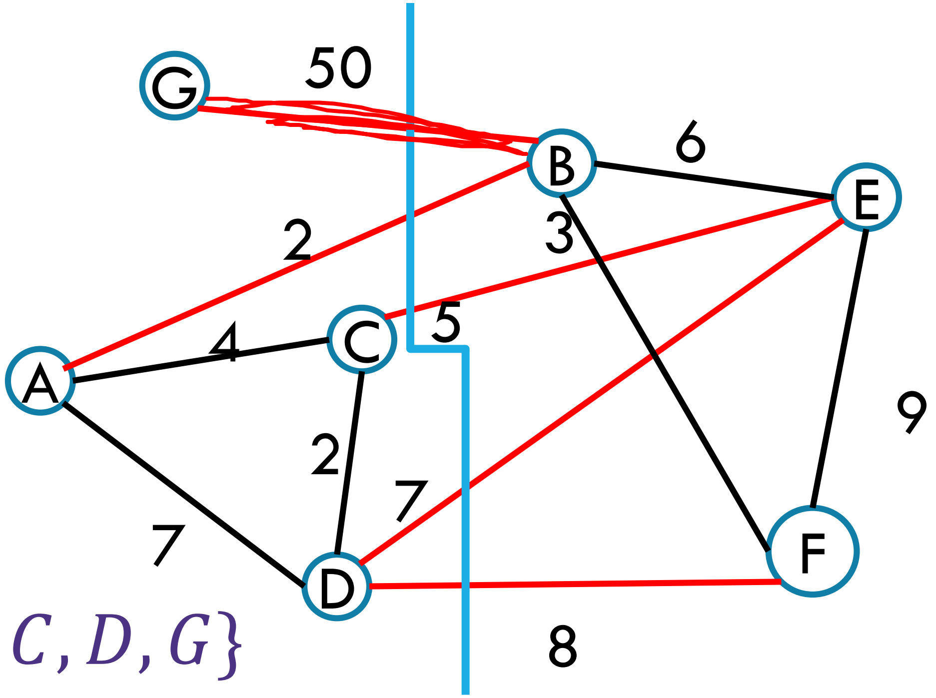
"Structural result" – the best solution **must** look like this, and the algorithm produces something that looks like this.

Example: every spanning tree has $n - 1$ edges.
So we better have our algorithm produce $n - 1$ edges.

Is that enough? No! Lots of different trees (including non minimum ones) have $n - 1$ edges. Need to say which edges are in the tree.

Safe Edge

A "cut" $(S, V \setminus S)$ is a split of the vertices into a subset S and the remaining vertices $V \setminus S$.

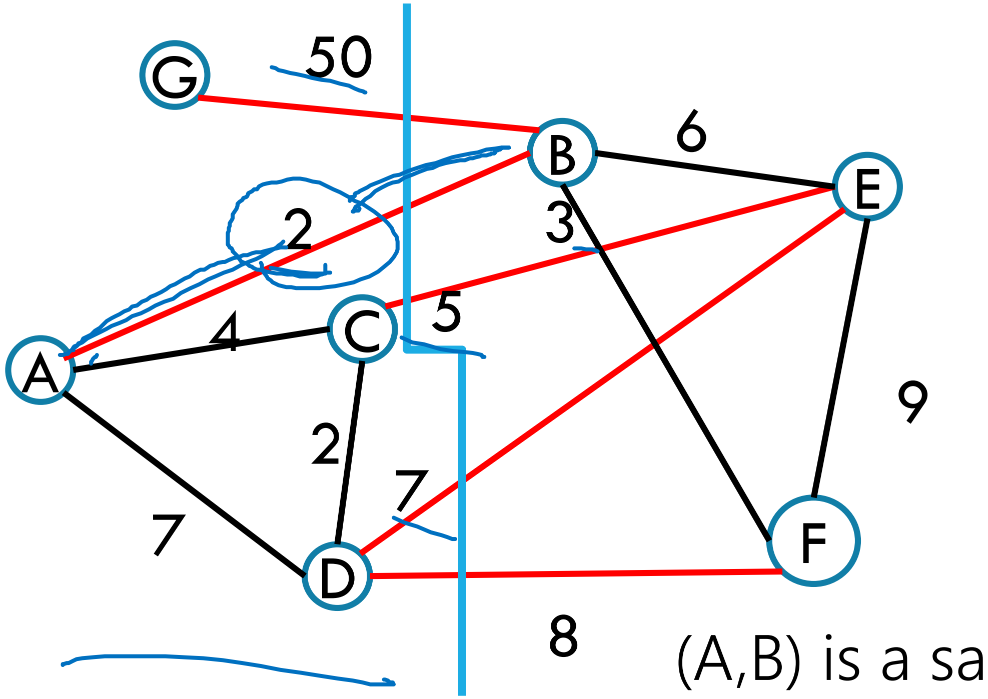


$S = \{A, C, D, G\}$

Edges in red "span" or "cross" the cut (go from S to $V \setminus S$).

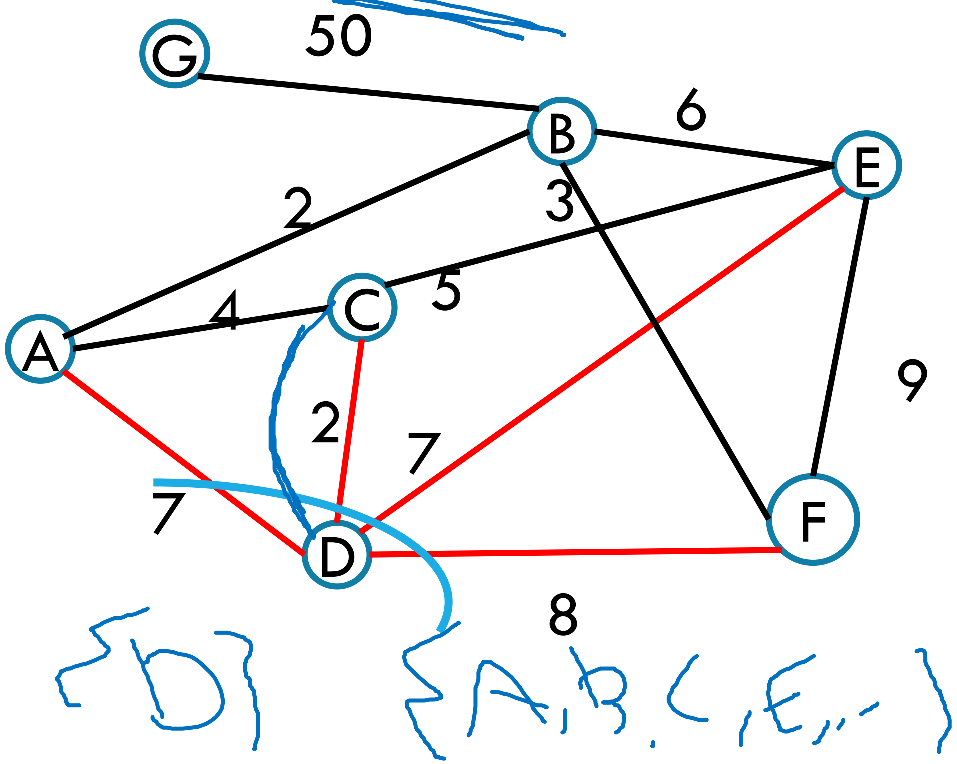
Safe Edge

Call an edge, e, a "safe edge" if there is some cut $(S, V \setminus S)$ where e is the minimum edge spanning that cut



(A,B) is a safe edge

(C,D) is a safe edge

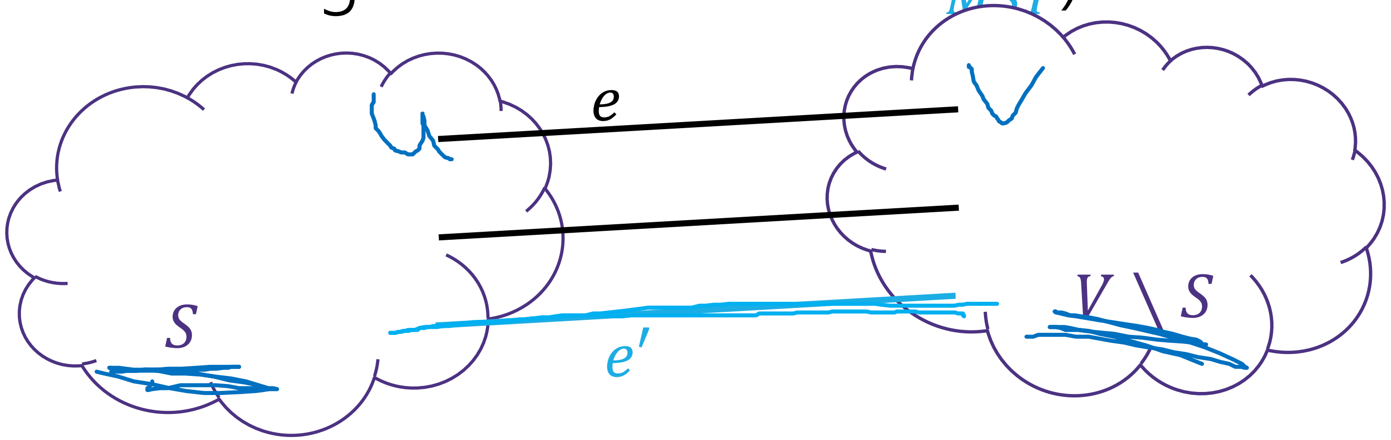


MSTs and Safe Edges

Every safe edge is in the MST.

Proof: Suppose, for the sake of contradiction, that $e = (u, v)$ is a safe edge, but not in the MST.

Let $(S, V \setminus S)$ be a cut where e is the minimum edge spanning $(S, V \setminus S)$. Let T_{MST} be the MST. The MST has (at least one) an edge e' that crosses the cut (since we can get from u to v in T_{MST})

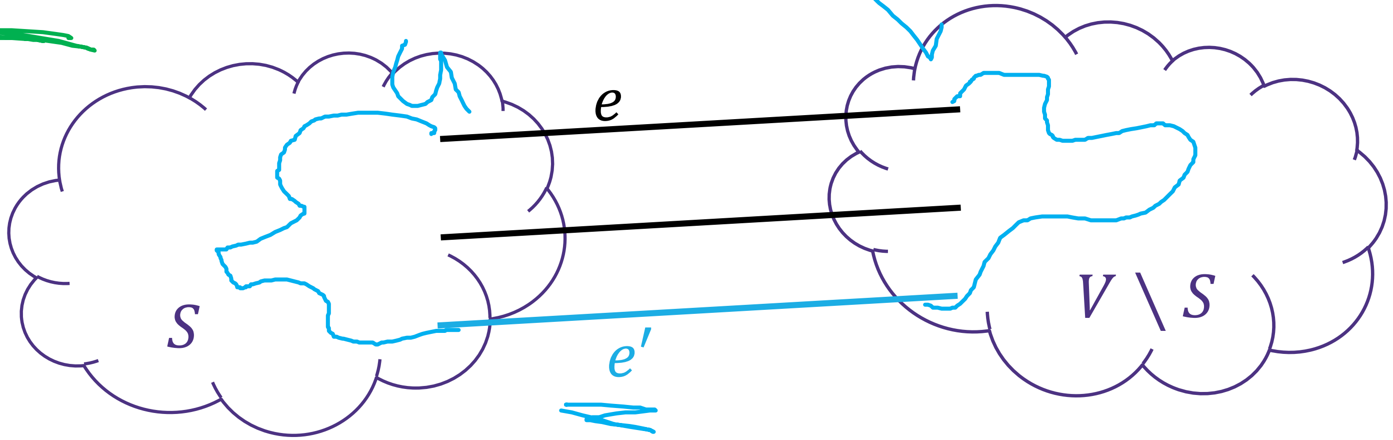


MSTs and Safe Edges

Add $e = (u, v)$ to T_{MST}.

The new graph has a cycle including both e and e', The cycle exists because u and v were connected to each other in T_{MST} (since it was a spanning tree).

Consider T_{NEW}, which is T_{MST} with e added and e' removed.



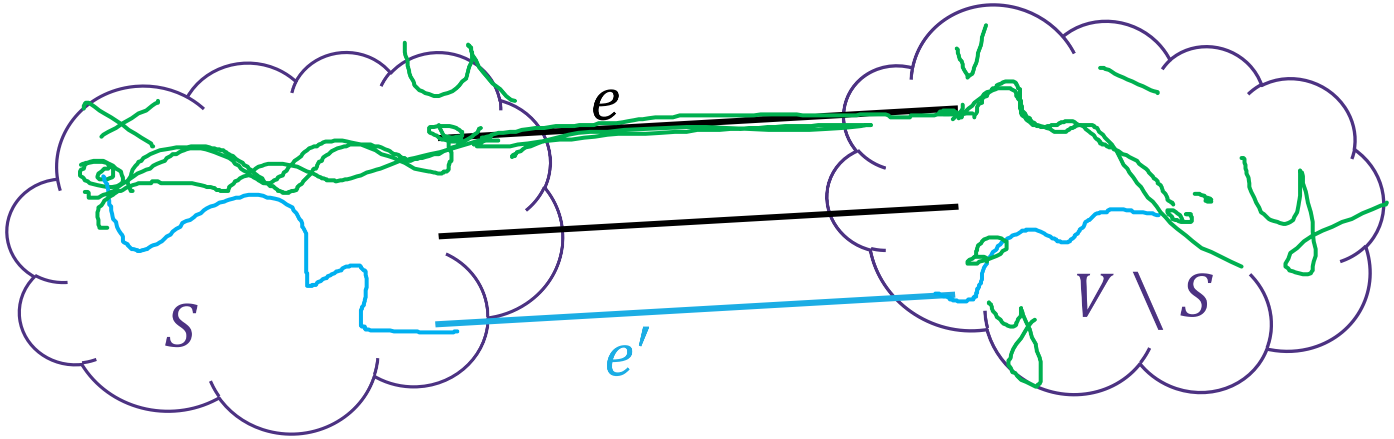
MSTs and Safe Edges

Consider T_{NEW}, which is T_{MST} with e added and e' removed.

T_{NEW} spans: if the path from x to y in T_{MST} didn't use e' it still exists. If it did use e', follow along the path to e', along the cycle through e to the other side.

And it's a tree (it has $n - 1$ edges).

What's its weight? Less than T_{MST}: e was the lightest edge spanning $(S, V \setminus S)$. That's a contradiction! T_{MST} was the minimum spanning tree.



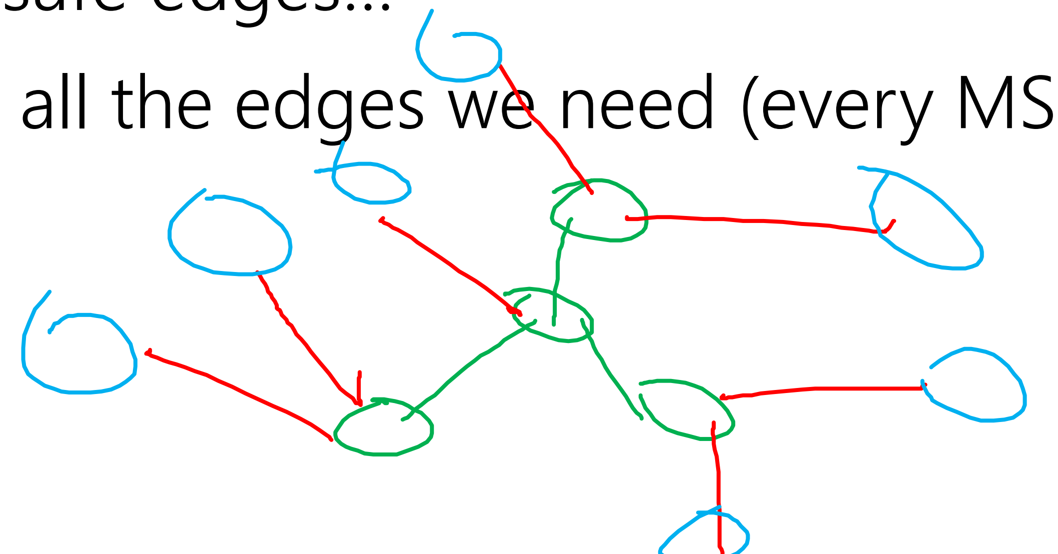
Prim's only adds safe edges

When we add an edge, we add the minimum weight one among those that span from the already connected vertices to the not-yet-connected ones.

That's a cut! And that cut shows the edge we added is safe!

So we only add safe edges…

…and we added all the edges we need (every MST has $n - 1$ edges)



What about Kruskal's?

Exchange argument:

General outline:

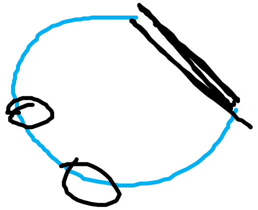
Suppose, you didn't find the best one.

Suppose there's a better MST

Then there's something in the algorithm's solution that doesn't match OPT. (an edge that isn't a safe edge/that's heavier than it needs to be)

Swap (**exchange**) them, and finish the proof (arrive at a contradiction or show that your solution is equal in quality)!

Kruskal's Proof



Suppose, for the sake of contradiction, T_K, the tree found by Kruskal's algorithm isn't a minimum spanning tree. Let T_{MST} be the true minimum spanning tree.

Let $e = (u, v)$ be the lightest edge in T_K but not in T_{MST}. Add e to T_{MST}, and we will create a cycle (because there is a way to get from u to v in T_{MST} by it being a spanning tree).

We claim that e is not the heaviest edge on the cycle. We put e in T_K so it didn't create a cycle there (since we check for cycles before adding it), since Kruskal's adds every edge it can without creating a cycle, every edge in T_{MST} lighter than e is also in T_K. That means there is an edge on the cycle that wasn't in T_K, so it is heavier than e (by the order Kruskal's processes). Delete one such edge, and call the resulting graph T_{NEW}. Observe that T_{NEW} is a spanning tree (it has $n - 1$ edges, and spans all the same vertices T_{MST} did since we deleted an edge from a cycle). But it has less weight than T_{MST} which was supposed to be the MST. That's a contradiction!

Hey…Wait a minute

Those arguments were pretty similar. They both used an "exchange" idea.

The boundaries between the proof principles are a little blurry…

They're meant to be useful for you for thinking about "where to start" with a proof, not be a beautiful taxonomy of exactly what technique is which.

More Greedy Problems

Trip Planning

Your goal is to follow a pre-set route from New York to Los Angeles.

You can drive 500 miles in a day, but you need to make sure you can stop at a hotel every night (all possibilities premarked on your map)

You'd like to stop for the fewest number of nights possible – what should you plan?

Greedy: Go as far as you can every night.

Is greedy optimal?

Or is there some reason to "stop short" that might let you go further the next night?

Trip Planning

Greedy works!

Because “greedy stays ahead”

Let g_i be the hotel you stop at on night i in the greedy algorithm.

Let OPT_i be the hotel you stop at in the optimal plan (the fewest nights plan).

Claim: g_i is always at least as far along as OPT_i.

Intuition: they start at the same point before day 1, and greedy goes as far as possible, so is “ahead” after day 1.

And if greedy is “ahead” at the start of the day, it will continue to be ahead at the end of the day (since it goes as far as possible, and the distance you can go doesn’t depend on where you start).

Therefore it’s always ahead. And so it uses at most the same number of days as all other solutions.

Induction

A formal version of the intuition on the last slide is a proof by induction.

The next two slides contain the formal version if you're curious

Trip Planning

Greedy works!

Because "greedy stays ahead"

Let g_i be the hotel you stop at on night i in the greedy algorithm.

Let OPT_i be the hotel you stop at in the optimal plan (the fewest nights plan).

Claim: g_i is always at least as far along as OPT_i.

Base Case: $i = 1$, OPT and the algorithm choose between the same set of hotels (all at most 500 miles from the start), g_i is the farthest of those by the algorithm definition, so g_i is at least as far as OPT_i.

Trip Planning

Inductive Hypothesis: Suppose through the first k hotels, g_k is farther along than OPT_k.

Inductive Step:

When we select g_{k+1}, we can choose any hotel within 500 miles of g_k, since g_k is at least as far along as OPT_k everything less than 500 miles after OPT_k is also less than 500 miles after g_k. Since we take the farthest along hotel, g_{k+1} is at least as far along as OPT_{k+1}.

More Greedy

Change-Making

Suppose you need to “make change” with the fewest number of coins possible.

Greedy algorithm:

Take the biggest coin less than the change remaining.

Is the greedy algorithm optimal if you have

1 cent coins, 10 cent coins, and 15 cent coins?

Change-Making

The greedy algorithm doesn’t always work!

We made you explain this on the homework problem during the DP section.

But there are times where it does

For “standard” US coinage, the greedy algorithm works

And it also always works if your coins always exactly double in value.

Another reason to be very careful with greedy algorithms!

Also a good example of how you can sometimes avoid greedy if you can’t figure out a proof – maybe there’s a way to write a DP instead!

Interval Scheduling

You have a single processor, and a set of jobs with fixed start and end times.

Your goal is to maximize the number of jobs you can process.

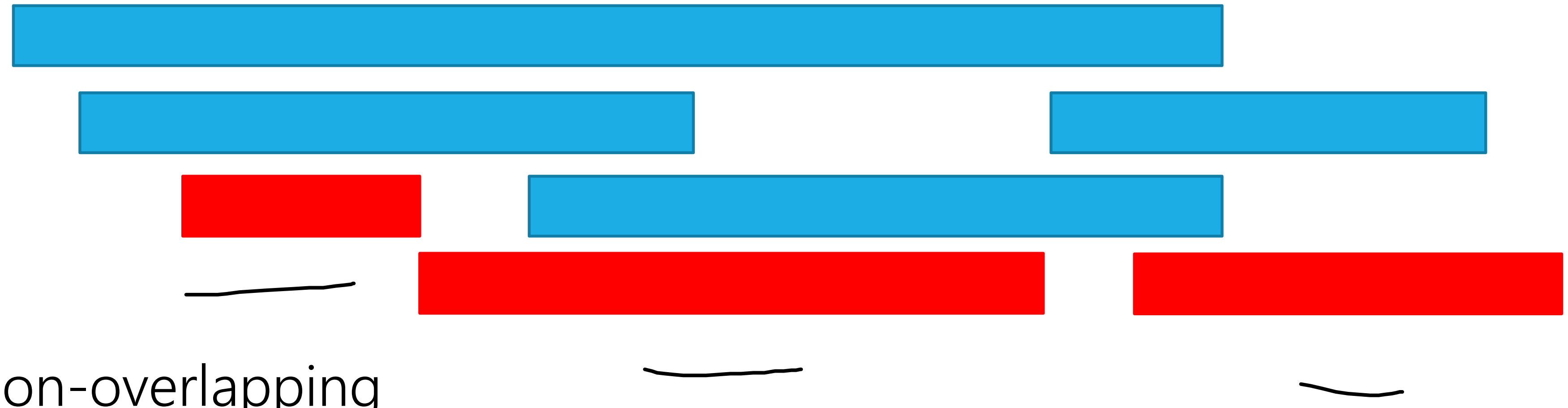
I.e. choose the maximum number of non-overlapping intervals.

Interval Scheduling

You have a single processor, and a set of jobs with fixed start and end times.

Your goal is to maximize the number of jobs you can process.

I.e. choose the maximum number of non-overlapping intervals.



3 non-overlapping intervals

Interval Scheduling

You have a single processor, and a set of jobs with fixed start and end times.

Your goal is to maximize the number of jobs you can process.

I.e. choose the maximum number of non-overlapping intervals.

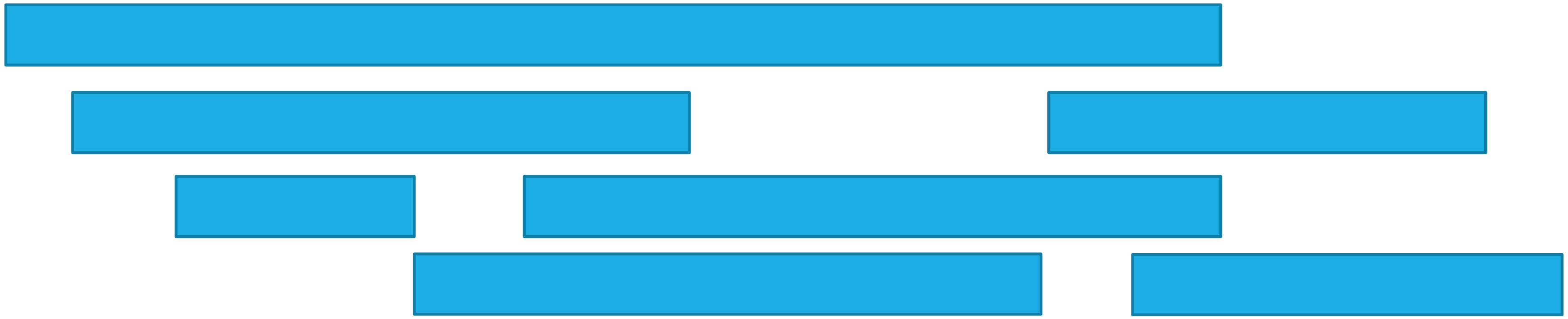
3 other non-overlapping intervals

Interval Scheduling

You have a single processor, and a set of jobs with fixed start and end times.

Your goal is to maximize the number of jobs you can process.

I.e. choose the maximum number of non-overlapping intervals.



OPT is 3 – there is no way to have 4 non-overlapping intervals; both the red and purple solutions are equally good.

Greedy Ideas

To specify a greedy algorithm, we need to:

Order the elements (intervals)

Choose a rule for deciding whether to add.
Rule: Add interval as long as it doesn’t overlap with those we’ve already selected.

What ordering should we use?

Think of **at least two** orderings you think might work.

Greedy Algorithm

Some possibilities

Earliest end time (add if no overlap with previous selected)

Latest end time

Earliest start time

Latest start time

Shortest interval

Fewest overlaps (with remaining intervals)

Greedy

That list slide is the real difficulty with greedy algorithms.

All of those look at least somewhat plausible at first glance.

With MSTs that was fine – those ideas all worked!

It’s not fine here.

They don’t all work.

As a first step – try to find counter-examples to narrow down

Greedy Algorithm

Earliest end time

Latest end time

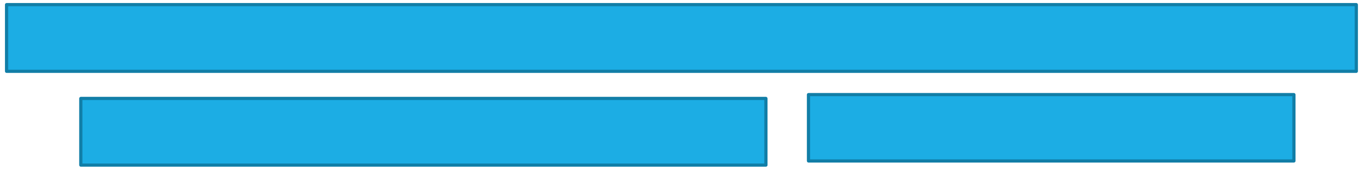
Earliest start time

Latest start time

Shortest interval

Fewest overlaps (with remaining intervals)

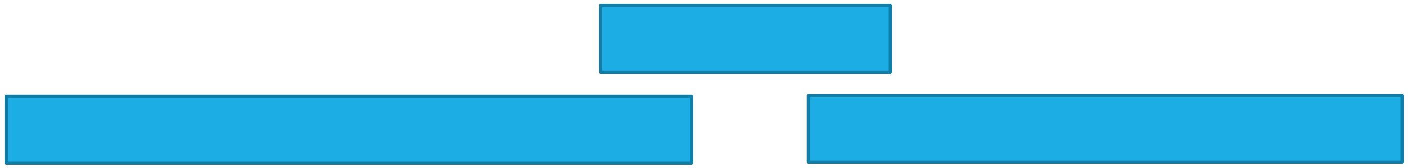
Take Earliest Start Time – Counter Example



Take Earliest Start Time – Counter Example

Taking the one with the earliest start time doesn't give us the best answer.

Shortest Interval



Shortest Interval

Taking the shortest interval first doesn't give us the best answer

Greedy Algorithm

Earliest end time

Latest end time ✖

Earliest start time ✖

Latest start time

Shortest interval ✖

Fewest overlaps (with remaining intervals)

Earliest End Time

Intuition: If u has the earliest end time, and u overlaps with v and w then v and w also overlap.

Why?

If u and v overlap, then both are "active" at the instant before u ends (otherwise v would have an earlier end time).

Otherwise v would have an earlier end time than u! By the same reasoning, w is also "active" the instant before u ends. So v and w also overlap with each other.

Earliest End Time

Can you prove it correct?

Do you want to use

Structural Result

Exchange Argument

Greedy Stays Ahead

Exchange Argument

Let $A = a_1, a_2, \dots, a_k$ be the set of intervals selected by the greedy algorithm, ordered by endtime

OPT= $o_1, o_2, \dots, o_\ell$ be the maximum set of intervals, ordered by endtime.

Our goal will be to “exchange” to show A has at least as many elements as OPT.

Let a_i, o_i be the first two elements where a_i and o_i aren’t the same. Since a_{i-1} and o_{i-1} are the same, neither a_i nor o_i overlaps with any of $o_1, \dots, o_{i-1}$. And by the greedy choice, a_i ends no later than o_i so a_i doesn’t overlap with o_{i+1}. So we can exchange a_i into OPT, replacing o_i and still have OPT be valid.

Exchange Argument

Repeat this argument until we have changed OPT into A.

Can OPT have more elements than A?

No! After repeating the argument, we could change every element of OPT to A. If OPT had another element, it wouldn't overlap with anything in OPT, and therefore can't overlap with anything in A after all the swaps. But then the greedy algorithm would have added it to A.

So A has the same number of elements as OPT does, and we really found an optimal

Greedy Stays Ahead

Let $A = a_1, a_2, \dots, a_k$ be the set of intervals selected by the greedy algorithm, ordered by endtime

OPT= $o_1, o_2, \dots, o_\ell$ be the maximum set of intervals, ordered by endtime.

Our goal will be to show that for every i, a_i ends no later than o_i.

Proof by induction:

Base case: a_1 has the earliest end time of any interval (since there are no other intervals in the set when we consider a_1 we always include it), thus a_1 ends no later then o_1.

Greedy Stays Ahead

Inductive Hypothesis: Suppose for all $i \le k$, a_i ends no later than o_i.

IS: Since (by IH) a_k ends no later than o_k, greedy has access to everything that doesn't overlap with a_k. Since a_k ends no later than o_k, that includes o_{k+1}. Since we take the first one that doesn't overlap, a_{k+1} will also end before o_{k+1}.

Therefore a_{k+1} ends no later than o_{k+1}

Wrapping Up: Since every a_i ends no later than o_i, the last interval greedy selects (a_n) is no later than o_n. There cannot be an o_{n+1}, as if it didn't overlap with o_n it also wouldn't overlap with a_n and would have been added by greedy.

Greedy Algorithm

Earliest end time ✓

Latest end time ✗

Earliest start time ✗

Latest start time

Shortest interval ✗

Fewest overlaps (with remaining intervals)

Fewest Overlaps counter-example

The top middle item will be selected first, eliminating the chance of getting the 4 intervals in OPT.

Other Greedy Algorithms

It turns out latest start time also works.

Latest start time is actually the same as earliest end time (imagine "reflecting" all the jobs along the time axis – the one with the earliest end time ends up having the last start time).

What about fewest overlaps?

Doesn't work!

Greedy Algorithm

Earliest end time ✓

Latest end time ✗

Earliest start time ✗

Latest start time ✓

Shortest interval ✗

Fewest overlaps (with remaining intervals) ✗

Summary

Greedy algorithms

You'll probably have 2 (or 3…or 6) ideas for greedy algorithms. Check some simple examples before you implement!
Greedy algorithms rarely work.

When they work AND you can prove they work, they're great!

Proofs are often tricky

Structural results are the hardest to come up with, but the most versatile.

Greedy stays ahead usually use induction

Exchange start with the **first** difference between greedy and optimal.