Minimum Spanning Trees and Greedy Algorithms

CSE 421 Fall 2022
Lecture 6

Announcements

Remember that we always care about efficiency of algorithms. Unless we explicitly say otherwise:

A slightly-less-efficient algorithm will receive slightly-less points.

A significantly-less-efficient algorithm (like an exponential one) will receive significantly-less points.

We don’t care about constant factors.

This Week

Another abrupt change of topic.

In fact, the whole course is a sequence of seemingly-abrupt topic changes…

We're giving you a list of tools – a list of common ways of thinking when approaching a new problem.

Think of each week as a new tool in your toolbox.

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.

Greedily do what looks best for you right here, right now.

Greedy Algorithms

PROS

Simple

Need to focus on proofs!

CONS

Rarely correct

Often multiple equally intuitive options

Hard to prove correct

- Usually need a fancy “structural result”
- Or complicated proof by contradiction
- Or subtle proof by induction

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 gives you the best answer, extremely rare that all would.

So you have to be EXTREMELY careful.

But they are very often useful when you need an answer that is very good, but not optimal (more on Friday).

Three Common Proof Styles

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

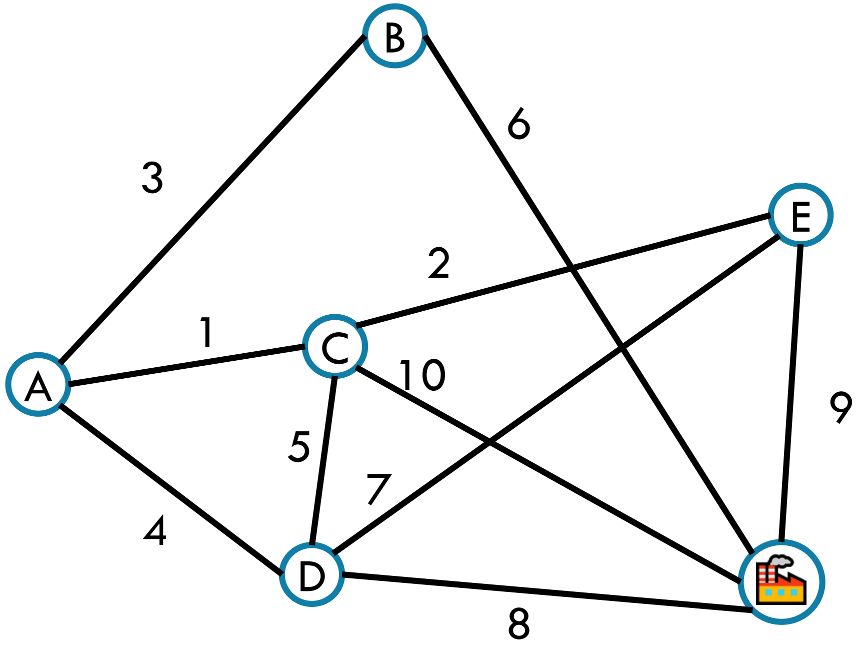
Greedy stays ahead – at every step of the algorithm, the greedy algorithm is at least as good as anything else could be.

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

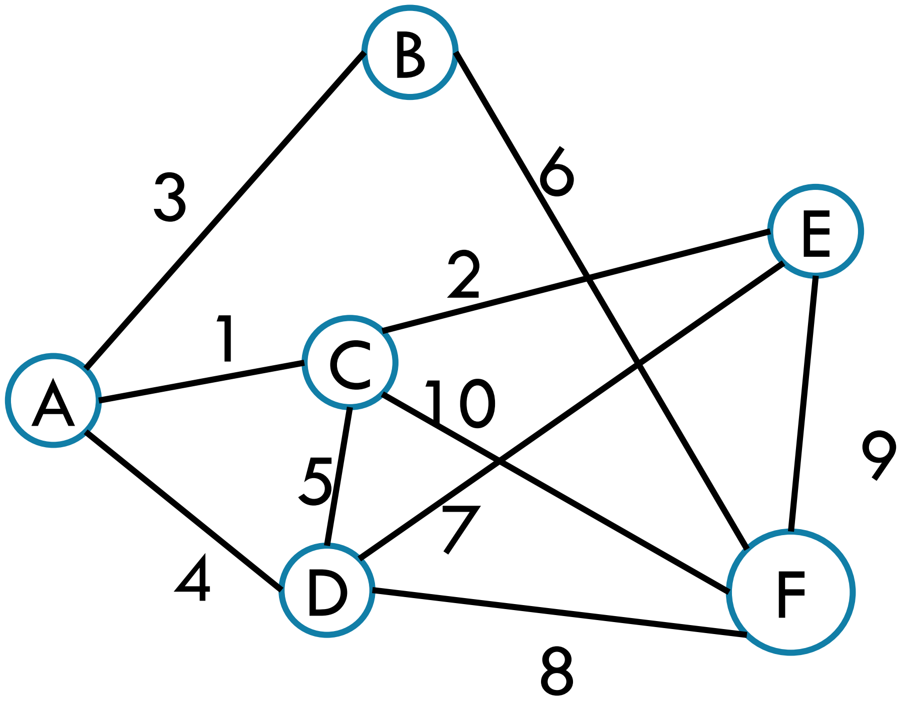
Rule: Just add it!

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
                    }
          }
```

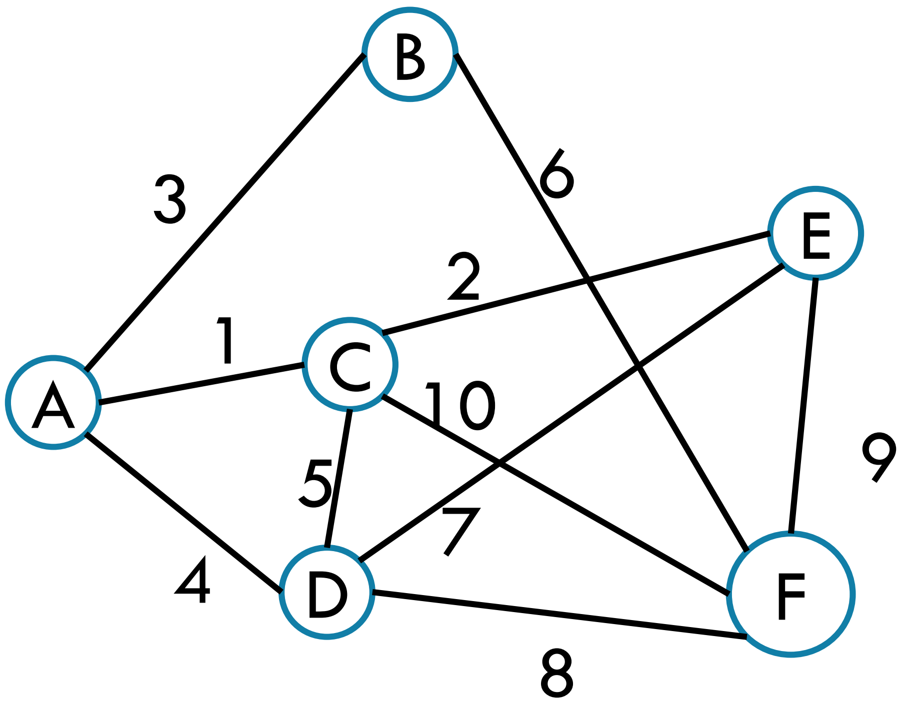


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

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

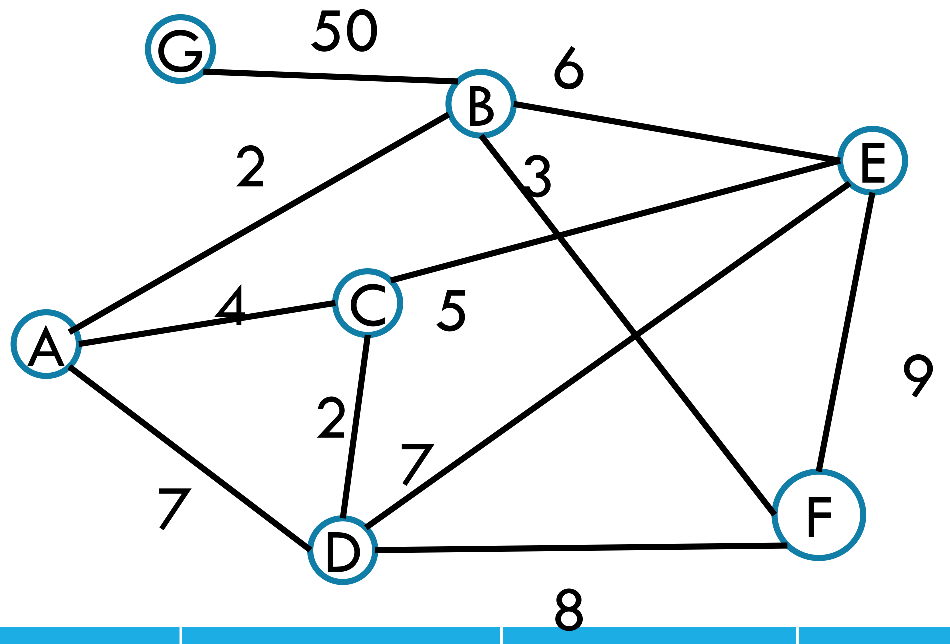
Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Prim's Algorithm

```
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

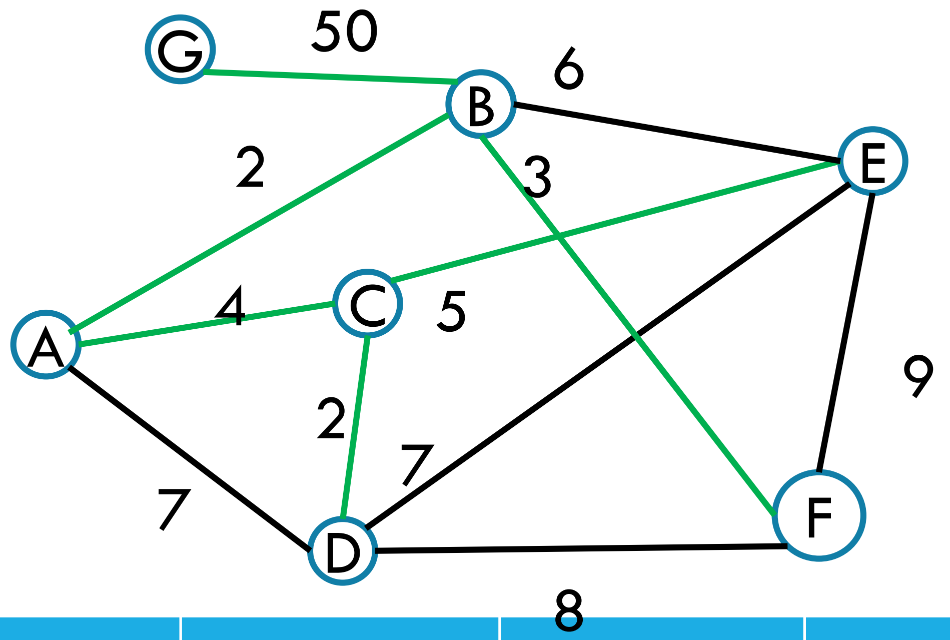
```
PrimMST(Graph G)
   initialize costToAdd to ∞
   mark source as costToAdd 0
   mark all vertices unprocessed
   mark source as processed
   foreach(edge (source, v) ) {
         v.costToAdd = weight(source,v)
         v.bestEdge = (source,v)
   }
   while(there are unprocessed vertices) {
            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.

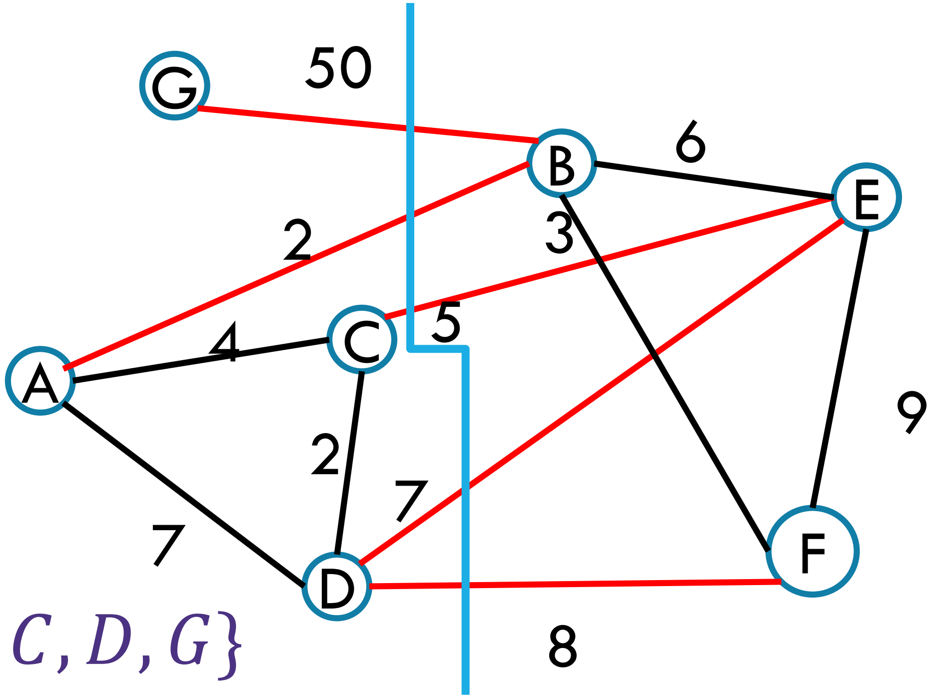
"Structural result" – the best solution **must** look like this, and the algorithm produces something that looks like this.

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 $V \setminus S$.

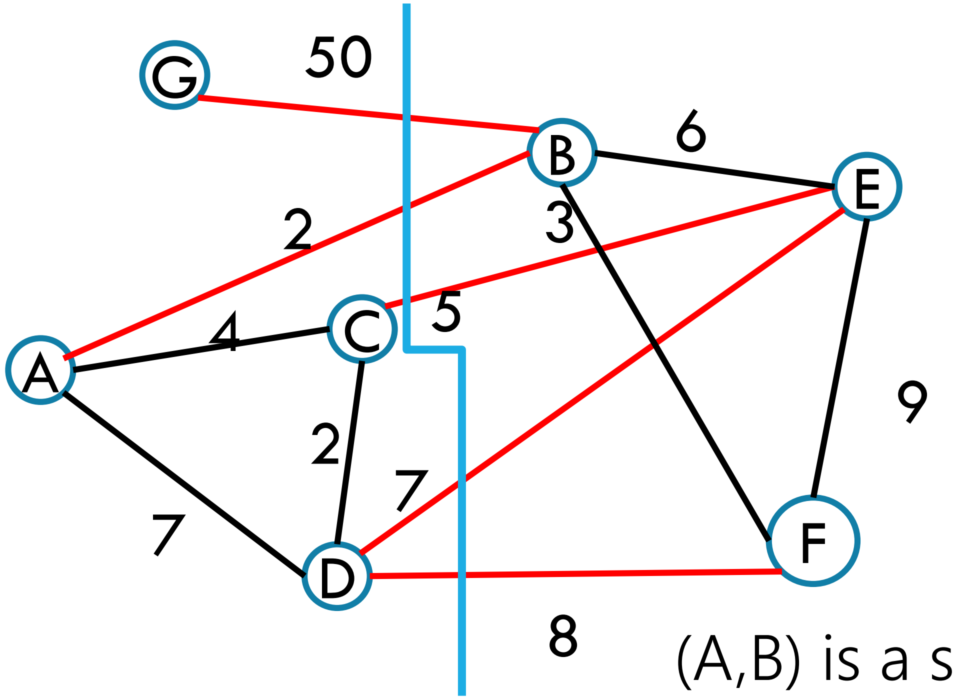


$S = \{A, C, D, G\}$

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

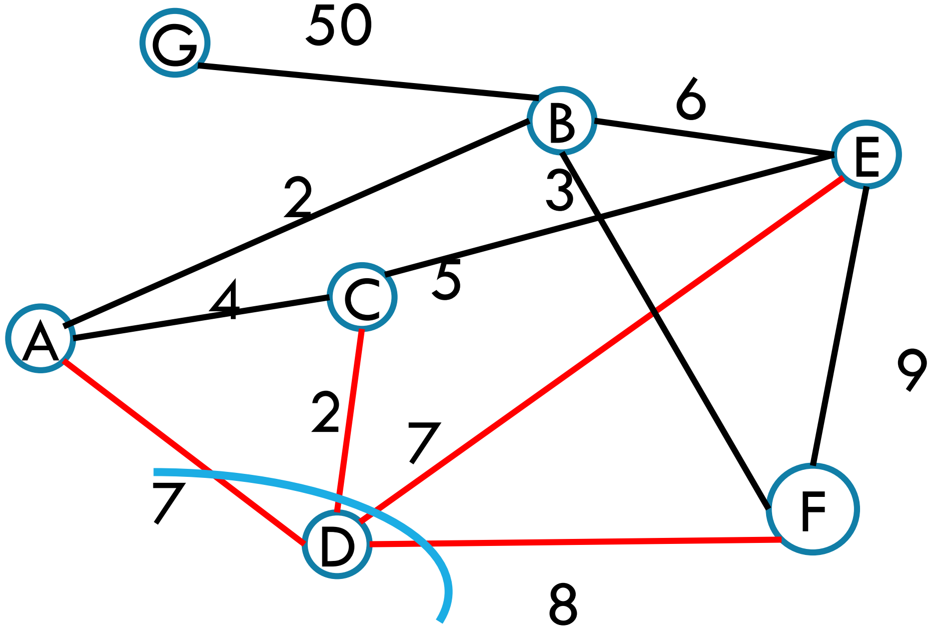
Safe Edge

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

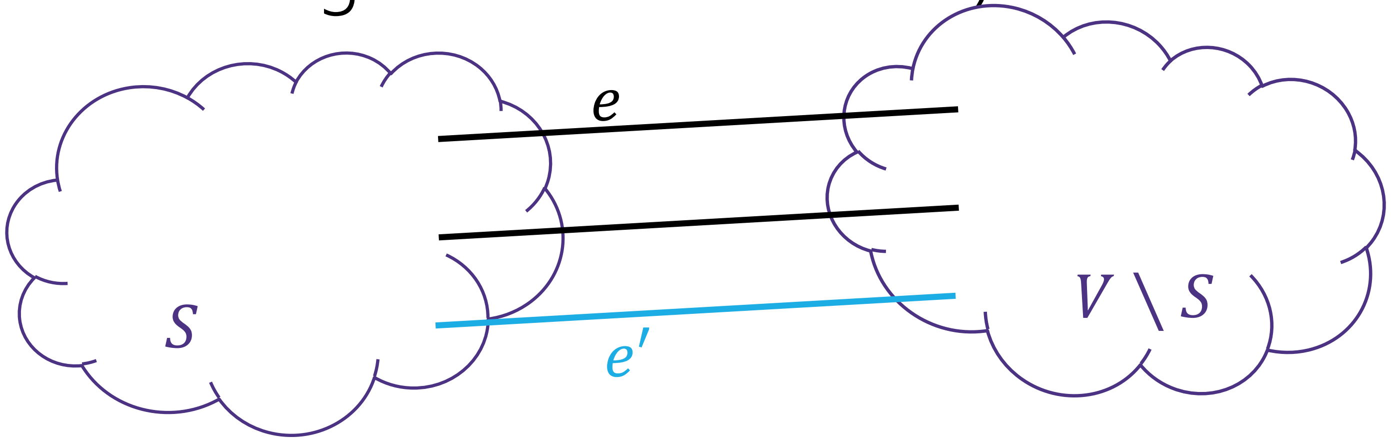


MSTs and Safe Edges

Claim: 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' 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')



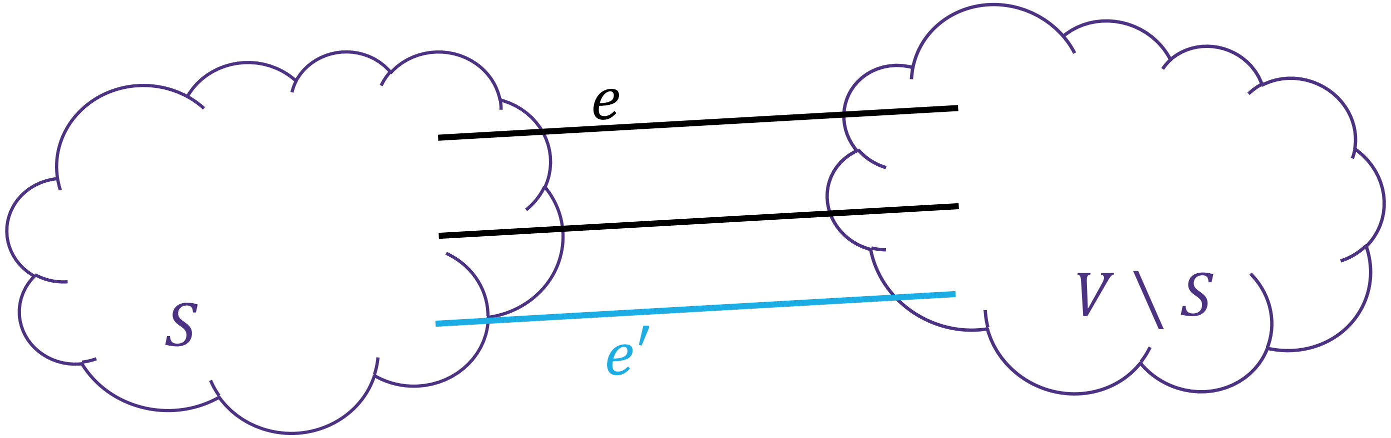
MSTs and Safe Edges

Claim: Every safe edge is in the MST.

Add $e=(u,v)$ to T'.

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' (since it was a spanning tree).

Consider T'', which is T' with e added and e' removed.



Claim: Every safe edge is in the MST.

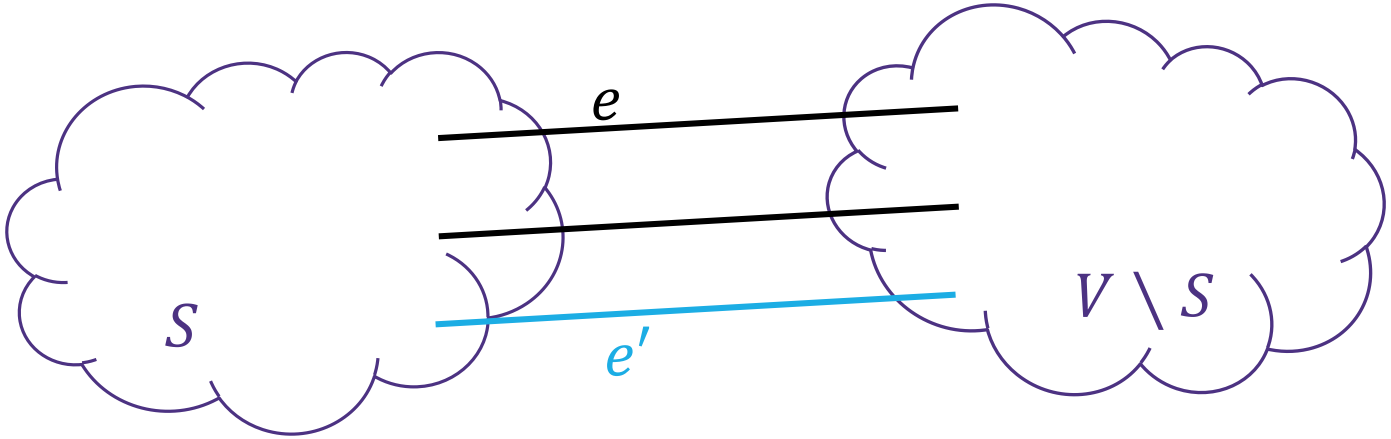
MSTs and Safe Edges

Consider T'', which is T' with e added and e' removed.

T'' spans: if the path from x to y in T' 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'; e was the lightest edge spanning $(S, V \setminus S)$. That's a contradiction! T' was the minimum spanning tree.



Structural Result

That's the structural result.

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

Theorem: Every safe edge is in the MST.

So what? The goal is to analyze an algorithm!

Let's start with Prim's!

Prim’s only adds safe edges

Claim: 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!

Are we done? Do we know Prim’s Algorithm is correct now?

Not yet! What if there are still other edges to add!

Why Aren't We Done?

Imagine we define an "ultra-safe" edge as an edge that is lighter than every other edge in the graph.

With a similar proof to our last one, you can show every "ultra-safe" edge is in the MST.

Now imagine Brim's Algorithm: sort the edges by increasing weight, add the first edge in the sorted list.

Brim's algorithm only adds ultra-safe edges!

But that's not a correct MST algorithm!!!

Prim's only adds safe edges

Claim: 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 produce an acyclic, connected, spanning graph (since each edge must connect new vertices, we can't create a cycle; the loop ends only when the graph is connected). So we have a (full) spanning tree.

An Exchange Argument

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 (v1)

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

Let $e = (u, v)$ be an edge in T_K but not T'. Add e to T'. In doing so we created a cycle, C, (e along with the path from u to v in T', which exists because T' spanned.).

Our goal is to do an exchange argument – we need a new lighter tree!

We divide into cases,

Case 1: e is not the heaviest edge in C. Then delete the heaviest edge to create T''. Since e replaced the heavier edge, T'' is lighter than T'. And T'' is a spanning tree (T'' has n-1 edges and spans because T' did and we just deleted an edge on a cycle). But that contradicts T' being the MST!

Kruskal's Proof (v1, cont.)

We won't be able to reach a contradiction from the cycle, but we will find another edge to examine

Case 2: e is the heaviest edge in C.

Since Kruskal's added e to our graph, there must be some edge, f, on the cycle which was not in T_K. But f was processed before e by Kruskal's (since e is heavier). Which means f would have formed a cycle, C' in T_K had it been added when it was processed.

By the process ordering, f is the heaviest edge in C'. There are no cycles in T' (since it's a tree) so there is an edge (call it e') in C' that is not in T'. This new edge e' meets exactly the assumptions we had on e, but is lighter.

Repeat the original argument on e'. Since the graph is finite, we must eventually hit Case 1, which gives our needed contradiction.

Kruskal's Proof (pretty version)

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

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

e is not the heaviest edge on the cycle. Anything lighter than e is already in T_K, and we put e in T_K so it didn't create a cycle there (since we check for cycles before adding it). That means there is an edge on the cycle heavier than e. Delete that edge, and call the resulting graph T''. Observe that T'' is a spanning tree (it has $n - 1$ edges, and spans all the same vertices T' did since we deleted an edge from a cycle). But it has less weight than T' 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 you do in every possible proof.

Wrapping MSTs

Another MST Algorithm

Boruvka's Algorithm (also called Sollin's Algorithm)

Start with empty graph, use BFS to find lightest edge leaving each component.

Add ALL such edges found (they're all safe edges)

Recurse until the graph is all one component (i.e. a tree)

Consider it for your practical applications!

It naturally parallelizes (unlike the other MST algorithms),

Has same worst case running time as Prim's/Kruskal's!

More Greedy

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.

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}.