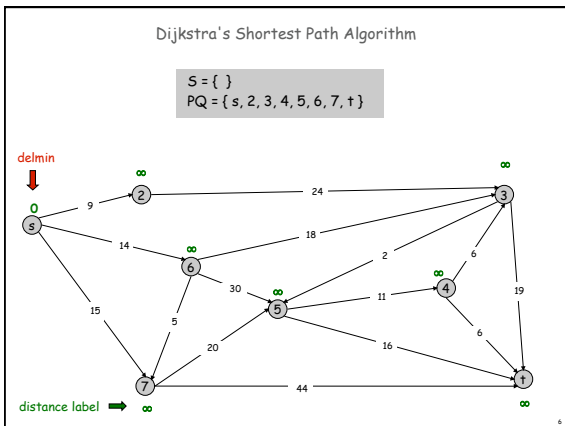
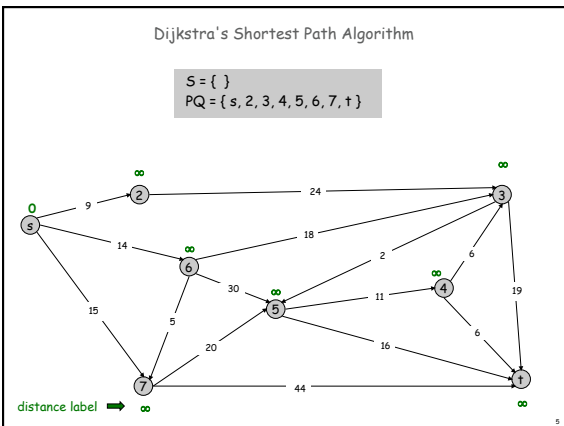
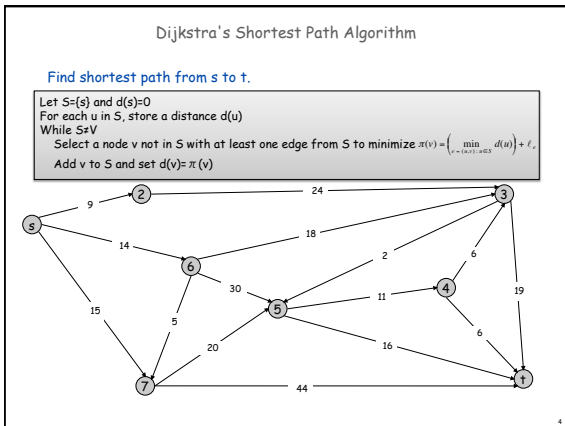
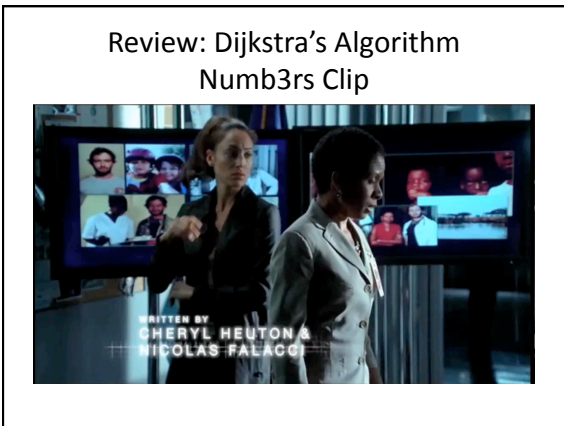


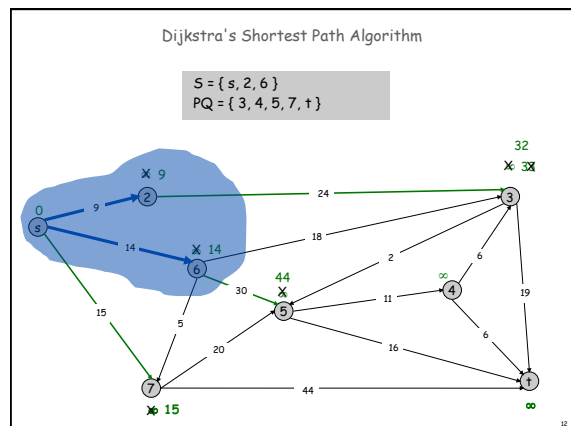
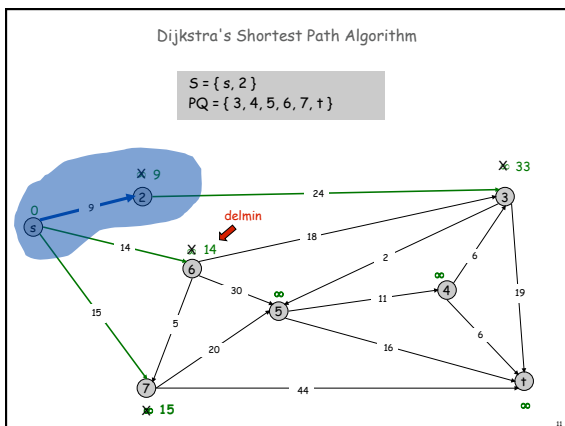
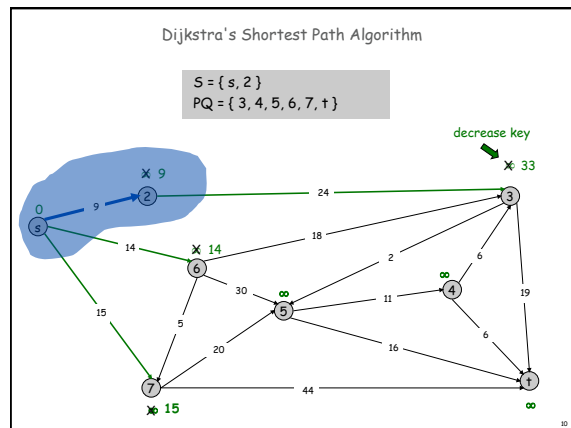
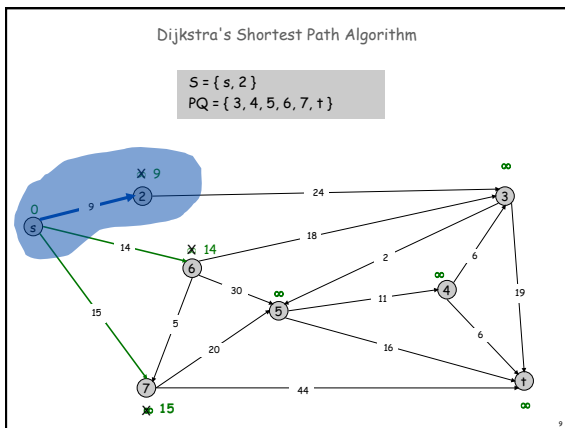
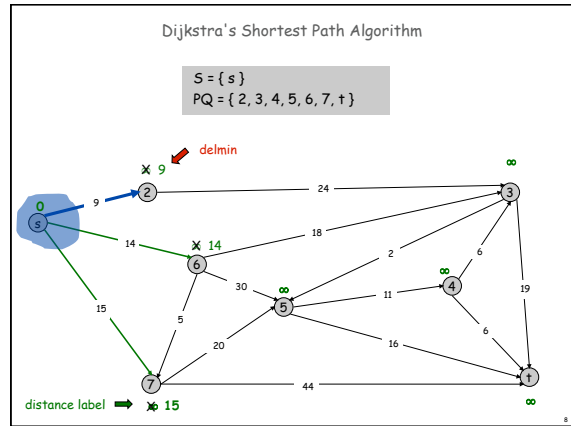
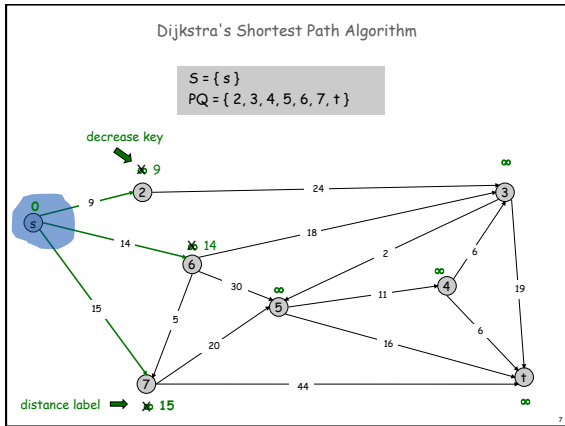
# Greedy Algorithms: Shortest Paths and Minimum Spanning Trees

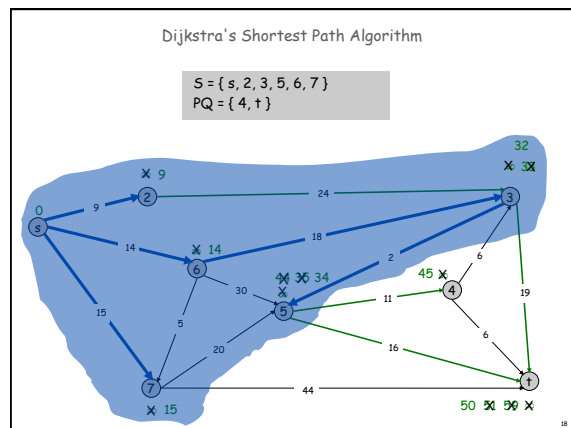
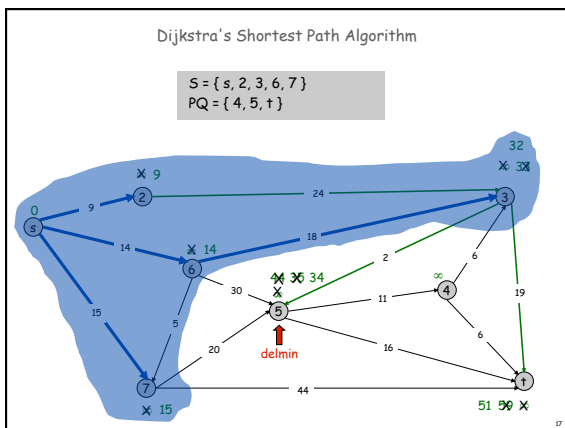
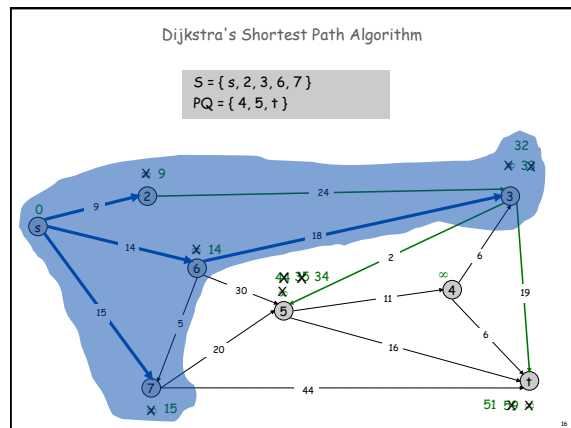
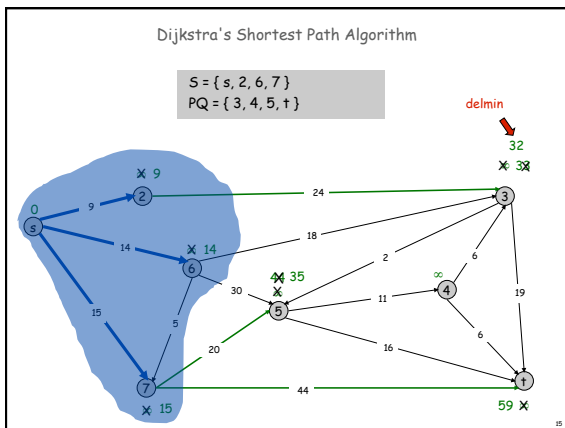
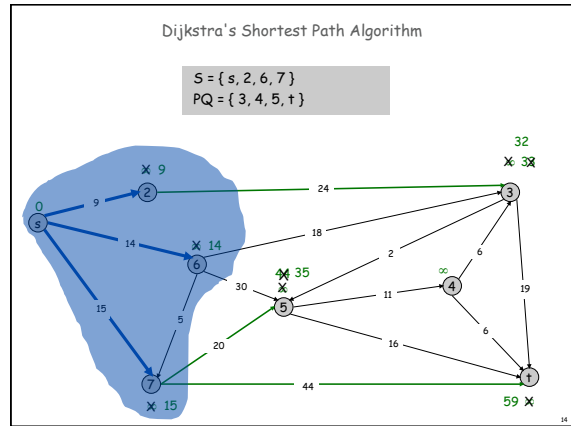
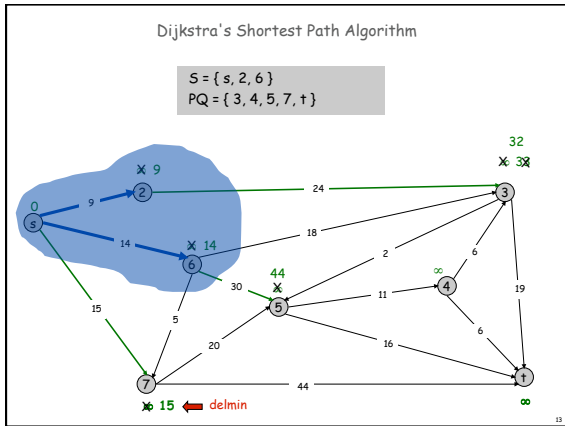
CMSC 451, Summer 2009  
Sorelle Friedler

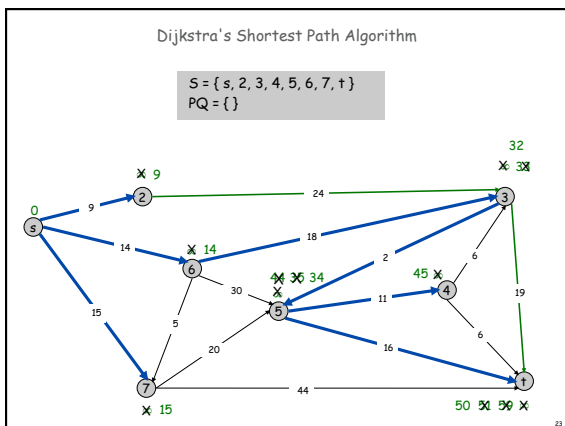
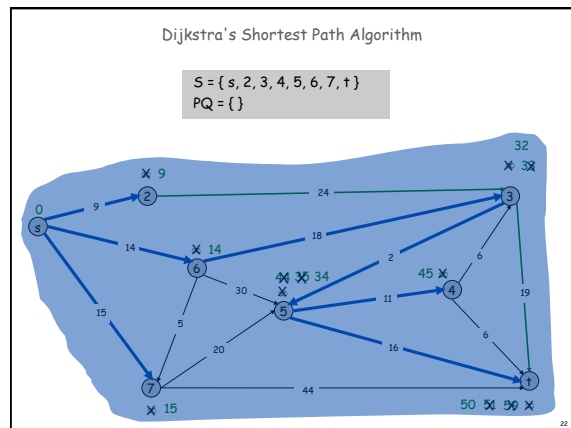
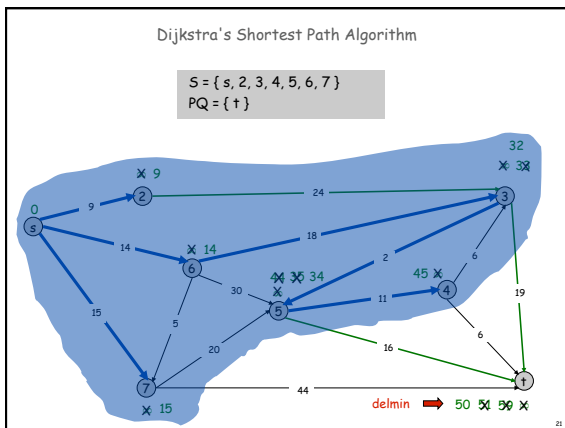
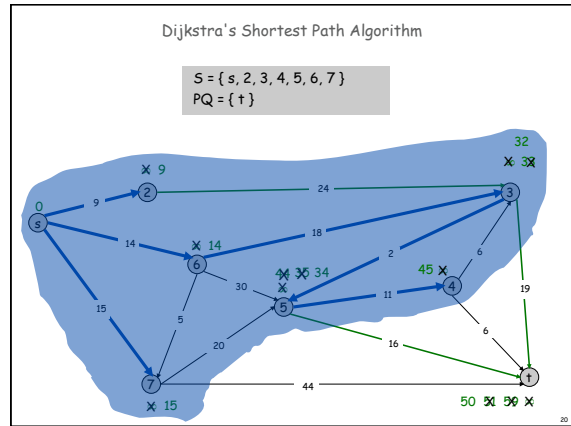
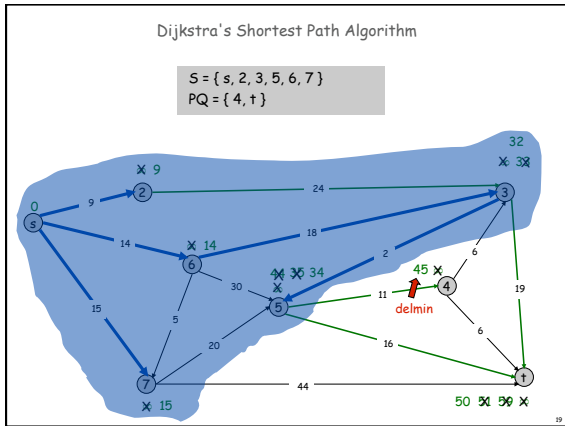
## Updates

- Survey says... we're done with graph review









Dijkstra's Algorithm: Proof of Correctness

**Invariant.** For each node  $v \in S$ ,  $d(v)$  is the length of the shortest  $s$ - $v$  path. (Applying for  $v=t$  immediately gives the proof of optimality.)

**Proof.** (by induction on  $|S|$ )

**Base case:**  $|S| = 1$ ,  $d(s)=d(t)=0$ , which is the shortest it could be.

**Inductive hypothesis:** Assume true for  $|S| = k \geq 1$ .

**Induction step: Proof for  $|S|=k+1$**

- Let  $v$  be next node added to  $S$ , and let  $u-v$  be the chosen edge.
- The shortest  $s$ - $u$  path plus  $(u, v)$  is an  $s$ - $v$  path of length  $\pi(v)$ .
- Consider any  $s$ - $v$  path  $P$ . We'll see that it's no shorter than  $\pi(v)$ .
- Let  $x-y$  be the first edge in  $P$  that leaves  $S$ , and let  $P'$  be the subpath to  $x$ .
- $P$  is already too long as soon as it leaves  $S$ .
- $d(v)$  is set to  $\pi(v)$

$$l(P) \geq l(P') + l(x,y) \geq d(x) + l(x,y) \geq \pi(y) \geq \pi(v)$$

nonnegative weights     inductive hypothesis     defn of  $\pi(y)$      Dijkstra chose  $v$  instead of  $y$

Dijkstra's Algorithm: Greedy Perspective

What is the "step" in our step-by-step creation of a solution?

What is the greedy choice being made?

Note on proof: The analysis was in the "stay ahead" form

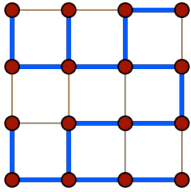
25

Spanning Tree

Given an undirected, connected graph  $G$ :

A **spanning tree** of  $G$  is a tree containing all vertices of  $G$  and some subset of the edges of  $G$ .

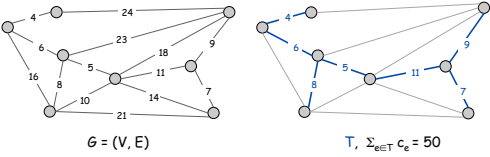
- maximal subset of edges of  $G$  with no cycle
- minimal subset of edges of  $G$  that connect all vertices



26

Minimum Spanning Tree

**Minimum spanning tree.** Given a connected graph  $G = (V, E)$  with real-valued edge weights  $c_e$ , an MST is a subset of the edges  $T \subseteq E$  such that  $T$  is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$                        $T, \sum_{e \in T} c_e = 50$

**Cayley's Theorem.** There are  $n^{n-2}$  spanning trees of  $K_n$ .

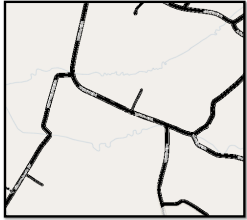
|  
can't solve by brute force

27

Applications

**MST is fundamental problem with diverse applications.**

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Cluster analysis.



28

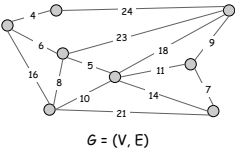
Developing a Greedy Algorithm to Find a Minimum Spanning Tree

**Options:**

- Look at all nodes and consider their adjacent edges
- Look at all edges and their weights

**Make greedy decisions to:**

- Minimize total edge cost
- Maximize cost of edges not chosen



$G = (V, E)$

29

Greedy Algorithms

**Kruskal's algorithm.** Start with  $T = \emptyset$ . Consider edges in ascending order of cost. Insert edge  $e$  in  $T$  unless doing so would create a cycle.

**Reverse-Delete algorithm.** Start with  $T = E$ . Consider edges in descending order of cost. Delete edge  $e$  from  $T$  unless doing so would disconnect  $T$ .

**Prim's algorithm.** Start with some root node  $s$  and greedily grow a tree  $T$  from  $s$  outward. At each step, add the cheapest edge  $e$  to  $T$  that has exactly one endpoint in  $T$ .

**Remark.** All three algorithms produce an MST.

30

### Kruskal's Algorithm

Find the minimum spanning tree using Kruskal's algorithm:  
Start with  $T = \emptyset$ . Consider edges in ascending order of cost. Insert edge  $e$  in  $T$  unless doing so would create a cycle.

### Reverse-Delete Algorithm

Find the minimum spanning tree using the Reverse-Delete algorithm:  
Start with  $T = E$ . Consider edges in descending order of cost. Delete edge  $e$  from  $T$  unless doing so would disconnect  $T$ .

### Prim's Algorithm

Find the minimum spanning tree using Prim's algorithm:  
Start with some root node  $s$  and greedily grow a tree  $T$  from  $s$  outward. At each step, add the cheapest edge  $e$  to  $T$  that has exactly one endpoint in  $T$ .

### Greedy Algorithms

**Simplifying assumption.** All edge costs  $c_e$  are distinct.

**Cut property.** Let  $S$  be any subset of nodes, and let  $e$  be the min cost edge with exactly one endpoint in  $S$ . Then the MST contains  $e$ .

**Cycle property.** Let  $C$  be any cycle, and let  $f$  be the max cost edge belonging to  $C$ . Then the MST does not contain  $f$ .

### Cycles and Cuts

**Cycle.** Set of edges the form  $a-b, b-c, c-d, \dots, y-z, z-a$ .

Cycle  $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$

**Cutset.** A cut is a subset of nodes  $S$ . The corresponding cutset  $D$  is the subset of edges with exactly one endpoint in  $S$ .

Cut  $S = \{4, 5, 8\}$   
Cutset  $D = 5-6, 5-7, 3-4, 3-5, 7-8$

### Cycle-Cut Intersection

**Claim.** A cycle and a cutset intersect in an even number of edges.

Cycle  $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$   
Cutset  $D = 3-4, 3-5, 5-6, 5-7, 7-8$   
Intersection =  $3-4, 5-6$

**Pf.** (by picture)

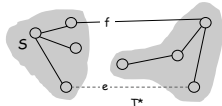
### Greedy Algorithms

**Simplifying assumption.** All edge costs  $c_e$  are distinct.

**Cut property.** Let  $S$  be any subset of nodes, and let  $e$  be the min cost edge with exactly one endpoint in  $S$ . Then the MST  $T^*$  contains  $e$ .

**Pf.** (exchange argument)

- Suppose  $e$  does not belong to  $T^*$ , and let's see what happens.
- Adding  $e$  to  $T^*$  creates a cycle  $C$  in  $T^*$ .
- Edge  $e$  is both in the cycle  $C$  and in the cutset  $D$  corresponding to  $S$   
 $\Rightarrow$  there exists another edge, say  $f$ , that is in both  $C$  and  $D$ .
- $T' = T^* \cup \{e\} - \{f\}$  is also a spanning tree.
- Since  $c_e < c_f$ ,  $\text{cost}(T') < \text{cost}(T^*)$ .
- This is a contradiction. ■



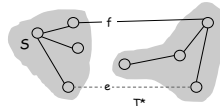
### Greedy Algorithms

**Simplifying assumption.** All edge costs  $c_e$  are distinct.

**Cycle property.** Let  $C$  be any cycle in  $G$ , and let  $f$  be the max cost edge belonging to  $C$ . Then the MST  $T^*$  does not contain  $f$ .

**Pf.** (exchange argument)

- Suppose  $f$  belongs to  $T^*$ , and let's see what happens.
- Deleting  $f$  from  $T^*$  creates a cut  $S$  in  $T^*$ .
- Edge  $f$  is both in the cycle  $C$  and in the cutset  $D$  corresponding to  $S$   
 $\Rightarrow$  there exists another edge, say  $e$ , that is in both  $C$  and  $D$ .
- $T' = T^* \cup \{e\} - \{f\}$  is also a spanning tree.
- Since  $c_e < c_f$ ,  $\text{cost}(T') < \text{cost}(T^*)$ .
- This is a contradiction. ■



### Implementation: Prim's Algorithm

**Implementation.** Use a priority queue ala Dijkstra.

- Maintain set of explored nodes  $S$ .
- For each unexplored node  $v$ , maintain attachment cost  $a[v] = \text{cost of cheapest edge } v \text{ to a node in } S$ .
- $O(n^2)$  with an array;  $O(m \log n)$  with a binary heap.

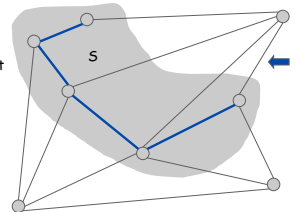
```

Prim(G, c) {
  foreach (v ∈ V) a[v] ← ∞
  Initialize an empty priority queue Q
  foreach (v ∈ V) insert v onto Q
  Initialize set of explored nodes S ← ∅
  while (Q is not empty) {
    u ← delete min element from Q
    S ← S ∪ {u}
    foreach (edge e = (u, v) incident to u)
      if ((v ∉ S) and (c_e < a[v]))
        decrease priority a[v] to c_e
  }
}
    
```

### Prim's Algorithm: Proof of Correctness (Sketch)

**Prim's algorithm.** [Jarník 1930, Dijkstra 1957, Prim 1959]

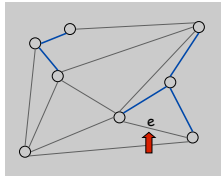
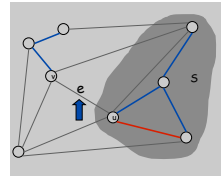
- Only edges belonging to the minimum spanning tree are added (by application of the cut property)
- A spanning tree is created since
  - No cycles exist since each added edge must have exactly one endpoint in  $S$
  - All nodes are added, since this is the what determines that the algorithm stops



### Kruskal's Algorithm

**Kruskal's algorithm.** [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding  $e$  to  $T$  creates a cycle, discard  $e$  according to cycle property.
- Case 2: Otherwise, insert  $e = (u, v)$  into  $T$  according to cut property where  $S = \text{set of nodes in } u\text{'s connected component}$ .

### Kruskal's Algorithm

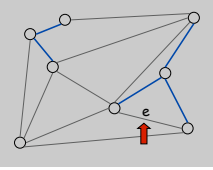
```

Kruskal(G, c) {
  Sort edge weights so that c_1 ≤ c_2 ≤ ... ≤ c_m.
  T ← ∅
  foreach (u ∈ V)
    make a set containing singleton u
  for i = 1 to m
    (u, v) = e_i
    if (u and v are in different sets) {
      T ← T ∪ {e_i}
      merge the sets containing u and v
    }
  return T
}
    
```

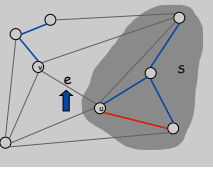
### Kruskal's Algorithm: Proof of Correctness

**Proof:**

- Only edges in the MST are added since  $e$  must span the cut since it doesn't create a cycle, and it is the cheapest (Cut property)
- A spanning tree is created
- It contains no cycles by design of the algorithm
- It is connected since otherwise there would be some edge that could be added without creating a cycle



Case 1



Case 2

43

### Implementation: Kruskal's Algorithm

```

Kruskal(G, c) {
  Sort edge weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .
  T ← ∅
  foreach (u ∈ V)
    make a set containing singleton u
  for i = 1 to m
    (u,v) =  $e_i$ 
    if (u and v are in different sets) {
      T ← T ∪ { $e_i$ }
      merge the sets containing u and v
    }
  return T
}
    
```

**Implementation.** Use the **union-find** data structure.

- **MakeUnionFind(S):** Returns a union-find data structure for set S
- **Find(u):** Return the name of the set containing node u
- **Union(A,B):** Merge the sets A and B into a single set

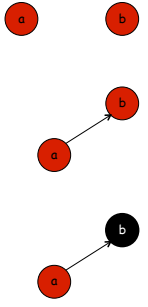
44

### Union-Find Data Structure

**MakeUnionFind(S={a,b})**  
Create singleton trees for all items in the set

**Union(A={a},B={b})**  
Merge two connected components by creating a pointer from the root of the smaller tree to the root of the larger tree. Store the size of its tree with each root.

**Find(a)**  
Traverse up the tree until finding the root. The name of the root is the name of the tree.



45

### Union-Find Data Structure: Analysis

**MakeUnionFind(S={a,b})**  
Create singleton trees for all items in the set

**Union(A={a},B={b})**  
Merge two connected components by creating a pointer from the root of the smaller tree to the root of the larger tree. Store the size of its tree with each root.

**Find(a)**  
Traverse up the tree until finding the root. The name of the root is the name of the tree.

Time analysis

$O(n)$

$O(1)$

$O(\log n)$ : Takes time on the order of the number of times the name changed. Since the larger set keeps its name, a name change implies that the set at least doubled. It can be of size at most  $n$ .

46