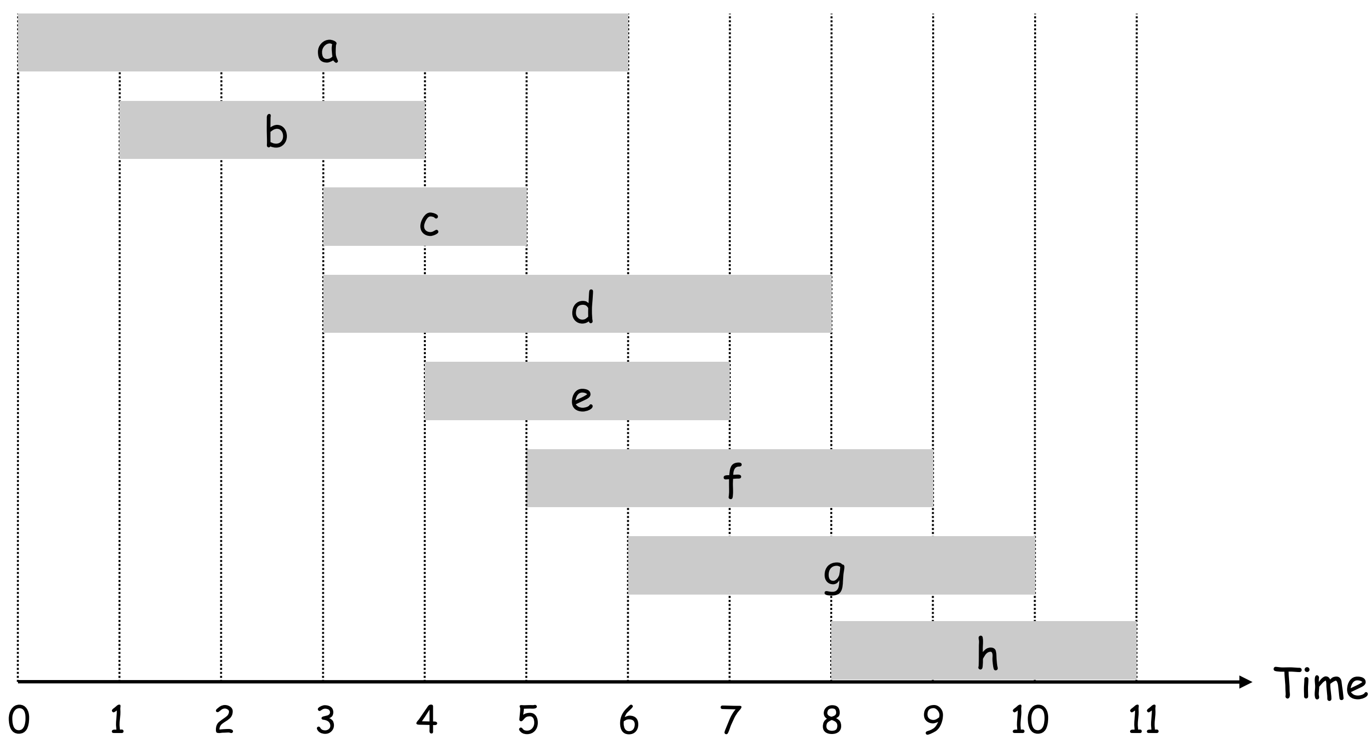
# 4.1 Interval Scheduling

# 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.

jobs selected

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

A ← ∅
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation. O(n log n).

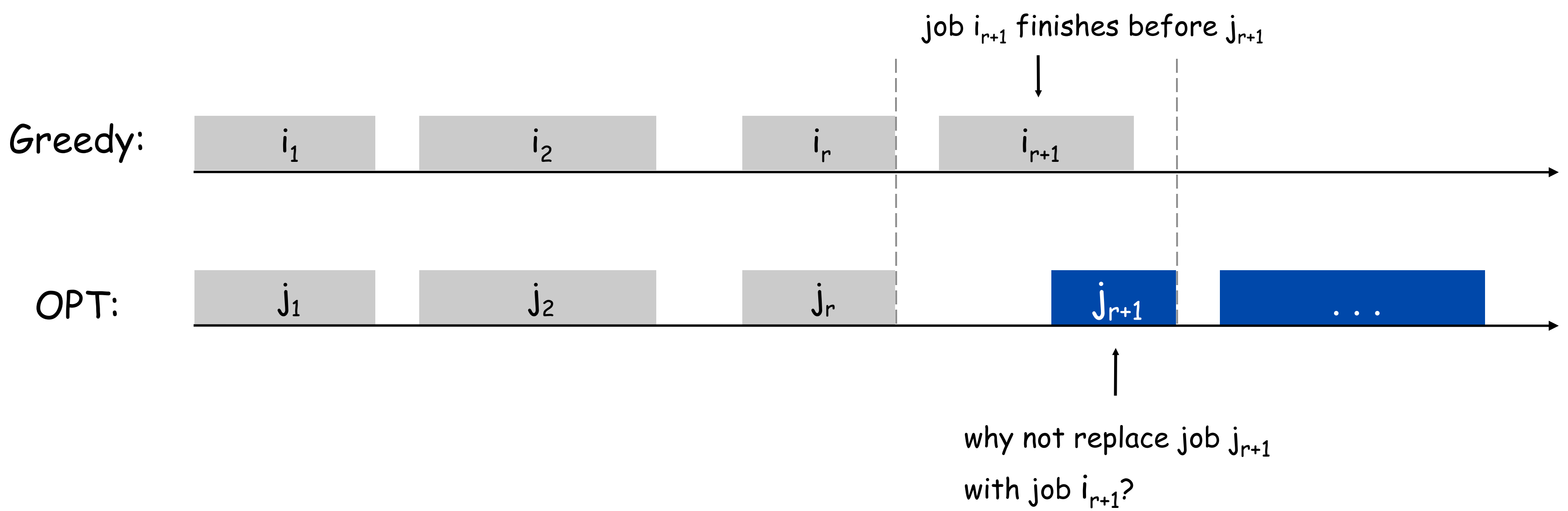
- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

# Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

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

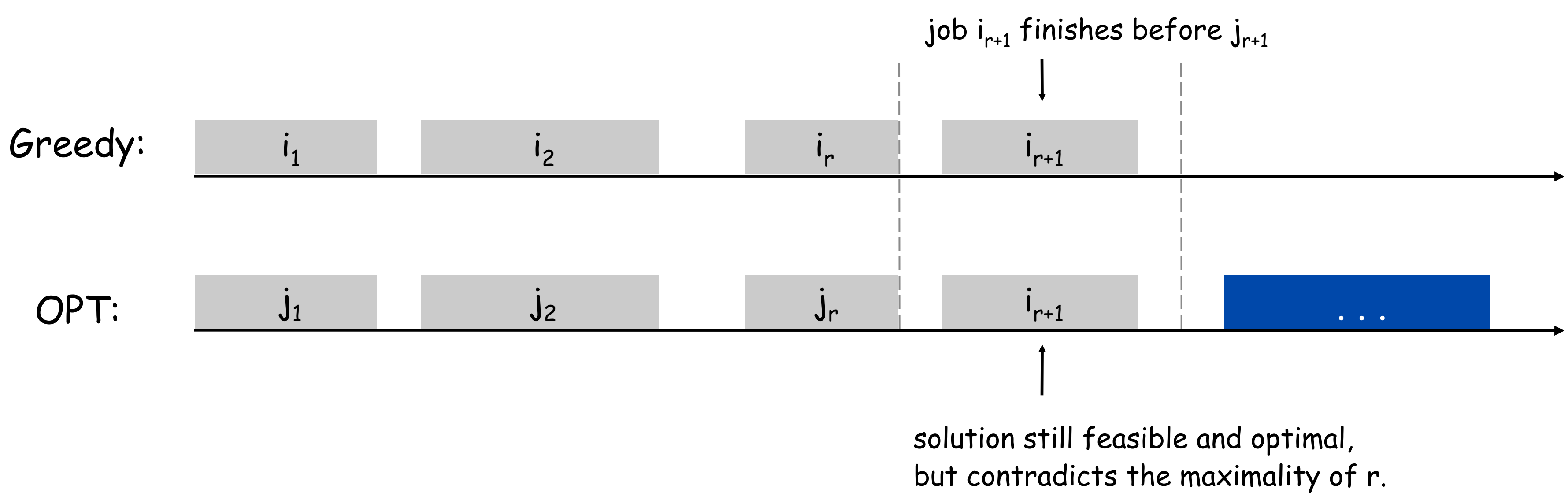


# Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

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



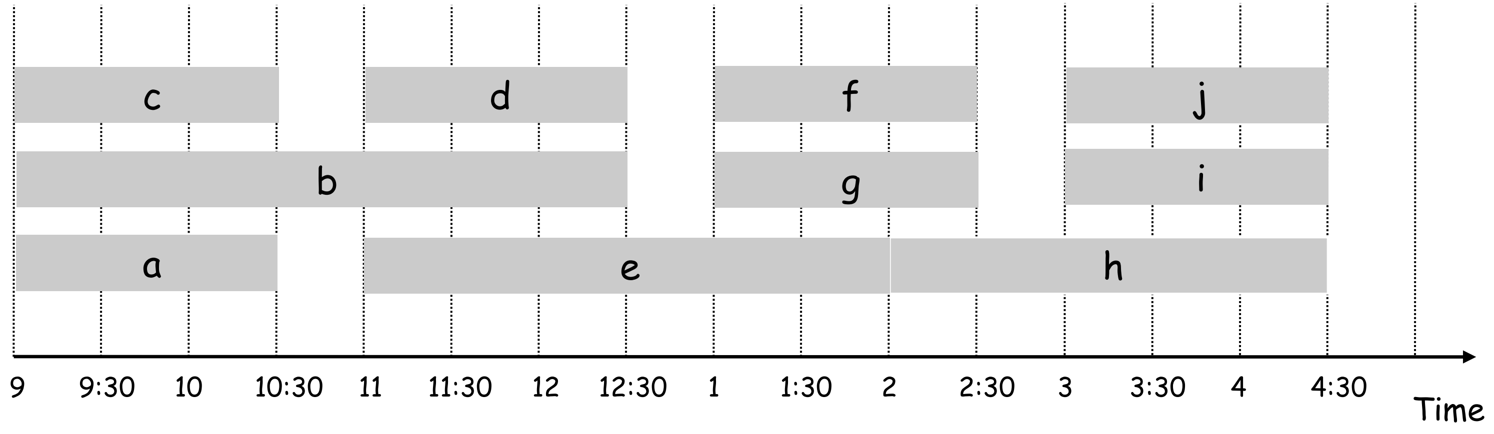
# 4.1 Interval Partitioning

# 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 only 3.



# Interval Partitioning: Lower Bound on Optimal Solution

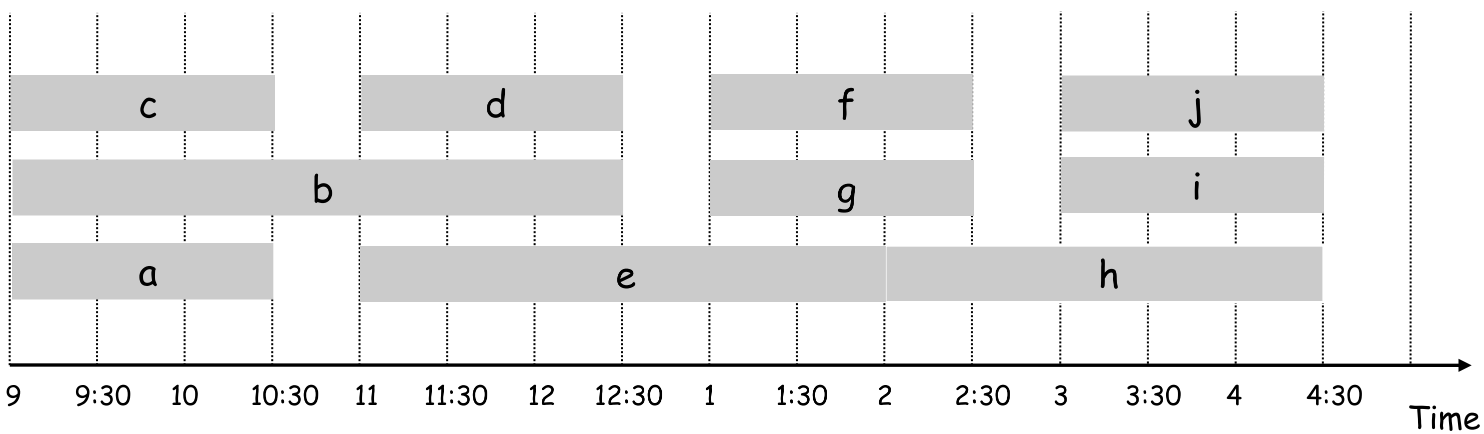
Def. The depth of a set of open intervals is the maximum number that contain any given time.

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 s1 ≤ s2 ≤ ... ≤ sn.
d ← 0    ← 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 ← 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 was allocated 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

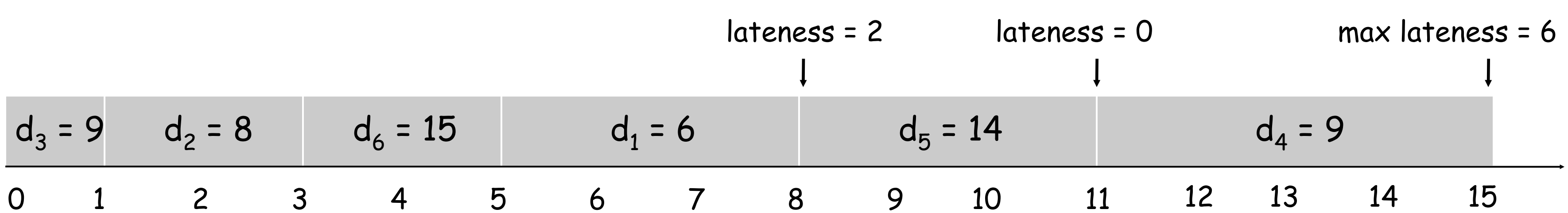
# Scheduling to Minimizing Lateness

## 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 $t_j$.

| | 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 |
|---|---|---|
| $t_j$ | 10 | 1 |
| $d_j$ | 10 | 2 |

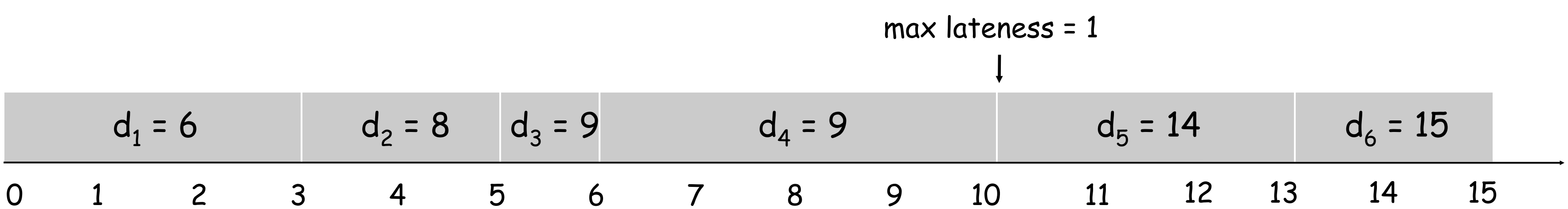
counterexample

# Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

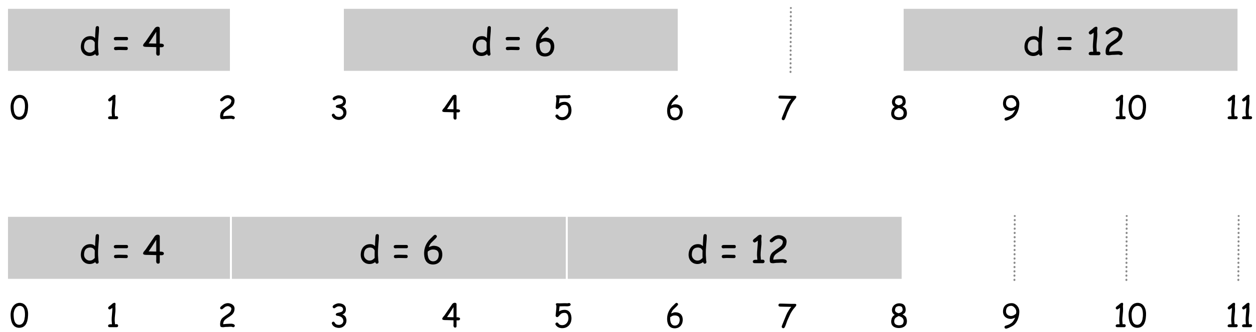
**Sort** n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

$t \leftarrow 0$
**for** $j = 1$ to $n$
  ( Assign job j to interval $[t, t + t_j]$ )
  $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
  $t \leftarrow t + t_j$
**output** intervals $[s_j, f_j]$



# Minimizing Lateness: No Idle Time

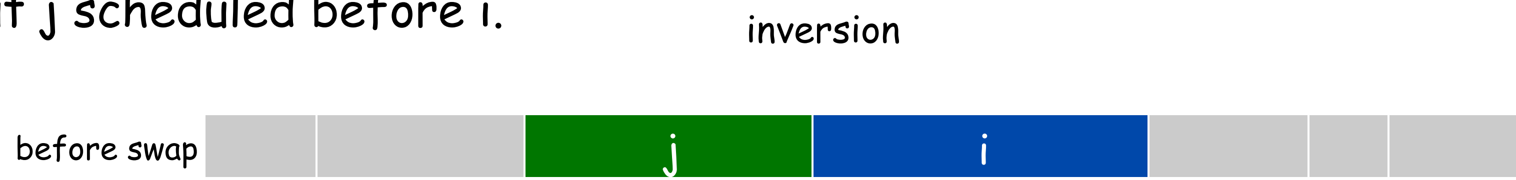
Observation. There exists an optimal schedule with no idle time.




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: $i < j$ but j scheduled before i.

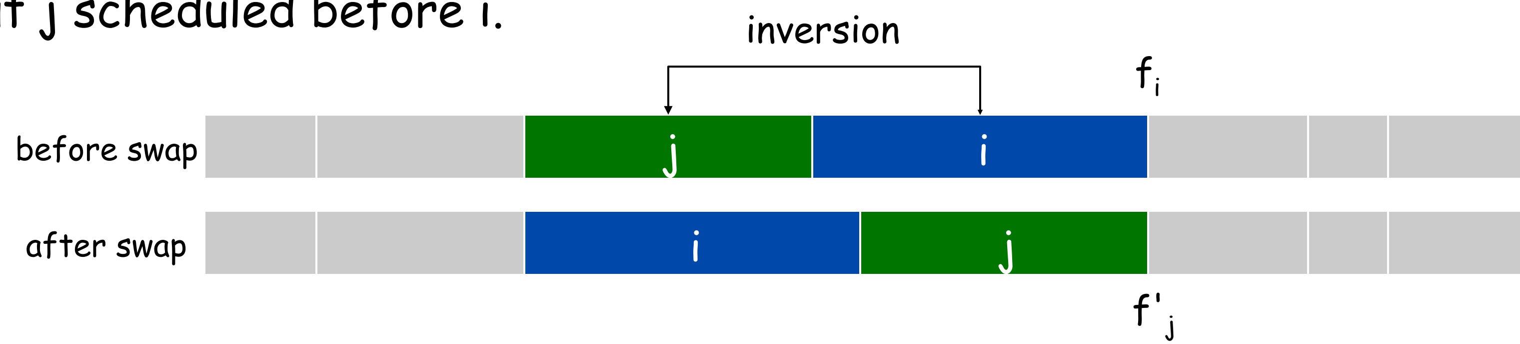



Observation. Greedy schedule has no inversions.

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:

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

# 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* ▪

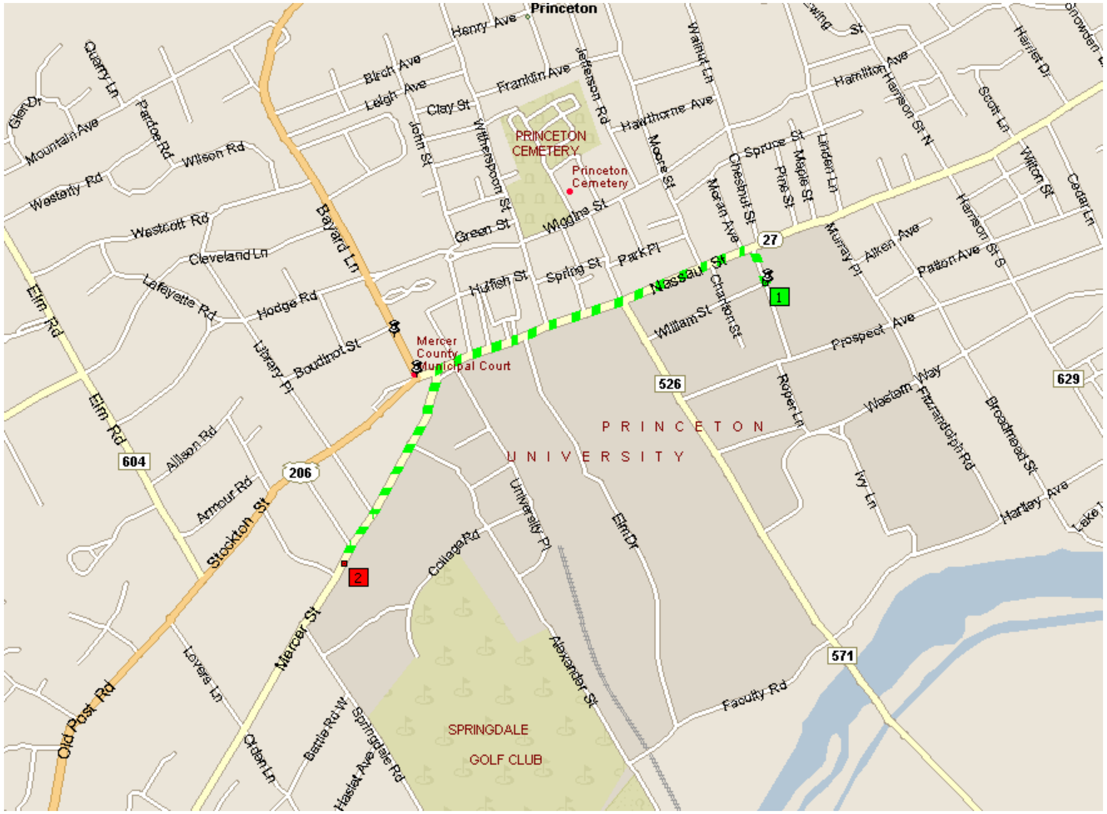
# Greedy Analysis Strategies

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.

# 4.4 Shortest Paths in a Graph




shortest path from Princeton CS department to Einstein's house

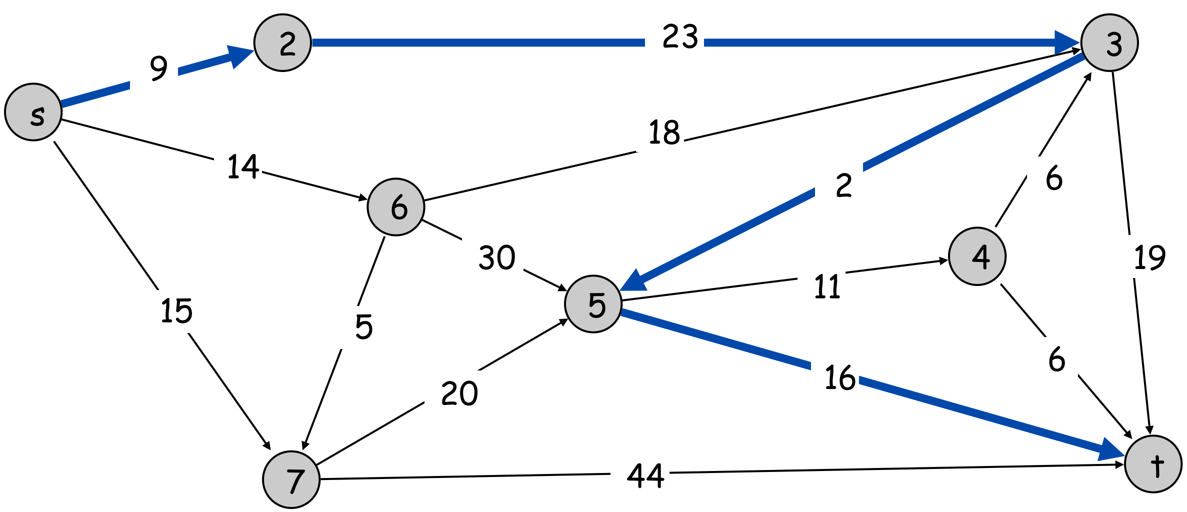
# Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source $s$, target $t$.
- 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.

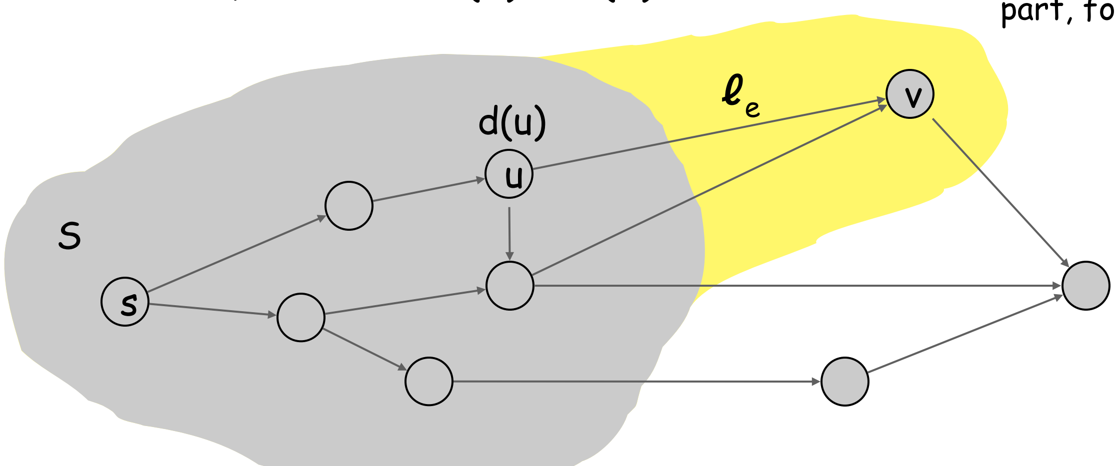
- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.

- Initialize S = { s }, d(s) = 0.
- Repeatedly (greedily) choose unexplored node $v \notin S$ which minimizes

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

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

add v to S, and set d(v) = ∂(v).

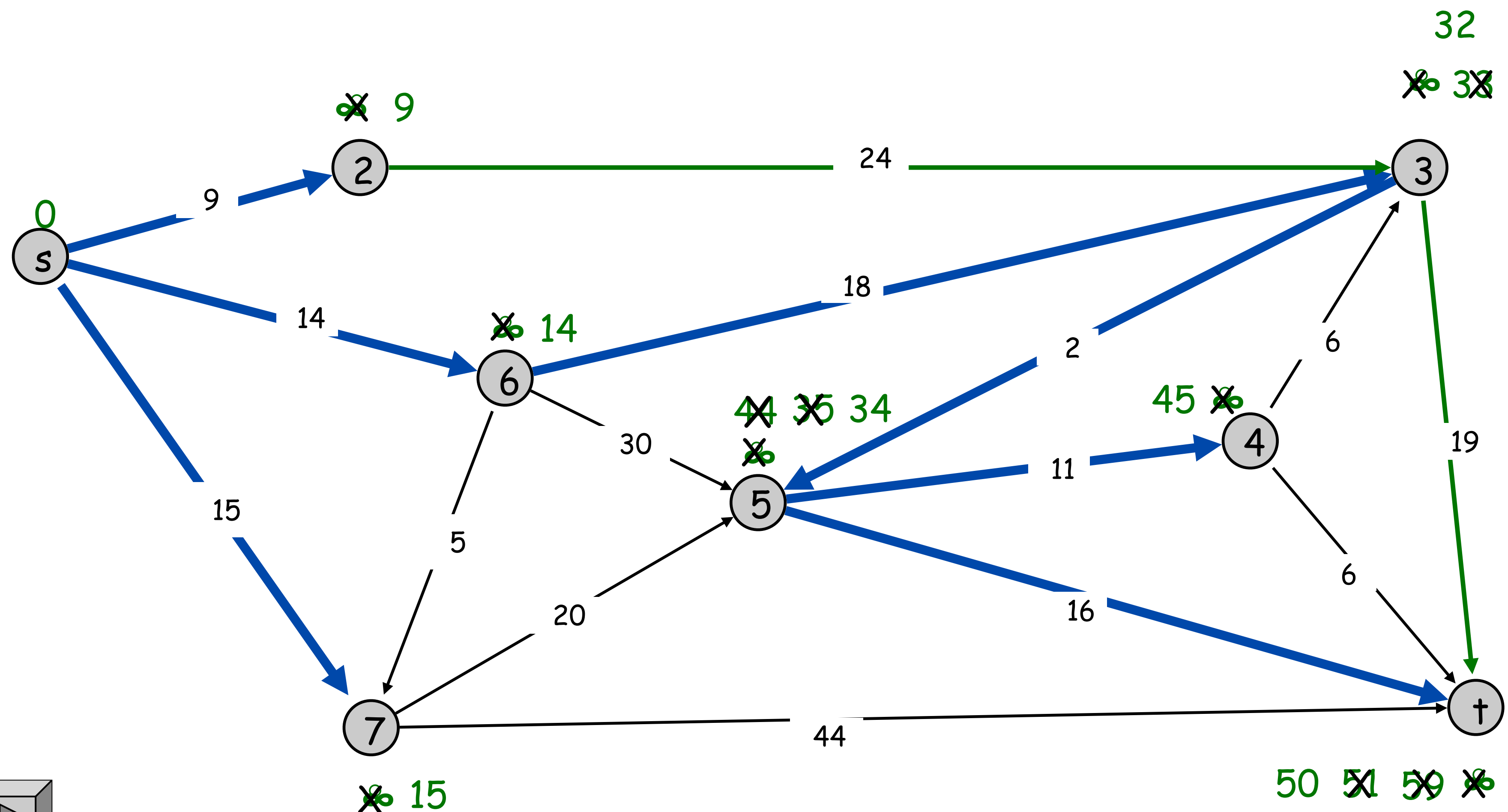


# Dijkstra's Shortest Path Algorithm

maintain keys of shortest path to each node, update if new path is shorter

$S = \{ s, 2, 3, 4, 5, 6, 7, t \}$

$PQ = \{ \}$



# Dijkstra's Algorithm: Proof of Correctness

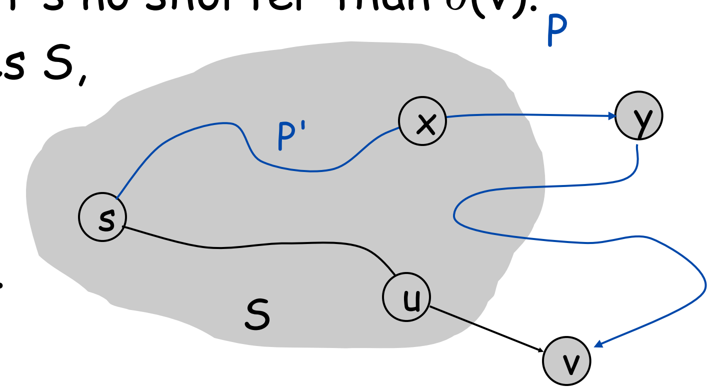
**Invariant.** For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.

**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 $\partial(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\partial(v)$.
- 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.




$$\underbrace{\ell(P) \geq \ell(P') + \ell(x,y)}_{\text{nonnegative weights}} \quad \underbrace{\geq d(x) + \ell(x,y)}_{\text{inductive hypothesis}} \quad \underbrace{\geq \partial(y)}_{\text{defn of } \partial(y)} \quad \underbrace{\geq \partial(v)}_{\text{Dijkstra chose v instead of y}}$$

# Dijkstra's Algorithm: Implementation

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

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

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

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

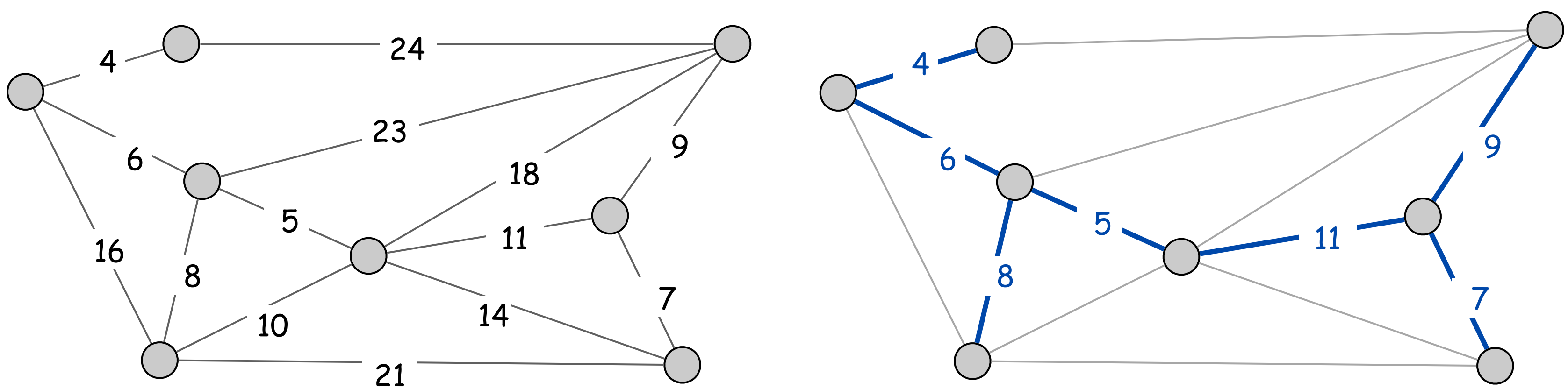
| PQ Operation | Dijkstra | Array | Binary heap | d-way Heap | Fib heap † |
|---|---|---|---|---|---|
| Insert | n | n | log n | $d \log_d n$ | 1 |
| ExtractMin | n | n | log n | $d \log_d n$ | log n |
| ChangeKey | m | 1 | log n | $\log_d n$ | 1 |
| IsEmpty | n | 1 | 1 | 1 | 1 |
| Total | | $n^2$ |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

† 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.




$G = (V, E)$

$T$, $\Sigma_{e \in T}\, c_e = 50$

**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 $T = \varnothing$. 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.

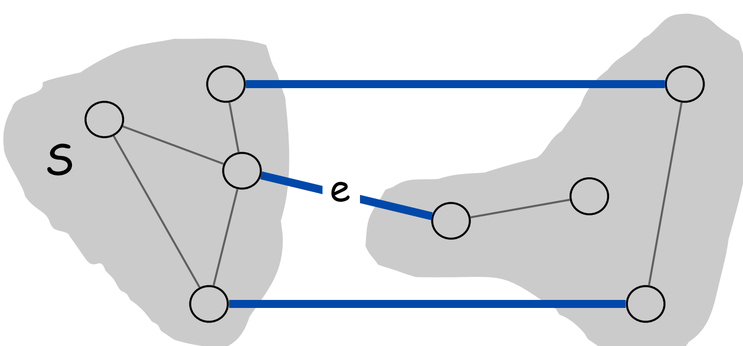
Remark. All three algorithms produce an MST.

# Greedy Algorithms

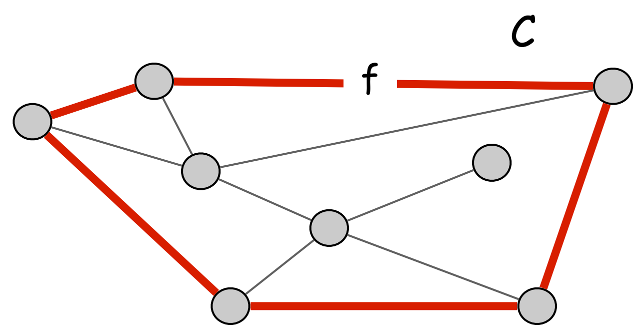
Simplifying assumption. All edge costs $c_e$ are distinct.

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.



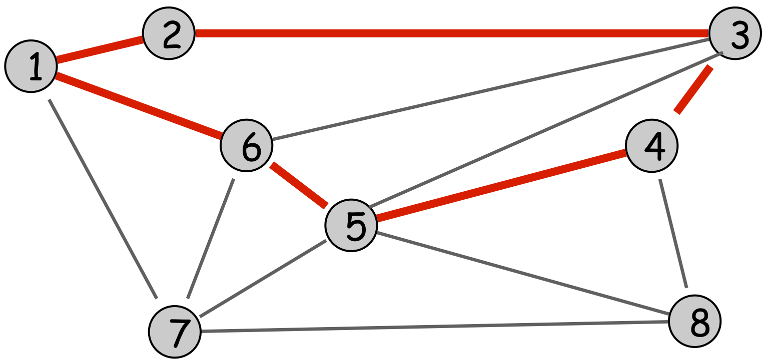

e is in the MST




f is not in the MST

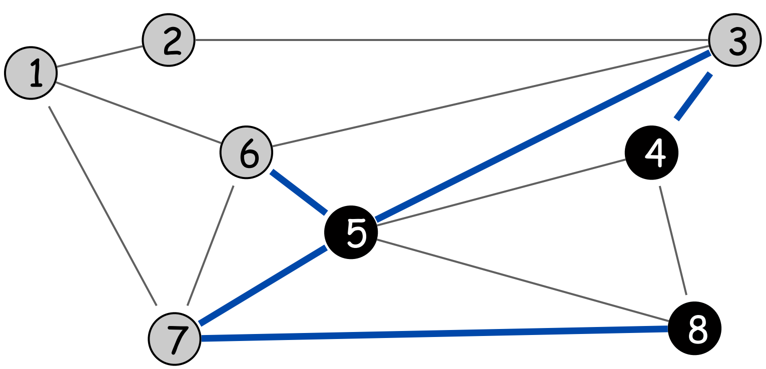
# Cycles and Cuts

Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.




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

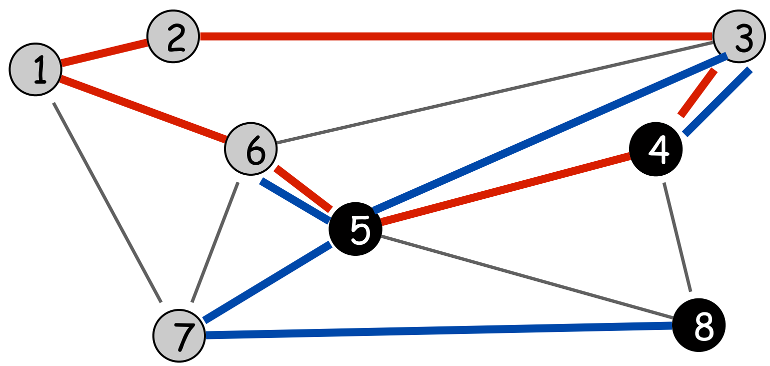
Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.




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.

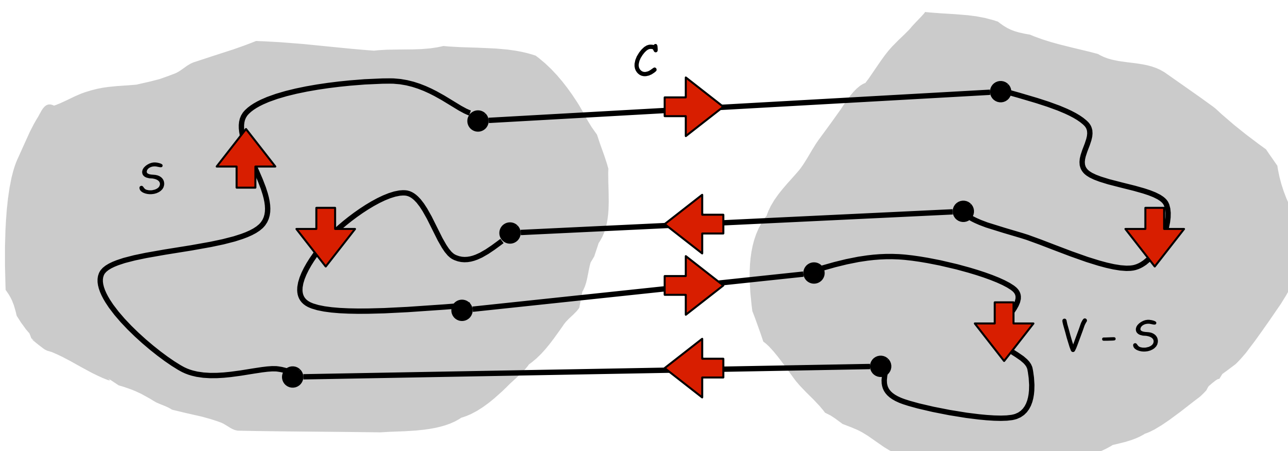



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8

Intersection = 3-4, 5-6

Pf. (by picture)

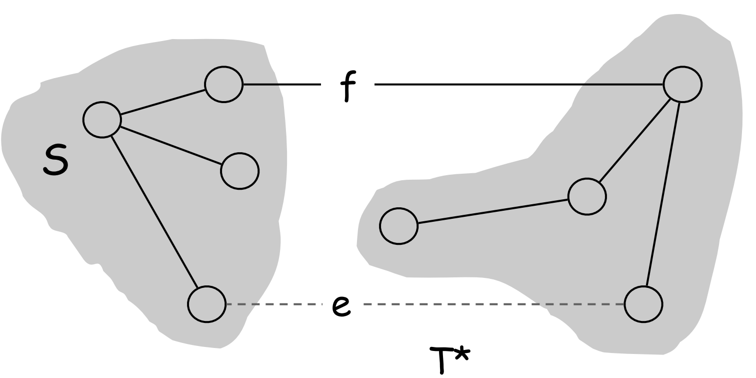


Simplifying assumption. All edge costs $c_e$ are distinct.

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 $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪



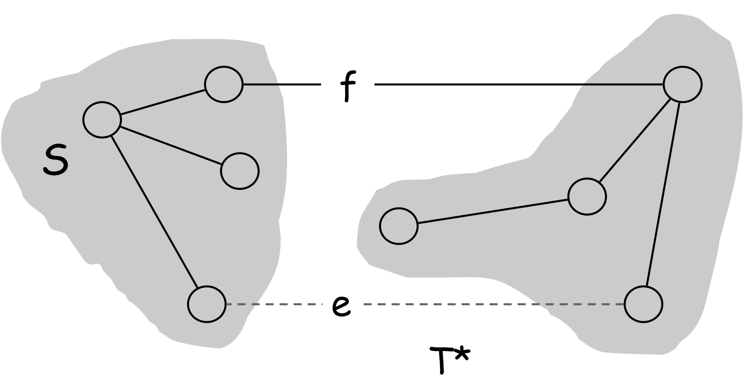
# Greedy Algorithms

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)

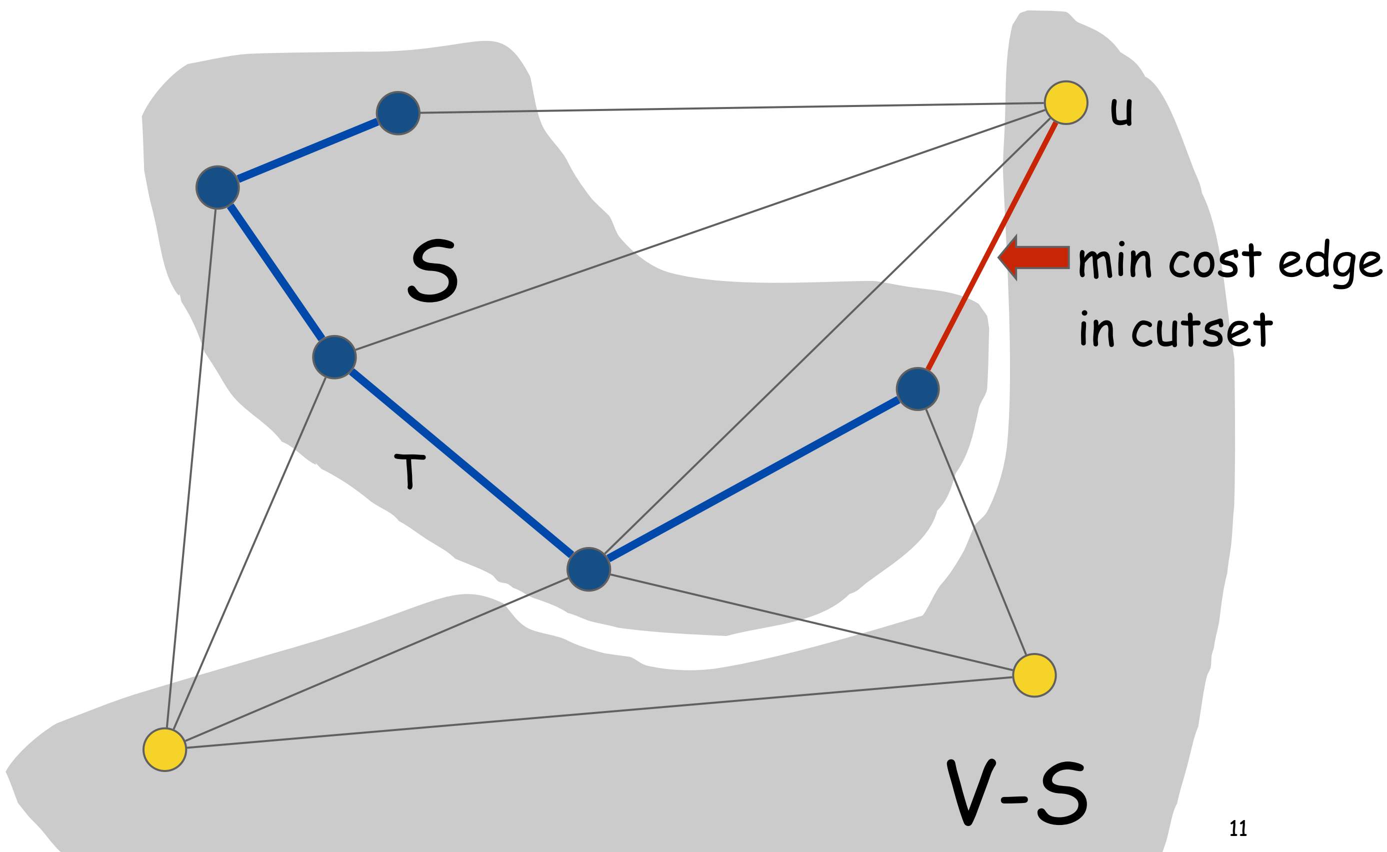
- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* creates a cut S in T*.
- Edge f is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say e, that is in both C and D.
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- 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 to T the min cost edge in cutset corresponding to S, and add one new explored node u to S.



# Implementation: Prim's Algorithm

Implementation. Use a priority queue à la 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.

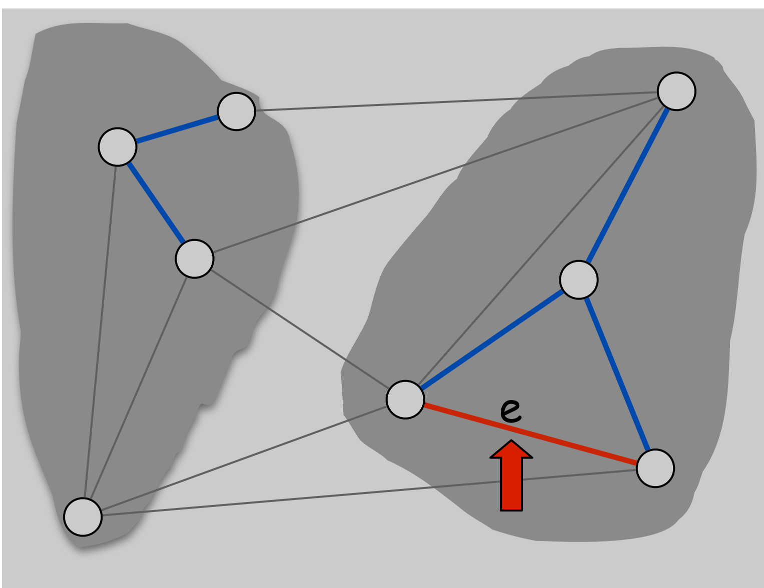
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
              {Initialize an empty priority queue Q}
   foreach (v ∈ V) insert v onto Q
           {Initialize set of explored nodes S ← ∅}

   while (Q is not empty) {
      u ← extract min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v))
          if ((v ∉ S) and (c_e < a[v]))
             decrease priority a[v] to c_e in Q
}
```

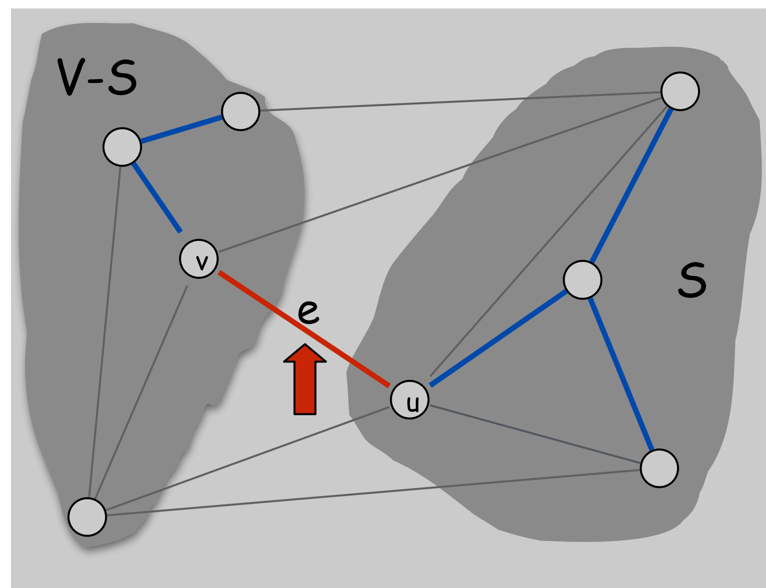
# Kruskal's Algorithm: Proof of Correctness

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.

$m \le n^2 \Rightarrow \log m \in O(\log n)$

```
Kruskal(G, c) {
   Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm.
   T ← ∅

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m
      (u,v) = ei
      if (u and v are in different sets) {
         T ← T ∪ {ei}
         merge the sets containing u and v
      }
   return T
}
```

are u and v in different connected components?

two different connected components

# 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 instead breaking ties lexicographically, according to index.

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

# 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. $O(m)$.

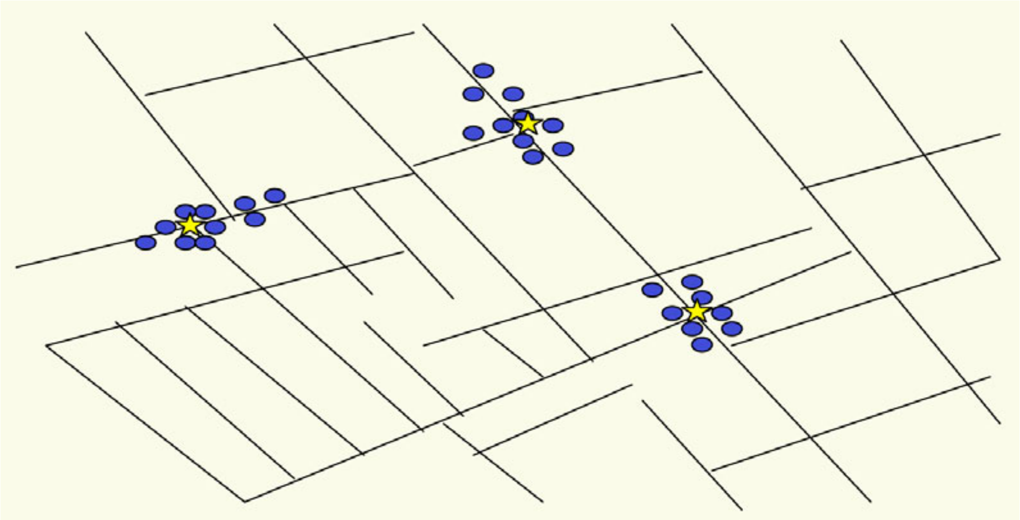
Notable.

- $O(m)$ 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

# 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 $p_1, \ldots, p_n$, 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

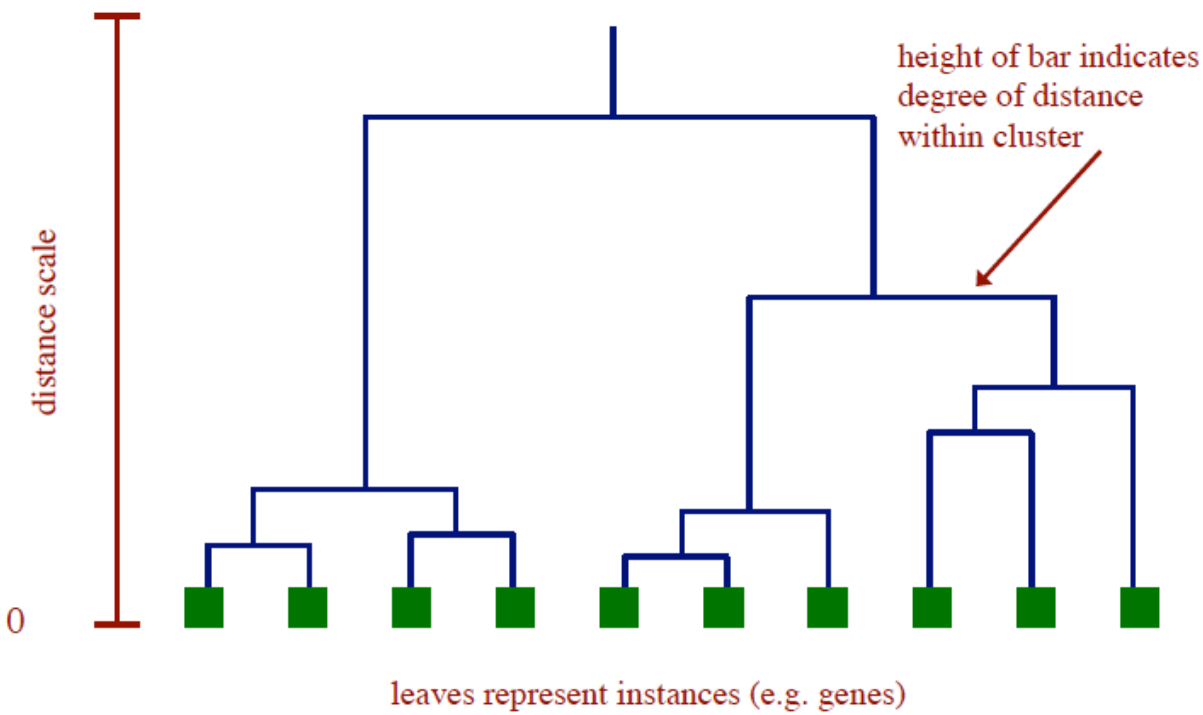
Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- 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.

# Dendrogram

Dendrogram. Scientific visualization of hypothetical sequence of evolutionary events.

- Leaves = genes.
- Internal nodes = hypothetical ancestors.




Reference: http://www.biostat.wisc.edu/bmi576/fall-2003/lecture13.pdf

# Clustering of Maximum Spacing

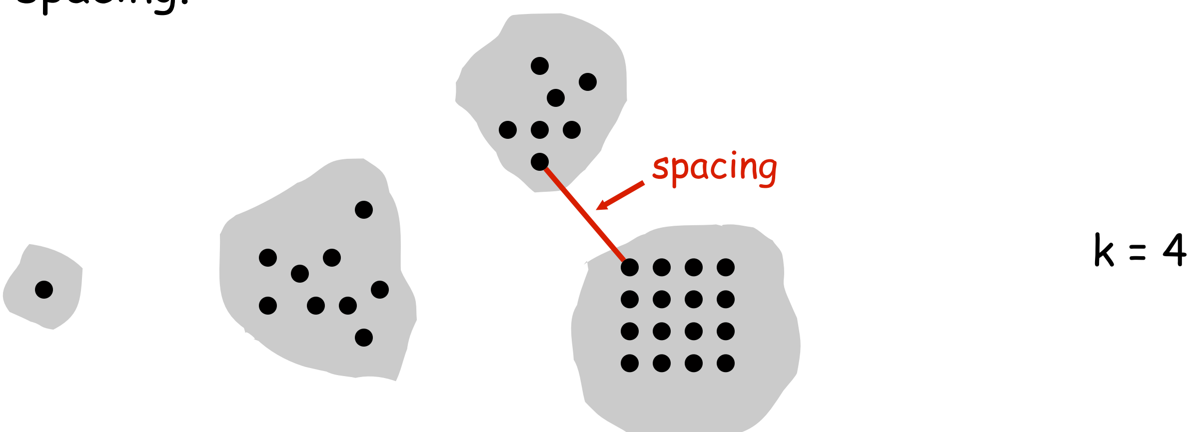
k-clustering. Divide objects into k non-empty groups.

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.



# 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 that we stop with k (instead of 1) 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 $C^*_1, \ldots, C^*_k$ 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 $C_1, \ldots, C_k$.

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