CSE373: Data Structures & Algorithms

Lecture 17: More Dijkstra’s and Minimum Spanning Trees

Aaron Bauer

Winter 2014
Dijkstra’s Algorithm: Idea

- Initially, start node has cost 0 and all other nodes have cost $\infty$
- At each step:
  - Pick closest unknown vertex $v$
  - Add it to the “cloud” of known vertices
  - Update distances for nodes with edges from $v$
- That’s it! (But we need to prove it produces correct answers)
The Algorithm

1. For each node \( v \), set \( v\.cost = \infty \) and \( v\.known = \text{false} \)
2. Set \( \text{source}.cost = 0 \)
3. While there are unknown nodes in the graph
   a) Select the unknown node \( v \) with lowest cost
   b) Mark \( v \) as known
   c) For each edge \( (v,u) \) with weight \( w \),
      \[
      c_1 = v\.cost + w \quad \text{// cost of best path through } v \text{ to } u \\
      c_2 = u\.cost \quad \text{// cost of best path to } u \text{ previously known}
      \]
      if\( (c_1 < c_2) \) \{ // if the path through \( v \) is better
         u\.cost = c_1 \\
         u.path = v \quad \text{// for computing actual paths}
      \}
**Efficiency, first approach**

Use pseudocode to determine asymptotic run-time

– Notice each edge is processed only once

```plaintext
dijkstra(Graph G, Node start) {
    for each node: x.cost=infinity, x.known=false
    start.cost = 0
    while(not all nodes are known) {
        b = find unknown node with smallest cost
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    a.cost = b.cost + weight((b,a))
                    a.path = b
                }
    }
}
```

- $O(|V|)$
- $O(|V|^2)$
- $O(|E|)$
- $O(|V|^2)$
Improving (?) asymptotic running time

• So far: $O(|V|^2)$

• We had a similar “problem” with topological sort being $O(|V|^2)$ due to each iteration looking for the node to process next
  – We solved it with a queue of zero-degree nodes
  – But here we need the lowest-cost node and costs can change as we process edges

• Solution?
  – A priority queue holding all unknown nodes, sorted by cost
  – But must support **decreaseKey** operation
    • Must maintain a reference from each node to its current position in the priority queue
    • Conceptually simple, but can be a pain to code up
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=∞, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    decreaseKey(a, “new cost - old cost”)
                    a.path = b
                }
    }
}
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=infinty, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    decreaseKey(a,"new cost - old cost")
                    a.path = b
                }
    }
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=\infty, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost) {
                    decreaseKey(a, “new cost - old cost”)
                    a.path = b
                }
    }
}
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```plaintext
dijkstra(Graph G, Node start) {
    for each node: x.cost=∞, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    decreaseKey(a, "new cost - old cost")
                    a.path = b
                }
    }
}
```

\[ O(|V|) \]

\[ O(|V|\log|V|) \]

\[ O(|E|\log|V|) \]
Efficiency, second approach

Use pseudocode to determine asymptotic run-time

dijkstra(Graph G, Node start) {
    for each node: x.cost=∞, x.known=false
    start.cost = 0
    build-heap with all nodes
    while(heap is not empty) {
        b = deleteMin()
        b.known = true
        for each edge (b,a) in G
            if(!a.known)
                if(b.cost + weight((b,a)) < a.cost){
                    decreaseKey(a, “new cost – old cost”)
                    a.path = b
                }
    }
}
Dense vs. sparse again

• First approach: \( O(|V|^2) \)

• Second approach: \( O(|V|\log|V|+|E|\log|V|) \)

• So which is better?
  – Sparse: \( O(|V|\log|V|+|E|\log|V|) \) (if \( |E| > |V| \), then \( O(|E|\log|V|) \))
  – Dense: \( O(|V|^2) \)

• But, remember these are worst-case and asymptotic
  – Priority queue might have slightly worse constant factors
  – On the other hand, for “normal graphs”, we might call \texttt{decreaseKey} rarely (or not percolate far), making \( |E|\log|V| \) more like \( |E| \)
Spanning Trees

- A simple problem: Given a connected undirected graph $G=(V,E)$, find a minimal subset of edges such that $G$ is still connected
  - A graph $G_2=(V,E_2)$ such that $G_2$ is connected and removing any edge from $E_2$ makes $G_2$ 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
   - So $|E| \geq |V|-1$

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: $O(|E|)$
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)

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 we know: use union-find!
  – Initially, each item is in its own 1-element set
  – Union sets when we add an edge that connects them
  – Stop when we have one set
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 \( \text{find}(u) \) equals \( \text{find}(v) \), then do not add the edge
  - Else add the edge and \( \text{union}(\text{find}(u), \text{find}(v)) \)
  - \( O(|E|) \) operations that are almost \( O(1) \) amortized
Summary So Far

The spanning-tree problem
- Add nodes to partial tree approach is $O(|E|)$
- Add acyclic edges approach is almost $O(|E|)$
  - Using union-find “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 (!)
The Algorithm

1. For each node \( v \), set \( v.cost = \infty \) and \( v.known = \text{false} \)
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 \( w \),
      \[
      \text{if}(w < u.cost) \{
      \begin{align*}
      u.cost &= w; \\
      u.prev &= v;
      \end{align*}
      \}
      \]
Example

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

A vertex is known if its row in the table is filled in. The cost is the minimum distance from the source to each vertex, and the prev column indicates the previous vertex in the shortest path.
Example

A graph with vertices labeled A, B, C, D, E, F, and G. Edges are labeled with weights, and the adjacency matrix is shown in the right table.

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>??</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>??</td>
<td></td>
</tr>
</tbody>
</table>
### Example

A diagram of a graph with labeled vertices and edges is shown. The edges are labeled with their costs.

A table is also presented, listing the vertices, whether they are known, their cost, and their previous vertex in the path:

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>6</td>
<td>D</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>5</td>
<td>D</td>
</tr>
</tbody>
</table>
Example

```
  0   2   2
A   B   E
  2   1   3
C   D   G
  2   5   5
F   D   G
```

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>5</td>
<td>D</td>
</tr>
</tbody>
</table>
Example

A 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

```
<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>3</td>
<td>E</td>
</tr>
</tbody>
</table>
```
Example

- **vertex**
- **known?**
- **cost**
- **prev**

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>3</td>
<td>E</td>
</tr>
</tbody>
</table>
Example

![Graph with vertex A, B, C, D, E, F, G and edges and distances]

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>Y</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>Y</td>
<td>3</td>
<td>E</td>
</tr>
</tbody>
</table>
Analysis

• Correctness ??
  – A bit tricky
  – Intuitively similar to Dijkstra

• Run-time
  – Same as Dijkstra
  – $O(|E| \log |V|)$ using a priority queue
    • Costs/priorities are just edge-costs, not path-costs
**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: $O(|E| \log |E|)$ (next course topic)
- Iterate through edges using union-find for cycle detection almost $O(|E|)$

Somewhat better:
- Floyd’s algorithm to build min-heap with edges $O(|E|)$
- Iterate through edges using union-find for cycle detection and \texttt{deleteMin} to get next edge $O(|E| \log |E|)$
- 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 \(\text{find}(u)\) and \(\text{find}(v)\) indicate \(u\) and \(v\) are in different sets
     • output \((u,v)\)
     • \(\text{union} (\text{find}(u), \text{find}(v))\)

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’s 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 \textit{one or more} MSTs for the graph

So far: \( F - \{e\} \subseteq T \):

Two disjoint cases:

• If \( \{e\} \subseteq T \): Then \( F \subseteq T \) 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: $F\backslash\{e\} \subseteq T$ and $e$ forms a cycle with $p \subseteq T$

• There must be an edge $e_2$ on $p$ such that $e_2$ is not in $F$
  – Else Kruskal would not have added $e$

• Claim: $e_2.\text{weight} == e.\text{weight}$
Staying a subset of some MST

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

So far: $F - \{e\} \subseteq T$
- $e$ forms a cycle with $p \subseteq T$
- $e2$ 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 $F - \{e\} \subseteq T$. 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: \( F - \{ e \} \subseteq T \)
- \( e \) forms a cycle with \( p \subseteq T \)
- \( e_2 \) on \( p \) is not in \( F \)
- \( e_2 \).weight == e.weight

- Claim: \( T - \{ e_2 \} + \{ e \} \) is an MST
  - It is a spanning tree because \( p - \{ e_2 \} + \{ e \} \) connects the same nodes as \( p \)
  - It is minimal because its cost equals cost of \( T \), an MST
- Since \( F \subseteq T - \{ e_2 \} + \{ e \} \), \( F \) is a subset of one or more MSTs

Done