

# Chapter 4

## Greedy Algorithms



Slides by Kevin Wayne.  
Copyright © 2005 Pearson-Addison Wesley.  
All rights reserved.

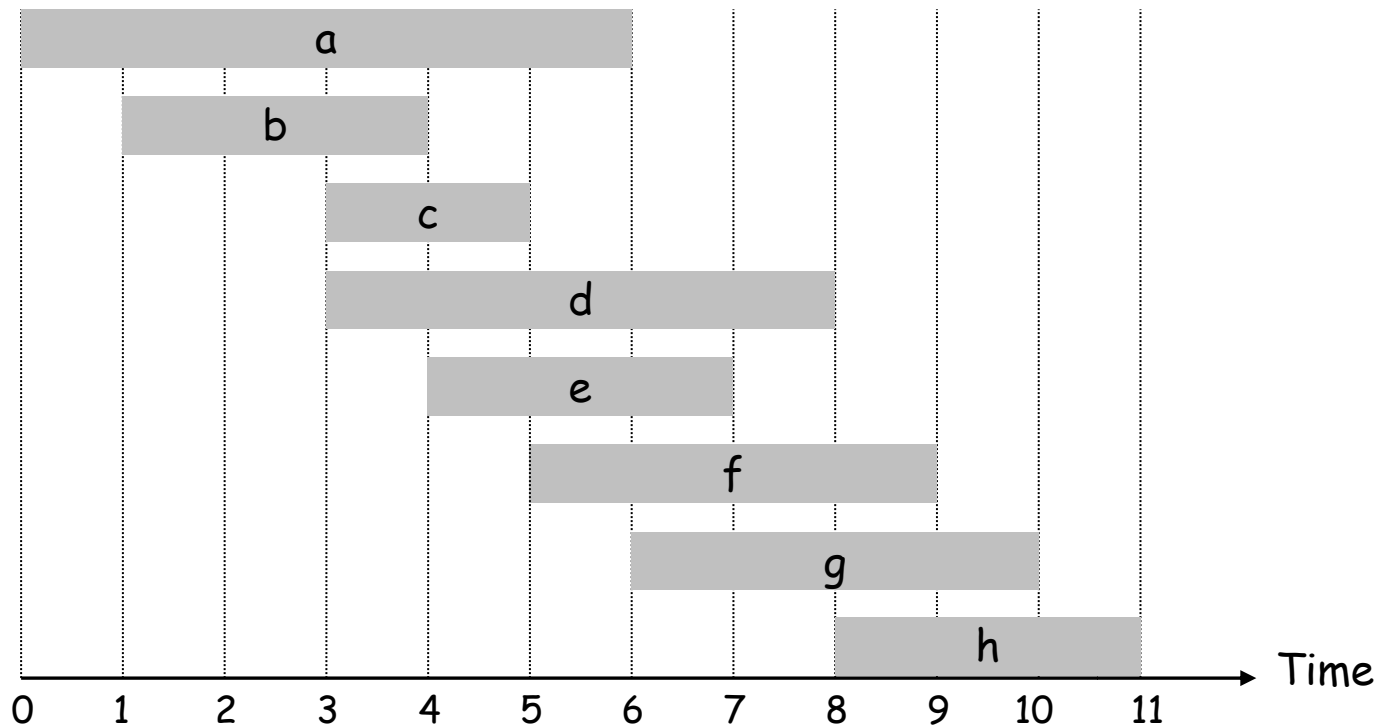
## 4.1 Interval Scheduling

---

# Interval Scheduling

## Interval scheduling.

- Job  $j$  starts at  $s_j$  and finishes at  $f_j$ .
- Two jobs **compatible** if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.



# Interval Scheduling: Greedy Algorithms

**Greedy template.** Consider jobs in some order. Take each job provided it's compatible with the ones already taken.

- [Earliest start time] Consider jobs in ascending order of start time  $s_j$ .
- [Earliest finish time] Consider jobs in ascending order of finish time  $f_j$ .
- [Shortest interval] Consider jobs in ascending order of interval length  $f_j - s_j$ .
- [Fewest conflicts] For each job, count the number of conflicting jobs  $c_j$ . Schedule in ascending order of conflicts  $c_j$ .

# Interval Scheduling: Greedy Algorithms

*Greedy template.* Consider jobs in some order. Take each job provided it's compatible with the ones already taken.



breaks earliest start time



breaks shortest interval

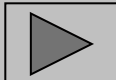


breaks fewest conflicts

# Interval Scheduling: Greedy Algorithm

**Greedy algorithm.** Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .  
  ↙ jobs selected  
A ←  $\phi$   
for j = 1 to n {  
    if (job j compatible with A)  
        A ← A  $\cup$  {j}  
}  
return A
```



**Implementation.**  $O(n \log n)$ .

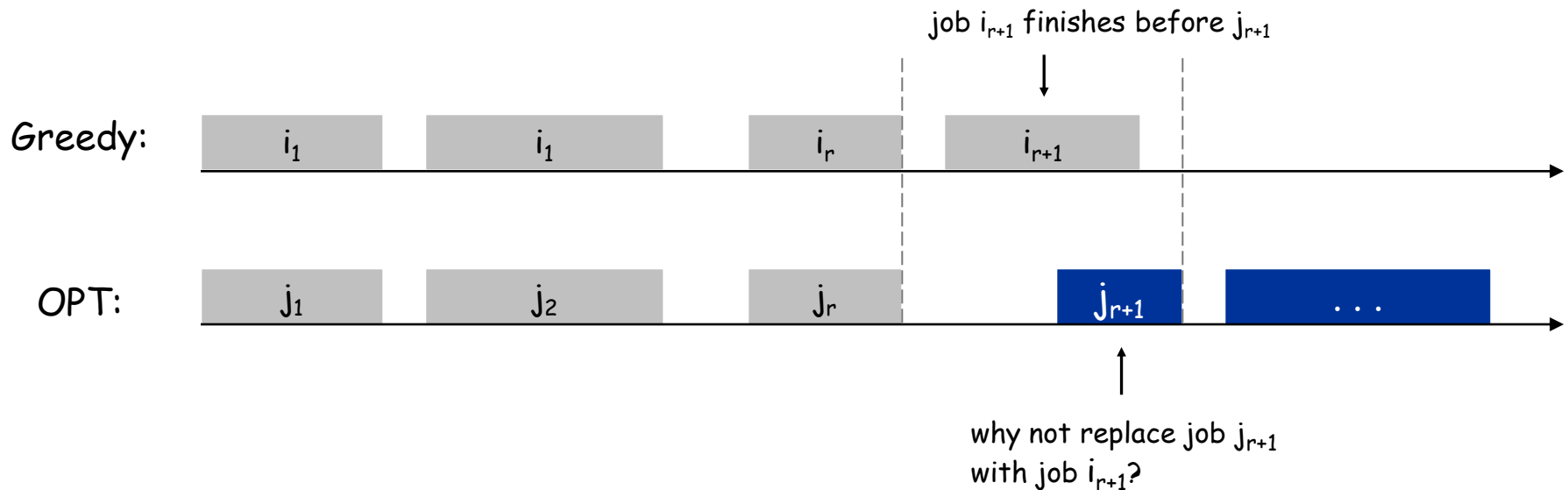
- Remember job  $j^*$  that was added last to A.
- Job j is compatible with A if  $s_j \geq f_{j^*}$ .

# Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let  $i_1, i_2, \dots, i_k$  denote set of jobs selected by greedy.
- Let  $j_1, j_2, \dots, j_m$  denote set of jobs in the optimal solution with  $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$  for the largest possible value of  $r$ .

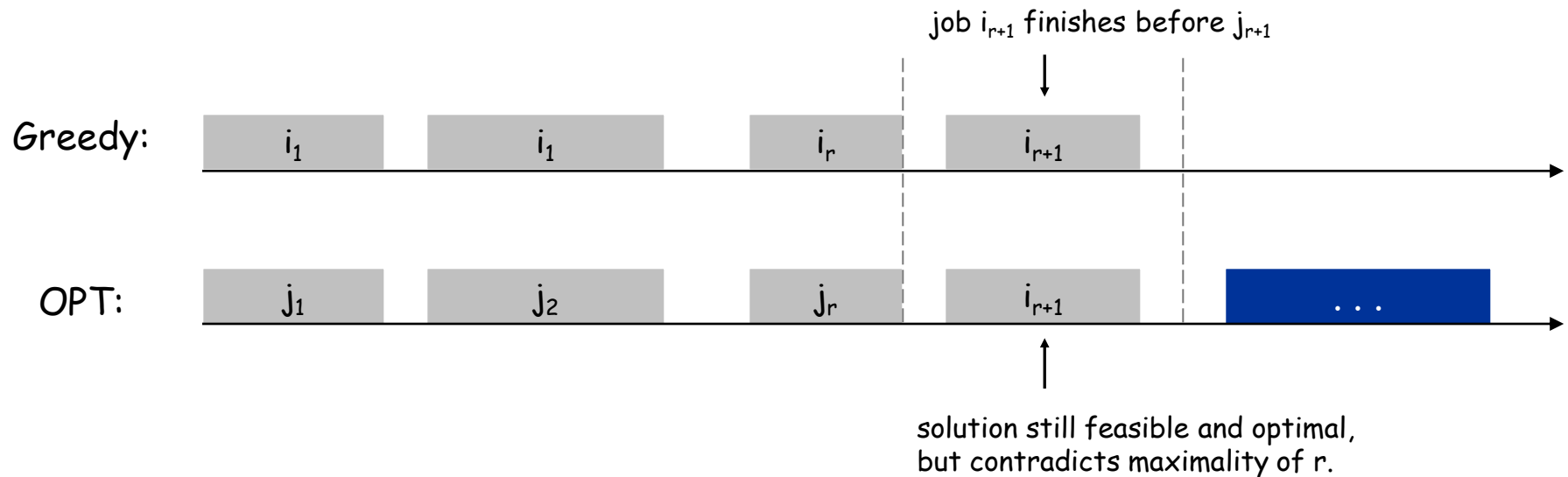


# Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let  $i_1, i_2, \dots, i_k$  denote set of jobs selected by greedy.
- Let  $j_1, j_2, \dots, j_m$  denote set of jobs in the optimal solution with  $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$  for the largest possible value of  $r$ .





## 4.1 Interval Partitioning

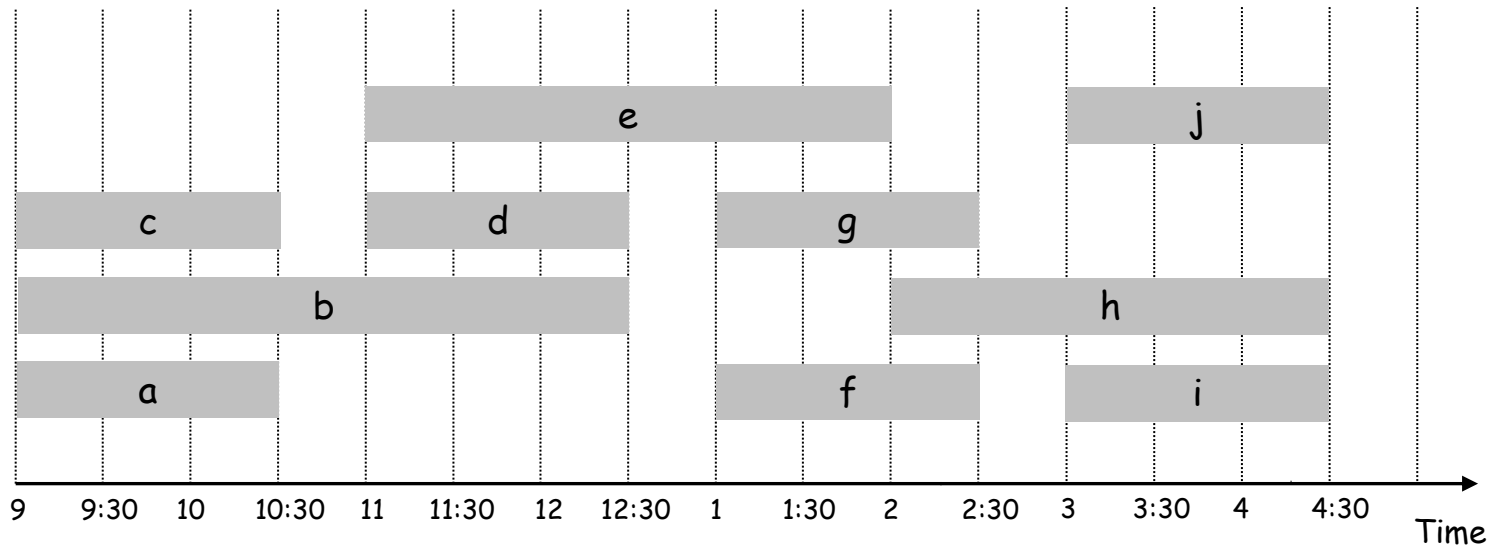
---

# Interval Partitioning

## Interval partitioning.

- Lecture  $j$  starts at  $s_j$  and finishes at  $f_j$ .
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

**Ex:** This schedule uses 4 classrooms to schedule 10 lectures.

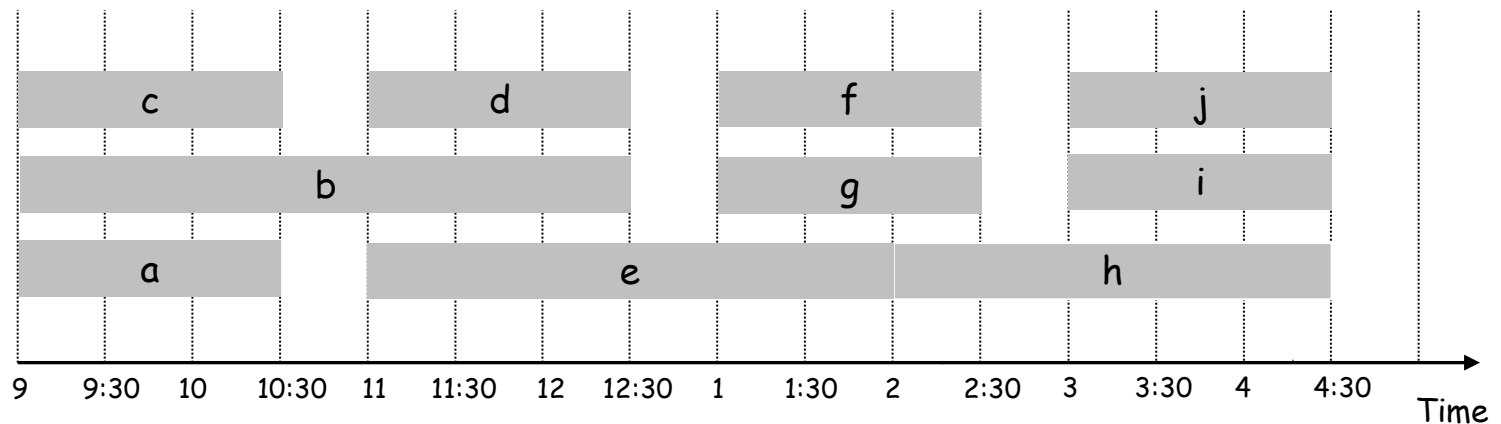


# Interval Partitioning

## Interval partitioning.

- Lecture  $j$  starts at  $s_j$  and finishes at  $f_j$ .
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.



# Interval Partitioning: Lower Bound on Optimal Solution

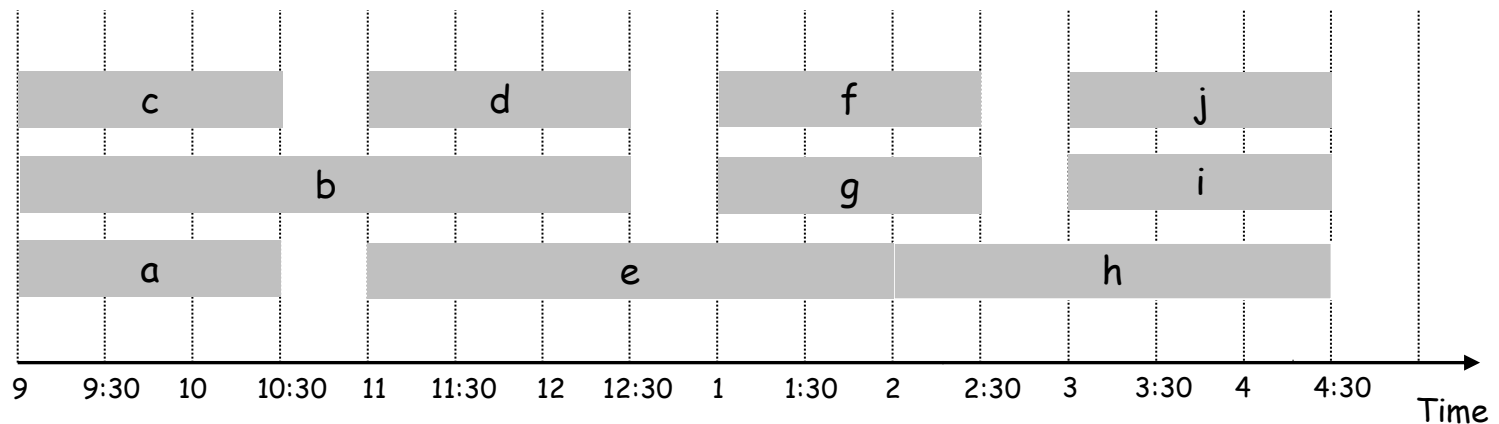
**Def.** The **depth** of a set of open intervals is the maximum number that contain any given time.

**Key observation.** Number of classrooms needed  $\geq$  depth.

**Ex:** Depth of schedule below = 3  $\Rightarrow$  schedule below is optimal.

↑  
a, b, c all contain 9:30

**Q.** Does there always exist a schedule equal to depth of intervals?



# Interval Partitioning: Greedy Algorithm

**Greedy algorithm.** Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that  $s_1 \leq s_2 \leq \dots \leq s_n$ .  
 $d \leftarrow 0$   $\leftarrow$  number of allocated classrooms  
  
for  $j = 1$  to  $n$  {  
    if (lecture  $j$  is compatible with some classroom  $k$ )  
        schedule lecture  $j$  in classroom  $k$   
    else  
        allocate a new classroom  $d + 1$   
        schedule lecture  $j$  in classroom  $d + 1$   
         $d \leftarrow d + 1$   
}
```

**Implementation.**  $O(n \log n)$ .

- For each classroom  $k$ , maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

# Interval Partitioning: Greedy Analysis

**Observation.** Greedy algorithm never schedules two incompatible lectures in the same classroom.

**Theorem.** Greedy algorithm is optimal.

**Pf.**

- Let  $d$  = number of classrooms that the greedy algorithm allocates.
- Classroom  $d$  is opened because we needed to schedule a job, say  $j$ , that is incompatible with all  $d-1$  other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than  $s_j$ .
- Thus, we have  $d$  lectures overlapping at time  $s_j + \varepsilon$ .
- Key observation  $\Rightarrow$  all schedules use  $\geq d$  classrooms. ■

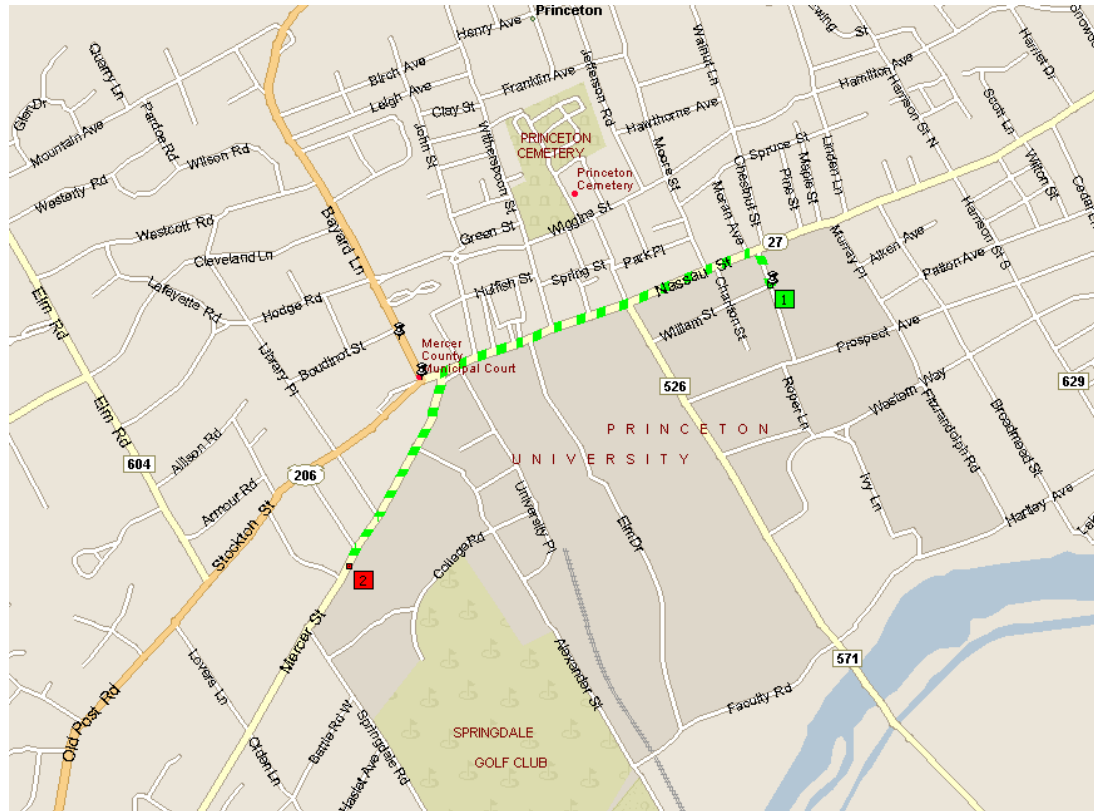
# Greedy Analysis Strategies

**Greedy algorithm stays ahead.** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

**Exchange argument.** Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

**Structural.** Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

## 4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house



# Shortest Path Problem

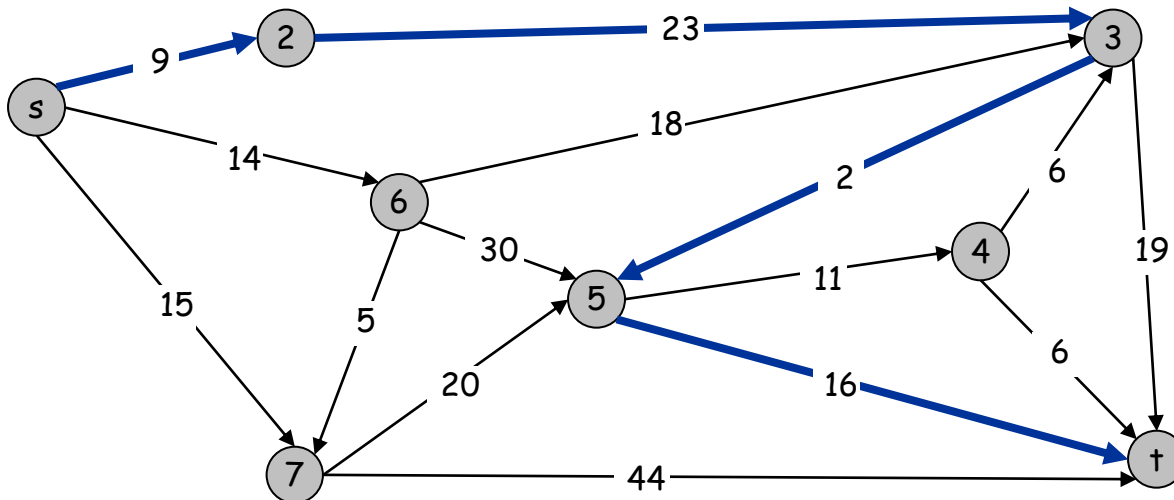
## Shortest path network.

- Directed graph  $G = (V, E)$ .
- Source  $s$ , destination  $t$ .
- Length  $\ell_e$  = length of edge  $e$ .

**Shortest path problem:** find shortest directed path from  $s$  to  $t$ .



cost of path = sum of edge costs in path



Cost of path  $s-2-3-5-t$   
=  $9 + 23 + 2 + 16$   
= 48.

# Dijkstra's Algorithm

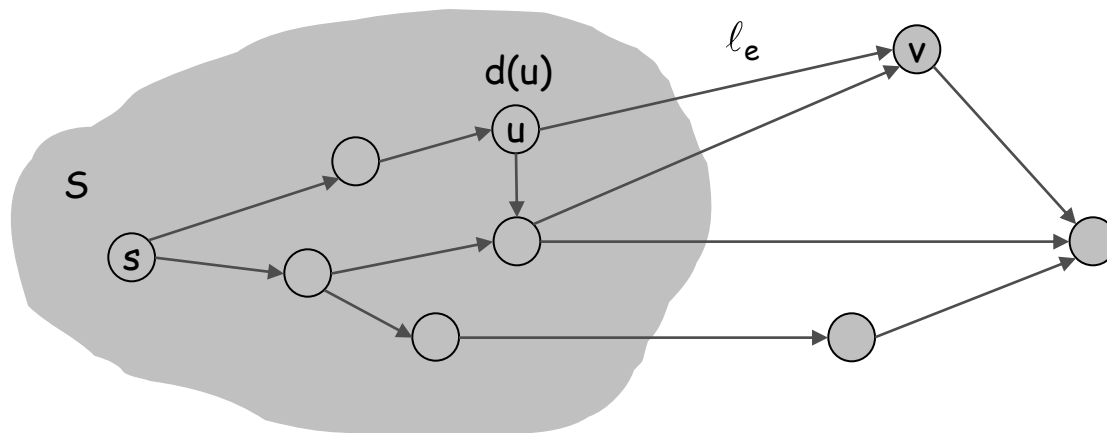
## Dijkstra's algorithm.

- Maintain a set of **explored nodes**  $S$  for which we have determined the shortest path distance  $d(u)$  from  $s$  to  $u$ .
- Initialize  $S = \{s\}$ ,  $d(s) = 0$ .
- Repeatedly choose unexplored node  $v$  which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,$$

add  $v$  to  $S$ , and set  $d(v) = \pi(v)$ .

← shortest path to some  $u$  in explored part, followed by a single edge  $(u, v)$



# Dijkstra's Algorithm

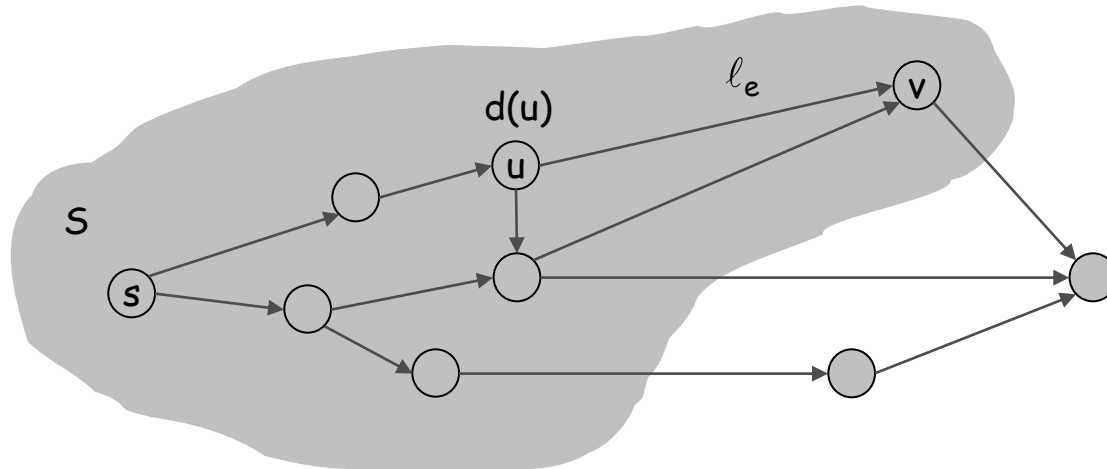
## Dijkstra's algorithm.

- Maintain a set of **explored nodes**  $S$  for which we have determined the shortest path distance  $d(u)$  from  $s$  to  $u$ .
- Initialize  $S = \{s\}$ ,  $d(s) = 0$ .
- Repeatedly choose unexplored node  $v$  which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e,$$

add  $v$  to  $S$ , and set  $d(v) = \pi(v)$ .

shortest path to some  $u$  in explored part, followed by a single edge  $(u, v)$



# Dijkstra's Algorithm: Proof of Correctness

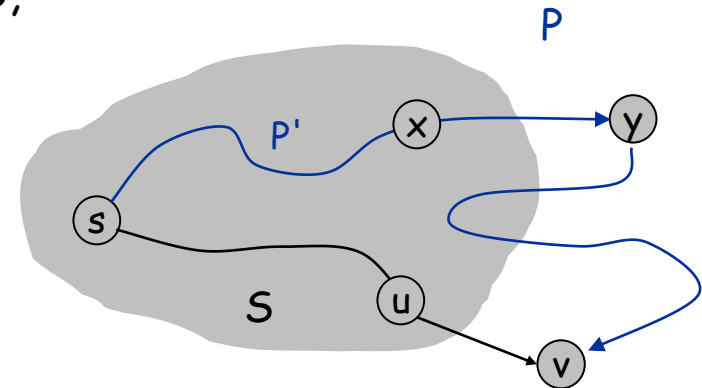
**Invariant.** For each node  $u \in S$ ,  $d(u)$  is the length of the shortest  $s$ - $u$  path.

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

**Base case:**  $|S| = 1$  is trivial.

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

- 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  $S$ .



$$\begin{array}{ccccccc} \ell(P) & \geq & \ell(P') + \ell(x, y) & \geq & d(x) + \ell(x, y) & \geq & \pi(y) \geq \pi(v) \\ \uparrow & & \uparrow & & \uparrow & & \uparrow \\ \text{nonnegative} & & \text{inductive} & & \text{defn of } \pi(y) & & \text{Dijkstra chose } v \\ \text{weights} & & \text{hypothesis} & & & & \text{instead of } y \end{array}$$

# Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain  $\pi(v) = \min_{e=(u,v): u \in S} d(u) + l_e$ .

- Next node to explore = node with minimum  $\pi(v)$ .
- When exploring  $v$ , for each incident edge  $e = (v, w)$ , update  $\pi(w) = \min \{ \pi(w), \pi(v) + l_e \}$ .

**Efficient implementation.** Maintain a priority queue of unexplored nodes, prioritized by  $\pi(v)$ .



PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap <sup>†</sup>
Insert	$n$	$n$	$\log n$	$d \log_d n$	1
ExtractMin	$n$	$n$	$\log n$	$d \log_d n$	$\log n$
ChangeKey	$m$	1	$\log n$	$\log_d n$	1
IsEmpty	$n$	1	1	1	1
Total		$n^2$	$m \log n$	$m \log_{m/n} n$	$m + n \log n$

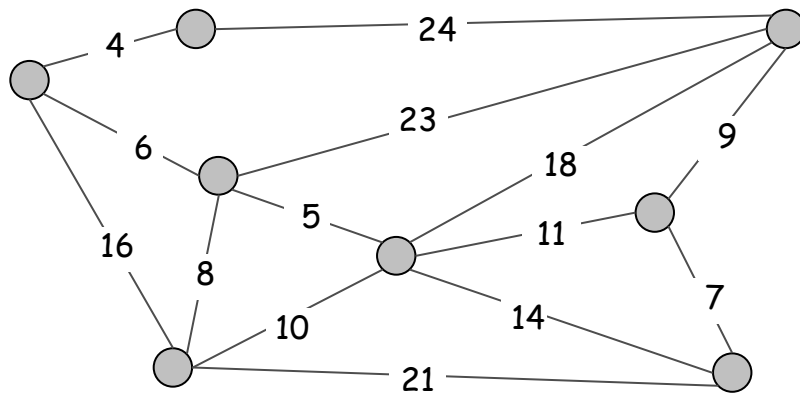
<sup>†</sup> Individual ops are amortized bounds

## 4.5 Minimum Spanning Tree

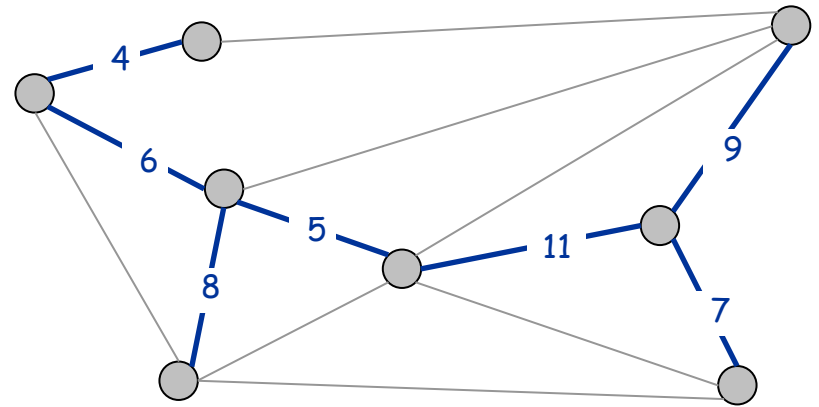
---

# 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

# Applications

MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.



# 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  $T$ .

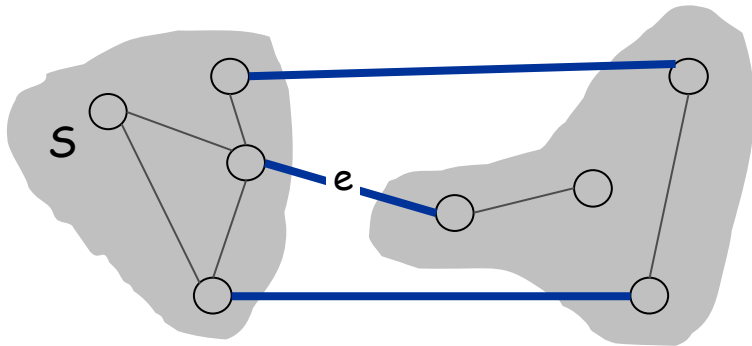
**Remark.** All three algorithms produce an MST.

# Greedy Algorithms

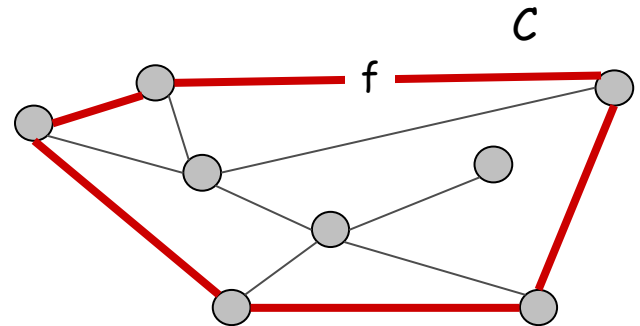
**Simplifying assumption.** All edge costs  $c_e$  are distinct.

**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  $f$ .



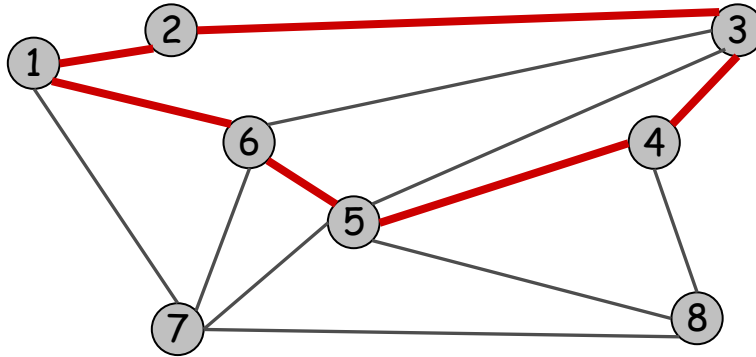
$e$  is in the MST



$f$  is not in the MST

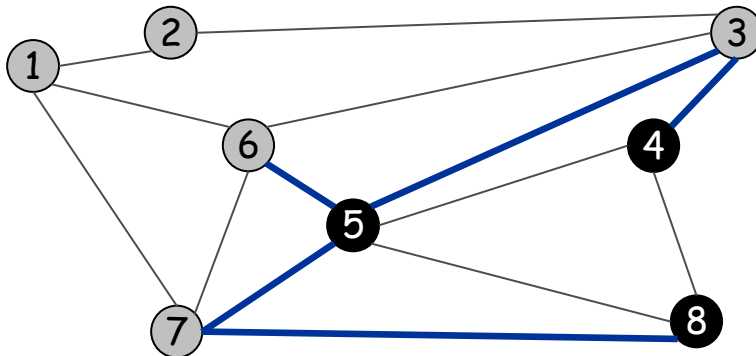
# Cycles and Cuts

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



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

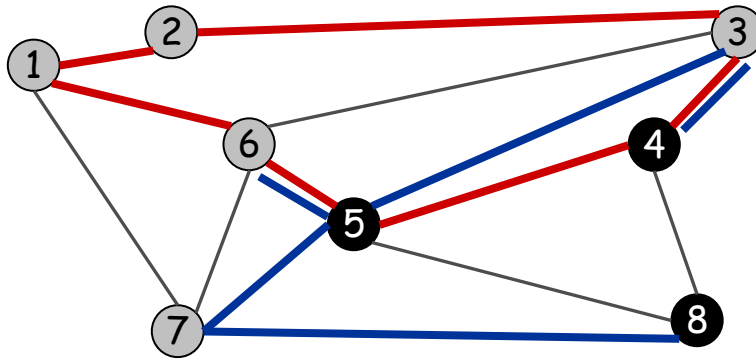
**Cutset.** A cut is a subset of nodes  $S$ . The corresponding cutset  $D$  is the subset of edges with exactly one endpoint in  $S$ .



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

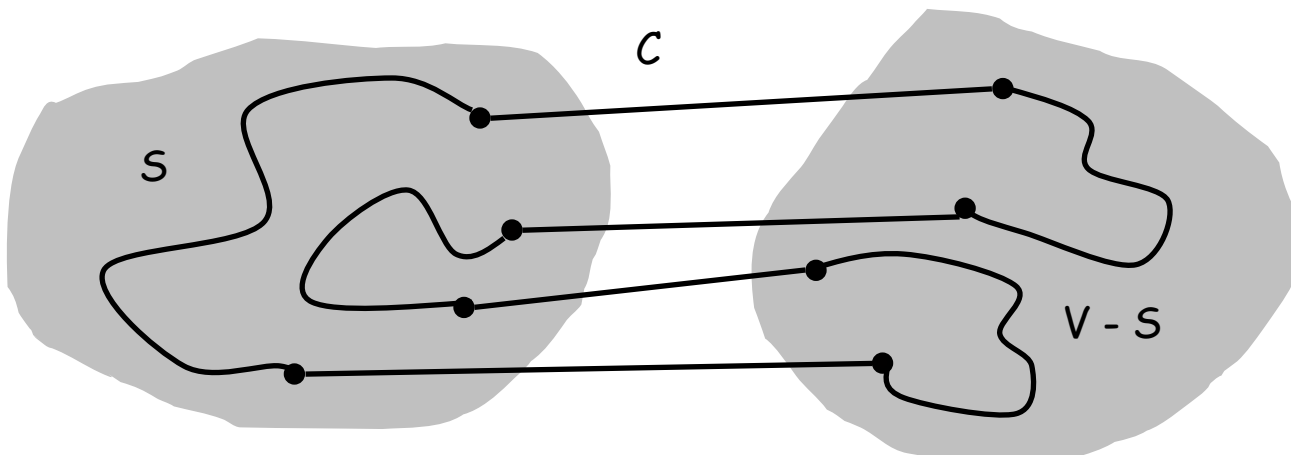
# Cycle-Cut Intersection

**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)



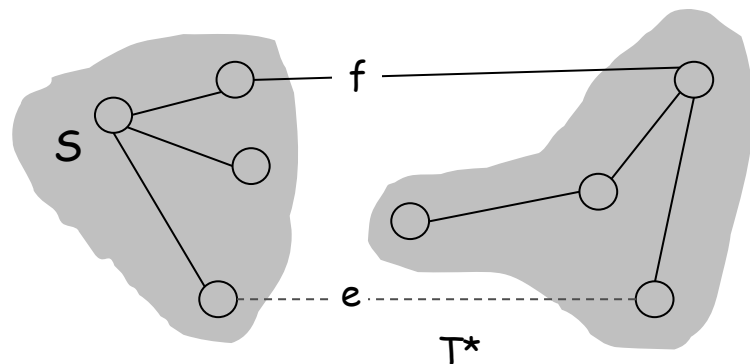
# Greedy Algorithms

**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. ▀



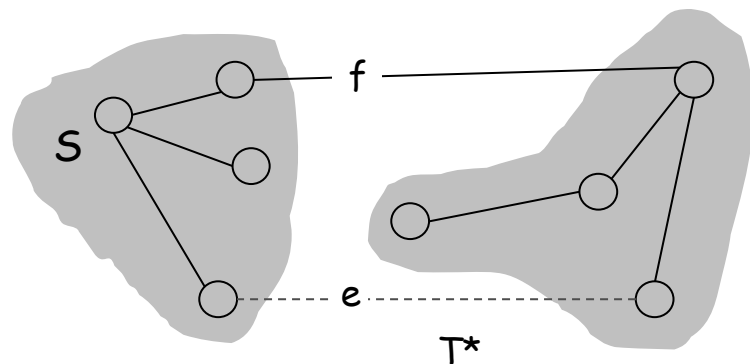
# Greedy Algorithms

**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)

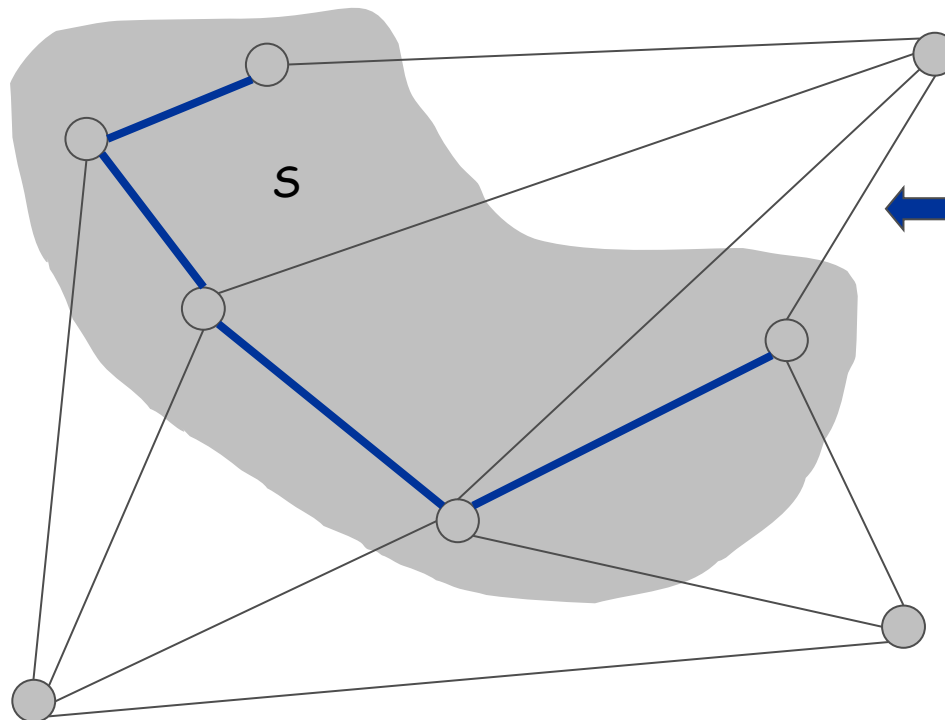
- Suppose  $f$  belongs to  $T^*$ , and let's see what happens.
- Deleting  $f$  from  $T^*$  creates a cut  $S$  in  $T^*$ .
- Edge  $f$  is both in the cycle  $C$  and in the cutset  $D$  corresponding to  $S$   
 $\Rightarrow$  there exists another edge, say  $e$ , that is in both  $C$  and  $D$ .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



# Prim's Algorithm: Proof of Correctness

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

- Initialize  $S$  = any node.
- Apply cut property to  $S$ .
- Add min cost edge in cutset corresponding to  $S$  to  $T$ , and add one new explored node  $u$  to  $S$ .



# 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  $a[v]$  = cost of cheapest edge  $v$  to a node in  $S$ .
- $O(n^2)$  with an array;  $O(m \log n)$  with a binary heap.

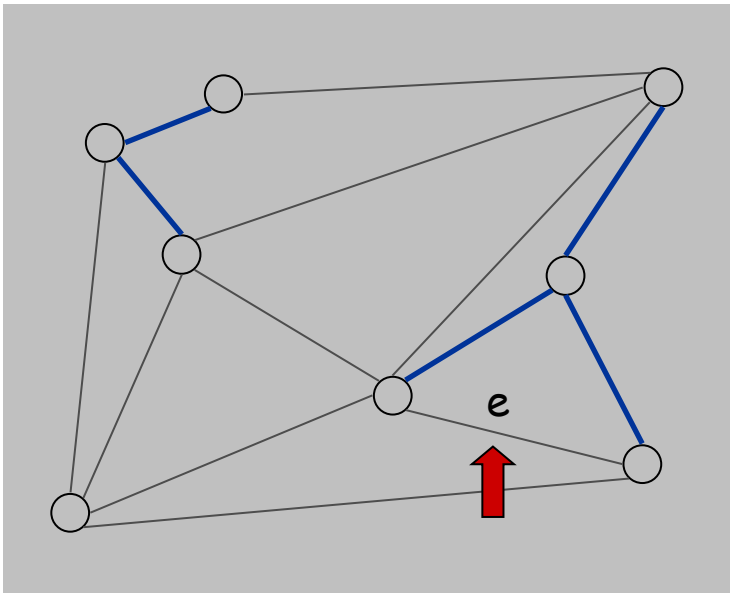
```
Prim(G, c) {  
    foreach (v ∈ V) a[v] ← ∞  
    Initialize an empty priority queue Q  
    foreach (v ∈ V) insert v onto Q  
    Initialize set of explored nodes S ← ∅  
  
    while (Q is not empty) {  
        u ← delete min element from Q  
        S ← S ∪ { u }  
        foreach (edge e = (u,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



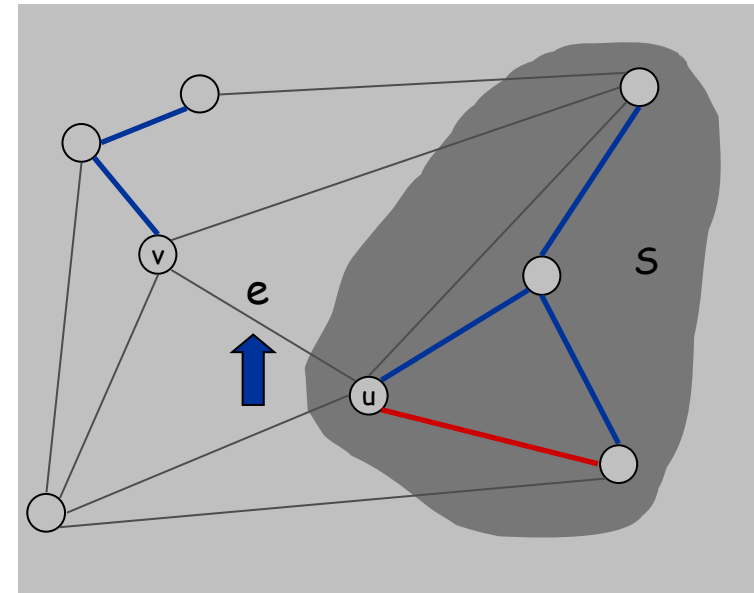
# Kruskal's Algorithm: Proof of Correctness

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  $S$  = set of nodes in  $u$ 's connected component.



Case 1



Case 2

# Implementation: Kruskal's Algorithm

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

- Build set  $T$  of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$  for sorting and  $O(m \underbrace{\alpha(m, n)}_{\text{essentially a constant}})$  for union-find.

$\swarrow$   $m \leq n^2 \Rightarrow \log m$  is  $O(\log n)$        $\underbrace{\hspace{1.5cm}}$  essentially a constant

```
Kruskal(G, c) {  
    Sort edges weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .  
     $T \leftarrow \phi$   
  
    foreach ( $u \in V$ ) make a set containing singleton  $u$   
  
    for  $i = 1$  to  $m$       are  $u$  and  $v$  in different connected components?  
         $(u, v) = e_i$        $\swarrow$   
        if ( $u$  and  $v$  are in different sets) {  
             $T \leftarrow T \cup \{e_i\}$   
            merge the sets containing  $u$  and  $v$   
        }  
         $\swarrow$  merge two components  
    return  $T$   
}
```