

# Chapter 4

## Greedy Algorithms



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

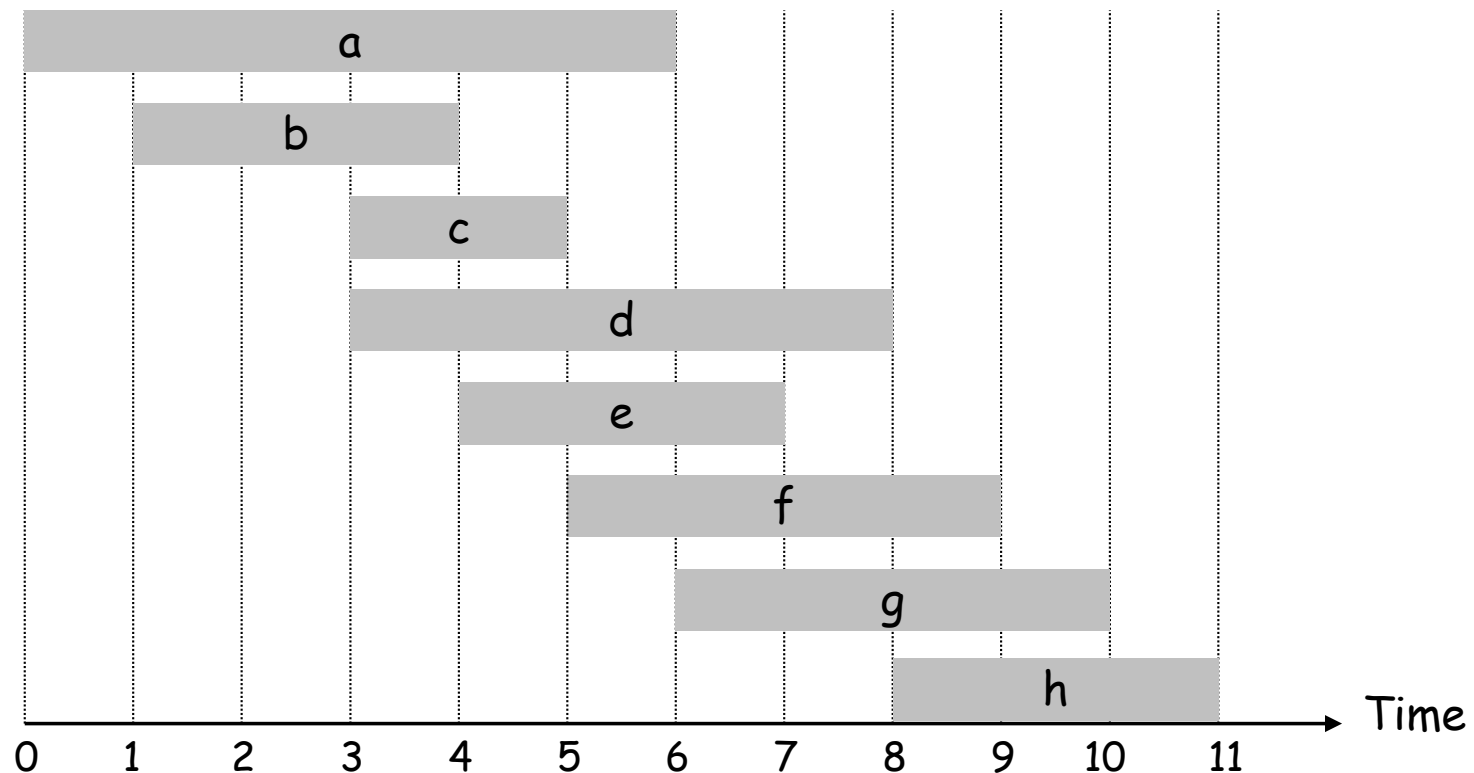
## 4.1 Interval Scheduling

---

# 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

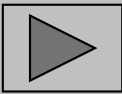
???

**Quiz:** Can you break earliest finish time?

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

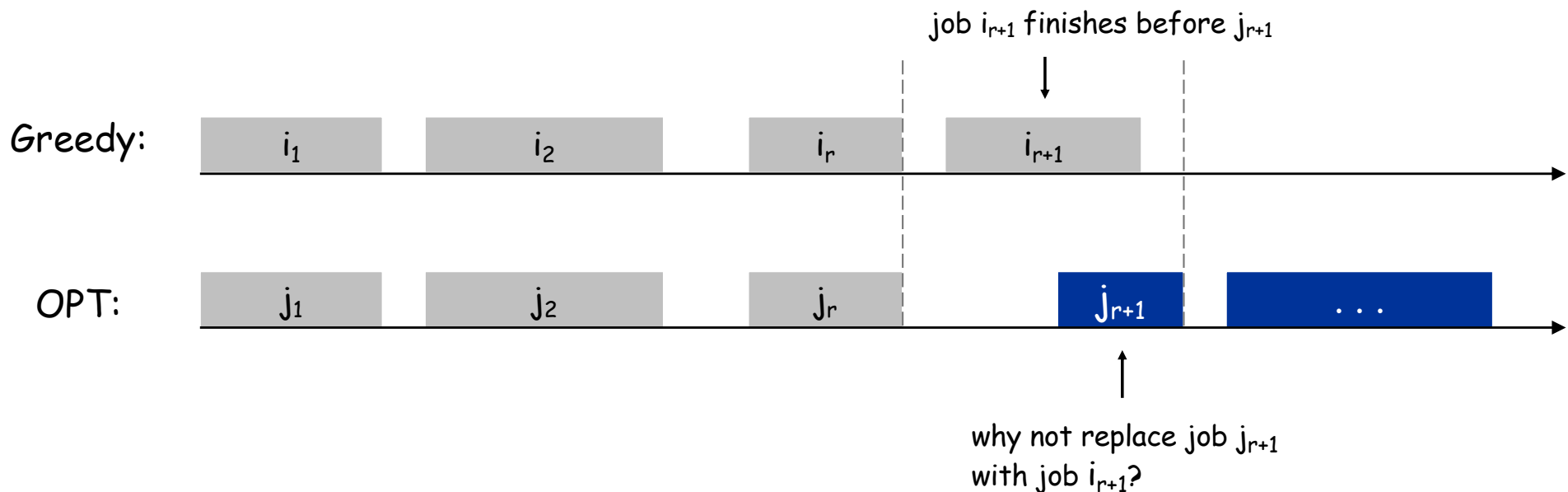
- Remember job  $j^*$  that was added last to A.
- Job j is compatible with A if  $s_j \geq f_{j^*}$ .

# 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 an optimal solution with  $i_1 = j_1, i_2 = j_2, \dots, i_r = j_r$  for the *largest possible value of  $r$* .

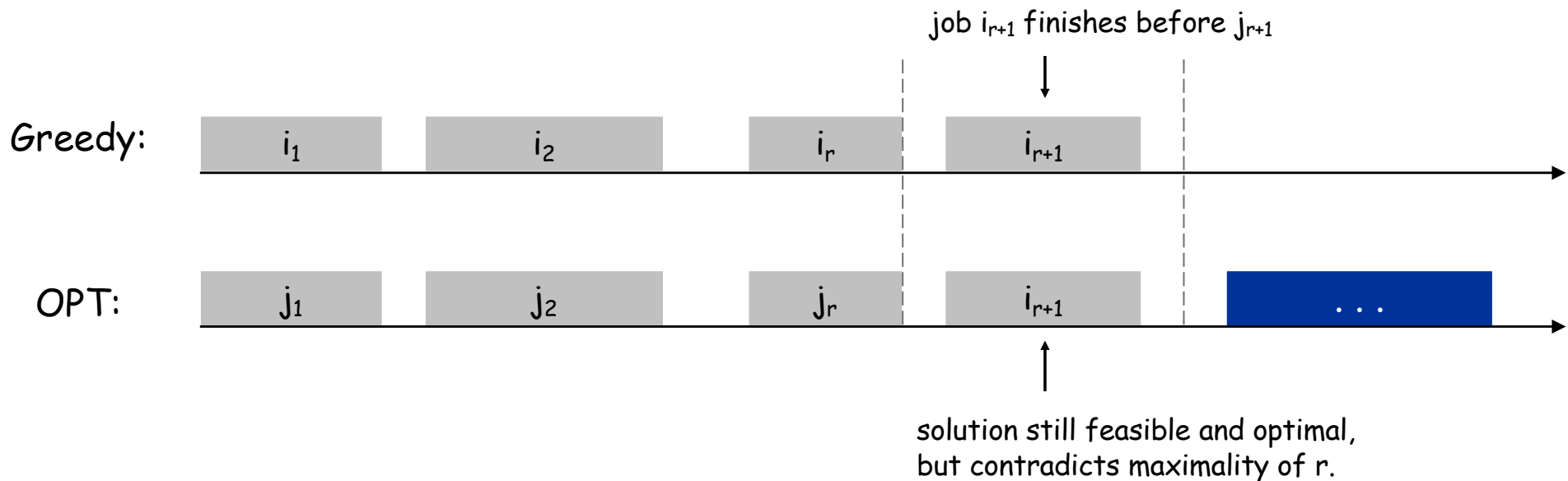


# Interval Scheduling: Analysis

**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

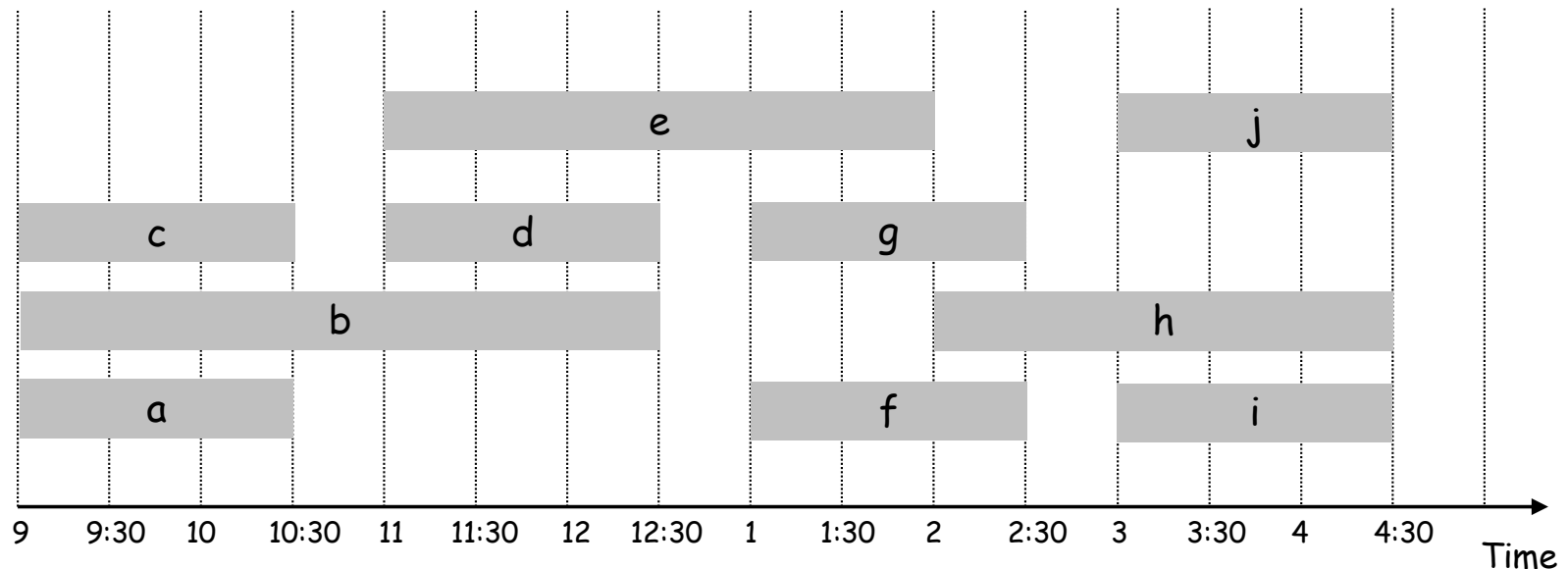
---

# 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.

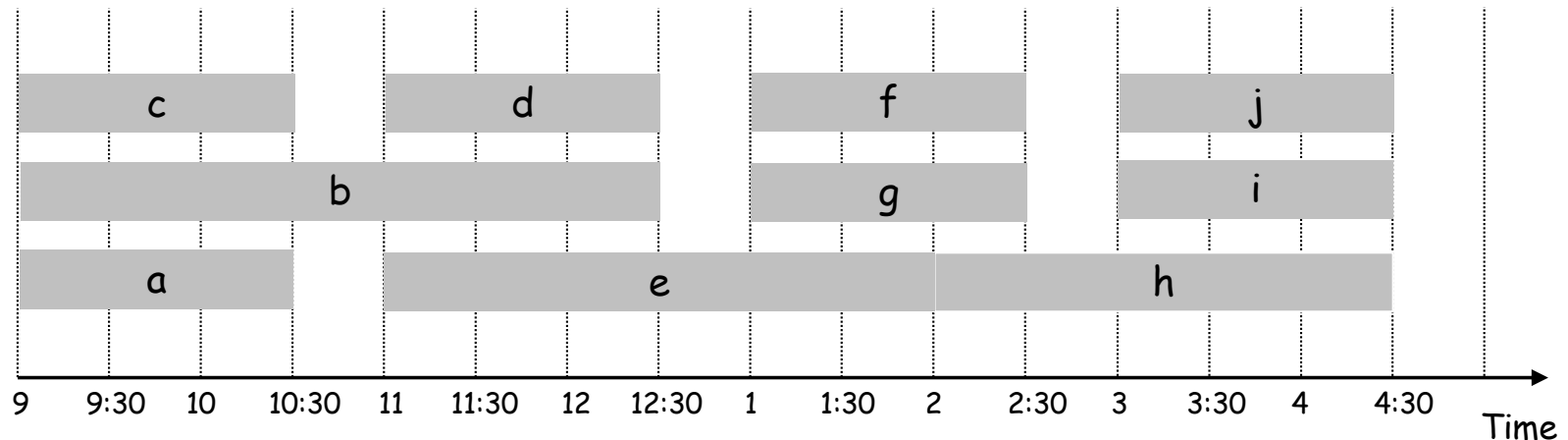


# Interval Partitioning

## 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

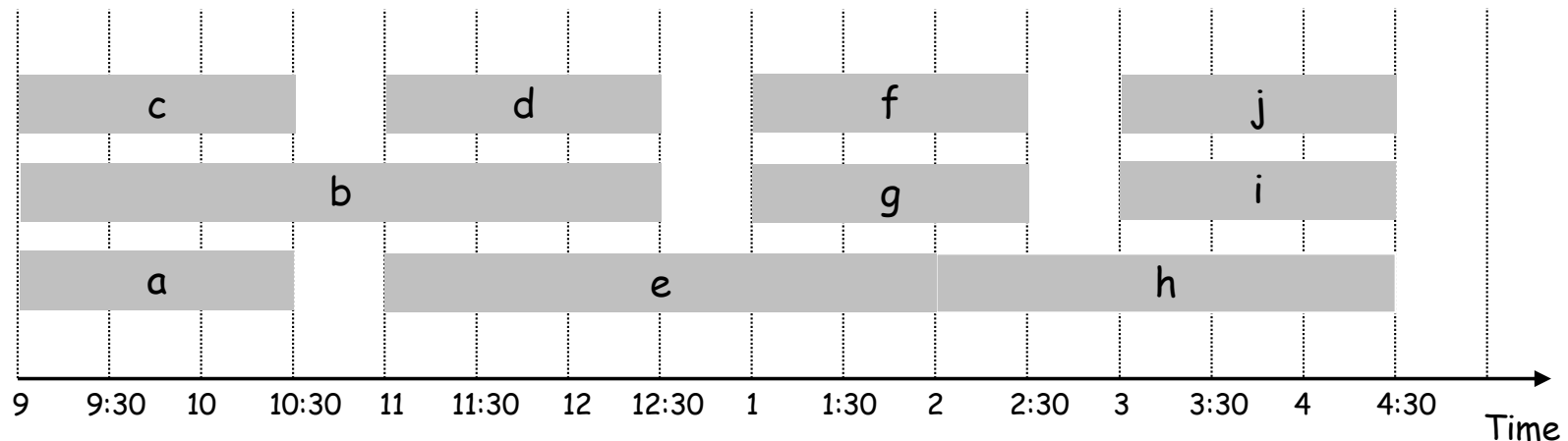
**Def.** The **depth** of a set of open intervals is the maximum number that contain any given time.

**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. ▪

## 4.2 Scheduling to Minimize Lateness

---

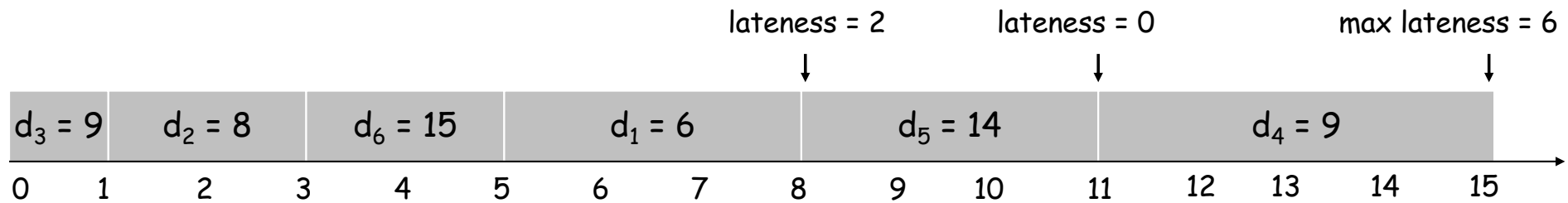
# Scheduling to Minimizing Lateness

## Minimizing lateness problem.

- Single resource processes one job at a time.
- Job  $j$  requires  $t_j$  units of processing time and is due at time  $d_j$ .
- If  $j$  starts at time  $s_j$ , it finishes at time  $f_j = s_j + t_j$ .
- Lateness:  $\ell_j = \max \{ 0, f_j - d_j \}$ .
- Goal: schedule all jobs to minimize **maximum** lateness  $L = \max \ell_j$ .

Ex:

	1	2	3	4	5	6
$t_j$	3	2	1	4	3	2
$d_j$	6	8	9	9	14	15





## Minimizing Lateness: Greedy Algorithms

*Greedy template.* Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time  $t_j$ .
- [Earliest deadline first] Consider jobs in ascending order of deadline  $d_j$ .
- [Smallest slack] Consider jobs in ascending order of slack  $d_j - t_j$ .

# Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time  $t_j$ .

	1	2
$t_j$	1	10
$d_j$	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack  $d_j - t_j$ .

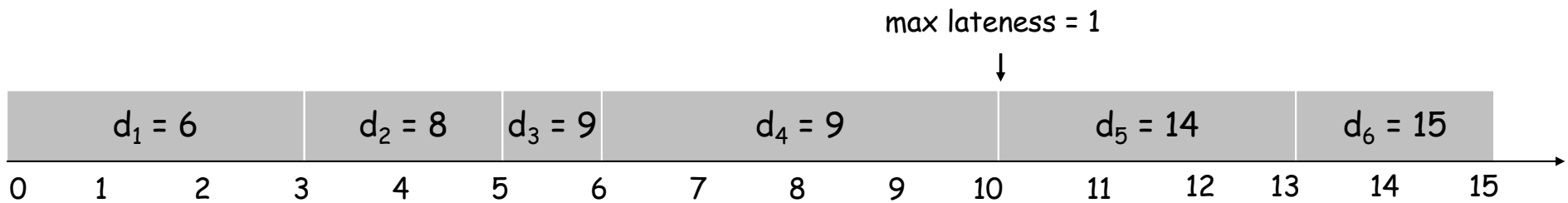
	1	2
$t_j$	1	10
$d_j$	2	10

counterexample

# Minimizing Lateness: Greedy Algorithm

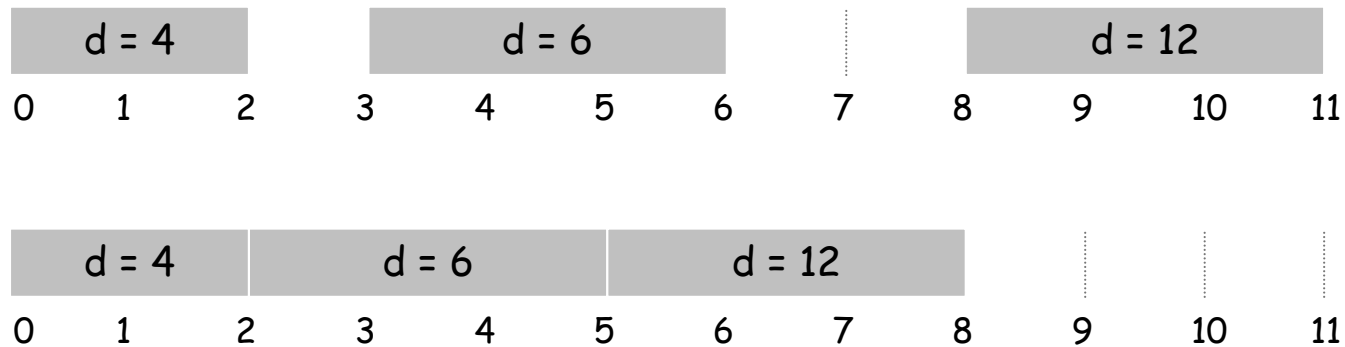
Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that  $d_1 \leq d_2 \leq \dots \leq d_n$   
  
t  $\leftarrow$  0  
for j = 1 to n  
    Assign job j to interval [t, t + tj]  
    sj  $\leftarrow$  t, fj  $\leftarrow$  t + tj  
    t  $\leftarrow$  t + tj  
output intervals [sj, fj]
```



## Minimizing Lateness: No Idle Time

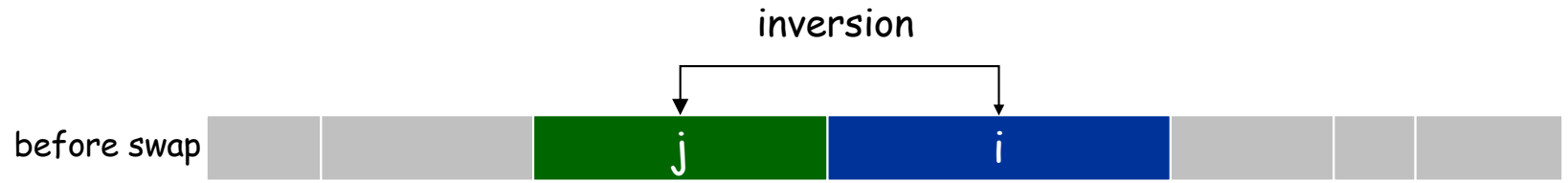
**Observation.** There exists an optimal schedule with no **idle time**.



**Observation.** The greedy schedule has no idle time.

## Minimizing Lateness: Inversions

**Def.** An **inversion** in schedule  $S$  is a pair of jobs  $i$  and  $j$  such that:  $(\text{deadline of } i) < (\text{deadline of } j)$  but  $j$  scheduled before  $i$ .

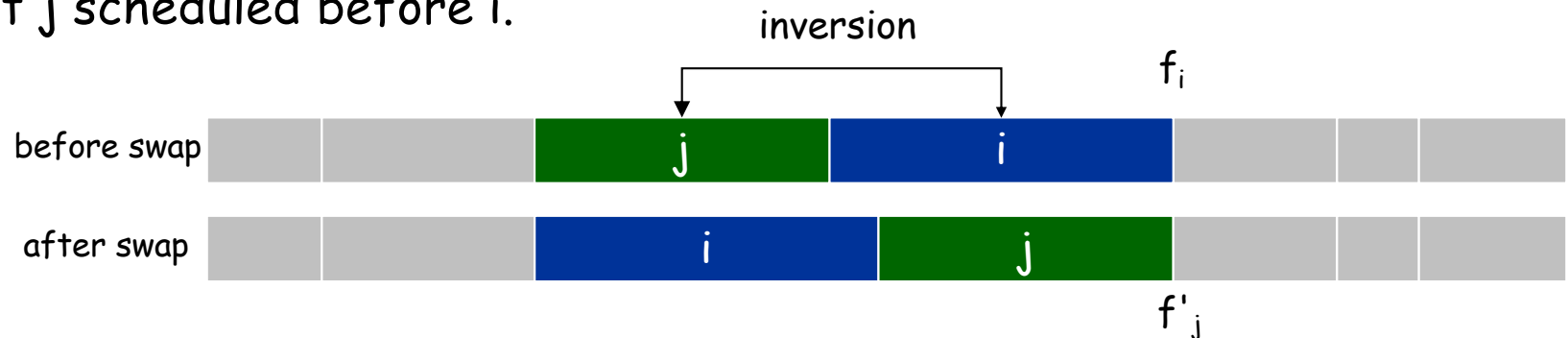


**Observation.** Greedy schedule has no inversions.

**Observation.** If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

## Minimizing Lateness: Inversions

**Def.** An **inversion** in schedule  $S$  is a pair of jobs  $i$  and  $j$  such that:  $i < j$  but  $j$  scheduled before  $i$ .



**Claim.** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

**Pf.** Let  $\ell$  be the lateness before the swap, and let  $\ell'$  be it afterwards.

- $\ell'_k = \ell_k$  for all  $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job  $j$  is late:

$$\begin{aligned}
 \ell'_j &= f'_j - d_j && \text{(definition)} \\
 &= f_i - d_j && (j \text{ finishes at time } f_i) \\
 &\leq f_i - d_i && (i < j) \\
 &\leq \ell_i && \text{(definition)}
 \end{aligned}$$

## Minimizing Lateness: Analysis of Greedy Algorithm

**Theorem.** Greedy schedule  $S$  is optimal.

**Pf.** Define  $S^*$  to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume  $S^*$  has no idle time.
- If  $S^*$  has no inversions, then  $S = S^*$ .
- If  $S^*$  has an inversion, let  $i$ - $j$  be an adjacent inversion.
  - swapping  $i$  and  $j$  does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of  $S^*$  .

## 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.



## 4.3 Optimal Caching

---

# Optimal Offline Caching

## Caching.

- Cache with capacity to store  $k$  items.
- Sequence of  $m$  item requests  $d_1, d_2, \dots, d_m$ .
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

**Goal.** Eviction schedule that minimizes number of cache misses.

**Ex:**  $k = 2$ , initial cache =  $ab$ ,  
requests:  $a, b, c, b, c, a, a, b$ .

**Optimal eviction schedule:** 2 cache misses.

a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b
requests	cache	

## Optimal Offline Caching: Farthest-In-Future

**Farthest-in-future.** Evict item in the cache that is not requested until farthest in the future.

current cache: 

a	b	c	d	e	f
---	---	---	---	---	---

future queries:    g a b c e d a b b a c d e a f a d e f g h ...

↑  
cache miss

↑  
eject this one

**Theorem.** [Bellady, 1960s] FF is optimal eviction schedule.

**Pf.** Algorithm and theorem are intuitive; proof is subtle.

## Reduced Eviction Schedules

**Def.** A **reduced** schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

**Intuition.** Can transform an unreduced schedule into a reduced one with no more cache misses.

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	x	b
b	a	c	b
c	a	c	b
a	a	c	b
a	a	c	b

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

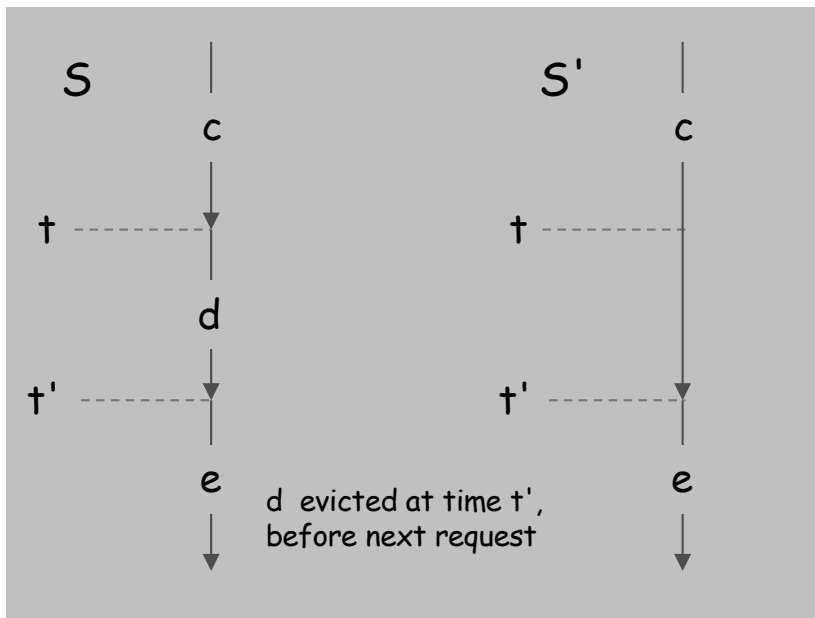
a reduced schedule

## Reduced Eviction Schedules

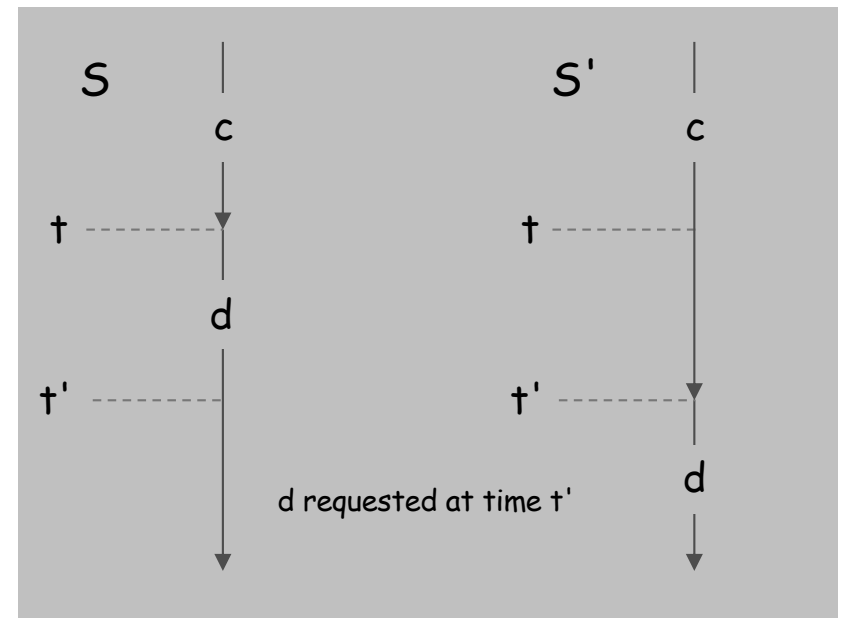
**Claim.** Given any unreduced schedule  $S$ , can transform it into a reduced schedule  $S'$  with no more cache misses.

**Pf.** (by induction on number of unreduced items) ← doesn't enter cache at requested time

- Suppose  $S$  brings  $d$  into the cache at time  $t$ , without a request.
- Let  $c$  be the item  $S$  evicts when it brings  $d$  into the cache.
- Case 1:  $d$  evicted at time  $t'$ , before next request for  $d$ .
- Case 2:  $d$  requested at time  $t'$  before  $d$  is evicted. ▪



Case 1



Case 2

## Farthest-In-Future: Analysis

**Theorem.** FF is optimal eviction algorithm.

**Pf.** (by induction on number of requests  $j$ )

Invariant: There exists an optimal reduced schedule  $S$  that makes the same eviction schedule as  $S_{FF}$  through the first  $j+1$  requests.

Let  $S$  be reduced schedule that satisfies invariant through  $j$  requests.

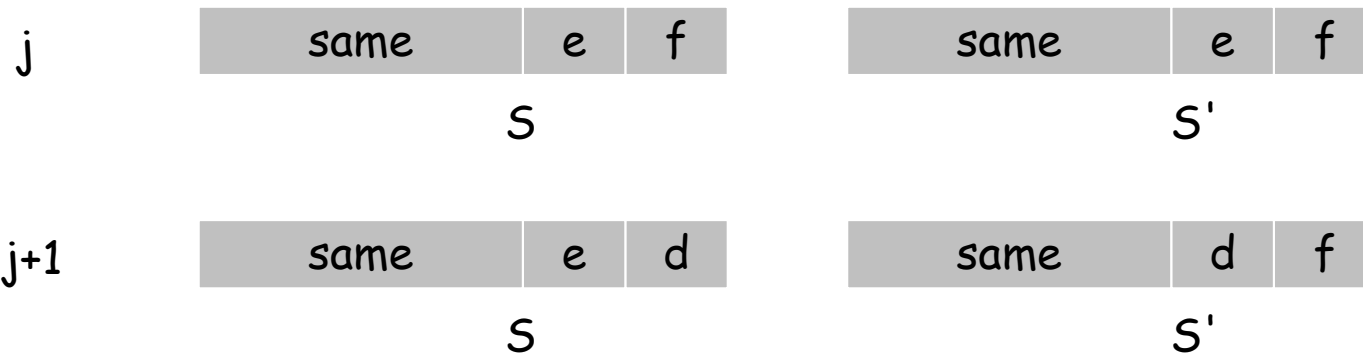
We produce  $S'$  that satisfies invariant after  $j+1$  requests.

- Consider  $(j+1)^{st}$  request  $d = d_{j+1}$ .
- Since  $S$  and  $S_{FF}$  have agreed up until now, they have the same cache contents before request  $j+1$ .
- Case 1: ( $d$  is already in the cache).  $S' = S$  satisfies invariant.
- Case 2: ( $d$  is not in the cache and  $S$  and  $S_{FF}$  evict the same element).  $S' = S$  satisfies invariant.

## Farthest-In-Future: Analysis

Pf. (continued)

- Case 3: ( $d$  is not in the cache;  $S_{FF}$  evicts  $e$ ;  $S$  evicts  $f \neq e$ ).
  - begin construction of  $S'$  from  $S$  by evicting  $e$  instead of  $f$

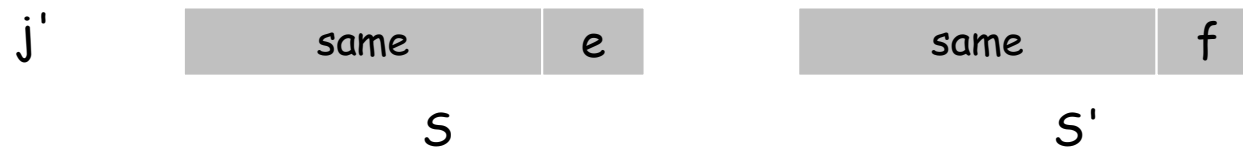


- now  $S'$  agrees with  $S_{FF}$  on first  $j+1$  requests; we show that having element  $f$  in cache is no worse than having element  $e$

## Farthest-In-Future: Analysis

Let  $j'$  be the **first** time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$ .

↑  
must involve  $e$  or  $f$  (or both)



- Case 3a:  $g = e$ . Can't happen with Farthest-In-Future since there must be a request for  $f$  before  $e$ .
- Case 3b:  $g = f$ . Element  $f$  can't be in cache of  $S$ , so let  $e'$  be the element that  $S$  evicts.
  - if  $e' = e$ ,  $S'$  accesses  $f$  from cache; now  $S$  and  $S'$  have same cache
  - if  $e' \neq e$ ,  $S'$  evicts  $e'$  and brings  $e$  into the cache; now  $S$  and  $S'$  have the same cache

↑

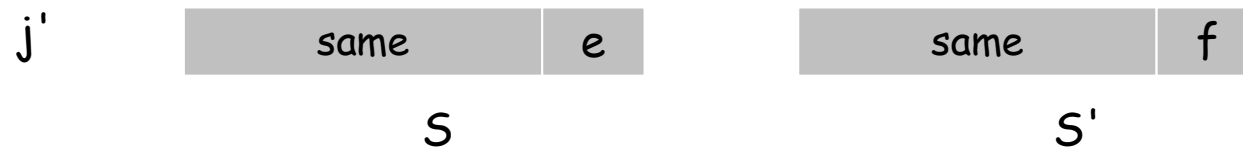
Note:  $S'$  is no longer reduced, but can be transformed into a reduced schedule that agrees with  $S_{FF}$  through step  $j+1$



## Farthest-In-Future: Analysis

Let  $j'$  be the **first** time after  $j+1$  that  $S$  and  $S'$  take a different action, and let  $g$  be item requested at time  $j'$ .

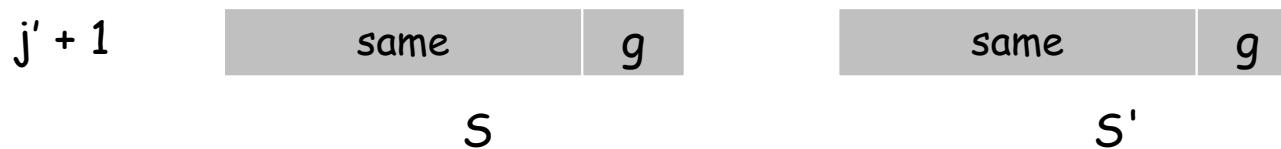
must involve  $e$  or  $f$  (or both)



otherwise  $S'$  would take the same action

- Case 3c:  $g \neq e, f$ .  $S$  must evict  $e$ .

Make  $S'$  evict  $f$ ; now  $S$  and  $S'$  have the same cache. ▀



# Caching Perspective

## Online vs. offline algorithms.

- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

**LIFO.** Evict page brought in most recently.

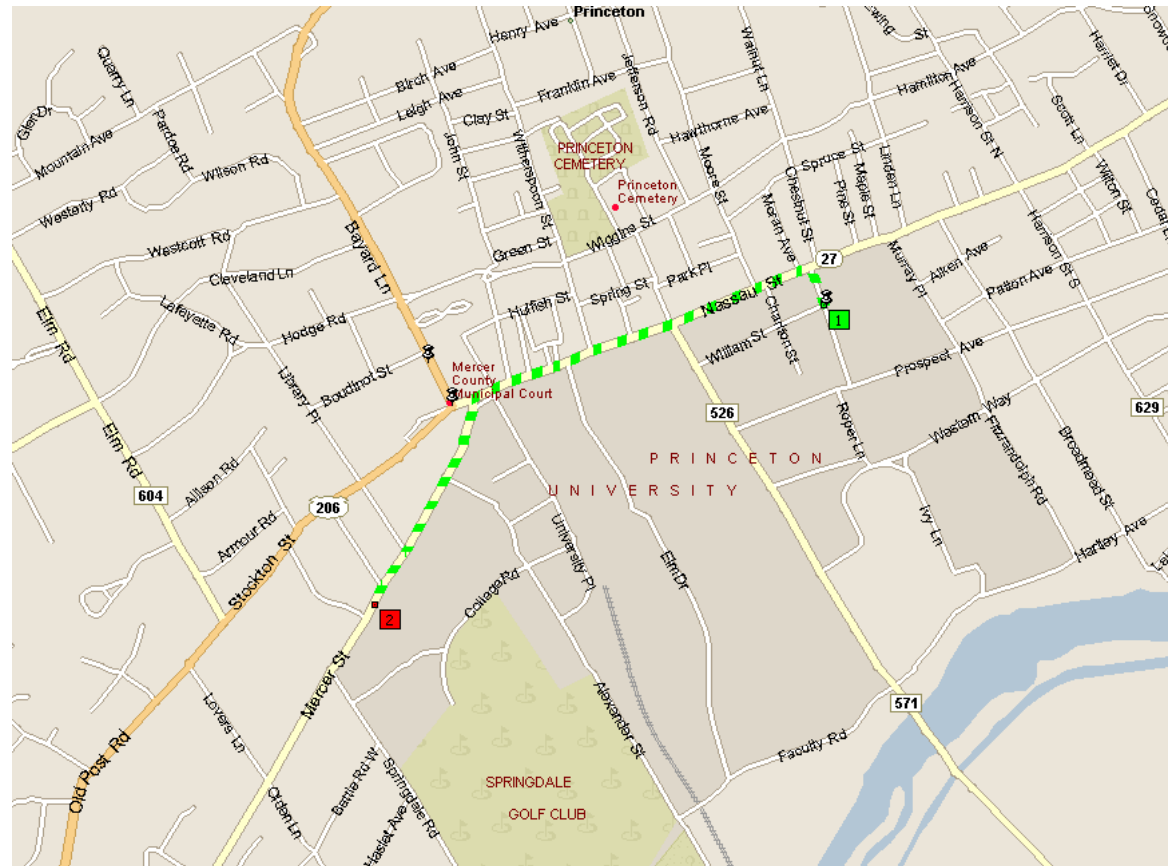
**LRU.** Evict page whose most recent access was earliest.

↑  
FF with direction of time reversed!

**Theorem.** FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is  $k$ -competitive. [Section 13.8]
- LIFO is arbitrarily bad.

## 4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

# Shortest Path Problem

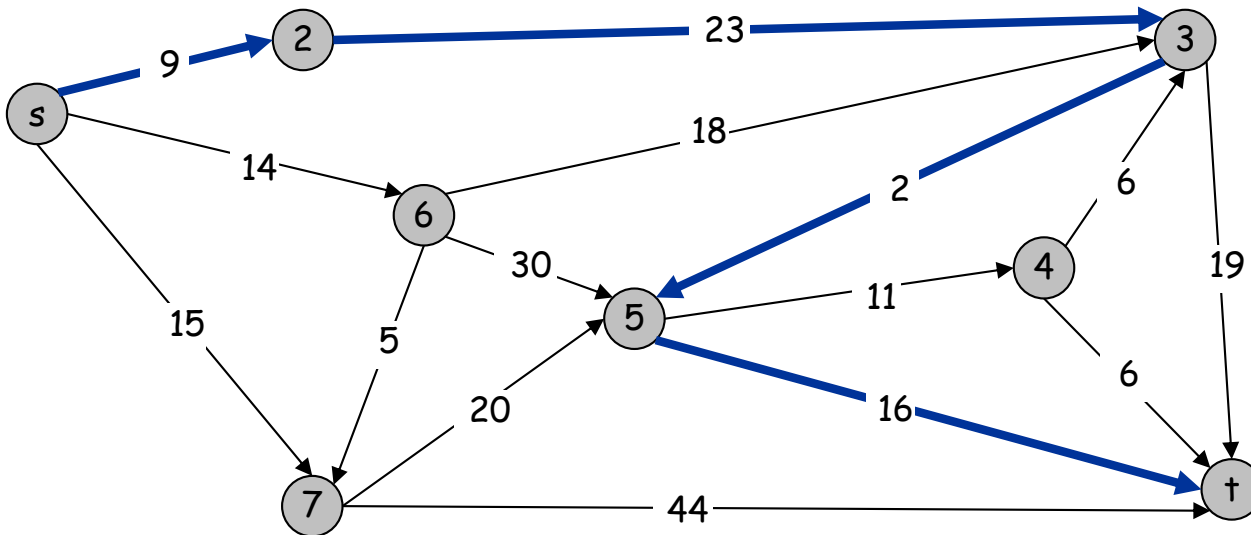
## Shortest path network.

- 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  $s-2-3-5-t$   
=  $9 + 23 + 2 + 16$   
= 50.

# Dijkstra's Algorithm

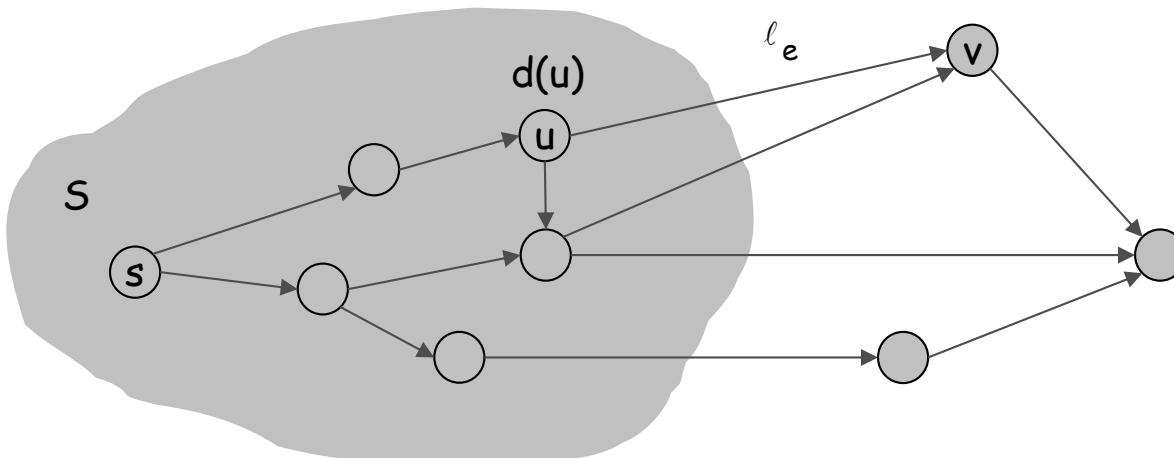
## Dijkstra's algorithm.

- 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

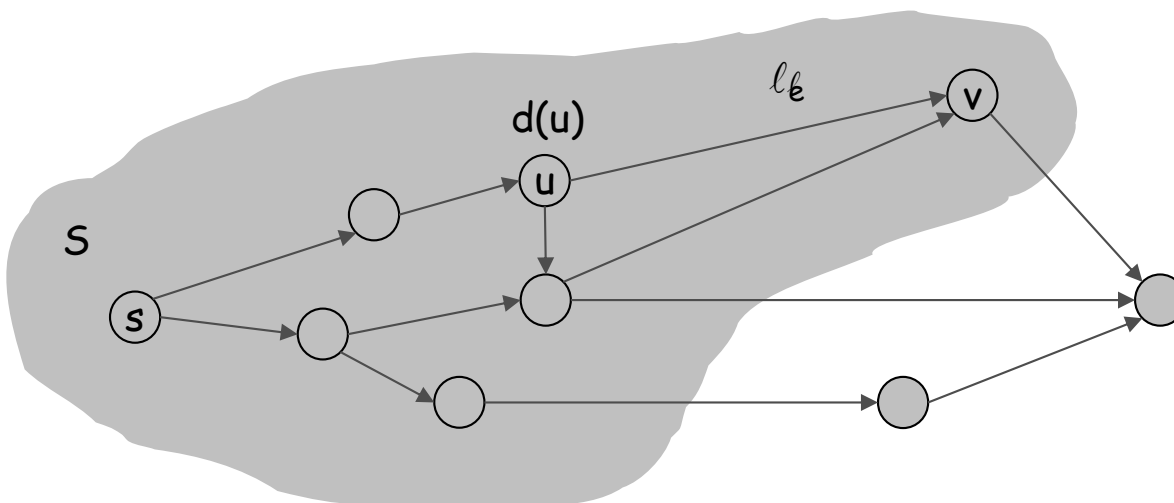
## Dijkstra's algorithm.

- 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

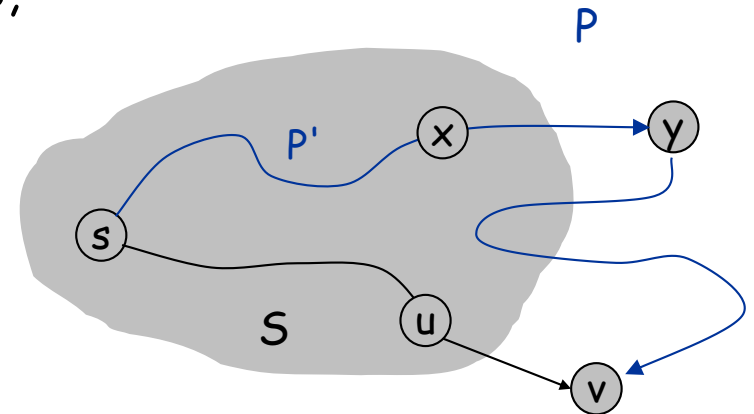
**Invariant.** For each node  $u \in S$ ,  $d(u)$  is the length of the shortest  $s$ - $u$  path.

**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  $S$ .



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

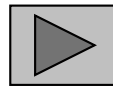
# Dijkstra's Algorithm: Implementation

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

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

$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

**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

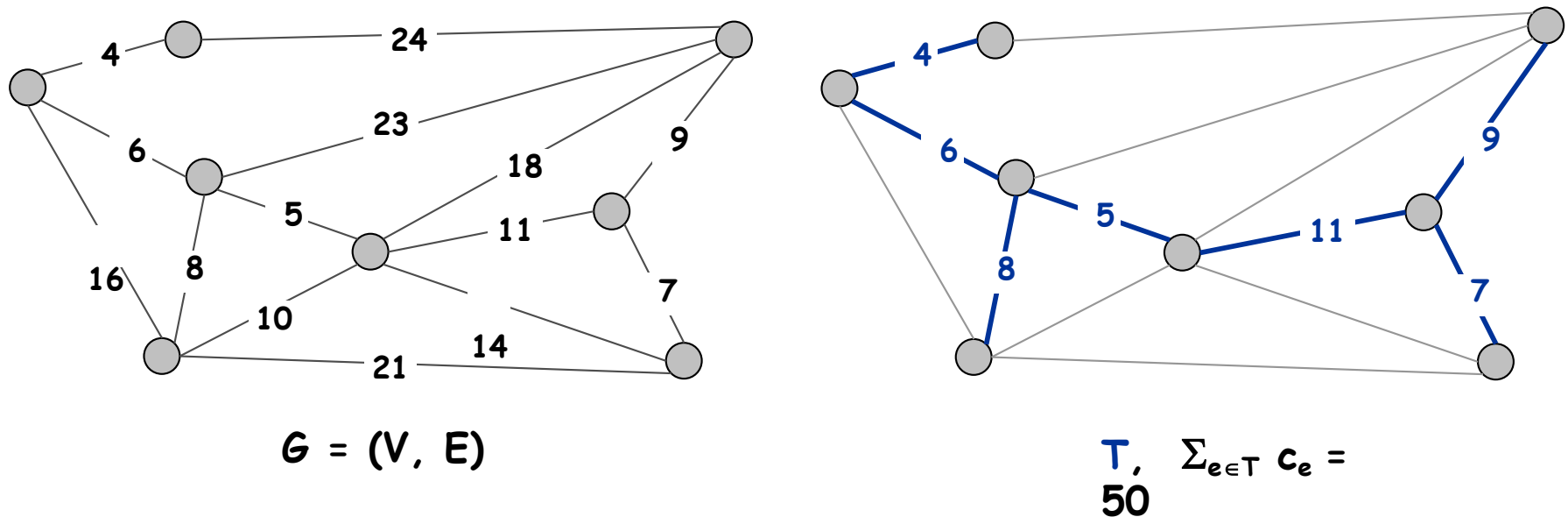


## 4.5 Minimum Spanning Tree

---

# 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.



**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  $T = \phi$ . 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  $T$ .

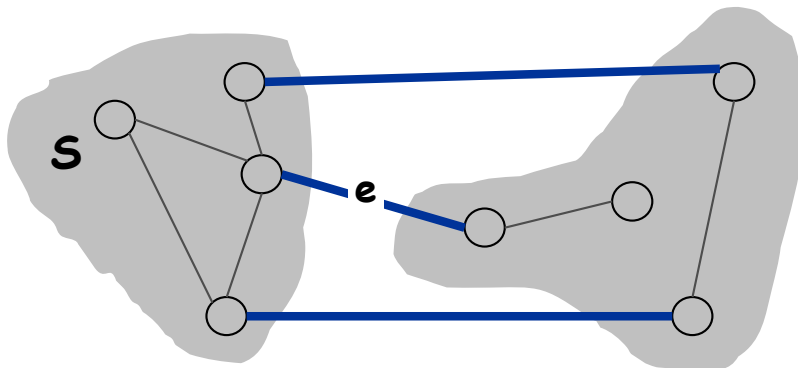
**Remark.** All three algorithms produce an MST.

## Greedy Algorithms

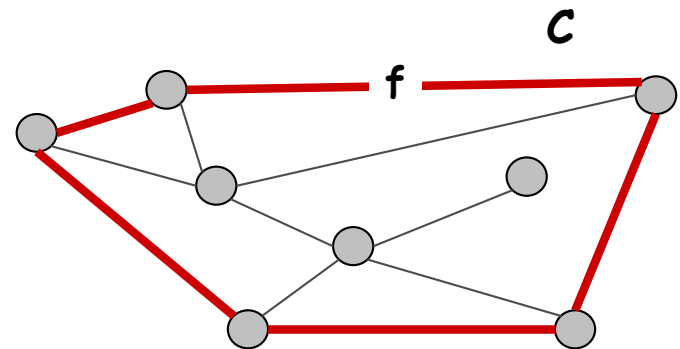
**Simplifying assumption.** All edge costs  $c_e$  are distinct.

**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  $f$ .



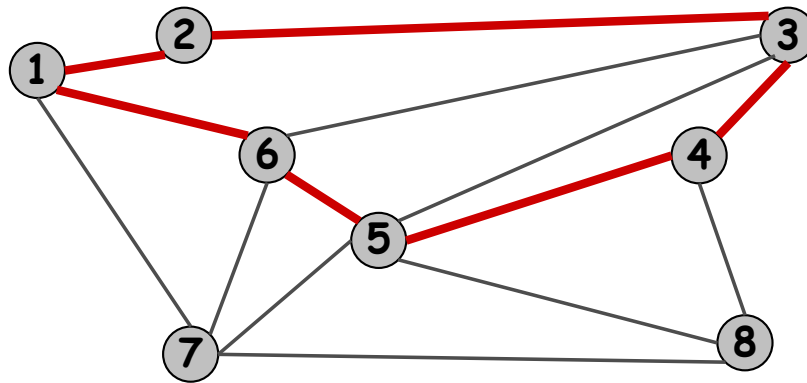
**$e$  is in the MST**



**$f$  is not in the MST**

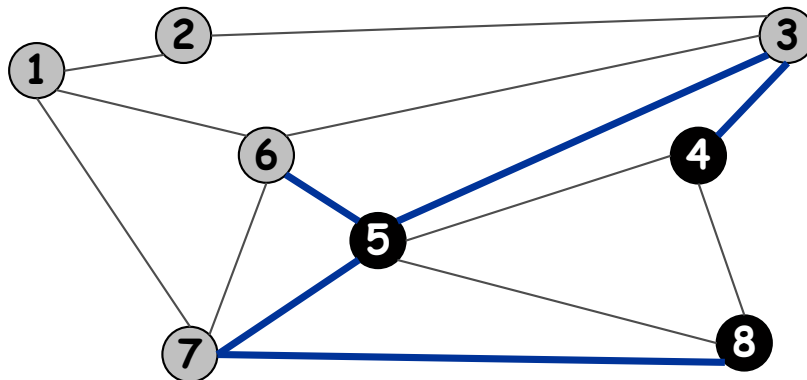
## Cycles and Cuts

**Cycle.** Set of edges of 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

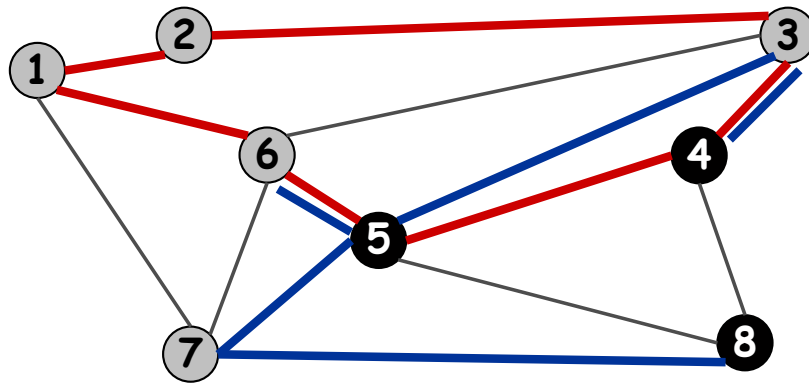
**Cutset.** A cut is a subset of nodes  $S$ . The corresponding cutset  $D$  is the subset of edges with exactly one endpoint in  $S$ .



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

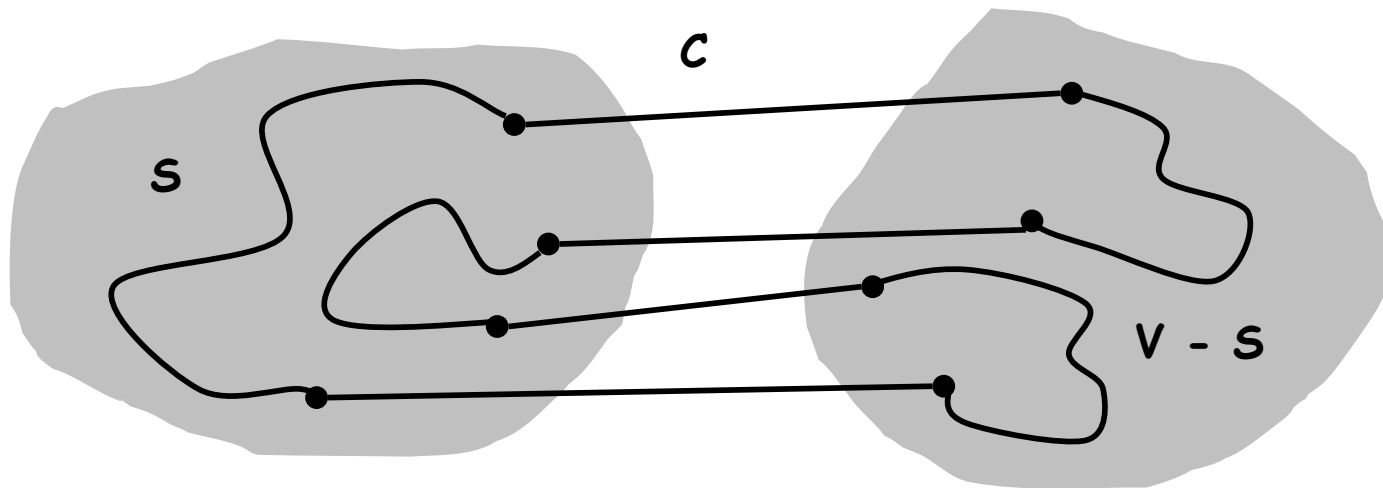
## Cycle-Cut Intersection

**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)



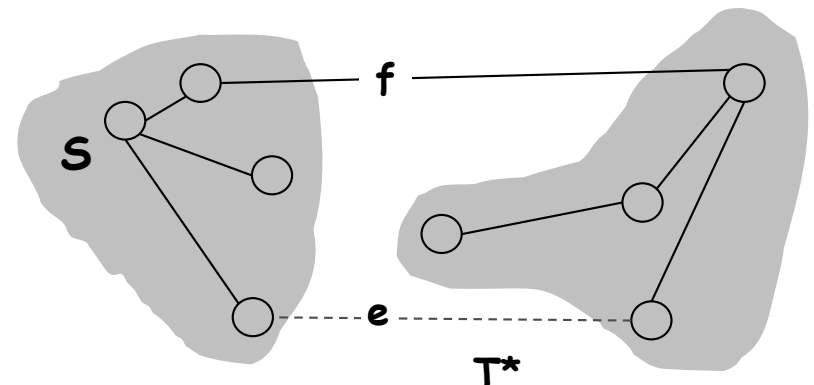
# Greedy Algorithms

**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. ■





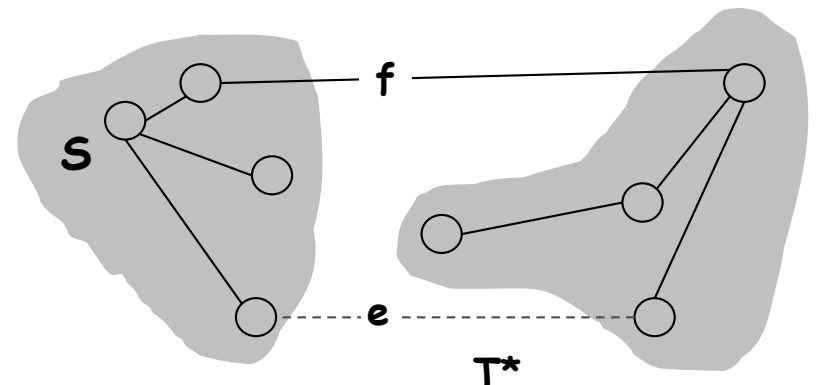
# Greedy Algorithms

**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)

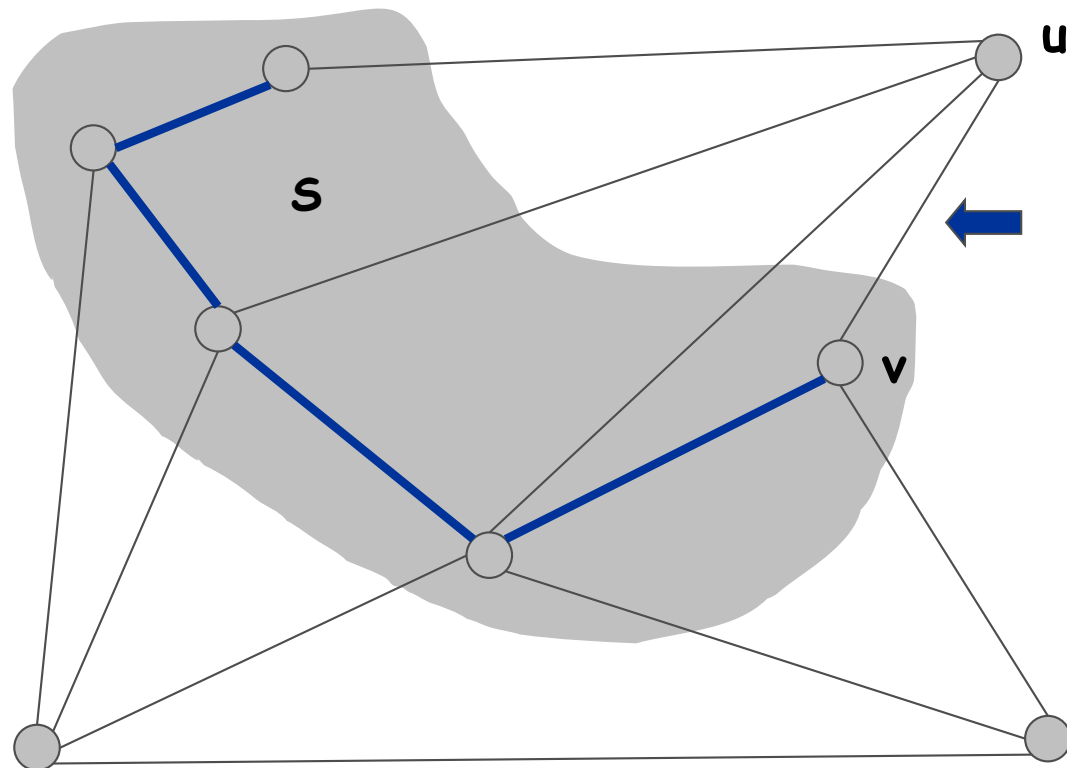
- Suppose  $f$  belongs to  $T^*$ , and let's see what happens.
- Deleting  $f$  from  $T^*$  creates a cut  $S$  in  $T^*$ .
- Edge  $f$  is both in the cycle  $C$  and in the cutset  $D$  corresponding to  $S$   
 $\Rightarrow$  there exists another edge, say  $e$ , that is in both  $C$  and  $D$ .
- $T' = T^* \cup \{e\} - \{f\}$  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  $S = \{\text{any node}\}$ .
- Apply cut property to  $S$ .
- Add min cost edge  $e = (v,u)$  in cutset corresponding to  $S$  to  $T$ , and add newly explored node  $u$  to  $S$ .



## Implementation: Prim's Algorithm

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

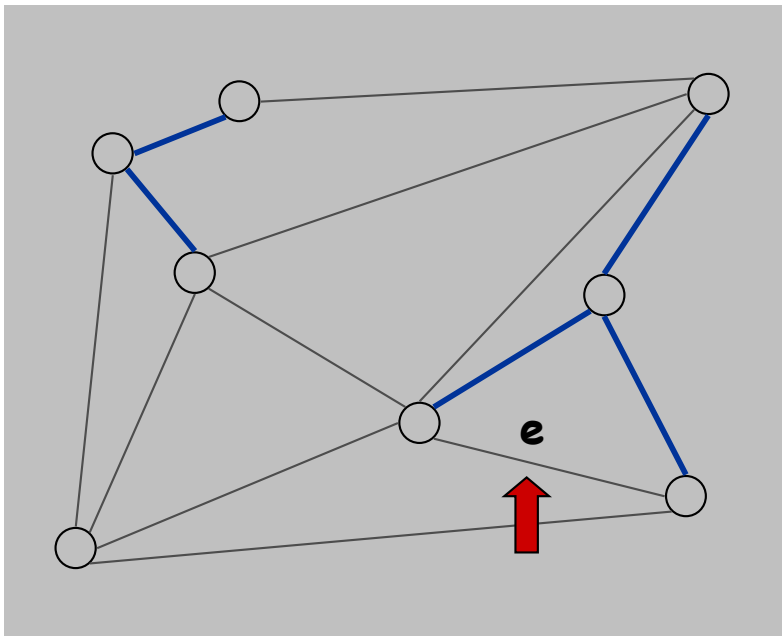
- Maintain set of explored nodes  $S$ .
- For each unexplored node  $v$ , maintain attachment cost  $a[v]$  = cost of cheapest edge from  $v$  to a node in  $S$ .
- $O(n^2)$  with an array;  $O(m \log n)$  with a binary heap.

```
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  
    }  
}
```

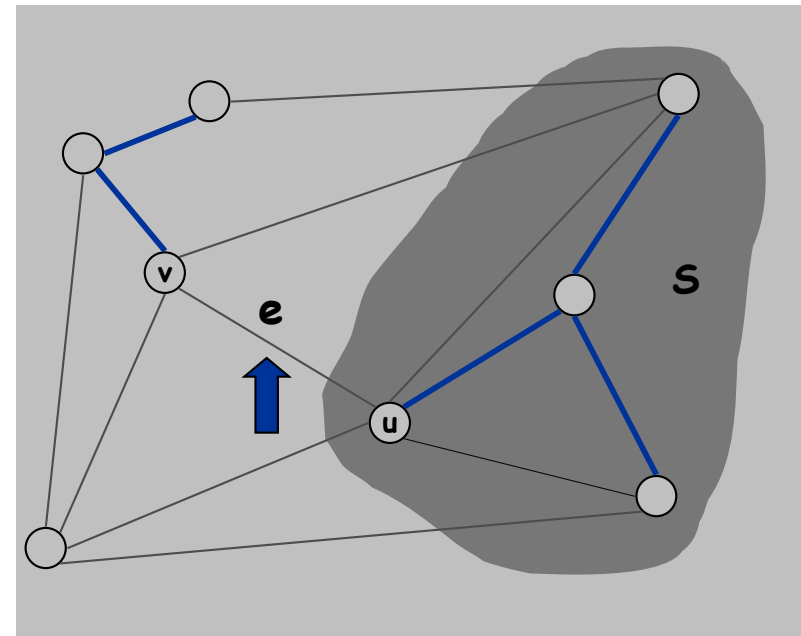
# Kruskal's Algorithm: Proof of Correctness

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.

$m \leq n^2 \Rightarrow \log m$  is  $O(\log n)$       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?  
         $(u, v) = e_i$       ↙  
        if ( $u$  and  $v$  are in different sets) {  
             $T \leftarrow T \cup \{e_i\}$   
            merge the sets containing  $u$  and  $v$   
        }      ↙ merge two components  
    return  $T$   
}
```

## Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

**Impact.** Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

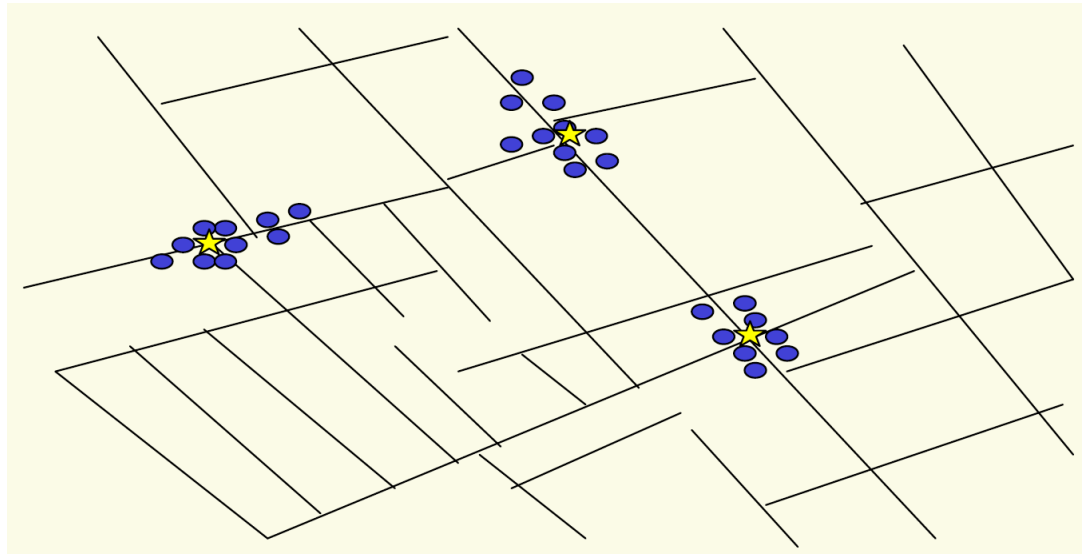
↑  
e.g., if all edge costs are integers,  
perturbing cost of edge  $e_i$  by  $i / n^2$

**Implementation.** Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```
boolean less(i, j) {  
    if      (cost(ei) < cost(ej)) return true  
    else if (cost(ei) > cost(ej)) return false  
    else if (i < j)                  return true  
    else                            return false  
}
```

## 4.7 Clustering

---



Outbreak of cholera deaths in London in 1850s.  
Reference: Nina Mishra, HP Labs

# Clustering

**Clustering.** Given a set  $U$  of  $n$  objects labeled  $p_1, \dots, p_n$ , classify into coherent groups.

↑  
photos, documents, micro-organisms

**Distance function.** Numeric value specifying "closeness" of two objects.

↑  
number of corresponding pixels whose  
intensities differ by some threshold

**Fundamental problem.** Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster  $10^9$  sky objects into stars, quasars, galaxies.



## Clustering of Maximum Spacing

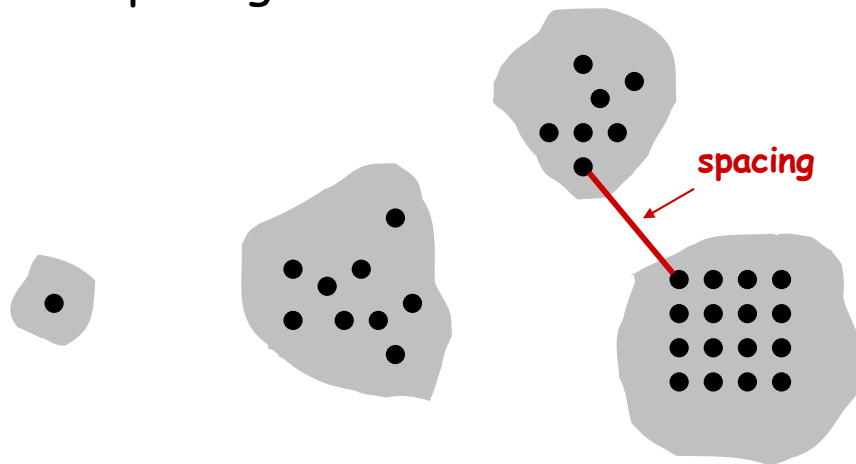
**k-clustering.** Divide objects into  $k$  non-empty groups.

**Distance function.** Assume it satisfies several natural properties.

- $d(p_i, p_j) = 0$  iff  $p_i = p_j$  (identity of indiscernibles)
- $d(p_i, p_j) \geq 0$  (nonnegativity)
- $d(p_i, p_j) = d(p_j, p_i)$  (symmetry)

**Spacing.** Min distance between any pair of points in different clusters.

**Clustering of maximum spacing.** Given an integer  $k$ , find a  $k$ -clustering of maximum spacing.



$k = 4$

# Greedy Clustering Algorithm

## Single-link k-clustering algorithm.

- Form a graph on the vertex set  $U$ , corresponding to  $n$  clusters.
- Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
- Repeat  $n-k$  times until there are exactly  $k$  clusters.

**Key observation.** This procedure is precisely Kruskal's algorithm (except we stop when there are  $k$  connected components).

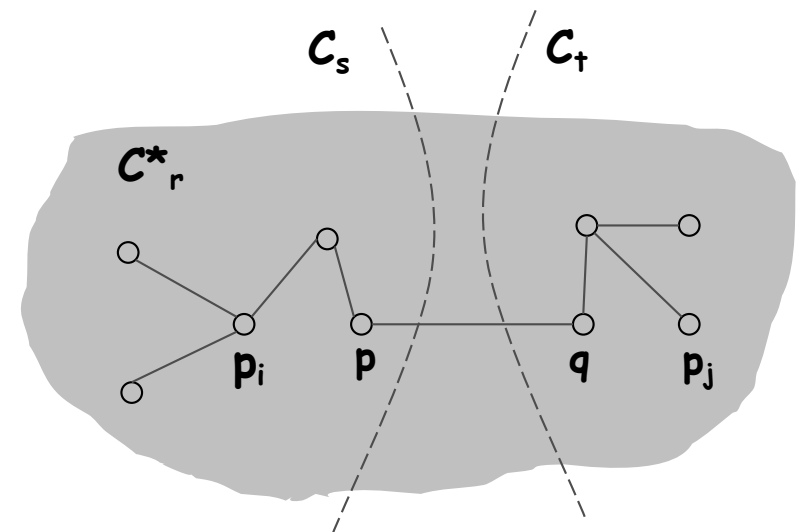
**Remark.** Equivalent to finding an MST and deleting the  $k-1$  most expensive edges.

## Greedy Clustering Algorithm: Analysis

**Theorem.** Let  $C^*$  denote the clustering  $C^*_1, \dots, C^*_k$  formed by deleting the  $k-1$  most expensive edges of a MST.  $C^*$  is a  $k$ -clustering of max spacing.

**Pf.** Let  $C$  denote some other clustering  $C_1, \dots, C_k$ .

- The spacing of  $C^*$  is the length  $d^*$  of the  $(k-1)^{\text{st}}$  most expensive edge.
- Let  $p_i, p_j$  be in the same cluster in  $C^*$ , say  $C^*_r$ , but different clusters in  $C$ , say  $C_s$  and  $C_t$ .
- Some edge  $(p, q)$  on  $p_i$ - $p_j$  path in  $C^*_r$  spans two different clusters in  $C$ .
- All edges on  $p_i$ - $p_j$  path have length  $\leq d^*$  since Kruskal chose them.
- Spacing of  $C$  is  $\leq d^*$  since  $p$  and  $q$  are in different clusters. ▀



# MST Algorithms: Theory

## Deterministic comparison based algorithms.

- $O(m \log n)$  [Jarník, Prim, Dijkstra, Kruskal, Boruvka]
- $O(m \log \log n)$ . [Cheriton-Tarjan 1976, Yao 1975]
- $O(m \beta(m, n))$ . [Fredman-Tarjan 1987]
- $O(m \log \beta(m, n))$ . [Gabow-Galil-Spencer-Tarjan 1986]
- $O(m \alpha(m, n))$ . [Chazelle 2000]

## Holy grail. $O(m)$ .

## Notable.

- $O(m)$  randomized. [Karger-Klein-Tarjan 1995]
- $O(m)$  verification. [Dixon-Rauch-Tarjan 1992]

## Euclidean.

- 2-d:  $O(n \log n)$ . compute MST of edges in Delaunay
- k-d:  $O(k n^2)$ . dense Prim