CSE332: Data Abstractions

Lecture 24: Minimum Spanning Trees

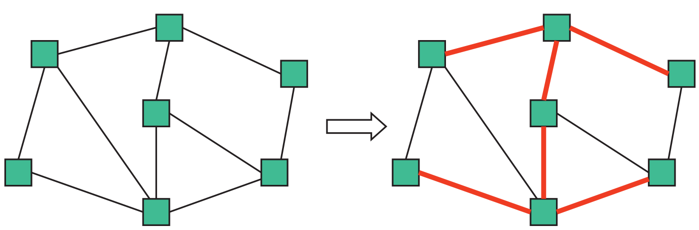
Dan Grossman
Spring 2012

"Scheduling note"

- "We now return to our interrupted program" on graphs
  - Last "graph lecture" was lecture 17
    - Shortest-path problem
    - Dijkstra's algorithm for graphs with non-negative weights
- Why this strange schedule?
  - Needed to do parallelism and concurrency in time for project 3 and homeworks 6 and 7
  - But cannot delay all of graphs because of the CSE312 co-requisite
- So: not the most logical order, but hopefully not a big deal

Spanning Trees

- A simple problem: Given a *connected* graph **G**=(**V**,**E**), find a minimal subset of the edges such that the graph is still connected
  - A graph **G2**=(**V**,**E2**) such that **G2** is connected and removing any edge from **E2** makes **G2** disconnected



Observations

1. Any solution to this problem is a tree
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected
2. Solution not unique unless original graph was already a tree
3. Problem ill-defined if original graph not connected
4. A tree with **|V|** nodes has **|V|-1** edges
   - So every solution to the spanning tree problem has **|V|-1** edges

Motivation

A spanning tree connects all the nodes with as few edges as possible

- Example: A "phone tree" so everybody gets the message and no unnecessary calls get made
  - Bad example since would prefer a balanced tree

In most compelling uses, we have a *weighted* undirected graph and we want a tree of least total cost

- Example: Electrical wiring for a house or clock wires on a chip
- Example: A road network if you cared about asphalt cost rather than travel time

This is the minimum spanning tree problem

- Will do that next, after intuition from the simpler case

Two Approaches

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree
2. Iterate through edges; add to output any edge that does not create a cycle

Spanning tree via DFS

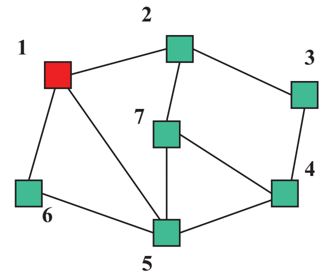
```
spanning_tree(Graph G) {
  for each node i: i.marked = false
  for some node i: f(i)
}
f(Node i) {
  i.marked = true
  for each j adjacent to i:
    if(!j.marked) {
      add(i,j) to output
      f(j) // DFS
    }
}
```

Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|\mathbf{E}|)$

Example

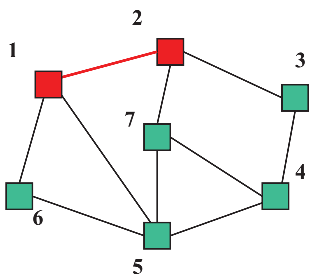
Stack
f(1)



Output:

Example

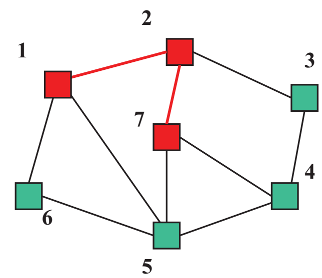
Stack
f(1)
f(2)



Output: (1,2)

Example

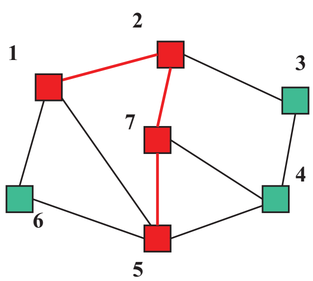
Stack
f(1)
f(2)
f(7)



Output: (1,2), (2,7)

Example

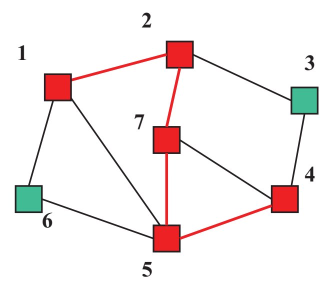
Stack
f(1)
f(2)
f(7)
f(5)



Output: (1,2), (2,7), (7,5)

Example

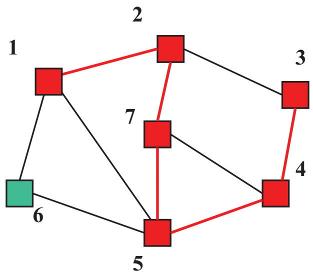
Stack
f(1)
f(2)
f(7)
f(5)
f(4)



Output: (1,2), (2,7), (7,5), (5,4)

Example

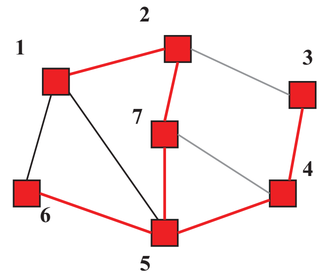
Stack
f(1)
f(2)
f(7)
f(5)
f(4)
f(3)



Output: (1,2), (2,7), (7,5), (5,4),(4,3)

Example

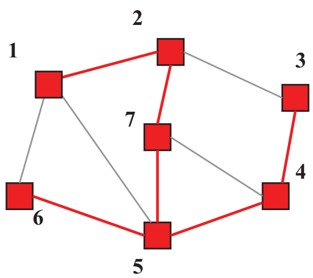
Stack
f(1)
f(2)
f(7)
f(5)
f(4) f(6)
f(3)



Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

Example

Stack
f(1)
f(2)
f(7)
f(5)
f(4) f(6)
f(3)



Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

Second Approach

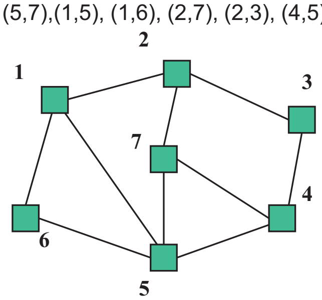
Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):
- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree
  - Else it would have created a cycle
- The graph is connected, so we reach all vertices

Efficiency:
- Depends on how quickly you can detect cycles
- Reconsider after the example

Example

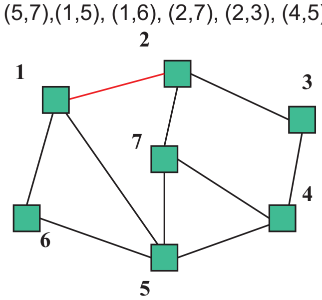
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output:

Example

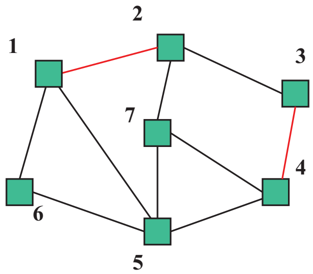
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2)

Example

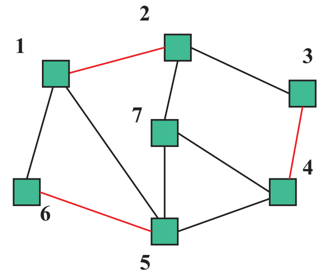
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4)

Example

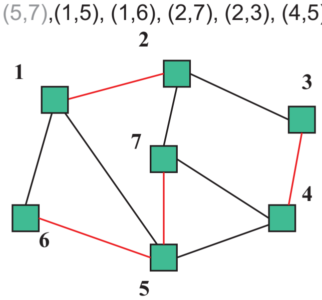
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6),

Example

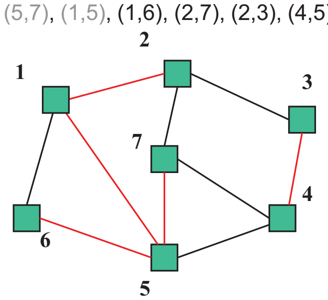
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6), (5,7)

Example

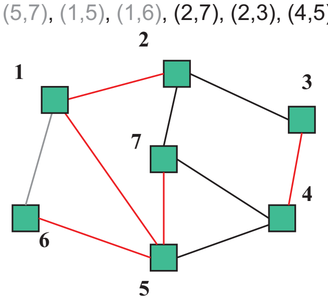
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

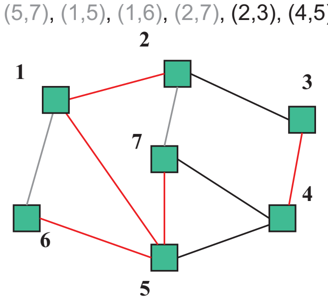
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

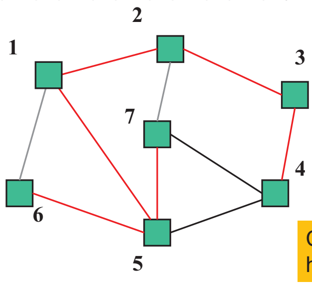


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Can stop once we have **|V|-1** edges

Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

Cycle Detection

- To decide if an edge could form a cycle is *O*(**|V|**) because we may need to traverse all edges already in the output
- So overall algorithm would be *O*(**|V||E|**)
- But there is a faster way using the disjoint-set ADT
  - Initially, each item is in its own 1-element set
  - `find(u,v)` : are `u` and `v` in the same set?
  - `union(u,v)` : union (combine) the sets containing `u` and `v`

  (Operations often presented slightly differently)

Using Disjoint-Set

Can use a disjoint-set implementation in our spanning-tree algorithm to detect cycles:

Invariant: `u` and `v` are connected in output-so-far
iff
`u` and `v` in the same set

- Initially, each node is in its own set
- When processing edge `(u,v)`:
  - If `find(u,v)`, then do not add the edge
  - Else add the edge and `union(u,v)`

Why Do This?

- Using an ADT someone else wrote is easier than writing your own cycle detection
- It is also more efficient
- Chapter 8 of your textbook gives several implementations of different sophistication and asymptotic complexity
  - A slightly clever and easy-to-implement one is *O*(`log` *n*) for `find` and `union` (as we defined the operations here)
  - Lets our spanning tree algorithm be *O*(**|E|**`log`**|V|**)

[We skipped disjoint-sets as an example of "sometimes knowing-an-ADT-exists and you-can-learn-it-on-your-own suffices"]

Summary So Far

The spanning-tree problem
- Add nodes to partial tree approach is *O*(**|E|**)
- Add acyclic edges approach is *O*(**|E|**`log`**|V|**)
  - Using the disjoint-set ADT "as a black box"

But really want to solve the minimum-spanning-tree problem
- Given a weighted undirected graph, give a spanning tree of minimum weight
- Same two approaches will work with minor modifications
- Both will be *O*(**|E|**`log`**|V|**)

Getting to the Point

Algorithm #1

Shortest-path is to Dijkstra's Algorithm
as
Minimum Spanning Tree is to Prim's Algorithm

(Both based on expanding cloud of known vertices, basically using a priority queue instead of a DFS stack)

Algorithm #2

Kruskal's Algorithm for Minimum Spanning Tree
is
Exactly our 2nd approach to spanning tree
but process edges in cost order

Prim's Algorithm Idea

Idea: Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices. *Pick the edge with the smallest weight that connects "known" to "unknown."*

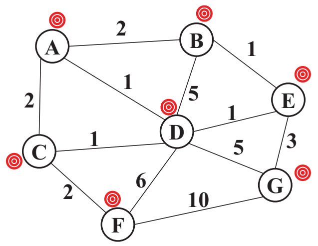
Recall Dijkstra "picked edge with closest known distance to source"
- That is not what we want here
- Otherwise identical
- Compare to slides in lecture 17 if you do not believe me

The Algorithm

1. For each node **`v`**, set **`v.cost = ∞`** and **`v.known = false`**
2. Choose any node **`v`**
   a) Mark **`v`** as known
   b) For each edge **`(v,u)`** with weight **`w`**, set **`u.cost=w`** and **`u.prev=v`**
3. While there are unknown nodes in the graph
   a) Select the unknown node **`v`** with lowest cost
   b) Mark **`v`** as known and add **`(v, v.prev)`** to output
   c) For each edge **`(v,u)`** with weight **`w`**,

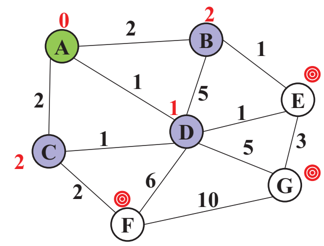
```
if(w < u.cost) {
  u.cost = w;
  u.prev = v;
}
```


Example



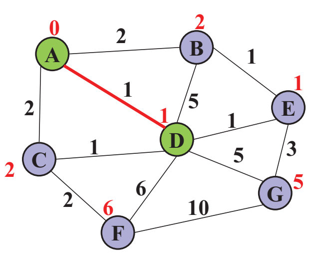
vertex	known?	cost	prev
A		??	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	

Example



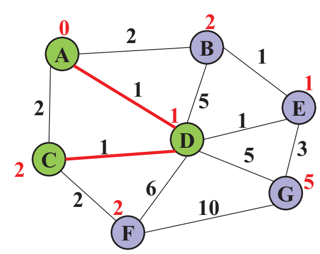
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		2	A
D		1	A
E		??	
F		??	
G		??	

Example



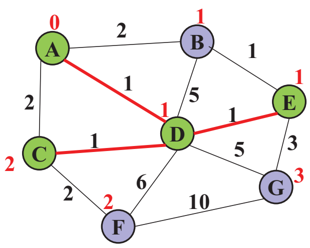
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

Example



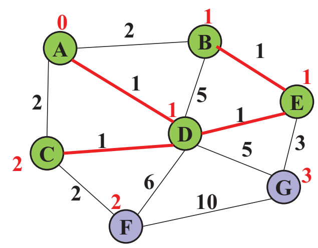
vertex	known?	cost	prev
A	Y	0	
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

Example



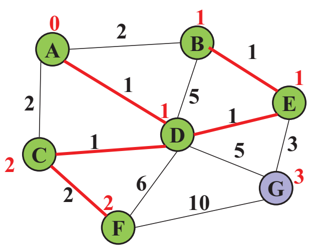
vertex	known?	cost	prev
A	Y	0	
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Example



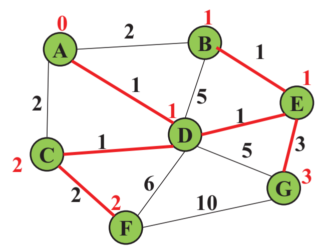
vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

Example



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E

Example



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

Analysis

- Correctness ??
  - A bit tricky
  - Intuitively similar to Dijkstra
- Run-time
  - Same as Dijkstra
  - $O(|\mathbf{E}|\log|\mathbf{V}|)$ using a priority queue

Kruskal's Algorithm

Idea: Grow a forest out of edges that do not grow a cycle, just like for the spanning tree problem.

- But now consider the edges in order by weight

So:

- Sort edges: $O(|\mathbf{E}|\log|\mathbf{E}|)$
- Iterate through edges using union-find for cycle detection $O(|\mathbf{E}|\log|\mathbf{V}|)$

Somewhat better:

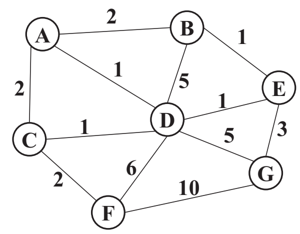
- Floyd's algorithm to build min-heap with edges $O(|\mathbf{E}|)$
- Iterate through edges using union-find for cycle detection and `deleteMin` to get next edge $O(|\mathbf{E}|\log|\mathbf{V}|)$
- Not better *worst-case* asymptotically, but often stop long before considering all edges

Pseudocode

1. Sort edges by weight (better: put in min-heap)
2. Each node in its own set
3. While output size **< |V|-1**
   – Consider next smallest edge `(u,v)`
   – if `find(u,v)` indicates `u` and `v` are in different sets
      • output `(u,v)`
      • `union(u,v)`

Recall invariant:
`u` and `v` in same set if and only if connected in output-so-far

Example

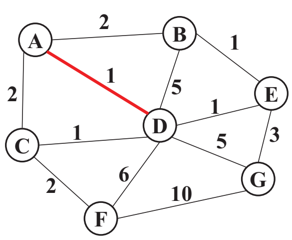


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output:

Note: At each step, the union/find sets are the trees in the forest

Example

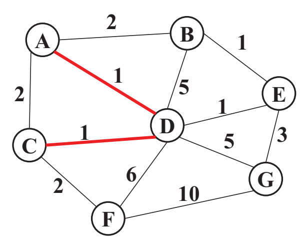


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D)

Note: At each step, the union/find sets are the trees in the forest

Example

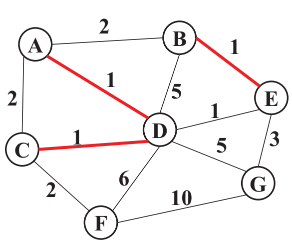


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D), (C,D)

Note: At each step, the union/find sets are the trees in the forest

Example

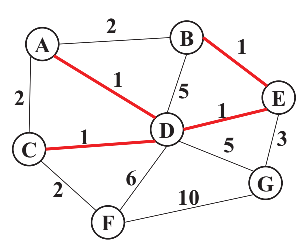


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D), (C,D), (B,E)

Note: At each step, the union/find sets are the trees in the forest

Example

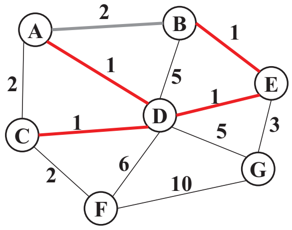


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

Example

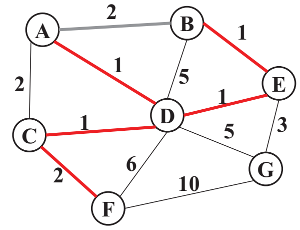


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

Example

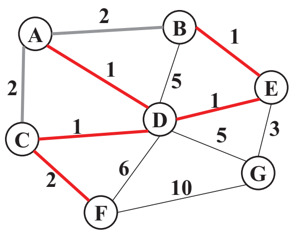


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

Example

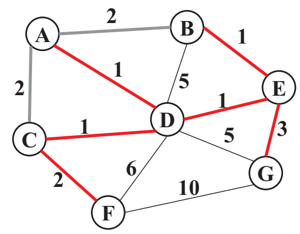


Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

Example



Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Note: At each step, the union/find sets are the trees in the forest

Correctness

Kruskal's algorithm is clever, simple, and efficient
- But does it generate a minimum spanning tree?
- How can we prove it?

First: it generates a spanning tree
- Intuition: Graph started connected and we added every edge that did not create a cycle
- Proof by contradiction: Suppose `u` and `v` are disconnected in Kruskal's result. Then there's a path from `u` to `v` in the initial graph with an edge we could add without creating a cycle. But Kruskal would have added that edge. Contradiction.

Second: There is no spanning tree with lower total cost…

The inductive proof set-up

Let **F** (stands for "forest") be the set of edges Kruskal has added at some point during its execution.

Claim: **F** is a subset of *one or more* MSTs for the graph
- Therefore, once **|F|=|V|-1**, we have an MST

Proof: By induction on **|F|**

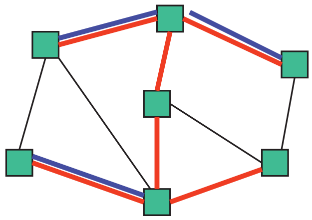
Base case: **|F|=0**: The empty set is a subset of all MSTs

Inductive case: **|F|=k+1**: By induction, before adding the $(k+1)^{th}$ edge (call it **e**), there was some MST **T** such that **F-{e}** $\subseteq$ **T** …

Staying a subset of ***some*** *MST*

Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** ⊆ **T**:



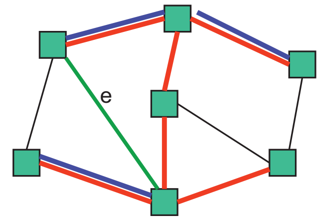
Two disjoint cases:

- If **{e}** ⊆ **T**: Then **F** ⊆ **T** and we're done
- Else **e** forms a cycle with some simple path (call it **p**) in **T**
  - Must be since **T** is a spanning tree

Staying a subset of ***some*** *MST*

Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** ⊆ **T** and
e forms a cycle with **p** ⊆ **T**

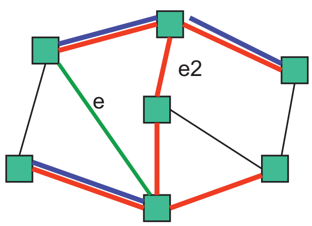


- There must be an edge **e2** on **p** such that **e2** is not in **F**
  - Else Kruskal would not have added **e**
- Claim: **e2.weight == e.weight**

Staying a subset of ***some*** *MST*

Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** ⊆ **T**
e forms a cycle with **p** ⊆ **T**
e2 on **p** is not in **F**

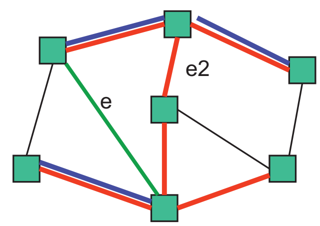


- Claim: **e2.weight == e.weight**
  - If **e2.weight > e.weight**, then **T** is not an MST because **T-{e2}+{e}** is a spanning tree with lower cost: contradiction
  - If **e2.weight < e.weight**, then Kruskal would have already considered **e2**. It would have added it since **T** has no cycles and **F-{e}** ⊆ **T.** But **e2** is not in **F**: contradiction

Staying a subset of ***some*** *MST*

Claim: **F** is a subset of *one or more* MSTs for the graph

So far: **F-{e}** ⊆ **T**
e forms a cycle with **p** ⊆ **T**
e2 on **p** is not in **F**
e2.weight == e.weight



- Claim: **T-{e2}+{e}** is an MST
  - It is a spanning tree because **p-{e2}+{e}** connects the same nodes as **p**
  - It is minimal because its cost equals cost of **T**, an MST
- Since **F** ⊆ **T-{e2}+{e}**, **F** is a subset of one or more MSTs

Done